Alexandria University

Faculty of Engineering

CSED 2025

Networks Lab 2

Report

Submitted to

Eng. Mohamed Essam

Faculty of engineering Alex. University

Submitted by

Islam Yasser Mahmoud 20010312

Mkario Michel Azer 20011982

Faculty of engineering Alex. University

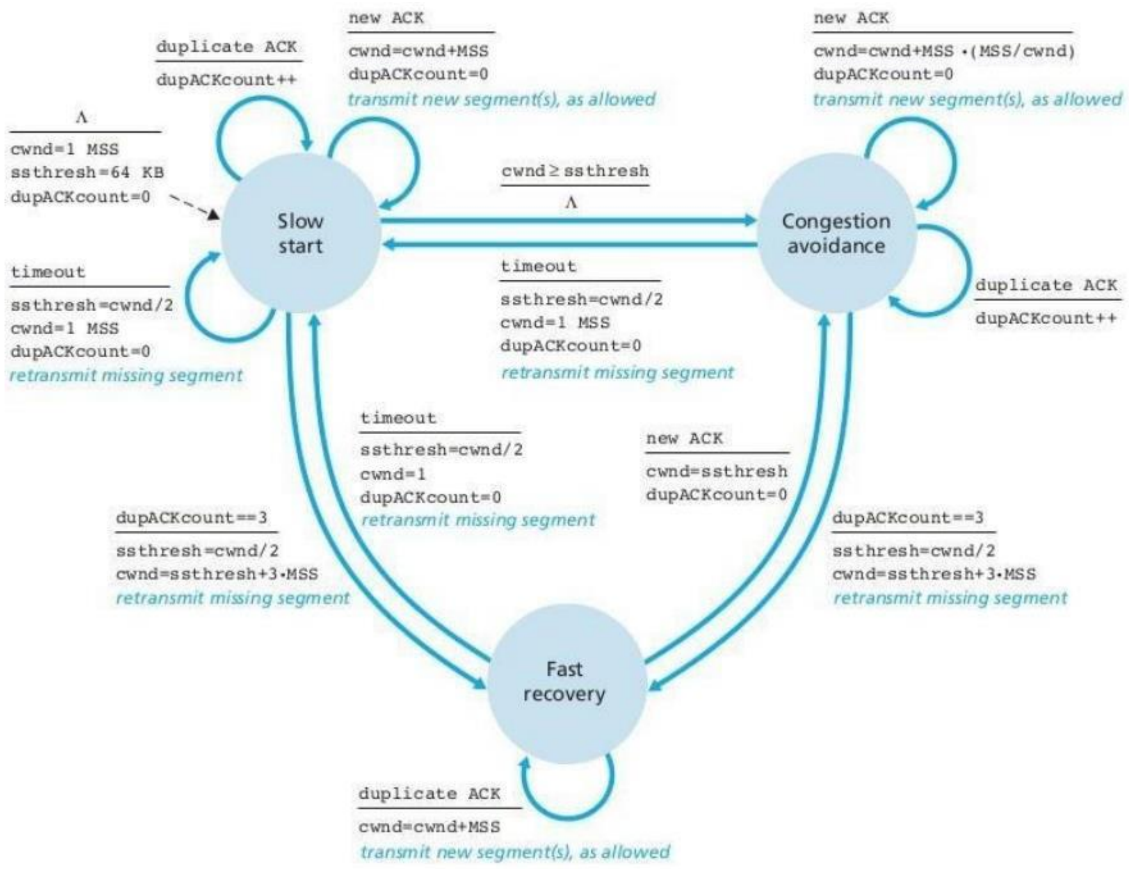
Dec 2023

Table of Contents

Lab overview:

The provided implementation demonstrates a client-server architecture for reliable data transfer using UDP (User Datagram Protocol) for communication. This architecture is designed to handle packet transmission with sequence numbers, acknowledgments (ACKs), and basic congestion control mechanisms which has three simple states:

- Slow Start:
    - **Objective:** Slow Start is designed to prevent a sudden surge of data into the network, which could lead to congestion. It aims to find the optimal sending rate for the current network conditions.
    - **Operation:** When a new connection is established or after a timeout, Slow Start begins with a conservative sending rate. The sender starts by sending a small number of segments and then doubles the congestion window size for each acknowledgment received. This exponential growth quickly increases the sending rate until congestion is detected or a predefined threshold is reached.
- Fast Recovery:
    - **Objective:** Fast Recovery is an enhancement to the traditional Fast Retransmit mechanism, which helps to recover from packet loss more quickly and efficiently.
    - **Operation:** When a sender detects a segment loss (due to triple duplicate acknowledgment), instead of dropping the congestion window to the Slow Start threshold, Fast Recovery reduces the window by half and enters a congestion avoidance phase. This helps us to quickly recover from the loss while still maintaining a reasonably high sending rate.
- Congestion avoidance:
    - **Objective:** Congestion Avoidance is aimed at maintaining an optimal sending rate without causing congestion. It helps in achieving a balance between utilizing available bandwidth and preventing network overload.
    - **Operation:** Once Slow Start has completed and the network reaches a certain congestion threshold, TCP switches to Congestion Avoidance mode. In this mode, the congestion window size increases more gradually, typically adding one segment for each round-trip time. This gradual increase helps to prevent congestion events and ensures a stable, sustainable sending rate.



new ACK
cwnd=cwnd+MSS
dupACKcount=0
transmit new segment(s), as allowed
duplicate ACK
dupACKcount++
Λ
cwnd=1 MSS
ssthresh=64 KB
dupACKcount=0
timeout
ssthresh=cwnd/2
cwnd=1 MSS
dupACKcount=0
retransmit missing segment
Slow start
cwnd ≥ ssthresh
Λ
timeout
ssthresh=cwnd/2
cwnd=1 MSS
dupACKcount=0
retransmit missing segment
Congestion avoidance
new ACK
cwnd=cwnd+MSS ·(MSS/cwnd)
dupACKcount=0
transmit new segment(s), as allowed
duplicate ACK
dupACKcount++
timeout
ssthresh=cwnd/2
cwnd=1
dupACKcount=0
retransmit missing segment
new ACK
cwnd=ssthresh
dupACKcount=0
dupACKcount==3
ssthresh=cwnd/2
cwnd=ssthresh+3·MSS
retransmit missing segment
dupACKcount==3
ssthresh=cwnd/2
cwnd=ssthresh+3·MSS
retransmit missing segment
Fast recovery
duplicate ACK
cwnd=cwnd+MSS
transmit new segment(s), as allowed

Server Program:

The server's role is to send packets of data to a client and adjust its sending rate based on network conditions. It simulates basic congestion control states similar to TCP, including slow start, congestion avoidance, and fast recovery.

Key Components:

- **Socket Initialization**: Establishes a UDP socket for communication.
- **Packet Structure**: Defines a **packet** structure containing length, sequence number, and payload.
- **Congestion Control**: Simulates slow start, congestion avoidance, and fast recovery states.
- **Packet Sending**: Sends packets based on the current congestion window (cwnd).
- **ACK Handling**: Receives ACKs from the client and adjusts the congestion window and state accordingly.
- **Timeout Handling**: Uses **select()** for timeout detection and retransmits packets if timeouts occur.

Client Program:

The client receives packets from the server, writes the payload to a file, and sends back ACKs. It also handles out-of-order packets.

Key Components:

- **Socket Setup**: Establishes a UDP socket for communication with the server.
- **Receiving Data**: Listens for incoming packets from the server.
- **Sequence Number Handling**: Ensures packets are written in order and handles out-of-order packets using a map (associative array).
- **ACK Sending**: Sends an acknowledgment back to the server for each received packet.

Major Functions and Data Structures

Server

- Functions:
  - **handleClient Function**: Handles communication with each client. It is responsible for sending data packets, receiving ACKs, handling timeouts, and managing the congestion control logic.
- Data Structures:
  - **struct packet**: Represents the data packet with sequence number and payload.
  - **struct ack_packet**: Represents the acknowledgment packet.
  - **map<uint32_t, packet>**: Stores packets that are sent but not yet acknowledged.

Client

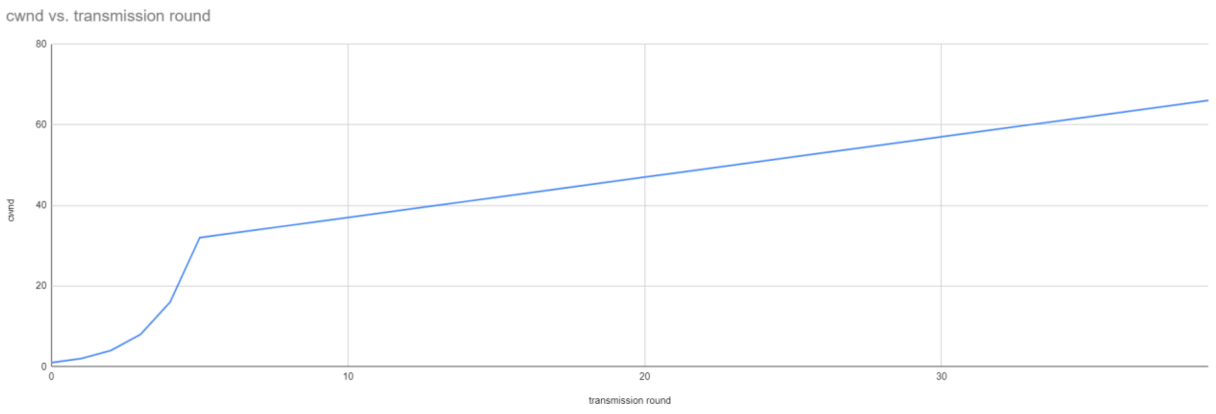
- Functions:
  - Connects to the server, receives data packets, manages packet ordering, writes data to a file, and sends ACKs.
- Data Structures:
  - **struct packet** and **struct ack_packet**: Similar to the server for representing packets and ACKs.
  - **map<uint32_t, packet>**: Buffers out-of-order packets until they can be written in sequence.

Conclusion

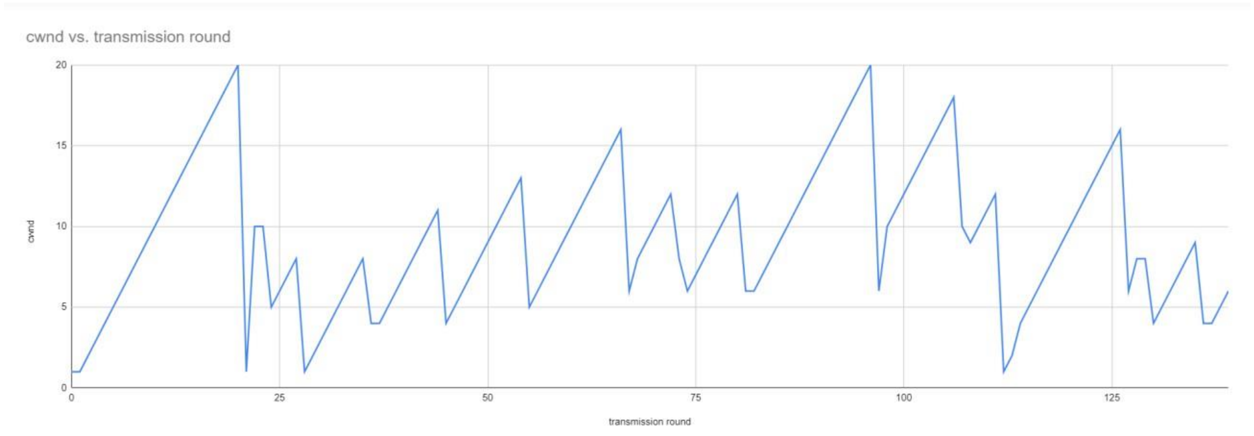
This implementation demonstrates a basic client-server communication model over UDP, incorporating elements of TCP-like sequence numbering, acknowledgments, and congestion control. While it simulates some aspects of TCP, it is important to note that UDP itself does not guarantee reliable or ordered delivery of packets. The program addresses these challenges by implementing custom logic for sequence numbering, ACKs, and packet reordering at the client side.

Analysis:

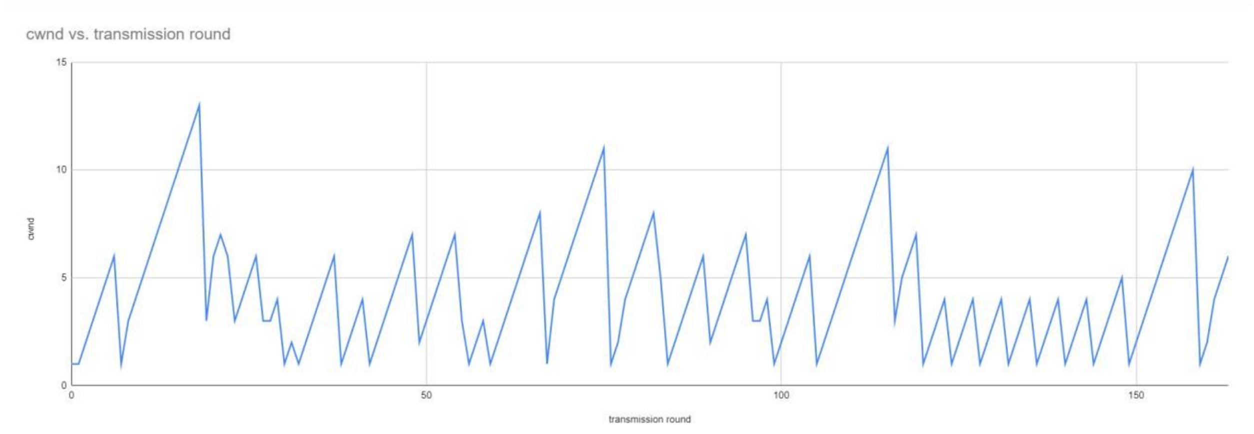
- PLP = 0
- ssthresh = 40



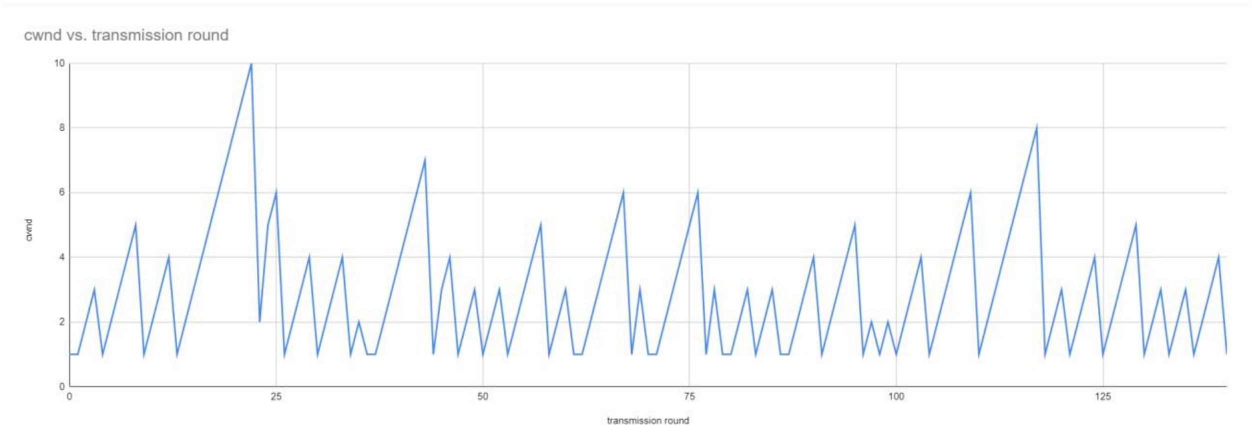
- PLP = 0.01
- ssthresh = 8



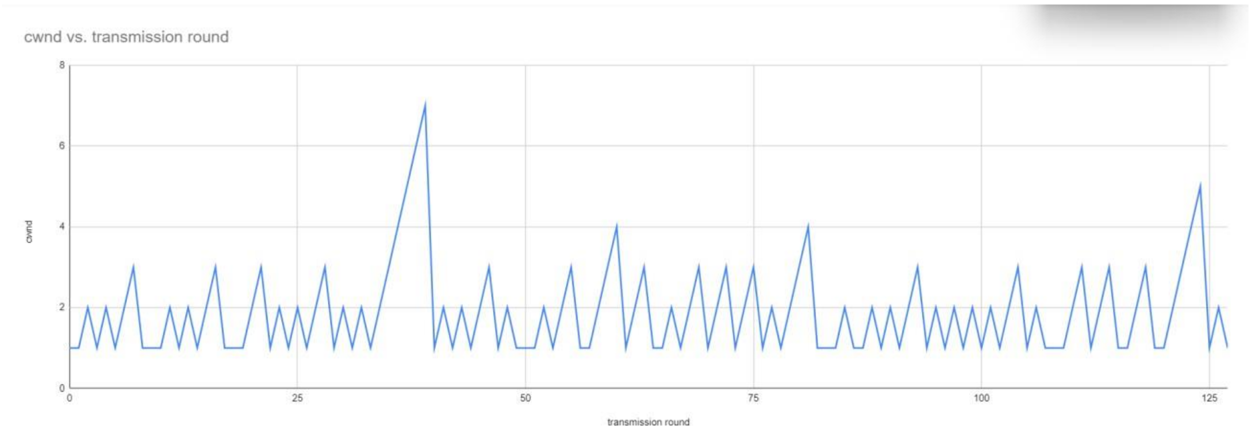
- PLP = 0.05
- ssthresh = 8



- PLP = 0.1
- ssthresh = 8

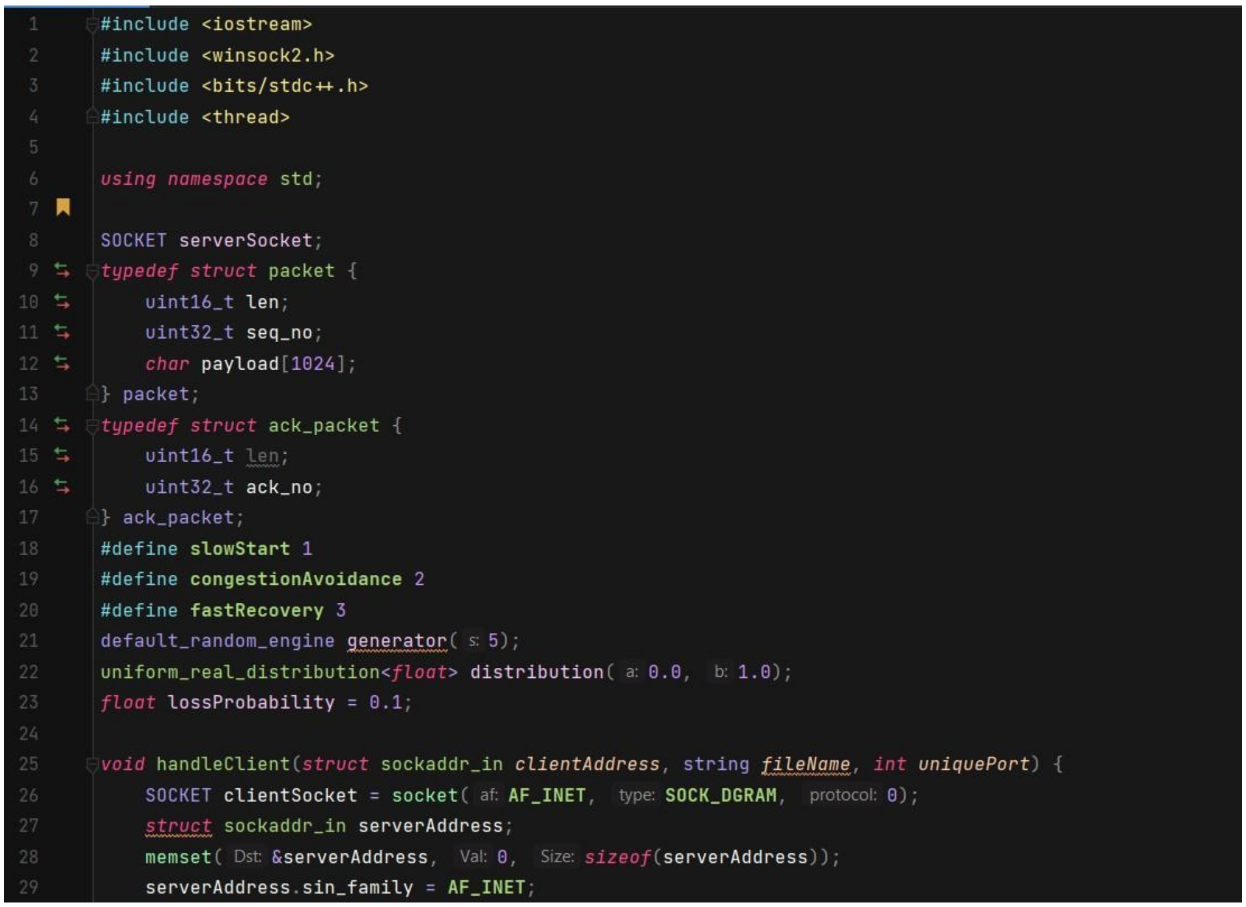


- PLP = 0.3
- ssthresh = 8

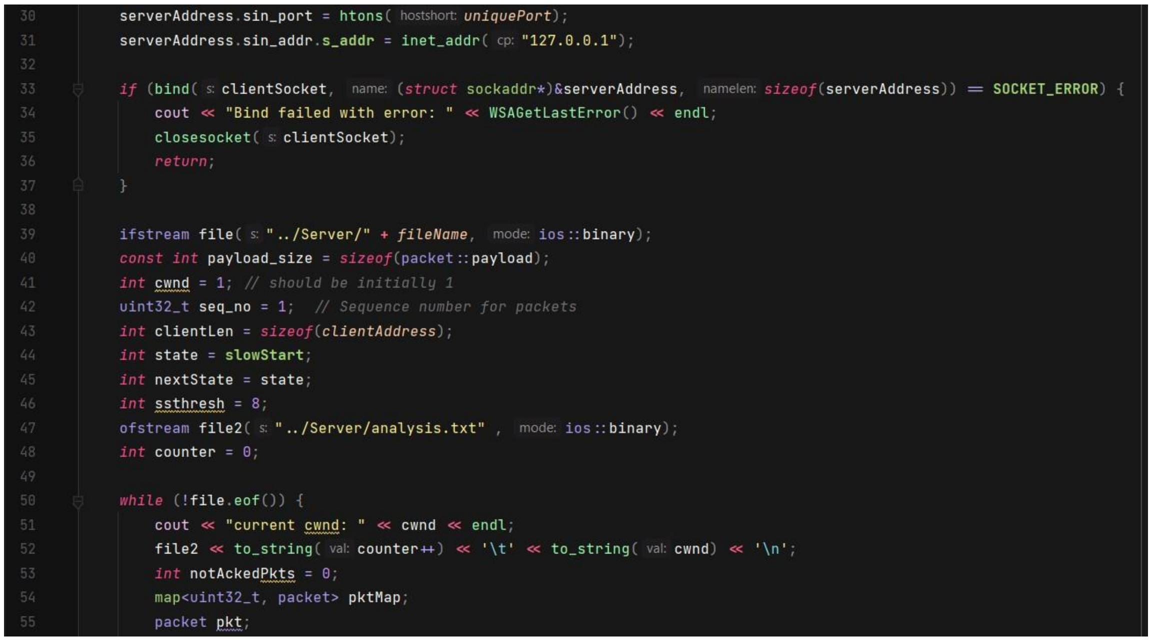


Some Code Snippets:

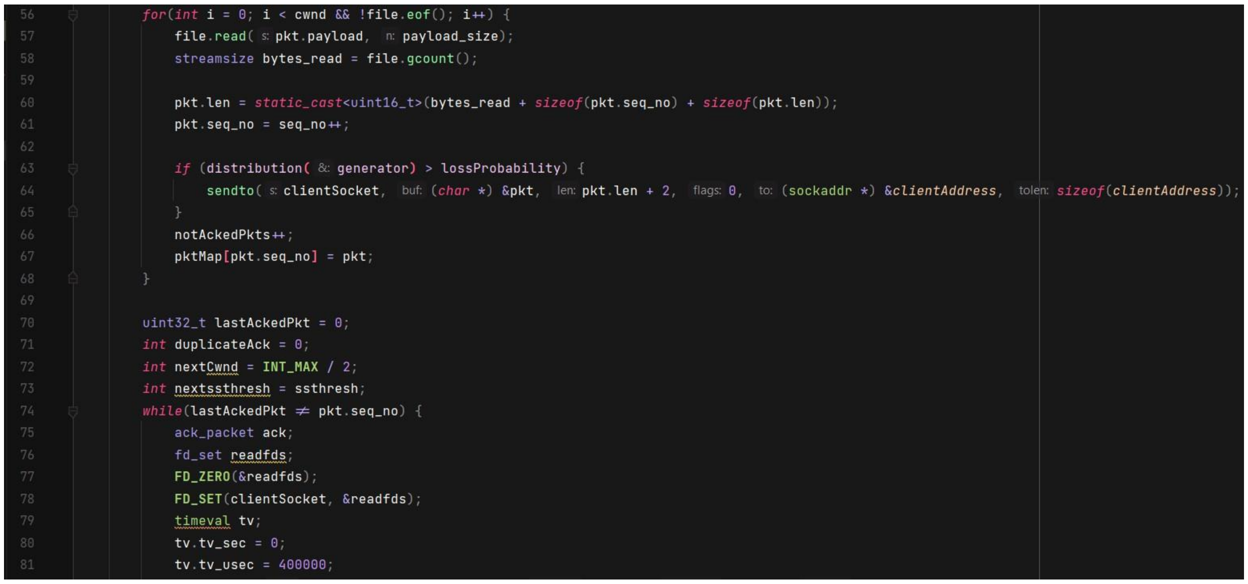
Server:



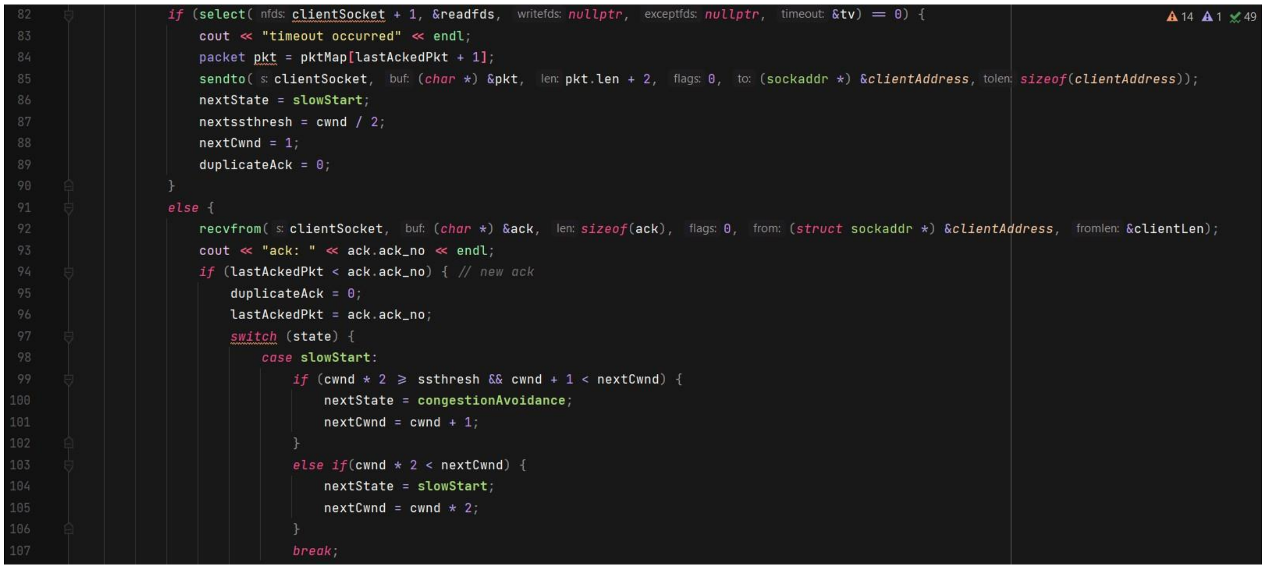
```
#include <iostream>
#include <winsock2.h>
#include <bits/stdc++.h>
#include <thread>

using namespace std;

SOCKET serverSocket;
typedef struct packet {
    uint16_t len;
    uint32_t seq_no;
    char payload[1024];
} packet;
typedef struct ack_packet {
    uint16_t len;
    uint32_t ack_no;
} ack_packet;
#define slowStart 1
#define congestionAvoidance 2
#define fastRecovery 3
default_random_engine generator( s: 5);
uniform_real_distribution<float> distribution( a: 0.0,  b: 1.0);
float lossProbability = 0.1;

void handleClient(struct sockaddr_in clientAddress, string fileName, int uniquePort) {
    SOCKET clientSocket = socket( af: AF_INET,  type: SOCK_DGRAM,  protocol: 0);
    struct sockaddr_in serverAddress;
    memset( Dst: &serverAddress,  Val: 0,  Size: sizeof(serverAddress));
    serverAddress.sin_family = AF_INET;
```



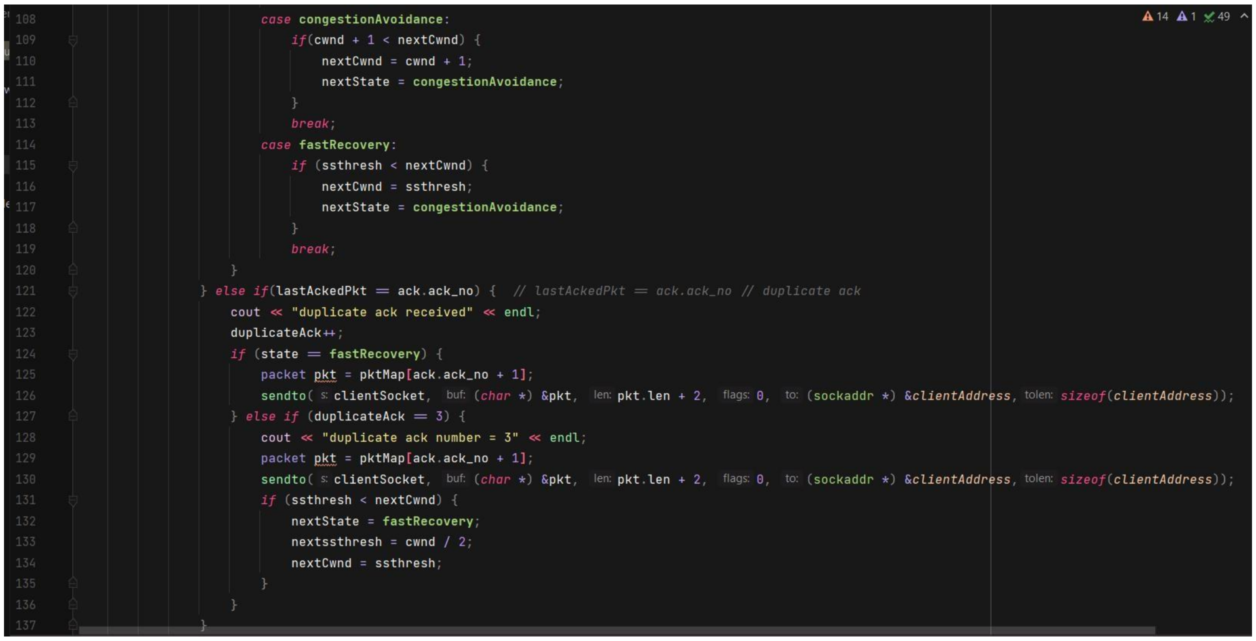
```cpp
    serverAddress.sin_port = htons( hostshort: uniquePort);
    serverAddress.sin_addr.s_addr = inet_addr( cp: "127.0.0.1");

    if (bind( s: clientSocket,  name: (struct sockaddr*)&serverAddress,  namelen: sizeof(serverAddress)) == SOCKET_ERROR) {
        cout << "Bind failed with error: " << WSAGetLastError() << endl;
        closesocket( s: clientSocket);
        return;
    }

    ifstream file( s: "../Server/" + fileName,  mode: ios::binary);
    const int payload_size = sizeof(packet::payload);
    int cwnd = 1; // should be initially 1
    uint32_t seq_no = 1;  // Sequence number for packets
    int clientLen = sizeof(clientAddress);
    int state = slowStart;
    int nextState = state;
    int ssthresh = 8;
    ofstream file2( s: "../Server/analysis.txt" ,  mode: ios::binary);
    int counter = 0;

    while (!file.eof()) {
        cout << "current cwnd: " << cwnd << endl;
        file2 << to_string( val: counter++) << '\t' << to_string( val: cwnd) << '\n';
        int notAckedPkts = 0;
        map<uint32_t, packet> pktMap;
        packet pkt;
```



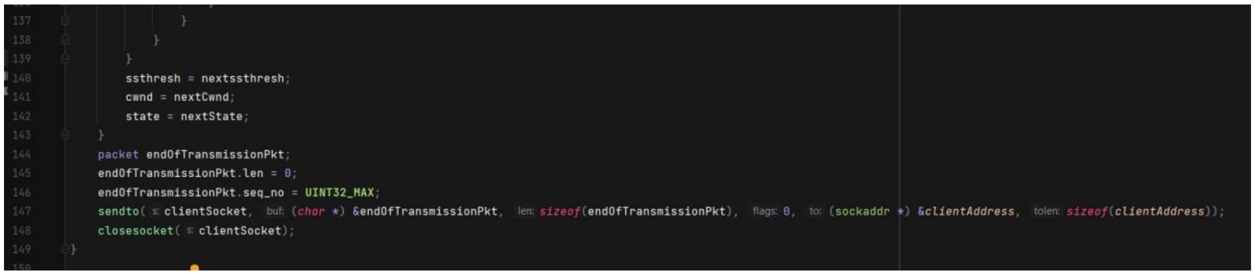
```cpp
        for(int i = 0; i < cwnd && !file.eof(); i++) {
            file.read( s: pkt.payload,  n: payload_size);
            streamsize bytes_read = file.gcount();

            pkt.len = static_cast<uint16_t>(bytes_read + sizeof(pkt.seq_no) + sizeof(pkt.len));
            pkt.seq_no = seq_no++;

            if (distribution( &: generator) > lossProbability) {
                sendto( s: clientSocket,  buf: (char *) &pkt,  len: pkt.len + 2,  flags: 0,  to: (sockaddr *) &clientAddress,  tolen: sizeof(clientAddress));
            }
            notAckedPkts++;
            pktMap[pkt.seq_no] = pkt;
        }

        uint32_t lastAckedPkt = 0;
        int duplicateAck = 0;
        int nextCwnd = INT_MAX / 2;
        int nextssthresh = ssthresh;
        while(lastAckedPkt != pkt.seq_no) {
            ack_packet ack;
            fd_set readfds;
            FD_ZERO(&readfds);
            FD_SET(clientSocket, &readfds);
            timeval tv;
            tv.tv_sec = 0;
            tv.tv_usec = 400000;
```



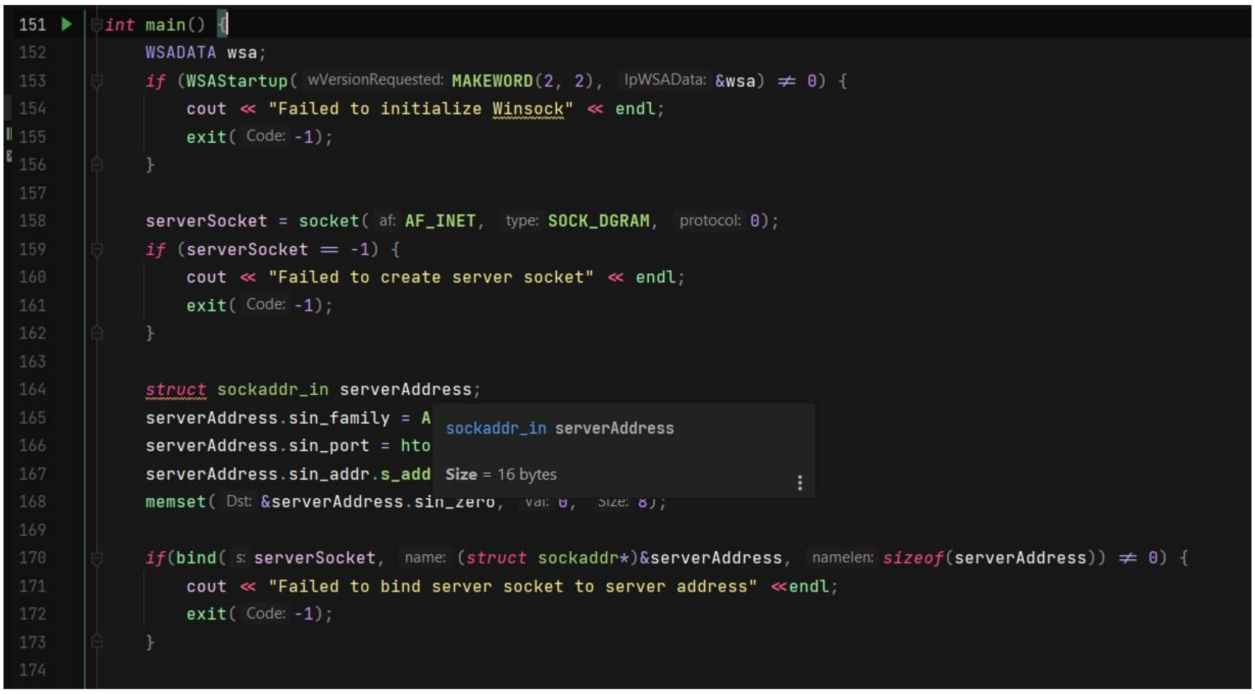
```cpp
        if (select(clientSocket + 1, &readfds, nullptr, nullptr, &tv) == 0) {
            cout << "timeout occurred" << endl;
            packet pkt = pktMap[lastAckedPkt + 1];
            sendto(clientSocket, (char *) &pkt, pkt.len + 2, 0, (sockaddr *) &clientAddress, sizeof(clientAddress));
            nextState = slowStart;
            nextssthresh = cwnd / 2;
            nextCwnd = 1;
            duplicateAck = 0;
        }
        else {
            recvfrom(clientSocket, (char *) &ack, sizeof(ack), 0, (struct sockaddr *) &clientAddress, &clientLen);
            cout << "ack: " << ack.ack_no << endl;
            if (lastAckedPkt < ack.ack_no) { // new ack
                duplicateAck = 0;
                lastAckedPkt = ack.ack_no;
                switch (state) {
                    case slowStart:
                        if (cwnd * 2 >= ssthresh && cwnd + 1 < nextCwnd) {
                            nextState = congestionAvoidance;
                            nextCwnd = cwnd + 1;
                        }
                        else if(cwnd * 2 < nextCwnd) {
                            nextState = slowStart;
                            nextCwnd = cwnd * 2;
                        }
                        break;
```



```cpp
                    case congestionAvoidance:
                        if(cwnd + 1 < nextCwnd) {
                            nextCwnd = cwnd + 1;
                            nextState = congestionAvoidance;
                        }
                        break;
                    case fastRecovery:
                        if (ssthresh < nextCwnd) {
                            nextCwnd = ssthresh;
                            nextState = congestionAvoidance;
                        }
                        break;
                }
            } else if(lastAckedPkt == ack.ack_no) {  // lastAckedPkt == ack.ack_no // duplicate ack
                cout << "duplicate ack received" << endl;
                duplicateAck++;
                if (state == fastRecovery) {
                    packet pkt = pktMap[ack.ack_no + 1];
                    sendto(clientSocket, (char *) &pkt, pkt.len + 2, 0, (sockaddr *) &clientAddress, sizeof(clientAddress));
                } else if (duplicateAck == 3) {
                    cout << "duplicate ack number = 3" << endl;
                    packet pkt = pktMap[ack.ack_no + 1];
                    sendto(clientSocket, (char *) &pkt, pkt.len + 2, 0, (sockaddr *) &clientAddress, sizeof(clientAddress));
                    if (ssthresh < nextCwnd) {
                        nextState = fastRecovery;
                        nextssthresh = cwnd / 2;
                        nextCwnd = ssthresh;
                    }
                }
            }
```



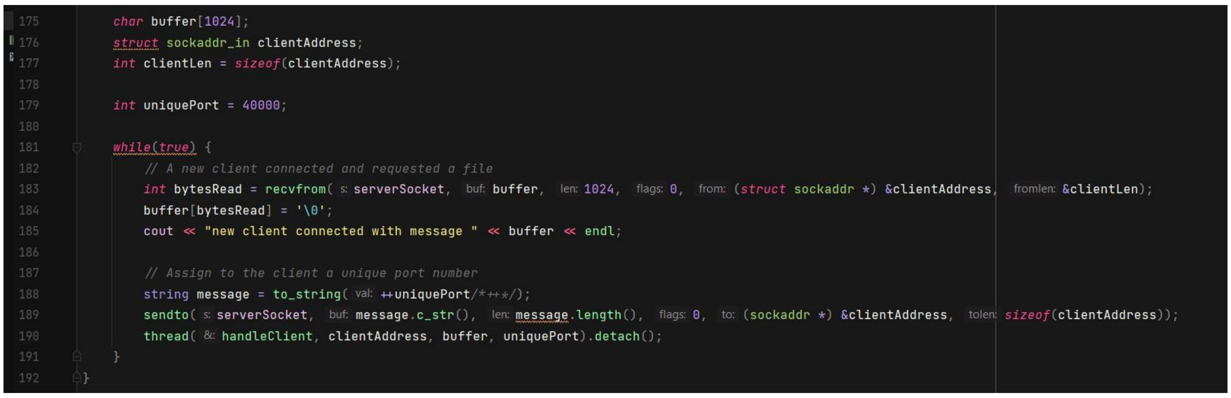
```cpp
                }
            }
        }
        ssthresh = nextssthresh;
        cwnd = nextCwnd;
        state = nextState;
    }
    packet endOfTransmissionPkt;
    endOfTransmissionPkt.len = 0;
    endOfTransmissionPkt.seq_no = UINT32_MAX;
    sendto(clientSocket, (char *) &endOfTransmissionPkt, sizeof(endOfTransmissionPkt), 0, (sockaddr *) &clientAddress, sizeof(clientAddress));
    closesocket(clientSocket);
}
```



```cpp
int main() {
    WSADATA wsa;
    if (WSAStartup( wVersionRequested: MAKEWORD(2, 2),  lpWSAData: &wsa) != 0) {
        cout << "Failed to initialize Winsock" << endl;
        exit( Code: -1);
    }

    serverSocket = socket( af: AF_INET,  type: SOCK_DGRAM,  protocol: 0);
    if (serverSocket == -1) {
        cout << "Failed to create server socket" << endl;
        exit( Code: -1);
    }

    struct sockaddr_in serverAddress;
    serverAddress.sin_family = A
    serverAddress.sin_port = hto
    serverAddress.sin_addr.s_add
    memset( Dst: &serverAddress.sin_zero,  Val: 0,  Size: 8);

    if(bind( s: serverSocket,  name: (struct sockaddr*)&serverAddress,  namelen: sizeof(serverAddress)) != 0) {
        cout << "Failed to bind server socket to server address" <<endl;
        exit( Code: -1);
    }

```

sockaddr_in serverAddress

Size = 16 bytes



```cpp
    char buffer[1024];
    struct sockaddr_in clientAddress;
    int clientLen = sizeof(clientAddress);

    int uniquePort = 40000;

    while(true) {
        // A new client connected and requested a file
        int bytesRead = recvfrom( s: serverSocket,  buf: buffer,  len: 1024,  flags: 0,  from: (struct sockaddr *) &clientAddress,  fromlen: &clientLen);
        buffer[bytesRead] = '\0';
        cout << "new client connected with message " << buffer << endl;

        // Assign to the client a unique port number
        string message = to_string( val: ++uniquePort/*++*/);
        sendto( s: serverSocket,  buf: message.c_str(),  len: message.length(),  flags: 0,  to: (sockaddr *) &clientAddress,  tolen: sizeof(clientAddress));
        thread( &: handleClient, clientAddress, buffer, uniquePort).detach();
    }
}
```