

Computer Networks and Communication (CSE 431). Laboratory [3] 2022

JANUARY 8



Name: Ahmed Akram Ahmed Shawky

ID: 18010056

Name: Amr Ahmed Fouad Mohammed Mostafe El-Zahabi

ID: 18011174

Name: Momen Ibrahim Fawzy Hassan

ID: 18011896

Name: Rana Ayman Hussein

ID: 18010662

Implementing a Reliable Data Transport Protocol

Specifications:

We've a file and we want to send this file from one side to the other (server to client) so, we split the file into chunks of data of fixed length and add the data of one chunk to a UDP datagram packet in the data field of the packet.
implementing TCP with congestion control.

Packet type and fields:

There are two kinds of packets:

Data packets:

```
struct packet {
    uint16_t cksum;
    uint16_t len;
    uint32_t seqno;
    char data [500];
};
```

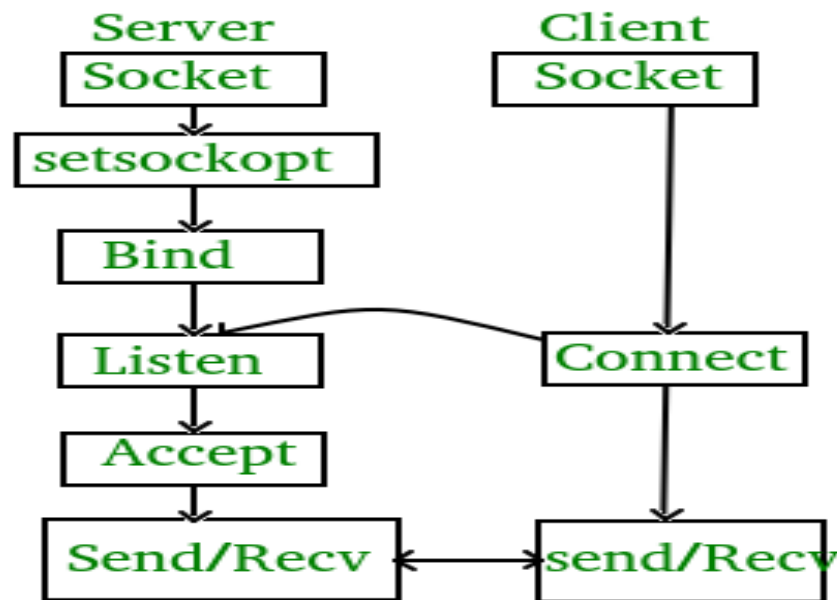
Ack-only packets:

```
struct ack_packet {
    uint16_t cksum;
    uint16_t len;
    uint32_t ackno;
};
```

```
1      /* Data-only packets */
2      struct packet {
3          /* Header */
4          uint16_t cksum; /* Optional bonus part */
5          uint16_t len;
6          uint32_t seqno;
7          /* Data */
8          char data[500]; /* Not always 500 bytes, can be less */
9      };
```

```
1      /* Ack-only packets are only 8 bytes */
2      struct ack_packet {
3          uint16_t cksum; /* Optional bonus part */
4          uint16_t len;
5          uint32_t ackno;
6      };
```

In the connection between the server and the clients we made like what happen in the coming picture: (from Geeks for Geeks)



Server:

Here is the code until the binding:

```
vector<string> the_args = readCommand();
port = stoi(the_args[0]);
RandomSeedGen = stoi(the_args[1]);
PLP = stod(the_args[2]);
//PLP = stod("0");
srand(RandomSeedGen);

int server_socket, client_socket;
int portNom = 8000;

struct sockaddr_in server_address{};
struct sockaddr_in client_address{};
int server_addrlen = sizeof(server_address);

if ((server_socket = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)
{
    perror("error creating server socket ! ");
    exit(1);
}

memset(&server_address, 0, sizeof(server_address));
memset(&client_address, 0, sizeof(client_address));
server_address.sin_family = AF_INET;
server_address.sin_port = htons(portNom);
server_address.sin_addr.s_addr = INADDR_ANY;
memset(&(server_address.sin_zero), '\0', ACK_PACKET_SIZE);

if (bind(server_socket, (struct sockaddr *) &server_address, sizeof(server_address)) < 0)
{
    perror("error in binding server ! ");
    exit(1);
}
```

Then the server will listen for the clients to connect:

```
while (true){
    socklen_t client_addrlen = sizeof(struct sockaddr);
    cout << "Waiting For A New Connection ... " << endl << flush;
    char rec_buffer[MSS];
    ssize_t Received_bytes = recvfrom(server_socket, rec_buffer, MSS, 0, (struct sockaddr*)&client_address, &client_addrlen);
    if (Received_bytes <= 0){
        perror("error in receiving bytes of the file name !");
        exit(1);
    }
    /** forking to handle request **/
    pid_t pid = fork();
    if (pid == -1){
        perror("error in forking a child process for the client !");
    } else if (pid == 0){
        if ((client_socket = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)
        {
            perror("error creating a socket for the client !");
            exit(1);
        }
        handle_client_request(server_socket, client_socket, client_address, rec_buffer, MSS);
        exit(0);
    }
}
close(server_socket);
```

and for multiple users the server forks off a child process to handle the client.

and That server (child) creates a UDP socket to handle file transfer to the client.

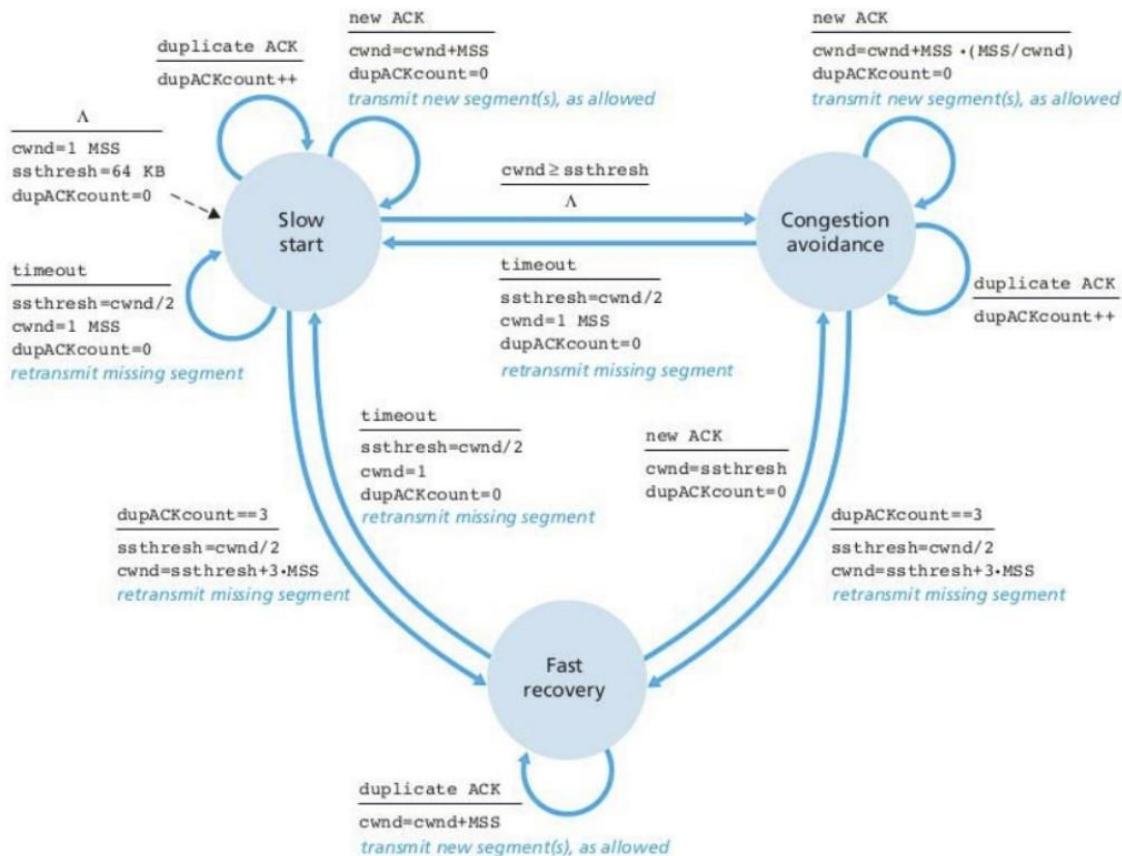
as shown in the code above.

Then we handle the client requests as follow:

- the client sends the name of the file that he wants the server to send to him.
- when the server recive the file name he checks if that file is available if not available show message "Error Open the requested file" and don't send any thing to the client.

If the file is available the server sends the number of packet that need to be sent to the client.

Then transmitting the packets will start following that chart:



For simulating the corrupted packets we subtract one from the check sum of the packet so that the client knows that the packet is corrupted.

EX:

```
is Lost val : 9.5 corrupting packet as lost val(calculated using probability) > 5.9
////////////////////Corrupt data
is Lost val : 0.8 not drop packet as lost val < 5.9
////////////////////Drop data
Error sending data packet ! : Success
Timed Out !
Re-transmitting the packet
```


For simulating lost packets the server doesn't send them so, no ack for them from the client will be received.

EX:

```
is Lost val : 1.8 not corubting packet as lost val < 5.9
is Lost val : 7.7 dropping packet as lost val > 5.9
////////////////////Drop data
Error sending data packet ! : Success
Timed Out !
Re-transmitting the packet
```

For the congestion window:

- The CWND increasing exponentially in the stage of slow start.
- When CWND value is more that the vale of ssthresh which we set initially 128, it starts to increas linearly in the stage of congestion avoidance.
- When there is packet lost the CWND is drop to equals 1 and start increasing again.
- When there is triple duplicate Ack the CWND size drop to the half of its size and the ssthresh will equal the half of CWND just before lossing the packet.

Network system analysis:

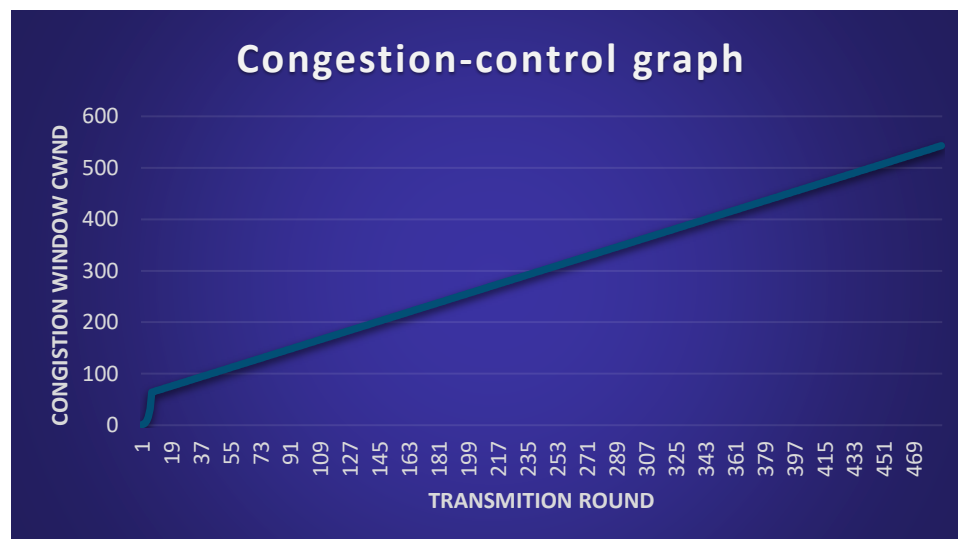
comparison between stop-and-wait and selective repeat strategies:

BASIS FOR COMPARISON	STOP-AND-WAIT (GO BACK N)	SELECTIVE REPEAT
Basic	Retransmits all the frames that sent after the frame which suspects to be damaged or lost.	Retransmits only those frames that are suspected to lost or damaged.

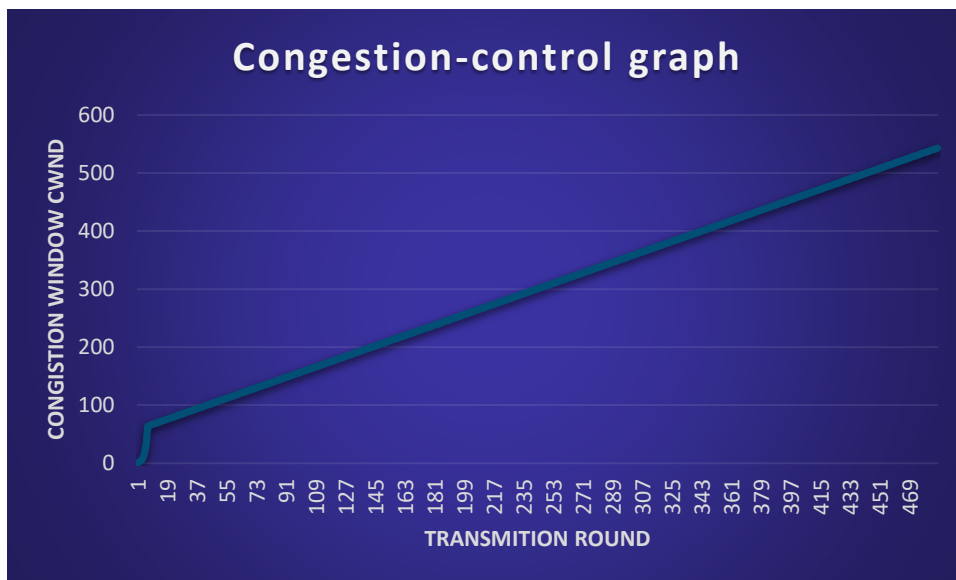
Bandwidth Utilization	If error rate is high, it wastes a lot of bandwidth.	Comparatively less bandwidth is wasted in retransmitting.
Complexity	Less complicated.	More complex as it requires to apply extra logic and sorting and storage, at sender and receiver.
Window size	$N-1$	$\leq (N+1)/2$
Sorting	Sorting is neither required at sender side nor at receiver side.	Receiver must be able to sort as it must maintain the sequence of the frames.
Storing	Receiver do not store the frames received after the damaged frame until the damaged frame is retransmitted.	Receiver stores the frames received after the damaged frame in the buffer until the damaged frame is replaced.
Searching	No searching of frame is required neither on sender side nor on receiver	The sender must be able to search and select only the requested frame.

congestion window Analysis:

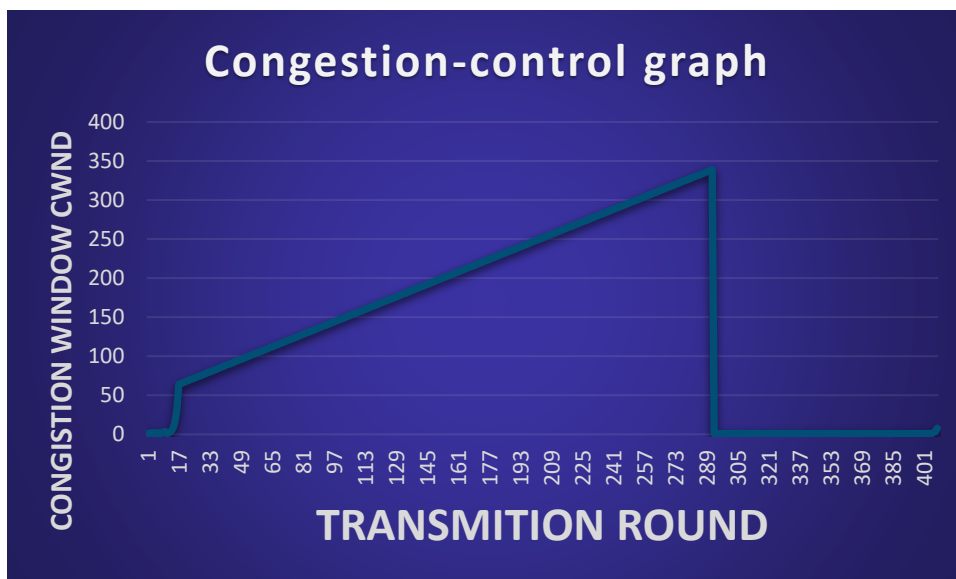
when probability is 0.01:



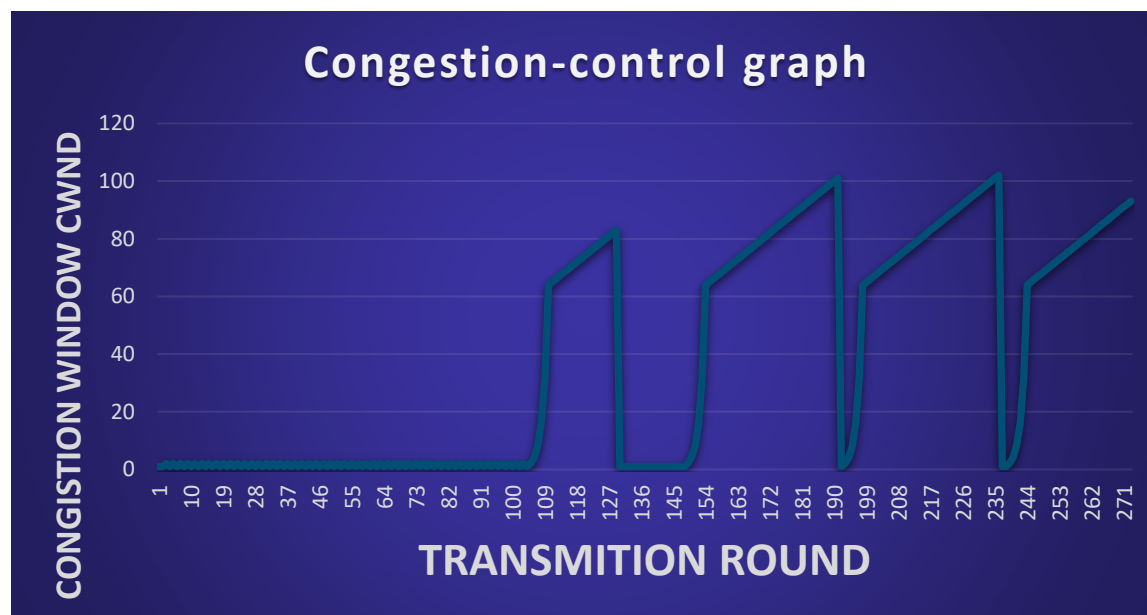
when probability is 0.05:



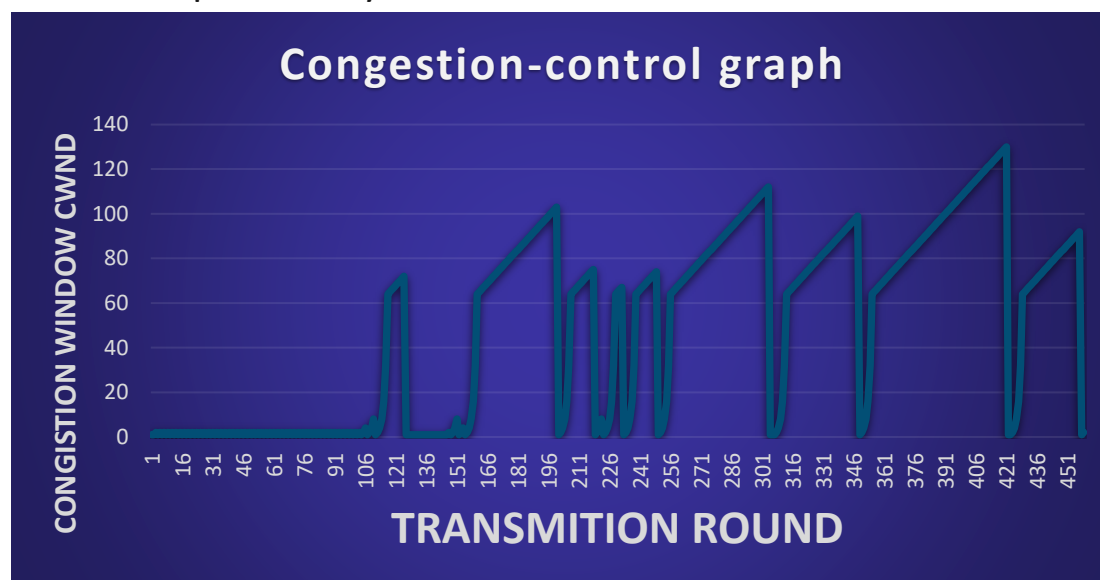
When probability is 0.1:



When the probability is 0.3:



When the probability is 0.4:



And here are more runs in that [Folder](#).

To set the Arguments for the server we make commands file (command.txt) where we put the coming arguments:

in the order shown, one item per line :

Well-known port number for server.

Random generator seed value.

Probability p of datagram loss (real number in the range [0.0 , 1.0]

Client:

Packet type and fields:

There are two kinds of packets: Data packets & Ack packets

```
struct ack_packet {  
    uint16_t cksum;  
    uint16_t len;  
    uint32_t ackno;  
};
```

```
struct packet {  
    uint16_t cksum;  
    uint16_t len;  
    uint32_t seqno;  
    char data [500];  
};
```

Create Packet method:

```
packet create_packet(string file_name){  
    struct packet p{};  
    strcpy( dest: p.data, src: file_name.c_str());  
    p.seqno = 0;  
    p.cksum = 0;  
    p.len = file_name.length() + sizeof(p.cksum) + sizeof(p.len) + sizeof(p.seqno);  
    return p;  
}
```

Send ack method:

```
void send_ack(int client_socket, struct sockaddr_in server_address, int seqNum){
    struct ack_packet ack;
    ack.ackno = seqNum;
    ack.len = sizeof(ack);
    ack.cksum = get_ack_checksum( len: ack.len, ackNo: ack.ackno);
    char* ack_buf = new char[MAXIMUM_SEGMENT_SIZE];
    memset( s: ack_buf, c: 0, n: MAXIMUM_SEGMENT_SIZE);
    memcpy( dest: ack_buf, src: &ack, n: sizeof(ack));
    ssize_t bytesSent = sendto( fd: client_socket, buf: ack_buf, n: MAXIMUM_SEGMENT_SIZE, flags: 0,
                                addr: (struct sockaddr *)&server_address, addr_len: sizeof(struct sockaddr));
    if (bytesSent == -1) {
        perror( s: "error in sending the ack ! ");
        exit( status: 1);
    } else {
        cout << "Ack for packet seq. Num " << seqNum << " is sent." << endl << flush;
    }
}
```

Main method:

Read IP address, port number and filename and then create connection.

```
int main() {
    vector<string> commands = readCommand();
    string IP_Address = commands[0];
    int port = stoi( str: commands[1]);
    string fileName = commands[2];
    struct sockaddr_in server_address;
    int client_socket;
    memset( s: &client_socket, c: '0', n: sizeof(client_socket));
    if ((client_socket = socket( domain: AF_INET, type: SOCK_DGRAM, protocol: IPPROTO_UDP)) < 0) {
        perror( s: "failed");
        exit( status: 1);
    }

    memset( s: &server_address, c: 0, n: sizeof(server_address));
    server_address.sin_family = AF_INET;
    server_address.sin_addr.s_addr = INADDR_ANY;
    server_address.sin_port = htons( hostshort: port);
```

Create packet with the file name and send it to server then receive ack.

```
cout << "File Name is : " << fileName << " The length of the Name : " << fileName.size() << endl << flush;
struct packet fileName_packet = create_packet( file_name: fileName);
char* buffer = new char[MAXIMUM_SEGMENT_SIZE];
memset( s: buffer, c: 0, n: MAXIMUM_SEGMENT_SIZE );
memcpy( dest: buffer, src: &fileName_packet, n: sizeof(fileName_packet));
ssize_t bytesSent = sendto( fd: client_socket, buf: buffer, n: MAXIMUM_SEGMENT_SIZE, flags: 0, addr: (struct sockaddr*)&server_address, addr_len: sizeof(struct sockaddr));
if (bytesSent == -1) {
    perror( s: "Error in sending the file name! ");
    exit( status: 1);
} else {
    cout << "Client Sent The file Name ." << endl << flush;
}
char rec_buffer[MAXIMUM_SEGMENT_SIZE];
socklen_t addrlen = sizeof(server_address);
ssize_t Received_bytes = recvfrom( fd: client_socket, buf: rec_buffer, n: MAXIMUM_SEGMENT_SIZE, flags: 0, addr: (struct sockaddr*)&server_address, addr_len: &addrlen);
if (Received_bytes < 0){
    perror( s: "error in receiving file name ack .");
    exit( status: 1);
}
```

Iterate over the number of packets and print the number of packets received and its sequence number and if it lost it will get error in receiving this packet.

We check on the checksum of the packet received and if it doesn't equal the checksum of the original file it will get data is corrupted.

```
auto* ackPacket = (struct ack_packet*) rec_buffer;
cout << "Number of packets " << ackPacket->len << endl;
long numberOfPackets = ackPacket->len;
string fileContents [numberOfPackets];
bool recieved[nOfPackets] = { [0]: false};
int i = 1;
int expectedSeqNum = 0;
while (i <= numberOfPackets){
    memset( s: rec_buffer, c: 0, n: MAXIMUM_SEGMENT_SIZE);
    ssize_t bytesReceived = recvfrom( fd: client_socket, buf: rec_buffer, n: MAXIMUM_SEGMENT_SIZE, flags: 0, addr: (struct sockaddr*)&server_address, addr_len: &addrlen);
    if (bytesReceived == -1){
        perror( s: "Error receiving data packet.");
        break;
    }
    auto* data_packet = (struct packet*) rec_buffer;
    cout << "packet "<<i<<" received" <<endl<<flush;
    cout << "Sequence Number : " << data_packet->seqno << endl<<flush;
    int len = data_packet->len;
    for (int j = 0 ; j < len ; j++){
        fileContents[data_packet->seqno] += data_packet->data[j];
    }
    if (get_data_checksum( content: fileContents[data_packet->seqno], len: data_packet->len, seqNo: data_packet->seqno) != data_packet->cksum){
        cout << "corrupted data packet !" << endl << flush;
    }
    send_ack(client_socket, server_address, seqNum: data_packet->seqno);
    i++;
}
```

Then we write in file.

```
string content = "";
for (int i = 0; i < numberOfPackets ; i++){
    content += fileContents[i];
}
writeFile(fileName, content);
cout << "File is received successfully . " << endl << flush;

return 0;
```

Get ack checksum method:

```
uint16_t get_ack_checksum(uint16_t len, uint32_t ackNo){
    uint32_t sum = 0;
    sum += len;
    sum += ackNo;
    while(sum >> 16)
        sum = (sum & 0xFFFF) + (sum >> 16);
    uint16_t CSum = (uint16_t)(~sum);
    return CSum;
}
```

Ge data checksum method:

```
uint16_t get_data_checksum(string content, uint16_t len, uint32_t seqNo){
    uint32_t sum = 0;
    sum += len;
    sum += seqNo;
    int n;
    n = content.length();
    char arr[n+1];
    strcpy(dest: arr, src: content.c_str());
    for (int i = 0; i < n; i++){
        sum += arr[i];
    }
    while (sum >> 16){
        sum = (sum & 0xFFFF) + (sum >> 16);
    }
    uint16_t OCSum = (uint16_t) (~sum);
    return OCSum;
}
```

Read command method:

```
vector<string> readCommand(){
    string fileName = "command.txt";
    vector<string> commands;
    string line;
    ifstream f;
    f.open( fileName);
    while(getline( f, line))
    {
        commands.push_back(line);
    }
    return commands;
}
```

Write File method:

```
void writeFile (string fileName, string content){
    ofstream f_stream( fileName.c_str());
    f_stream.write( content.c_str(), content.length());
}
```

To set the Arguments for the client we make commands file (command.txt)

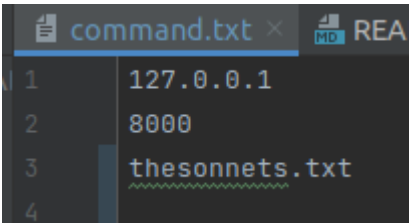
where we put the coming arguments:

in the order shown, one item per line :

IP address of server.

Well-known port number of server.

Filename to be transferred (should be a large file).



The screenshot shows a text editor window titled 'command.txt' with a line number column on the left. The content of the file is as follows:

Line	Content
1	127.0.0.1
2	8000
3	thesonnets.txt
4	