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;
};
```

```
        /* Data-only packets */
struct packet {
        /* Header */
        uint16_t cksum;  /* Optional bonus part */
        uint16_t len;
        uint32_t seqno;
        /* Data */
        char data[500];  /* Not always 500 bytes, can be less */
};
```

```
        /* Ack-only packets are only 8 bytes */
struct ack_packet {
        uint16_t cksum;  /* Optional bonus part */
        uint16_t len;
        uint32_t ackno;
};
```

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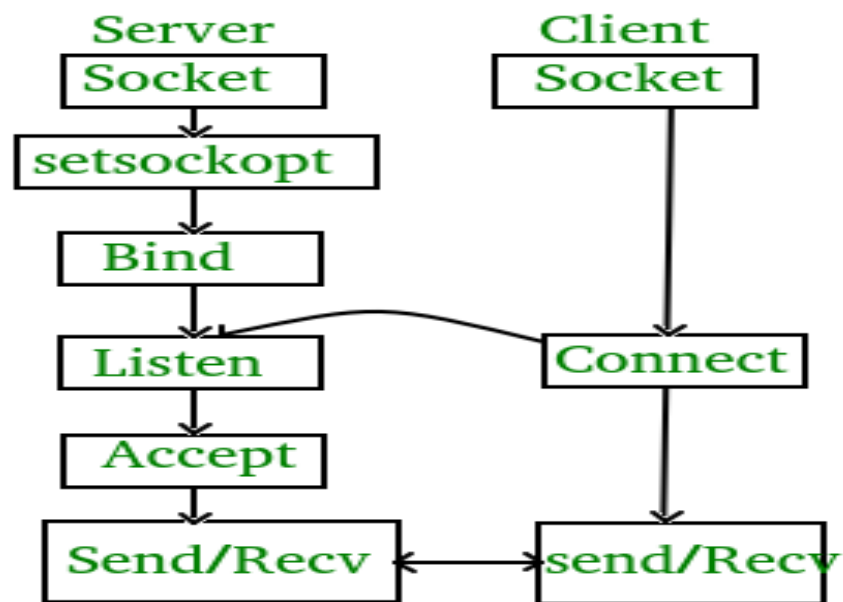
Server:

Here is the code until the binding:

```cpp
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:

```cpp
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.
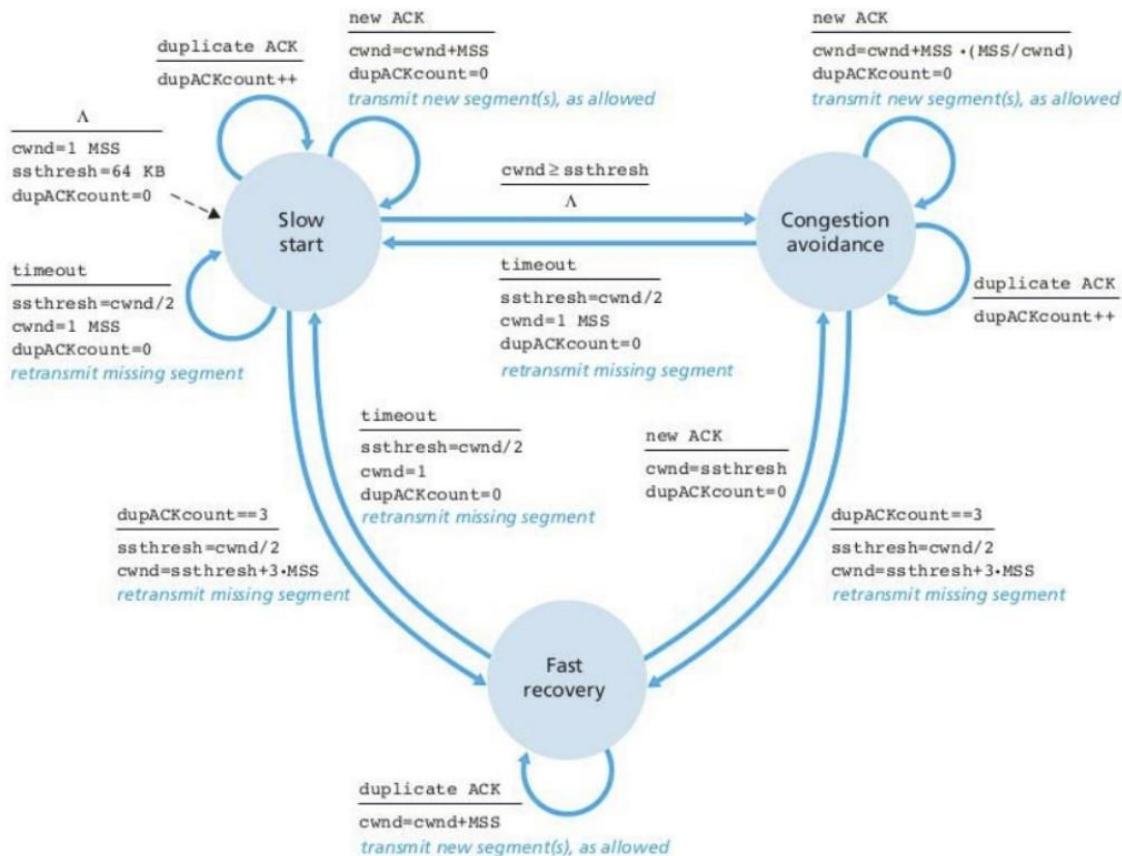as shown in the code above.

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 backet that need to be sent to the client.

Then transmitting the packets will starts following that chart:

duplicate ACK
dupACKcount++

new ACK
cwnd=cwnd+MSS
dupACKcount=0
transmit new segment(s), as allowed

new ACK
cwnd=cwnd+MSS ·(MSS/cwnd)
dupACKcount=0
transmit new segment(s), as allowed

Λ
cwnd=1 MSS
ssthresh=64 KB
dupACKcount=0

cwnd ≥ ssthresh
Λ

Slow start

Congestion avoidance

timeout
ssthresh=cwnd/2
cwnd=1 MSS
dupACKcount=0
retransmit missing segment

timeout
ssthresh=cwnd/2
cwnd=1 MSS
dupACKcount=0
retransmit missing segment

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

For simulating the corrubted packets we subtract one fronm 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(calculatedusing 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:

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	<= (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:
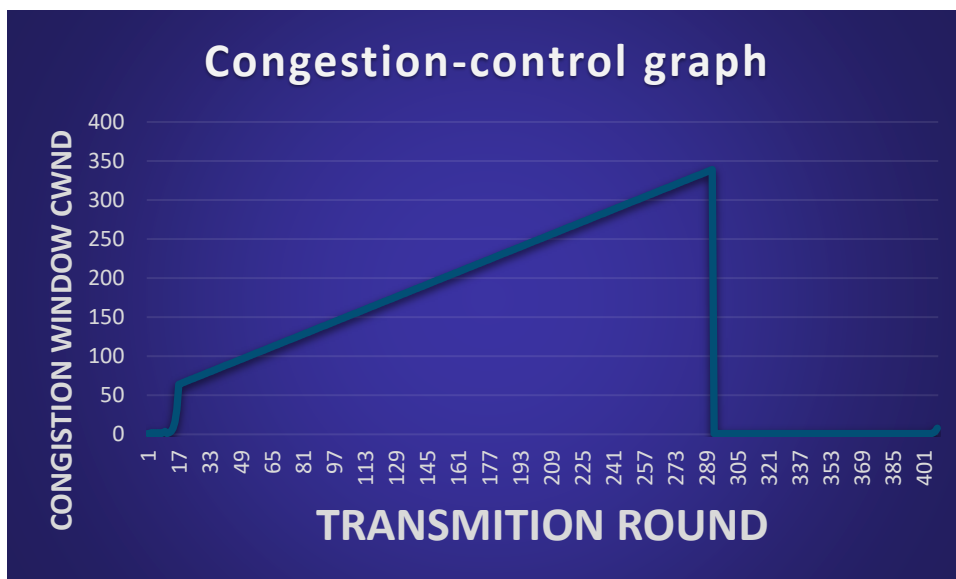
when probability is 0.01:
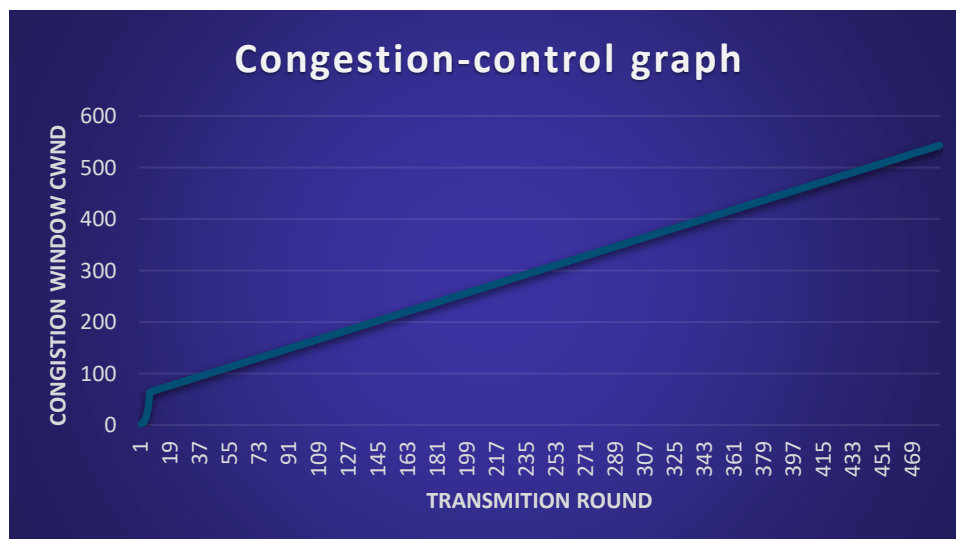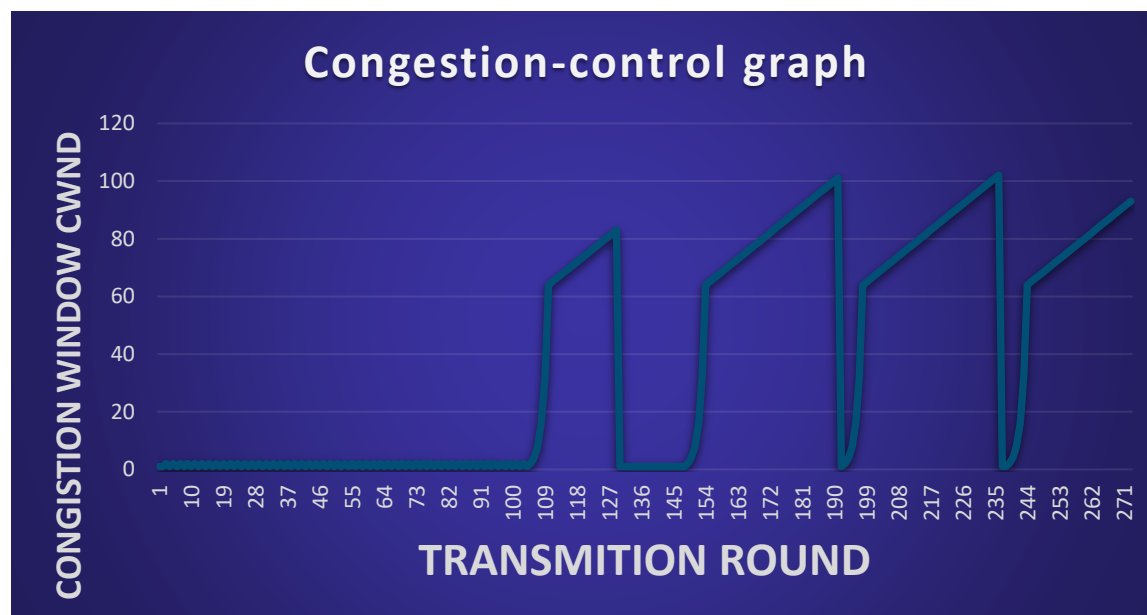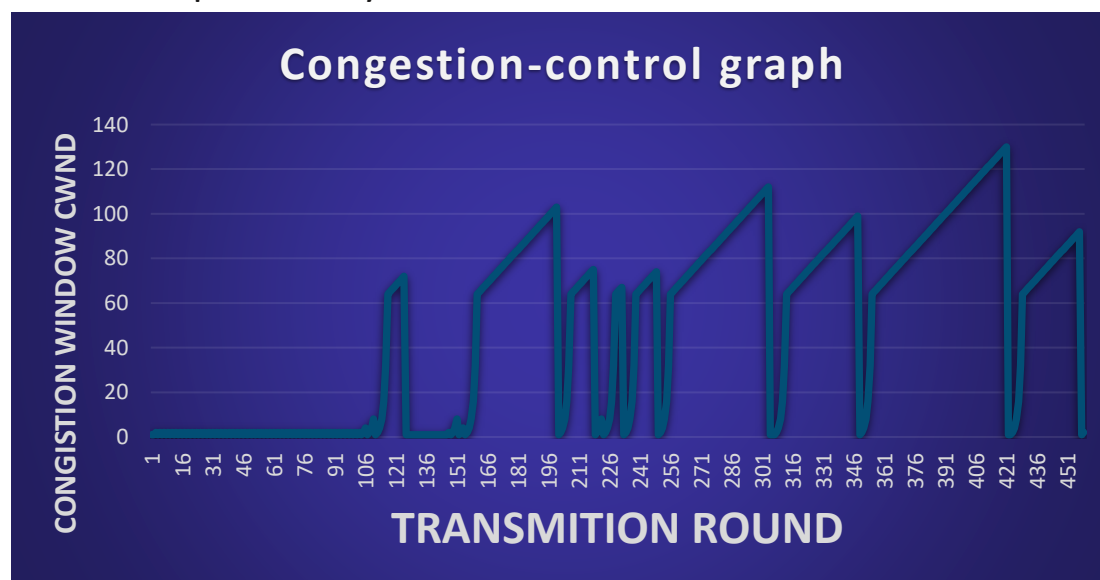
when probability is 0.05:
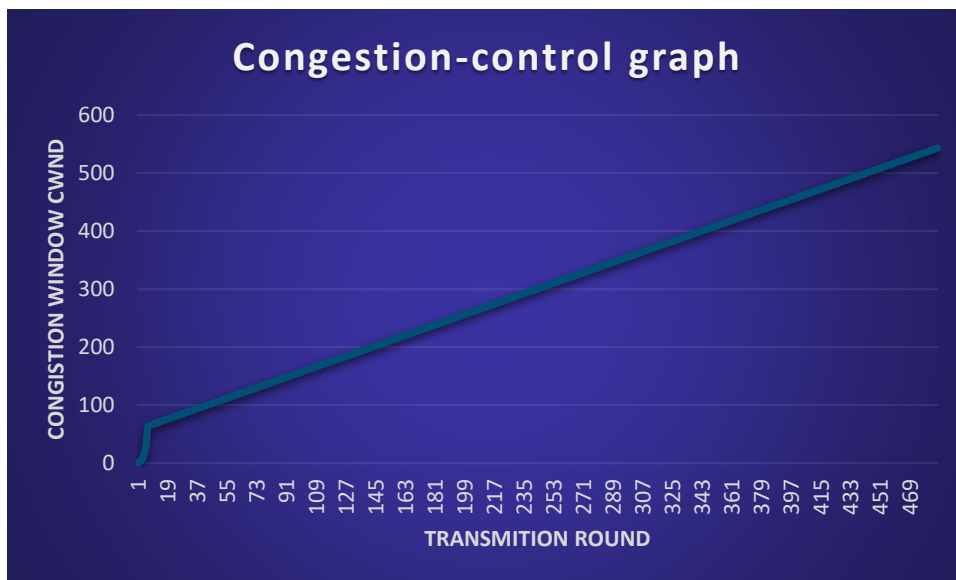
When probabiliy is 0.1:

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:

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).