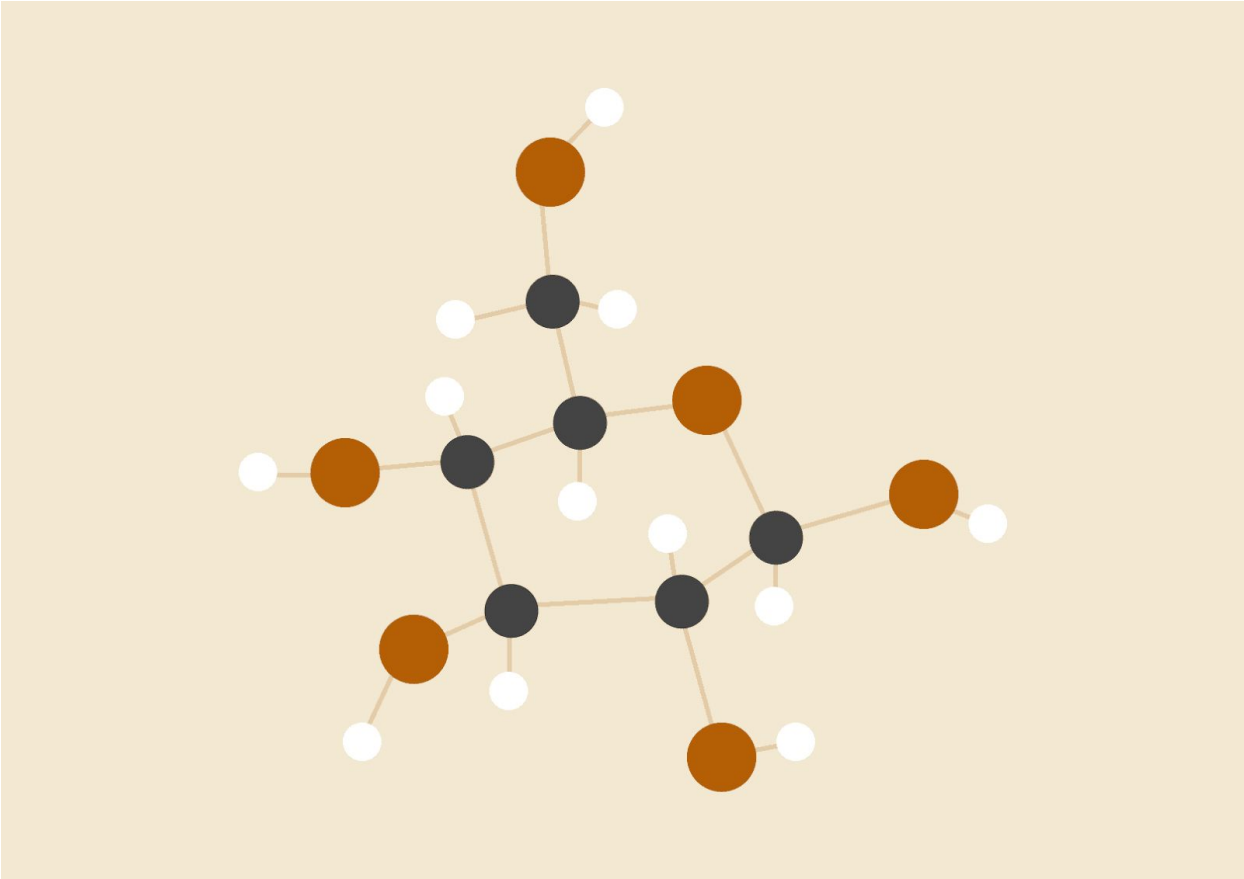
Assignment2

Implementing a Reliable Data Transport Protocol



Mostafa Mohamed Elsayed Fares

51

Problem statement:

Implement your own socket layer and reliable transport layer. You'll get to learn how the socket interface is implemented by the kernel and how a reliable transport protocol like TCP runs on top of an unreliable delivery mechanism (which is the real world, since in real world networks nothing is reliable).

Reliability Specifications

1. Specifications

Suppose you've a file and you want to send this file from one side to the other (server to client). You will need to 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.

implementation:
FIXED DATA SIZE = 256

2. Packet types and fields

- Data Packet:

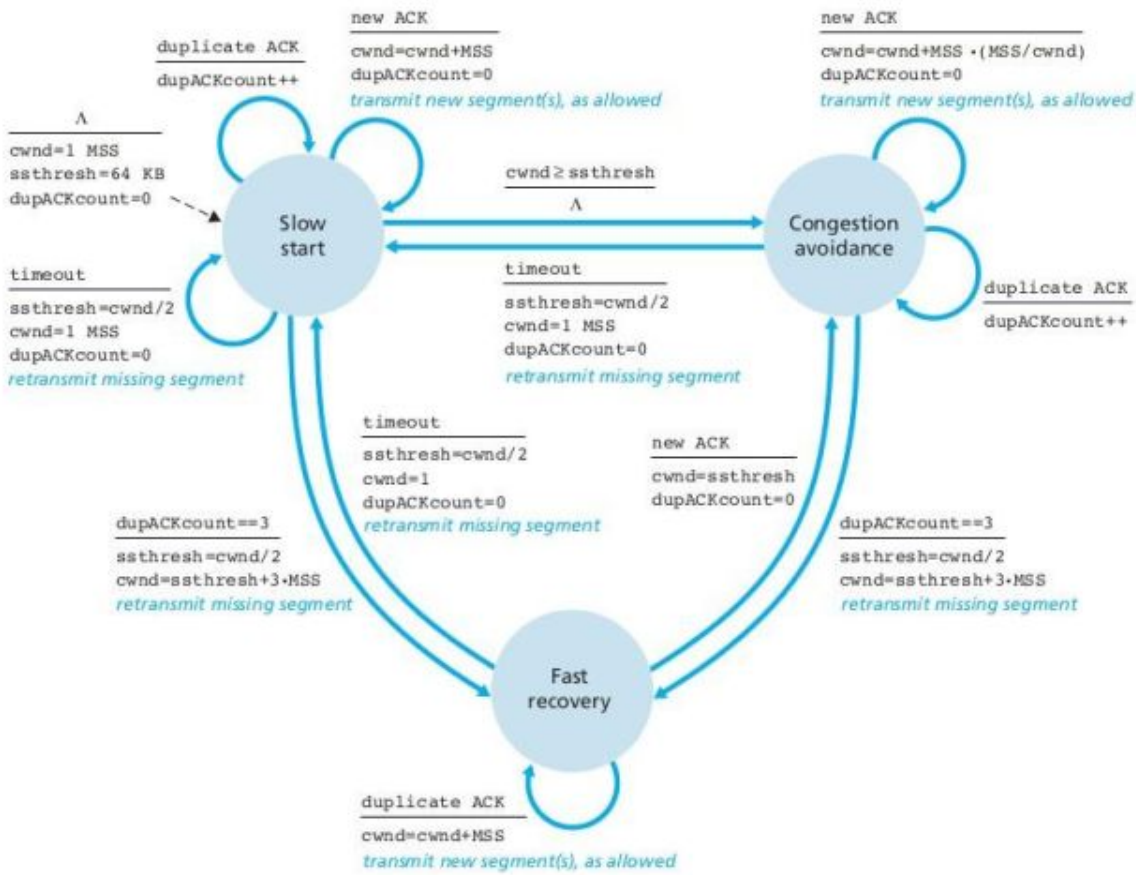
```
struct packet {
    /*header*/
    uint32_t seqno = 0;
    uint32_t ackno = 0;
    uint16_t cksum = 0;
    bool SYN = false;
    bool ACK = false;
    bool FIN = false;
    uint16_t len = 0;
    /*Data*/
    char data[DATASIZE];
};
```

- Ack/SYN Packet:

```
struct ack_packet {
    /*header*/
    uint32_t seqno = 0;
    uint32_t ackno = 0;
    uint16_t cksum = 0;
    bool SYN = false;
    bool ACK = false;
    bool FIN = false;
};
```

3. Congestion Control:

Implement congestion control, following the FSM of TCP congestion control given below:



- **Some Variables meaning:**
  - Vector packets → to store recently created packets.
  - Vector tracwin → to record window_size in each transmission.

- windowSize → represents current window size.
- slowStartThreshold = (SSTHRESH * 1024) / DATASIZE; (ex: SSTHRESH = 64)k

- Send mechanism:

```
if (curseqno >= windowBase && curseqno < windowBase + windowSize &&
    !(feof(fp) && curseqno >= packcount)) {

    packet* newPacket;
    /*read from file*/
    char* dataRead = new char[DATASIZE];
    int bytesRead;
    if (!feof(fp)) {
        newPacket = new packet;
        bytesRead = fread(dataRead, 1, DATASIZE, fp);
        memcpy(newPacket->data, dataRead, bytesRead);
        //cout << "data in packet from file" << newPacket->data << endl;
        newPacket->len = bytesRead;
        newPacket->seqno = packcount;
        newPacket->ackno = clientSeqNo;
        if (feof(fp)) {
            newPacket->FIN = true;
        }
        packets.push_back(newPacket);
        packcount++;
    }

    /*print packet*/
    cout << "send packet:";
    h.ppacket(*packets[curseqno]);
    cout << endl;

    /*send packet*/
    h.send(this->mysock, this->clientAddr, packets[curseqno]);
    /*set timer on base*/
    if (curseqno == windowBase)
        timer.startTimer();
    curseqno++;
    tracwin.push_back(windowSize);
}
```

- Receive mechanism and **Congestion Control** :
  STATUS state : It is an enum representing the system state (ex: SLOW START , ..).

- if base_window packet timer time out:
TIMEOUT = 2 sec

```
/*receive ack*/
ZeroMemory(ackPacket, sizeof(struct ack_packet));
int SenderAddrSize = sizeof(this->clientAddr);
int status = h.receiveAck(this->mysock, &this->clientAddr, ackPacket, &SenderAddrSize);
if (status == WSAEWOULDBLOCK) {
    if (timer.getElapsedTimeInSec() > TIMEOUT) {
        /*Time_out*/
        cout << "Time_out base: " << windowBase;
        slowStartThreshold = windowSize / 2;
        windowSize = 1;
        dupAckCount = 0;
        curseqno = windowBase;
        if (state != SLOWSTART)
            state = SLOWSTART;
    }
}
```

-if ack_packet received and it is duplicant:

```
else if (status != SOCKET_ERROR) {
    /*recv ack packet*/
    cout << "recevied ack packet:" << endl;
    h.pAckPacket(ackPacket);
    if (ackPacket->ACK) {
        /*update window*/
        windowBase = ackPacket->ackno + 1;
        clientSeqNo = ackPacket->seqno;
        if (curseqno == windowBase)
            timer.stopTimer();
        else
            timer.startTimer();
        /*check dupAcks*/
        if (lastAckNo == ackPacket->ackno) {
            /*dupAck*/
            if (state == FASTRECOV) {
                windowSize += 1;
            }
            else {
                dupAckCount++;
                /*if 3 dupAcks*/
                if (dupAckCount >= 3) {
                    slowStartThreshold = windowSize / 2;
                    windowSize = slowStartThreshold + 1;//3
                    curseqno = windowBase;
                    state = FASTRECOV;
                }
            }
        }
        else {
            /*new Ack*/
```

-if ack_packet received and it is **not** duplicant:

```
            else {
                /*new Ack*/
                lastAckNo = ackPacket->ackno;
                dupAckCount = 0;
                if (state == SLOWSTART) {
                    windowSize++;
                }
                else if (state == CONGAVOID) {
                    congCount++;
                    if (congCount >= windowSize) {
                        windowSize++;
                        congCount = 0;
                    }
                }
                else {
                    windowSize = slowStartThreshold;
                    state = CONGAVOID;
                }
            }
            /*flip from slow start to congestion avoidance*/
            if (windowSize >= slowStartThreshold && state == SLOWSTART)
                state = CONGAVOID;
        }
        if (feof(fp) && lastAckNo == packcount - 1) {
            break;
        }
    }
```

Handling time-out :

- Just put a timer on the first packet at the window base.

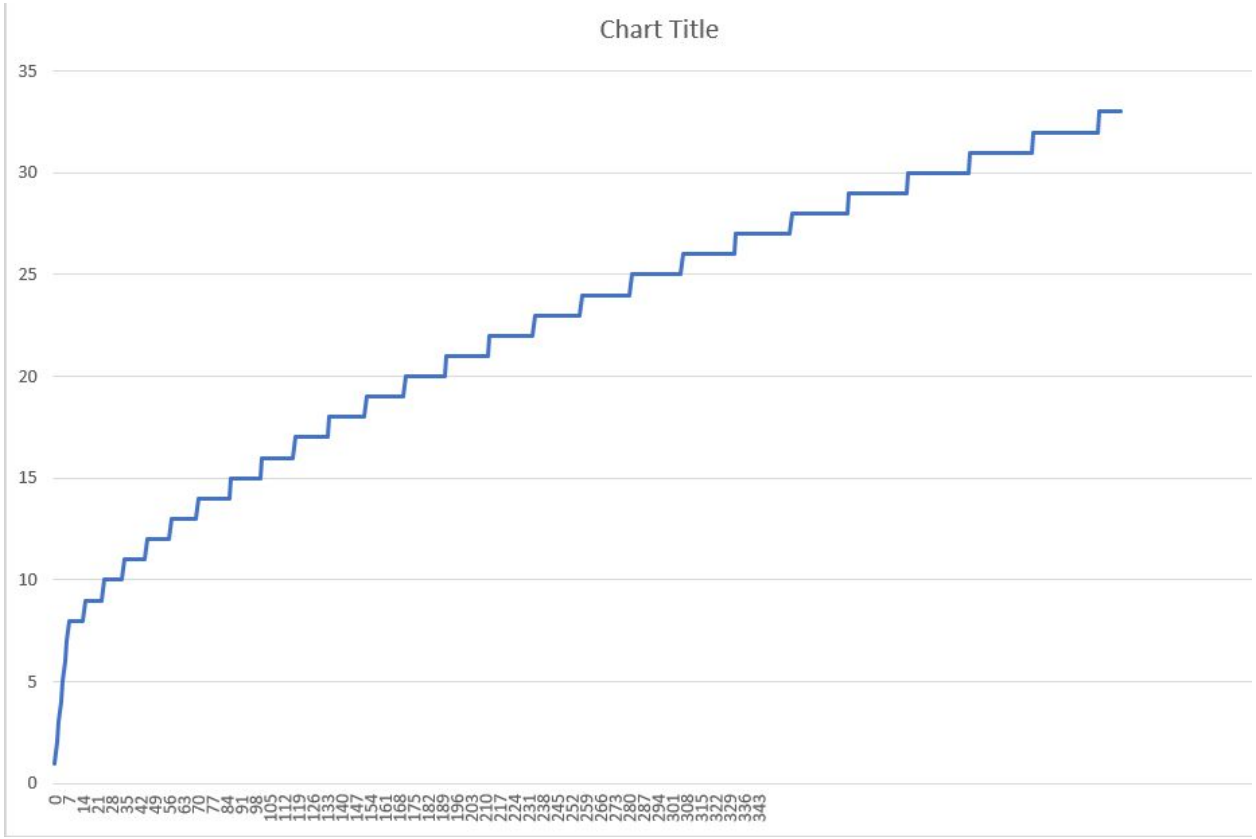
Arguments for the client and the server :

- Using a fixed IP address for both / same machine.
- Client: get filename from terminal , auto two-way handshaking with server.
- Server: listen for coming SYN requests and delegate a server thread using a new port to communicate with the new client and serve him.
- Probability p of datagram loss : get from terminal as well.

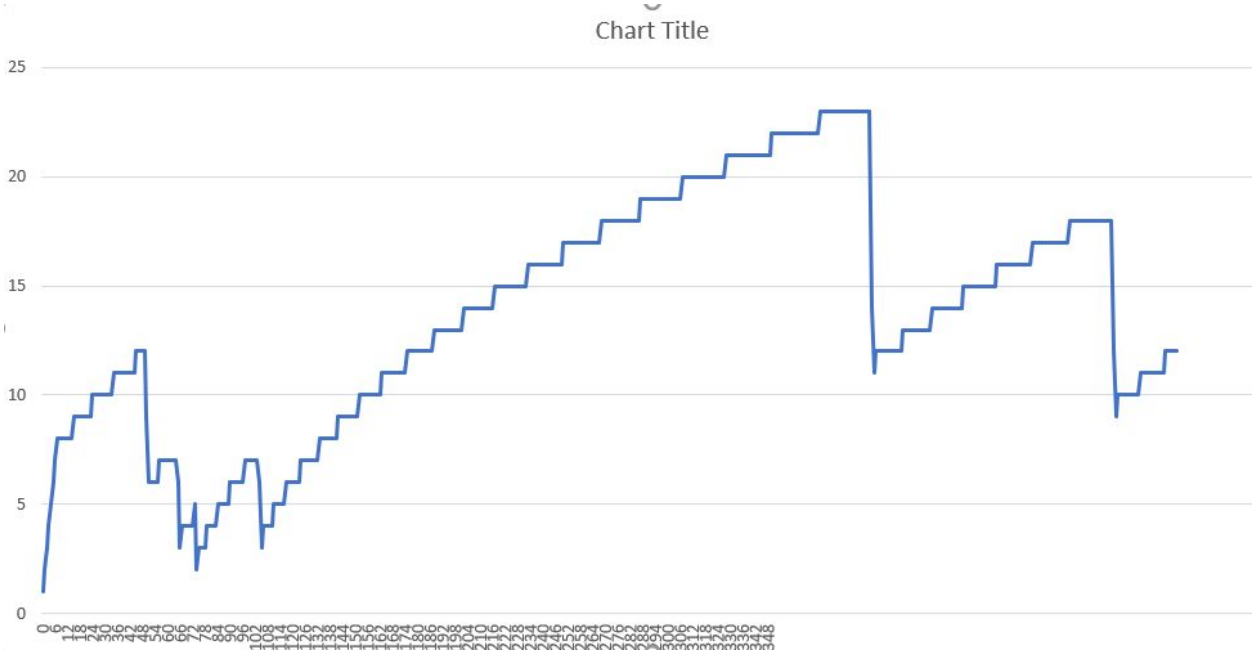
Network system analysis :

MSS = 8192 and file size = 4MB.

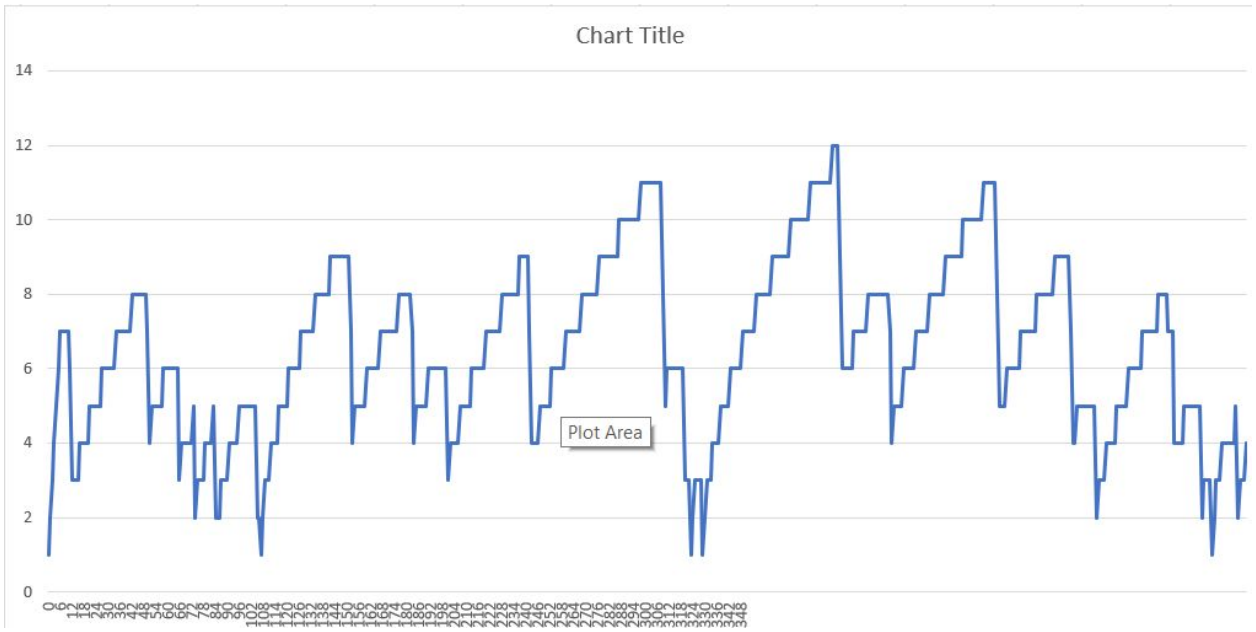
- At PLP = 0: 4.1MB image



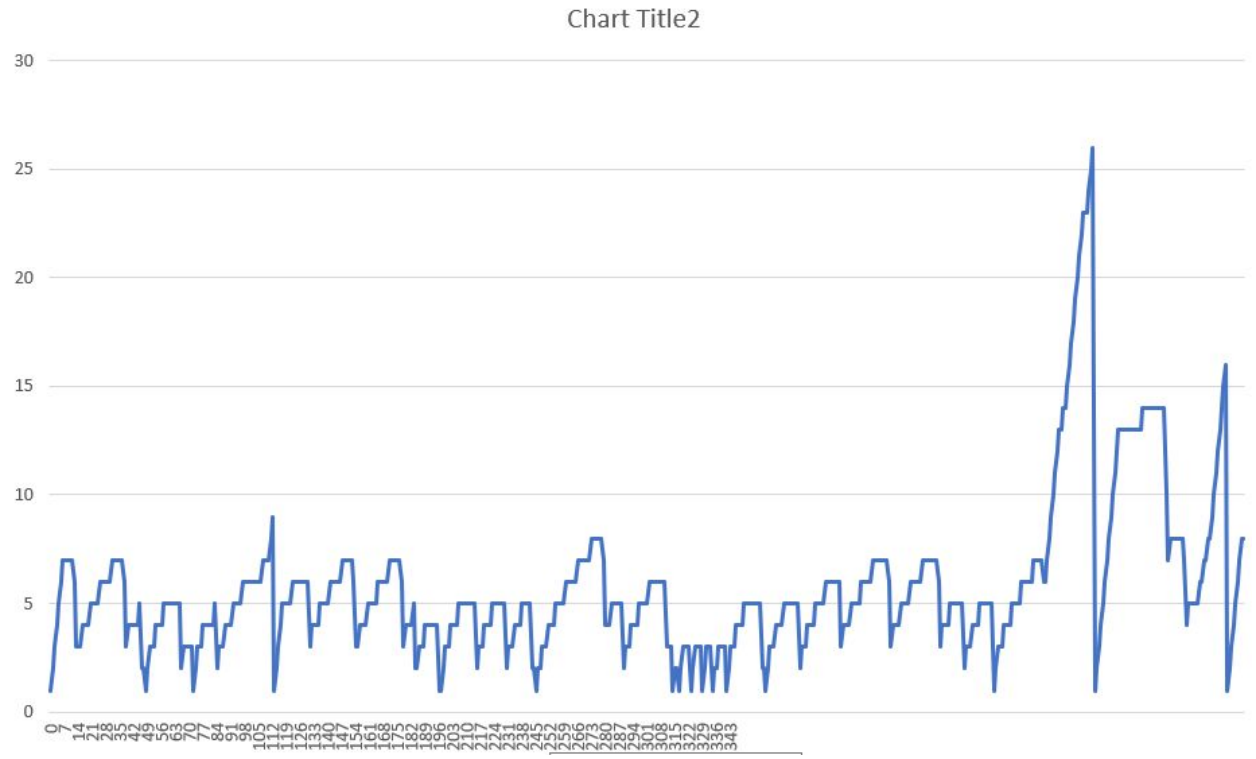
- At PLP = 1:



- At PLP = 5:



- At PLP = 10:



- At PLP = 30: