**P1**

1. A to S : Source port:16500, Destination port: 80
2. S to A : Source port : 80, Destination port: 16500
3. Yes
4. No

**P2**

**Host A:**

Source port = 80, Source IP address = B,

Destination port = 26145, Destination IP address = A

**Host C left process:**

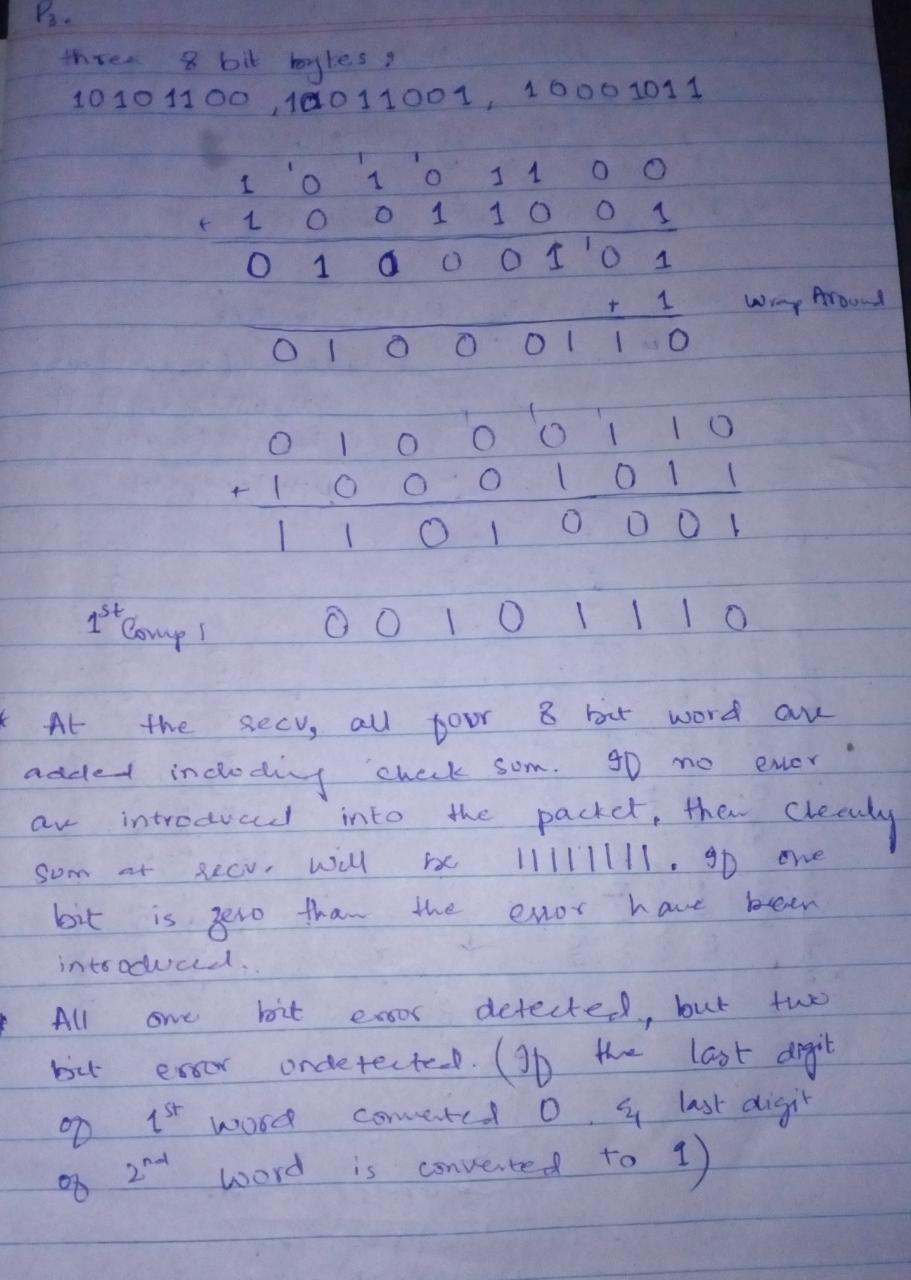
Source port = 80, Source IP address = B,

Destination port = 7532, Destination IP address = C

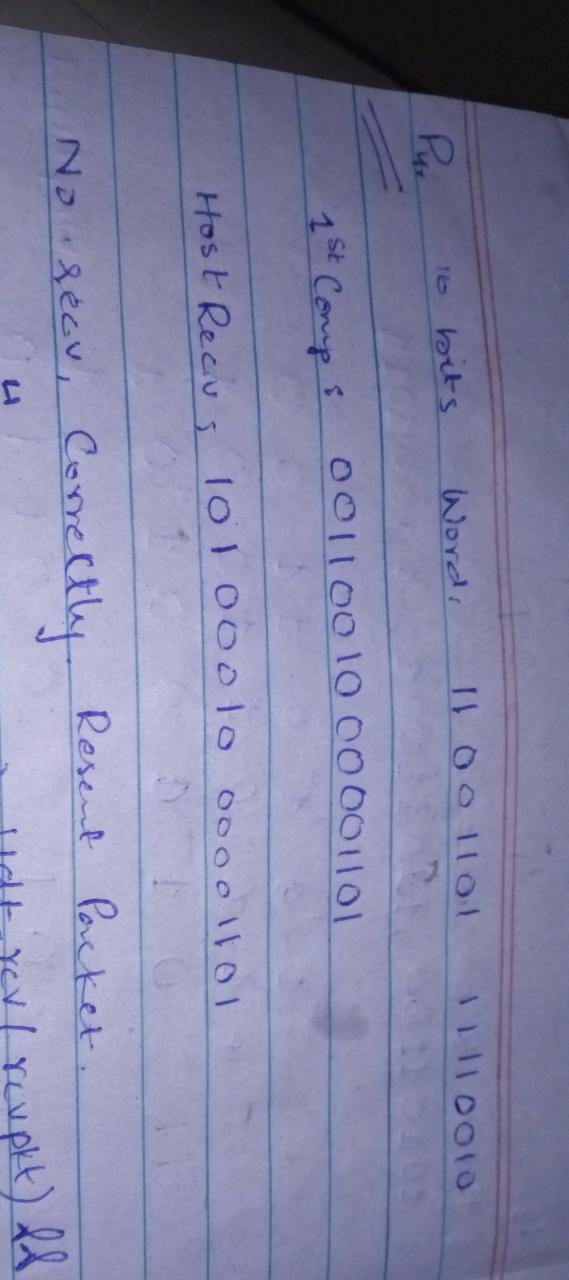
**Host C right process:**

Source port =80, Source IP address = b,

Destination port = 26145, Destination IP address = c

**P3**

**P4**



**P5**

No, the receiver cannot be absolutely certain that no bit errors have occurred because if the corresponding bits (that would be added together) of two 16-bit words in the packet were 0 and 1 then even if these get flipped to 1 and 0 respectively, the sum still remains the same and 1s complement will also be the same. This means the checksum will verify even if there was transmission error.

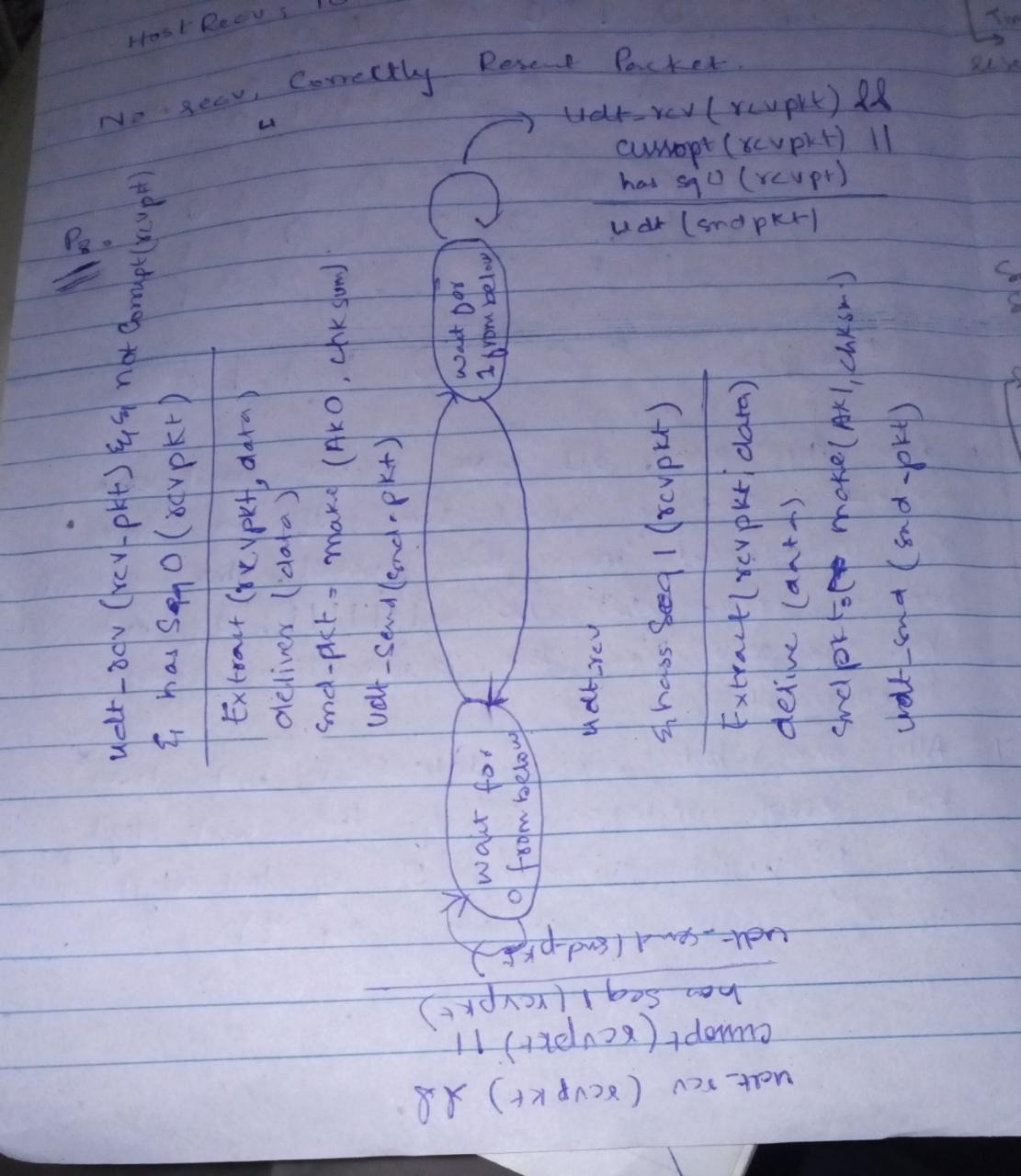
**P6**

Suppose the receiver receives the packet with sequence number 1 correctly, sends an ACK, and transitions to state **Wait for 0 from below**, waiting for a data packet with sequence number 0. However, the ACK is corrupted. When the rdt2.1 (sender) gets the corrupted ACK, it resends the packet with sequence number 1. However, the receiver is waiting for a packet with sequence number 0 and always sends a NAK when it doesn't receive a packet with sequence number 0. Hence the sender will always send a packet with sequence number 1, and the receiver will always send NAK for that packet. Hence the dead lock occur.

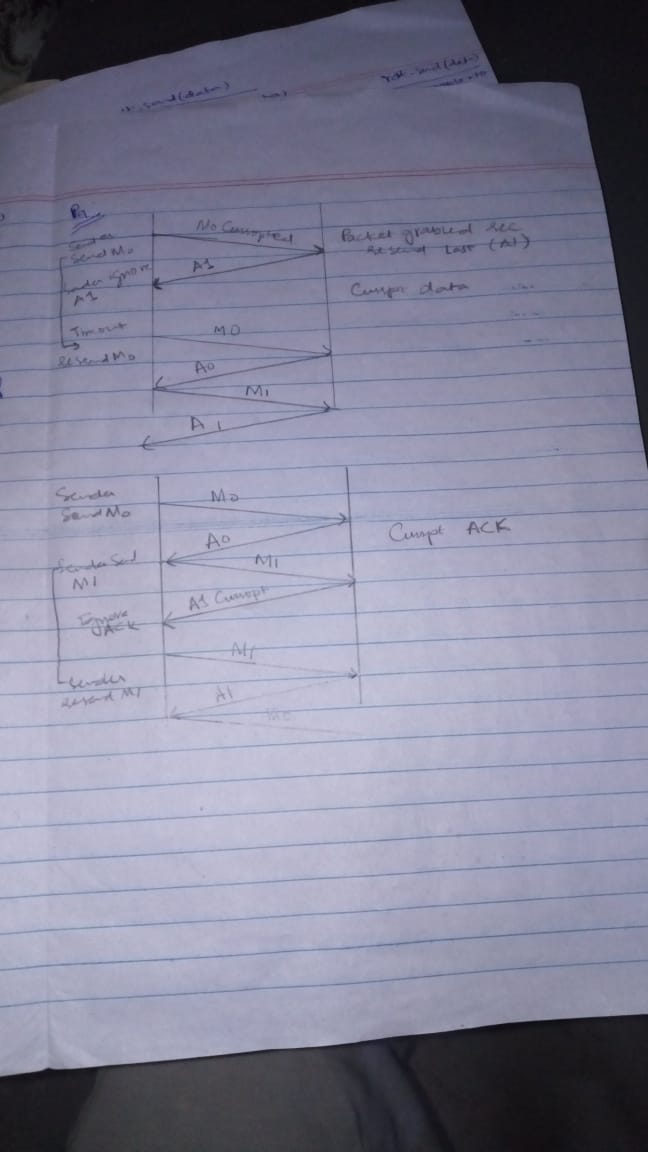
**P7**

The sender does not need a sequence number on an ACK to tell detect a duplicate ACK. A duplicate ACK is obvious to the rdt3.0 receiver, since when it has received the original ACK it transitioned to the next state. The duplicate ACK is not the ACK that the sender needs and hence is ignored by the rdt3.0 sender.

**P8**



**P9**



**P10**

We add a timer whose value is greater than the known round-trip propagation delay. If the timeout occurs, the most recently transmitted packet is retransmitted. Let’s see why this protocol will still work with the rdt2.1 receiver.

1. If timeout is caused by a lost of data packet. In this case, the receiver never received the previous transmission. If the timeout retransmission is received, it looks exactly the same as if the original transmission is being received.
2. If an ACK is lost. The receiver will eventually retransmit the packet on a timeout. But a retransmission is exactly the same action that if an ACK is garbled. The rdt 2.1 receiver can already handle the case of a garbled ACK.

**P11**

A deadlock situation occur. Here’s a scenario:

1. Sender sends pkt0, enter the **Wait for ACK0 state**, and waits for a packet back from the receiver
2. Receiver is in the **Wait for 0 from bellow state**, and receives a corrupted packet from the sender. Suppose it does not send anything back, and simply re-enters the **wait for 0 from below state**.

Now, the sender is waiting an ACK from the receiver, and the receiver is waiting for a data packet form the sender. Deadlock occur.

**P12**

The protocol would still work, since a retransmission would be what would happen if the packet received with errors has actually been lost and from the receiver standpoint, it never knows which of these events, if either, will occur.

In this case, if each extra copy of the packet is ACKed and each received extra ACK causes another extra copy of the current packet to be sent, the number of times packet n is sent will increase without bound as n approaches infinity.

**P13**

Tcp will adjust automatically. Udp not, application must do it. Eg: By calculating a rate and limiting it through.

**P14**

Yes, upon timeout the sender can conclude that packet x is received and send x+1.

No, the lost packet at the receiver x can only be noticed if packet x+1 is received, but the sender cannot send it as it does not have enough info to conclude that x was received and move on to x+1.

**P15**

RTT: 30 ms

L: 1500 \* 8 = 12000

R = 1Gbps (10^9 bit per seconds)

N =? ( windowsize )

Dtrans = L / R = 12000 / 10^9 = 12 microseconds.

Util = 98%

Usender = (L / R) / Rtt + L / R

0.98 = (0.012ms) N / 30.012ms

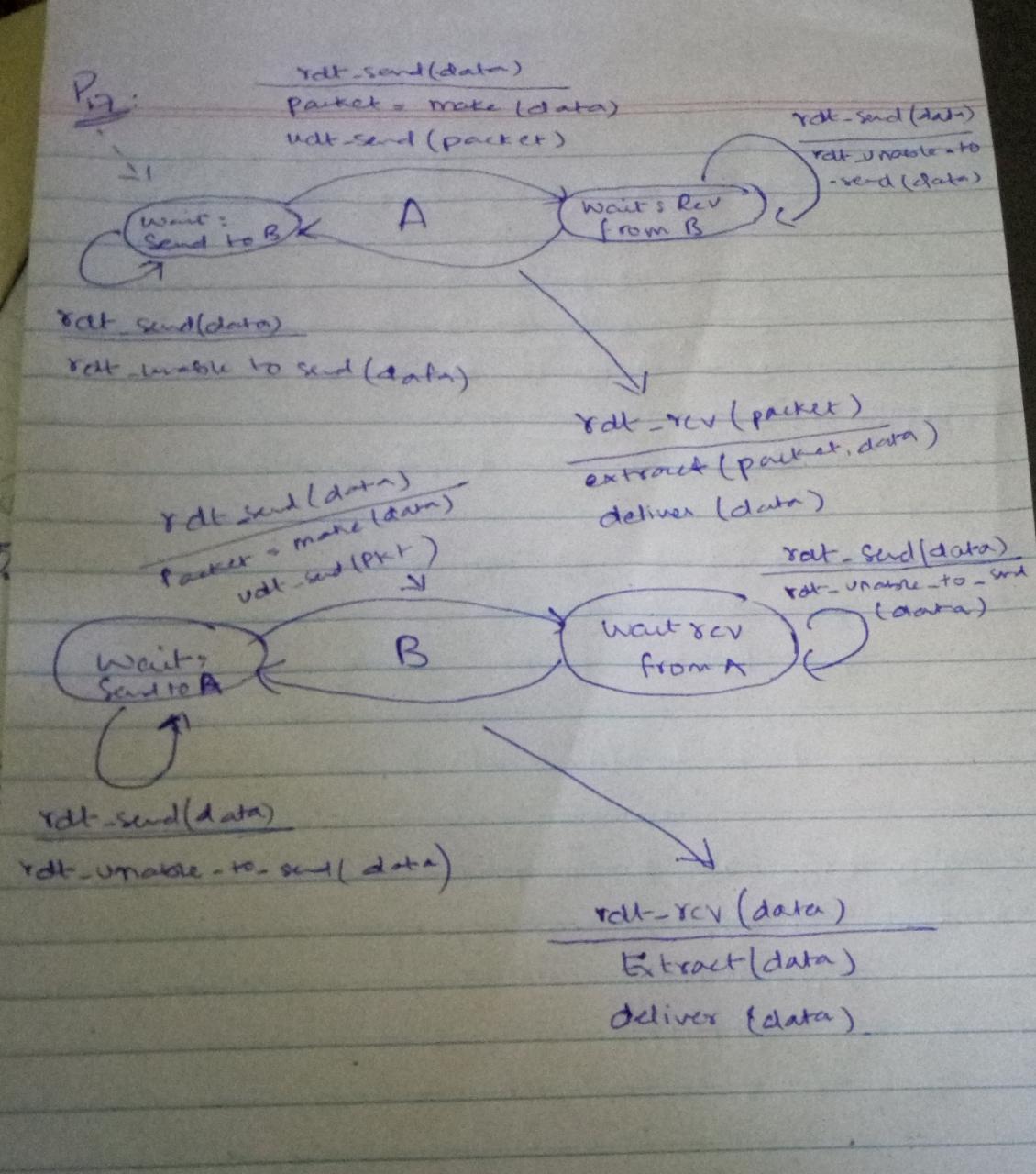
N = 2451 packets (approx).

**P16**

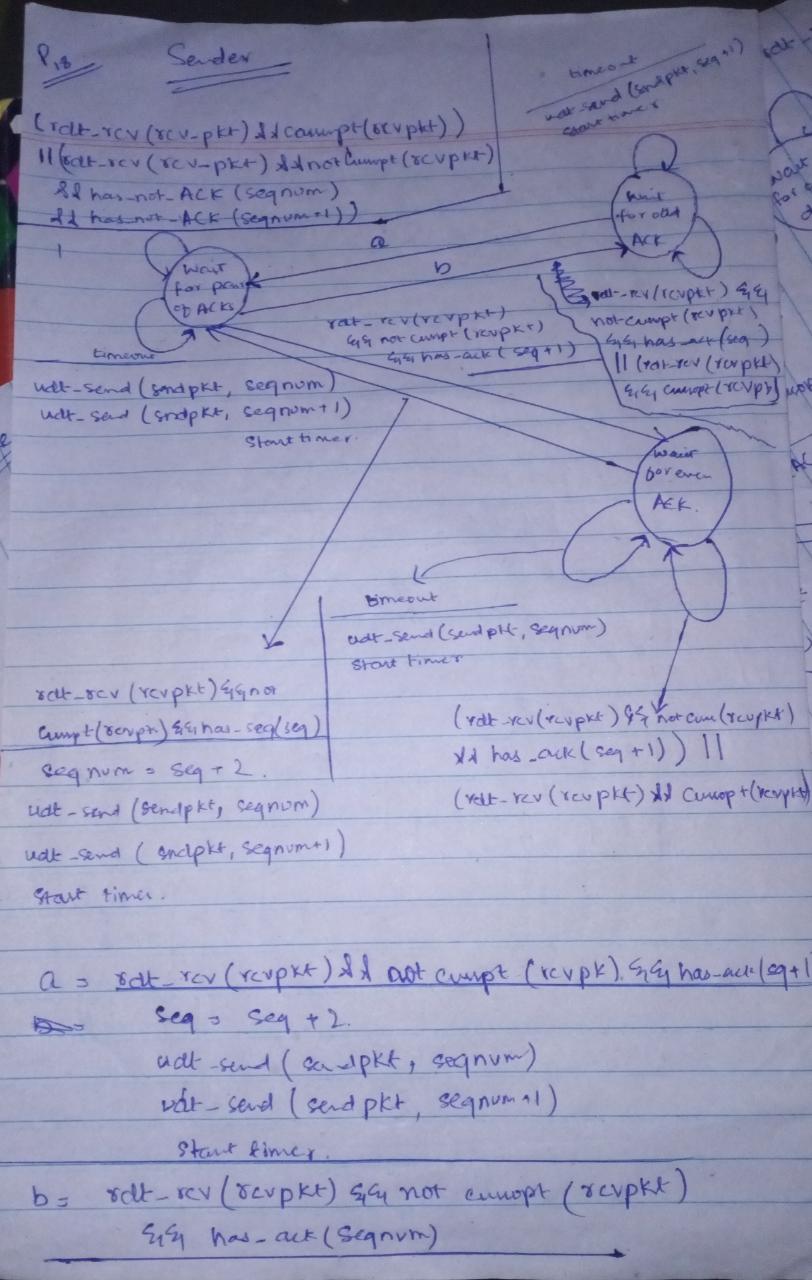
Yes. This actually causes the sender to send a number of pipelined data into the channel.

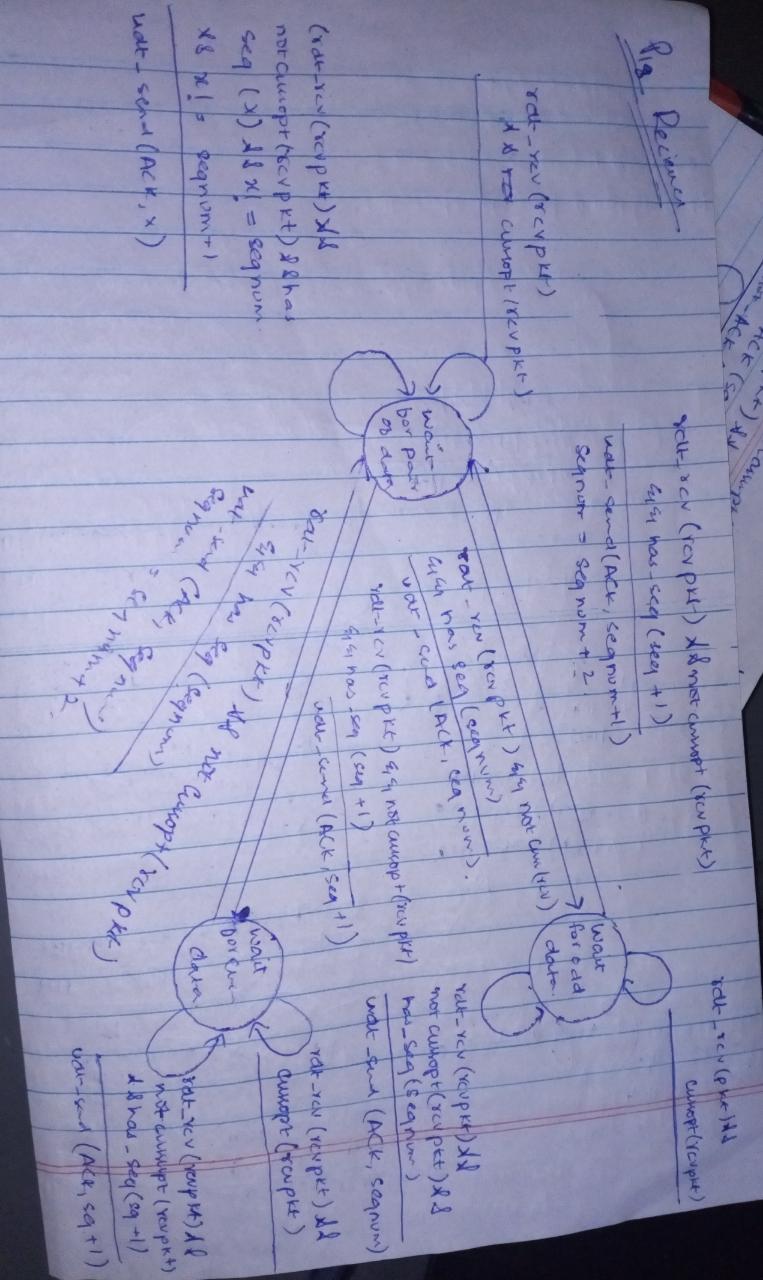
Here is one potential problem. If data segments are lost in the channel, then the sender of rdt 3.0 won’t re-send those segments. There are some additional mechanism in the application to recover from loss.

**P17**

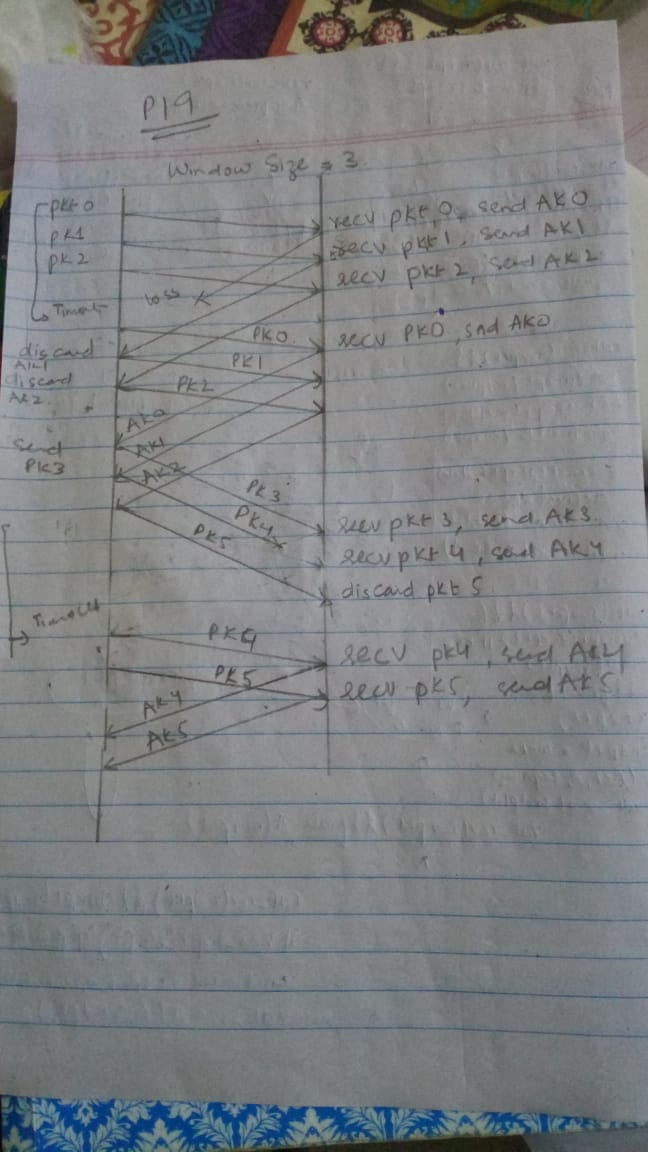
****

**P18**

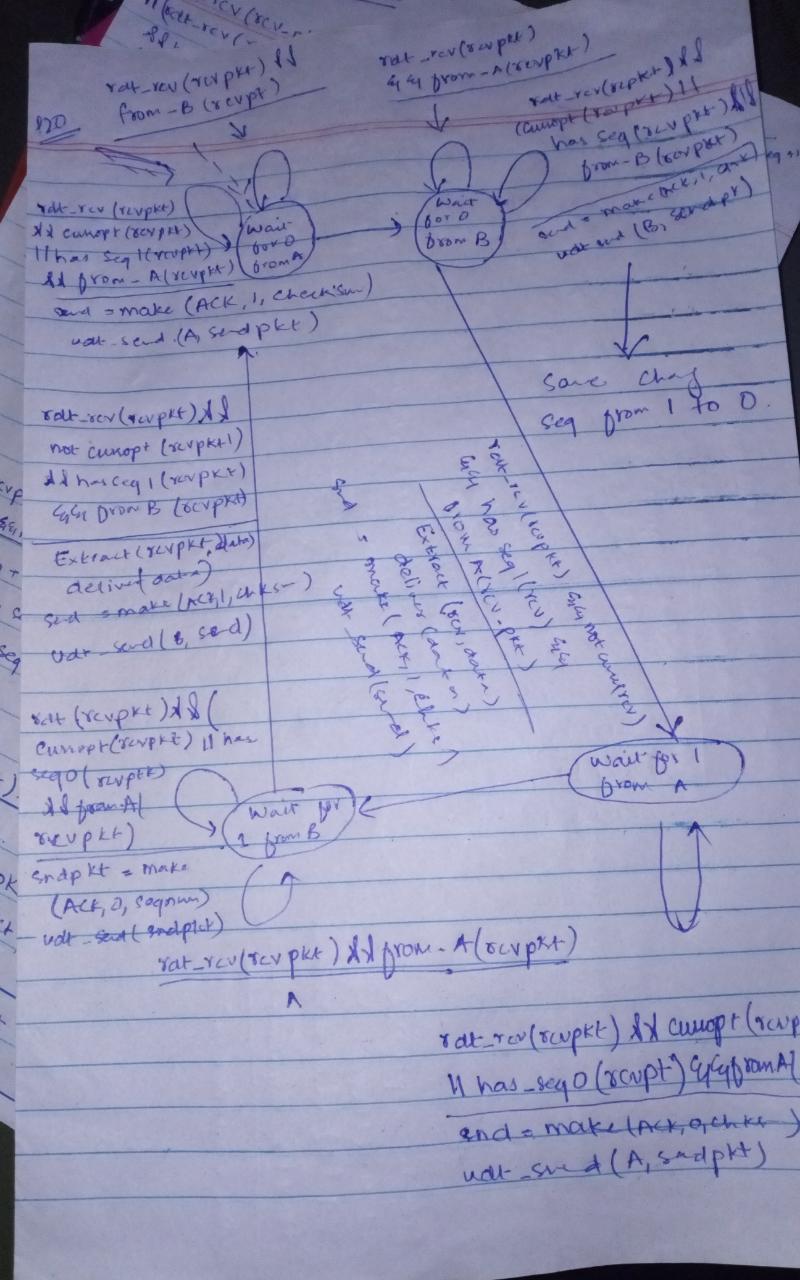
****

****

**P19**

****

**P20**

****