TCP packet exchange diagram between a client and a server (Back End) with a proxy (Front End) between them.

From this diagram we see that the total time is 4X + Y+ ST = 4*RTTFE + RTTBE + Search time

Chapter 3 Problems

Problem 1

	source port numbers	destination port numbers
a) A→S	467	23
b) B→S	513	23
c) S→A	23	467
d) S→B	23	513

e) Yes.
f) No.

Problem 2

Suppose the IP addresses of the hosts A, B, and C are a, b, c, respectively. (Note that a, b, c are distinct.)

To host A: Source port =80, source IP address = b, dest port = 26145, dest IP address = a

To host C, left process: Source port =80, source IP address = b, dest port = 7532, dest IP address = c

To host C, right process: Source port =80, source IP address = b, dest port = 26145, dest IP address = c

Problem 3

Note, wrap around if overflow.

$$\begin{array}{cccccccccc} & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\ + & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ \hline & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{array}$$

```
    1  0  1  1  1  0  0  1
+   0  1  1  1  0  1  0  0
-------------------------
    0  0  1  0  1  1  1  0
```

One's complement = 1 1 0 1 0 0 0 1.
To detect errors, the receiver adds the four words (the three original words and the checksum). If the sum contains a zero, the receiver knows there has been an error. All one-bit errors will be detected, but two-bit errors can be undetected (e.g., if the last digit of the first word is converted to a 0 and the last digit of the second word is converted to a 1).

Problem 4

a) Adding the two bytes gives 11000001. Taking the one's complement gives 00111110.

b) Adding the two bytes gives 01000000; the one's complement gives 10111111.

c) First byte = 01010100; second byte = 01101101.

Problem 5

No, the receiver cannot be absolutely certain that no bit errors have occurred. This is because of the manner in which the checksum for the packet is calculated. 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. Hence, the 1s complement the receiver calculates will also be the same. This means the checksum will verify even if there was transmission error.

Problem 6

Suppose the sender is in state "Wait for call 1 from above" and the receiver (the receiver shown in the homework problem) is in state "Wait for 1 from below." The sender sends a packet with sequence number 1, and transitions to "Wait for ACK or NAK 1," waiting for an ACK or NAK. Suppose now 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 (as shown in the home work problem) always sends a NAK when it doesn't get a packet with sequence number 0. Hence the sender will always be sending a packet with sequence number 1, and the receiver will always be NAKing that packet. Neither will progress forward from that state.

Problem 7

To best answer this question, consider why we needed sequence numbers in the first place. We saw that the sender needs sequence numbers so that the receiver can tell if a data packet is a duplicate of an already received data packet. In the case of ACKs, the sender does not need this info (i.e., 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.

Problem 8

The sender side of protocol rdt3.0 differs from the sender side of protocol 2.2 in that timeouts have been added. We have seen that the introduction of timeouts adds the possibility of duplicate packets into the sender-to-receiver data stream. However, the receiver in protocol rdt.2.2 can already handle duplicate packets. (Receiver-side duplicates in rdt 2.2 would arise if the receiver sent an ACK that was lost, and the sender then retransmitted the old data). Hence the receiver in protocol rdt2.2 will also work as the receiver in protocol rdt 3.0.

Problem 9

Suppose the protocol has been in operation for some time. The sender is in state “Wait for call from above” (top left hand corner) and the receiver is in state “Wait for 0 from below”. The scenarios for corrupted data and corrupted ACK are shown in Figure 1.

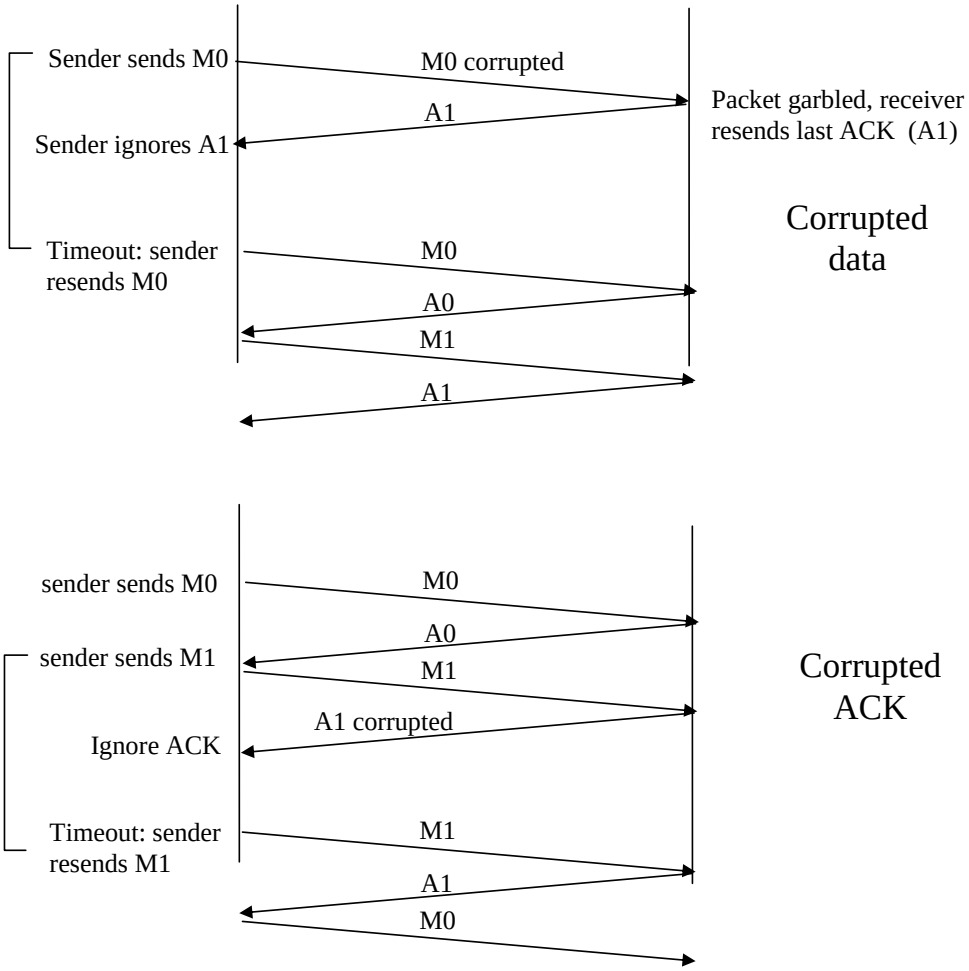


Figure 1: rdt 3.0 scenarios: corrupted data, corrupted ACK

Problem 10

Here, we add a timer, whose value is greater than the known round-trip propagation delay. We add a timeout event to the "Wait for ACK or NAK0" and "Wait for ACK or NAK1" states. If the timeout event occurs, the most recently transmitted packet is retransmitted. Let us see why this protocol will still work with the rdt2.1 receiver.

- Suppose the timeout is caused by a lost data packet, i.e., a packet on the sender-to-receiver channel. In this case, the receiver never received the previous transmission and, from the receiver's viewpoint, if the timeout retransmission is received, it looks *exactly* the same as if the original transmission is being received.
- Suppose now that 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. Thus the sender's reaction is the same with a loss, as with a garbled ACK. The rdt 2.1 receiver can already handle the case of a garbled ACK.

Problem 11

If the sending of this message were removed, the sending and receiving sides would deadlock, waiting for an event that would never occur. Here's a scenario:

- Sender sends pkt0, enter the "Wait for ACK0 state", and waits for a packet back from the receiver
- Receiver is in the "Wait for 0 from below" 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.

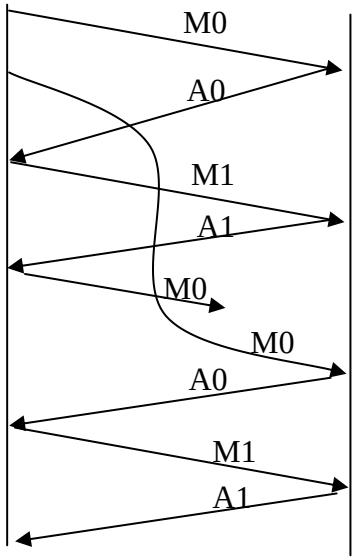
Now, the ender is awaiting an ACK of some sort from the receiver, and the receiver is waiting for a data packet form the sender – a deadlock!

Problem 12

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).

To get at the more subtle issue behind this question, one has to allow for premature timeouts to 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.

Problem 13



old version of M0 accepted!

Problem 14

In a NAK only protocol, the loss of packet *x* is only detected by the receiver when packet *x+1* is received. That is, the receivers receives *x-1* and then *x+1,* only when *x+1 is* received does the receiver realize that *x* was missed. If there is a long delay between the transmission of x and the transmission of *x+1,* then it will be a long time until *x* can be recovered, under a NAK only protocol.

On the other hand, if data is being sent often, then recovery under a NAK-only scheme could happen quickly. Moreover, if errors are infrequent, then NAKs are only occasionally sent (when needed), and ACK are never sent – a significant reduction in feedback in the NAK-only case over the ACK-only case.

Problem 15

It takes 12 microseconds (or 0.012 milliseconds) to send a packet, as $1500*8/10^9=12$ microseconds. In order for the sender to be busy 98 percent of the time, we must have

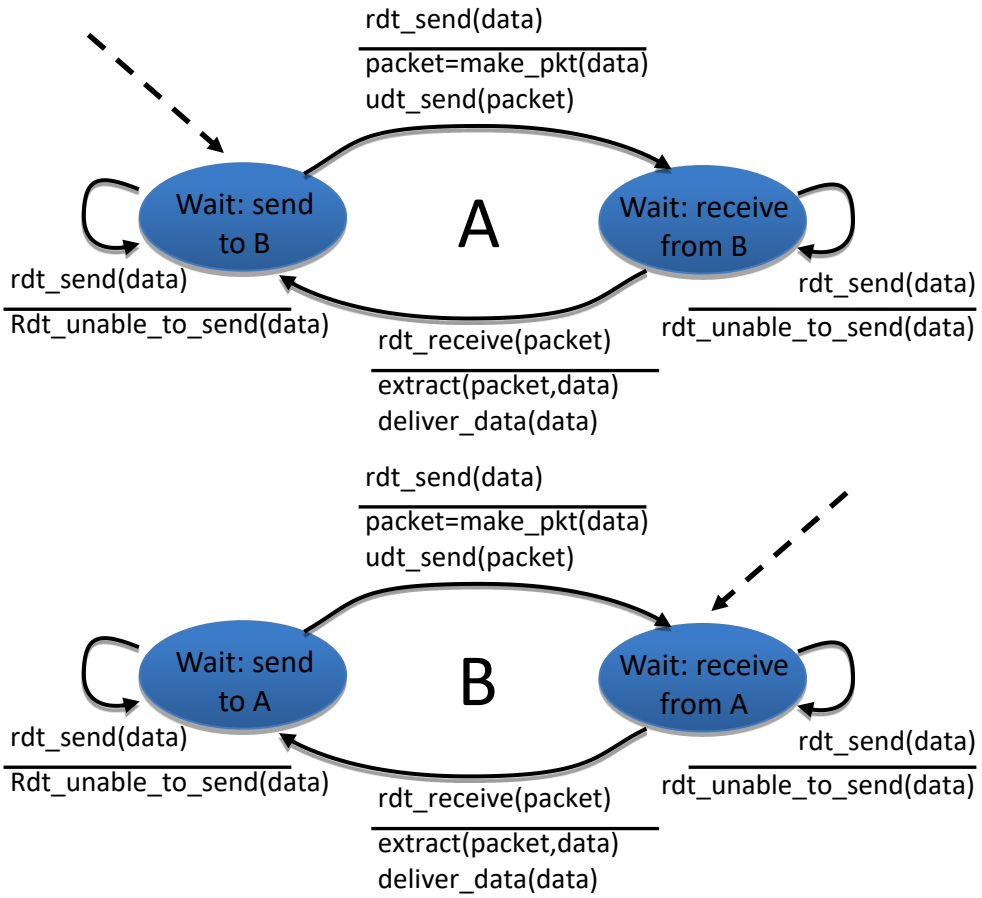
$$util = 0.98 = (0.012n)/30.012$$

or n approximately 2451 packets.

Problem 16

Yes. This actually causes the sender to send a number of pipelined data into the channel.
Yes. 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, unless there are some additional mechanism in the application to recover from loss.

Problem 17



Problem 18

In our solution, the sender will wait until it receives an ACK for a pair of messages (seqnum and seqnum+1) before moving on to the next pair of messages. Data packets have a data field and carry a two-bit sequence number. That is, the valid sequence numbers are 0, 1, 2, and 3. (Note: you should think about why a 1-bit sequence number space of 0, 1 only would not work in the solution below.) ACK messages carry the sequence number of the data packet they are acknowledging.

The FSM for the sender and receiver are shown in Figure 2. Note that the sender state records whether (i) no ACKs have been received for the current pair, (ii) an ACK for seqnum (only) has been received, or an ACK for seqnum+1 (only) has been received. In this figure, we assume that the seqnum is initially 0, and that the sender has sent the first

two data messages (to get things going). A timeline trace for the sender and receiver recovering from a lost packet is shown below:

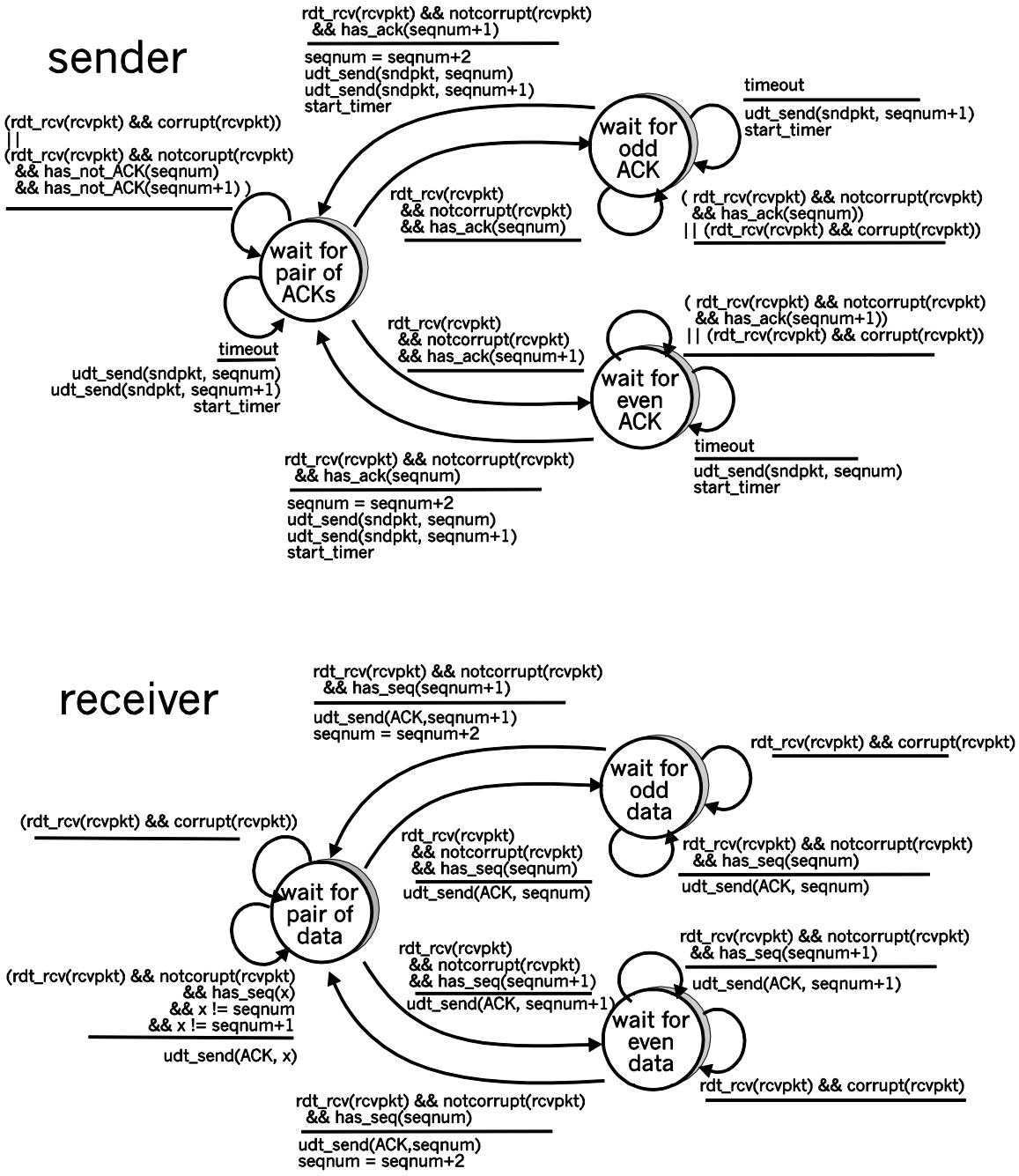


Figure 2: Sender and receiver for Problem (3.18)

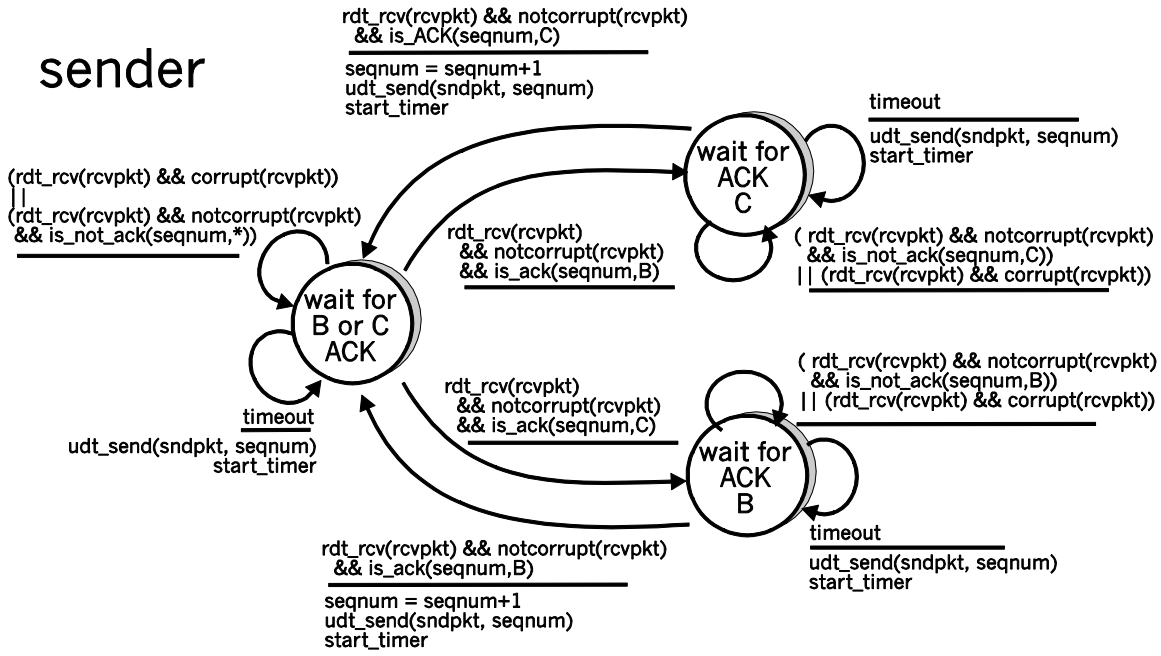
```
Sender                                  Receiver

make pair (0,1)
send packet 0
```

```
Packet 0 drops
send packet 1
                                        receive packet 1
                                        buffer packet 1
                                        send ACK 1
receive ACK 1
(timeout)
resend packet 0
                                        receive packet 0
                                        deliver pair (0,1)
                                        send ACK 0
receive ACK 0
```

Problem 19

This problem is a variation on the simple stop and wait protocol (rdt3.0). Because the channel may lose messages and because the sender may resend a message that one of the receivers has already received (either because of a premature timeout or because the other receiver has yet to receive the data correctly), sequence numbers are needed. As in rdt3.0, a 0-bit sequence number will suffice here.

The sender and receiver FSM are shown in Figure 3. In this problem, the sender state indicates whether the sender has received an ACK from B (only), from C (only) or from neither C nor B. The receiver state indicates which sequence number the receiver is waiting for.



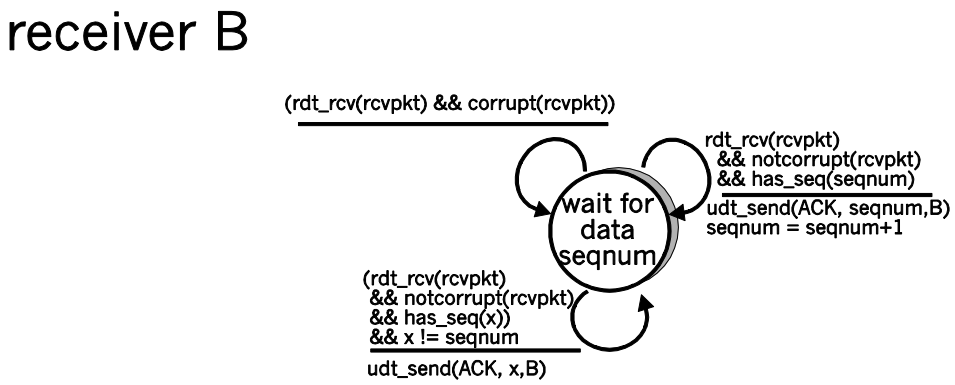


Figure 3. Sender and receiver for Problem 3.19(Problem 19)

Problem 20

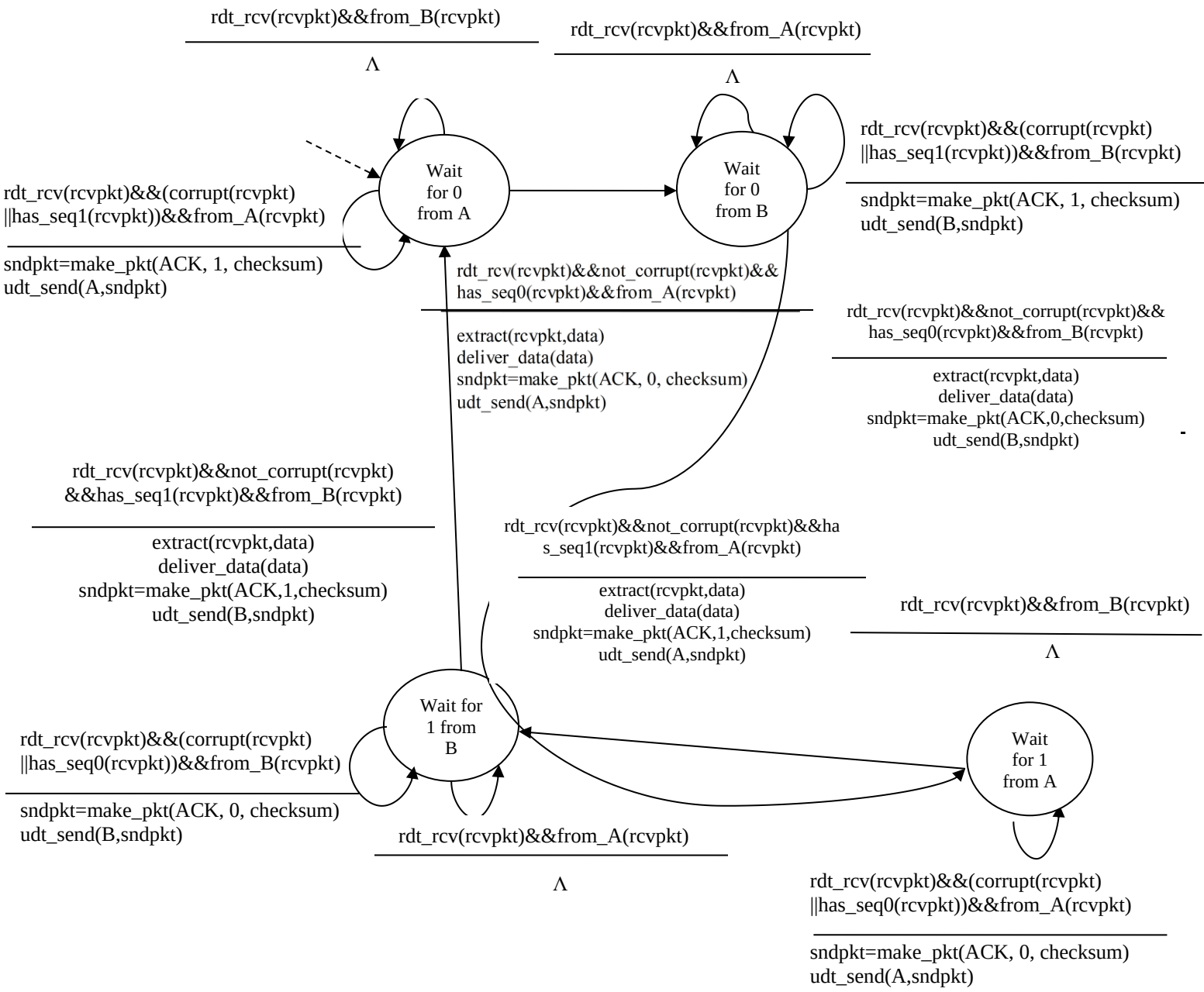


Figure 4: Receiver side FSM for 3.18

Sender
The sender side FSM is exactly same as given in Figure 3.15 in text

Problem 21

Because the A-to-B channel can lose request messages, A will need to timeout and retransmit its request messages (to be able to recover from loss). Because the channel delays are variable and unknown, it is possible that A will send duplicate requests (i.e., resend a request message that has already been received by B). To be able to detect duplicate request messages, the protocol will use sequence numbers. A 1-bit sequence number will suffice for a stop-and-wait type of request/response protocol.
A (the requestor) has 4 states:

- **"Wait for Request 0 from above."** Here the requestor is waiting for a call from above to request a unit of data. When it receives a request from above, it sends a request message, R0, to B, starts a timer and makes a transition to the "Wait for D0" state. When in the "Wait for Request 0 from above" state, A ignores anything it receives from B.

- **"Wait for D0"**. Here the requestor is waiting for a D0 data message from B. A timer is always running in this state. If the timer expires, A sends another R0 message, restarts the timer and remains in this state. If a D0 message is received from B, A stops the time and transits to the "Wait for Request 1 from above" state. If A receives a D1 data message while in this state, it is ignored.

- **"Wait for Request 1 from above."** Here the requestor is again waiting for a call from above to request a unit of data. When it receives a request from above, it sends a request message, R1, to B, starts a timer and makes a transition to the "Wait for D1" state. When in the "Wait for Request 1 from above" state, A ignores anything it receives from B.

- **"Wait for D1".** Here the requestor is waiting for a D1 data message from B. A timer is always running in this state. If the timer expires, A sends another R1 message, restarts the timer and remains in this state. If a D1 message is received from B, A stops the timer and transits to the "Wait for Request 0 from above" state. If A receives a D0 data message while in this state, it is ignored.

The data supplier (B) has only two states:

- **"Send D0."** In this state, B continues to respond to received R0 messages by sending D0, and then remaining in this state. If B receives a R1 message, then it knows its D0 message has been received correctly. It thus discards this D0 data (since it has been received at the other side) and then transits to the "Send D1" state, where it will use D1 to send the next requested piece of data.

- **"Send D1."** In this state, B continues to respond to received R1 messages by sending D1, and then remaining in this state. If B receives a R1 message, then it knows its D1 message has been received correctly and thus transits to the "Send D1" state.

Problem 22

a) Here we have a window size of N=3. Suppose the receiver has received packet k-1, and has ACKed that and all other preceding packets. If all of these ACK's have been received by sender, then sender's window is [k, k+N-1]. Suppose next that none of the ACKs have been received at the sender. In this second case, the sender's window contains k-1 and the N packets up to and including k-1. The sender's window is thus [k-N,k-1]. By these arguments, the senders window is of size 3 and begins somewhere in the range [k-N,k].

b) If the receiver is waiting for packet k, then it has received (and ACKed) packet k-1 and the N-1 packets before that. If none of those N ACKs have been yet received by the sender, then ACK messages with values of [k-N,k-1] may still be propagating back.Because the sender has sent packets [k-N, k-1], it must be the case that the sender has already received an ACK for k-N-1. Once the receiver has sent an ACK for k-N-1 it will never send an ACK that is less that k-N-1. Thus the range of in-flight ACK values can range from k-N-1 to k-1.

Problem 23

In order to avoid the scenario of Figure 3.27, we want to avoid having the leading edge of the receiver's window (i.e., the one with the "highest" sequence number) wrap around in the sequence number space and overlap with the trailing edge (the one with the "lowest" sequence number in the sender's window). That is, the sequence number space must be large enough to fit the entire receiver window and the entire sender window without this overlap condition. So - we need to determine how large a range of sequence numbers can be covered at any given time by the receiver and sender windows.

Suppose that the lowest-sequence number that the receiver is waiting for is packet m. In this case, it's window is [m,m+w-1] and it has received (and ACKed) packet m-1 and the w-1 packets before that, where w is the size of the window. If none of those w ACKs have been yet received by the sender, then ACK messages with values of [m-w,m-1] may still be propagating back. If no ACKs with these ACK numbers have been received by the sender, then the sender's window would be [m-w,m-1].

Thus, the lower edge of the sender's window is m-w, and the leading edge of the receivers window is m+w-1. In order for the leading edge of the receiver's window to not overlap with the trailing edge of the sender's window, the sequence number space must

thus be big enough to accommodate 2w sequence numbers. That is, the sequence number space must be at least twice as large as the window size, $k \geq 2w$.

Problem 24

a) True. Suppose the sender has a window size of 3 and sends packets 1, 2, 3 at $t0$. At $t1$ ($t1 > t0$) the receiver ACKS 1, 2, 3. At $t2$ ($t2 > t1$) the sender times out and resends 1, 2, 3. At $t3$ the receiver receives the duplicates and re-acknowledges 1, 2, 3. At $t4$ the sender receives the ACKs that the receiver sent at $t1$ and advances its window to 4, 5, 6. At $t5$ the sender receives the ACKs 1, 2, 3 the receiver sent at $t2$. These ACKs are outside its window.

b) True. By essentially the same scenario as in (a).

c) True.

d) True. Note that with a window size of 1, SR, GBN, and the alternating bit protocol are functionally equivalent. The window size of 1 precludes the possibility of out-of-order packets (within the window). A cumulative ACK is just an ordinary ACK in this situation, since it can only refer to the single packet within the window.

Problem 25

a) Consider sending an application message over a transport protocol. With TCP, the application writes data to the connection send buffer and TCP will grab bytes without necessarily putting a single message in the TCP segment; TCP may put more or less than a single message in a segment. UDP, on the other hand, encapsulates in a segment whatever the application gives it; so that, if the application gives UDP an application message, this message will be the payload of the UDP segment. Thus, with UDP, an application has more control of what data is sent in a segment.

b) With TCP, due to flow control and congestion control, there may be significant delay from the time when an application writes data to its send buffer until when the data is given to the network layer. UDP does not have delays due to flow control and congestion control.

Problem 26

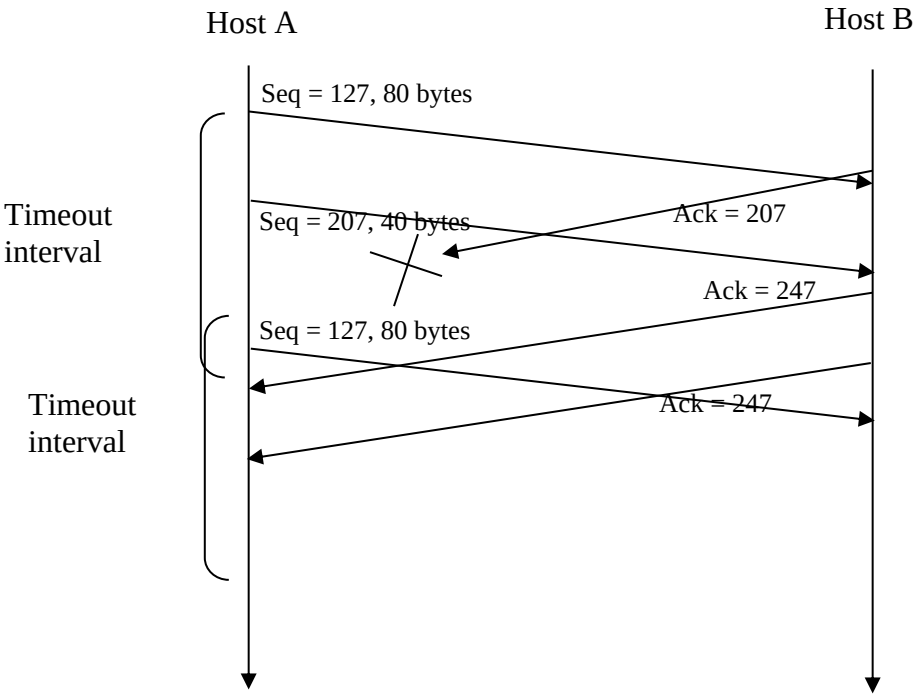
There are $2^{32} = 4{,}294{,}967{,}296$ possible sequence numbers.

a) The sequence number does not increment by one with each segment. Rather, it increments by the number of bytes of data sent. So the size of the MSS is irrelevant -- the maximum size file that can be sent from A to B is simply the number of bytes representable by $2^{32} \approx 4.19\,\text{Gbytes}$.

b) The number of segments is $\left\lceil \frac{2^{32}}{536} \right\rceil = 8{,}012{,}999$. 66 bytes of header get added to each segment giving a total of 528,857,934 bytes of header. The total number of bytes transmitted is $2^{32} + 528{,}857{,}934 = 4.824 \times 10^9$ bytes.
Thus it would take 249 seconds to transmit the file over a 155~Mbps link.

Problem 27

a) In the second segment from Host A to B, the sequence number is 207, source port number is 302 and destination port number is 80.
b) If the first segment arrives before the second, in the acknowledgement of the first arriving segment, the acknowledgement number is 207, the source port number is 80 and the destination port number is 302.
c) If the second segment arrives before the first segment, in the acknowledgement of the first arriving segment, the acknowledgement number is 127, indicating that it is still waiting for bytes 127 and onwards.
d)



Problem 28

Since the link capacity is only 100 Mbps, so host A's sending rate can be at most 100Mbps. Still, host A sends data into the receive buffer faster than Host B can remove data from the buffer. The receive buffer fills up at a rate of roughly 40Mbps. When the buffer is full, Host B signals to Host A to stop sending data by setting RcvWindow = 0. Host A then stops sending until it receives a TCP segment with RcvWindow > 0. Host A will thus repeatedly stop and start sending as a function of the RcvWindow values it

receives from Host B. On average, the long-term rate at which Host A sends data to Host B as part of this connection is no more than 60Mbps.

Problem 29

a) The server uses special initial sequence number (that is obtained from the hash of source and destination IPs and ports) in order to defend itself against SYN FLOOD attack.

b) No, the attacker cannot create half-open or fully open connections by simply sending and ACK packet to the target. Half-open connections are not possible since a server using SYN cookies does not maintain connection variables and buffers for any connection before full connections are established. For establishing fully open connections, an attacker should know the special initial sequence number corresponding to the (spoofed) source IP address from the attacker. This sequence number requires the "secret" number that each server uses. Since the attacker does not know this secret number, she cannot guess the initial sequence number.

c) No, the sever can simply add in a time stamp in computing those initial sequence numbers and choose a time to live value for those sequence numbers, and discard expired initial sequence numbers even if the attacker replay them.

Problem 30

a) If timeout values are fixed, then the senders may timeout prematurely. Thus, some packets are re-transmitted even they are not lost.

b) If timeout values are estimated (like what TCP does), then increasing the buffer size certainly helps to increase the throughput of that router. But there might be one potential problem. Queuing delay might be very large, similar to what is shown in Scenario 1.

Problem 31

DevRTT = (1- beta) * DevRTT + beta * | SampleRTT - EstimatedRTT |
EstimatedRTT = (1-alpha) * EstimatedRTT + alpha * SampleRTT
TimeoutInterval = EstimatedRTT + 4 * DevRTT

After obtaining first SampleRTT 106ms:
DevRTT = 0.75*5 + 0.25 * | 106 - 100 | = 5.25ms
EstimatedRTT = 0.875 * 100 + 0.125 * 106 = 100.75 ms
TimeoutInterval = 100.75+4*5.25 = 121.75 ms

After obtaining 120ms:
DevRTT = 0.75*5.25 + 0.25 * | 120 – 100.75 | = 8.75 ms
EstimatedRTT = 0.875 * 100.75 + 0.125 * 120 = 103.16 ms

TimeoutInterval = 103.16+4*8.75 = 138.16 ms

After obtaining 140ms:
DevRTT = 0.75*8.75 + 0.25 * | 140 – 103.16 | = 15.77 ms
EstimatedRTT = 0.875 * 103.16 + 0.125 * 140 = 107.76 ms
TimeoutInterval = 107.76+4*15.77 = 170.84 ms

After obtaining 90ms:
DevRTT = 0.75*15.77 + 0.25 * | 90 – 107.76 | = 16.27 ms
EstimatedRTT = 0.875 * 107.76 + 0.125 * 90 = 105.54 ms
TimeoutInterval = 105.54+4*16.27 =170.62 ms

After obtaining 115ms:
DevRTT = 0.75*16.27 + 0.25 * | 115 – 105.54 | = 14.57 ms
EstimatedRTT = 0.875 * 105.54 + 0.125 * 115 = 106.72 ms
TimeoutInterval = 106.72+4*14.57 =165 ms

Problem 32

a)
Denote $EstimatedRTT^{(n)}$ for the estimate after the *n*th sample.

$$\begin{aligned} EstimatedRTT^{(4)} &= xSampleRTT_1 + \\ &\quad (1-x)[xSampleRTT_2 + \\ &\quad (1-x)[xSampleRTT_3 + (1-x)SampleRTT_4]] \\ &= xSampleRTT_1 + (1-x)xSampleRTT_2 \\ &\quad + (1-x)^2 xSampleRTT_3 + (1-x)^3 SampleRTT_4 \end{aligned}$$

b)

$$\begin{aligned} EstimatedRTT^{(n)} &= x\sum_{j=1}^{n-1}(1-x)^{j-1}SampleRTT_j \\ &\quad + (1-x)^{n-1} SampleRTT_n \end{aligned}$$

c)

$$\begin{aligned} EstimatedRTT^{(\infty)} &= \frac{x}{1-x}\sum_{j=1}^{\infty}(1-x)^j SampleRTT_j \\ &= \frac{1}{9}\sum_{j=1}^{\infty}.9^j SampleRTT_j \end{aligned}$$

The weight given to past samples decays exponentially.

Problem 33

Let's look at what could wrong if TCP measures `SampleRTT` for a retransmitted segment. Suppose the source sends packet P1, the timer for P1 expires, and the source then sends P2, a new copy of the same packet. Further suppose the source measures `SampleRTT` for P2 (the retransmitted packet). Finally suppose that shortly after transmitting P2 an acknowledgment for P1 arrives. The source will mistakenly take this acknowledgment as an acknowledgment for P2 and calculate an incorrect value of `SampleRTT`.

Let's look at what could be wrong if TCP measures `SampleRTT` for a retransmitted segment. Suppose the source sends packet P1, the timer for P1 expires, and the source then sends P2, a new copy of the same packet. Further suppose the source measures `SampleRTT` for P2 (the retransmitted packet). Finally suppose that shortly after transmitting P2 an acknowledgment for P1 arrives. The source will mistakenly take this acknowledgment as an acknowledgment for P2 and calculate an incorrect value of `SampleRTT`.

Problem 34

At any given time *t*, `SendBase − 1` is the sequence number of the last byte that the sender knows has been received correctly, and in order, at the receiver. The actually last byte received (correctly and in order) at the receiver at time *t* may be greater if there are acknowledgements in the pipe. Thus

$$\texttt{SendBase-1} \leq \texttt{LastByteRcvd}$$

Problem 35

When, at time *t*, the sender receives an acknowledgement with value *y*, the sender knows for sure that the receiver has received everything up through *y*-1. The actual last byte received (correctly and in order) at the receiver at time *t* may be greater if $y \leq$ `SendBase` or if there are other acknowledgements in the pipe. Thus

$$\texttt{y-1} \leq \texttt{LastByteRvcd}$$

Problem 36

Suppose packets n, n+1, and n+2 are sent, and that packet n is received and ACKed. If packets n+1 and n+2 are reordered along the end-to-end-path (i.e., are received in the order n+2, n+1) then the receipt of packet n+2 will generate a duplicate ack for n and would trigger a retransmission under a policy of waiting only for second duplicate ACK for retransmission. By waiting for a triple duplicate ACK, it must be the case that *two*

packets after packet n are correctly received, while n+1 was not received. The designers of the triple duplicate ACK scheme probably felt that waiting for two packets (rather than 1) was the right tradeoff between triggering a quick retransmission when needed, but not retransmitting prematurely in the face of packet reordering.

Problem 37

a) GoBackN:
A sends 9 segments in total. They are initially sent segments 1, 2, 3, 4, 5 and later re-sent segments 2, 3, 4, and 5.
B sends 8 ACKs. They are 4 ACKS with sequence number 1, and 4 ACKS with sequence numbers 2, 3, 4, and 5.

Selective Repeat:
A sends 6 segments in total. They are initially sent segments 1, 2, 3, 4, 5 and later re-sent segments 2.
B sends 5 ACKs. They are 4 ACKS with sequence number 1, 3, 4, 5. And there is one ACK with sequence number 2.

TCP:
A sends 6 segments in total. They are initially sent segments 1, 2, 3, 4, 5 and later re-sent segments 2.
B sends 5 ACKs. They are 4 ACKS with sequence number 2. There is one ACK with sequence numbers 6. Note that TCP always send an ACK with expected sequence number.

b) TCP. This is because TCP uses fast retransmit without waiting until time out.

Problem 38

Yes, the sending rate is always roughly cwnd/RTT.

Problem 39

If the arrival rate increases beyond R/2 in Figure 3.46(b), then the total arrival rate to the queue exceeds the queue's capacity, resulting in increasing loss as the arrival rate increases. When the arrival rate equals R/2, 1 out of every three packets that leaves the queue is a retransmission. With increased loss, even a larger fraction of the packets leaving the queue will be retransmissions. Given that the maximum departure rate from the queue for one of the sessions is R/2, and given that a third or more will be transmissions as the arrival rate increases, the throughput of successfully deliver data can not increase beyond λ_{out}. Following similar reasoning, if half of the packets leaving the queue are retransmissions, and the maximum rate of output packets per session is R/2, then the maximum value of λ_{out} is (R/2)/2 or R/4.

Problem 40

a) TCP slowstart is operating in the intervals [1,6] and [23,26]
b) TCP congestion avoidance is operating in the intervals [6,16] and [17,22]
c) After the 16th transmission round, packet loss is recognized by a triple duplicate ACK. If there was a timeout, the congestion window size would have dropped to 1.
d) After the 22nd transmission round, segment loss is detected due to timeout, and hence the congestion window size is set to 1.
e) The threshold is initially 32, since it is at this window size that slow start stops and congestion avoidance begins.
f) The threshold is set to half the value of the congestion window when packet loss is detected. When loss is detected during transmission round 16, the congestion windows size is 42. Hence the threshold is 21 during the 18th transmission round.
g) The threshold is set to half the value of the congestion window when packet loss is detected. When loss is detected during transmission round 22, the congestion windows size is 29. Hence the threshold is 14 (taking lower floor of 14.5) during the 24th transmission round.
h) During the 1st transmission round, packet 1 is sent; packet 2-3 are sent in the 2nd transmission round; packets 4-7 are sent in the 3rd transmission round; packets 8-15 are sent in the 4th transmission round; packets 16-31 are sent in the 5th transmission round; packets 32-63 are sent in the 6th transmission round; packets 64 – 96 are sent in the 7th transmission round. Thus packet 70 is sent in the 7th transmission round.
i) The threshold will be set to half the current value of the congestion window (8) when the loss occurred and congestion window will be set to the new threshold value + 3 MSS . Thus the new values of the threshold and window will be 4 and 7 respectively.
j) threshold is 21, and congestion window size is 1.
k) round 17, 1 packet; round 18, 2 packets; round 19, 4 packets; round 20, 8 packets; round 21, 16 packets; round 22, 21 packets. So, the total number is 52.

Problem 41

Refer to Figure 5. In Figure 5(a), the ratio of the linear decrease on loss between connection 1 and connection 2 is the same - as ratio of the linear increases: unity. In this case, the throughputs never move off of the AB line segment. In Figure 5(b), the ratio of the linear decrease on loss between connection 1 and connection 2 is 2:1. That is, whenever there is a loss, connection 1 decreases its window by twice the amount of connection 2. We see that eventually, after enough losses, and subsequent increases, that connection 1's throughput will go to 0, and the full link bandwidth will be allocated to connection 2.

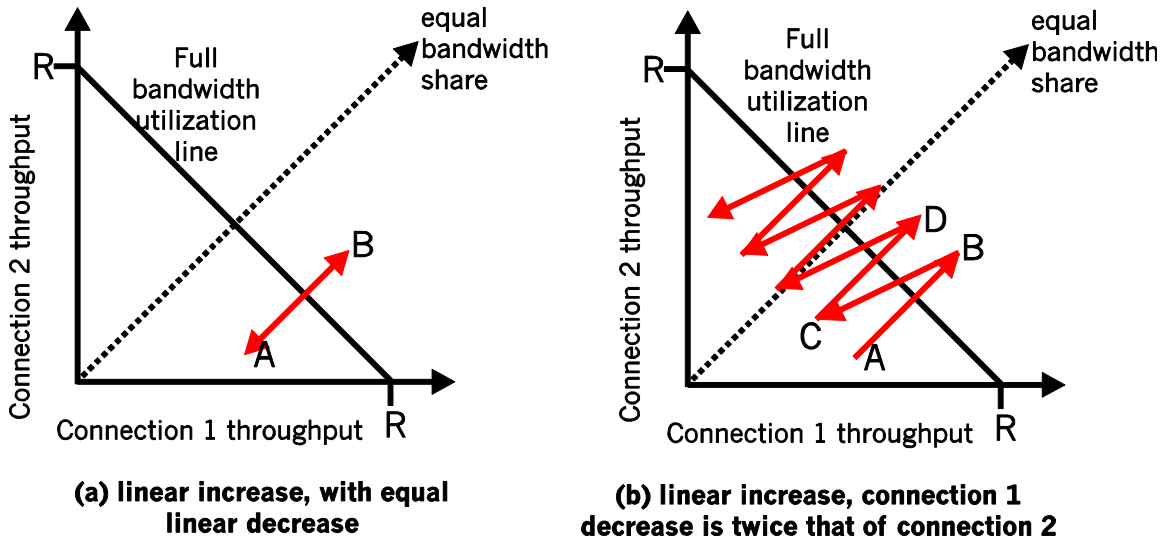


Figure 5: Lack of TCP convergence with linear increase, linear decrease

Problem 42

If TCP were a stop-and-wait protocol, then the doubling of the time out interval would suffice as a congestion control mechanism. However, TCP uses pipelining (and is therefore not a stop-and-wait protocol), which allows the sender to have multiple outstanding unacknowledged segments. The doubling of the timeout interval does not prevent a TCP sender from sending a large number of first-time-transmitted packets into the network, even when the end-to-end path is highly congested. Therefore a congestion-control mechanism is needed to stem the flow of "data received from the application above" when there are signs of network congestion.

Problem 43

In this problem, there is no danger in overflowing the receiver since the receiver's receive buffer can hold the entire file. Also, because there is no loss and acknowledgements are returned before timers expire, TCP congestion control does not throttle the sender. However, the process in host A will not continuously pass data to the socket because the send buffer will quickly fill up. Once the send buffer becomes full, the process will pass data at an average rate or R << S.

Problem 44

a) It takes 1 RTT to increase CongWin to 7 MSS; 2 RTTs to increase to 8 MSS; 3 RTTs to increase to 9 MSS; 4 RTTs to increase to 10 MSS; 5 RTTs to increase to 11 MSS; 6 RTTs to increase to 12 MSS.

b) In the first RTT 6 MSS was sent; in the second RTT 7 MSS was sent; in the third RTT 8 MSS was sent; in the fourth RTT 9 MSS was sent; in the fifth RTT, 10 MSS was sent; and in the sixth RTT, 11 MSS was sent. Thus, up to time 6 RTT,

6+7+8+9+10+11 = 51 MSS were sent. Thus, we can say that the average throughput up to time 6 RTT was (51 MSS)/(6 RTT) = 8.5 MSS/RTT.

Problem 45

a) The loss rate, L, is the ratio of the number of packets lost over the number of packets sent. In a cycle, 1 packet is lost. The number of packets sent in a cycle is

$$\frac{W}{2}+\left(\frac{W}{2}+1\right)+\cdots+W=\sum_{n=0}^{W/2}(\frac{W}{2}+n)$$

$$=\left(\frac{W}{2}+1\right)\frac{W}{2}+\sum_{n=0}^{W/2}n$$

$$=\left(\frac{W}{2}+1\right)\frac{W}{2}+\frac{W/2(W/2+1)}{2}$$

$$=\frac{W^2}{4}+\frac{W}{2}+\frac{W^2}{8}+\frac{W}{4}$$

$$=\frac{3}{8}W^2+\frac{3}{4}W$$

Thus the loss rate is

$$L=\frac{1}{\frac{3}{8}W^2+\frac{3}{4}W}$$

b) For W large, $\frac{3}{8}W^2>>\frac{3}{4}W$. Thus $L\approx 8/3W^2$ or $W\approx\sqrt{\frac{8}{3L}}$. From the text, we therefore have

$$\text{average throughput}=\frac{3}{4}\sqrt{\frac{8}{3L}}\cdot\frac{MSS}{RTT}$$

$$=\frac{1.22\cdot MSS}{RTT\cdot\sqrt{L}}$$

Problem 46

a) Let W denote the max window size measured in segments. Then, W*MSS/RTT = 10Mbps, as packets will be dropped if the maximum sending rate exceeds link capacity. Thus, we have W*1500*8/0.15=10*10^6, then W is about 125 segments.

b) As congestion window size varies from W/2 to W, then the average window size is 0.75W=94 (ceiling of 93.75) segments. Average throughput is 94*1500*8/0.15 =7.52Mbps.
c) When there is a packet loss, W becomes W/2, i.e., 125/2=62.
(125 - 62) *0.15 = 9.45 seconds, as the number of RTTs (that this TCP connections needs in order to increase its window size from 62 to 125) is 63. Recall the window size increases by one in each RTT.

Problem 47

Let W denote max window size. Let S denote the buffer size. For simplicity, suppose TCP sender sends data packets in a round by round fashion, with each round corresponding to a RTT. If the window size reaches W, then a loss occurs. Then the sender will cut its congestion window size by half, and waits for the ACKs for W/2 outstanding packets before it starts sending data segments again. In order to make sure the link always busying sending data, we need to let the link busy sending data in the period $W/(2*C)$ (this is the time interval where the sender is waiting for the ACKs for the W/2 outstanding packets). Thus, S/C must be no less than $W/(2*C)$, that is, $S>=W/2$.

Let Tp denote the one-way propagation delay between the sender and the receiver.
When the window size reaches the minimum W/2 and the buffer is empty, we need to make sure the link is also busy sending data. Thus, we must have W/2/(2Tp)>=C, thus, $W/2>=C*2T$p.

Thus, $S>=C*2T$p.

Problem 48

a) Let W denote the max window size. Then, W*MSS/RTT = 10Gbps, as packets will be dropped if maximum sending rate reaches link capacity. Thus, we have W*1500*8/0.15=10*10^9, then W= 125000 segments.
b) As congestion window size varies from W/2 to W, then the average window size is 0.75W=93750 segments. Average throughput is 93750*1500*8/0.1=7.5Gbps.
c) 93750/2 *0.15 /60= 117 minutes. In order to speed up the window increase process, we can increase the window size by a much larger value, instead of increasing window size only by one in each RTT. Some protocols are proposed to solve this problem, such as ScalableTCP or HighSpeed TCP.

Problem 49

As TCP's average throughput B is given by $B = \frac{1.22 \cdot MSS}{RTT \cdot \sqrt{L}}$, so we know that,

$L= (1.22*MSS / (B*RTT))^2$
Since between two consecutive packet losses, there are $1/L$ packets sent by the TCP sender, thus, $T=(1/L)*MSS/B$. Thus, we find that $T=B*RTT^2/(1.22^2*MSS)$, that is, T is a function of B.

Problem 50

a) The key difference between C1 and C2 is that C1's RTT is only half of that of C2. Thus C1 adjusts its window size after 50 msec, but C2 adjusts its window size after 100 msec. Assume that whenever a loss event happens, C1 receives it after 50msec and C2 receives it after 100msec. We further have the following simplified model of TCP. After each RTT, a connection determines if it should increase window size or not. For C1, we compute the average total sending rate in the link in the previous 50 msec. If that rate exceeds the link capacity, then we assume that C1 detects loss and reduces its window size. But for C2, we compute the average total sending rate in the link in the previous 100msec. If that rate exceeds the link capacity, then we assume that C2 detects loss and reduces its window size. Note that it is possible that the average sending rate in last 50msec is higher than the link capacity, but the average sending rate in last 100msec is smaller than or equal to the link capacity, then in this case, we assume that C1 will experience loss event but C2 will not.

The following table describes the evolution of window sizes and sending rates based on the above assumptions.

	C1		C2	
Time (msec)	Window Size (num. of segments sent in next 50msec)	Average data sending rate (segments per second, =Window/0.05)	Window Size(num. of segments sent in next 100msec)	Average data sending rate (segments per second, =Window/0.1)
0	10	200 (in [0-50]msec]	10	100 (in [0-50]msec)
50	5 (decreases window size as the avg. total sending rate to the link in **last 50msec** is 300= 200+100)	100 (in [50-100]msec]		100 (in [50-100]msec)
100	2 (decreases window size as the avg. total sending rate to the link in **last 50msec** is 200= 100+100)	40	5 (decreases window size as the avg. total sending rate to the link in **last 100msec** is 250= (200+100)/2 +	50

			(100+100)/2)	
150	1 (decreases window size as the avg. total sending rate to the link in last 50msec is 90= (40+50)	20		50
200	1 (no further decrease, as window size is already 1)	20	2 (decreases window size as the avg. total sending rate to the link in **last 100msec** is 80= (40+20)/2 + (50+50)/2)	20
250	1 (no further decrease, as window size is already 1)	20		20
300	1 (no further decrease, as window size is already 1)	20	1 (decreases window size as the avg. total sending rate to the link in **last 100msec** is 40= (20+20)/2 + (20+20)/2)	10
350	2	40		10
400	1	20	1	10
450	2	40		10
500	1 (decreases window size as the avg. total sending rate to the	20	1	10

	link in last 50msec is 50= (40+10)			
550	2	40		10
600	1	20	1	10
650	2	40		10
700	1	20	1	10
750	2	40		10
800	1	20	1	10
850	2	40		10
900	1	20	1	10
950	2	40		10
1000	1	20	1	10

Based on the above table, we find that after 1000 msec, C1's and C2's window sizes are 1 segment each.

b) No. In the long run, C1's bandwidth share is roughly twice as that of C2's, because C1 has shorter RTT, only half of that of C2, so C1 can adjust its window size twice as fast as C2. If we look at the above table, we can see a cycle every 200msec, e.g. from 850msec to 1000msec, inclusive. Within a cycle, the sending rate of C1 is (40+20+40+20) = 120, which is thrice as large as the sending of C2 given by (10+10+10+10) = 40.

Problem 51

a) Similarly as in last problem, we can compute their window sizes over time in the following table. Both C1 and C2 have the same window size 2 after 2200msec.

	C1		C2	
Time (msec)	Window Size (num. of segments sent in next 100msec)	Data sending speed (segments per second, =Window/0.1)	Window Size(num. of segments sent in next 100msec)	Data sending speed (segments per second, =Window/0.1)
0	15	150 (in [0-100]msec]	10	100 (in [0-100]msec)
100	7	70	5	50
200	3	30	2	20
300	1	10	1	10
400	2	20	2	20
500	1	10	1	10
600	2	20	2	20
700	1	10	1	10
800	2	20	2	20
900	1	10	1	10
1000	2	20	2	20

1100	1	10	1	10
1200	2	20	2	20
1300	1	10	1	10
1400	2	20	2	20
1500	1	10	1	10
1600	2	20	2	20
1700	1	10	1	10
1800	2	20	2	20
1900	1	10	1	10
2000	2	20	2	20
2100	1	10	1	10
2200	2	20	2	20

b) Yes, this is due to the AIMD algorithm of TCP and that both connections have the same RTT.
c) Yes, this can be seen clearly from the above table. Their max window size is 2.
d) No, this synchronization won't help to improve link utilization, as these two connections act as a single connection oscillating between min and max window size. Thus, the link is not fully utilized (recall we assume this link has no buffer). One possible way to break the synchronization is to add a finite buffer to the link and randomly drop packets in the buffer before buffer overflow. This will cause different connections cut their window sizes at different times. There are many AQM (Active Queue Management) techniques to do that, such as RED (Random Early Detect), PI (Proportional and Integral AQM), AVQ (Adaptive Virtual Queue), and REM (Random Exponential Marking), etc.

Problem 52

Note that W represents the maximum window size.
First we can find the total number of segments sent out during the interval when TCP changes its window size from W/2 up to and include W. This is given by:
$S = W/2 + (W/2)*(1+\alpha) + (W/2)*(1+\alpha)^2 + (W/2)*(1+\alpha)^3 + \ldots + (W/2)*(1+\alpha)^k$
We find $k=\log_{(1+\alpha)}2$, then $S=W*(2\alpha+1)/(2\alpha)$.

Loss rate L is given by:
$L = 1/S = (2\alpha) / (W*(2\alpha+1))$.

The time that TCP takes to increase its window size from W/2 to W is given by:
$k*RTT = (\log_{(1+\alpha)}2) * RTT$,
which is clearly independent of TCP's average throughput.

Note, TCP's average throughput is given by:
$B=MSS * S/((k+1)*RTT) = MSS / (L*(k+1)*RTT)$.

Note that this is different from TCP which has average throughput: $B = \frac{1.22 \cdot MSS}{RTT \cdot \sqrt{L}}$, where the square root of L appears in the denominator.

Problem 53

Let's assume 1500-byte packets and a 100 ms round-trip time. From the TCP throughput equation $B = \frac{1.22 \cdot MSS}{RTT \cdot \sqrt{L}}$, we have

10 Gbps = 1.22 * (1500*8 bits) / (.1 sec * srqt(L)), or

sqrt(L) = 14640 bits / (10^9 bits) = 0.00001464, or

L = 2.14 * 10^(-10)

Problem 54

An advantage of using the earlier values of `cwnd` and `ssthresh` at t_2 is that TCP would not have to go through slow start and congestion avoidance to ramp up to the throughput value obtained at t_1. A disadvantage of using these values is that they may be no longer accurate. In particular, if the path has become more congested between t_1 and t_2, the sender will send a large window's worth of segments into an already (more) congested path.

Problem 55

a) The server will send its response to Y.

b) The server can be certain that the client is indeed at Y. If it were at some other address spoofing Y, the SYNACK would have been sent to the address Y, and the TCP in that host would not send the TCP ACK segment back. Even if the attacker were to send an appropriately timed TCP ACK segment, it would not know the correct server sequence number (since the server uses random initial sequence numbers.)

Problem 56

a) Referring to the figure below, we see that the total delay is

RTT + RTT + S/R + RTT + S/R + RTT + 12S/R = 4RTT + 14 S/R

b) Similarly, the delay in this case is:

RTT+RTT + S/R + RTT + S/R + RTT + S/R + RTT + 8S/R = 5RTT +11 S/R

c) Similarly, the delay in this case is:

RTT + RTT + S/R + RTT + 14 S/R = 3 RTT + 15 S/R