

Chapter 3 Review Questions

1.
 - a) Call this protocol Simple Transport Protocol (STP). At the sender side, STP accepts from the sending process a chunk of data not exceeding 1196 bytes, a destination host address, and a destination port number. STP adds a four-byte header to each chunk and puts the port number of the destination process in this header. STP then gives the destination host address and the resulting segment to the network layer. The network layer delivers the segment to STP at the destination host. STP then examines the port number in the segment, extracts the data from the segment, and passes the data to the process identified by the port number.
 - b) The segment now has two header fields: a source port field and destination port field. At the sender side, STP accepts a chunk of data not exceeding 1192 bytes, a destination host address, a source port number, and a destination port number. STP creates a segment which contains the application data, source port number, and destination port number. It then gives the segment and the destination host address to the network layer. After receiving the segment, STP at the receiving host gives the application process the application data and the source port number.
 - c) No, the transport layer does not have to do anything in the core; the transport layer “lives” in the end systems.
2.
 1. For sending a letter, the family member is required to give the delegate the letter itself, the address of the destination house, and the name of the recipient. The delegate clearly writes the recipient’s name on the top of the letter. The delegate then puts the letter in an envelope and writes the address of the destination house on the envelope. The delegate then gives the letter to the planet’s mail service. At the receiving side, the delegate receives the letter from the mail service, takes the letter out of the envelope, and takes note of the recipient name written at the top of the letter. The delegate then gives the letter to the family member with this name.
 2. No, the mail service does not have to open the envelope; it only examines the address on the envelope.
 3. Source port number y and destination port number x.
 4. An application developer may not want its application to use TCP’s congestion control, which can throttle the application’s sending rate at times of congestion. Often, designers of IP telephony and IP videoconference applications choose to run their applications over UDP because they want to avoid TCP’s congestion control. Also, some applications do not need the reliable data transfer provided by TCP.
 5. Since most firewalls are configured to block UDP traffic, using TCP for video and

voice traffic lets the traffic through the firewalls.

6. Yes. The application developer can put reliable data transfer into the application layer protocol. This would require a significant amount of work and debugging, however.
7. Yes, both segments will be directed to the same socket. For each received segment, at the socket interface, the operating system will provide the process with the IP addresses to determine the origins of the individual segments.
8. For each persistent connection, the Web server creates a separate “connection socket”. Each connection socket is identified with a four-tuple: (source IP address, source port number, destination IP address, destination port number). When host C receives an IP datagram, it examines these four fields in the datagram/segment to determine to which socket it should pass the payload of the TCP segment. Thus, the requests from A and B pass through different sockets. The identifier for both of these sockets has 80 for the destination port; however, the identifiers for these sockets have different values for source IP addresses. Unlike UDP, when the transport layer passes a TCP segment’s payload to the application process, it does not specify the source IP address, as this is implicitly specified by the socket identifier.
9. Sequence numbers are required for a receiver to find out whether an arriving packet contains new data or is a retransmission.
10. To handle losses in the channel. If the ACK for a transmitted packet is not received within the duration of the timer for the packet, the packet (or its ACK or NACK) is assumed to have been lost. Hence, the packet is retransmitted.
11. A timer would still be necessary in the protocol rdt 3.0. If the round trip time is known then the only advantage will be that, the sender knows for sure that either the packet or the ACK (or NACK) for the packet has been lost, as compared to the real scenario, where the ACK (or NACK) might still be on the way to the sender, after the timer expires. However, to detect the loss, for each packet, a timer of constant duration will still be necessary at the sender.
12.
 - a) The packet loss caused a time out after which all the five packets were retransmitted.
 - b) Loss of an ACK didn’t trigger any retransmission as Go-Back-N uses cumulative acknowledgements.
 - c) The sender was unable to send sixth packet as the send window size is fixed to 5.

13.

- a) When the packet was lost, the received four packets were buffered the receiver. After the timeout, sender retransmitted the lost packet and receiver delivered the buffered packets to application in correct order.
- b) Duplicate ACK was sent by the receiver for the lost ACK.
- c) The sender was unable to send sixth packet as the send window size is fixed to 5

When a packet was lost, GO-Back-N retransmitted all the packets whereas Selective Repeat retransmitted the lost packet only. In case of lost acknowledgement, selective repeat sent a duplicate ACK and as GO-Back-N used cumulative acknowledgment, so that duplicate ACK was unnecessary.

14. a) false b) false c) true d) false e) true f) false g) false

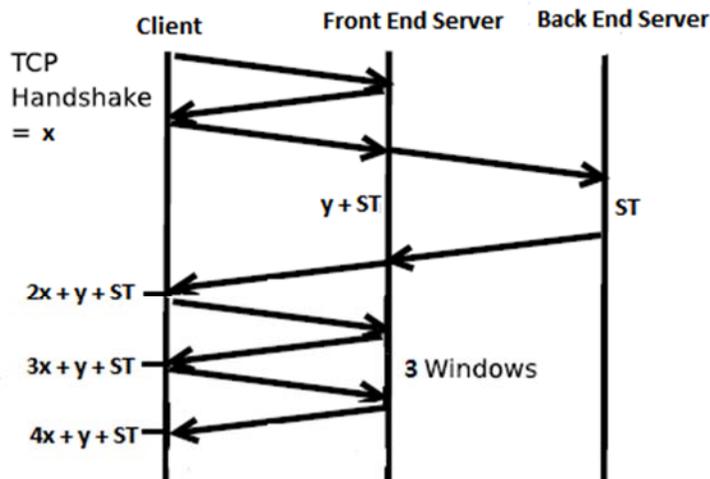
15. a) 20 bytes b) ack number = 90

16. 3 segments. First segment: seq = 43, ack = 80; Second segment: seq = 80, ack = 44; Third segment; seq = 44, ack = 81

17. R/2

18. False, it is set to half of the current value of the congestion window.

19. Let $X = RTT_{FE}$, $Y = RTT_{BE}$ and $ST =$ Search time. Consider the following timing diagram.



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 $4X + Y + ST = 4 \cdot RTT_{FE} + RTT_{BE} + \text{Search time}$

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.

$$\begin{array}{r}
 0\ 1\ 0\ 1\ 0\ 0\ 1\ 1 \\
 +\ 0\ 1\ 1\ 0\ 0\ 1\ 1\ 0 \\
 \hline
 1\ 0\ 1\ 1\ 1\ 0\ 0\ 1
 \end{array}$$

$$\begin{array}{r}
 1\ 0\ 1\ 1\ 1\ 0\ 0\ 1 \\
 +\ 0\ 1\ 1\ 1\ 0\ 1\ 0\ 0 \\
 \hline
 0\ 0\ 1\ 0\ 1\ 1\ 1\ 0
 \end{array}$$

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.

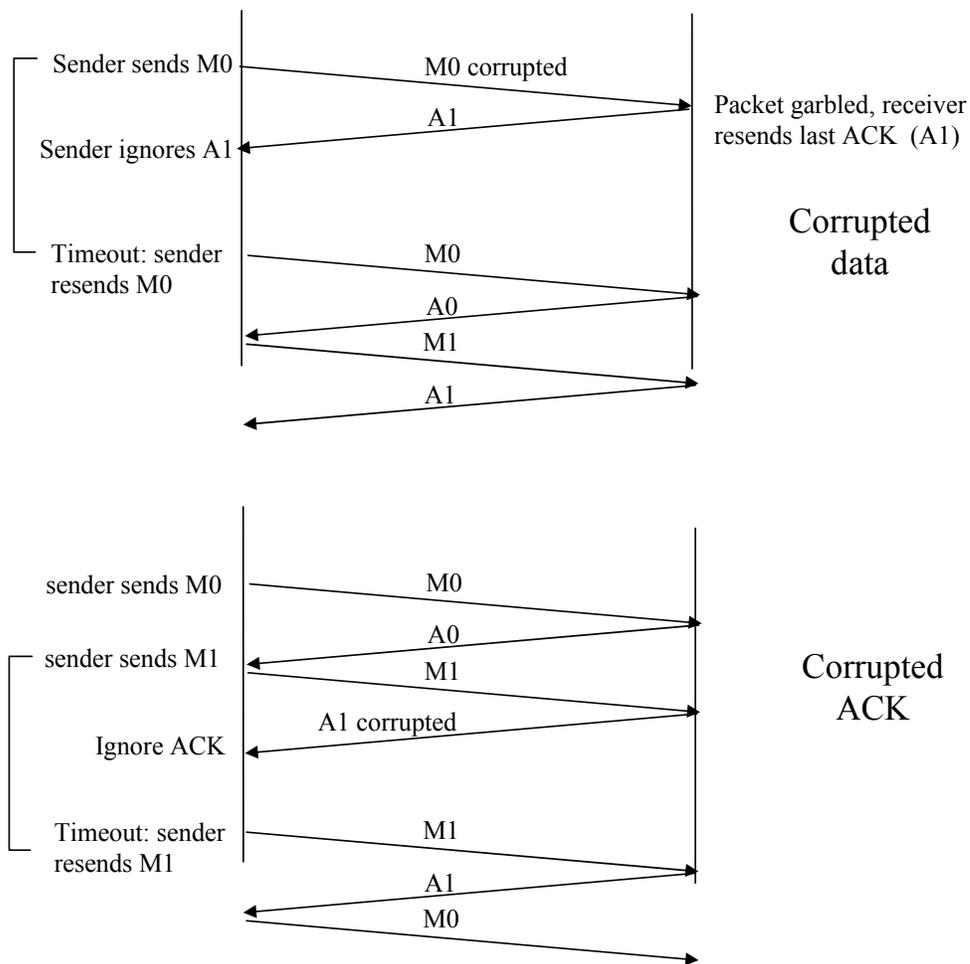


Figure 1: rdt 3.0 scenarios: corrupted data, corrupted ACK

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 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 receiver 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 $1500 \cdot 8 / 10^9 = 12$ microseconds. In order for the sender to be busy 98 percent of the time, we must have

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

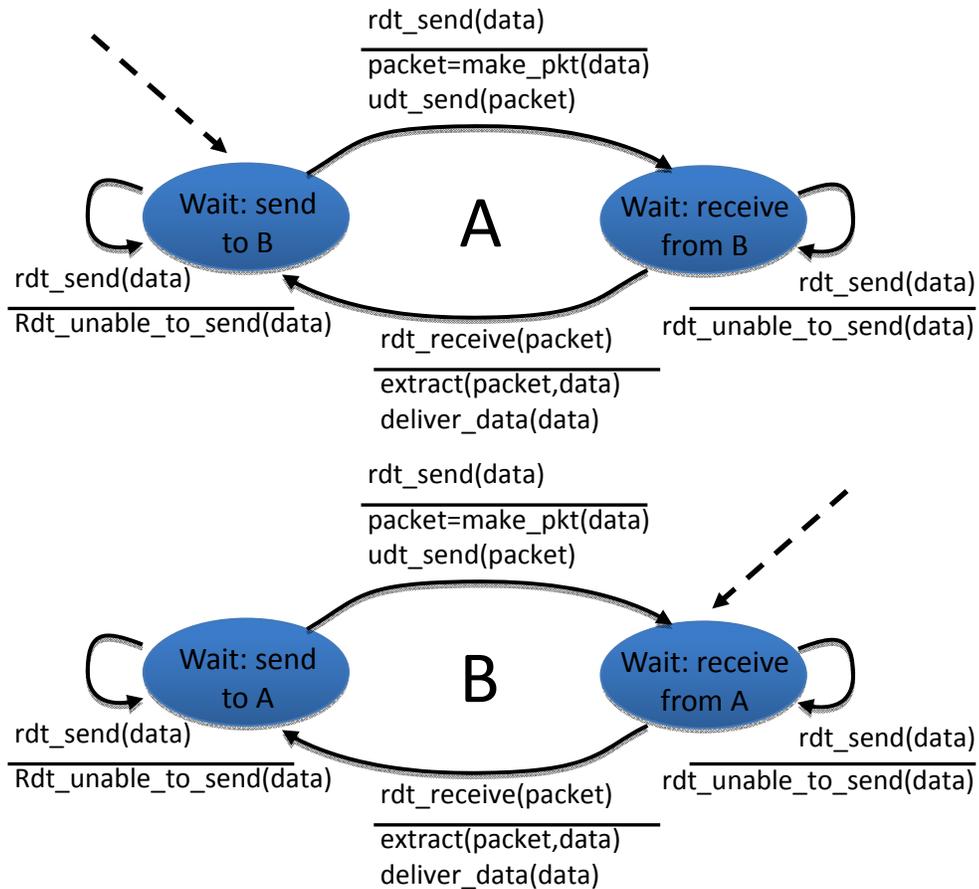
or n approximately 2451 packets.

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:

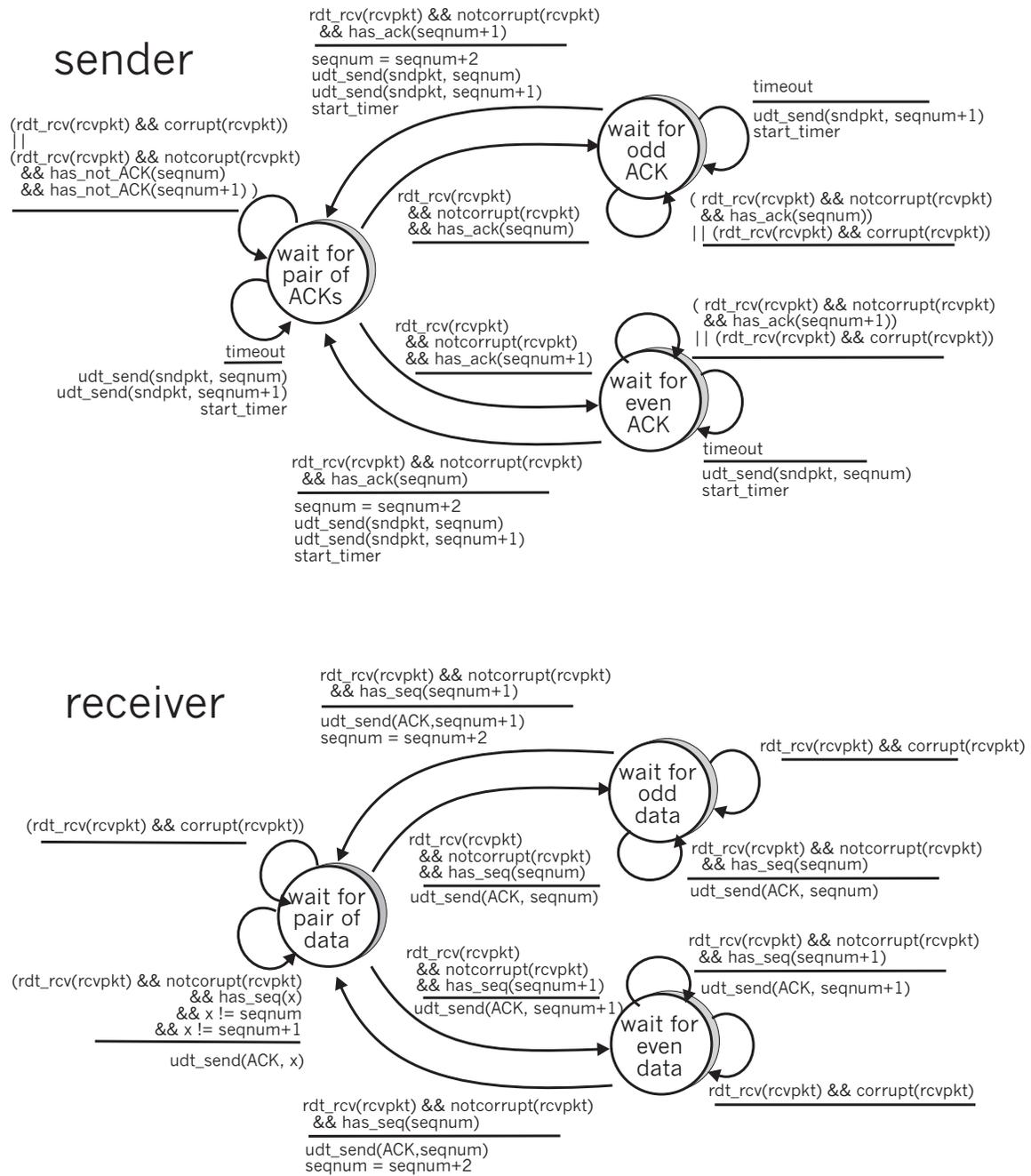


Figure 2: Sender and receiver for Problem (3.18)

Sender

Receiver

make pair (0,1)
send packet 0

Packet 0 drops
send packet 1

receive ACK 1
(timeout)
resend packet 0

receive ACK 0

receive packet 1
buffer packet 1
send ACK 1

receive packet 0
deliver pair (0,1)
send 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.

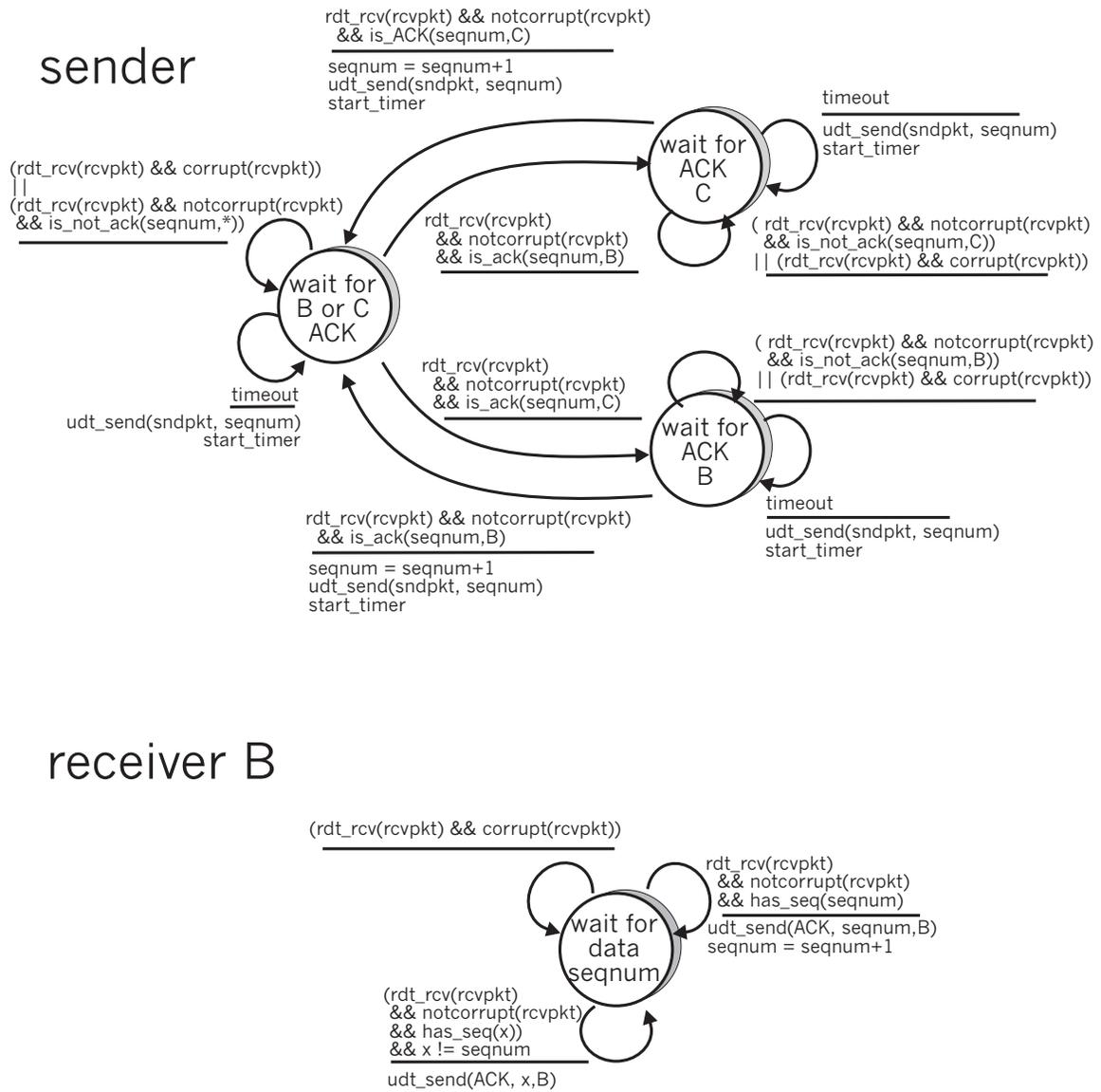


Figure 3. Sender and receiver for Problem 3.19(Problem 19)

Problem 20

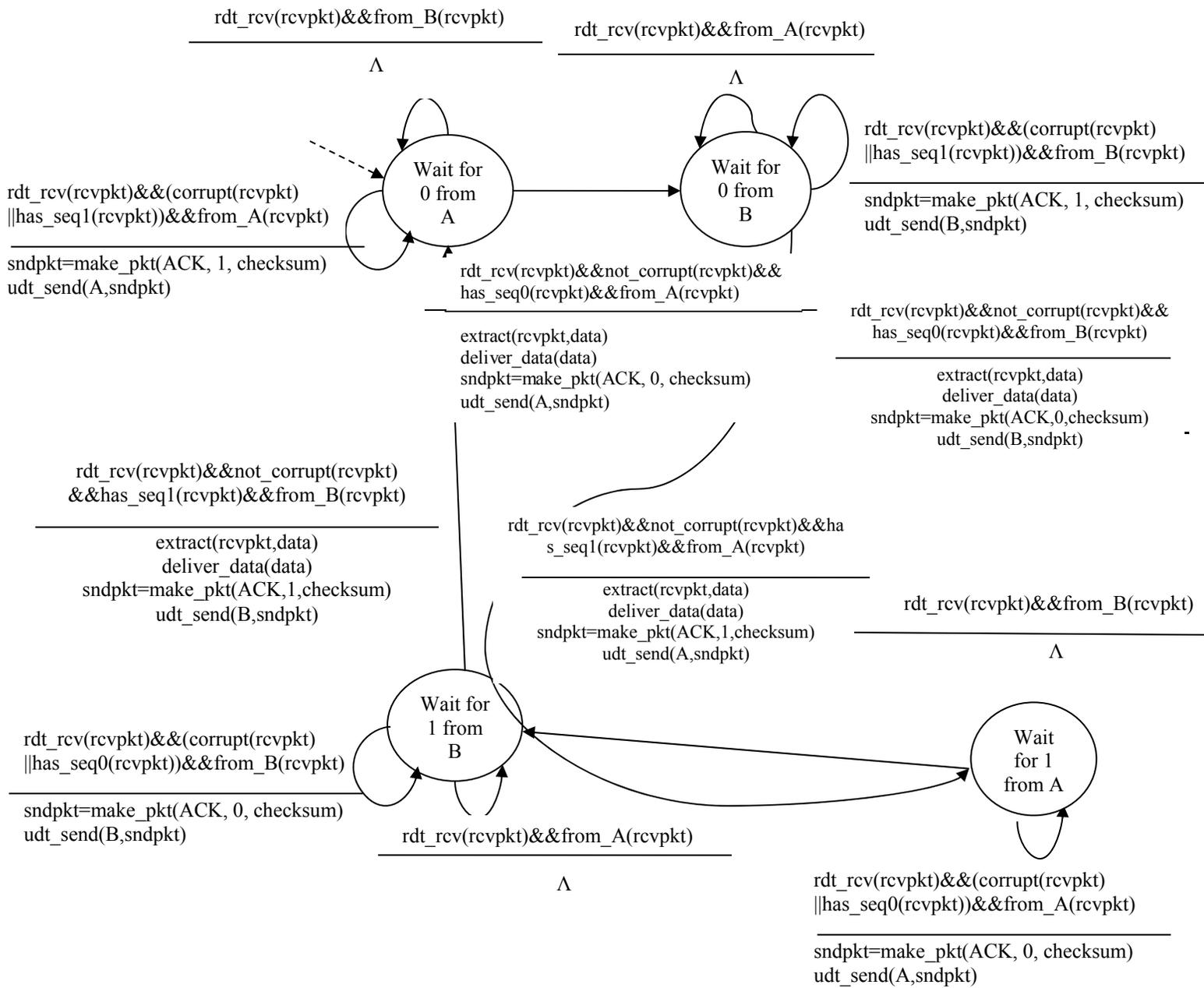


Figure 4: Receiver side FSM for 3.18

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 sender's 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 than $k-N-1$. 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, its 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 receiver's 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 t_0 . At t_1 ($t_1 > t_0$) the receiver ACKS 1, 2, 3. At t_2 ($t_2 > t_1$) the sender times out and resends 1, 2, 3. At t_3 the receiver receives the duplicates and re-acknowledges 1, 2, 3. At t_4 the sender receives the ACKs that the receiver sent at t_1 and advances its window to 4, 5, 6. At t_5 the sender receives the ACKs 1, 2, 3 the receiver sent at t_2 . 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.

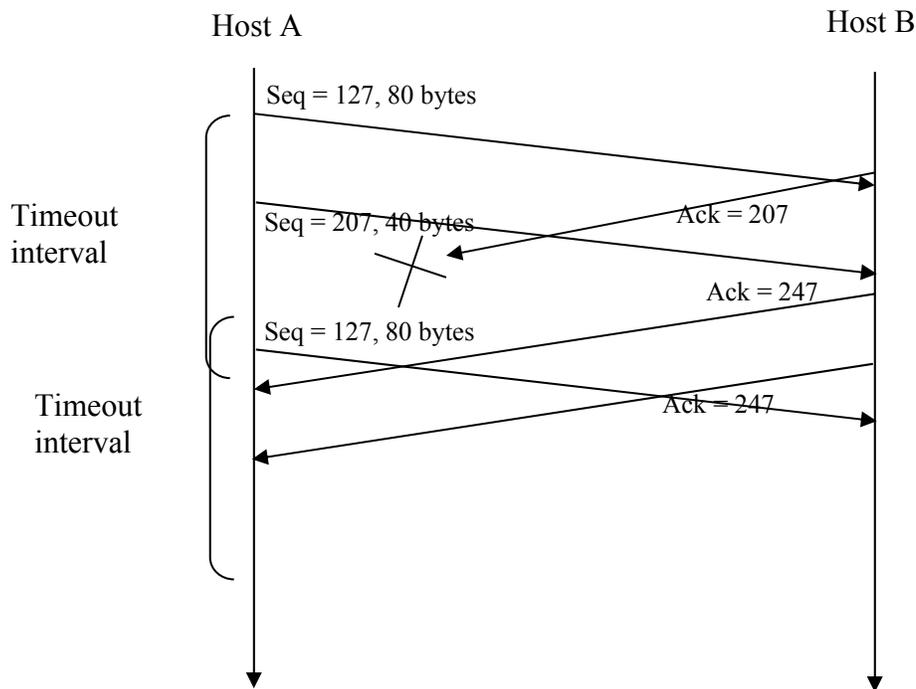
- a) The sequence number does not increment by one with each segment. Rather, it increments by the number of bytes of data sent. So the size of the MSS is irrelevant -- the maximum size file that can be sent from A to B is simply the number of bytes representable by $2^{32} \approx 4.19$ Gbytes.

$$\left\lceil \frac{2^{32}}{536} \right\rceil = 8,012,999$$

- 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

- 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.
- 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.
- 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.
-



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

$$EstimatedRTT = xSampleRTT + (1 - x)EstimatedRTT$$

$$DevRTT = y|SampleRTT - EstimatedRTT| + (1 - y)DevRTT$$

$$TimeoutInterval = EstimatedRTT + 4 * DevRTT$$

After obtaining first sampleRTT is

$$EstimatedRTT = 0.125 * 106 + 0.875 * 100 \\ = 100.75ms$$

$$DevRTT = 0.25 * |106 - 100.75| + 0.75 * 5 \\ = 5.06ms$$

$$TimeoutInterval = 100.75 + 4 * 5.06$$