Rob Johansen
u0531837

CS 4480 - Homework Assignment 4

P4

a. We begin by calculating the sum of the two bytes:

```
01011100
01100101 +
----------
11000001
```

The one's complement is then obtained by inverting the bits of the sum:

```
00111110
```

b. We follow the same process as above, but wrap the result since there is one bit of overflow:

```
11011010
01100101 +
----------
00111111
       1 +  <--- overflow bit
----------
01000000
```

The one's complement is then obtained by inverting the bits of the sum:

```
10111111
```

c. If we flip the least significant bit in each byte, the one's complement is unchanged:

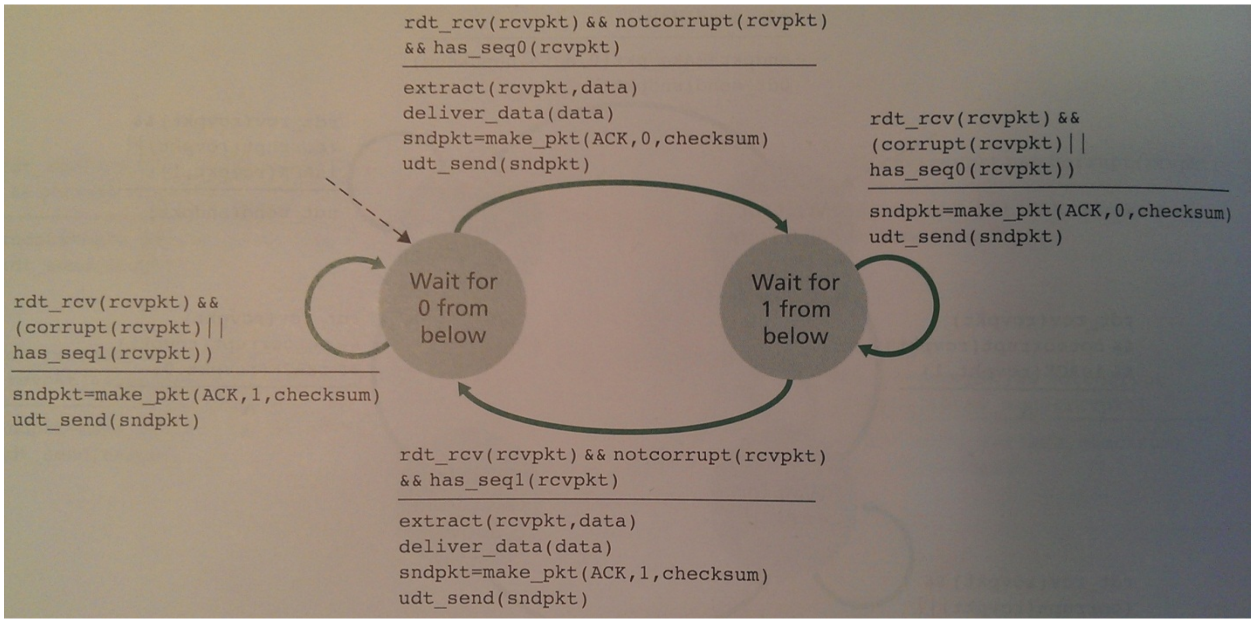
```
01011101
01100100 +
----------
11000001
```

The one's complement is then obtained by inverting the bits of the sum:

```
        00111110
```

P8

The only difference between the rdt2.2 and rdt3.0 protocols is that rdt3.0 handles lost packets. However, the burden of detecting and recovering from lost packets is placed on the *sender*. As a result, the receiver side of the rdt3.0 protocol is identical to that of rdt2.2:


rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq0(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt=make_pkt(ACK,0,checksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)|| has_seq0(rcvpkt))
sndpkt=make_pkt(ACK,0,checksum)
udt_send(sndpkt)
Wait for 0 from below
Wait for 1 from below
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)|| has_seq1(rcvpkt))
sndpkt=make_pkt(ACK,1,checksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq1(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt=make_pkt(ACK,1,checksum)
udt_send(sndpkt)

P11

The protocol would *still* work correctly if this action...

```
    sndpkt=make_pkt(ACK, 0, checksum)
```

...was removed from the self-transition in the "Wait-for-1-from-below" state. This is because there's no need to re-create the packet that was created during the transition from state 0 to 1 (even if that packet is corrupt). As the book itself indicates at the bottom of page 210 (emphasis added):

> "The sender knows that a received ACK or NAK packet (***whether garbled or not***) was generated in response to its most recently transmitted data packet."

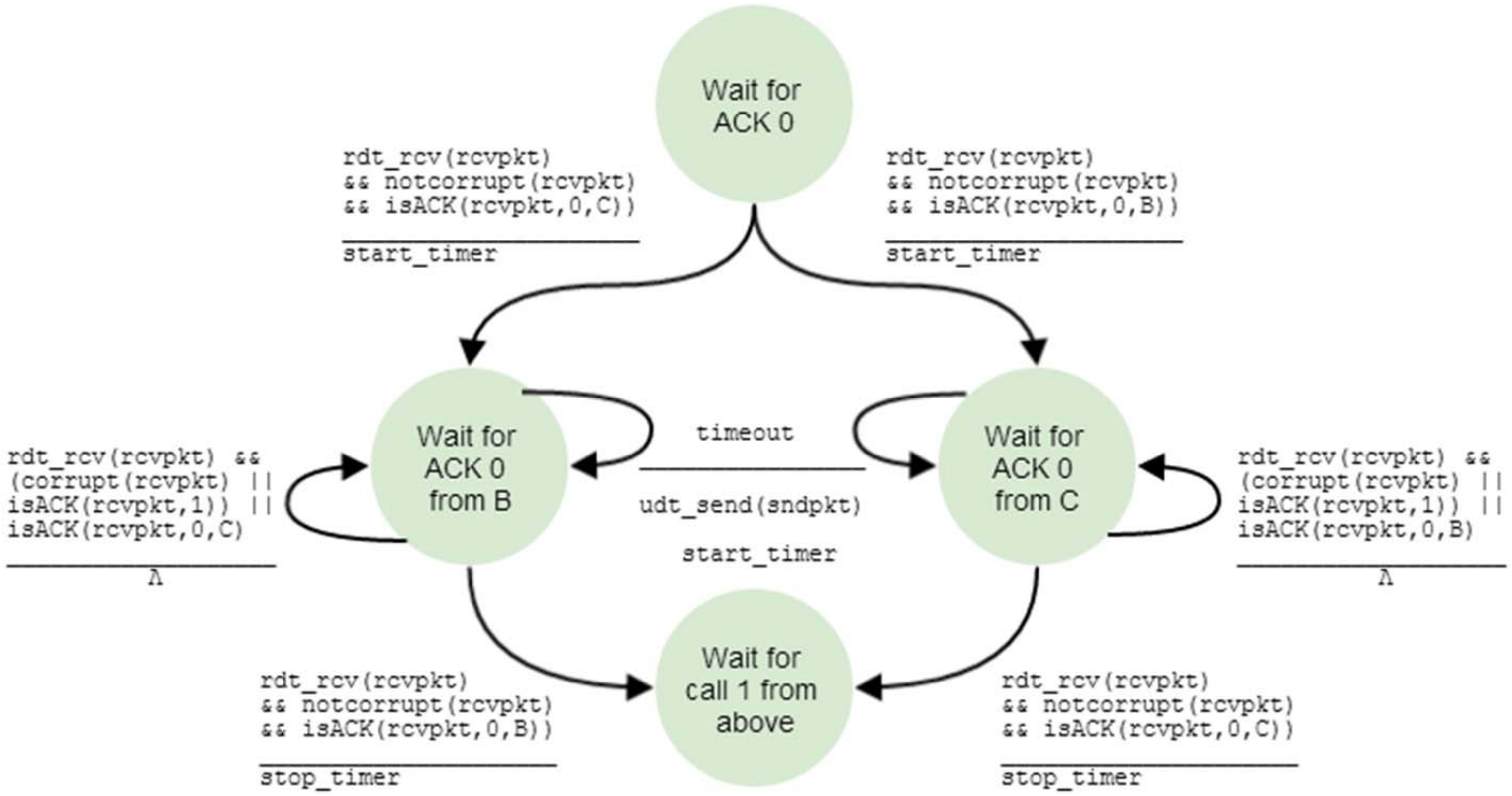
However, the protocol would *not* work correctly if this action...

```
    sndpkt=make_pkt(ACK, 1, checksum)
```

...was removed from the self-transition in the "Wait-for-0-from-below" state. As the hint given in the problem indicates, if the very first sender-to-receiver packet was corrupted, the receiver would not have an existing packet to send.

P19

The FSM for Host A (the sender) could be updated to include new states that wait for ACKs from specific hosts. For example, in the following diagram two states have been added between the "Wait-for-ACK-0" and "Wait-for-call-1-from-above" states:


Wait for ACK 0
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,0,C)) / start_timer
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,0,B)) / start_timer
Wait for ACK 0 from B
Wait for ACK 0 from C
timeout / udt_send(sndpkt) start_timer
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) \|\| isACK(rcvpkt,1)) \|\| isACK(rcvpkt,0,C) / Λ
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) \|\| isACK(rcvpkt,1)) \|\| isACK(rcvpkt,0,B) / Λ
Wait for call 1 from above
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,0,B)) / stop_timer
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,0,C)) / stop_timer

It would also be necessary to add two similar states between the "Wait-for-ACK-1" and "Wait-for-call-0-from-above" states.

The FSM for Host C (and B) could be *almost* identical to the rdt2.2 receiver (see the diagram on page 214 of the book). The only change would be in the receiver's make_pkt() function: The receiving host would need to include its hostname. For example, when Host C sends the ACK for packet 0, it would include its hostname like this:

```
    sndpkt=make_pkt(ACK,0,checksum,C)
```

The packet formats could be identical to those used in rdt3.0, with the exception that ACK packets sent by receivers would need extra bits for the hostname of the receiver.

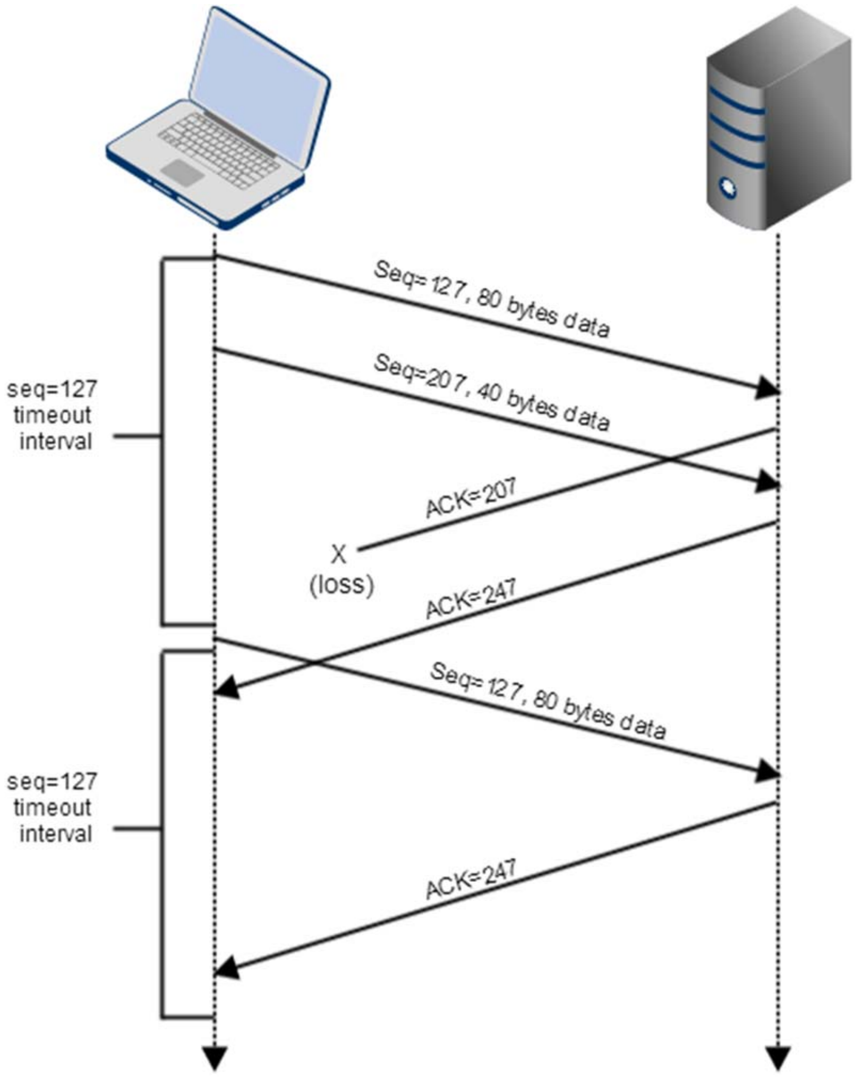
P27

a. In the second segment, the sequence number is 207, the source port number is still 302, and the destination port number is still 80.

b. The acknowledgement number is 207, the source port is 80, and the destination port is 302.

c. In this case the acknowledgement number will be 127.

d. Below is my diagram for your enjoyment:



P29

a. So that the server can verify that the ACK is returned from a legitimate client before allocating resources (such as a connection along with a socket).

b. No. The client must first send a SYN packet, then receive a carefully crafted initial sequence number (the "cookie") from the server in a SYNACK packet. The client may then send an ACK to the server to get a fully open connection – but only after the server re-calculates the cookie and verifies that the cookie plus one is equal to the value of the acknowledgment field in the ACK packet.

c. No. The server will not create fully open connections unless the client sends an ACK packet with the acknowledgment field equal to one plus the server's initial sequence number.

P40

a. In TCP slow start the congestion window begins at 1 and increases exponentially until the value of "ssthresh" is reached. You can see this pattern in figure 3.53 twice: Once during transmission rounds 1-6, then again during transmission rounds 23-26.

b. TCP congestion avoidance begins when the congestion window reaches the "ssthresh" value, immediately after the exponential increase of TCP slow start. At that point, TCP congestion avoidance grows the congestion window linearly. You can see this pattern during transmission rounds 6-16.

c. When a triple duplicate ACK is detected, TCP Reno sets the congestion window to half its current value plus 3 (cwnd/2 + 3). After the 16th transmission round, the congestion window clearly goes from 42 to 24 (42/2 = 21 + 3 = 24), which means the loss was detected by a triple duplicate ACK. If the loss had been detected by a timeout, TCP Reno would have set the value of the congestion window to 1.

d. After the 22nd round, the congestion window clearly goes to 1. As described in the previous answer, this indicates that the loss was detected by a timeout.

e. The initial value of "ssthresh" at the first transmission round is 32. This is obvious because the congestion window goes from exponential to linear growth starting at 32.

f. At the 18th transmission round, the value of "ssthresh" is 24. As described previously, this is because a triple duplicate ACK was received when the congestion window was at 42, so the congestion window reduced to half its value plus 3 (42/2 = 21 + 3 = 24).

g. At the 24th transmission round, the value of "ssthresh" is 14. This is because a loss was detected by a timeout when the congestion window was at 29, so the value of "ssthresh" was cut in half (and rounded down).

h. The 70th segment will be sent during the 7th transmission round. Here is a breakdown of the number of segments sent in each round, along with a running total in parentheses:

    Round 1 = 1 segment (1)
    Round 2 = 2 segments (3)
    Round 3 = 4 segments (7)
    Round 4 = 8 segments (15)
    Round 5 = 16 segments (31)
    Round 6 = 32 segments (63)
    Round 7 = 33 segments (96) <--- The 70th segment is sent here

i. The values of both the congestion window and "ssthresh" will be set to half the value of the current congestion window plus 3. Since the value of the congestion window after the 26th round is 8, the new values of the congestion window and "ssthresh" will be 8/2 = 4 + 3 = 7.

j. When TCP Tahoe receives triple duplicate ACKs, it sets the congestion window to 1 and "ssthresh" to half the value of the congestion window when the loss was detected. TCP Tahoe then goes back to TCP slow start. Thus, at round 17 "ssthresh" would be set to 21 and the congestion window set to 1. At round 18, the congestion window would be doubled to 2 and "ssthresh" would remain at 21. At round 19, the congestion window would be doubled again to 4 and "ssthresh" would remain at 21.

k. Round 17 is the beginning of TCP slow start, with "ssthresh" set to 21. Thus, during rounds 17-22 inclusive, packets are sent exponentially as follows:

    Round 17 = 1 packet
    Round 18 = 2 packets
    Round 19 = 4 packets
    Round 20 = 8 packets
    Round 21 = 16 packets
    Round 22 = 21 packets <--- "ssthresh" is reached here

Thus, the total number of packets sent during rounds 17-22 is 52.

P44

a. Without slow start, the increase of the congestion window would be additive: 1 MSS per RTT. Thus, assuming no loss events, it would take 6 RTTs to increase the congestion window from 6 MSS to 12 MSS.

b. The average throughput of this connection through time = 6 RTT can be calculated as follows:

$$\text{average throughput} = \frac{0.75 * 6 \text{ MSS}}{\text{RTT}}$$