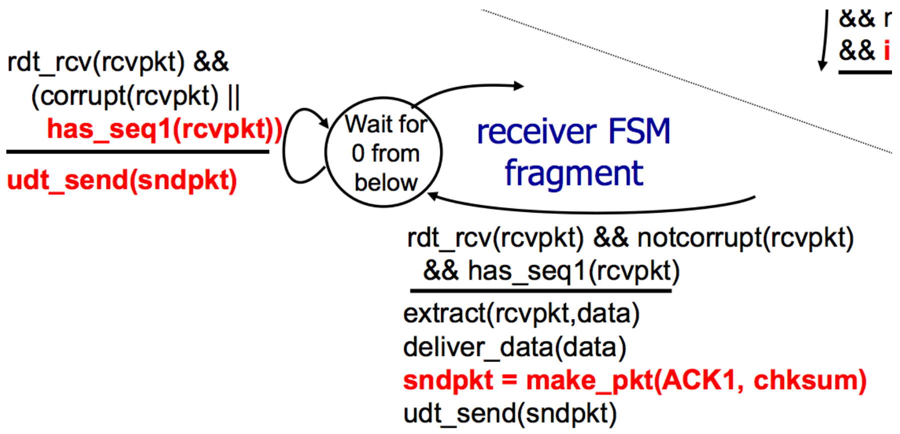
University of Waterloo
ECE 358, Spring 2017
Pencil-n-Paper Assignment 4

Shiranka Miskin, Dhruv Lal, Sam Maier

Problem 1

The only difference between `rdt 3.0` and `rdt 2.2` is on the sender side. In the following diagram only the "Wait for 0 from below" side of the FSM is shown, as the other "Wait for 1 from below" state is a near exact copy, simply with 1 replaced with 0.



Problem 2

The only case where the receiver will accept a packet is if the sequence number of the packet matches its expected sequence number. This means the only way for a packet to be delivered out of sequence would be if either packet $i + 2^k$ is sent and received without packet i being received first, or packet $i..i + 2^k$ are sent and received, but packet i is then resent.

(a)

Since the window size is $\leq 2^k - 1$, packet $i + 2^k$ cannot be sent until packet $i + 1$ has been acknowledged. Packet $i + 1$ is only acknowledged once packet i has been received, therefore the first case is impossible.

Packet $i + 2^k - 1$ is sent only once packet i is acknowledged. Since packets are never reordered, even if packet i is heavily delayed, that would mean the following packets are also delayed, therefore the only difference would be that packet i might be resent and received by the receiver after it receives some number of packets in $[i, i+2^k-2]$. If the receiver never received packet i, it would ignore all other packets until it gets the packet i that was resent, and order would still be preserved. In all other cases, the receiver's expected sequence number would be $\in [i+1, i+2^k-1]$, therefore the resent packet i would be ignored. Only once packet $i+2^k-1$ is sent will the expected sequence number be i again, but that packet will not be sent until packet i has already been successfully acknowledged. For these reasons the second case is also impossible, therefore the assertion holds.

(b)

This assertion would not hold in the unlikely case that:

1. Packets $i..i+2^k-2$ are sent
2. The timeout completes, packet i is resent and received by the receiver
3. Packets $i+1..i+2^k-2$ are received
4. The sender receives acks for those packets and the resent packet, then sends packet $i+2^k-1$
5. The receiver receives packet $i+2^k-1$ and sets its expected sequence number to i
6. The first packet i that the sender sent is finally received by the receiver

Since the receiver was expecting number i, it will accept the result and we will have a duplicated and out of order packet sent to the application.

Problem 3

(a)

No incorrectness is introduced as the receiver will always respond with the sequence number that it expects to receive next. A duplicate will not change this value, therefore no incorrectness is introduced.

(b)

It wouldn't introduce any incorrectness, but it may introduce some inefficiency. Suppose packet i and $i+1$ are sent, then the ack for packet i is received right before the timeout, but the ack for packet $i+1$ is lost. With minimal transmission delay, both packets i and $i+1$ were sent at around the same time, however since the ack for packet i resets the timer, the effective timer for packet $i+1$ is now slightly less than twice the usual timeout.

(c)

The receiver always sends back the value of the next ack it expects to receive, however on reception of an ACK, the sender sets its base to `getacknum(rcvpkt) + 1`. This sets the base to after the expected sequence number, which means it will skip that packet when the timeout clears. A fix would be to have the sender set its base to `getacknum(rcvpkt)` on `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)`.

(d)

This would allow for packets to be delivered out of sequence at the receiver. Given a window size of $N > 1$ and a k bit sequence number, packet i could be sent but delayed, while all $2^k - 1$ packets after it are sent and acknowledged. As long as the window size is > 1, it is always possible for packet $j > i$ to be sent since as packets after i are acknowledged, subsequent packets are marked as usable. Once packets enough packets in $[i+1, i+2^k-1]$ are sent, packet $i + 2^k$ can be sent, which will be indistinguishable from packet i on the receiver's end, therefore it will be transmitted to the application layer.

Problem 4

To send 10 megabytes via packets which can contain 536 bytes of application data, it will take 18657 packets. Each of these packets contain 66 bytes of headers, therefore a total of 11,231,514 bytes (or 89,852,112 bits) are sent. The transmission rate is 100Mbits per second, therefore there will be a transmission delay of 898.521 miliseconds.

In order to set up a TCP connection, a handshake is first made. This consitutes of 3 round trip delays, and then the final ack from the sender can start containing application data. It is possible that the client adds application data on the last packet it sends during the handshake.

Following from the diagram in slide 358, the maximum value for the receive window is $2^{16} - 1 = 65535$, therefore we will assume that that is the constant value of the receive window. This means that around $\lfloor \frac{65535}{536} \rfloor = 122$ packets can be in flight at a time. Assuming that data is added to handshake packets we have the following timeline:

1. At $t = 0$, a SYN from A is sent
2. At $t = 50$, the SYN is received on receiver, and an ACK is sent back to the client
3. At $t = 100$, the ACK is received, and the client begins transmitting data along with its ACK response to the server
4. 122 packets can be sent at a time, and for every RTT (100ms), another set can begin transferring (the ACKs get received at the same rate as the packets were initially sent which was the transmission rate). This goes on for $\lceil \frac{18657}{122} \rceil = 153$ window lengths, thereby taking $100 * 153 = 15,300$ miliseconds of propagation delay

5. A initiates connection closure at around time $100 + 15,300 + 898.5 = 16298.5$ miliseconds