TELCOM 2310 Spring 2022
HW 2: Solutions

Each Question weighs 25 points. Total: 4 $\times$25 = 100 pts (+10 bonus points from question 5).

1. In lecture, we mentioned that for a reliable data transfer protocol using the Selective Repeat approach, the sequence number space must be at least twice the window size (assume the sender and receiver use the same window size, and the network does not re-order messages).

   (a) Explain in your own words why this is true, and give an example that shows why the sequence space cannot be smaller. Specifically, for your example, consider a window size of 4. In this case, we need at least 8 valid sequence numbers (e.g. 0-7). Give a specific scenario that shows where we could encounter a problem if the sequence space was less than 8 (i.e. explain what messages and acks are sent and received; it may be helpful to draw sender and receiver windows). (10pts)

   The sequence space must be at least twice the window size to ensure that the receiver cannot misinterpret a retransmitted message as a valid new transmission that fits within its window. The receiver can buffer up to one window (N) packets, and the sender can have one window (N) of unacknowledged packets at any time. In the worst case, the receiver has received the full window of packets and advanced its window, but all of the acknowledgments have been lost, so the sender's window is exactly one full window "behind" the receiver. Let the sequence number at the beginning of the sender's window be 0. Then, in the worst case, the sender's window contains sequence numbers 0 through $N-1$, and the receiver's window contains sequence numbers N through $2N-1$. Therefore, to ensure that no sequence number in the sender's window can be interpreted as a valid new packet in the receiver's window, we must have that the largest possible sequence number is at least $2N-1$, for a total of at least $2N$ distinct sequence numbers.

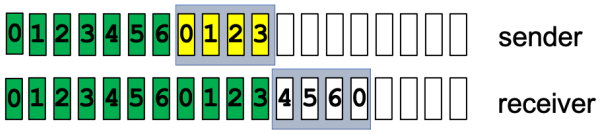

Figure 1: Example illustrating that 7 sequence numbers are not sufficient for selective repeat with a window N=4

In Figure 1, we have a sequence space of 7 sequence numbers and window size of 4. In this scenario, the sender has sent sequence numbers 0-3, the receiver has received all of those messages and advanced its window, but all of those acks were lost, so the sender has not advanced its window. This scenario illustrates why we must have at least 8 valid sequence numbers: in this case, both the sender and receiver windows contain the

sequence number 0. So, if the sender times out and retransmits the packet with sequence number 0 (which was already received by the receiver), the receiver will misinterpret it as a new packet, instead of correctly recognizing it as a duplicate.

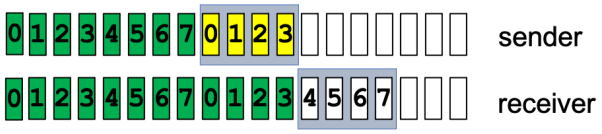

Figure 2: Example illustrating that 8 sequence numbers are sufficient for selective repeat with window N=4

In Figure 2, we see that if we use 8 distinct sequence numbers, the problem described above cannot occur: the sender and receiver windows cannot contain the same sequence number but have it refer to different packets.

(b) What relationship between sequence number space and window size is needed for Go-Back-N? Explain in your own words why this is true, and give an example that shows why the sequence space cannot be smaller. (15pts)

For a window of size N, the sequence space must be at least $N + 1$. This will ensure that no message retransmitted by the sender is incorrectly interpreted as a valid new message by the receiver. In contrast to selective repeat, the receiver does not buffer packets received out of order. Therefore, a packet will only be accepted as valid if it has the next expected sequence number.

In the worst case, the sender's window contains sequence numbers 0 through $N - 1$, but the receiver has actually already received all of those packets and is expecting sequence number N. So, to make sure that the receiver's expected sequence number is not also a valid sequence in the sender's window, we need that the maximum sequence number is at least N (giving us at least $N + 1$ distinct sequence numbers).

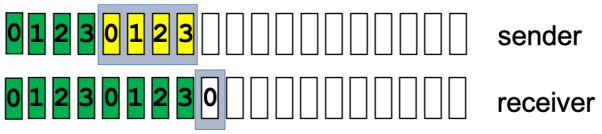

Figure 3: Example illustrating minimum sequence number space for Go-Back-N with N=4

In the example above, the window size is 4 and we have exactly 4 distinct sequence numbers (0-3). In this case, the sender has sent messages with sequence numbers 0-3. The receiver has received all of those messages, so it is expecting sequence number 0. But, all of the acks sent by the receiver were lost, so the sender will try to retransmit the old message with sequence number 0. The receiver will misinterpret this as a new message. Extending the sequence space to include sequence numbers 0-4 solves this problem (in this example, the receiver would expect sequence number 4, which is not in the sender's window)

2. Consider TCP's algorithm for setting and dynamically adapting its timeout value, as described in lecture. Assume $\alpha = 0.125$ and $\beta = 0.25$. Assume that at some point in time the EstimatedRTT is 200ms and the DevRTT is 40ms. From this point on, all measured RTTs are exactly 20ms.

(a) What is the timeout interval after 10 RTT measurements? (10pts)
Timeout = 336.88 ms (see attached code)

(b) How many such RTT measurements are needed before the timeout interval falls below 50ms? (15pts)
29 At that point, the timeout interval is 49.7 ms (see attached code, run with -i 30)

Iteration	EstimatedRTT	DevRTT	Timeout
1	177.5000	69.3750	455.0000
2	157.8125	86.4844	503.7500
3	140.5859	95.0098	520.6250
4	125.5127	97.6355	516.0547
5	112.3236	96.3075	497.5537
6	100.7832	92.4264	470.4889
7	90.6853	86.9911	438.6498
8	81.8496	80.7058	404.6726
9	74.1184	74.0589	370.3541
10	67.3536	67.3826	336.8840
11	61.4344	60.8955	305.0166
12	56.2551	54.7354	275.1968
13	51.7232	48.9824	247.6527
14	47.7578	43.6762	222.4628
15	44.2881	38.8292	199.6049
16	41.2521	34.4349	178.9917
17	38.5956	30.4751	160.4959
18	36.2711	26.9241	143.9675
19	34.2372	23.7524	129.2467
20	32.4576	20.9287	116.1723
21	30.9004	18.4216	104.5868
22	29.5378	16.2007	94.3405
23	28.3456	14.2369	85.2932
24	27.3024	12.5033	77.3155
25	26.3896	10.9749	70.2890
26	25.5909	9.6289	64.1064
27	24.8920	8.4447	58.6707
28	24.2805	7.4036	53.8950
29	23.7455	6.4891	49.7018
30	23.2773	5.6861	46.0218

Table 1: Table of calculations for problem 2

Note: You will likely find it useful to write a small program or use a spreadsheet (particularly for part b). If you use this approach, please attach the code or spreadsheet to your solution.

3. Consider a TCP connection where the sender sends segments at a constant rate of one every 10ms, and the receiver sends ACKs back at the same rate without delay. Assume the roundtrip propagation delay is 300ms. A segment is lost, and upon receipt of the third duplicate ACK, the fast retransmit algorithm detects the loss and retransmits the lost segment. Assume that

the lost segment is not the first data segment of the connection, and ignore transmission, processing, and queuing delays.

If the sender must wait to receive an acknowledgment for the lost segment before sliding its window forward and transmitting new data:

(a) How much total time does the sender lose compared with lossless transmission (i.e. how much longer does it wait to receive the ack and slide the window forward, compared to the case where there is no loss)?

Hint: consider that the lost segment is initially sent at time $T = 0$. At what time would the sender receive the ACK for that segment in the no-loss case? At what time does that happen if the sender first needs to detect the loss, retransmit the segment, and *then* receive the ACK? (15pts)

Consider that the lost segment is initially sent at time T=0. If there is no loss, we would expect to receive an ACK and move the window forward exactly one RTT later, or at time $T = 300\,\text{ms}$.

The next three segments after the lost segment are sent 10 ms apart, at $T = 10\,\text{ms}, T = 20\,\text{ms}, T = 30\,\text{ms}$.

If the segment is lost, we will receive our third duplicate ACK at time $T = 330\,\text{ms}$ (1 RTT after the third packet is sent at time $T = 30$).

We will then immediately retransmit the lost segment and receive an ACK for it 1 RTT later, at time $T = 630\,\text{ms}$. Therefore, we lose $630\,\text{ms} - 300\,\text{ms} = 330\,\text{ms}$ compared with lossless transmission.

(b) Assuming a timeout of 400ms is set at the time the lost segment is originally transmitted, how much total time does the sender save by using fast retransmit, compared with waiting for the timeout to expire before retransmitting the segment (i.e. how much sooner is it able to slide the window forward than in the case where it waits for the timeout to occur)? (10pts)

In this case, the lost segment will be retransmitted when the timeout expires at $T = 400\,\text{ms}$. The acknowledgment will be received 1 RTT later, at time $T = 700$. Therefore, fast retransmit lets the sender receive the ACK $700\,\text{ms} - 630\,\text{ms} = 70\,\text{ms}$ sooner than waiting for the timeout to expire.

4. Consider that only a single TCP (Reno) connection uses one 100Mbps link, which does not buffer any data. Suppose that this link is the only congested link between the sending and receiving hosts. Assume that the TCP sender has a huge file to send to the receiver, and the receiver's receive buffer is much larger than the congestion window. We also make the following assumptions: each TCP segment size is 1460 bytes; the roundtrip propagation delay of this connection is 50msec; and this TCP connection is always in congestion avoidance phase (i.e. ignore slow start).

(a) What is the maximum window size (in segments) that this TCP connection can achieve (round to the nearest whole segment if needed)? (10pts)

$$100\,\text{Mbps} = \frac{W_{\max} \times 1460 \times 8}{0.050\,\text{sec}}$$
$$5\,\text{Mbits} = W_{\max} \times 11680\,\text{bits}$$

$$\frac{5 \times 10^6 \text{ bits}}{11680 \text{ bits}} = W_{\max}$$
$$W_{\max} = 428.08 \approx 428$$

(b) What is the average window size (in segments) and average throughput (in Mbps) of this TCP connection? (15pts)

Since we ignore slow start, the average window size is repeatedly increasing linearly from $\frac{W_{\max}}{2}$ up to $W_{\max}$. Therefore the average window size is: $\frac{\frac{W_{\max}}{2}+W_{\max}}{2} = 0.75W_{\max} = 0.75 \times 428 = 321$ segments.

The average throughput is 0.75×100 Mbps $= 75$ Mbps.

Alternatively, we could calculate this as $\frac{321 \text{ segments} \times 1460 \text{ bytes/segment} \times 8 \text{ bits/byte}}{0.05 \text{ sec}} = 74.9856$ Mbps $\approx$ 75 Mbps (rounding to whole segments gives us a slightly lower throughput here)

5. (Bonus) Recall that parallel HTTP connections are often used to improve load performance for webpages.

(a) Given what we've learned about TCP, why are parallel connections not an ideal solution to HTTP performance problems (from a network perspective)? (5pts)

Because TCP congestion control attempts to enforce per-connection fairness, an application that opens up multiple parallel connections can get a larger bandwidth share than an application that only uses a single connection.

(b) Why are parallel HTTP connections commonly used anyway? (Recall our discussion from Chapter 2, and be as specific as you can) (5pts)

The introduction of request pipelining in HTTP/1.1 attempted to reduce incentives to open up parallel connections by allowing multiple requests to be sent over a single connection without waiting for responses. However, this approach still suffers from *Head-of-line blocking*. Because requests are processed in order, small objects can get stuck "waiting in line" for a larger object to be transmitted, resulting in poor perceived page load time. By requesting objects on separate parallel connections instead, the HOL blocking issue is eliminated: small objects requested on separate connections can be received and rendered without waiting for a large object that is being retrieved in parallel. Note that HTTP/2.0 interleaved object transmission tries to fix this problem.

Partial credit for contrasting with non-pipelined HTTP and explaining how parallelization can improve response time by allowing propagation delays for requesting/retrieving different objects to overlap with each other.