Sample Solution to Problem Set 2

1. Performance of web caching

Problem 9 of Chapter 2, page 173.

Answer:

(a) The time to transmit an object of size L over a link or rate R is L/R. The average time is the average size of the object divided by R:

$$\Delta = (900,000 \text{ bits})/(1,500,000 \text{ bits/sec}) = .6 \text{ sec.}$$

The traffic intensity on the link is (1.5 requests/sec)(.6 msec/request) = .9. Thus, the average access delay is (.6 sec)/(1 - .9) = 6 seconds. The total average response time is therefore 6 sec + 2 sec = 8 sec.

(b) The traffic intensity on the access link is reduced by 40% since the 40% of the requests are satisfied within the institutional network. Thus the average access delay is (.6 sec)/[1 - (.6)(.9)] = 1.2 seconds. The response time is approximately zero if the request is satisfied by the cache (which happens with probability .4); the average response time is 1.2 sec + 2 sec = 3.2 sec for cache misses (which happens 60% of the time). So the average response time is (.4)(0 sec) + (.6)(3.2 sec) = 1.92 seconds. Thus the average response time is reduced from 8 sec to 1.92 sec.

2. Stop-and-wait for broadcast

This problem is the same as Problem 15 of Chapter 3 of text, with some clarifications.

Consider a scenario in which a host, A, wants to simultaneously send messages to hosts B and C. A is connected to B and C via a broadcast channel – a packet sent by A is carried by the channel to both B and C. Suppose that the broadcast channel connecting A, B, and C can independently lose and corrupt messages (and so, for example, a message sent from A might be correctly received by B, but not by C). Assume that there are no out-of-order message deliveries. Design a stop-and-wait-like protocol for reliably transfering a packet from A to B and C, such that A will not get new data from the upper layer until it knows that both B and C have correctly received the current packet. Give finite-state-machine descriptions for A and B.

Hints: Extend the rdt3.0 protocol studied in class and described in the text. The FSM for B and C are essentially the same; they are also essentially the same as for the receiver in the rdt3.0 protocol.

Answer: 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 the attached figure. 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.

3. Maximum sequence number in GBN

Consider the GBN protocol with $N = 3$, with no out-of-order arrivals and no bound on the sequence numbers. In the following, assume that the packets and acknowledgments are numbered from 0, and that ACK[i] is a cumulative acknowledgment indicating that packets 0 through i have been received.

Answer: We first note that the data below the sending window (that is, less than the lower bound in the window) is never sent again, and hence – because out-of-order arrival is disallowed – if PKT[N] arrives at the receiver, then nothing at or before PKT[N-3] can arrive later. Similarly, for ACKs, if ACK[N] arrives then (because ACKs are cumulative) ACK[N-1] (or before) cannot arrive later. By the convention adopted in class, ACK[N] acknowledges the receipt of frames 0 through N.

(a) Show that if the receiver window contains 6, then PKT[0] cannot arrive at the receiver. Generalize this argument to show that if a number i is in the receiver window, then PKT[$i-6$] or any older data cannot arrive at the receiver.

Answer: If PKT[6] is in the receive window, then the earliest the window can be is [4-6]. This in turn implies ACK[3] has been sent, and thus PKT[1], PKT[2], and PKT[3] were received. Thus PKT[0], by our remark above, can no longer arrive.

Similarly, if PKT[i] is in the receiver window, then the earliest the window can be is $[(i-2), i]$. This in turn implies ACK[$i-3$] has been sent, and thus all data until PKT[$i-3$] were received. Thus PKT[$i-6$], by our remark above, can no longer arrive.

(b) Show that if the sender window contains 6, then ACK[2] (or earlier) cannot arrive at the sender. Generalize this argument to show that if a number i is in the sender window, then ACK[$i-4$] or any older acknowledgment cannot arrive at the sender.

Answer: If 6 is in the window of the sender, then the lowest the sending window can be is [4-6]. This means that ACK[3] must have been received. Once an ACK is received, no smaller ACK can ever be received later.

Generalizing this, if i is in the sending window of the sender, then the lowest the sending window can be is $[(i-2), i]$. This means that ACK[$i-3$] must have been received. Thus ACK[$i-4$] or any older acknowledgment cannot arrive at the sender.

(c) Using (a) and (b), argue that it is sufficient to use sequence numbers 0 through 5 for the GBN protocol with $N = 3$.

Answer: By (a), when the receiver window contains i, it cannot receive packet $i-6$ or higher. Similarly, by (b) when the sender window contains i, it cannot receive an ack for $i-4$ or lower. Thus, at any time there may be at most 6 distinct packets in flight (or 4 distinct acks in flight). It thus suffices to use 6 sequence numbers (0 through 5) with wraparound since the sender (receiver) can distinguish the packets (acks) using the different sequence numbers.

(d) **Bonus problem:** Generalize your answer for part (c) to arbitrary N. That is, show that it is sufficient to use sequence numbers 0 through $2N-1$ for GBN for arbitrary N. Also give an example that illustrates that the range $[0-4]$ will not suffice for $N=3$.

Answer: Let N be the window size of both the sender and the receiver. If i is in the receiver window, then the lowest the receiver window can be is $[(i-N+1), i]$. This means that packet $i-N$ must have been received. This in turn implies ACK$[i-N]$ has been sent, and thus all data until PKT$[i-N]$ were received. Thus, the earliest the sender window can be is $[i-2N+1, i-N]$. Therefore, the receiver cannot receive packet $i-2N$ or earlier.

Similarly, we can show that if i is in the sender window, then the earliest the sender window can be is $[i-N+1, i]$. This implies that ACK$[i-N]$ has been received. So, no ACK earlier than this can be received now since the acks are cumulative and we have no out-of-order delivery.

Putting these two together, we obtain that there may be at most $2N$ distinct packets in flight at any time and at most N acks in flight at any time. It thus suffices to use $2N$ sequence numbers (0 through $2N-1$) with wraparound since the sender (receiver) can distinguish the packets (acks) using the different sequence numbers.

4. Sequence number and receive window size in TCP

You are hired to design a reliable byte-stream protocol that uses a sliding window (like TCP). This protocol will run over a 100 Mbps network. The RTT of the network is 100ms, and the maximum segment lifetime is 60 seconds. How many bits would you include in the "Sequence number" and "Receive window" fields of your protocol header?

Answer: As discussed in class, the sequence number field must be large enough that it does not wrap around within the maximum segment lifetime. The maximum amount of data that can be sent in the maximum segment lifetime is $100\text{Mbps} \times 60\text{s} = 6 \times 10^9$ bits ≈ 750MB. The number of bits needed to represent 750 million is 30.

The receive window must be large enough so that it allows the receiver the receive as much data as possible in a single round-trip time without wraparound. The maximum amount of data that a receiver can receive in one RTT is 100 Mbps times 100 ms, which equals 10 million bits, which is 1.25 million bytes. The number of bits need to represent the number 1.25 million is 21.

5. RTT estimation and timeout in TCP

Recall that the Jacobson-Karels algorithm for RTT estimation and timeout calculation is given as:

$$\begin{aligned}
\texttt{DevRTT} &= (1-\beta)\cdot\texttt{DevRTT} + \beta\cdot|\texttt{SampleRTT} - \texttt{EstimatedRTT}|,\\
\texttt{EstimatedRTT} &= (1-\alpha)\cdot\texttt{EstimatedRTT} + \alpha\cdot\texttt{SampleRTT},\\
\texttt{TimeoutInterval} &= \texttt{EstimatedRTT} + 4\cdot\texttt{DevRTT}.
\end{aligned}$$

Suppose that `EstimatedRTT` is 400ms at some point and subsequent `SampleRTT`s are all 100ms. Assume an initial `DevRTT` value of 50ms; use $\alpha = .125$ and $\beta = .25$.

How long does it take before the `TimeoutInterval` value falls below 200ms? If the same measurements and calculations continue *ad infinitum*, what will be the value of `TimeoutInterval` in the limit?

Answer: It takes 25 rounds for the `TimeoutInterval` value to fall below 200ms. The value of `TimeoutInterval` in the limit is 100ms. See the attached excel sheet.

6. TCP congestion control

Parts (a)-(d) and (h) of Problem 27, pages 291-292.

- **(a)** TCP slowstart is operating in the intervals [1,6] and [23,26].
- **(b)** TCP congestion advoidance 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.
- **(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 15-31 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.