

# Homework 3

## Solution Key

### Problem 1

The picture above shows the famous TCP saw tooth behavior. We are assuming that fast retransmit and fast recovery always work, i.e. there are no timeouts and there is exactly one packet lost at the end of each “tooth”. We are assuming that the flow control window is large and that the sender always has data to send, i.e. throughput will be determined by TCP congestion control.

In the picture,  $W$  represents the congestion window size at which a congestion packet lost occurs (expressed in maximum transfer units). You can assume that  $W$  is large, so feel free to approximate  $(W-1)$  or  $(W+1)$  by  $W$ .  $STT$  represents the “saw tooth time” expressed in seconds.

- Calculate the average throughput  $T$  for this connection as a function of the roundtrip time ( $RTT$ ), the maximum transfer unit size ( $MTU$ ), and packet loss rate  $PLR$  for this connection. Please use the notation suggested by the figure, i.e.  $W$  and  $STT$ , as intermediate values if you need them. Explain each step of your answer. Hint: Calculate  $STT$  and amount of data sent in one  $STT$  and then divide.

(soon to be released)

- The above TCP connection is used in an idle network, and the link capacity of the bottleneck link is in  $B$ . What throughput will the user observe? Please explain briefly.

(soon to be released)

- What packet loss rate will the connection suffer? Explain.

(soon to be released)

### Problem 2

The window-scale option of TCP (see the notes, page VI-51) provides a means for supporting a large advertised window. PAWS (see page VI-54)

provides a means for extending the 32-bit sequence number of TCP.

- a. Suppose we have a one-gigabit/second terrestrial network (see page VI-48). What is the maximum throughput we can attain if we use 16 bits for the window size? Explain.

The maximum-size segment we can send is the maximum window size, 64k bytes in this case. After sending such a segment, the send window drops to zero and nothing more can be sent until we receive a window update. If the receiver consumes the data instantly and sends back an ack for all the data along with a window update for another maximum window, the sender can send another maximum-size segment after waiting roundtrip-time seconds after sending the first. In this case, since the roundtrip time is 60 milliseconds, the maximum throughput is 64k-bytes/60 milliseconds, or 1.092 megabytes/second (or 8.738 megabits/second).

- b. With the window-scale option we can have windows up to  $2^{30} - 1$  in length (to keep things simple, let's say the maximum window size is 1 gigabyte). With a gigabyte window, what's the maximum throughput we can attain over the network of part 1?

One gigabit/second. With a gigabyte window, we are no longer constrained by the window size and can, in principle, send one gigabyte every roundtrip time. However, we can't transfer data any faster than the network's bandwidth.

- c. With 32-bit sequence numbers and no PAWS, what problems will we have if we have sustained maximum-speed transfers using the parameters of part 2? Explain, showing an example.

With 32-bit sequence numbers and one gigabit/second transmission, wrap-around will happen in around 32 seconds. If a segment starting with sequence number  $k$  is sent but is somehow delayed, it might still be active up to the end of the maximum segment lifetime of two minutes. If four gigabytes of data are sent successfully in the meantime and our lost segment reappears, since its sequence number is what's now expected, the receiver might accept it rather than the proper segment with sequence number  $k$ .

- d. Explain how PAWS would solve these problems.

What's needed is a mechanism that insures that all segments transmitted within the period of one maximum segment lifetime are dis-

tinguishable. To keep the arithmetic simple, let's assume that 32-bit wrap-around occurs in 30 seconds. Then we need at least two additional bits to distinguish the four sets of indistinguishably sequenced segments that could be produced in two minutes. PAWS provides these bits by adding a 32-bit timestamp to each segment. This timestamp must be such that it cycles more slowly than once every two minutes and that its least significant bit changes at least once each time the sequence number cycles (otherwise it doesn't distinguish the four indistinguishably sequenced segments).

### Problem 3

*Suppose we have a ten-gigabit/second network with a roundtrip time of 100 milliseconds. The maximum segment size (MSS) is 1500 bytes.*

- a. *How long will it take for a TCP connection to reach maximum speed after starting in slow-start mode?*

The transmission rate starts at one MSS-size segment per roundtrip time, or 15,000 bytes/second instantaneously. The maximum speed is 1.25 megabytes/second. Thus it must increase its speed by a factor of 83,333.33. Assuming no packet loss, after each roundtrip time (100 milliseconds), the speed doubles. So, after  $\log_2(83,333.33) (\approx 16.347)$  roundtrips, the speed will reach the maximum. Thus maximum speed is reached in about 1.7 seconds.

- b. *Assume there is some probability that a packet is lost in transmission, even though there is no congestion. Give a scenario in which the congestion window becomes 1 MSS (i.e., 1500 bytes) and TCP begins growing the window linearly rather than exponentially (i.e., it's not in slow-start mode).*

Suppose the first segment sent is received and ack'd, but the second segment sent is lost. Thus after timing out on the ack for the second segment, the sending TCP sets its congestion window to one MSS and begins the linear increase of its congestion window.

- c. *How long will it take the TCP connection to achieve maximum speed after the scenario of part 2, assuming no further packet loss?*

83,333 roundtrips, or 8,333 seconds (two hours, eighteen minutes, and 53 seconds).

- d. Suppose (exactly) every 100th packet is dropped. Approximate the throughput of TCP on this network.

For the first 100 packets, TCP will be in slow start mode, but after it drops the 100th packet, it will always be increasing the rate linearly except for the cases where it drops a packet and divides the rate by 2. Eventually TCP will converge on a steady state loop increasing linearly to the same rate each time and then cutting that by half to start over at the same place.

Let  $x$  be the number of packets sent at the beginning of the loop. Every  $RTT$  seconds, TCP will receive all the acks and it will send out  $x + 1$  packets. Then another  $RTT$  seconds goes by and it sends out  $x + 2$  packets until it has sent a total of 100 packets. Since on the last packet, the rate must be halved and the result must be  $x$ , we know that the pattern tops out at  $2x$ .

So, TCP is in a loop where it linearly increases from  $x$  packets to  $2x$  packets over and over again. Since the increase is linear, we know that the process takes  $2x - x$  or  $x$  steps and that the average number of packets sent at each step is  $(x + 2x)/2$  or  $1.5x$ . So, the total number of packets sent is  $x * 1.5x$  or  $1.5x^2$ .

We already know, however, that the total number of packets sent is 100, so we just need to solve  $1.5x^2 = 100$  for  $x$  to determine what  $x$  is.  $x = \sqrt{\frac{100}{1.5}} = 8.165$ .

Now, as we determined before, the average number of packets sent at each step is  $1.5x$  and each step takes  $RTT$  seconds to complete. Given that each packet is  $MTU$  bytes, the throughput of the connection will be  $\frac{1.5*x*1500}{0.1} = 183711.731$  or  $183.7KB$ .

## Problem 4

We have learned that TCP provides a “reliable” byte stream. Consider a simple client/server pair. The client has 1 kilobyte of data it wishes to send to the server. It calls `connect`, `write`, and `close` appropriately, and each returns without error. Explain how:

- a. The server might not receive the data.

There are a few solutions to this problem.

The most realizable scenario is the following: `write` returns on insert-

ing the data into the TCP buffers/queue, not on actual sending, and `close` returns on just telling the TCP stack to close the connection, not on successful closure. Thus, you could successfully get past these calls, and end up with the data still in your local buffers. While this data is still in the local buffer, the physical line could be cut or the machine could crash, and thus the server never gets the data.

The other two easily realizable scenarios are attacks: One is a Mal-lory attack, in which an attacker spoofs the server's address, and has all packets routed for the server to it instead; thus the server never gets the data. The second is a Man-in-the-Middle attack, in which a machine hijacks the initiated TCP connection between the client and server, receives the client data, sends an ACK and goes through the appropriate close procedure, and just immediately closes (sends a FIN) to the server.

*b. The server might receive corrupted data.*

Again, there is more than one answer. The simplest is that the data within the TCP packet is garbled in such a (admittedly unlikely) way that the checksum still matches all the data.

*You should assume this is the only TCP connection between the two hosts, ever.*