Assignment 4 - Computer Networks

Question 1:

a. Minimal Round Trip Delay (1 point)

8.8 years

b. Why Setup/Teardown Per Message Was a Mistake (2 points)

Send hello and SYN and wait for SYN/ACK: 8.8 years

Send the actual message and ACK and wait for that response: 8.8 years

Then close it is another 8.8 years to be polite

That is 26.4 years, and for closing it and opening is the 17.6 years. You don't want to do that every time.

c. Keeping Connection Open Permanently (1 point)

Well you never send FIN/ACK so you never need that extra 8.8 years to close and also to open again. Now you just keep it open and send when you want/need.

d. Window Sizes Needed (2 points)

1000 bytes/month aka 8000 bits/month

30 days × 24h × 3600s = 2,592,000 and RTT is 8.8 × 365 × 24 × 3600 = 277,516,800

month = 2,592,000s, 8000/2,592,000s = 0.00309 bits/s

Window = 0.00309 bits/s × 277,516,800 = 857,526 bits

PROF

e. Piggyback ACKs (2 points)

Yes and no. You should piggyback ACK when you have a message ready so you can just instantly send it back as soon as possible, but if you don't then you send the ACK straight away. Don't make them wait even if it is a short time, not counting the 8.8 years.

Question 2:

a. Why Messages Take Over 12 Years (2 points)

Well let's say we send a message and in 8.8 years we should get an ACK for that message. Something got lost/corrupted so we send again and that will take 4.4 years. So 8.8 + 4.4 = 13.2 years.

Multiple retransmissions could occur if packets are lost repeatedly, adding 4.4 years per loss.

b. High-Level Redesign (6 points)

You could use UDP instead to have more control over the packets, wait time, ACK time and so forth.

You would ID every message with a sequence number so you know what has arrived and what has not, and you would also ACK immediately upon receiving a message. The sender would maintain a list of unacknowledged messages. When an ACK arrives, the sender marks that message as successfully delivered.

Then you make it behave like TCP/IP but instead of using the RTT to send again right away, you account for the fact that the message might just be delayed rather than lost. So you wait 9 years to be certain before retransmitting. And also you can add onto it a safety layer - you could send the message again every 1 year/2 years/3 years or whatever you choose. This way if the original message gets lost, one of the copies might still arrive at 4.4 years instead of having to wait 9 years for the timeout. The receiver uses the sequence numbers to detect and discard any duplicate messages.

Question 3: TCP Reno Analysis

a. Fast Retransmit and Fast Recovery (4 points)

Fast Retransmit

When the **receiver** detects an out-of-order segment, it generates an immediate duplicate ACK.

Example:

- Receiver gets segments 78, 79 in order → sends ACK 80 (expecting 80 next)
- Segment 80 gets lost!
- Receiver gets segment 81 → still needs 80 → sends ACK 80 (1st duplicate)
- Receiver gets segment 82 → still needs 80 → sends ACK 80 (2nd duplicate)
- Receiver gets segment 83 → still needs 80 → sends ACK 80 (3rd duplicate)

When the **sender** receives 3 consecutive duplicate ACKs (all saying "I need segment 80"), it immediately retransmits the missing segment without waiting for a timeout.

PROF

Problem it solves: Without Fast Retransmit, the sender would wait for a timeout (which can take seconds) before resending. Fast Retransmit detects the loss immediately and resends right away, making recovery much faster.

Fast Recovery

After Fast Retransmit, TCP Reno does the following:

- Set ssthreshold = CWND / 2
- Set CWND = ssthreshold (NOT back to 1 like TCP Tahoe)
- Skip slow start and continue with congestion avoidance

Problem it solves: In TCP Tahoe, any packet loss causes CWND to reset to 1 and restart slow start, which grinds the network throughput to a halt. TCP Reno recognizes that 3 duplicate ACKs indicate mild congestion (packets are still getting through), so it only cuts the window in half and continues sending. This prevents the network from becoming underutilized due to a single lost packet.

Together: Fast Retransmit and Fast Recovery allow TCP Reno to recover from packet loss quickly without drastically reducing throughput.

b. Slow start

Slow start occurs during RTT 0-6 and RTT 23-25.

c. SSHthreshold

It is about (CWND aka y axis) at 32/33. around there where slow start stops and sshthreshold begins

d. when and what?

TCP congestion avoidance is triggered when the congestion window (CWND) reaches or exceeds the slow start threshold (ssthreshold). At this point, TCP stops growing exponentially and switches to linear growth, increasing CWND by 1 MSS per RTT instead of doubling.
This happens because slow start's exponential growth could quickly overwhelm the network if continued indefinitely. By switching to linear growth at ssthreshold, TCP probes for available bandwidth more cautiously.

In Figure 1, congestion avoidance operates during RTT 6-14 and RTT 15-21.

Question 4: Fix server/client

a. UDP or TCP and what transport layer

It is a UDP and IPV4.
I found that the socket has DGRAM and AF_INET aka
`socket(AF_INET, SOCK_DGRAM, 0)) < 0)`
AF_INET tells us it is ipv4, if it would be for example AF_INET6 it would be ipv6

Server and Client both have the SOCK_DGRAM and AF_INET.

PROF

b. What is wrong and how to fix

What is cousing the Errors aka bad protocal

1. buffer size is 25. But that just fits in 25 char including spaces and symbols. So fo the client to be able to recv all of the message he would have to increase the size of it to 29 or bigger.
2. You have the while loop checking `c++ while (n > 0)` But in doing so when the client sends a empty message he will stop the recv since he got a message of size 0. So you clould check if n == exit or quit or for example bye like he said in the code.

c. What happens changes when sending over internet

When you send over the internet with UDP there is no protection like TCP has so things can go wrong without errors:

1. Packets get lost:
Some messages just never show up at the server. Like the server might only get msg 1, 2, and 6 but msg

3, 4, 5 are just gone. No error shows up because UDP doesn't care if it arrives or not.

2. Packets come in wrong order:
The messages might arrive in different order than you sent them. So server could get msg 3 before msg 2 because they take different routes through the internet.

3. Packets get duplicated:
Same message could show up multiple times. Like "msg 1: Hi" could be printed twice by the server.

Why this happens:
UDP does not offer guarantee, reliability, no ACK and no error notifications. So when packets go through routers and the internet, they can get lost, take different paths, or get copied. But the `send()` and `recv()` both work fine on each side so you don't get any error messages even though the packets never made it or came in wrong order.