CPSC 3600 002

Spring 2020

# Exam 3 (last updated 4/20/2020)

This is a take home exam. You can use your notes, books, computer and the Internet. However, you must work on your own. Please submit the word document or a pdf to the exam3 submission on canvas by the due date.

Name: **Reagan Leonard**

**Statement** Aside from any possible assistance from our course Tas, I have completed this exam without the help of anyone.

Signature: Reagan Leonard

Q1 . Look at the TCPEcho program in code/TCPEcho. This is the example from Donahoo Chapter 6 involving the TCPServer-Fork-SIGCHILD. The client does 1 iteration – sends the input character string and waits for the server to echo the message.

Answer the following questions, issue (in vi) a :set number to show the line numbers. You can reference line numbers in your answers.

* What line of code initiates the TCP 3-way handshake?
  + **The 3-way handshake begins when the client issues a connect(). Line 60**
* Explain the function of the AcceptTCPConnection() on line 42 (a single sentence please).
  + **Its purpose is to block on an accept call. When the client issues a connect, this initiates the 3-way handshake. Once complete, the ‘connect’ is moved to the ‘active queue’. When a server issues an accept, it blocks until there is a TCP connection waiting in the active queue.**
* Why does the AcceptTCPConnection() routine not issue a listen()?
  + **The main reason I can think of is that it doesn’t know what to “listen()” to. This file does not know which socket to issue the listen() to. Instead, it simply issues an accept() and waits for a socket to connect to it.**
* In HandleTCPClient, how would you confirm that this is running on the child process?
  + **One way to do this would be to display the pid of the parent in server.c and then display the pid once HandleTCPClient begins to run. I would use getpid() to get the current running procees’s process id.**
* What line of code initiates the TCP 4-way teardown. More specifically, what line initiates an active close? What line initiates a passive close?
  + **There are actually two lines of code that do this. The side (client or server) that issues a close() first performs an ‘active close’. This initiates the 4 way teardown.**

Q2. Is it possible for a UDP application to use the send() socket function rather than the sendto() ? If no, explain why not. If yes, explain why.

**It is not possible for a UDP application to use send() rather than sendto() because UDP is not guaranteed to know where it should send the packets/message. Therefore, it is necessary to include this information in the function call. send() does not take a parameter for an IP address like sendto() does, so send() would not work.**

Q3. Assume we start the TCPSend receive side program on your VM using the following:

./TCPRx 5000 1000000 100000

If we were to run the alias lss on your VM in another terminal before starting the send side application (TCPTx), what should we see in the lss output?

I ran both these processes and got the following output from lss:

rpl@rpl-VirtualBox:~/git/CPSC3600-Students/code/TCPSend$ lss

cupsd 694 root 10u IPv6 15152 0t0 TCP [::1]:631 (LISTEN)

cupsd 694 root 11u IPv4 15153 0t0 TCP 127.0.0.1:631 (LISTEN)

dnsmasq 853 nobody 5u IPv4 17713 0t0 TCP 127.0.1.1:53 (LISTEN)

**TCPRx 25931 rpl 3u IPv4 56193 0t0 TCP \*:5000 (LISTEN)**

Q4. From your VM, run traceroute to [www.google.com](http://www.google.com). (If for whatever reason you are not able to run traceroute successfully from your VM, login to a SoC linux machine and run the program from there)

* Show your results
* Explain what each line tells us

rpl@rpl-VirtualBox:~/git/CPSC3600-Students/code/TCPEcho$ traceroute www.google.com

traceroute to www.google.com (172.217.12.36), 30 hops max, 60 byte packets

***(This first line shows the hostname and ip that is to be reached, the maximum number of hops to the host that traceroute will attempt and the size of the byte packets to be sent.)***

1. 172.27.0.254 (172.27.0.254) 11.623 ms 11.595 ms 11.571 ms

***(Then each line lists a hop to get to the destination. The hostname is given, followed by the ip of the hostname in parentheses, followed by the roundtrip time that it takes for a packet to get to the host and back to the initiating computer. By default, traceroute sends three packets for each host so three response times are listed.)***

**2**  108-206-208-1.lightspeed.gnvlsc.sbcglobal.net (108.206.208.1) 29.976 ms 33.742 ms 33.723 ms

**3** 99.5.25.14 (99.5.25.14) 28.165 ms 29.456 ms 32.056 ms

**4** 99.168.141.126 (99.168.141.126) 32.323 ms 33.628 ms 33.622 ms

**5** 12.83.113.9 (12.83.113.9) 52.497 ms 56.460 ms 56.463 ms

**6** gar1.nsvtn.ip.att.net (12.122.96.85) 56.453 ms 14.787 ms 18.644 ms

**7** 12.255.10.12 (12.255.10.12) 22.024 ms 21.987 ms 21.934 ms

**8** \* \* \*

***(Asterisks denote packet loss)***

**9** \* \* 108.170.236.236 (108.170.236.236) 35.474 ms

**10** 108.170.249.163 (108.170.249.163) 21.669 ms 108.170.249.35 (108.170.249.35) 35.420 ms \*

**11** 216.239.59.153 (216.239.59.153) 35.353 ms 108.170.236.128 (108.170.236.128) 35.274 ms 108.170.228.78 (108.170.228.78) 41.925 ms

**12** \* 209.85.249.45 (209.85.249.45) 32.872 ms 108.170.252.129 (108.170.252.129) 32.803 ms

**13** 108.170.233.116 (108.170.233.116) 35.720 ms 216.239.62.212 (216.239.62.212) 41.656 ms 216.239.63.252 (216.239.63.252) 33.686 ms

**14** 108.170.252.161 (108.170.252.161) 35.676 ms 108.170.252.129 (108.170.252.129) 41.544 ms 108.170.252.161 (108.170.252.161) 41.567 ms

**15** 108.170.226.55 (108.170.226.55) 29.115 ms 29.944 ms dfw28s04-in-f4.1e100.net (172.217.12.36) 34.025 ms

***(We can see in line 15 that this trace took 15 hops to reach the destination server, dfw28s04-in-f4.1e100.net (172.217.12.36), which serves the Google website.)***

Q5 Consider the TCPSend program. Assume we run the server on ada8 as follows:

*Ada8: TCPRx 5000 1000000*

And we run the client on your VM or another SoC linux machine

*TCPTx ada8.computing.clemson.edu 5000 1500000 1000000*

Notice the potential issue…the dataSize is not aligned with the chunkSize. Will the current code work? Referencing the code, explain why it does or does not work.

**Yes, to my understanding of the TCPSend code, this will work. In lines 114-120 of the TCPSender.c file, we compare dataSize with chunkSize to see which one’s bigger. If the dataSize > chunkSize (which it is in this case, because 1500000 > 1000000), then numberChunks is set equal to dataSize / chunkSize, with the remaining bytes going to modBytes. So the code does allow for these 2 variables to not match.**

Q6 For the following network, what is the bandwidth delay product of the pipe between H1 and H2.

L1 l2 l3 l4 l5

H1 ---- R1 -------R2 -----R3 ------R4 ------H2

The propagation delay and data rate for each of the 5 links is 0.05 seconds and a data rate of 10000000bps. The exception is the Link3 (between R2 and R3) - the propagation delay is .100 seconds and the data rate is 1000000bps.

Hint: As a simplification, reduce the network to a single link that reflects the sum of each link prop delay and a data rate of 1000000 bps.

**H1 ------ 1Mbps ----- H2**

**RTT = (.100 + 4\* 0.05) \* 2 = 0.60 seconds = 600 ms**

**BDP = 1000000 bps \* 0.60 seconds \* 1 bytes / 8 bits = 75,000 bytes**

Q7 Building off Q6,

Q7 part A: if a sliding window transport protocol is used to send data between H1 and H2, what is the optimal window the sender should use? By this we mean the window (W) that fills the pipe. Assume the segment size is 1448 bytes and all frames are exactly 1528 bytes.

**BDP / frame size = 75000 / 1528 bytes = 49.08 ~ 49 frames**

**This means the sender's window must be 49 segments in order for the connection to consume the available 1 Gbps.**

Q7 part B: Double check your answer- given the optimal Window size you find along with an the RTT (easily found based on the simplifying assumptions), what is the expected goodput H2 should see? Assume the sending application always has data to send, NO packets are ever dropped, AND that there are no competing flows.

**If the sender sends 1460 bytes of application data (leading to frames of size 1528 bytes) with a W=49 and an RTT of 600ms what is the goodput the receiving application sees:        1460\*49 \*8  / 0.60 = 953,866.6 bps.**

Q8 . Chose the single correct answer. The pipe capacity (in the context of a Unix IPC pipe) is:

1. The maximum throughput one would expect through the pipe
2. The bandwidth delay product associated with the code path between the sender (of the pipe) to the receiver.
3. The largest number of bytes that the pipe can hold.



1. The average number of bytes that the pipe holds (averaged over time period the pipe is being used).

Q9 . Chose the single correct answer. The pipeSize parameter to our TCPSend program is best represented by which of the following statements

1. The average number of bytes that are in flight (averaged over a time period the pipe is being used).
2. The largest number of bytes that can be outstanding over the path.
3. The maximum throughput one would expect over the connection.



1. The bandwidth delay product associated with the code path between the sender (of the pipe) to the receiver.

**Q10 Related to named pipes**

**Part A**

Term1: issue ‘cd ~/tmp’ (mkdir ~/tmp if it does not exist).

And then issue ‘mkfifo sendPipe; mkfifo recvPipe’

And then in term 1: issue

*cat < recvPipe | wc*

And in Term2

*echo “hello” > sendPipe*

After you issue the echo command, what would you expect to see in both terms?

**Both terminals stall (wait) on these commands because the command issued in term2 is continuously echoing “hello” to the recvPipe in term1.**

**Q10 Part B**

in term 1 issue

1. ***cat < recvPipe | cat > sendPipe***

And in Term2, issue

1. ***echo “hello” > recvPipe &***

And then in Term 2 issue:

1. ***cat < sendPipe***

If you did commands A,B, and C sequentially, explain what you would expect to see in both terms after issuing each command. To make it clear what I’m asking, please fill in the blanks below. In addition, please add two sentences to explain what you think is happening.

After command A, I would see **a stall (simply waiting)** in term 1 and **nothing** in term 2

After command B, I would see **a stall (still waiting)** in term 1 and **[1] 25641** in term 2

After command C, I would see **command A resolved** in term 1 and **“hello”**

**[1]+ Done echo “hello” > recvPipe** in term 2

**Basically, what I think is happening here is the commands in term2 begin echoing “hello” to recvPipe and then concatenate the contents of sendPipe (cat < sendPipe). This is what prints out in term2 after command C. (Command A seems to be concatenating what is in recvPipe and piping that into sendPipe, possibly.)**

Q11 Modify the TCPsender.c code so that each chunk begins with the serialized struct messageHeader that was used in UDPEchoV3. The TCPTx program should send an updated sequence number and send time stamp for each new chunk sent. You do not have to modify the TCPReceiver.c code. Submit just the modified TCPsender.c code. Even if you do not get it working, document your code so we see your thinking. A part of the grading will be to use your TCPReceiver.c, build it with our solution and make sure your client is correctly sending the messageHeader (we will modify our TCPReceiver.c to extract the sequence number and sender’s timestamp).