Project #3: TCP Data Transfer Solution Guide

434/534 Network Design

March 31, 2017

Must compile: StudentSocketImpl-data.java, client3.java, server3.java
Command for testing server: java -DUDPPORT=8816 -DLOSSRATE=0.50 server3 8817
Command for testing client: java -DUDPPORT=8816 -DLOSSRATE=0.50 client3 128.239.26.66 8817

Step 1: Depending on how you did the packet resend timer cancelations, you may want to revise it before starting the bulk of project 3. In project 2 I did my timer cancelations based on the state it was in, but this would not work in the actual data transfer. Its best to do the cancellations based on the ackNum that the receiver sends with an ack packet. So the keys into the timer and packet hashtables should be the seqNum of the packet that the timer is resending, and each time an ack is received, all timers with seqNums < p.ackNum should be cancelled. I also changed the incrementation system to add the amount of bytes in the packet, even those without data in them. So instead of adding 1 for a syn or a fin, I added 20 (since that's the size of a packet with nothing in the data buffer)... note that this isn't a requirement, and technically it doesn't properly reflect what actually happens with sockets, but I made the decision since it made the most sense to me.

Test 1: Run the test at the end of the TCP State Machine project. Make sure it still works with and without packet loss

Step 2: I followed the data path in my implementation of TCP. So I started with dataFromApp, got everything sent from the App onto the sendBuffer, and called sendData(). Note that length is the exact number of bytes to send from the buffer, not necessarily the entire buffer. Also, **Be careful with the infiniteBuffer objects**. Read and understand the class thoroughly before trying to use it. It's really easy to lose your place or not account for an offset. In most cases, an append takes an offset of 0 and a copyOut takes an offset of getBase(), and after each copyOut() you must advance() the base however many bytes of data you copied out (since you no longer need it).

Test 2: Have the sendData function print out the information you just inserted onto the sendBuffer to make sure the data being pulled off the buffer correct. Use server3 and client3 for testing from here on out.

Step 3: Implement sendData. Take everything off the sendBuffer, split it up into chunks of 1000 or less, put them into packets, and call sendPacket (which I had implented as a wrapper function from the TCP State Machine project. It was really handy.) One thing I needed to account for is the difference between incrementing the seqNum while sending a data packet and incrementing the ackNum while sending an ack. For data packets I called increment after I sent, while for ack packets I called increment before I sent. This is an arbitrary detail that you may not encounter with your implementation, but I needed to account for it. Also, copyOut() is called here, so again, be careful.

Test 3: Modify receivePacket to have it handle receiving a data packet and have it print out its data buffer to see that everything was sent correctly.

Step 4: Again, modify receivePacket to handle a data packet, but this time dont print it (or keep the print for debugging purposes). Firstly, only accept dataPackets which you need (p.seqNum == ackNum), then take the data buffer from the packet and append it to the recvBuffer.

Test 4: Have the getData function print out the information you just inserted onto the sendBuffer. Make sure the data being put on the buffer correct.

Step 5: Implement getData. Note that length is the max amount to be copied, so the data available could be less than length. Make sure to account for that. Also, copyOut() is called here, so again, be careful.

Test 5: At this point the majority of the data transfer should be working properly. Looks at the code for server3 and client3 and make sure your output corresponds to what they are trying to do.

Step 6: Properly closing the connection was a big hassle with data transfer, mainly because the application would call close before the data transfer was done. So you must put in some sort of flag and wait loop combination that makes sure the actual close() code isnt run until the final data transfer is complete. For mine, I had a flag saying to hold when dataFromApp was called, and once the corresponding ack was received, then the flag could be lowered and a close could be done if desired.

Test 6: At this point the project should be nearly complete for ideal conditions and for packet loss. Try a bunch of different packetloss scenarios to make sure everything is accounted for.

Point Distribution

20% -Test 2 works
20% -Test 3 works
20% -Test 4 works
20% -Project works without packet loss
20% -Project works at .50 packet loss consistently (Give partial credit if it works sometimes and not other times. Use your best judgment.)