Nathan Wichman
Project One: UDP Datagram Socket Documentation

I wrote my project in java. It contains three main classes: udpserver, udpclient, and Packet. The Server class creates a datagram channel and waits to receive a file name from a client. If that file name exists, it is broken down into packets of size 1024 bytes and sent one by one to the client. The client then sends acknowledgments when it receives any given packet. Sliding window is implemented so that the server will not send more than five packets before receiving an acknowledgement from the earliest packet. After twenty cycles of a while loop, if no acknowledgement is received, the server will resend that first packet. Packet is a class that only contains two instance variables, to hold its number in the over all file and the byte array data in that packet.

The server first checks if the file exists. If it does it determines the amount of packets to send by dividing the size of the packet in bytes by 1024, and adding 1 for the remainder. It then starts a for loop that iterates the number of packets there are. It will send up to five without needing to check that the correct acknowledgment has been received. Every iteration, it attempts to receive an acknowledgment, and stores it in an arrayList. Then, it checks to see if the lower limit of the sliding window is in that arrayList, if so, it is removed and the sliding window is incremented. Then, it checks to see if the upper limit has been reached, if it has it stops the ability to send packets (via boolean) but allows the for loop to continue, although the iterator is decremented to repeat the same iteration as to not skip packets. After 20 cycles, if the correct acknowledgement has not been received, the packet corresponding to the lower limit is send to the client again. After all packets have been sent, the server sends a string “done” to the client to signal termination. Then that for loop is broken back to the original while loop, allowing the server to wait for a new connection.

The client will take user input for a port number, ip address, and file name. It will send that file name to the corresponding server. It then attempts to receive packets in a while loop from the server. When a packet is received, a code is extracted from the beginning that represents the packets number. Then the packet is put in an arrayList. A variable called currentPacketNumber is evaluated, and if a packet in the array is equal to it, it is removed from the array and concatenated to a byte array that will be written to the file later. Then the current number is incremented. If a packet that is less that the current number is found in the array, it has already been written and arrived by mistake from the server, and is so deleted from the array list. After the array list has been evaluated, if the current packet has not been received, then the previous acknowledgement is sent to the server. After the code “done” is received, the recievedData byte array is written to the file and the program terminates.

Several major difficulties arose during this project. For one, I found integers hard to send and extract from bytebuffers. So, strings were sent instead of integers for acknowledgments and packet numbers. These strings had null bytes attached to the end of them, which was unknown to me at first. When I tried to evaluate if they were equal to other strings, they always came up false. When I figured that out, I removed the null bytes. For the packet number, a string of integers is attached to the beginning of the send packet followed by a ‘D’. The client will convert the byte array packet it received into a string then read it off into an a substring until it reaches the ‘D’. Then, that substring is parsed to an int. The remaining portion of the string is reconverted back into a byte array, but this time without the integer + ‘D’ portion, for writing to the file or storing in the array list. This would have been much easier if I could have figured out how to use getInt and putInt methods for ByteBuffer correctly.

Another major difficulty that arose was that I never was able to use mininet to test my programs. First I tried to graphically remote access to the EOS computers following the online

directions. After watching tutorials and trying several times, I was unable to work TightVNC to tunnel in. Then I tried to download mininet to my own computer and test it myself. After several hours and many youtube tutorials, I could get to the point of sudo mn -x, but the graphical portion did not work. My final chance was to come to the eos labs themselves, but I was unable to compile java code in the mininet environment even after following the professors instructions. The lack of mininet was a major hit to my productivity for two main reasons. One, it wasted many hours of me trying to access it. Two, I was unable to test my code's ability to navigate packet corruption/loss errors. To remedy this, I added and deleted code within my classes that would occasionally not send a packet, or send a packet in the wrong order. That way I could test my software's functionality, although it lacked the power of mininet and I do not believe it was sufficient.