Wesleyan University, Fall 2022, COMP 411

Homework 5: Transport protocols and a chat app

Due by 5pm on October 14, 2022

---

1. Written problems (5 points)

Problem 1. *Consider the stop-and-wait protocol and suppose the channel can reorder packets: i.e., if the sender sends packet i followed by packet j, packet j may arrive before packet i. We know that the stop-and-wait protocol uses a window size of 1. Now assume there are only 2 sequence numbers, 0 and 1. Show using a timeline that the final stop-and-wait protocol we discussed (and analyzed) in class can result in each of the following two scenarios.*

a: *A packet is delivered to the receiver-side application layer twice.*

b: *A packet is never delivered to the receiver-side application.*

Problem 2. *This problem looks at the selective repeat and Go-Back-N protocols. Answer true or false to the following questions and briefly justify your answer. Assume a window size of three.*

a: *For the selective repeat protocol, is it possible for the sender to receive an ACK for a packet that falls outside of its current window?*

b: *For the Go-Back-N protocol, is it possible for the sender to receive an ACK for a packet that falls outside of its current window?*

Problem 3. *This goal of this problem is to help you better understand UDP datagrams by looking at them in Wireshark.*

a: *Record some traffic from web-browsing, and select one UDP datagram from your trace. Include a screenshot of this datagram in your submitted homework. From this datagram, determine how many fields there are in the UDP header. (You shouldn't look in the textbook! Answer these questions directly from what you observe in the packet trace.) Name these fields.*

b: *By consulting the displayed information in Wireshark's content field for this datagram, determine the length (in bytes) of each of the UDP header fields.*

c: *The value in the Length field is the length of what? (You may consult the textbook for this answer.) Verify your claim with your captured UDP datagram.*

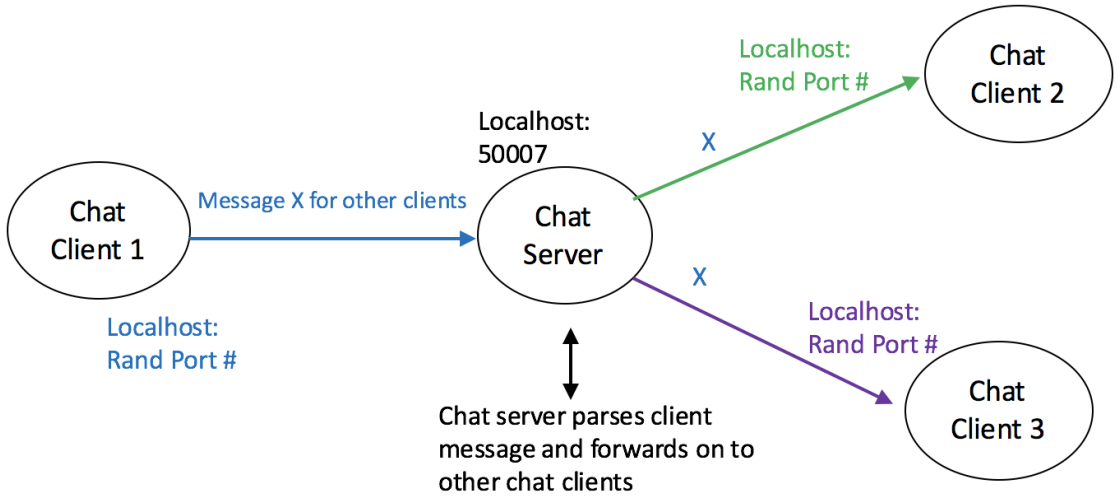

FIGURE 1. Architecture for group chat client.

d: *What is the maximum number of bytes that can be included in a UDP payload? Hint: the answer to this question can be determined by your answer to part b.*

e: *What is the largest possible source port number? Hint: see the hint in part d.*

f: *What is the protocol number for UDP? Give your answer in both hexadecimal and decimal notation. To answer this question, you'll need to look into the Protocol field of the IP packet containing this UDP datagram.*

g: *Examine a pair of UDP datagrams in which the first datagram is sent by your host and the second datagram is a reply to the first datagram. Describe the relationship between the port numbers in the two datagram.*

2. CODING PROBLEM (15 POINTS)

PROBLEM 4. *In this problem, you will create a simple chat app. Because in this class we will not cover how to use threads or how to use locks to control access to shared variables (typically you would learn how to use these in an operating systems class), I have made a number of simplifying assumptions as well as provided a bit more structure to the code. Thus, you should not need to do too much coding. However, some thought will be required in designing the communication protocol you implement to mediate interactions between the chat client and the chat server.*

Your chat app will function as a group chat and operate as in Figure 1. All chat clients first connect to the chat server. When any chat client wants to send a message to the other chat clients,

it sends the message over its connection with the chat server, which then sends it to each of the other chat clients. Thus, whenever one client sends a message, all client receive it.

- **Chat client.** *You will see that the chat client is multi-threaded with a* `write_sock` *function called in one thread and a* `read_sock` *function called in the other. These sockets write and read data respectively from the chat server. These are the only functions you need to fill out.*
  - *In the* `write_sock` *function you will continuously read data from the command line (i.e., user input), put your protocol header on it, and write it to the chat server. Your protocol header should comprise several fields, including at least the length (in bytes) of the data. You will need some way of determining when the header terminates, and when the data being sent (payload) begins.*
  - *In the* `read_sock` *function you will continuously read data from the socket with the server, parse the protocol header that the server put on the data it sent, determine how much data to read, read until you get the expected amount of data, and display it (print) to the screen. When you print to the screen, you should format the display so that the name of the user who sent the data comes first, followed by a colon, followed by the data, as in, "user: data".*
- **Chat server.** *The chat server spawns a thread to serve each client. The chat server, however, when serving a client in one thread, may need to write data to clients in other threads, and so will now need to have access to all client sockets regardless of which thread is currently being run. To handle this, a list of sockets will be maintained, along with their associated IDs: this has already been implemented for you. What you need to fill out are the following functions.*
  - *In the* `serve_user` *function you will use the* `read_data` *function to continuously read data from the socket (i.e., chat client) being served in that thread. Whenever you have read a complete message you will send it to all other clients using the* `send_data` *function. Note: you should not access the* `chat_list` *variable in this function.*
  - *In the* `read_data` *function you will read from the socket passed to the function, check whether a full message has been received, and when it has, return that message so it can be sent to the other clients. You will want to check whether an empty string has been read from the socket, indicating that the client has left.*
  - *In the* `send_data` *function you will loop through all of the available connections and send the message to every other client, excepting the original sending client.*
  - *in the* `cleanup` *function you will close the socket being served in the thread as well as remove the connection from the list of connections available.*

I recommend writing out pseudocode for what you need to do, then adding comments for the pieces to fill in. That way, if something isn't working or you don't have time to finish something, I

can see what you were trying to do and possibly give you partial credit.

Going further: this is not to be turned in, but a nicer way to write networking applications such as this chat app is to use the python twisted module `https://twisted.org/`. It takes a bit of thought to wrap your brain around event-driven networking (hence the name twisted), but once you do, you can eliminate the issues we have with threading and making sure that clients are able to both read and write in separate threads.

3. Submission

Upload your written work as `hw5.pdf`, your *.py files to the Google Drive directory I have created for you named `comp411-f22-USERNAME/hw5/`. You should replace `USERNAME` with your Wesleyan username.

Do not forget that your written work must be submitted as a PDF! Make sure that at the top of each file you have put your name! Do not, however, change the names of the files.