Computer Networks (COIS 4310H)
Assignment 1 - Winter 2021

Programming Component

Due date: February 4th, 2021 (noon)

All of the programming portions of your assignments in this course will be interdependent. By this I mean that you will not be able to complete assignment #2 unless assignment #1 is working. Similarly, you can't finish #3 without completing assignment #2.

This assignment requires you to create a client-server environment. Your submission will consist of two programs and an RFC of your own creation. There are no restriction on which language you use to create your solution as long as it can run on loki.trentu.ca. It is expected that your code contain the appropriate level of comments, proper variables names and good coding practices (as described in the requirements section on Blackboard: Documenting your Code).

This will be difficult during the pandemic where we are all separated BUT if you want to, you may partner up with another student for the assignments. If you are struggling with the code, sometimes another set of eyes can help. Only groups of two though! If you prefer to work on your own, I'm perfectly fine with that.

Information on the components contained in the assignment will be discussed in class before you are expected to implement them in code. (i.e. RFC, "packet header", port #...)

Before reading any further: DON'T PANIC !

This is meant as a learning exercise. Should you need help, I'll give you as much as possible for this first assignment. Certainly to the point where messages can be sent to/from the server to the client.

Assignment #1:

For this assignment you are being asked to create a chat application. The server component should accept network connections from up to **X** different clients. I'm more than happy to have X range from 2 to 5 depending on your comfort level with coding. For those of you that aren't comfortable with C programming yet, 2 is fine and can make for a much simpler experience. For those of you who like a challenge, the gauntlet has been dropped!

When writing your application, The server side will need to present a static port number to the network. Please use the last 4 digits of your student number and add 50000 to it to determine the port number you should use. As an example, my student number finishes with 2019 so my port number is 52019.

All of the applications should run at the command line. Once started the server code should be set to run "forever" and only terminated by ^C when you are done testing your app. Similarly, the client code should run until the user enters "bye" as a command. Here's how the app should work:

1. The server code is started and waits for connections.
2. A client connects to the server and passes it a username.
    a. The username can be the currently logged in user or an alias provided on the command line.
    b. The client code then accepts keyboard input of the form "destination:message" or "bye" to exit. (you are permitted to split this into 2 inputs: dest/message).
3. When the client connects, the server will broadcast a message to all currently connected clients that: "username" has joined the conversation.
4. Should a client disconnect (bye), again a message should be sent to all connected clients indicating: "username" has left the conversation.
5. Clients should be able to:
    a. send a message to the whole group (prefix text with "all:")
    b. send a message to an individual (prefix text with "username" e.g. jacques:don't forget to bring doughnuts to the next class.)
    c. get a list of all connected client names ("who:")
    d. disconnect from the group chat.

The application you are writing will have to be able to distinguish the different type of information being transmitted:

- Login request
- This is a message for X or ALL or this is a private message from X to Y
- Who?
- bye

These messages are sent by the clients. The clients need to understand messages from the server as well.

- Welcome
- You have a message from Y:
- Sorry to see you go

As such, the "packet" of information being exchanged must be made up of two different components: a header part and a data part. For assignment #1 the header should contain a "packet" number, a code version number (1 for assignment 1, 2 for #2...), the source client name, the destination client name (or "all") as well as a field to describe the "verb" being asked for of the server.

Save room in your header for expected changes brought about by assignments #2 and #3!!! The data portion should be able to accommodate 256 text characters (user input).

DOCUMENTATION: For the purpose of this part of the application you will need to create an RFC of your own design to document the commands/responses that your application will adhere to. This document should describe, in detail, both the "packet" header and the data field. The RFC must also include a detailed description of the "verbs" the application will recognize. As well, the RFC must document the actions to be performed by the "server" side upon receipt of the commands.

I do NOT expect a document anywhere as detailed as the sample RFCs given below. Your RFC should follow the "spirit" of the RFC format not its stringent syntax/structure.

With the RFC in place, the person coding the server side should be able to write the complete server application independently from their partner. Similarly, the person writing the client code should be able to write a functional client solely from the information contained in the RFC. I do not expect anyone to work in a vacuum. RFCs however allow for independent groups to write applications which work seamlessly together even when the groups have never met or exchanged email.

Ensure that your RFC properly describes your message format. This would include a description of all of the header fields as well as the data field(s). As an example see the "Functional Specifications" section (3.1) of https://tools.ietf.org/html/rfc793.

```
TCP Header Format


    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |          Source Port          |       Destination Port        |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                        Sequence Number                        |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                    Acknowledgment Number                      |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |  Data |           |U|A|P|R|S|F|                               |
   | Offset| Reserved  |R|C|S|S|Y|I|            Window             |
   |       |           |G|K|H|T|N|N|                               |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |           Checksum            |         Urgent Pointer        |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                    Options                    |    Padding    |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |                             data                              |
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

                            TCP Header Format
```

Ensure that your code is properly **documented** and that it runs on loki. Once the team is done their work, please ZIP up the application and RFC document and upload it to blackboard. If you are working in teams of two, each member must submit a copy of the assignment. Also, in the RFC please make sure it is clearly marked who your partner is. I will also require that you demonstrate your app, by logging both a server and client sessions, to ensure that it satisfies all of the above requirements.

RFCs: https://www.ietf.org/standards/rfcs/

RFCs are simply documents which clearly define the internals of a program/standard. The RFC for SMTP (788, 821...1427...1651...2034) can be used as a guide. I do not expect any where near that level of detail. Here's what I do expect to find in your document/RFC:

- Authors and Dates
- Application name
- Version of the RFC
- A summary of the application and its purpose.
- A description of how the application is architected into server and client components. (don't forget network protocols, port numbers...)
- A description of the packet header detailing the type, size and purpose of every field. Show the packet header as an ASCII box chart
- A description of the "data" portion of the packet. Again, size, type and purpose.
- There should also be a section describing each verb: syntax, purpose and how the application should behave once the verb has been received.

All in all I do not expect your "RFC" to be longer than 4-5 pages (no cover but references if you use them). In any event, you will have to figure out all of this information before/while you are writing the solution. This will just formalize the documentation process.

Good examples of RFCs which will show:

header details: https://tools.ietf.org/html/rfc793 (section 3.1)
    "verbs": https://tools.ietf.org/html/rfc5321 (Section 4.1.1.x).

Hints: **Pick the approach complexity that you are most comfortable with !**

There are a number of obstacles you are going to have to overcome to get this to work properly for multiple simultaneous clients. I would suggest strongly that you get this application working for a SINGLE client. Just to make sure you have messages flowing from one client to the server. Adding a second user to the chat just means another person is running the same client code (no extra work there!).

For the simplest of models, only 2 clients, the server will sit and wait for a message from the client A.

- It will then pass that message along to client B.
- There's also a requirement for the server to acknowledge back to client A that it received and passed the message along.
- After passing the message to B, the server should wait for an acknowledgement that B received the message.

The process now works the same way from B to A. The server will stop and wait for a message from B, then:

- It will then pass that message along to client A.
- There's also a requirement for the server to acknowledge back to client B that it received and passed the message along.
- After passing the message to A, the server should wait for an acknowledgement that A received the message.

Approach A) The simplest non-realistic chat server will round robin through the sockets for their client. trhat means everything runs in lock-step. There's no concurency in this approach but it is the simplest to code.

Approach B) This approach is quite complex and full of logic landmines. It will, however work with a fair amount of effort. If you recall when you fork() a process, it inherits a copy of the parent's data. In the perfect world, we would like to have the parent and its children to share data elements. This way, a message received by a child process (on the server) could pass a message to all of the children. Unfortunately, the parent does not share any memory areas with the children. fork() therefore, may not be the technique you want to employ. Unless of course you want to leverage shared memory areas and the challenges they present?

Another issue you will encounter involves BLOCKING on the I/O channels when you make your system calls. In some languages, you can implements a "timeout" value where the call will only block for so many seconds before failing and falling through to the remainder of your code. In pure C programming, you aren't that lucky. The blocking will occur on your accept(), sendto()/recvfrom() or send()/recv() calls. You could choose to go for non-blocking I/O (recall fcntl() from 3380) but that introduces other artefacts.

You could also use the select() call to determine which socket descriptor (File descriptor) is reading for reading/writing before you try to read/write from them. It is a bit more complex but only slightly so. Again, the issue with this is that unless a socket is FULL, select() will see it as ready to do I/O.

Another tool in your toolset is to use alarm() with a signal handler. Set that up before calling send()/recv(), have it wait for a second. Then is the call is successful, you have data to process. If the signal handler fires off, you can set a flag to indicate the lack of data transfer. (hint: I would be surprised if send() didn't work 100% of the time for this application).

To start coding this, accept 2 client connections then proceed to allow them to exchange messages. At this stage you can allow for blocking reads. This will allow you to walk through the message passing and debug your logic. Once you have the process working as it should, you can then move onto the stages of implementing non-blocking calls and/or alarms.

I would suggest creating an array to hold the usernames and socket numbers for each participant. Then when you want to send a message to "Mickey" you simply need to scan the array, find the proper socket descriptor and send(). Similarly to send to all, walk through the array and use each individually. I hope this hints at how to break up the application into modules.

Approach C) CHALLENGE ROUND! You can use threads if you want. This makes the logic much simpler, if you understand threads.

Some of these pages will give you a leg up on some of the code you need to write. REMEMBER that if you use any of these references you MUST acknowledge where you got the code/ideas from. In addition, you must write your OWN comments explaining what's going on in your code. You aren't allowed to just lift the code off the internet and call it your work.

Start here: https://beej.us/guide/bgnet/

Other sources:

I have not vetted any of the work below. Don't get lost down the rabbit hole looking at code you don't understand. Use these as a guide and ask for help when you get stuck.

https://rosettacode.org/wiki/Chat_server#C
https://github.com/yorickdewid/Chat-Server
http://www.mathcs.emory.edu/~cheung/Courses/455/Syllabus/9-netw-prog/timeout.html
https://jameshfisher.com/2017/04/05/set_socket_nonblocking/