The server/client uses a minimal set of features from the RFC 1459 plus some irc reverse engineered in the wild. Quoted from RFC 1459 the message format is:

```
<message>  ::= [':' <prefix> <SPACE> ] <command> <params> <crlf>
<prefix>   ::= <servername> | <nick> [ '!' <user> ] [ '@' <host> ]
<command>  ::= <letter> { <letter> } | <number> <number> <number>
<SPACE>    ::= ' ' { ' ' }
<params>   ::= <SPACE> [ ':' <trailing> | <middle> <params> ]

<middle>   ::= <Any *non-empty* sequence of octets not including SPACE
               or NUL or CR or LF, the first of which may not be ':'>
<trailing> ::= <Any, possibly *empty*, sequence of octets not including
                 NUL or CR or LF>

<crlf>     ::= CR LF
```

My server only implements: connecting, join a channel, part a channel, quit the server, privmsg a channel or a nickname. My client implements the same as the above except without privmsg directly to another nick.

What is supposed to be in this document? I don't know. The only RFC I have experience with is rfc 1459. It is pretty clear that rfc 1459 was written before it was known that engineers need clear written communication skills. The RFC is wrong in multiple place, vague and ambiguous or just simply lacking important information. I learned a lot of the irc protocol by using wireshark to watch the messages and reverse engineer them. The labs certainly came in handy.

It was about the 70th hour mark into the project I thought I might have misinterpreted the assignment. I didn’t make a new protocol. I just used rfc 1459. So, I will describe why I chose to use a already written protocol instead of making my own.

There were many aspects of designing a protocol that I thought was out of my grasp. The Internet Engineering Task Force handles protocol proposals very seriously. Each proposal has to be thoroughly tested. With this in mind I didn’t think I could create a good protocol with through experimentation and testing. Perhaps this assignment wasn’t about writing a good protocol though, perhaps is was just about making one that worked. I didn’t consider that until writing this document.

Here are some examples of protocol design considerations I knew I didn’t know enough about to make my own protocol. I happened to know that irc uses a message terminator “\r\n”, so one design aspect is should my protocol use a line terminator or not? What are the benefits of using one? Don’t know. What are the benefits of not using one? Don’t know. I learned at least one benefit of using a line terminator while implementing the irc server. One recv call by the server can be more than one send call by the client.This set me back significantly while making the server. I don’t know how I would have solved this if I was making my own protocol. I could make my protocol use multiple ports for an out-of-band control system. That would make it pretty simple to implement a file send feature, But what if the control messages where significantly desynchronized with the data port? Well I could merge the ports back into one and

TCP would manage getting the order correct, but if I sent a log of transmissions the log would contain control data and the server would get confused. I could escape the control information in the payload section of a message, but that would require a lot of processing.

There are just a few design considerations I don't know the right answer to, and I know there are many more I am unaware of. So I didn't make my own protocol. If I had made my own protocol that was naive, but sufficient, the server and client probably would have been much easier to write. Here are some design considerations of the actually server and client implementation.

For the whole project, I had some development goals. I wanted to use modern c++, cmake to handle the build system and design individual modules, test driven design and OOP. There are a few flaws in each goal like naked new calls in the client for pointer upcasting, include_directory in my CMakeLists.txt which is frowned upon now, and when my application became multithreaded testing and OOP became nearly impossible. Those thing are not network stack related though so I will move on.

For the server, I started with trying to learn BSD sockets because I am making this to build and run on ubuntu linux. The man pages for socket programming are almost useless. The examples from the class slides didn't build and were very difficult to read. Eventually I found a nice guide at https://beej.us/guide/bgnet/html/multi/index.html. This guide didn't talk about the flags parameter on accept and recv, so I used blocking calls to accept and recv and made my server multithreaded to handle the different blocking calls. Towards the end when I had a working server I reviewed the grading sheet and noticed the client and server need to disconnect gracefully. That mostly happened, but the server couldn't actually close the socket because of the blocking recv call. close() and shutdown() don't force the call to unblock. One solution I found for this was that I could use a flag to make the call non-blocking which allowed the server to close the sockets properly. However, I had a multithreaded system that relied on blocking calls. Now the reads that just read form the socket were basically spinning continuously. My patch solution here was to just a sleep call so its a throttled spinning. If I could start over, I would change just about everything.

For the client, I wanted a lot of thing, but the server took more like 3 weeks to make, which left me with very little time for the client as I also had to catch up on homework due the last week. The client is very buggy, but sort of works. I'm not sure I would start over on the client if I could but rather just give it a little more time to fill it out and add polish.

This project was the first start-to-finish application I've ever really made. Another reason I didn't make my own protocol. It shows as well, there are many parts I'm not proud of. But I was betting I couldn't build a working irc server and client in the time allocated either, and I did.