CS447 : Networks and Data Communications

Programming Assignment #02

Total Points: 150

Assigned Date : Thursday, June 03, 2021
Due Date : Tuesday, June 15, 2021 @ 10:59:59 a.m.

Now that you've gotten your feet wet on *Socket Programming*, let's try to get some experience in implementing a protocol to its RFC specifications.

Overview

Your second programming assignment is to **implement an Internet Relay Chat (IRC) application** using the basic socket API. The IRC protocol specification is spread over several (relatively) short RFCs, that you will need to (at least) skim through. Given the less than 2 week window however, you are only expected to implement a subset of fully-functional core IRC features that support both `one-to-one` and `one-to-many` communication (through IRC channels). The assignment objectives are:

a. to build up on your basic socket programming experience from the first assignment;
b. to further reinforce the concept of "*protocol*" using hands-on programming;
c. to learn how to read and understand RFCs; and
d. to understand the TCP concurrency management model

Back Story

Haddock invested his fortune from Red Rackham's treasure on an international ocean freight company. While business is booming, Haddock is having all sorts of communication issues with and between his captains. "*Blue blistering barnacles!!*". Screamed Haddock. "*These Sea-gherkin Carrier Pigeons are not reliable at all*". Professor Calculus, after over hearing Haddock cursing and kicking around the furniture in frustration, (again) approaches him with a solution. "*Let me build you a relay chat network Haddock. You'll never have to deal with the carrier pigeons again. You can chat with any of your captains when you want, and they can also chat with any other captains as they wish.*". Calculus said proudly. "*Everyone will know about everyone's whereabouts all the time*".

Technical Requirements

1. Provide both `one-to-one` (RFC #2810 Sec. 5.1) and `one-to-many` (RFC #2810 Sec. 5.2.1) relay chat functionality in your implementation. *(It is advised to skim through the whole RFC #2810 first to get the big picture idea behind the IRC architecture*: https://tools.ietf.org/html/rfc2810

2. Enable `client-to-server` communication by implementing the following Connection Registration (#RFC 2812 Sec. 3.1) messages: `NICK`, `USER`, `MODE`, and `QUIT` (*RFC #2812 describes the IRC Client protcol*: https://tools.ietf.org/html/rfc2812)
3. Enable `server-to-server` communication by implementing the following Connection Registration (#RFC 2813 Sec. 4.1) messages: `SERVER`, `NICK`, and `QUIT` (*RFC #2813 describes the IRC Server protcol*: https://tools.ietf.org/html/rfc2813)
4. Enable `one-to-many` communication by implementing the following Channel Operation (#RFC 2812 Sec. 3.2) messages: `JOIN`, `PART`, `TOPIC`, and `LIST`
5. Implement `PRIVMSG` (#RFC 2812 Sec. 3.3) message for sending messages to other users or channels.
    - Closely related to the above message types are their corresponding command responses (RFC #2812 Sec. 5.1) and error replies (RFC #2812 Sec. 5.2). At minimum, I expect you'll need the following reply codes: `001`, `002`, `004`, `301`, `322`, `323`, `401`, `403`, `404`, `431`, `432`, `433`, `461`, `501`, `502` (*though additional reply codes may be necessary*). Each message (items 2–5 above) has a list of applicable replies listed in the RFC, thus, use the RFC to guide your design decisions appropriately.
6. Both `client-to-server` and `server-to-server` communication should be over TCP.
7. Trivially, more than one user should be able to chat with any number of other users (either *one-to-one* or through *channels*) at the same time.

Functional Requirements

1. IP addresses/hostnames and port numbers should not be hard coded. They are provide at runtime through configuration files.

    - Your server executable will accept runtime arguments through a configuration file as follows: `./server server.conf`

server.conf

```
[SELF_NICK]
PORT=

[REMOTE_NICK] //remote server #1
IP=
PORT=

[REMOTE_NICK] //remote server #2
IP=
PORT=
```

      Here, `SELF_NICK` is the server's own `NICKNAME` enclosed in square brackets (for easier parsing). The `REMOTE_NICK` block identifies remote server socket addresses. You many duplicate the remote server block as needed. It is not necessary for these nicknames on the `.config` files to match the `NICK` command arguments in `server-to-server` communication above. We will deal with authentication in PR03.
    - Similarly, your client executable will two runtime arguments through a configuration file as follows: `./client client.conf`

client.conf

```
SERVER_IP=
SERVER_PORT=
```

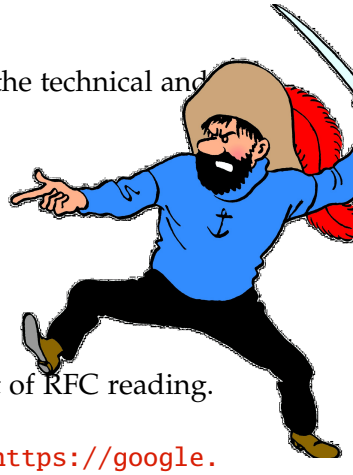
A single client only directly communicates with the server its connected with. This are considered the default execution behavior for the servers and clients. Any deviations should get approved by the instructor first.

2. All clients should exit gracefully. Server process is permitted to be forcefully killed.
3. We will test **several** clients and servers with different permutations of `nicknames`, `channels` and `topics`. Properly and adequately test and document (with screenshots) as many different scenarios as time permits. (*See RFC #2810 Sec. 3 for an interesting case study*)

    - Have a look at the following page for some *(very nicely illustrated)* sample interactions. **ABSOLUTELY** do not use any of the code listed in this page or any of the parent pages (or any other online source).
      http://chi.cs.uchicago.edu/chirc/irc_examples.html.

4. You may consider a file-based server management strategy (for users, channels, and topics). This is not a requirement, but likely the most efficient approach given the short deadline. If you do decide on a file-based strategy, make sure to **PROGRAMATICALLY** create any files and/or folders.
5. While not required, you may find parsing messages for their arguments a lot more efficient using the *`regex`* API of your programming language, rather than simple boolean logic.
6. Your client(s) and server(s) should be able to run on two separate end systems. You should be able to spawn up to 10 different container instances on the `zone server`, which should be plenty for this project. Bare minimum, try to mimic the case study listed in RFC #2810 Sec. 3 as good baseline.
7. At the end of your implementation, you should be able to:

    - Compile and run your code in a linux machine. Include a `README` file with clear compilation instructions and any non-standard Linux software requirements (if you used any).
    - Run your server program(s) first.
    - Run several clients and connect to server(s).
    - Perform both `one-to-one` and `one-to-many` chat functionality while meeting the technical and functional requirements listed above.
    - Exit the client(s) gracefully.



Instructions

- **Start early!!**. It goes without saying that this is a loaded assignment with a fair bit of RFC reading.
- **Take backups of your code often!!**.
- Follow a good coding standard. Use one of Google's coding standard found here https://google.github.io/styleguide/, if you don't already follow one.
- Your code must compile and run on a typical Linux setup. Neither the instructor nor his grader will use (or entertain the use of) any IDE to test your implementation. Be sure to test command line compilation and run before submission.
- First, carefully read **RFC #2810 Internet Relay Chat: Architecture** https://tools.ietf.org/html/rfc2810 to familiarize yourself with the lingo and to get an overall understanding of how IRC work.

  - You are not expected to implement **`operators`** or **`services`**. Simply focus on **`user`** `clients`
  - Feel free to have a look at `http://chi.cs.uchicago.edu/chirc/irc.html` as it has a more visual (and bit clearer) presentation of IRC than RFCs. Stay away from any code that you see in that website. Also, your project specifications significantly different from the one's found on this website. *Don't get caught trying to retrofit their code to this project.*
- At minimum, fully read all RFC sections relevant to the technical requirements. You may find reading other sections (as necessary) will only strengthen your understanding. Refer the RFCs before defaulting to the instructor. I will not entertain questions whose answers are found on the RFC.
- Spend adequate amount of time on planning and designing your server management strategy.
- If you end up having to make your own RFC based design choices that's not explicitly explained here, make sure to properly justify and document them in your report *(so we know why you did what you did)*. Be descriptive with proper citations/references to the RFC sections. You are more than welcome to bounce your plan/idea off the instructor before diving in.
- **Absolutely do not** include executables, folders created by your programs, hidden files, version control repositories, or any other testing related files in this tarball. All project relevant file formatting standards (**`PDF, README, .email, .txt, .tar.gz`**) will be strictly monitored and is subject to penalties.
- The due date of this assignment is **Tuesday, June 15, 2021 @ 10:59:59 a.m.**. A Moodle dropbox will be opened for submission.

Deliverables

A complete solution comprises of:

- A short report of the design and implementation of your system. The report should be **`PDF`** format. At a minimum, your report should include the following sections:
  - Introduction: Your objective and what you hope to gain from the assignment.
  - Overall design, specific design choices, and reply codes used.
  - The output of a sample run. Include plenty screenshots wherever applicable. In situations were we can't verify expected behavior, your screenshots maybe considered for partial credit.
  - Summary and Issues encountered. What you were able to achieve from your own objectives (from the introduction) as well as project specifications. Make sure to explicitly list functionality you failed to implement (or buggy).
- A compressed tarball that contains:
  - a directory containing (only) your source code and config files. **Do not** include executables, folders created by your programs, or any other files not specifically listed here as required.
  - A short `README` file with compilation and run instructions.
  - A `makefile` (**mandatory**) to compile your code especially if it involves compiling multiple executables with flag options. Python based submissions should use the `makefile` to echo the content of your `README` file.

To create a compressed tarball of the directory source, use the following command:
`tar -zcvf siue-id-pr2.tar.gz source/`.
e.g. `tar -zcvf tgamage-pr2.tar.gz PR02/`

Collaborating on ideas or answering each other's questions is always encouraged. Most times, I find that you learn a lot from your peers. However, do not share/copy/duplicate code from others, online sources, or self-plagiarism from previous attempts. The exercise is meant for you to learn network programming, not to test your googling abilities. Issues related to academic integrity and plagiarism have **ZERO** tolerance.

Useful Resources

- Linux Man pages – found in all linux distributions
- Beej's Guide to Network Programming – A pretty thorough free online tutorial on basic network programming http://beej.us/guide/bgnet/output/print/bgnet_USLetter.pdf
- Internet Relay Chat: Client Protocol RFC #2812 https://tools.ietf.org/html/rfc2812
- Internet Relay Chat: Architecture RFC #2810 https://tools.ietf.org/html/rfc2810
- Internet Relay Chat: Channel Management RFC #2811 https://tools.ietf.org/html/rfc2811
- Internet Relay Chat: Server Protocol RFC #2813 https://tools.ietf.org/html/rfc2813
- The University of Chicago χ-Project http://chi.cs.uchicago.edu/chirc/irc.html