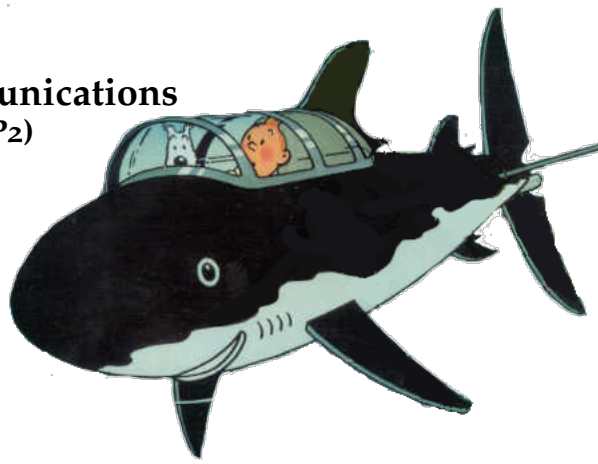
CS447 : Networks and Data Communications

Programming Assignment #2 (P2)

Total Points: 150



Assigned Date : Thursday, October 17, 2024
Due Date : Thursday, October 31, 2024 @ 01:59:59 p.m.

Overview

Your second programming assignment is to implement an **Internet Relay Chat (IRC) application** using the socket interface. Ever used *Discord* or *Slack*? At its core, such apps share fundamental similarities with IRC, a classic real-time communication protocol that powers many chat platforms. This assignment gives you the chance to peek under the hood and build your own simplified version of a system like Discord, exploring key networking concepts like client-server architecture, text-based communication, and handling multiple concurrent connections.

While the full IRC protocol description is quite extensive (and has evolved over several RFCs), this assignment focuses on building a simplified IRC server that supports some of the most essential commands. Given our 2-week window, it's not even realistic to implement a fully-fledged IRC client. Instead, you will primarily focus on building a robust and efficient IRC-based chat solution that can handle a bustling crowd of users, all while gracefully managing the flow of messages and ensuring everyone stays connected. In doing so, you will gain valuable experience in:

a. Able to Proficiently read and interpret RFC specifications, using RFC 2812 (Internet Relay Chat Protocol) as a guide/case study.
b. Implement an application using both TCP and UDP, developing a nuanced understanding of their strengths, weaknesses, and appropriate use cases.
c. Master techniques for handling multiple clients concurrently
d. Design and build a complete client-server system, solidifying your understanding of network communication principles.

Back Story

"*Thundering typhoons!*" roared Captain Haddock, sending a stack of shipping manifests flying. "*This blasted carrier pigeon arrived three weeks late! The Unicorn nearly collided with a tanker in the fog because of this delay!*" He paced the deck of his flagship, the *Sirius*, fuming over yet another communication mishap plaguing his burgeoning shipping empire. Lost shipments, missed client meetings, and now a near disaster – all thanks to the unreliable pigeon post.

Professor Calculus, ever attentive to his friend's woes, emerged from his cabin with a twinkle in his eye. "*Haddock, my dear friend,*" he announced, "*I believe I have the perfect solution to your communication conundrum! I've been studying this fascinating new technology called IRC, Internet Relay Chat! Imagine, a network where your captains can instantly send messages to each other, join specific channels for different topics like 'Atlantic Crossings' or 'Caribbean Trade Winds' and even have private conversations!*"

Haddock, initially skeptical, perked up at the prospect of ditching the troublesome pigeons. "*Blistering barnacles, Calculus! If this IRC contraption can prevent another near-collision, I'm all for it! But it better be more secure than those*

gossiping seagulls! I don't want our rivals intercepting our shipping routes!"

With a mischievous grin, Calculus beckoned Haddock towards his cabin. "*Come, my friend*," he said, "*Let me show you the wonders of Internet Relay Chat!*"

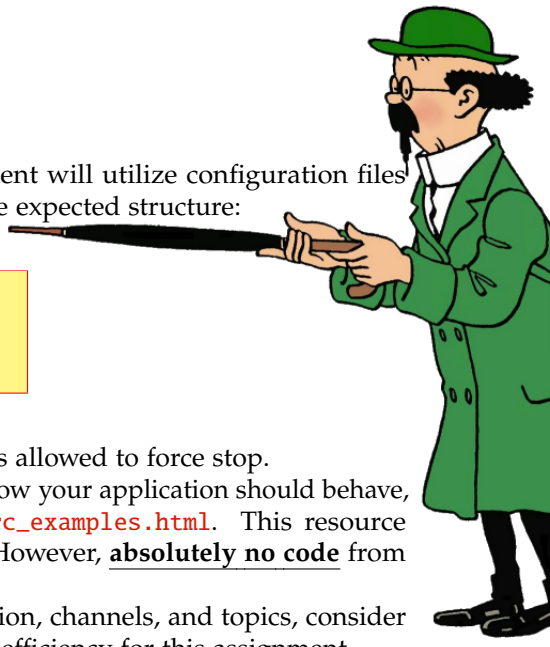
Technical Requirements

1. **A Tale of Two Protocols**: Your IRC application will utilize both TCP and UDP for communication.
   - **TCP**: Employ TCP for the core IRC functionality, ensuring reliable delivery of messages and commands between `clients` and the `server`.
   - **UDP**: Utilize UDP for "*out-of-band*" communication, such as:
     - **Heartbeat Signals**: Implement a heartbeat mechanism where the server periodically sends UDP packets to clients to check if they are still active. Clients should respond with a corresponding UDP packet to confirm their presence. This helps the server maintain an accurate list of active users and efficiently manage resources.
     - **Server Statistics**: The server should periodically broadcast server statistics using UDP. These statistics could include the number of connected users, the number of active channels, CPU usage, or any other relevant performance metrics. Clients can utilize this information to monitor the server's health and performance.
     - **Real-time Notifications**: Implement a system for delivering real-time notifications to clients using UDP. These notifications could include alerts about new users joining a channel, urgent messages, or server-wide announcements. This adds a layer of immediacy and responsiveness to your chat application.
2. **Server-Side Focus**: Develop a robust single server IRC application capable of handling multiple clients concurrently. `Server-to-server` communication is outside the scope of this assignment.
3. **Protocol Conformance**: Adhere to the protocol grammar specified in **RFC 2812, Internet Relay Chat: Client Protocol** (`https://tools.ietf.org/html/rfc2812`), Section 2.3.1. Utilize regular expressions (`regex`) for efficient message parsing.
4. **Command Support**: Implement the following core IRC commands:
   - **Connection Registration (Section 3.1)**:
     - `NICK`: Allows a user to change their nickname.
     - `USER`: Registers a user with the server.
     - `MODE`: View or change user or channel modes.
     - `QUIT`: Disconnects a user from the server.
   - **Channel Operations (Section 3.2):**
     - `JOIN`: Allows a user to join a channel.
     - `PART`: Allows a user to leave a channel.
     - `MODE`: View or change channel modes.
     - `TOPIC`: View or change the topic of a channel.
     - `LIST`: List all channels or channels matching a given mask.
     - `NAMES`: List all visible users on a channel.
   - **Messaging (Section 3.3):**
     - `PRIVMSG`: Send a private message to a user or channel.
5. **Response Codes**: Implement the following numeric reply codes as defined in Section 5 of RFC 2812:
   - `001 (RPL_WELCOME)`: Welcome message upon successful connection.
   - `002 (RPL_YOURHOST)`: Server host and version information.
   - `004 (RPL_MYINFO)`: Server information.
   - `301 (RPL_AWAY)`: User is marked as away.
   - `322 (RPL_LIST)`: Channel information (from LIST command).
   - `323 (RPL_LISTEND)`: End of channel list (from LIST command).

- `401 (ERR_NOSUCHNICK)`: No such nickname exists.
- `403 (ERR_NOSUCHCHANNEL)`: No such channel exists.
- `404 (ERR_CANNOTSENDTOCHAN)`: Cannot send message to channel.
- `431 (ERR_NONICKNAMEGIVEN)`: No nickname given.
- `432 (ERR_ERRONEUSNICKNAME)`: Erroneous nickname.
- `433 (ERR_NICKNAMEINUSE)`: Nickname is already in use.
- `461 (ERR_NEEDMOREPARAMS)`: Not enough parameters provided for command.
- `501 (ERR_UMODEUNKNOWNFLAG)`: Unknown user mode flag.
- `502 (ERR_USERSDONTMATCH)`: Cannot change mode for other users.

You may implement additional reply codes to enhance your server's functionality, but provide justification for their inclusion in your report.

6. **Client Implementation**: Develop a simple text-based `client` to interact with your server. The `client` should be able to connect to the `server`, send commands, and display received messages.



Functional Requirements

1. **Configuration Files**: Similar to the first assignment, your server and client will utilize configuration files (`server.conf` and `client.conf`) to specify runtime arguments. Here's the expected structure:

server.conf

```
TCP_PORT=
HEARTBEAT_INTERVAL=
STATS_INTERVAL=
```

client.conf

```
SERVER_IP=
SERVER_PORT=
```

2. **Graceful Exit**: Implement graceful exit mechanisms for `clients`. Server is allowed to force stop.
3. **chi IRC**: To gain a better understanding of IRC interactions and visualize how your application should behave, explore the chi IRC project at http://chi.cs.uchicago.edu/chirc/irc_examples.html. This resource provides nicely illustrated examples of IRC commands and responses. However, **absolutely no code** from this or any other online source is to be used in your implementation.
4. **Data Management**: For managing server-side data such as user information, channels, and topics, consider using a *file-based approach*. This strategy offers a balance of simplicity and efficiency for this assignment.
5. **UDP Port Allocation**:
   - **Client**: Each `client` must dynamically allocate a UDP (listening) port during startup using a fixed offset (e.g., adding `1000` to its assigned TCP port). For example, if the `client`'s assigned TCP port is `50000`, its UDP port would be `51000`.
   - **Server**: The `server` does not need to explicitly define a UDP port because it doesn't need to listen for incoming UDP connections. It will calculate each client's UDP port by adding the same fixed offset to the port number reported by the client's TCP connection. The `server` should maintain a mapping of nicknames/clients and their udp port numbers for out-of-band communication.
6. **Case-Insensitive Commands**: The server should be case-insensitive when processing commands. For example, `/join`, `/JOIN`, and `/JoIn` should all be treated as the same command.

Logistics

1. **Starter Code**: Feel free to leverage the starter code from the first assignment as a foundation for this project. However, you'll need to adapt it to handle both TCP and UDP connections as described in the Technical Requirements section.
2. **Testing Environment**: Leverage the *zone server* during testing to mimic real-world IRC behavior using it's interconnected container network topology.

3. **Testing and Documentation**:
   - Thoroughly test your application with various scenarios, documenting your process and results with screenshots.
   - Ensure your code compiles and runs on a Linux machine. Use `Makefiles`. Provide a `README` with clear compilation instructions in case `make` fails.
4. At the end of your implementation, you should be able to:
   - Configure the `server.conf` and `client.conf` files with the appropriate parameters.
   - Compile your code using the `Makefile` and create `server` and `client` executables.
   - Run your IRC `server` program first.
   - Run one or more IRC `clients` to connect to the `server`.
   - Perform various IRC chat functions, such as joining channels, sending private messages, and changing nicknames.
   - Simulate a `client` disconnecting abruptly and observe the server's response to the lack of heartbeat replies.
   - Verify that `clients` receive and display the `server` statistics correctly. Use the `/quit` command in the client to initiate a graceful exit and observe the server's response.

Instructions

- **Start early!!** This assignment is a multi-faceted adventure involving substantial RFC reading and intricate implementation. Starting early is crucial to avoid a last-minute scramble.
- Begin with a minimum viable solution, gradually adding complexity and features. Maintain good version control habits throughout the development process. **Start simple, finish BIG!**.
- Adhere to a consistent coding standard. If you don't have a preferred one, the Google C++ coding standard `https://google.github.io/styleguide/` is recommended.
- Begin by reading **RFC 2810, Internet Relay Chat: Architecture** (`https://tools.ietf.org/html/rfc2810`) to gain a foundational understanding of IRC concepts and terminology.
- Then, thoroughly read and understand all relevant sections of **RFC #2812 Internet Relay Chat: Client Protocol** `https://tools.ietf.org/html/rfc2812`
- Plan and design your server's architecture and data management strategy carefully. Document any design decisions you make, especially those based on your interpretation of the RFCs, with clear explanations and citations.
- Make design decisions within the boundaries of the RFC specifications. If unsure, consult the instructor for clarification.
- DEADLINE: **Thursday, October 31, 2024 @ 01:59:59 p.m.** through Moodle. Email submissions are not honored.

Deliverables

A complete solution comprises of:

1. Report (**`pdf`**):
   - Introduction: Your objectives for the assignment and what you hope to learn.
   - Design: Explain your overall system design, key choices you made, and the protocol's reply codes.
   - Sample Run: Include screenshots showcasing a typical interaction with your system. These can help illustrate functionality and potentially earn partial credit if certain features aren't fully implemented.
   - Summary and Issues: Summarize your achievements and explicitly list any unimplemented or buggy functionality.
2. Compressed Tarball (**`siue-id-p2.tgz`**):

- Source Code Directory: Include all your C++ source code. `config` files are not required.
- `Makefile`: A Makefile is required, especially if your project involves multiple executables or compilation flags.
- `README`: Provide clear instructions on how to compile and run your code, in case `make` fails.

Use the following command to create the compressed tarball:
`tar -zcvf siue-id-p2.tgz source/`.

(Replace `siue-id` with your actual SIUE email login (e.g. `tgamage`) and `source/` with the name of your source code directory (e.g. `p2`))

Your code should be a testament to your abilities, not a copy of someone else's work. Collaborate, don't copy! Learning from peers is great, but copying code (from classmates or online) is strictly prohibited. Cite any external resources you use for ideas, and then implement those ideas in your own way.

This assignment is a challenge designed to push your boundaries and foster growth. Embrace the learning process and don't hesitate to seek help when needed. Remember, the goal is to master these concepts, not just complete the assignment. Plagiarism, whether from classmates, online sources, or AI tools, stifles learning and compromises academic integrity. The instructor actively uses MOSS http://theory.stanford.edu/~aiken/moss/ to check for software similarity. Plagiarism has severe consequences including and will result in a failing grade. No exceptions.

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