Systems Programming Project 3

April 16, 2024

1 Introduction

For our final project, you will be implementing a peer-to-peer chat program. This is a non-trivial program; our base repository for the project is of meaningful size:

```
===============================================================================
 Language            Files        Lines         Code     Comments       Blanks
===============================================================================
 Markdown                1           59            0           33           26
 TOML                    3           58           43            4           11
-------------------------------------------------------------------------------
 Rust                   25         3740         2540          560          640
 |- Markdown            21          246            0          225           21
 (Total)                           3986         2540          785          661
===============================================================================
 Total                  29         3857         2583          597          677
===============================================================================
```

You can accept the GitHub classroom assignment (which is forked from the template repository) at `https://classroom.github.com/a/tB4hTcTC`.

Please note the `README.md` file in the repository; it has instructions on how to run some of the binaries. You should go ahead and update this `README.md` as appropriate for your submission.

This project is an iteration on the 2022 class project, and has undergone some refinement and refactoring, however, it is still not "perfect." In fact, the current template repository's code does not include the fancy UI as I need to finish updating that for newer library versions.

It will be iterated on again for future classes. With this in mind, there are definitely rough edges, places where the code is confusing or verbose, and *many* places where the code can be improved in terms of both ergonomics (e.g., the API) as well as refactoring (making more structs, renaming things, etc.).

ASK QUESTIONS IF YOU DON'T UNDERSTAND SOMETHING IN THE CODE! It might be that it's something that I can easily explain, something that I'm aware of but just didn't have time to refactor/fix, or it might be a real problem/bug in the code. I wrote all of this, so I should be able to help you figure out what is going on.

2 Project Description

This project is about building a peer-to-peer chat client. We have seen what a *very* basic chat server looks like. While our chat server had tons of limitations, it also ran just fine on a single thread using async programming. For each client that connected, we started two tasks: one that received data from the network (i.e., data from whatever client was sending us) and a second that received data from the server (in our case, a broadcast channel that the server set up when a client connected). Note that while both of these tasks ended up interacting with the network, they didn't have to. This is an important observation because it means we can launch tasks for a bunch of different purposes, and as long as we can reduce the amount of blocking code in those tasks, we can very efficiently scale.

2.1 Scaling with async

As a concrete example, I write a lot of data collection code (i.e., crawlers) for my research. This is fundamentally not a CPU bound problem because it involves: 1) a lot of network traffic from the Internet and 2) relatively loose deadlines. The design pattern I use is, at its core, a producer/consumer system where different "jobs" are run for each chunk of the logic that drives the crawling process.

I use an open source server that scheduling of jobs. I.e., handing out jobs to a set of waiting "workers" that process those jobs. The server has a protocol to communicate with clients, but it is agnostic about how that client works and how jobs are actually executed by workers.

I have my own client, written in Rust. It is relatively simple, and handles a lot of things that are relevant specifically to crawling social media data, e.g., rate limiting. At its core, it manages a collection of long lived workers, where each worker runs in its own *task*. My library can scale to tens of thousands of concurrently executing workers, on a single thread.

However, there is relatively little computationally expensive code involved (and when there is, that is farmed out to a secondary service). It takes microseconds (less?) to do whatever data manipulation and logic it takes to control the crawling process; it takes *milliseconds* to make a request to a web site. Thus, the overwhelming majority time a worker is executing a job it is waiting for some I/O request to get data. While any particular job might be "slow" to execute, we can run so many of them concurrently that our *throughput* scales way higher.

3 bing2bing

`bing2bing` is the name of our peer-to-peer chat system. Our protocol is going to be more complicated than the simple chat server we made in class. It will support several *commands*, and instead of being simply line oriented, we will use a higher level abstraction for our wire protocol: JSON.

3.1 Frames

A valid frame in `bing2bing` can be composed of the following types:

- Text – a string.
- Error – an error.
- Number – an unsigned 64 bit number.
- Bulk – raw bytes.
- Bool – true/false.
- Array – an array of frames.
- Float – a 64 bit floating point.

3.2 Supported Commands

A command is a logical abstraction. they get encoded/decoded into frames and sent/received over the wire.

We will support the following commands:

- Broadcast – deliver data (i.e., arbitrary bytes) to all peers in the network.
- Ping – allows peers to measure latency between them.
- Pong – a response to a Ping command.
- Register – register with a tracking server.
- Say – send a chat message to all peers in the network.
- Deliver – deliver data (i.e., arbitrary bytes) to a specific peer.
- Announce – a command that is propagated through the network to provide peers knowledge about the network topography. I.e., this is how peers let each other know who they are connected to.

- Whisper – send a chat message to a specific peer in the network.
- Extension – an arbitrary protocol extension. Peers that know how to handle a specific extension will handle them, peers that don't know how to handle them will forward/broadcast on to the rest of the network.

4 Grading

Grading will be similar to the style of Projects 1 and 2.

- -1,000 points: If you do not update `CREDITS.md` with the names of your group members, as well as an honest break down of the work each group member did, you will receive **NEGATIVE ONE THOUSAND** points. I.e., it would be next to impossible to get anything higher than a zero on this project.
- -1,000 points: If running `cargo fmt` results in any change to your repository, you will get **NEGATIVE ONE THOUSAND** points. Be sure to run `cargo fmt`!!!!!!!!!
- -1,000 points: No warnings should be produced when running 'cargo check' or 'cargo clippy'.
- -1,000 points: If you do not do a demo (only one group member needs to be present for a demo), you will receive -1,000 points.
- -1,0000 points: If your implementation breaks a basic peer (i.e., it won't work with the base project template clients), you will lose -1,000 points.
- 1,0000 points: If your implementation breaks a basic peer *for a good/justifiable reason* (as judged by Jeremy), you will get +1,000 points. I.e., you will be at net zero points for breaking things instead of -1,000.
- 50 points: Program compiles, as well as the items noted below as required for graduate groups.
- 10 points: Implement at least one extension.
- 20 points: **REQUIRED FOR GRADUATE GROUPS.** Implement *routing* of messages. Instead of just broadcasting commands that have an expected destination (e.g., `Ping`, `Deliver`, `Whisper`), use a peer's knowledge of the network topology to route a message via shortest path.
- 5 points: Make things fully decentralized. Our base project will make use of a distinct *tracker* to let peers boostrap their first connection. It is not necessary for this to be a separate program. Implement handling of the `Register` command in your peer, thus allowing any other client to bootstrap off your peer instead of having a stand alone tracker.
- 5 points: Comprehensive documentation. Points will be determined by looking at the documentation built when running `cargo doc --document-private-items --no-deps`.

- 10 points: Performance analysis. E.g., how many connections can your peer handle? Are there any overlay network topologies that could cause performance/robustness issues? How well can the network handle churn (i.e., peers joining/leaving a lot)?
- 5 points: Proper error handling. There will be at least *some* `unwraps()`s in the base project. Also, right now, most of the error handling is via very weakly developed custom error types that could be made *much* better. Fix this by adding proper error handling (e.g., with the `thiserror` crate) and nice error messaging. Points will be determined by how comprehensive error handling is.
- 10 points: Fancy UI. There are a lot of opportunities here!
- 10 points: Refactoring, code clean up, and ergonomics. Clean up the code and make it easier to use! Be sure to summarize what you have done here in a `CHANGELOG.md` file in your repository.
- 5 points: Add meaningful logging. Using the `tracing` crate to generate logs. Points will be determined on comprehensiveness of logs, as well as the usage of different log levels.
- Unlimited points: Cool factor. Points are determined subjectively by me, but if I find anything about your project to be particularly unique, difficult, clever, etc., you can get some extra points. Note that this applies to other items above. E.g., if you do something cool with an `Extension`, you will get cool points in addition to the 10 points for implementing an `EXTENSION`.
- Unlimited points: Bug hunting. If you find a bug in the template code, and can provide a reproducible report to Jeremy, you will get some points.

In addition to the above, you must update the README.md to summarize what you did and give any and all instructions on how to run things! I'm going to go off what's in the README.md to grade, so make sure you test all the instructions, etc.

4.1 Allowed Crates

Generally speaking, you are free to use any crates listed on `https://blessed.rs`, *except* networking crates. I want networking to be handled online with things from the Rust standard library and tokio.

Any crates not on Blessed.rs need to be approved by Jeremy.

Be sure to note what crates you use in your `README.md`!

4.2 Due Date

All code must be committed to your repositories by Thursday, May 7th at 23:59 PM.

Time slots for live demos will go up the last week of April.

4.3 Academic Honesty Statement

In addition to the code in your repository, you *must* submit an academic honesty statement as per the class syllabus submitted via Brightspace. If you do not upload this academic honesty statement, you will receive a *ZERO* on the assignment.

REMEMBER THAT IF YOU USE A MACHINE LEARNING TOOL THAT IS CHEATING AND YOU WILL FAIL THE CLASS!