peas-rf-cp.exe

Erik Rimskog, Patrik Johansson,
Alexander Eriksson, Staffan Annerwall

October 29, 2023

Contents

1 Introduction

This document is the final project report of our efforts to build a decentralized room-based chat system in rust for the course Computer Networks and Distributed Systems (1DT102) at Uppsala University, taken in the fall of 2018.

2 System design

The system contains two main items: clients and trackers. The trackers are not complicated; their sole purpose is to act as a look-up table for new clients joining rooms. Most complexity is instead limited to the clients and in particular how clients interact with each-other.

2.1 Clients

Also called *nodes*, clients are the main components of the system and represent individual users on the service. The clients are connected directly to each-other without any servers in-between, and it is this network of connected nodes that comprises the chat rooms.

2.2 Trackers

Trackers exist mostly because of utility needs and are not central to any room. The main purpose of a tracker is to store information about which clients acts as entrances into rooms, so called bootstrap nodes. The procedure is that a client asks a tracker for bootstrap nodes into a given room, and the tracker returns a collection of available clients, if any. The asking client then attempts to join the room using one of these bootstrap nodes.

2.3 Rooms

The room files work similar to torrent files; they contain a 64-bit random ID and the name of the room. If two clients have the same room file then they can talk to each other. The main reason for going this route is to prevent room name collisions. Two rooms can have the same name but have different IDs.

2.4 Communication channels

To prevent a fully connected graph between the nodes we are using a method heavily inspired by a distributed hash table (DHT) implementation called Kademlia[1]. The main

[1]See `https://en.wikipedia.org/wiki/Kademlia`.

idea is to use a distance metric to limit connections between nodes to those that are considered close in that metric. The route taken by Kademlia and us is to interpret the bit-wise XOR operation on unsigned integers as the distance between two nodes, where the integers are the node's ID's.

When a network is established messages are sent by each node to all its neighbours, and that to all its neighbours and so on. Every node remembers the most recent messages they have broadcasted, so they don't broadcast the same message twice.

When nodes disconnect, the network could split into two disconnected partitions, which of course is a problem. So to combat this every node is periodically pinging another node that it knows about. That node has to then send a response back through the broadcast network. So if the original node receives a response on this network withing a time frame, it knows that it can reach that node and everything is probably fine. But if it doesn't receive a response, it establishes a new connection with the pinged node to repair the network.

UDP is used for everything, simply because everything is a one packet request with a one packet response. The overhead TCP would give is too high in this application.

2.5 External libraries

Our initial idea while discussing the implementation was to use as few libraries as possible, even going so far as to planning to use only one external library (which was for building the user-interface). As the project progressed however, we realized that the number of additional things to implement – while individually being small libraries – was growing to an unfeasible size in total. Adding only one library at a time, it did not come as a sudden surprise that we would need several libraries but it was rather a slow realization. In some sense we ended up writing our own library for networking tools.

Below is a list of all libraries used as of the time of presentation and demonstration (2018-12-13), sorted alphabetically:

bincode (1.0.1) Rust-specific compacting serializer and deserializer, used for sending data between clients. Implements the interface/facade from Serde.

clap (2.32.0) Responsible for building the command-line argument structure and parsing the arguments given.

cursive (0.10.0) Text-based user interface front end library for Rust. The back end is `ncurses` from the GNU project.

flexi_logger (0.10.1) An implementation of the common logging interface from `log`.

log (0.4.6) Standardized logging interface without implementation. Actual logging is implemented in flexi_logger.

pnet (0.21.0) Low-level networking library.

rand (0.6.1) Pseudo-random number generation. This is used to generate ID's, to pick an element at random from a set of values, and other things.

serde (1.0.82) Serialization and deserialization interface. Actual serialization is implemented in Bincode.

serde_derive (1.0.82) Utility for automated serialization and deserialization implementations.

All libraries are taken from Crates, the Rust community library registry available at `https://www.crates.io`.

3 System usage

3.1 Running a tracker

```
./tracker
```

The tracker takes no command-line arguments and has no interaction with the user once started. Running the executable starts a tracker on port 12345.

3.2 Running a client

A client can be run in two different modes; one mode for creating a new room, and one mode for joining a room. Creating a new room generates a file with a room name and ID. Joining a room requires that a room file for that room is locally available.

3.2.1 Creating a room

```
./client --new-room <ROOM_NAME>
```

Command-line arguments:

`--new-room` The name of the room to create.

3.2.2 Joining a room

```
./client -j <ROOM_FILE> -u <USER> -t <TRACKER>
```

Command-line arguments:

`-j, --join` Determines which room to connect to. This must be the name of a file created by the application using the `--new-room` argument (see Section 3.2.1).

`-u, --user` Specifies the username associated with this client. Messages sent from this client will be displayed together with this username. The value can be any string of UTF-8 characters.

`-t, --tracker` This must be an IPv4 address and port of a running tracker on the format `x.x.x.x:port`.

4 Reflections

4.1 Library usage

We decided to not use a library for networking, we thought that it would be more fun to do it manually. We discovered that this was a mistake because we basically made our own library, and that took a lot of time. So we should have used some sort of high-level library like Tokio or something similar from the start instead.

To be able to send packages using low-level interfaces that operated on bytes we needed a library for serialization, and those were Serde and Bincode. Serde is just a generic interface for serialization and Bincode is an actual implementation of one serialization algorithm. We also used this for creating the files we had.

The library Pnet was used for searching the network interfaces on the local computer to find a candidate that seemed to be connected to some network. This was needed because we needed to know which interface to open the ports on.

4.2 Redesign

4.3 Progression

Initially the work plan for the project was to clearly define interfaces between the different components of our code. Work on these interfaces took a lot of effort and a lot of time was spent arguing over how it should be done, ultimately they were never fully completed. The project still managed to stay afloat, thanks largely to good communication within the group as well as frequent meetings and group programming sessions.

4.4 Different approaches

4.4.1 Room files

In the current build of the program you are required to have a room file in order to join an existing room. This approach works but is of course not optimal since you have to send a file to someone before being able to talk to them, effectively forcing you to use another program to send the room file first before being able to chat using our program. If we had more time we would probably have chosen a different design where you could connect without a room file. For example, you could make a program where you use the IP-address of the rooms host to connect or something similar.

4.4.2 Programming language

In hindsight our choice of programming language might not have been the best from an efficiency standpoint. Only one person in our group had previous knowledge of Rust and

thus the rest of the group had to spend time learning it before we could start working. We also realized fairly quickly that even though rust has a lot of useful libraries and documentation, it would probably have been easier to use a programming language that had been used for similar applications before.

For example it was quite hard to find other chat program implementations in rust which forced us to come up with a lot of the solutions by our selves instead of taking inspiration from other programs made before. This was probably for the best though since we learned a lot from the experience. In the end we recognize that this project would have been easier to do in a more established programming language like C++, however if we were to do it again we would still use Rust since the fun part of the project for us was to learn something new.

4.4.3 Group work

After finishing the work on the project we realized that the whole process would have been far easier if we had been able to agree on details in the beginning rather than spending a lot of time discussing which solution was better. In hindsight, a lot of things could be improved about the way we approached the project and our work ethic as a group. More planning and starting earlier would certainly have helped in making sure that we finished the project on time.

4.5 Enhancements

4.5.1 User interface (UI)

There are multiple things in the program that could and should be improved it it were to ever be released for the public. The first and most obvious thing that needs optimizing is the UI. Since the point of the project was to deliver a distributed system, the UI was not our highest priority and thus we did not spend a lot of time on it. In the programs current state the UI works but it is not very intuitive to use and the colour scheme is practically just gray.

If we were to continue working on the program it would probably still be a terminal application, however the colours could be changed to something that is more inviting and easier on the eyes instead of just being gray. The overall layout of the UI in the terminal could also be improved to make it more user friendly.

4.5.2 Chatting

Disregarding the looks of the program, the actual chatting in and of itself works just fine however there are multiple things that we could add and implement support for. Unicode characters, pictures and GIF's are just some examples of things that would be fun to implement. You could even argue that supporting these things would probably be necessary to compete against other modern chat programs.

4.5.3 Joining rooms

As mentioned previously the process of joining existing rooms is both time consuming and inefficient. To further enhance the program we would make the "room joining" procedure simpler and more user friendly, either by removing the concept of room files completely, or by making it implicit, that is to hide the handling of room files from the users and instead let the program take care of it in the background.

4.5.4 Universal Plug and Play (UPnP)

Right now we do not have support for UPnP, so users can not connect to anyone behind a NAT. This means that everyone has to be on the same LAN in order for this program to work.

4.5.5 Trackers

Right now we only support having one tracker set at the beginning of the program. Ideally, each client should be able to seamlessly switch between multiple ones, or even use multiple ones at the same time. We also thought about having a list of local trackers that can be expanded and configured on a user-by-user basis, and having each room file contain a small list of trackers where bootstrap nodes are supposed to exist on.