Decentralized Chat

deand17, anla119, chvim21

June 2, 2022

1 Introduction

In this project we set out to create a decentralized chat system. In a decentralized network there can not be one primary server, that exchanges data with all clients. In a decentralized network, the data is spread out across multiple servers, and is therefore more reliable in many cases. In a centralized system, a simple DDoS attack could take down the entire system, whereas in a decentralized system all the servers would have to be attacked in order to take it down. Another form of decentralization is peer-to-peer networks, also known as P2P networks. In a P2P network, two or more computers are connected to each other, and share resources without having to go through a server. In a P2P network, any node can be both client and server, or either at a time. All exchanging of information is communicated directly between the peers [ADK08a]. The P2P network architecture opens up to many possibilities, and we'll use that to our advantage for our chat system.

2 Overview

The chat system should focus on scalability and reliability, meaning that the system should be able to scale and handle a growing amount of users and resources. The system has to live up to its expectations, and send/deliver messages reliably. The requirements of the chat system are the following:

1. Users must be able to create, join and leave groups.
2. A user must be able to search for groups based on their name.
3. A user must be able to list the users of a group
4. A user must be able to search for other users based on their name.
5. Causal ordering of messages.
6. A message history.'
7. An interactive UI.

To ensure reliability we chose to use TCP rather than UDP. You expect your messages to be sent without issue in a chat system, and UDP can not ensure reliability as it is a connectionless protocol where errors are allowed and recovery mechanisms aren't present. TCP is a connection-oriented protocol, that makes sure data comes across reliably and in the order they were sent. One of the requirements for the chat system, is causal ordering, which is something TCP can help us achieve. It gets rid of possible failures, so we assume no failure model.

3 Methodology

Our strategy is as follows:

1. To approach the problem, we first decide what programming language we wanna use. When a programming language has been decided, we need to find libraries that can help us set up the necessary network.

2. The next step is to set up a basic P2P network and prepare it for the list of features. To do this, we use the previously found programming language and respective libraries. Once the network is set up and prepared, we'll begin to plan how to implement each feature. We'll describe our thought process and ideas, and eventually our decision on how it is going to be implemented.

3. When a development plan has been created, we'll start to implement the different features, not necessarily in the order specified above, but in an order which makes sense to us. We will explain in detail how the implementation works.

4. Lastly, we will test our chat system, look for bugs, document our findings and discuss them. Hereunder, we will explain what could have been done differently and why. We finish our project off with a conclusion.

4 Tools

To help us develop a P2P chat system, we used python as it is very well suited for quick prototyping. Furthermore, we used a python library called Twisted. Twisted is a networking engine developed in python, for python, making it easy to implement custom network applications. The Twisted library is the main component and backbone of the project, as it gives us the possibility to set up our P2P network.

Since many of the requirements for the chat system include some sort of searching, we wanted to make sure the search mechanisms worked smoothly. We used the python library "prompt-toolkit", which offers an autocomplete function that makes searching easier. It will give the user a list of possible users based on a string the user has typed. It uses the client's input as a substring to find potential matches.

Lastly, we use a handful of other libraries, but amongst them the most important are "socket" and "threading". The socket library helps with translating hostnames into IP addresses, and threading is widely used throughout the code, but is mainly used for handling messages. Every time a message is received with TCP, a thread will start and process the message. When a message is sent, it is locked by a condition until it is finally released and delivered. The thread will wait for the condition to be met, and once it's released the thread can let all the other waiting threads know that the condition has changed. The threads work together to ensure causal ordering. The causal ordering mechanism follows algorithm 6.3 from the book "Distributed Computing" by Ajay Kshemkalyani and Mukesh Singhai [ADK08b]

5 The P2P network

The network constructed by peers of the program is very simple. Usually larger P2P networks require structured overlays and data indexing. Having such indexing and overlays makes it easier for peers to find the files they are looking for, whenever they are querying. Since our network is a simple chat and peers will not exchange large amounts of data over a broader network, we can use an unstructured approach. Unstructured overlays in P2P networks are efficient when data replication takes place, which it does in our case, as all TCP packets are nearly identical. Large networks can also lead to scalability problems [ADK08c], however, scalability has not specifically been taken into account.

In order to simulate a network, we use Docker. Docker allows us to create a private network, enabling us to have each container running an instance of our program simulate a peer. This allows us to have each peer have its own ip-address and to specify port numbers to listen on.

6 Implementation

In this section we give an overall description of the main parts of the implementation. The implementation consists of the following files and classes.

* *main.py* : The entry point of the program.
* *controller.py* : Consists of the Controller class and is what glues the program together.
* *factories.py* : Consists of the ServerProtocolFactory and ClientProtocolFactory classes, which are responsible for constructing incoming and outgoing connection objects respectively.
* *protocols.py* : Consists of the DiscoverProtocol and the ServerProtocol and ClientProtocol classes. The responsibility of the former is network or peer discovery and the responsibility of the latter is interacting with the transport layer of the network stack.
* *causal.py* : Consists of the CausalOrderManager class, which is responsible for enforcing causal order delivery of messages
* *chat.py* : Consists of the Chat class, which is responsible for taking input from the user.

We now give additional details on each of the above classes in turn, with the aim of giving an impression of the overall architecture of the program, starting with the *Controller* class.

6.1 Controller

The *Controller* class has the overall responsibility of the program and glues everything together. Its main responsibility is managing the local data structures of a peer, that is, keeping track of connections to other peers, user- and group names, as well as keeping a message history for each group that the user is a member of. The *Controller* is also responsible for initiating the *DiscoverProtocol* to identify other hosts on the network such that connections can be established. The management of the local data structures also means that the *Controller* is responsible for updating these data structures whenever a peer joins or leaves the network as well as when a user joins or leaves a group. In addition, the *Controller* acts as a delegator other classes can call, ensuring that tasks are dispatched to the appropriate places.

6.2 Factories

The sole responsibility of the *ServerProtocolFactory* and *ClientProtocolFactory* classes are to construct incoming and outgoing connection objects respectively, where each connection essentially is a reference to a *ServerProtocol* or *ClientProtocol* instance. This abstracts away most of the details with respect to sending messages, as all that is needed to send a message is referencing the appropriate connection object.

6.3 Protocols

Moving on to protocols, the DiscoverProtocol is a simple protocol for finding other peers on the network. The protocol works by having each peer generate a random real number in $\{0, 1\}$ and then broadcast that number. Let *P1* be a peer receiving a broadcast message from *P2*, and let *r1* and *r2* be the random numbers generated by *P1* and *P2* respectively. Upon receiving the broadcast message from *P2*, *P1* then compares the received random number *r2* to its own random number *r1*. If $r1 < r2$, then *P1* initiates a connection to *P2*. Otherwise, *P1* sends *r1* back to *P2* to allow *P2* to initiate the connection. The reason for sending *r1* to *P2* is that the broadcast of *P1* may have happened long before *P2* joined the network, meaning *P2* knows nothing about the random numbers at other

peers. Returning to connection objects, these are represented by the *ServerProtocol* and *ClientProtocol* respectively and are mainly used as an interface for passing messages to and receiving messages from the transport layer. To minimize coupling, received messages are passed to the *Controller* for processing, which in turns hands them off to the CausalOrderManager.

6.4 Causal Order Manager

The CausalOrderManager implements one of the main requirements of the project, namely causal ordering of messages, by implementing the algorithm by Kshemkalyani–Singhal, [ADK08b], for optimal causal ordering of messages. The implementation is effectively a one-to-one implementation of the pseudo-code. As described in the algorithm, the send and recv procedures are intended to happen atomically, however, since messages may arrive out of order, there is an asynchronous aspect to the recv method, there must be a way to wait for a certain condition to be satisfied. In python, we achieve this by using a Condition from the threading module. Besides acting as a locking mechanism, a Condition also allows a thread to specify a condition that it wishes to wait for to be satisfied. The thread then releases the lock and starts to wait. Other threads can then do work and upon finishing, waiting threads are notified that changes have been made and can check to see if their respective condition is now satisfied. After atomically processing a message, it is either sent to the appropriate peer or in the case of delivery, passed to the *Controller*, which handles final delivery of the message.

6.5 Chat

Lastly, we have the Chat class, which acts as the interface to the program. The user interface is a simple command line interface, with a prompt allowing the user to write messages to everyone or specific users and groups, as well as providing commands allowing a user to create, join and leave groups, list users and groups and search for users and groups by name. The following is a list of all available commands.

* -h or -help Show a help message
* -q, -quit or -exit Exit the program
* -lu or -listusers Show currently logged on users
* -lg or -listgroups Show groups available to join
* -delay $< delay > < message >$ Allows sending of an intentionally delayed message.
* -finduser $< user >$ Search for the user with name $< user >$
* -findgroup $< group >$ Search for the group with name $< group >$
* -creategroup $< group >$ Create a new group with name $< group >$
* -joingroup $< group >$ Join the group with name $< group >$
* -leavegroup $< group >$ Leave the group with name $< group >$
* @$< user >$ Send a direct message to the user $< user >$
* @$< group >$ Send a group message to the group $< group >$

Commands start with the dash '-' symbol and targeted messages with the at '@' symbol. On a more technical note, the prompt uses the "prompt-toolkit" module, which provides the nice feature of ensuring that arriving text is not printed on top of whatever input the user is currently typing. The prompt however, runs in an asynchronous blocking loop, causing it to be delegated to its own thread to avoid blocking the main thread.

As a final note regarding the implementation, for the sake of simplicity with respect to managing the network and passing of messages, we decided that it should not be possible to leave the 'all' group. In other words, this causes everyone to be connected to everyone else, making the network a complete graph. Thus when joining, a peer automatically becomes a member of the 'all' group.

7 Tests

We tested out all of the features we implemented. The first one we wanted to test, was the command that lists all the users in the network.

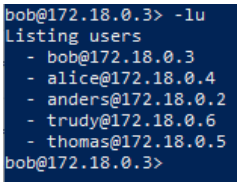

Figure 1: Testing command "-lu" for listing users

Bob has joined the network, and has been given a unique IP-address. Bob wanted to see who else were on the chat, so he used the command *-lu*, which stands for "list users". By running the command, he is presented with a list of users.

The next command we tested was creating groups, joining them, and listing them.

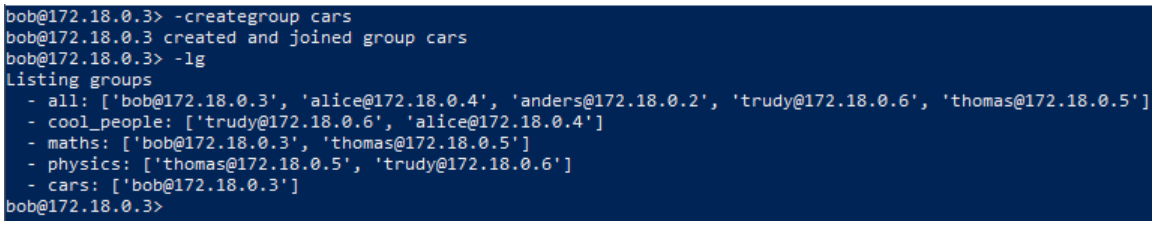

Figure 2: Testing commands "-creategroup" and "-lg" for creating and listing groups

As seen above, Bob creates a group called "cars" by running the command *-creategroup cars*, and then he runs *-lg* to list all other groups. As shown in Figure 2, all groups are displayed, and you can see the group "cars" which Bob is the only member of. The list shows all groups currently in the network, and displays which members are apart of them.

Alice was unsure what commands she could use, so she ran the command *-h* to be presented with a list of all commands, as a helping hand.

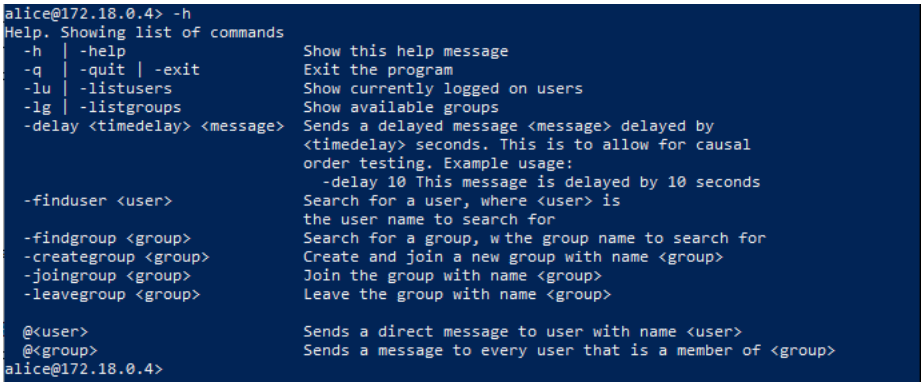

Figure 3: Testing command "-h" for the help menu

Alice decides to look for groups about maths, so she performs the command *-findgroup maths* to list all groups that has a substring of "maths".

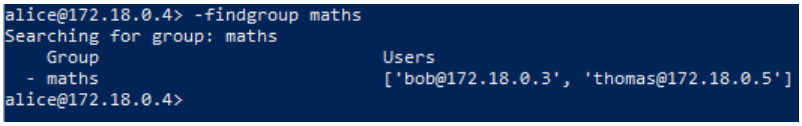

Figure 4: Testing command "-findgroup"

She finds a group called "maths" and it lists all the users that are currently members of that group.

Alice wants to see if her friend, Bob, is online, so she runs the command *-finduser bob*

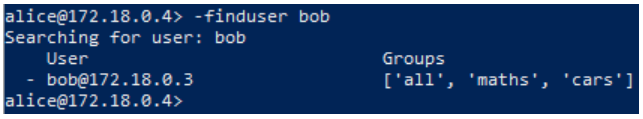

Figure 5: Testing command "-finduser"

There was a successful match in her search for Bob, which means he is online at this current moment. It also shows which group chats he's apart of.

8 Discussion

All of our commands work, and we succeeded in implementing all the required features. However, there may still be bugs in the code, and the overall user interface is not the most intriguing as it is just a simple command prompt window. For further development, it would be a priority to implement a graphical user interface with buttons instead of having to run commands yourself. For the message history they only show the last 5 messages sent to make it simple. This only works as the proof of concept. The messages are stored on local memory, so for further development, perhaps another solution would be smarter.

9 Conclusion

To conclude, we were tasked with creating a decentralized chat in which scalability and reliability took precedence. In addition to this, our chat had to include certain features such as causal ordering of messages, the ability to create, list and search for groups and users in a group chat, message history and an interactive UI. We created a P2P chat in python, using Twisted, where we implemented the aforementioned features. The benefits of a decentralized chat, our methodology, code review as well as discussion of possible bugs and their solutions is outlined in the report.

References

[ADK08a] Mukesh Singhal Ajay D. Kshemkalyani. *Distributed Computing: Principles, Algorithms, and Systems*, chapter 18, page 677. Cambridge University Press, 2008.

[ADK08b] Mukesh Singhal Ajay D. Kshemkalyani. *Distributed Computing: Principles, Algorithms, and Systems*, chapter 6.5, pages 206–215. Cambridge University Press, 2008.

[ADK08c] Mukesh Singhal Ajay D. Kshemkalyani. *Distributed Computing: Principles, Algorithms, and Systems*, chapter 18, page 681. Cambridge University Press, 2008.