SOFTWARE ENGINEERING

PROJECT REPORT

P2P MESSENGER APPLICATION

Roshin Rasheed
NYU ID: N15764851
Net ID: rr3474

Vinay Kumar
NYU ID: N12876792
Net ID: vk1176

Table of Contents

1. INTRODUCTION

The internet has transformed the way that people communicate with each other. Where once we would have had to wait days, or even weeks, to send a simple message to a loved one, today we can send anything from text messages to video files instantaneously to anyone on the planet, and even have live video conversations with them, all at the click, or touch, of an icon. Thanks to the internet, we now possess the power to communicate with whomever we choose, at the tips of our fingers.

The advent of the internet has brought about various technologies for communication, with vastly different underlying principles, from social networks to instant messengers, from e-mails to video chatting applications. One of the most popular kinds of internet-based communication is the online chat, i.e. instant messengers and messaging applications such as Whatsapp. By 2014, messaging apps had more users than social networks. Most of these applications provide features such as one-on-one chat, group chat, voice and video calls, and file sharing. The technical architecture of such applications can be either peer-to-peer (direct point-to-point transmission) or client-server (where a centralized server retransmits messages from the sender to the receiver).

For our project, we aim to build a messaging application that follows the peer-to-peer, or p2p, architecture.

2. PROJECT OVERVIEW

The aim of this project is to build a messaging application that allows two or more people connected to the same network to communicate via text messaging over the LAN. We will be following a p2p architecture, so we will not have any centralized server that reroutes messages between the peers. Being a real-time communication system, all peers involved in a chat session must be online simultaneously.

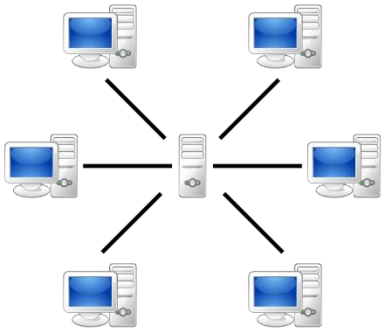


Fig.1. Client-Server

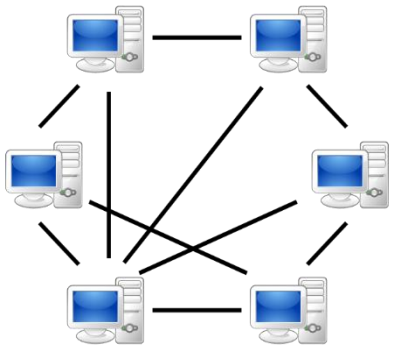


Fig.2. Peer-to-Peer

The application will prioritize speed of communication and ease of use, and will therefore assume a minimalist approach with regards to the user-interface and added functionalities.

3. PROJECT MANAGEMENT PLAN

3.1. Project Organization

The first part of the software design cycle was to understand and define the functional and non-functional requirements of the application. Since this was a relatively small project, no software development tool was used. Only freely available open source tools and libraries were used. The various use cases were identified and condensed into a class diagram. The code was written in JavaScript and executed and tested using Nodejs. The GUI of the application was designed to run on the user's web browser.

3.2. Life Cycle Model

We followed the evolutionary prototyping software development methodology, in the interests of quick design and coding, and to

improve our understanding of the requirements along with the project devolopment. We ended with two versions, or prototypes, of the application. The first, primitive prototype, had a command line interface. This was improved upon with a graphical user interface in the second prototype, making it more hassle-free and user-friendly.

4. REQUIREMENTS SPECIFICATIONS

4.1. Functional Requirements

Since a p2p connection is used, there is only one type of user, namely the peer. All peers will have access to the following functionalities:

- User login
- View list of peers
- Request connection to peer
- Accept connection from peer
- Save contacts
- Send message
- Receive message
- Broadcast message
- Close connection
- Menu

Use Cases:

4.1.1. User Login

- User is prompted for username.
- User enters username.
- User is logged onto network.

4.1.2. View peers on the network

- User requests list of peers connected to network.
- System provides the list.

4.1.3. Request/Accept connection

- User requests a connection to a contact (peer).
- System checks for the peer within network.
- If found, the contact is notified of connection request. Else, failure message displayed.
- The contact accepts or rejects connection request. If accepted, connection is established and chat enabled between the two. Else, failure message displayed to user.

4.1.4. Send/Receive messages

- User types a message and sends it to recipient.
- System checks if connection has been established. If yes, the message is sent to recipient, and displayed on his system. Else, error message displayed.

4.1.5. Close connection

- The user selects log out option from the menu.
- All connections to the user are closed.

4.2. Non-Functional Requirements

4.2.1. Security

Since a p2p network does not have inherent security features, high standards of security should be guaranteed at the application level.

4.2.2. Privacy

The system should ensure that messages sent by the user can be read only by the intended recepient. This can be done through end-to-end encryption.

4.2.3. Speed

The application should ensure that messages are received instantaneously, in real time, with a lag of at most 1.5 seconds.

4.2.4. Data Integrity

The system should ensure that sent messages are received in full by the receipient, without corruption, and that packets are not lost along the way. This is done by using the TCP protocol.

4.2.5. Robustness

The system should be robust to loss of network connection at both sender and receiver ends, too many connected peers, and other non-idealities, and print appropriate error messages.

4.2.6. Graphical User Interface

The system should have a user-friendly and minimalist GUI that works over a web browser, and allows the user to view his connections, type messages into a message bar, send messages, and view received messages.

4.3. Operational Requirements

4.3.1. Browser

The application will use the browser for the GUI. It should run on any of the commonly used browsers, such as Google Chrome Mozilla Firefox and Microsoft Edge, so no browser-specific code should be used.

4.3.2. Operating System

The system shall run on any commonly used operating systems, such as Windows 8 and above, Linux, and MacOS.

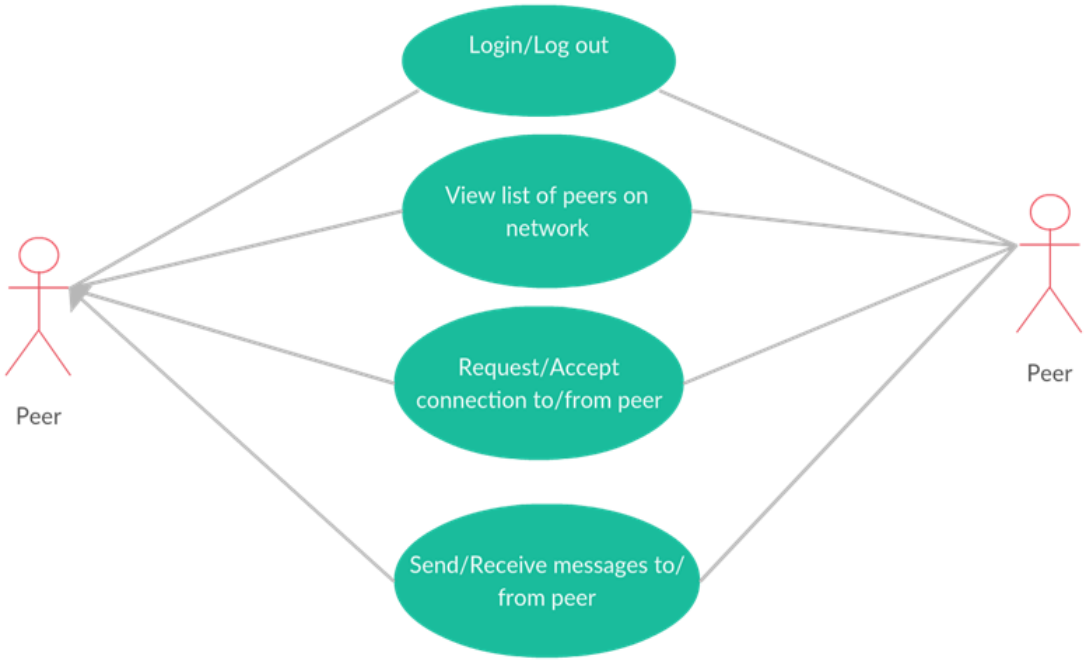
5. ARCHITECTURE

We followed a p2p framework for the application, which made the architecture much easier to set up than client-server. Thus, all users of the application belong to the same category of user, namely the peer, and can be represented using a single class diagram. All interactions in the system are between two or more peers.

The GUI runs on the web server, and provides the user with a list of connected peers, and a messaging area for each of them. Each peer on the network is identified by their username. When a connection is established between two peers, their respective addresses are retained, and all messages are passed directly between them without any centralized server.

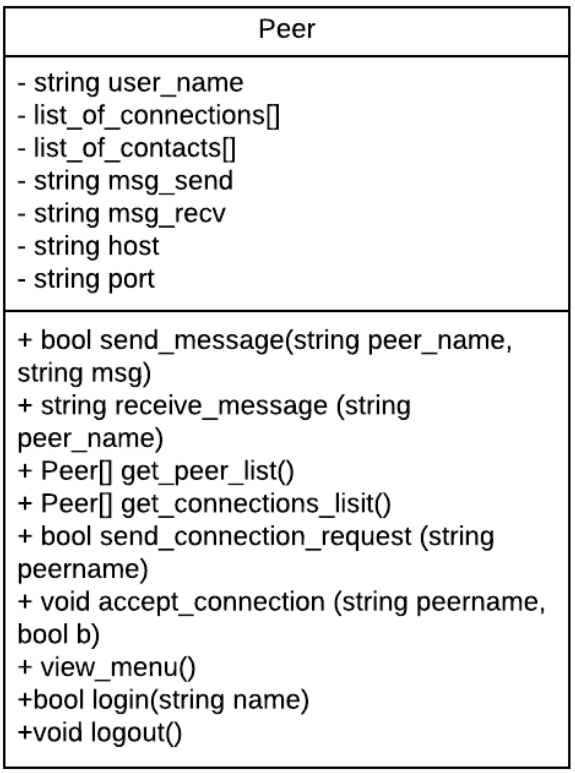
6. DESIGN

6.1. Use Case Diagram

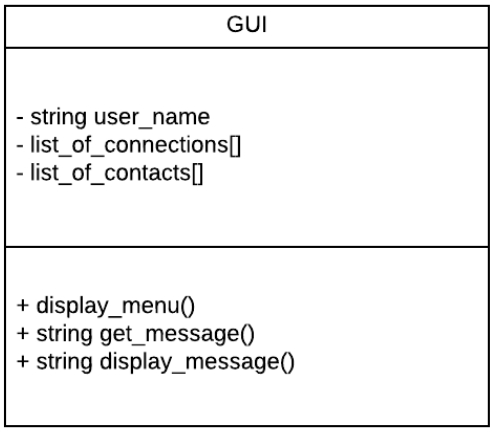


6.2. Class Diagrams:

6.2.1. Peer



6.2.2. GUI



7. IMPLEMENTATION

Many sources were consulted in the process of determining implementation protocol. At first, we consulted several python libraries, such as pyp2p, but as neither of us have experience in network programming, most of the libraries proved too complex, especially when adding GUI elements. In the end, we decided to focus on creation of an application that would fulfill our most basic and essential use cases and requirements, specifically those involving purely text based, decentralized chat between 2 or more peers. Naturally, the final version of such an application would allow communication between multiple devices, but for demo purposes we chose to focus on being able to chat locally across tabs or windows.

We decided to switch to JavaScript, as we found the web based language to be a bit simpler to understand and have more options available. Specifically, we turned to Node.js, a popular JavaScript framework that allows for quick, simple integration of various libraries and packages. These could be as extensive as Socket.IO, which was the main framework we used to manage connections and interactions between peers in a decentralized manner, or as basic as Hash-To-Port, a simple library that takes an input string and hashes it to a number able to be used as a Port code, used to assign networks.

Ultimately 2 applications were created. The first was a prototype application that functioned as a proof of concept of sorts, allowing for a terminal chat between any 2 or more command terminals. This fulfilled the most basic functional requirements we had of 1, 2, and 4, but offered little else. Essentially, it is a command line application that is run by running a JS file and providing a list of command inputs, that consists of the available usernames for the chat room, which can be claimed by anyone with prior knowledge of it from a separate terminal.

The second application was created with extensive help from following an online tutorial, and represents an idealistic but not necessarily formal version of the application. This was run in the web browser, which allowed for additional HTML and CSS elements to create a visual GUI. Details of the workings of the program can be found in comments of the code. This version of the app of course fulfilled the same requirements and allowed for the same

basic use cases, meeting base requirements, but also added a few more to the table, such as a GUI and the ability for any one to be able to use the app through any browser, as opposed to one needing a UNIX terminal in the prototype.

8. TESTING

As the application was intentionally kept simple, testing was done as the application was being built, and also after it was mostly completed. In essence, testing was simply done by testing the most barebones of features, ensuring they worked as intended, and then moving on to more complex requirements and use cases. First, the ability to achieve a connection between two peers at all, by notification using a simple message such as "New peer found!" was verified. Next, the ability to connect to a server port, a localhost port in this case, was ensured. After this, the ability to send messages between ports was tested. Naturally this was the most vital feature, so after it was tested to make sure it worked for 2 users for all messages, it was tested to work for various message lengths. After this, it was expanded to 3 users, or more. In this manner, additional features were slowly added and tested, such as the addition of usernames.

9. FUTURE PLANS

We could expand on the current implementation of our messaging application, and move on to a bigger and better version. Some of the plans that we have for the future are as follows:

- Scale to a larger number of users.
- Add functionalities to send audio, video and other files.
- Improve the GUI, and make it more robust.
- Save peers to a saved contacts list.
- Save messages to be read later, if the peer is not connected.
- Enable communication from multiple separate devices as opposed to sharing on localhost.
- Deploy the application in such a manner so as to eliminate the need for specific libraries.

10. REFERENCES

1. http://cs.berry.edu/~nhamid/p2p/index.html
2. https://softdevel.com/
3. https://nodejs.org/en/about/
4. https://pypi.org/project/pyp2p/
5. https://socket.io/blog/socket-io-p2p/
6. https://github.com/mafintosh
7. https://javabeginnerstutorial.com/author/sandip-salunke/
8. https://peerjs.com/