Chat System report

Paula Šalković, Ana Gršković

Table of Contents

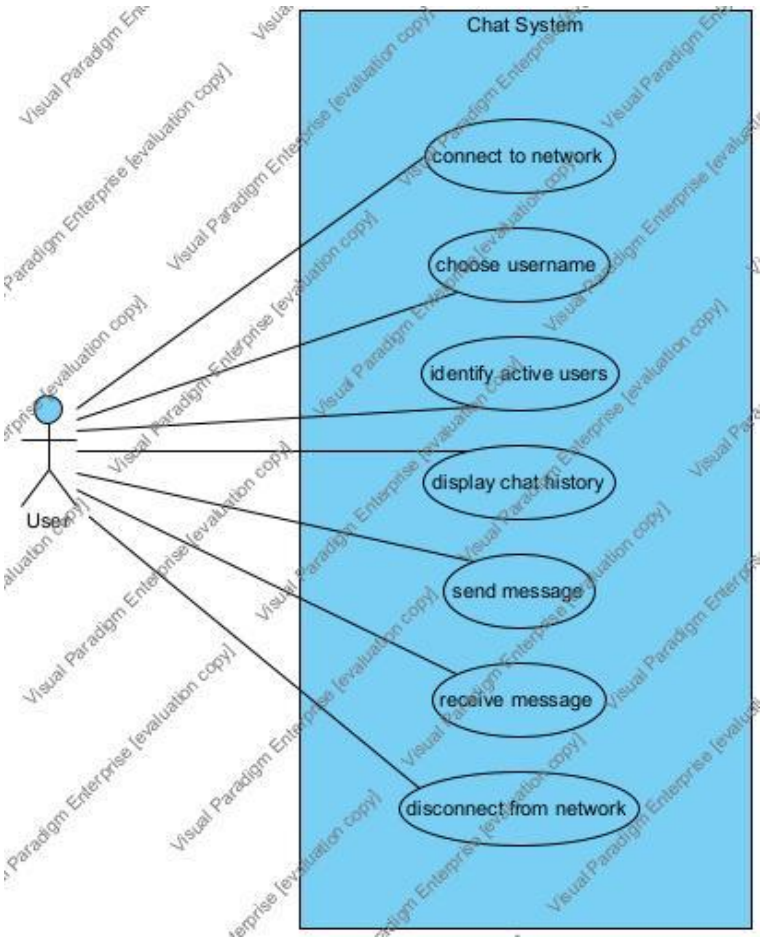
1. UML part

1.1. Actors and assumptions (if any)

In Chat System, there is only one type of actor, the simple user. This user connects to the network, discovers other users and chats with them. This user is identified by their IP address and a unique username. There is no administrator user since this is a distributed peer-to-peer system.

1.2. Use case diagrams

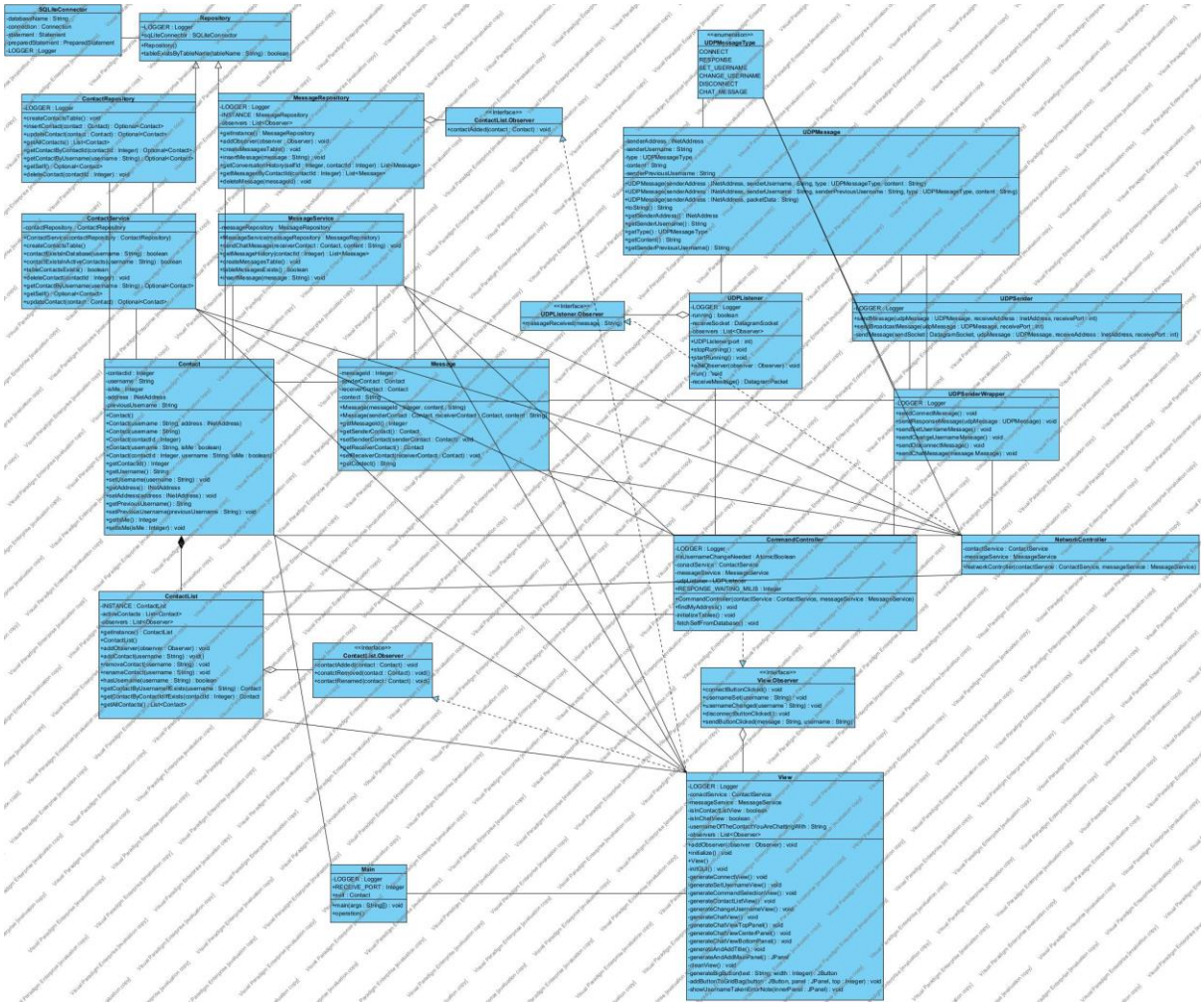
There are a few use cases in which the user can participate. The first one is connecting to the network. Once the application is ran and the GUI window opens on the user's machine, the only use case they can take is connecting to the network. After successfully exchanging the connect and response messages with the other active users in the system, based on the previous use of the system on the given IP address, the user can be asked to choose their username. Once that is done, the user is forwarded to the command selection window. From there, the user can take one of the following use cases: changing username, disconnecting or displaying a list of currently active users and picking one to chat with.



Picture 1. Chat System UML use case diagram

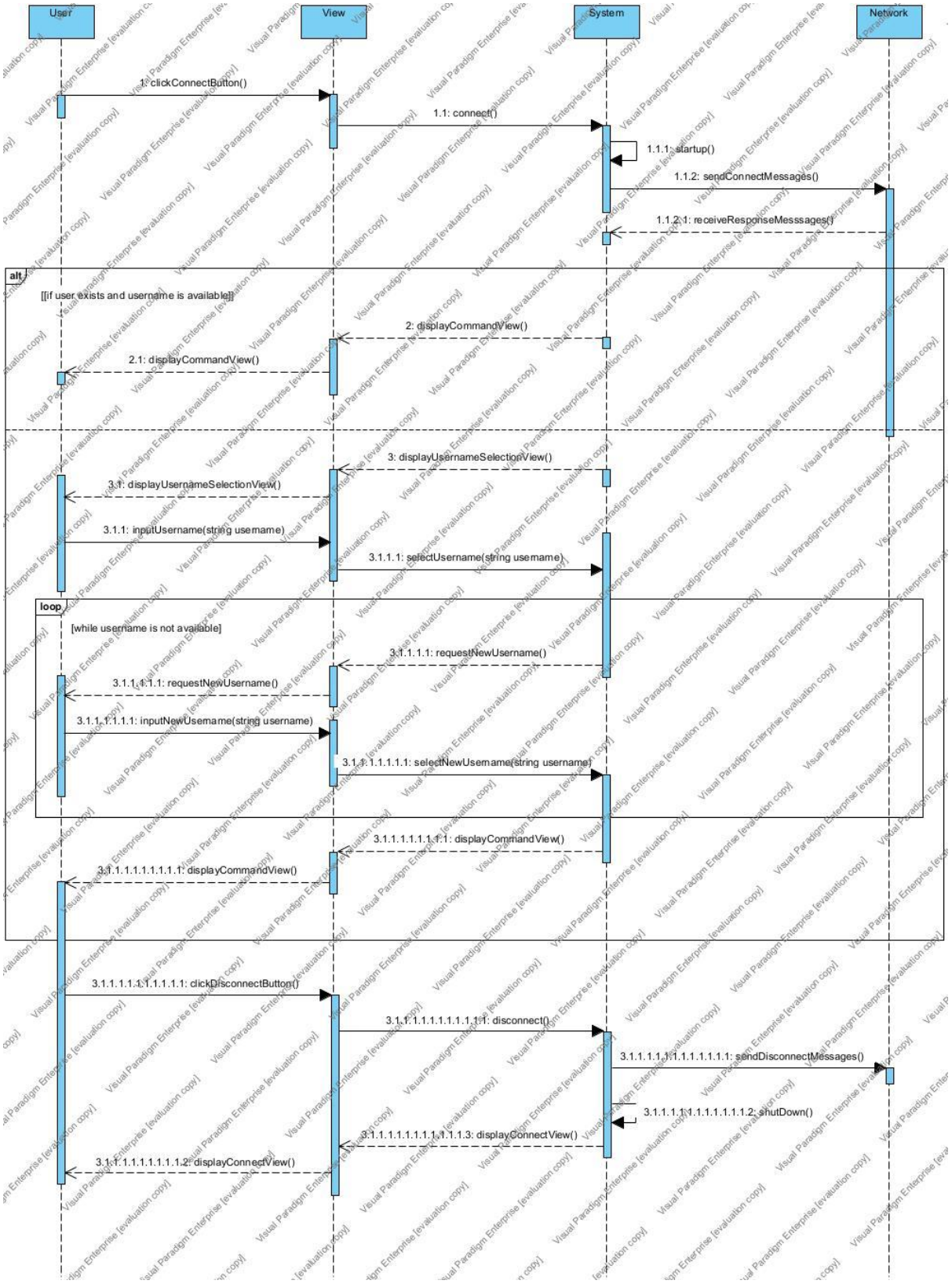
1.3. Class diagram

This system follows a simple MVC architecture, as can be seen on the class diagram below. There is one view class, two controller classes and a lot of model classes. The communication between the three components is often achieved through the Observer design pattern.



Picture 2. Chat System UML class diagram

1.4. Sequence diagrams

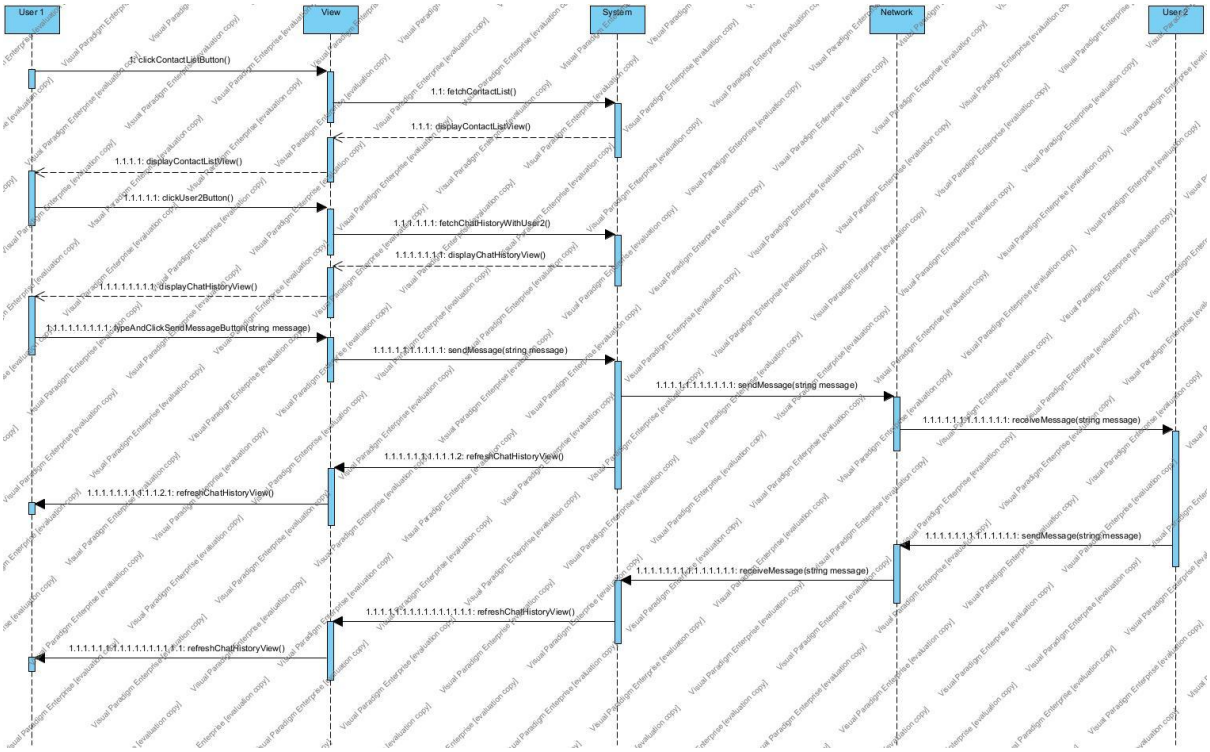


Picture 3. Chat System UML sequence diagram representing connect and disconnect actions

In the first sequence diagram, the connect and disconnect actions are presented. Once the application is ran and the GUI window opens on the user's machine, they can press the button to the

network. Then the application fully starts and a connect message is broadcasted accross the network. The application waits a bit of time to receive the response messages of the other active contacts in the network. Afterwards, the system checks if there already exists an entry in the contacts table corresponding to this user. If there does, the system also checks if that username clashes with any on the usernames of the currently connected users. If the user entry doesn't exist or it exists with a non-unique username, the user is asked to choose a username. This username is then validated. If it has successfully passed validation it is sent to other users in the network and the user will be forwarded to the command selection window. On the other hand, if the chosen username is still not unique in the network, the user is asked to choose antoher one, and so on until they choose a unique username and the user can be forwarded to the command selection window.

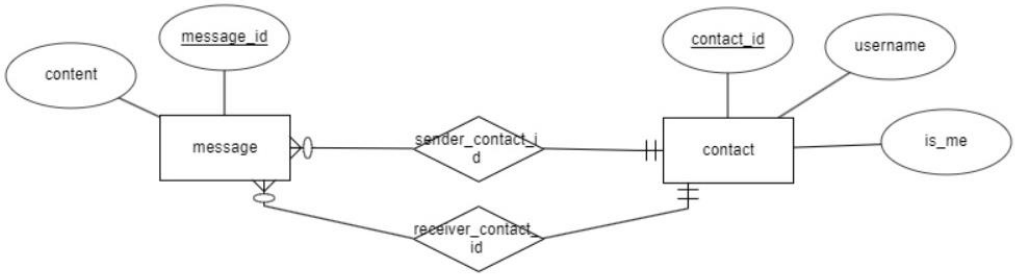
In the end, when the user wants to disconnect from the network, they can click the disconnect button present on the command selection screen. Then, a disconnect message is sent to the active contacts in the network without expecting a response, a system is shut down and the user is disconnected.



Picture 4. Chat System UML sequence diagram representing the chatting action

The second sequence diagram represents the chatting action. It begins by the user clicking on the contact list button, which triggers fetching the list of currently active contacts. It continues by the user clicking on a username of another user, with whom the want to chat. From there, two things can happen. Firstly, the user can type a new message and press the send button, which triggers the system sending a UDP message to that users machine and that user receiving the message. Secondly, the other user can also do that, and then our user receives the message. Upon message sending and receiving, the chat message history is rerendered. It is important to note that the user can receive messages from other users also, whose chat history they are not currently viewing.

1.5. Database schema



Picture 5. Chat System Entity-Relationship database diagram

Picture 6. Chat System Relational database diagram

The database for this system was modeled using SQLite, a software library that provides a relational database management system (RDBMS) and is embedded into the end program. Unlike traditional client-server database management systems, SQLite is self-contained and serverless, meaning it operates as a library linked directly to the application that uses it. Two diagrams were chosen to represent this simple database. The database consists of tho entities, contacts and messages, connected by two one-to-many relationships, message senders and message receivers.

1.6. Architecture overview

As mentioned before, the architecture of this system is MVC. The Model-View-Controller architectural pattern is a design approach widely used in software development to organize code in a way that separates concerns and improves maintainability. These elements are the internal representations of information (the model), the interface (the view) that presents information to and accepts it from the user, and the controller software linking the two. The user sees the view and uses the controller. The controller manipulates the model and the model updates the view. There is one view class, implemented in Java Swing, two controllers and a lot of model classes. Those could

technically all be joined into one, more traditional model class, but it was developer's decision to keep them separated in order to achieve the so-called separation of concerns, one of the important principles in coding. The communication between model, view and controller is often implemented by the observer design pattern. It is a behavioral design pattern that defines a one-to-many dependency between objects so that when one object changes state, all its dependents (observers) are notified and updated automatically. It is a widely used pattern for implementing distributed event handling systems or implementing communication between components in a loosely coupled manner.

2. PDLA part

2.1. Development Methodology and Project Management Approach

In our project, we used the Agile Development methodology, specifically opting for the Scrum framework. Agile, renowned for its adaptability and iterative approach, redefines the traditional linear development process by breaking it down into smaller cycles known as sprints, promoting adaptability and collaboration. Within the Agile framework, Scrum offers a structured approach to software development, dividing the project into fixed-length iterations called sprints, typically lasting two to four weeks. Key roles introduced by Scrum include the Product Owner, responsible for prioritizing features, the Scrum Master facilitating the Scrum process, and the Development Team. Since we were only two developers, we both combined the responsibilities and took on the roles of product owners and development team members. We held weekly to bi-weekly meetings to discuss our progress, covering what was accomplished since the last session and outlining plans for the next one.

Our project was divided into six sprints, each lasting between two and four weeks. In the first sprint, we focused on modeling the UDP communication between the users. In the second one, we shifted our focus on maintaining the list of active contacts, implementing the functionality for changing the username and a simple user interface. Then, in the third sprint, we worked on refactoring our code into a more suitable MVC structure, while in the fourth sprint, we managed to implement a graphical user interface, as well as the functionality for sending and receiving chat messages. In our fifth spring we implemented storing the messages in the database, and in our final, sixth sprint we added the chat window to the graphical user interface. We also improved our GUI with rerendering when a contact is added, renamed or removed as well as when a chat message is sent or received.

To efficiently plan, track, and manage our work, we selected Jira as our project management tool. Within Jira, we created user stories and corresponding tasks required to fulfill each user story. Our tasks were annotated with one of the following statuses: „*To Do*“, „*In Progress*“ and „*Done*“, which helped us track the work as it was being done. For each sprint, we utilized the Poker method to estimate and select a suitable number of user stories to be completed within the sprint timeframe. This approach allowed us to maintain a clear focus on our goals, adapt to changes swiftly, and deliver tangible progress in each iteration.

For version management, we used Git, a flexible and distributed version control system. To host our Git repository, we used GitHub. A very useful feature of GitHub for us was the GitHub Actions feature which allowed to check the build and run the tests for each of our Pull Requests. This made Continuous Integration (CI) part of our project significantly easier.