Università di Pisa

Artificial Intelligence and Data Engineering

Distributed Systems and Middleware Technologies

UniSup

Project Documentation

TEAM MEMBERS:
Edoardo Fazzari
Sina Gholami
Mirco Ramo

Academic Year: 2020/2021

Contents

1 — Introduction

UniSup is an instantaneous chat application that allows users to exchange short text messages among them.

1.1 Description

UniSup name is composed by *Uni* that stands for University, which is the main application scope; and *Sup* which is a popular slang abbreviation that stands for *"What's up?"*. Every time a **user** logs in correctly (an authentication check is performed), he/she will be able to see his/her chat history. After a click on a specific chat, he/she can visualize the list of the last messages exchanged with that particular contact. Filling the text field and clicking on the **SEND** button will send a message to the selected contact. At any time, he/she can start a new conversation with a new contact: it only requires a click on the corresponding button, typing destination username and the text Payload and click on the **SEND** button.

When a user logs into the system, he/she will receive every message sent to him/her while he/she was offline. On the contrary, while he/she is online, he/she receives messages on **REAL TIME** and the interface is automatically updated reporting the new message. Of course, messages within a chat are always displayed in chronological send order, and they are forwarded according to a **FIFO** policy.

At the application start, the user will visualize an authentication form: he/she can login with an existing account or register a new one, of course no duplicated usernames are allowed.

From the application Scene, by clicking on the **LOGOUT** button, the user logs out the system and goes back to the authentication form. The user can now login again, even with a different account.

2 — Analysis Stage

2.1 Main Use Cases

- An *unregistered user* can
  - Sign up using a non-duplicate username and a custom password
- A *registered (not logged) user* (can
  - Login using his/her own credentials
- A *logged user* (can
  - Visualize his/her list of **contacts**
  - Send a new **message** to the selected **contact**
  - Send a new **message** to a new **contact**
  - If a **message** is received, visualize it thanks to the real-time interface update
  - Logout
- The *system* should
  - Correctly forward each **message** to the correct receiver
  - Store **messages** whose destination is an offline user: those messages will be forwarded when the receiver is online again
  - Store all the **message** histories and send them to the specific **user** each time he/she logs in

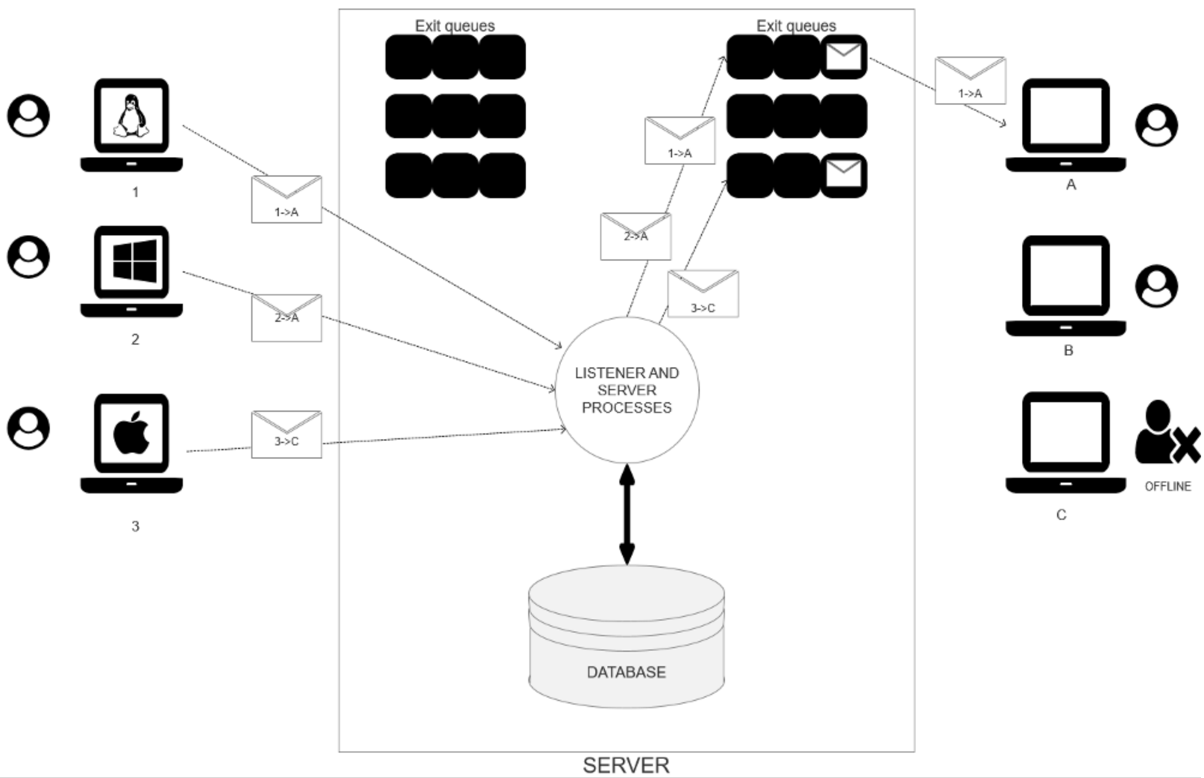
2.2 Size and Scope of the Application

As cited in the Introduction (chapter 1), the application has been designed for working within limited entities/environments, for example among close friends attending the same University. This is mainly because the selected approaches and technologies (for more details see next chapters) are not very scalable and they are suitable for a limited number of nodes. Anyway, the following properties are guaranteed:

- No message can ever be lost, regardless the fact that the receiving user is online
- The application is totally OS-independent
- The GUI provides a user-friendly experience and makes application easy to use d
- Within small clusters, the application ensures good performance

3 — Project Stage

3.1 System Architecture



As shown in the previous picture, the application is based on a client-server architecture, in which each client, in order to send a **message** to a **user**, contacts the main server which is in charge of determining receiver's physical address and forward the **message** if it is online. In the image some typical scenarios are represented to help better understand how *Unisup* works. In particular:

1. The message 1 → A is sent from the client 1 destined to the client A: it arrives at the main server that pushes it into the corresponding queue. The client A is online and there is no message to consume on the queue, so it is immediately forwarded.

2. The message 2 → A is sent from the client 2 destined to the client A: as the previous one, it is pushed into A's queue but this time the channel is busy. The message will be forwarded as soon as the channel comes idle again.

3. The message 3 → C is sent from client 3 destined to C: again, it is pushed on the correct queue. C is offline, so the message is not forwarded; it will be delivered as soon as C turns online again.

The OS picture inside clients means that the system works on every OS. Eventually, the database icon has been added since it is required for mapping clients' addresses and store chat histories.

3.2 Clients

3.2.1 Role of the Client

During the normal usage of the application, the users interact with their client device, so the client is the principal actor of our application. As described in the use cases analysis user i can register to the application, sign in into his/her account, then he/she can do all the operation of a typical instant-message application like sending/receiving messages and read old messages through clicking on chats. Because the applicative isn't bound to a specific client, as *Whatsapp* is, a client device can be used by multiple **users**, they only need to register/login to their account. From an architectural point of view, the **client** is only in charge of providing the **user** a GUI and the communication with the server. The **client** device does not store the history of the messages, nor information about the user, but it is in charge of showing chats and messages in the correct natural order that is sorted by ascending timestamp. On client-side, a multithreaded approach has been developed according to the following DAG.

Since SEND A MESSAGE and RECEIVE A MESSAGE are concurrent actions, they are performed by different flows of execution. The only shared data structured is the message list of the relative chat, so the access to it must be synchronized.

3.2.2 Technologies

The applicative code runs entirely on the clients: every interaction with the GUI is handled locally and may trigger a send request to the server. The principles technologies used in the client side are JavaFX and Jinterface.

- The GUI is implemented using JavaFX classes, some of them were extended for creating ad-hoc classes that can be found in the javafxexstension package. The use of JavaFX is due to make the application more user-friendly.

- The Jinterface package provides a set of tools for communication with Erlang processes. In this way the client can send and receive messages to the server, define receive mailboxes and so on.

3.3 The Server

3.3.1 Role of the Server

The server is the core of our system, every client has to communicate with it in order to accomplish any operation of that one's listed in chapter 2.1. The server is in charge of:

- Register user data at registration time, remembering username and password.

- Login users by checking username and password, binding usernames with the physical client process in charge of the receiving of the message.

- Forward any message to the correct client: every sender client contacts firstly the server (so that clients are not requested to store physical addresses of other clients, they specify WHAT and not HOW to deliver messages). The server is capable, having as input a username, of determining the relative physical address and forward the message.

- Register every in-transit message in order to permit the restoring of the chat list for every client

- Queuing correctly the messages that are destined to the same client, so that to handle concurrency and buffering of messages whose receiver is offline.

3.3.2 Implementation of the Server

In order to achieve a high-performance application, it is crucial to have a lightweight **server** code, capable of handling quickly every request and of parallelizing work when possible. As discussed in the paragraph 3.2.1, concurrent actions inside a **client** are handled by the **client** itself; the **server** is in charge of handling concurrency between different **clients**. In order to accomplish these requirements, we chose to implement the server entirely in Erlang, so that:

- The lightweight of the language is particularly suitable to ensure high performance on the simple actions that the server must perform
- The Mnesia persistent support guarantees fast operations and internal handling of concurrent accesses to the data (see par. 3.3.3)
- The RabbitMq library queues messages destined to every client with a FIFO policy. It ensures correct concurrency handling and buffering of messages whose receiver is offline.

In addition, to improve performance and abstract the **server** structure, we decided to adopt the *Gen_Server* behavior to handle **client** requests. Moreover, to decrease the coupling between **client** and the **server**, a listener module has been provided. At each request it spawns a process that prepares data structures, forward the request to the *Gen_Server* after a preliminary pattern matching and finally changes the format of the response in a *client-side-easy-to-parse* way.

3.3.3 Persistent Data Storing

For storing all the information regarding the users and their relative messages we make use of Mnesia. The choice to use Mnesia is driven by the fact that Mnesia is designed with requirements like the following:

- Fast real-time key/value lookup
- Complicate non-real-time queries mainly for operation and maintenance
- High fault tolerance

Mnesia is also interesting because of its tight coupling to Erlang, thus almost turning Erlang into a database programming language. This has many benefits, the foremost is that the impedance mismatch between the data format used by the DBMS and the data format used by the programming language, which is used to manipulate the data, completely disappears.

The information is store in two tables named *unisup_users* and *unisup_messages* in the following mode:

3.3.4 Queueing

3.4 Synchronization Management

3.4.1 Client-Side

3.4.2 Server-Side

3.5 Sequence UML Diagrams