INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Université Fédérale Toulouse Midi-Pyrénées

Département de Génie Électrique & Informatique

Rapport de projet

4ème année Informatique et Réseaux

Programmation Orientée Objet 4IR ChatSystem

Students

Jeanne	BERTRAND	jebertra@etud.insa-toulouse.fr
Marie	LAUR	m_laur@etud.insa-toulouse.fr

6 février 2020

Tutors

Sami	YANGUI
Cédric	LEFEBVRE

Table des matières

Introduction

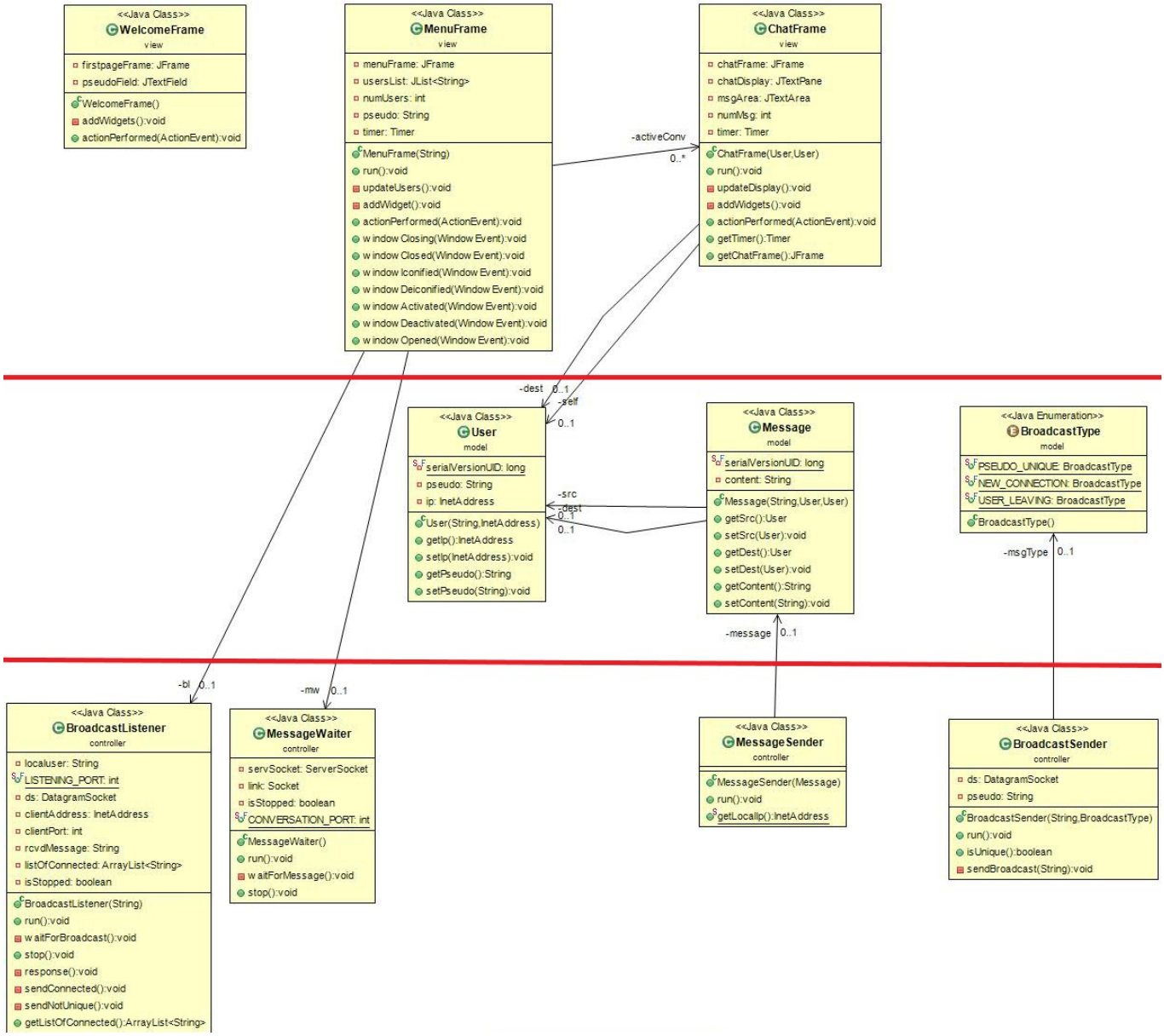
In this report, we present our project of a Chat System. We created a system based on the requirements and also making some hypotheses. For example, we admitted that the users were always using the same computer, therefore we chose to implement a local database.

We provide here a documentation of the conception and development of our application, with a class diagram, precise explanations about the UDP and TCP communications, and the administration of the database (in order to save an history of the chat). We also provide a user manual of the Chat System and explain the tests procedures leading to the validation of the system. In addition, our project is provided with a **JavaDoc** which can be found in the javadoc directory of the project.

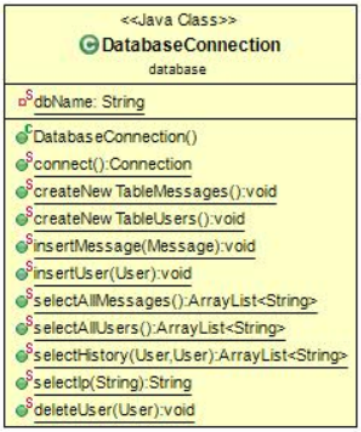
In the end, our application is a functional simple and robust chat, we did not implement all the required features but we think it makes sense because it has been a lot of thinking to develop and it can be used by anyone.

1 Class Diagrams, Conception of the ChatSystem

To present the conception of the ChatSystem, we will use this class diagram.

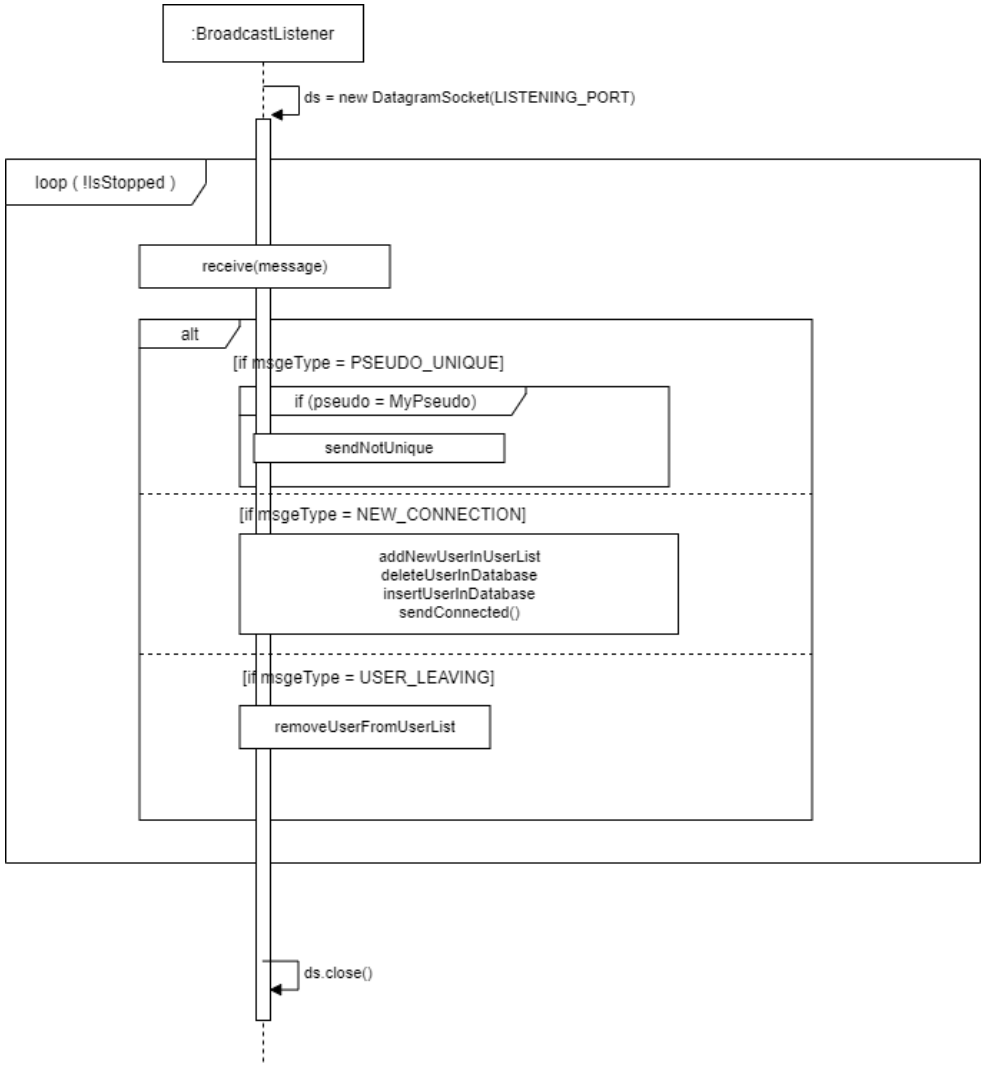


We intented to use a MVC architecture for our project. In the upper part, we have the three windows of the chat (the View package). The second layer represents classes of the Model package, and the lower part is the Controller package. We also have a Database package which only contains one class named DatabaseConnection. This is its class diagram.

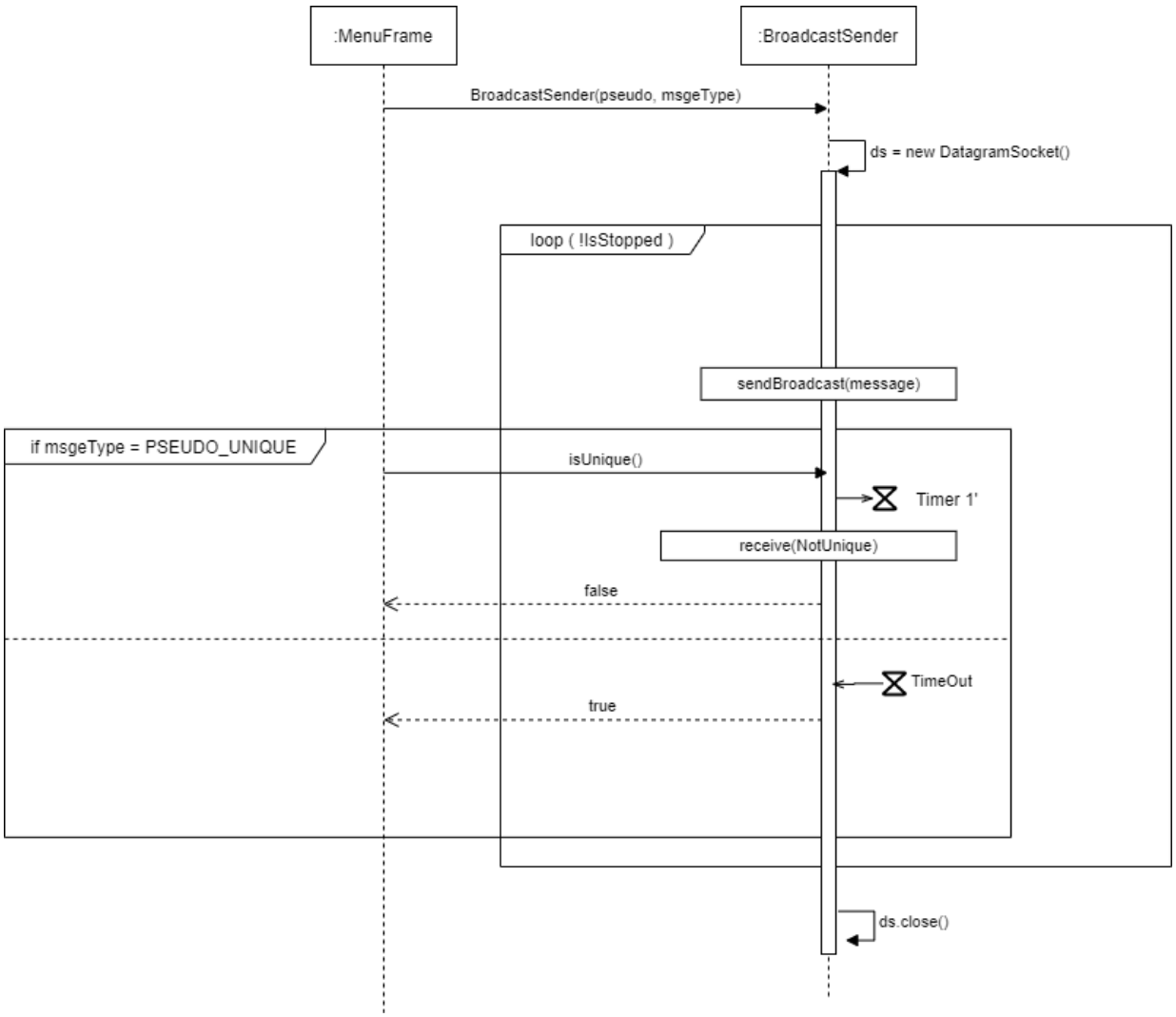


2 UDP Communications

In this part, we will present our implementation of UDP broadcasts to manage the authentication and login of the users, and also their indoors discovery. We use two classes called BroadcastListener and BroadcastSender in order to receive and send broadcasts.



The BroadcastListener class is a thread which waits for an incoming broadcast (sent by the distant BroadcastSender) and respond to it in a different way depending on the type of the broadcast.



The BroadcastSender is also a thread, it is called to inform all the users anytime a user is connecting itself or leaving.

2.1 Authentication and login handling

The authentication and login handling takes place in the WelcomeFrame. When a user enters the application and tries to connect with a pseudonym a new BroadcastSender is instantiated with a **PSEUDO_UNIQUE** broadcast type. After that, the WelcomeFrame is calling the *isUnique()* method from the BroadcastSender object which is waiting for answers from other users. After 1000 millisecond without any answers the pseudonym is considered unique. If the pseudonym chosen by the user is already used, the connection is denied and the user has to try with another pseudo.

2.2 Indoors users discovery

The indoors users discovery takes place in the MenuFrame. The **NEW_CONNECTION** BroadcastType and the **CONNECTED** BroadcastType both allow indoors users discovery.

After login in the application, we send a **NEW_CONNECTION** broadcast (using a new BroadcastSender object) to inform other users of our pseudonym and IP address. When a user receives a **NEW_CONNECTION** broadcast, it is processed by the BroadcastListener always waiting for incoming messages and the BroadcastListener sends back a **CONNECTED** UDP datagram so that the new user knows who is actually connected. This UDP datagram is only sent to the new connected user, not in broadcast because that would be superfluous.

Also, the **USER_LEAVING** broadcast informs users that a user has disconnected from the application. It is sent by a BroadcastSender object in the MenuFrame, when closing the application

or using the *LOGOUT* button. We did not implement the outdoors users discovery (with the HTTP servlet) because we felt we would be short in time and we would rather have the "simple" features working properly than have all the features half-working.

2.3 Broadcast Formats

While implementing the application, we did not find a way to send serialized objects using UDP datagrams. This is the reason why we use a simple String format with a number to identify the type of the Broadcast and the pseudonym of the user sending the broadcast concatenated to it. We only created a BroadcastType enumeration to facilitate the use of the BroadcastSender constructor.

```
public void run() {
    String message="";
    switch(this.msgType) {
    case PSEUDO_UNIQUE :
        message = "0"+this.pseudo;
        System.out.println(message);
        this.sendBroadcast(message);
        break;
    case NEW_CONNECTION :
        message = "1"+this.pseudo ;
        System.out.println(message);
        this.sendBroadcast(message);
        break;
    case USER_LEAVING :
        message = "2"+this.pseudo;
        System.out.println(message);
        this.sendBroadcast(message);
        break;
    }
}
```

FIGURE 1 – *run()* method of BroadcastSender

When the BroadcastListener receives the broadcast, we use a regex to identify its type. We identify the types of broadcast as follow :

- Type 0 for PSEUDO_UNIQUE
- Type 1 for NEW_CONNECTION
- Type 2 for USER_LEAVING
- Type 3 for CONNECTED

3 Sending and receiving messages between users

We use TCP to send messages between users, as it was requested. We have two classes to handle the communication : MessageSender and MessageWaiter.

3.1 MessageSender

MessageSender is instantiated in the ChatFrame whenever the *SEND* button is clicked on. The only attribute for the MessageSender constructor is a Message object (containing a destination User, a source User and a content, see global class diagram). The constructor launches a thread which is going to send the Message object (Message is a Serializable class) over TCP. The socket is bind to the specific port **CONVERSATION_PORT** which is set to 12347 in the MessageWaiter class.

Thus, all MessageWaiter classes of different applications are listening to the same port. It is then essential to have at your disposal at least two computers to test the ChatSystem. Otherwise, an *Address already in use : Cannot bind* error will occur.

After being sent, the Message is also added in the local database. The thread stops and the socket is closed when the Message is sent.

3.2 MessageWaiter

A MessageWaiter object is instantiated in the constructor of the MenuFrame, so from the moment a user is connected to the application, the other users can send messages to him or her.

When the MessageWaiter is instantiated, a thread is started, opening a socket on the specific **CONVERSATION_PORT** mentioned before, and blocking on the "accept" method to receive a serialized message from the MessageSender of the distant user. When a message is received, it is added to the local database, and displayed on the ChatFrame (detailed in the next section).

This thread is a background task, closed by the stop method when the MenuFrame is closed by the user. To be able to close the socket properly when the thread would be stopped (otherwise when we re-launch the port is not released immediately and an exception occurs), we chose to use a *volatile* boolean attribute which is passed as a condition for the while loop. The stop method just closes the socket and changes the value of this volatile boolean to make the main loop breaks.

3.3 Displaying messages

In the ChatFrame, we display a conversation between two users. To get the messages from the history, we use the DatabaseConnection class and the method selectHistory.

Then we want to display all messages of the conversation : to avoid a twinkle effect, we check the amount of messages in the database, and reload them only if there are new messages (if the length of the array keeping the messages is bigger).

We chose to display messages sent and received in different colors, (blue when we sent the message and red when it comes from the other user) it makes the conversation immediately readable by the user.

We implemented the checking and displaying process as a method called updateDisplay. We decided that the ChatFrame should be implementing a TimerTask, allowing us to reload the frame after a fixed period of time. We fixed this period as 500ms.

Finally, this timed thread is stopped from the MenuFrame when the user leaves the application, so all the active conversations can close properly.

4 History handling

About the history handling in the application, we assumed that users are always using the same IP address, so we chose to implement a local history, based on the IP addresses of the users. The technology we are using is SQLite, allowing us to use SQL requests, detailed in the DatabaseConnection class.

Thus, if this application is used in the GEI, it is recommendable to use it in two different INSA logins because launching the application by ssh in the same login is going to join histories of the receiver and sender of the messages. All messages will be displayed twice.

Our history table (the *messages* table) contains for each row, the IP address of the sender, the IP address of the receiver, the message (String format) and a timestamp of the moment it was sent or received. Indeed using IP addresses and not pseudonyms in the history table allow us to manage easily the changes of pseudonyms.

Our DatabaseConnection class also handles a second table, the *users* table, containing all IP addresses of users associated with their pseudonyms.

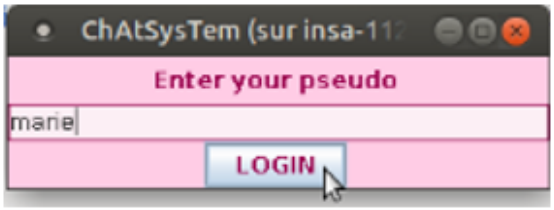
Each time a user is connecting itself, we use a method called *deleteUser()*[1] and then a method *insertUser()*, this process is a safety avoiding having various users using the same IP address or the same pseudonyms and keeping the *users* table up to date with the most recent connections. Thereby, the only moment when a user can be deleted from the table and not inserted immediately after is when someone "steal" the pseudonyms of a unconnected user. Therefore, this table is nearly a memory of all the previous users, it is a backup to know who is or was connected and with which IP.

1. deletes the user row which matches the given pseudonym OR the given IP address

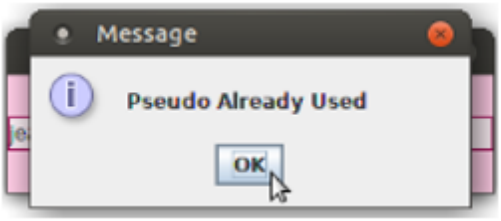
5 User manual

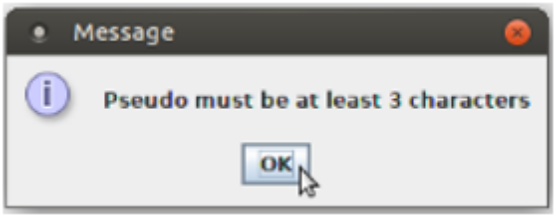
5.1 The Welcome window

To connect in the ChatSystem you just need to enter your pseudo in the field of the first frame and click on the *LOGIN* button. The pseudo must be of more than three characters.



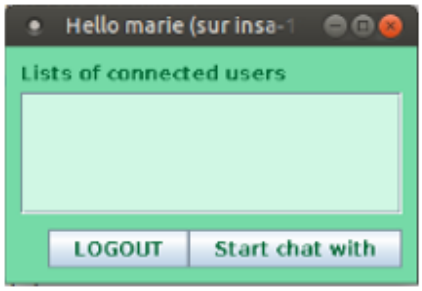
If a message window appears just as follow, you need to click on OK and re-enter a different pseudo to fit the number of characters or the unicity constraint.

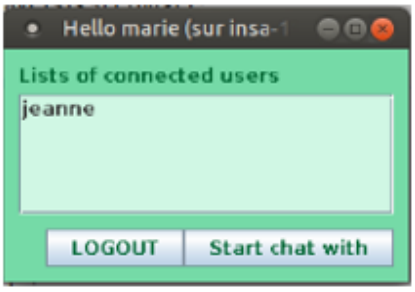




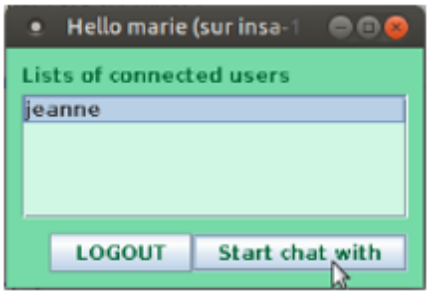
5.2 The Menu window

Once you are logged in, the menu window shows up with your pseudo. If nobody is currently using the application right now, the list of connected users is empty. When a user connects, his or her pseudo is displayed in the scroll pane.

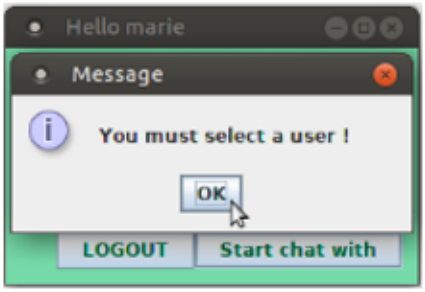




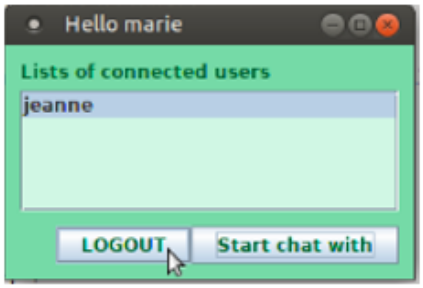
You can now select the user you want to chat with and click on the *Start chat with* button.



If this kind of message window appears, it means you forgot to select a user. Just click on OK and proceed.



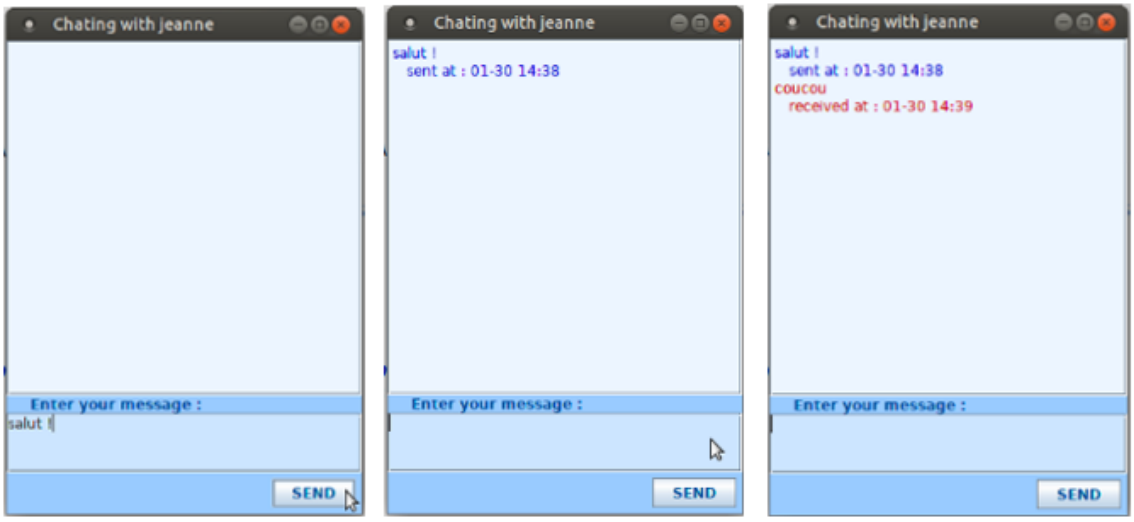
When you are done with chating you have two choices. You can either disconnect using the *LOGOUT* button and go back to the Welcome window. You may re-connect using a different pseudonym. Your messages with other users would be conserved in the history as the messages are identified with IP addresses and not pseudonyms.



If you want to quit the application, you can click on the window cross.

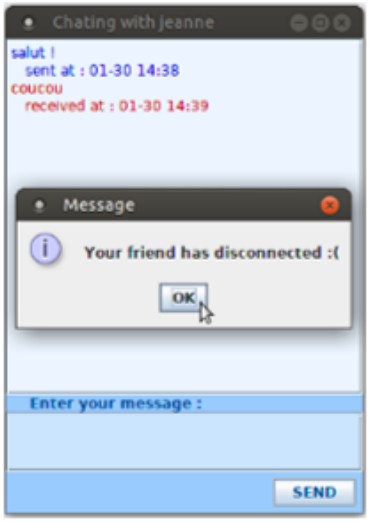
5.3 The Chat window

When you have chosen a user to chat with, the Chat window opens with the name of the distant user. You can send a message filling the message field and clicking on the *SEND* button. The message is then displayed in the scroll pane.



Your received messages are also displayed in the scroll pane but in a different colour. Red for the received messages and blue for the sent ones.

If this window is displayed when you click on the *SEND* button, it means that the user you are chating with has disconnected while you wrote your message. You won't be able to send him anymore messages before he re-connects, so just close the Chat window.



If you want to close the Chat window, click on the window cross. It will only close this chat window, not the Menu window nor the application.

6 Tests procedures

We suggest here a test procedure for our ChatSystem which was carried out and complied with the latest version of the project. As many computers as simulated users (and sessions if tested in the GEI) are needed for the reasons exposed below. During all the trial, all received and sent messages with some details are being displayed in the console or the terminal.

6.1 Authentication and indoors users discovery

To start with, just connect a first user. We can use any pseudonym because nobody else is connected. The list of connected user is empty for the same reason. Then, another ChatSystem can be open in another computer. We can try to enter the same pseudonym than the first user and notice that it is rejected for not being unique. When a valid pseudonym is finally entered, we can observe the appearance of the name of the first user in the MenuFrame of the second one and the other way around. If a third user would connect (with a different pseudo), it would also appear in the other users MenuFrames.

6.2 User leaving

To test the user leaving advertising, we can try to disconnect some users and see if corresponding name actually disappear from the MenuFrame of other users. If a ChatFrame is opened with a connected user and this user leaves the application, we would also be able to check that we cannot communicate with him or her anymore.

6.3 History

To test the history, users can simply exchange some messages and then disconnect. When the two users re-connect using identical or different pseudonyms, the history will be kept and displayed anyway.

6.4 Local network

Our application is meant to be used in a local network, to test this condition, we can simply connect two users from different networks and see that each one won't appear in the MenuFrame of the other.