



Application de chat client/serveur

MODE CONNECTE (TCP)

Miri Mohamed | 27 Avril 2021

Sommaire

Introduction	2
I. C'est quoi une socket	3
1. Définition	3
2. modes de communication	3
II. Contexte du projet	4
1. L'objet du projet	4
2. Présentation du projet	4
❖ L'environnement client/serveur ?	4
❖ Les bibliothèques utilisées ?	4
❖ Architecture ?	6
III. Implémentation	7
1. Partie Client :	7
Partie Client (Client Thread):	9
2. Partie Serveur :	10
Partie Server (ServerThread) :	12
IV. La mise en œuvre des interfaces :	13
Conclusion	Error! Bookmark not defined.
Ressources	Error! Bookmark not defined.

Introduction

Ce rapport d'écrit la réalisation d'une application de chat en utilisant des sockets en JAVA.

Ce projet avait pour objectif de permettre aux clients connectés au serveur de parler entre eux en même temps.

Le développement de ce projet est passé par 5 étapes :

- **Etape 1** : Un serveur simple qui acceptera une seule connexion client et affichera tout ce que le client dit à l'écran. Si l'utilisateur client tape «.bye", le client et le serveur se fermeront tous les deux.
- **Etape 2** : Un serveur comme avant, mais cette fois, il restera « ouvert » pour une connexion supplémentaire une fois qu'un client aura quitté. Le serveur peut gérer au plus une connexion à la fois.
- **Etape 3** : Un serveur comme avant, mais cette fois, il peut gérer plusieurs clients simultanément. La sortie de tous les clients connectés apparaîtra sur l'écran du serveur.
- **Etape 4** : Un serveur comme auparavant, mais cette fois, il envoie tout le texte reçu de l'un des clients connectés à tous les clients. Cela signifie que le serveur doit recevoir et envoyer, et que le client doit aussi bien envoyer que recevoir
- **Etape 5** : création de deux interfaces simple (JFrame), une pour le client l'autre pour le serveur

Mais ce rapport traite que les deux dernières étapes.

Vous pouvez accéder au code de toutes les étapes sur mon répertoire GITHUB :

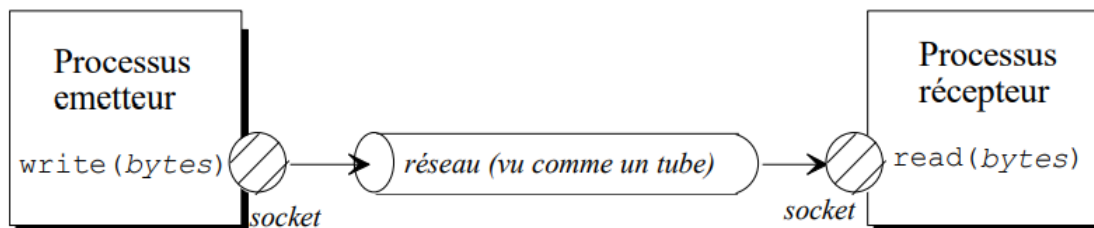
[Commits · Bouddha-ctrl/ClientServer_chat \(github.com\)](https://github.com/Bouddha-ctrl/ClientServer_chat)

I. C'est quoi une socket

1. DEFINITION

Une socket est connue comme une interface de communication logicielle qui agit comme un point d'extrémité qui fonctionne en établissant une liaison de communication réseau bidirectionnelle entre l'extrémité du serveur et le programme de réception du client.

Via cette communication logicielle une application peut envoyer/recevoir des données.



2. MODES DE COMMUNICATION

Le mode connecté :

- Utilisant le protocole TCP qui contrôle si le paquet est arrivé à destination, si ce n'est pas le cas il le renvoie. Et établie une connexion durable entre les deux processus

Le mode non connecté :

- Utilisant le protocole UDP qui ne vérifie pas si le paquet est bien arrivé.

II. Contexte du projet

1. L'OBJET DU PROJET

Ce projet est pour le but d'implémenter les concepts de la programmation réseau par l'utilisation d'API socket, et aussi de mettre en place les notions de programmation java.

2. PRESENTATION DU PROJET

❖ L'environnement client/serveur ?

L'environnement client/serveur désigne un mode de communication à travers un réseau entre plusieurs programmes : l'un qualifié de **client**, envoie des requêtes l'autre qualifié de **serveur**, attendent les requêtes des clients et répond.

Caractéristique du **Serveur** :

- Il attend une connexion entrante
- A la connexion d'un client sur le port en écoute, il ouvre une socket local
- Suite à la connexion, le serveur communique avec le client

Caractéristique du **Client** :

- Il établit une connexion au serveur avec une adresse IP et un port
- Lorsque la connexion est acceptée par le serveur, il communique comme prévoit la couche applicative du modèle OSI

Le client et le serveur doivent bien sur utiliser le même protocole de communication.

❖ Les bibliothèques utilisées ?

Le package **java.net**

```
import java.net.*;
```

Ce package fournit les classes pour la mise en œuvre des application réseau, comme :

- **La classe ServerSocket** : utilisé au coté serveur, c'est un objet de type socket, elle est utilisé avec le constructeur **ServerSocket(int port)**, et cette classe possède deux méthodes : **accept()** qu'elle est une instruction

bloquant qui attende simplement les appels du client et renvoie une socket qui encapsule la communication avec ce client et **close()** qui ferme la socket.

- **La classe Socket** : utilisé avec le constructeur **Socket(string IPServeur, int Port)**, les méthodes utilisées sont : **getInputStream()** qui renvoie un flux pour recevoir les données de la socket, **getOutputStream()** pour émettre les données de la socket, **close()** pour fermer le socket.

Le package **java.io**

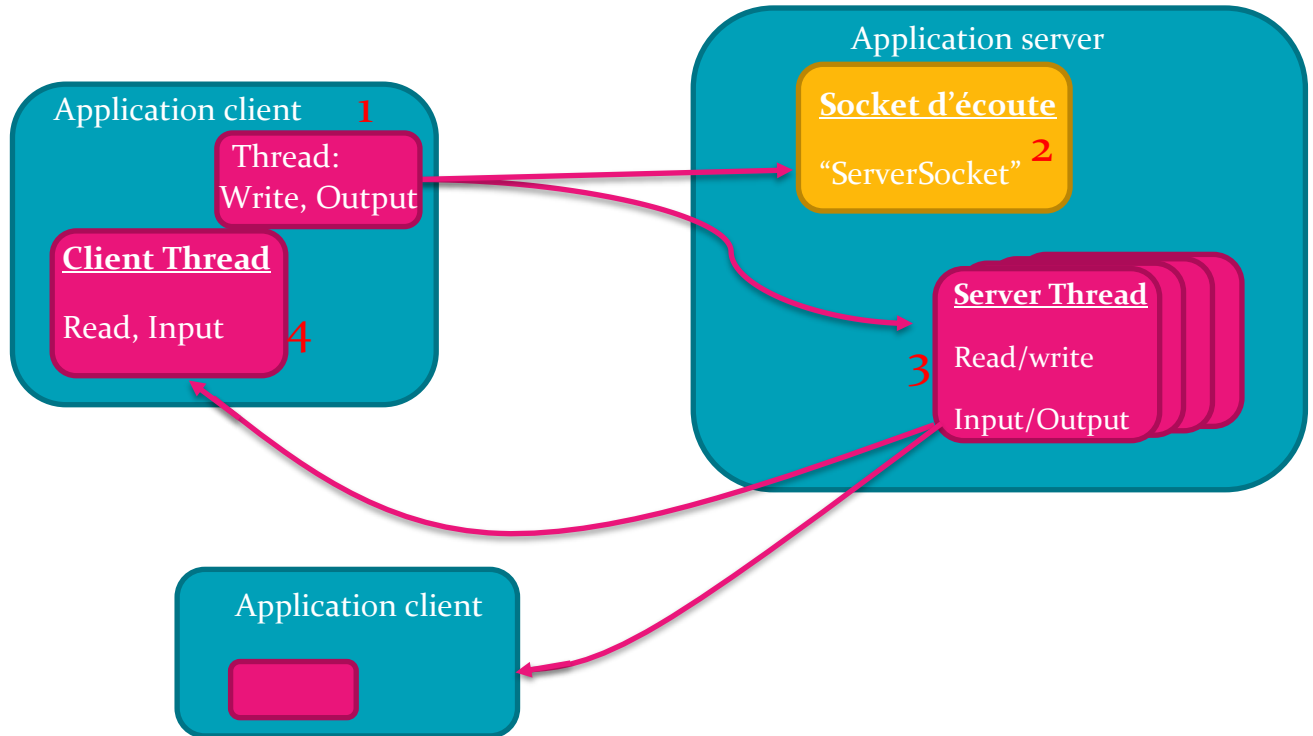
```
import java.io.*;
```

Ce package fournit l'entrée et sortie du système via les flux de données.

- **La classe DataInputStream** : crée un flux de données de sortie, elle est utilisée avec une méthode du Socket à travers son constructeur **DataInputStream(InputStream Socket.getInputStream())**, et on utilise deux méthodes pour cette classe : **readUTF()** et **close()**.
- **La classe DataOutputStream** : le même processus que DataInputStream, Le constructeur **DataOutputStream (OutputStream Socket.getOutputStream())**, et les deux méthodes : **writeUTF()** et **close()**.
- **La classe BufferedReader** : cette classe est utilisée pour lire au clavier.

❖ Architecture ?

L'implémentation de ce projet va suivre l'architecture suivante :



Cette architecture suit des étapes :

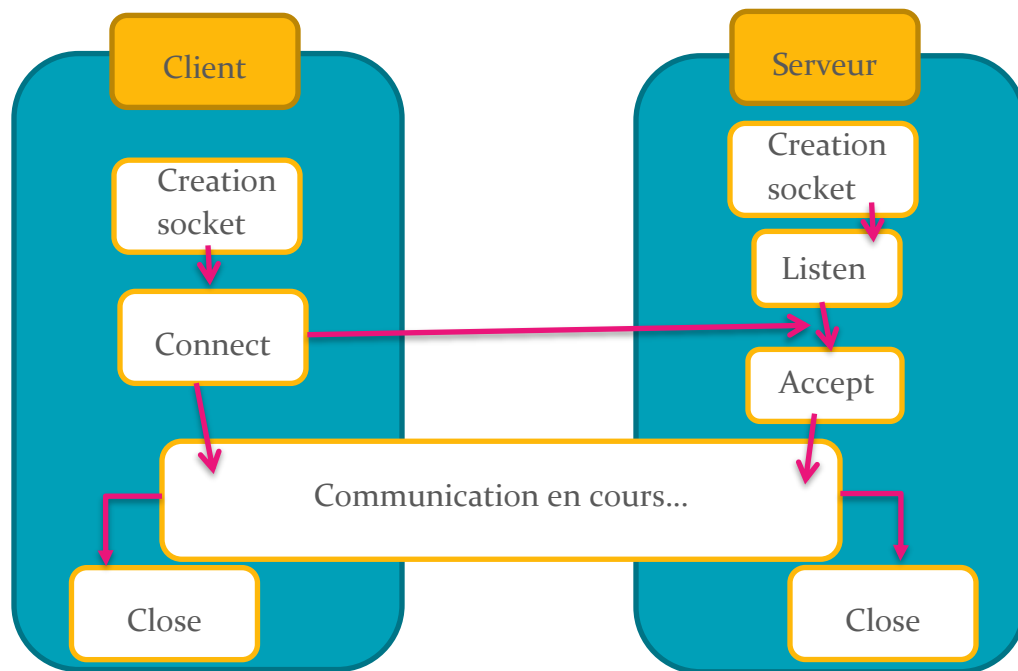
1. Dans cette partie, on a la création du socket, la connexion, ouvre les flux et la création d'un Thread qui s'occupe de l'envoi des données et la création d'un objet **ClientThread** qui s'occupe de la réception des données.
2. Le serveur contient un Thread dans un état d'attend, il reçoit les demandes de connexion, s'il l'accepte, il crée un objet **ServerThread** pour chaque client accepté et il l'ajoute dans une liste de **ServerThread**, quand un client se déconnecte, le **ServerThread** associé est détruit et retiré de la liste.

Quand le client est connecté il peut commencer à envoyer les données

3. Le **Server Thread** reçoit les données et il les diffuse à tous les clients connectés.
4. Le **Client Thread** reçoit les données.

III. Implémentation

Afin d'accomplir ce projet, on va le diviser en deux parties, la première partie concerne la partie serveur, et la deuxième pour la partie client.



1. PARTIE CLIENT :

Création du Socket :

```
Socket socket = null;
```

Connexion au serveur :

```
socket =new Socket(Server_Ip,port);
```

Ouverture des flux :


```

public void open() {
    try {
        dout = new DataOutputStream(socket.getOutputStream());
        sc = new BufferedReader(new InputStreamReader(System.in));

    } catch (IOException e) {
        System.out.println("Error getting output stream: " + e);
        e.printStackTrace();
    }
}

```

Création du Client Thread :

```
client= new ClientThread(this, socket);
```

Envoi des données :

```

dout.writeUTF(sc.readLine());
dout.flush();

```

Fonction pour le control de l'output : (réception des données)

```

public void handle(String msg)
{
    if (msg.equals(".bye"))
    {
        System.out.println("Good bye.");
        stop();
    }
    else
        System.out.println(msg);
}

```

Fermeture :

```

public void close() throws IOException {
    System.out.println("Disonnected! ");
    System.exit(0);
    dout.close();
    sc.close();
    socket.close();
}

```

Partie Client (Client Thread):

Après la création d'une instance dans la partie Client, le constructeur :

```

public ClientThread(Client client, Socket socket) {
    this.client = client;
    this.socket = socket;
    open();
    start();
}

```

Ouverture des flux :

```
din=new DataInputStream(socket.getInputStream());
```

Réception des données :

```
client.handle(din.readUTF());
```

Fermeture :

```
din.close();
```

2. PARTIE SERVEUR :

Dans cette partie il y a des méthodes préfixées par **synchronized** , Cette instruction empêche 2 thread ou plus d'utiliser la même méthode au même temps , pour éviter les erreurs.

L'ouverture des flux et leur fermeture reste identique.

Création de la socket du serveur :

```
ServerSocket server = null;
```

Initialisation du serveur :

```
server = new ServerSocket(port);
```

Initialisation d'une liste de client avec une capacité de 50 clients :

```
ServerThread clients[] = new ServerThread[50];
```

Ecoute et connexion des clients : (dans une boucle)

```
addClients(server.accept());
```

La méthode addClients vérifie la capacité du serveur puis crée une ServerThread pour le client :

```
clients[clientCount] = new ServerThread(this, socket);
```

Une méthode findClient , qui trouve l'indice d'un client dans la table du serveur par son numero de service (port) :

```
private int findClient(int ID)
{
    for (int i = 0; i < clientCount; i++)
        if (clients[i].getID() == ID)
            return i;
    return -1;
}
```

Une méthode remove() qui supprime un client de la liste des client une fois qu'il est déconnecté :

```
public synchronized void remove(int ID)
{
    int pos = findClient(ID);
    if (pos >= 0)
    {
        try {
            ServerThread toTerminate = clients[pos];
            //System.out.println("[SERVER] Client disconnected : " + ID
            if (pos < clientCount-1)
                for (int i = pos+1; i < clientCount; i++)
                    clients[i-1] = clients[i];
            clientCount--;

            toTerminate.close();
        } catch (IOException ioe)
        {
            System.out.println("Error closing thread: " + ioe);
        } catch (Exception e) {
            System.out.println("test");
        }
    }
}
```

Une méthode handle qui contrôle les outputs : (réception des données)

```
public synchronized void handle(int ID, String input)
{
    System.out.println("Msg received from "+ID+" : "+input);
    if (input.contains(".bye"))
    {
        remove(ID);
    }
    else
    {
        for (int i = 0; i < clientCount; i++) {
            clients[i].send(input);
        }
        System.out.println("Msg broadcasted");
    }
}
```

Partie Server (ServerThread) :

Cette partie gère les entrées et les sorties.

Après l'instanciation, le constructeur :

```
public ServerThread(Server _server, Socket _socket)
{
    server = _server;
    socket = _socket;
    ID = socket.getPort();
}
```

Réception des données :

```
public void run()
{
    while (true)
    {
        try
        {
            String line = din.readUTF();

            server.handle(ID, line); // broadcast in handle with send()
        }
    }
}
```

La méthode send() qui envoie les données:

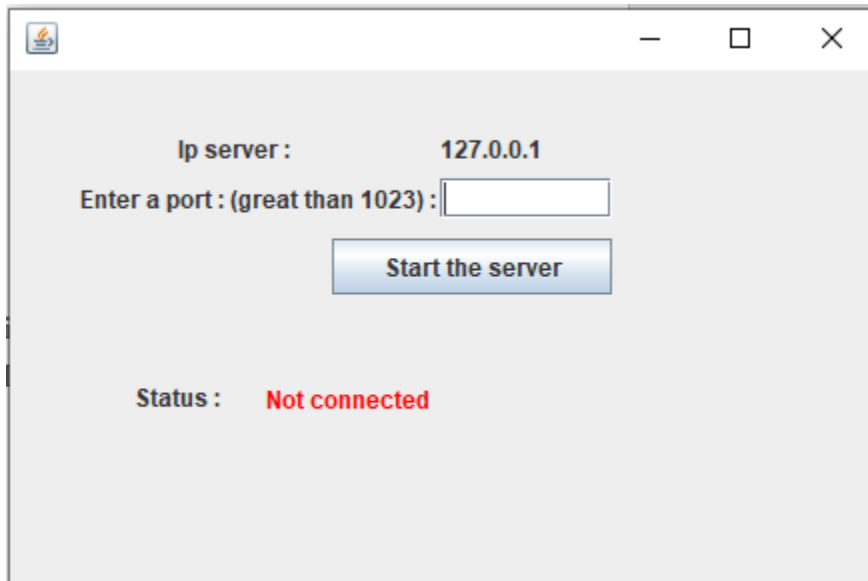
```
public void send(String msg)
{
    try
    {
        dout.writeUTF(msg);
        dout.flush();
    }
}
```

IV. La mise en œuvre des interfaces :

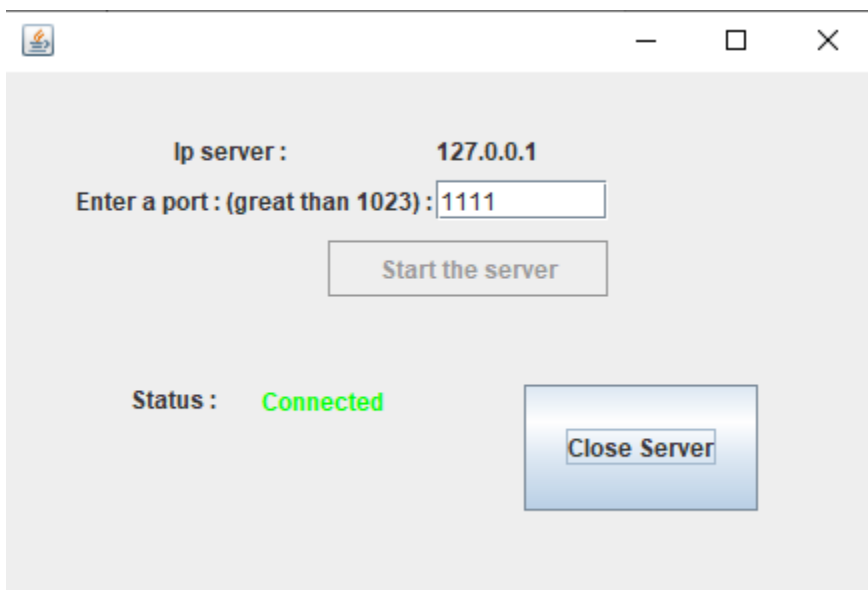
Après le bon fonctionnement du code au terminal, j'ai développé une petite interface simple avec `javax.swing`.

Le code est un peu différent, par exemple à la classe Client le thread qui envoie les données est remplacé par [event dispatch thread](#) Le thread du GUI.

JServer :

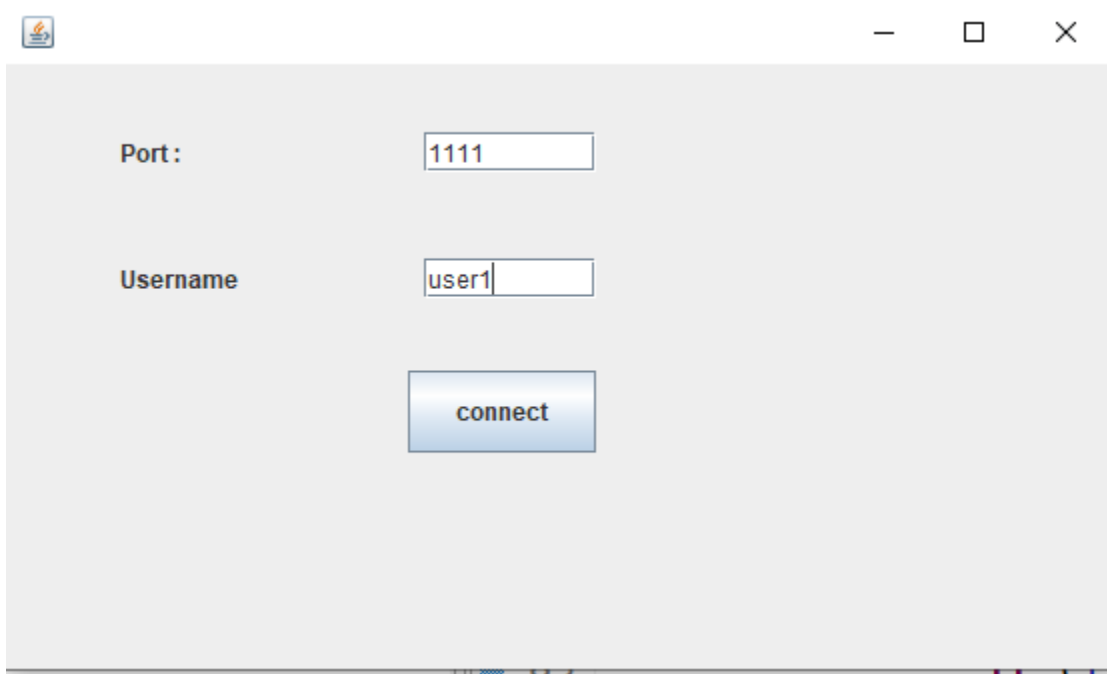


On lance le serveur :



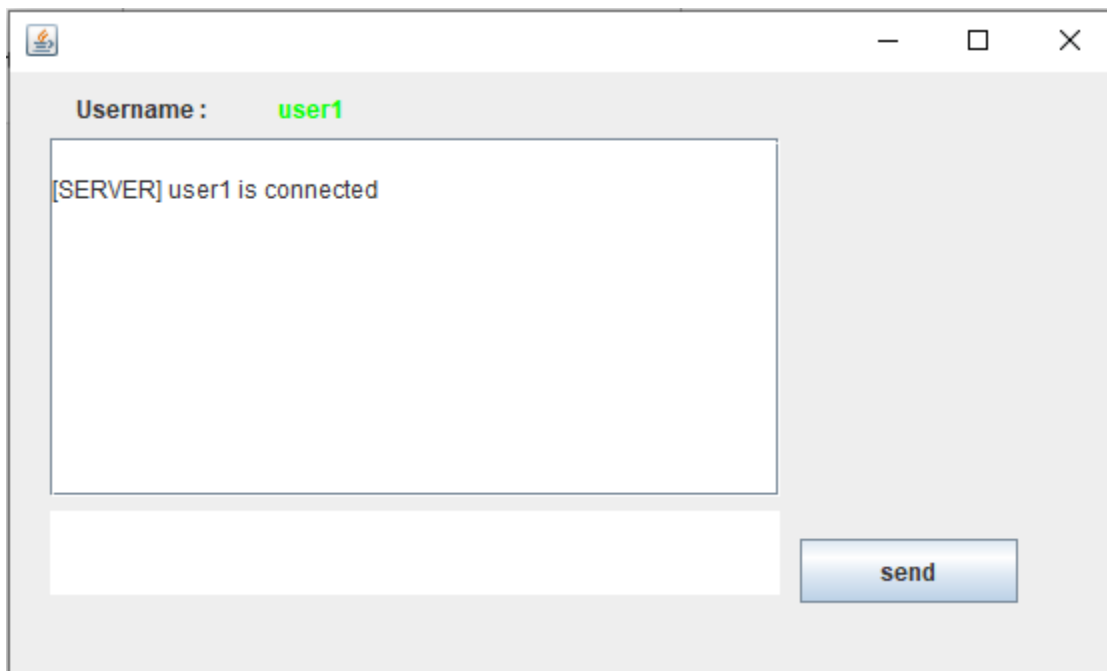
Le serveur est bien lancé (pas d'exception lever)

JClient :



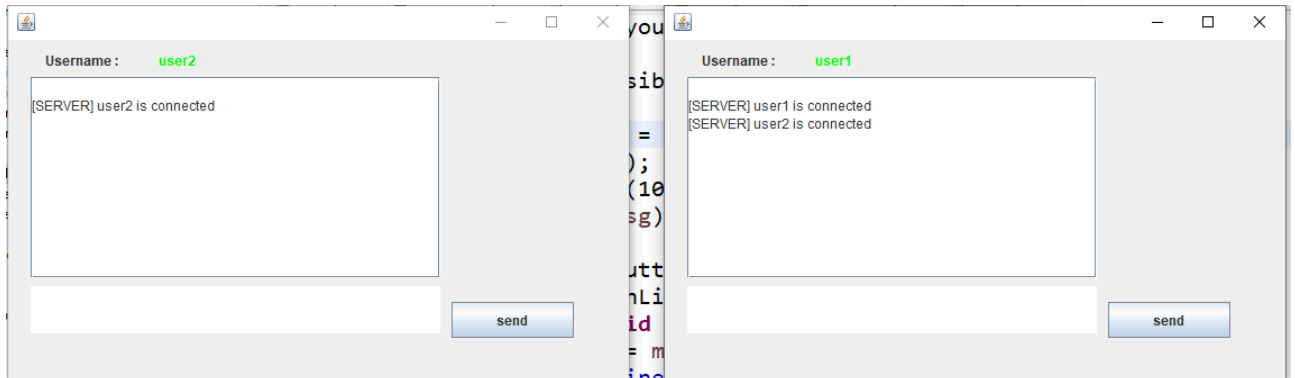
The screenshot shows a Java Swing window titled 'JClient'. It has a standard title bar with a small icon on the left and minimize, maximize, and close buttons on the right. The main content area is light gray and contains three elements: a label 'Port :' followed by a text input field containing '1111', a label 'Username' followed by a text input field containing 'user1', and a blue button labeled 'connect' centered below the input fields.

On se connecte :

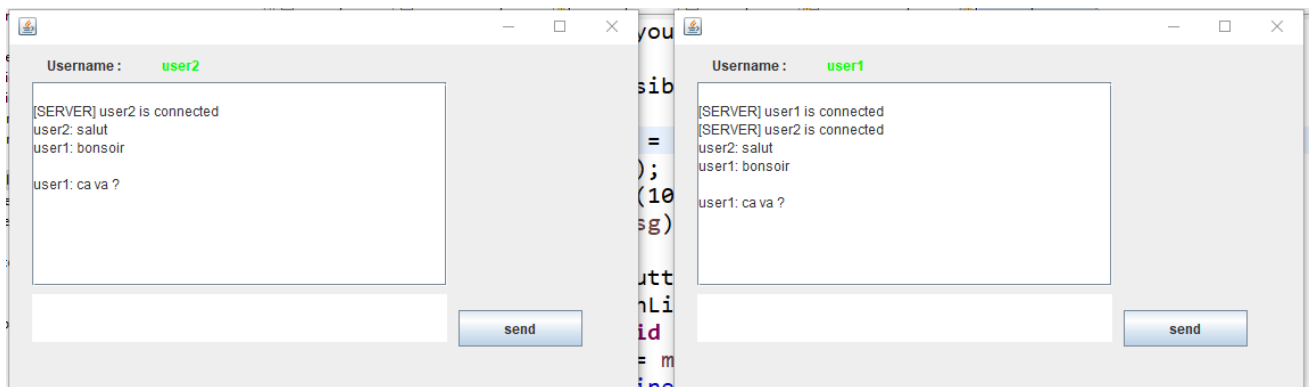


The screenshot shows the same 'JClient' window after a successful connection. The title bar remains the same. The main content area now displays 'Username : user1' in green text. Below this, a large white rectangular area contains the message '[SERVER] user1 is connected'. At the bottom of the window, there is a new white text input field and a blue button labeled 'send' to its right.

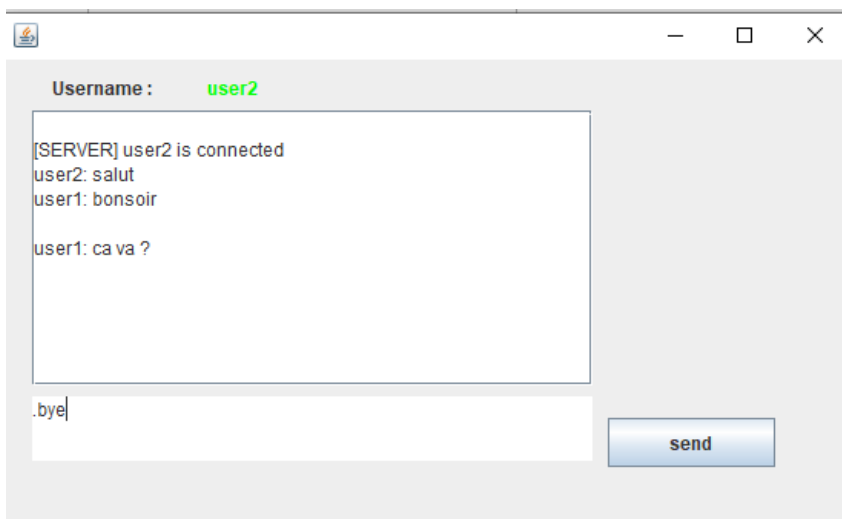
On ajoute un autre client :



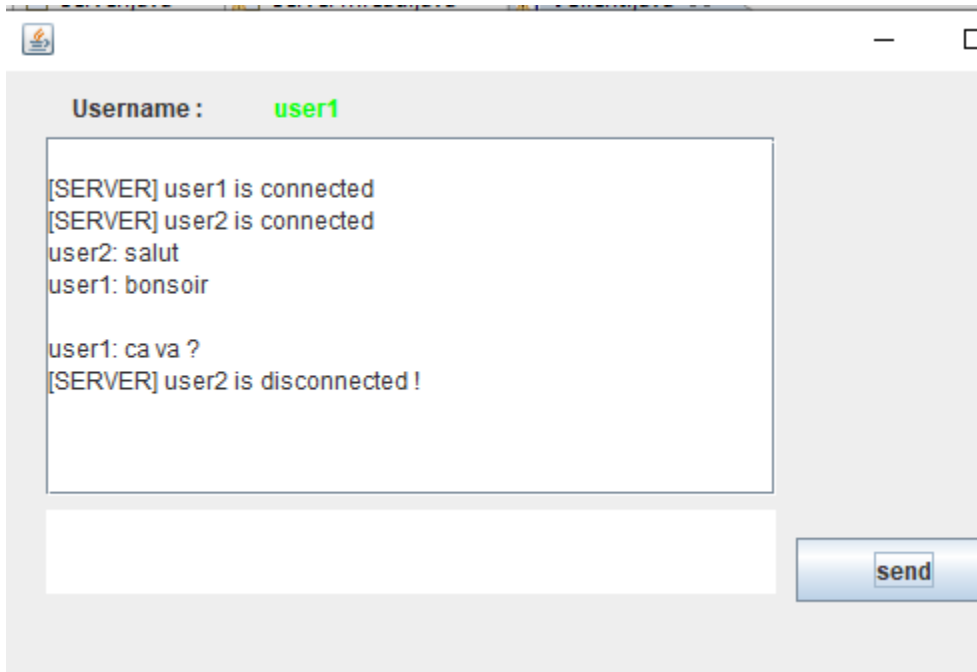
Les clients peuvent maintenant chatter :



Et pour se déconnecter, il suffit de taper « .bye » :



Client déconnectés :



Puis on ferme le serveur .

Conclusion

Durant projet, ma mission consistait à créer une application de chat client/serveur en mode connecté, pour s'un utilisateur peut se connecter et de se déconnecter avec plusieurs contacts par un serveur en utilisant les sockets en JAVA.

Au cours de la période, on a eu l'opportunité de mettre en évidence les différentes connaissances acquises et d'acquérir de nouveaux concepts savoir java.net . De plus j'ai eu l'occasion d'appliquer mes connaissances en JAVA, et les principes du système client/serveur en réseau, ainsi le rôle principal des sockets.

La partie que j'ai développé correspond aux objectifs de départ, Mais ceci n'empêche pas d'améliorer l'application au futur.

Ressources

- <https://www.instructables.com/Creating-a-Chat-Server-Using-Java/>
- <http://pirate.shu.edu/~wachsmut/Teaching/CSAS2214/Virtual/Lectures/chat-client-server.html>
- [Rapport application chat \(slideshare.net\)](#)
- https://drive.google.com/file/d/1Nhmyk2deZbYmgMClsUbTH_X-GFfxa_92/view?usp=sharing