

---

## Compte Rendu : Projet programmation multitâche

---

Dans le cadre de notre cours de programmation multitâche et temps réel, nous avons réalisé ce projet. L'objectif est de créer un serveur de chat et d'implémenter au fur et à mesure les clients.

Le projet peut être présenté sous trois versions :

- V0 est composée du serveur et d'un client (le but étant de vérifier la communication à travers les sockets)
- V1 est une implémentation complète du cahier des charges, un serveur et n clients, capable d'échanger entre eux
- V2 intègre le serveur et n clients, avec une interface graphique côté client, un ensemble de commandes serveur, et quelques fonctionnalités en plus pour l'expérience utilisateur. La V2 est programmée avec des verrous « synchronized » sur chaque bloc de code qui utilise des ressources partagées.
- La V3 est une V2 améliorée. Elle ne possède pas de fonctionnalités supplémentaires en revanche l'ajout de wait() dans les fonctions run() des threads et de notify() dans les blocs « synchronized » permet d'optimiser l'utilisation des ressources et de garantir des meilleures performances.

Le projet nous a permis de mieux comprendre le fonctionnement et les enjeux de la programmation multitâche. Nous avons appris comment faire communiquer client et server, en centralisant la gestion des messages sur le serveur, puis en optimisant le partage des ressources.

Le code du projet est disponible à l'adresse : [https://github.com/n0ss/java\\_chat\\_server](https://github.com/n0ss/java_chat_server)

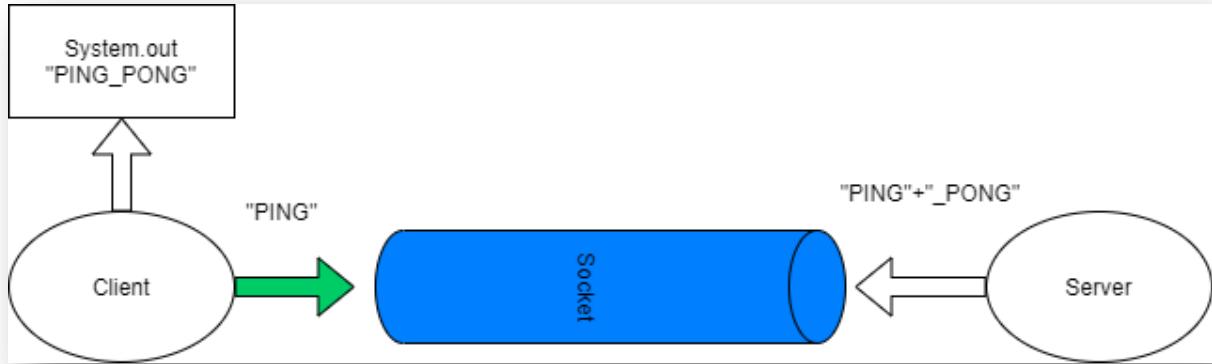
Vous pouvez le télécharger directement en cliquant sur :  
[https://github.com/n0ss/java\\_chat\\_server/archive/main.zip](https://github.com/n0ss/java_chat_server/archive/main.zip)



Compte Rendu : Projet programmation multitâche.....	1
Partie 1 : Modélisation .....	3
a)    Modélisation du projet : V0.....	3
b)    Modélisation du projet : V1.....	3
c)    Modélisation du projet : V2 et V3 .....	4
Partie 2 : Code et fonctionnalités CLIENT/SERVER .....	5
a)    Server V0 .....	5
b)    Client V0 .....	6
c)    Server V1 .....	6
d)    Client V1 .....	14
e)    Server V2 .....	16
f)    Client V2 .....	22
g)    Ajout de la V3 .....	24
Partie 3 : Exécution.....	25
a)    Version 0 .....	25
b)    V1 .....	26
c)    Version 2 (test similaire pour la v3).....	27
Conclusion .....	30

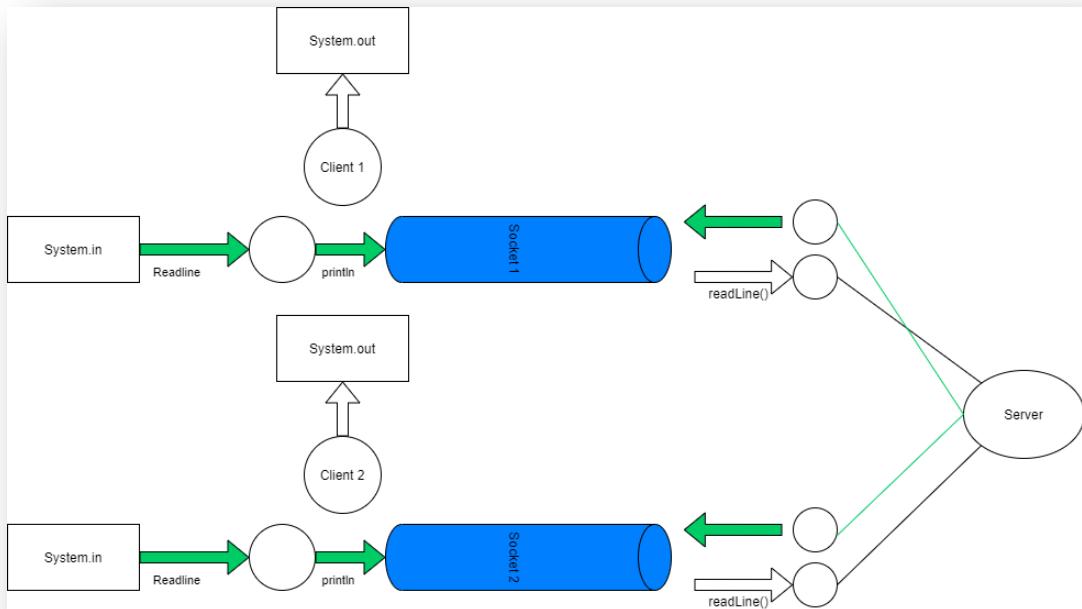
Partie 1 : Modélisation

a) Modélisation du projet : V0



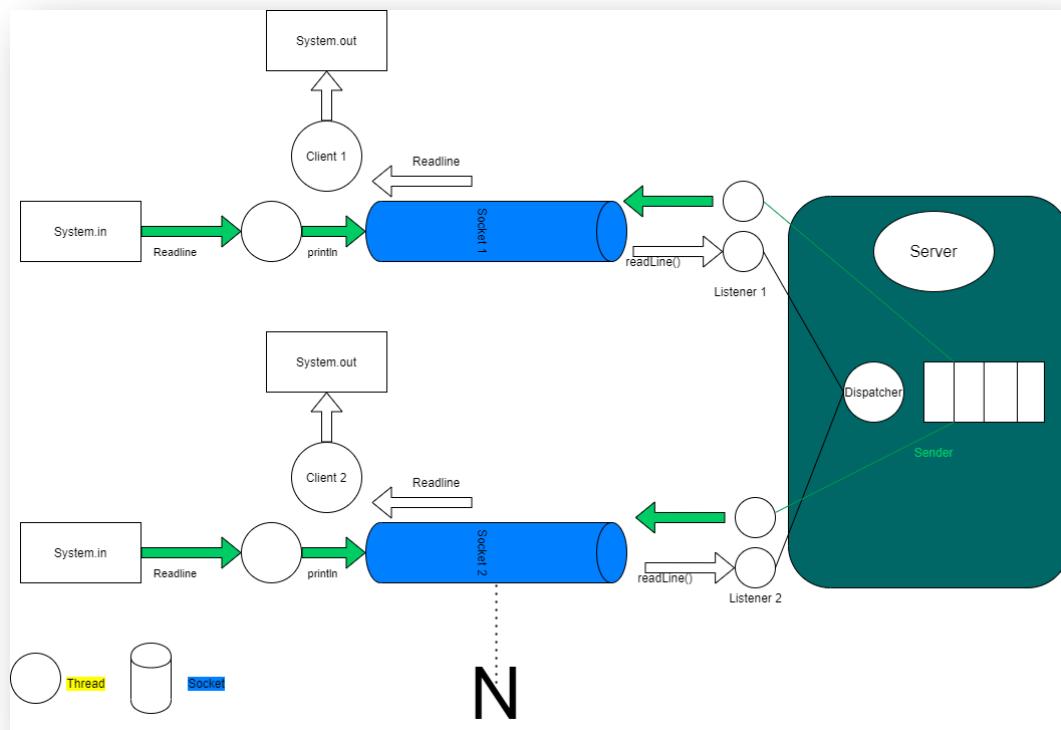
Dans ce schéma on peut voir comment fonctionne la V0. Lorsque le server est lancé, sur le port correspondant (par exemple 3333 dans cette V0) il est alors en attente de client. On vient alors simplement vérifier la communication entre les deux à l'aide d'un simple message « PING » du client, puis d'une réponse concaténé « PING »+ « \_PONG ».

b) Modélisation du projet : V1



Dès la version 2, le schéma se complexifie. On peut voir ici la présence de plusieurs threads qui vont venir gérer les entrées et les sorties puis départager la réception et l'envoie au server. Dans ce cas-là, les deux clients peuvent communiquer. C'est un chat en duplex. Si l'on regarde l'exemple pour un seul client, on a tout d'abord une entrée (System.in) qui est traitée par **Readline** puis affiché par **println**. Ensuite du côté server on applique un **Readline** pour lire l'input et le renvoyer vers le thread destiné à l'autre client qui va alors parcourir le trajet annexe pour enfin être géré par le thread client qui va s'occuper d'afficher le message (**System.out**). Cependant, après avoir directement réalisés cette étape nous sommes passé à la version « n-clients », donc vous aurez à disposition sur la V2 la version CLI du projet. Le fonctionnement est détaillé sur le prochain schémas.

### c) Modélisation du projet : V2 et V3



La version 2 est donc celle qui est la plus aboutie. Avec cette version on peut faire communiquer plusieurs Clients, sans avoir de limite théoriquement (mise à part pour les ressources physique de la machine). L'ajout par rapport à la version précédente est notamment le dispatcher qui permet de traiter les différentes arrivées des threads. Dans ce cas il est plus simple de voir ça comme un thread qui empile chaque envoi des utilisateurs. De cette manière le Dispatcher de quelle manière envoyer ce qu'il reçoit. Nous allons le voir plus tard mais ceci est particulièrement utile pour une des fonctionnalités supplémentaire que nous avons ajoutés (PM). La version 3 est une amélioration de celle-ci. Elle dispose des *wait()*, *notify()* et les *synchronize*.

## Partie 2 : Code et fonctionnalités CLIENT/SERVER

### a) Server V0

```
public class Server {  
  
    public static void main(String[] args) {  
  
        try {  
  
            ServerSocket server = new ServerSocket(3333);  
  
            System.out.println("Server : waiting on port 3333");  
  
            Socket client = server.accept();  
  
            System.out.println("Server : connection successful on port 3333");  
  
            BufferedReader in = new BufferedReader(new InputStreamReader(client.getInputStream()));  
  
            PrintWriter out = new PrintWriter(new BufferedWriter(new OutputStreamWriter(client.getOutputStream())),true);  
  
            String message = in.readLine();  
  
            out.println("Server at '" + Calendar.getInstance().getTime()+"' : "+ message + "_PONG");  
  
            in.close(); out.close(); client.close();server.close();  
  
        } catch (Exception e) {System.out.println("Server error");}  
  
    }  
}
```

*Server.java*

Le premier Server représente le squelette pour toutes les versions. On trouve le ServerSocket server et le Socket client. Le socket client est créé côté server après que le premier client accède au port du localhost. A l'aide du PrintWriter, le Server renvoie le message reçu par le client, permettant d'avoir la réponse « PING\_PONG » du côté client.

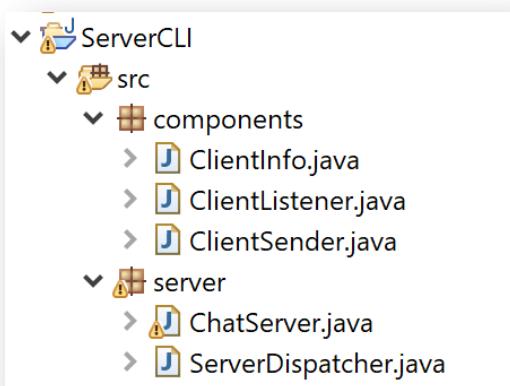
### b) Client V0

```
import java.io.BufferedReader;  
  
public class Client {  
    public static void main(String[] args) {  
        try {  
            Socket client = new Socket("localhost",3333);  
            BufferedReader in = new BufferedReader(new InputStreamReader(client.getInputStream()));  
            PrintWriter out = new PrintWriter(new BufferedWriter(new OutputStreamWriter(client.getOutputStream())),true);  
            out.println("Client_PING");  
            String message = in.readLine();  
            System.out.println("Client - message received : ["+ message + "]");  
            in.close(); out.close(); client.close();  
        }  
        catch (Exception e) {System.out.println("Client error");}  
    }  
}
```

*Client.java*

La classe Client est composé d'un socket (localhost :3333). La lecture en entrée est faite grâce au BufferedReader **in** tandis que la sortie est faite grâce à PrintWriter **out**. De ce fait il est possible d'envoyer « Client\_PING » au Serveur. L'exception permet de gérer le cas où le client ne trouve pas de Serveur associé au port correspondant.

### c) Server V1



Voici la structure générale du ServerCLI qui est la version améliorée de la V1 du serveur. Ici nous avons la capacité de faire communiquer deux clients différents. Il n'y a pas de dispatcher pour gérer de potentiels clients en plus.

ClientInfo :

```
public class ClientInfo {  
  
    public Socket mSocket;  
    public ClientListener mClientListener;  
    public ClientSender mClientSender;  
    public String pseudo;  
  
    public ClientInfo(Socket client, ServerDispatcher dispatcher) {  
        // TODO Auto-generated constructor stub  
  
        mSocket = client;  
        mClientListener = new ClientListener(this, dispatcher);  
        mClientSender = new ClientSender(this);  
  
        Thread t_listener = new Thread (mClientListener);  
        t_listener.setDaemon(true);  
        t_listener.setPriority(Thread.NORM_PRIORITY);  
        t_listener.start();  
  
        Thread t_sender = new Thread (mClientSender);  
        t_sender.setDaemon(true);  
        t_sender.setPriority(Thread.NORM_PRIORITY);  
        t_sender.start();  
    }  
}
```

*ClientInfo.java*

ClientListener :

```
public class ClientListener implements Runnable {  
  
    private ServerDispatcher mServerDispatcher;  
    private ClientInfo mClientInfo;  
    private BufferedReader mIn;  
  
    public ClientListener(ClientInfo info, ServerDispatcher dispatcher) {  
        // TODO Auto-generated constructor stub  
  
        try {  
  
            mServerDispatcher = dispatcher;  
            mClientInfo = info;  
            mIn = new BufferedReader(new InputStreamReader(mClientInfo.mSocket.getInputStream()));  
  
        } catch (IOException e) {  
            // TODO Auto-generated catch block  
            System.out.println("Erreur Serveur ClientListener() : constructeur");  
            e.printStackTrace();  
        }  
    }  
}
```

*ClientListener.java – screenshot 1*

```
public void run() {
    // TODO Auto-generated method stub

    String message;

    try {
        message = mIn.readLine();
        mClientInfo.pseudo = message;
        mServerDispatcher.printClients();

    } catch (IOException e1) {
        // TODO Auto-generated catch block
        System.out.println("Erreur Serveur ClientListener() : attribution du pseudo");
        e1.printStackTrace();
    }

    while (!mClientInfo.mSocket.isClosed()) {

        try {

            message = mIn.readLine();

            if (message!=null) {

                if (message.charAt(0)=='/') {
                    if (message.substring(1).startsWith("exit")) {
                        mClientInfo.mSocket.close();
                    }
                    else if (message.substring(1).startsWith("pm ")) {
                        message = message.substring(4, message.length());
                    }
                }
            }
        }
    }
}
```

*ClientListener.java – screenshot 2*

```
String dest = message.split("\\s+")[0];
message = message.substring(dest.length()+1,message.length());

synchronized (mServerDispatcher) {
    mServerDispatcher.dispatchPrivateMessage("PM: "+mClientInfo.pseudo+" > "+message,dest);
    //notify();
}

else if (message.substring(1).startsWith("shout ")) {
    message = message.substring(7, message.length());
    message = message.toUpperCase();

    synchronized (mServerDispatcher) {
        mServerDispatcher.dispatchMessage(mClientInfo.pseudo+" > "+message);
        //notify();
    }
}
else {
    synchronized (mServerDispatcher) {
        mServerDispatcher.dispatchMessage(mClientInfo.pseudo+" > "+message);
        //notify();
    }
}
else {
    synchronized (mServerDispatcher) {
        mServerDispatcher.dispatchMessage(mClientInfo.pseudo+" > "+message);
        //notify();
    }
}
```

*ClientListener.java – screenshot 3*

```
    }

}

}

} catch (IOException e) {
    // TODO Auto-generated catch block
    System.out.println("Erreur Serveur ClientListener() : récupération message - dispatch");
    e.printStackTrace();
}

mServerDispatcher.deleteClient(mClientInfo);

}

}
```

*ClientListener.java*

ClientSender :

```
public class ClientSender implements Runnable {  
  
    private Vector<String> mMessageQueue;  
    private ClientInfo mClientInfo;  
    private PrintWriter mOut;  
  
    public ClientSender(ClientInfo info) {  
        // TODO Auto-generated constructor stub  
  
        try {  
  
            mMessageQueue = new Vector<String>();  
            mClientInfo = info;  
            mOut = new PrintWriter(new BufferedWriter(new OutputStreamWriter(mClientInfo.mSocket.getOutputStream())), true);  
  
        } catch (IOException e) {  
            // TODO Auto-generated catch block  
            System.out.println("Erreur Serveur ClientSender() : constructeur");  
            e.printStackTrace();  
        }  
    }  
}
```

```
>     private void sendMessageToClient() {  
        // TODO Auto-generated method stub  
        synchronized (mMessageQueue) {  
            mOut.println(nextMessageFromQueue());  
            mMessageQueue.removeElementAt(0);  
            //notify();  
        }  
    }  
  
>     public void sendMessage(String message) {  
        // TODO Auto-generated method stub  
        synchronized (mMessageQueue) {  
            mMessageQueue.add(message);  
            //notify();  
        }  
    }  
  
>     private String nextMessageFromQueue() {  
        // TODO Auto-generated method stub  
  
        return mMessageQueue.firstElement();  
    }  
  
    public void run() {  
        // TODO Auto-generated method stub  
  
        while (!mClientInfo.mSocket.isClosed()) {  
  
            if (!mMessageQueue.isEmpty()) {  
                sendMessageToClient();  
            }  
  
        }  
    }  
}
```

ChatServer : Main

Seul attribut : `public static final int LISTENING_PORT = 5555;`

```
public static void main(String[] args) {  
  
    try {  
  
        ServerSocket server = new ServerSocket(LISTENING_PORT);  
        System.out.println("Server : attente sur le port "+LISTENING_PORT);  
  
        ServerDispatcher dispatcher = new ServerDispatcher();  
        Thread t_dispatcher = new Thread(dispatcher);  
        t_dispatcher.setDaemon(true);  
        t_dispatcher.setPriority(Thread.NORM_PRIORITY);  
        t_dispatcher.start();  
  
        Socket new_client;  
  
        while (true) {  
            //dispatcher.printClients();  
  
            new_client = server.accept();  
            System.out.println("Un client s'est connecté.");  
  
            dispatcher.addClient(new_client);  
  
        }  
  
    } catch (IOException e) {  
        // TODO Auto-generated catch block  
        System.out.println("Erreur Serveur ChatServer : main() thread");  
        e.printStackTrace();  
    }  
}
```

*ChatServer.java*

ServerDispatcher :

```
public class ServerDispatcher implements Runnable {  
  
    private Vector<String> mMessageQueue;  
    private Vector<ClientInfo> mClients;  
    //private ClientSender mServerDispatcher;  
  
    public ServerDispatcher() {  
        // TODO Auto-generated constructor stub  
  
        mMessageQueue = new Vector<String>();  
        mClients = new Vector<ClientInfo>();  
        //mServerDispatcher = new ClientSender();  
    }  
  
    public void addClient(Socket client) {  
        // TODO Auto-generated method stub  
        ClientInfo new_info = new ClientInfo(client, this);  
  
        synchronized (mClients) {  
            mClients.add(new_info);  
            //notify();  
        }  
  
    }  
}
```

*ServerDispatcher.java – screenshot 1*

```
public void deleteClient(ClientInfo old_client) {  
    // TODO Auto-generated method stub  
    synchronized (mClients) {  
        mClients.removeElement(old_client);  
        //notify();  
  
        System.out.println("Un client s'est déconnecté : " + old_client.pseudo+ ".");  
    }  
    printClients();  
}  
  
public void printClients() {  
    // TODO Auto-generated method stub  
    System.out.println("----- CLIENTS ACTIFS -----");  
    synchronized (mClients) {  
        for (ClientInfo client: mClients) {  
            printClient(client);  
        }  
    }  
    System.out.println("-----\n");  
}  
  
private void printClient(ClientInfo client) {  
    // TODO Auto-generated method stub  
    System.out.println("\t\t"+client.pseudo);  
}
```

*ServerDispatcher.java – screenshot 2*

```
private String nextMessageFromQueue() {
    // TODO Auto-generated method stub

    return mMessageQueue.firstElement();
}

@Override
public void run() {
    // TODO Auto-generated method stub

    while (true) {

        if (!mMessageQueue.isEmpty()) {
            sendMessageToAllClients();
        }
        else {

        }

    }
}
```

*ServerDispatcher.java – screenshot 3*

```
public void dispatchMessage(String message) {
    // TODO Auto-generated method stub
    synchronized (mMessageQueue) {
        mMessageQueue.add(message);
        //notify();
    }
}

public void dispatchPrivateMessage(String message, String dest ) {
    for (ClientInfo client: mClients) {
        if (client.pseudo.equals(dest)) {
            client.mClientSender.sendMessage(message);
        }
    }
}

private void sendMessageToAllClients() {
    // TODO Auto-generated method stub

    String message = nextMessageFromQueue();

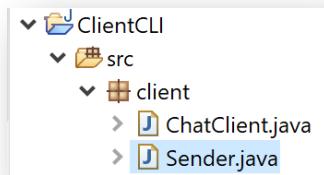
    for (ClientInfo client: mClients) {
        client.mClientSender.sendMessage(message);
    }

    synchronized (mMessageQueue) {
        mMessageQueue.removeElement(message);
    }
}
```

*ServerDispatcher.java – screenshot 4*

d) Client V1

*ChatClient :*



```
public class ChatClient {  
  
    public static final int SERVER_PORT = 5555;  
    public static final String SERVER_HOSTNAME = "localhost";  
  
    public static void main(String[] args) {  
        try {  
            Socket socket = new Socket(SERVER_HOSTNAME, SERVER_PORT);  
            Sender sender = new Sender(socket);  
            Thread t_sender = new Thread(sender);  
            t_sender.setDaemon(true);  
            t_sender.setPriority(Thread.NORM_PRIORITY);  
            t_sender.start();  
            BufferedReader mIn = new BufferedReader(new InputStreamReader(socket.getInputStream()));  
            String chat_output = "0";
```

*ChatClient.java – screenshot 1*

```
while (chat_output!=null) {  
  
    try {  
        chat_output = mIn.readLine();  
        System.out.println(chat_output);  
    }  
    catch (SocketException e) {  
        // TODO Auto-generated catch block  
        System.out.println("Erreur Client ChatClient : main() thread - lecture BR socket");  
        e.printStackTrace();  
    }  
  
    } catch (IOException e) {  
        // TODO Auto-generated catch block  
        System.out.println("Erreur Client ChatClient : main() thread");  
        e.printStackTrace();  
    }  
}
```

*ChatClient.java – screenshot 2*

Sender:

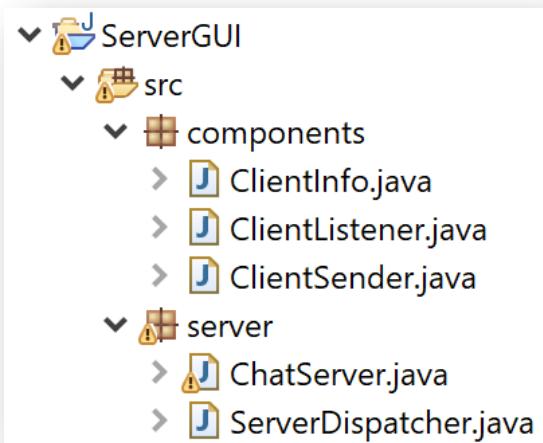
```
public class Sender implements Runnable {  
  
    private static PrintWriter mOut;  
    private static Socket mSocket;  
  
    public Sender(Socket sock) {  
        // TODO Auto-generated constructor stub  
  
        try {  
            mOut = new PrintWriter(new BufferedWriter(new OutputStreamWriter(sock.getOutputStream())),true);  
            mSocket = sock;  
        } catch (IOException e) {  
            // TODO Auto-generated catch block  
            System.out.println("Erreur Client - Sender() : constructeur");  
            e.printStackTrace();  
        }  
    }  
}
```

*Sender.java – screenshot 1*

```
@Override  
public void run() {  
    // TODO Auto-generated method stub  
  
    Scanner scan = new Scanner(System.in);  
  
    System.out.print("Choisissez un pseudo : ");  
    mOut.println(scan.nextLine());  
  
    while (!mSocket.isClosed()) {  
        if (scan.hasNextLine()) {  
  
            mOut.println(scan.nextLine());  
  
        }  
    }  
    scan.close();  
}
```

*Sender.java – screenshot 2*

e) Server V2



```
public class ClientInfo {

    public Socket mSocket;
    public ClientListener mClientListener;
    public ClientSender mClientSender;
    public String pseudo;

    public ClientInfo(Socket client, ServerDispatcher dispatcher) {
        // TODO Auto-generated constructor stub

        mSocket = client;
        mClientListener = new ClientListener(this, dispatcher);
        mClientSender = new ClientSender(this);

        Thread t_listener = new Thread (mClientListener);
        t_listener.setDaemon(true);
        t_listener.setPriority(Thread.NORM_PRIORITY);
        t_listener.start();

        Thread t_sender = new Thread (mClientSender);
        t_sender.setDaemon(true);
        t_listener.setPriority(Thread.NORM_PRIORITY);
        t_sender.start();
    }

}
```

*ClientInfo.java – screenshot 1*

ClientListener :

```
public class ClientListener implements Runnable {  
  
    private ServerDispatcher mServerDispatcher;  
    private ClientInfo mClientInfo;  
    private BufferedReader mIn;  
  
    public ClientListener(ClientInfo info, ServerDispatcher dispatcher) {  
        // TODO Auto-generated constructor stub  
  
        try {  
  
            mServerDispatcher = dispatcher;  
            mClientInfo = info;  
            mIn = new BufferedReader(new InputStreamReader(mClientInfo.mSocket.getInputStream()));  
  
        } catch (IOException e) {  
            // TODO Auto-generated catch block  
            System.out.println("Erreur Serveur ClientListener() : constructeur");  
            e.printStackTrace();  
        }  
    }  
}
```

*ClientListernet.java – screenshot 1*

```
@Override  
public void run() {  
    // TODO Auto-generated method stub  
  
    String message;  
  
    try {  
        message = mIn.readLine();  
        mClientInfo.pseudo = message;  
        mServerDispatcher.printClients();  
        mClientInfo.mClientSender.welcomeMessage();  
  
    } catch (IOException e1) {  
        // TODO Auto-generated catch block  
        System.out.println("Erreur Serveur ClientListener() : attribution du pseudo");  
        e1.printStackTrace();  
    }  
}
```

*ClientListener.java – screenshot 2*

```
while (!mClientInfo.mSocket.isClosed()) {  
    try {  
        message = mIn.readLine();  
        if (message!=null) {  
            if (message.charAt(0)=='/') {  
                if (message.substring(1).startsWith("exit")) {  
                    mClientInfo.mSocket.close();  
                }  
                else if (message.substring(1).startsWith("pm ")) {  
                    message = message.substring(4, message.length());  
                    String dest = message.split("\\s+")[0];  
                    message = message.substring(dest.length()+1,message.length());  
                    synchronized (mServerDispatcher) {  
                        mServerDispatcher.dispatchPrivateMessage("PM: "+mClientInfo.pseudo+" > "+message,dest);  
                    }  
                }  
                else if (message.substring(1).startsWith("shout ")) {  
                    message = message.substring(7, message.length());  
                    message = message.toUpperCase();  
                    synchronized (mServerDispatcher) {  
                        mServerDispatcher.dispatchMessage(mClientInfo.pseudo+" > "+message);  
                    }  
                }  
            }  
        }  
    }  
}
```

*ClientListener.java – screenshot 3*

```
else if (message.substring(1).startsWith("shout ")) {  
    message = message.substring(7, message.length());  
    message = message.toUpperCase();  
    synchronized (mServerDispatcher) {  
        mServerDispatcher.dispatchMessage(mClientInfo.pseudo+" > "+message);  
    }  
}  
else if (message.substring(1).startsWith("help")) {  
    mClientInfo.mClientSender.sendMessage("[INFO > /pm pseudo message : Envoyer un message privé");  
    mClientInfo.mClientSender.sendMessage("INFO > /shout message : Envoyer un message en CAPSLOCK");  
    mClientInfo.mClientSender.sendMessage("INFO > /help : Afficher l'aide");  
    mClientInfo.mClientSender.sendMessage("INFO > /exit : Déconnecter la session");  
}  
else {  
    synchronized (mServerDispatcher) {  
        mServerDispatcher.dispatchMessage(mClientInfo.pseudo+" > "+message);  
    }  
}  
}  
else {  
    synchronized (mServerDispatcher) {  
        mServerDispatcher.dispatchMessage(mClientInfo.pseudo+" > "+message);  
    }  
}  
}  
}
```

*ClientListener.java – screenshot 4*

```
} catch (IOException e) {  
    // TODO Auto-generated catch block  
    System.out.println("Erreur Serveur ClientListener() : récupération message - dispatch");  
    e.printStackTrace();  
}  
}  
mServerDispatcher.deleteClient(mClientInfo);  
}  
}
```

*ClientListener.java – screenshot 5*

ClientSender :

```
public class ClientSender implements Runnable {  
  
    private Vector<String> mMessageQueue;  
    private ClientInfo mClientInfo;  
    private PrintWriter mOut;  
  
    public ClientSender(ClientInfo info) {  
        // TODO Auto-generated constructor stub  
  
        try {  
            mMessageQueue = new Vector<String>();  
            mClientInfo = info;  
            mOut = new PrintWriter(new BufferedWriter(new OutputStreamWriter(mClientInfo.mSocket.getOutputStream())), true);  
  
        } catch (IOException e) {  
            // TODO Auto-generated catch block  
            System.out.println("Erreur Serveur ClientSender() : constructeur");  
            e.printStackTrace();  
        }  
    }  
  
    private void sendMessageToClient() {  
        // TODO Auto-generated method stub  
        synchronized (mMessageQueue) {  
            mOut.println(nextMessageFromQueue());  
            mMessageQueue.removeElementAt(0);  
        }  
    }  
}
```

*ClientSender.java – screenshot 1*

```
public void sendMessage(String message) {  
    // TODO Auto-generated method stub  
    synchronized (mMessageQueue) {  
        mMessageQueue.add(message);  
    }  
}  
  
public void welcomeMessage () {  
    synchronized (mMessageQueue) {  
        mMessageQueue.add("SERVER > Bienvenue sur le chat "+mClientInfo.pseudo+" !");  
    }  
}  
  
private String nextMessageFromQueue() {  
    // TODO Auto-generated method stub  
  
    return mMessageQueue.firstElement();  
}  
  
@Override  
public void run() {  
    // TODO Auto-generated method stub  
  
    while (!mClientInfo.mSocket.isClosed()) {  
  
        if (!mMessageQueue.isEmpty()) {  
            sendMessageToClient();  
        }  
    }  
}
```

*ClientSender.java – screenshot 2*

ChatServer : Le code est ici le même

ServerDispatcher :

```
public class ServerDispatcher implements Runnable {

    private Vector<String> mMessageQueue;
    private Vector<ClientInfo> mClients;

    public ServerDispatcher() {
        // TODO Auto-generated constructor stub

        mMessageQueue = new Vector<String>();
        mClients = new Vector<ClientInfo>();
    }

    public void addClient(Socket client) {
        // TODO Auto-generated method stub
        ClientInfo new_info = new ClientInfo(client, this);

        synchronized (mClients) {
            mClients.add(new_info);
        }
    }

    public void deleteClient(ClientInfo old_client) {
        // TODO Auto-generated method stub
        synchronized (mClients) {
            mClients.removeElement(old_client);

            System.out.println("Un client s'est déconnecté : " + old_client.pseudo+ ".");
        }
        printClients();
    }
}
```

*ServerDispatcher.java – screenshot 1*

```
public void printClients() {
    // TODO Auto-generated method stub
    System.out.println("----- CLIENTS ACTIFS -----");
    synchronized (mClients) {
        for (ClientInfo client: mClients) {
            printClient(client);
        }
    }
    System.out.println("-----\n");
}

private void printClient(ClientInfo client) {
    // TODO Auto-generated method stub
    System.out.println("\t\t"+client.pseudo);
}

public void dispatchMessage(String message) {
    // TODO Auto-generated method stub
    synchronized (mMessageQueue) {
        mMessageQueue.add(message);
    }
}

public void dispatchPrivateMessage(String message, String dest ) {
    for (ClientInfo client: mClients) {
        if (client.pseudo.equals(dest)) {
            client.mClientSender.sendMessage(message);
        }
    }
}
```

*ServerDispatcher.java – screenshot 2*

```
private void sendMessageToAllClients() {
    // TODO Auto-generated method stub

    String message = nextMessageFromQueue();

    for (ClientInfo client: mClients) {
        client.mClientSender.sendMessage(message);
    }

    synchronized (mMessageQueue) {
        mMessageQueue.removeElement(message);
    }
}

private String nextMessageFromQueue() {
    // TODO Auto-generated method stub

    return mMessageQueue.firstElement();
}
```

*ServerDispatcher.java – screenshot 3*

f) Client V2

```
public class ChatClient {  
    public static final int SERVER_PORT = 5555;  
    public static final String SERVER_HOSTNAME = "localhost";  
    public static Sender sender;  
    private static GUI window = null;  
  
    public static void main(String[] args) {  
        try {  
            Socket socket = new Socket(SERVER_HOSTNAME, SERVER_PORT);  
            sender = new Sender(socket);  
            Thread t_sender = new Thread(sender);  
            t_sender.setDaemon(true);  
            t_sender.setPriority(Thread.NORM_PRIORITY);  
            t_sender.start();  
  
            EventQueue.invokeLater(new Runnable() {  
                public void run() {  
                    try {  
                        window = new GUI();  
                        window.getFrame().setVisible(true);  
                    } catch (Exception e) {  
                        e.printStackTrace();  
                    }  
                }  
            });  
        }  
    }  
}
```

*ChatClient.java – screenshot 1*

```
BufferedReader mIn = new BufferedReader(new InputStreamReader(socket.getInputStream()));  
  
String chat_output = "0";  
  
while (chat_output!=null) {  
  
    try {  
        chat_output = mIn.readLine();  
        window.getChatArea().append(chat_output+"\n");  
        //OLD CLI System.out.println(chat_output);  
    }  
    catch (SocketException e) {}  
    // TODO Auto-generated catch block  
    System.out.println("Erreur Client ChatClient : main() thread - lecture BR socket");  
    e.printStackTrace();  
}  
  
}  
  
socket.close();  
window.getChatArea().append("Client déconnecté. Vous pouvez fermer la fenêtre.");  
window.allowExit();  
  
} catch (IOException e) {  
    // TODO Auto-generated catch block  
    System.out.println("Erreur Client ChatClient : main() thread");  
    e.printStackTrace();  
}
```

*ChatClient.java – screenshot 2*

Dans *ChatClient.java* on trouve aussi l'interface graphique dans la classe GUI. Je vous laisserais aller la voir, nous avons utilisés Window Builder, qui permet de créer simplement l'architecture d'une interface par widget.

Sender :

```
public class Sender implements Runnable {  
  
    private static PrintWriter mOut;  
    private static Socket mSocket;  
    private static Vector<String> mMessageQueue;  
  
    public Sender(Socket sock) {  
        // TODO Auto-generated constructor stub  
  
        try {  
            mOut = new PrintWriter(new BufferedWriter(new OutputStreamWriter(sock.getOutputStream())),true);  
            mSocket = sock;  
            mMessageQueue = new Vector<String>();  
        } catch (IOException e) {  
            // TODO Auto-generated catch block  
            System.out.println("Erreur Client - Sender() : constructeur");  
            e.printStackTrace();  
        }  
    }  
  
    public void pushMessage (String message) {  
        synchronized (mMessageQueue) {  
            mMessageQueue.add(message);  
        }  
    }  
  
    private String popMessage () {  
        return mMessageQueue.remove(0);  
    }  
}
```

Sender.java – screenshot 1

```
private String popMessage () {  
    return mMessageQueue.remove(0);  
}  
  
@Override  
public void run() {  
    // TODO Auto-generated method stub  
  
    //OLD CLI Scanner scan = new Scanner(System.in);  
  
    //OLD CLI System.out.print("Choisissez un pseudo : ");  
    //OLD CLI mOut.println(scan.nextLine());  
  
    while (!mSocket.isClosed()) {  
  
        if (!mMessageQueue.isEmpty()) {  
            synchronized (mMessageQueue) {  
                mOut.println(popMessage());  
            }  
        }  
    }  
  
    //OLD CLI scan.close();  
}  
}
```

Sender.java – screenshot 2

g) Ajout de la V3

Nous avons ajouté plusieurs notify() par exemple dans *ClientSender.java* :

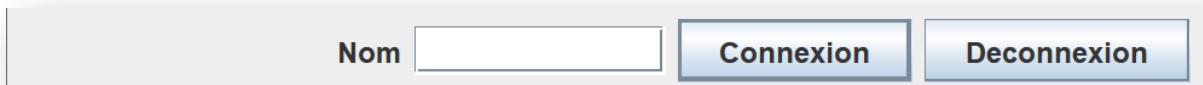
```
public void sendMessage(String message) {  
    // TODO Auto-generated method stub  
    synchronized (this) {  
        mMessageQueue.add(message);  
        this.notify();  
    }  
}
```

Le second est placé dans *ServerDispatcher.java* :

```
public void dispatchMessage(String message) {  
    // TODO Auto-generated method stub  
    synchronized (this) {  
        mMessageQueue.add(message);  
        this.notify();  
    }  
}
```

Dans la méthode run() des threads, il y a, en cas de manque de travail à effectuer, des wait() (pile vide par exemple) qui permettront aux notify() d'être utile dans le réveil des processus.

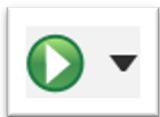
Nous avons aussi corrigé quelques erreurs comme par exemple celle-ci :



Anciennement le bouton « connexion » était « envoyer ». Ce qui est plus claire que « envoyer ». De cette manière le bouton « envoyer » correspondant au message, et « connexion » pour mettre le nom d'utilisateur sont différenciables.

Partie 3 : Exécution

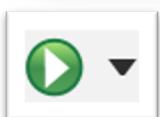
a) Version 0



: Server

A screenshot of the Eclipse IDE's Console view. The tab bar at the top shows 'Problems', 'Javadoc', 'Declaration', and 'Console'. The 'Console' tab is active. The console output window displays the following text:

```
Server [Java Application] C:\Program Files\Java\jdk-14.0.2\bin\javaw.exe (24 oct. 2020 à 18:39:34)
Server : waiting on port 3333
```



Client :

A screenshot of the Eclipse IDE's Console view. The tab bar at the top shows 'Problems', 'Javadoc', 'Declaration', and 'Console'. The 'Console' tab is active. The console output window displays the following text:

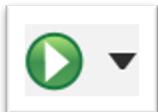
```
<terminated> Client [Java Application] C:\Program Files\Java\jdk-14.0.2\bin\javaw.exe (24 oct. 2020 à 18:43:09 – 18:43:09)
Client - message received : [Server at 'Sat Oct 24 18:43:09 CEST 2020' : client_PING_PONG]
```

Réponse côté serveur :

A screenshot of the Eclipse IDE's Console view. The tab bar at the top shows 'Problems', 'Javadoc', 'Declaration', and 'Console'. The 'Console' tab is active. The console output window displays the following text:

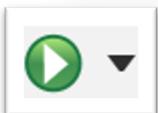
```
<terminated> Server [Java Application] C:\Program Files\Java\jd
Server : waiting on port 3333
Server : connection successful on port 3333
```

b) V1



: Serveur

ChatServer (1) [Java Application] C:\Program Files\  
Server : attente sur le port 5555



: Client

Première étape, on entre son nom d'utilisateur :

Choisissez un pseudo : John

Choisissez un pseudo : Michel

Côté serveur :

```
Server : attente sur le port 5555
Un client s'est connecté.
----- CLIENTS ACTIFS -----
John
-----
Un client s'est connecté.
----- CLIENTS ACTIFS -----
John
Michel
-----
```

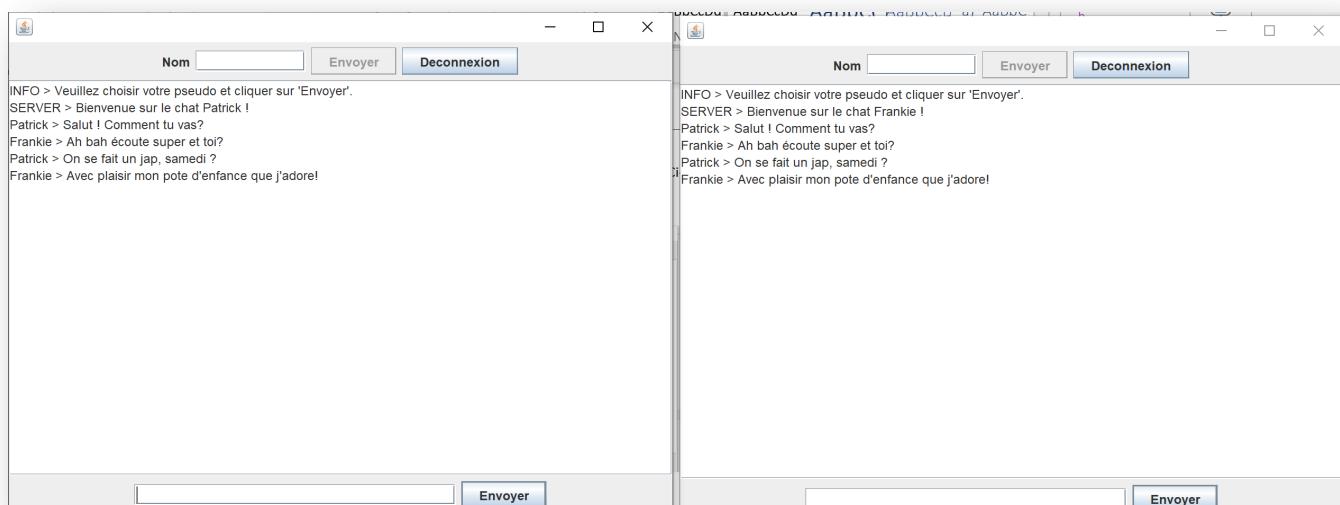
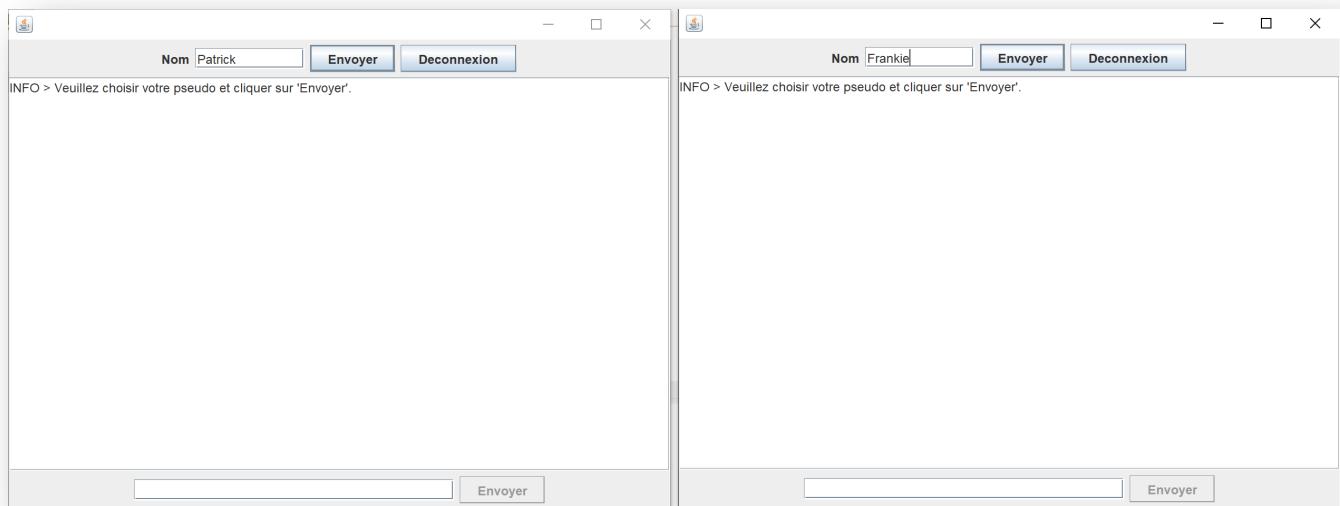
Les clients peuvent maintenant communiquer !

Choisissez un pseudo : John  
Salut mon pote !  
John > Salut mon pote !

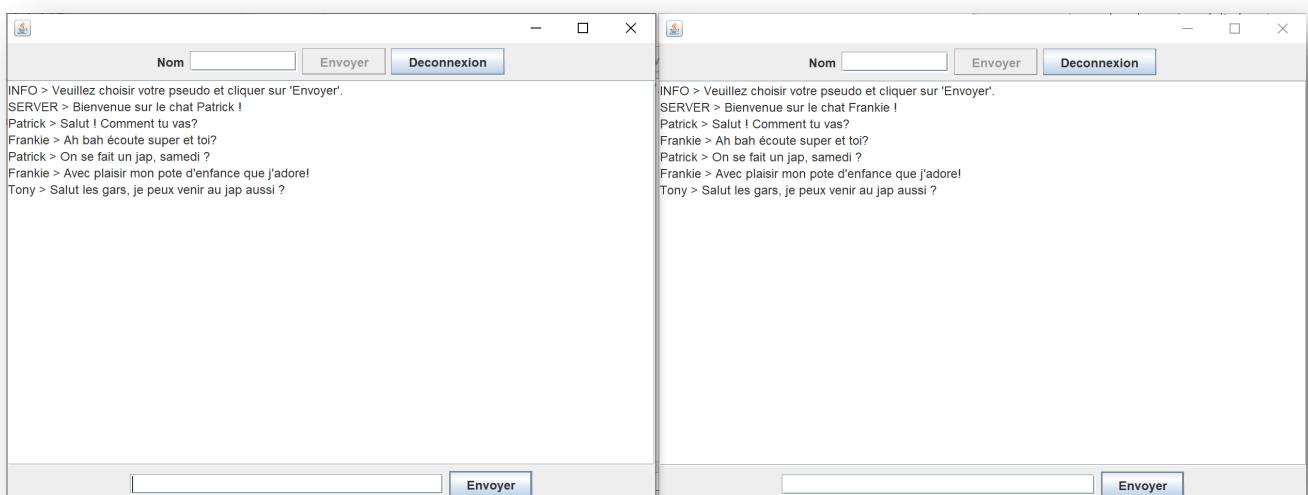
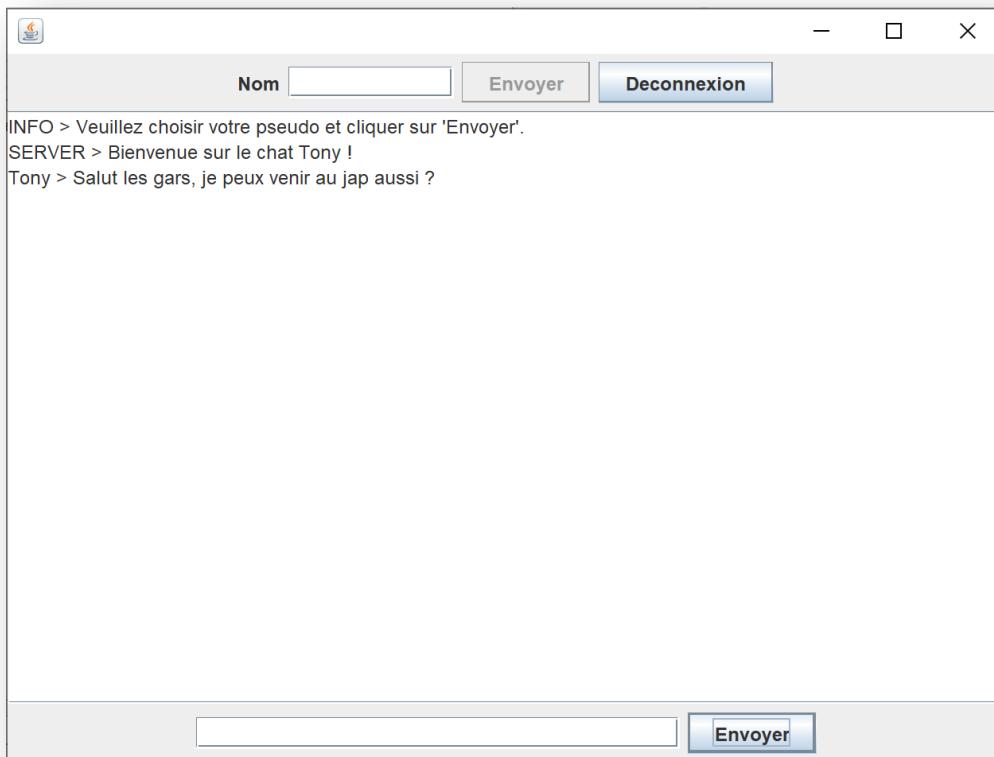
Choisissez un pseudo : Michel  
John > Salut mon pote !

c) Version 2 (test similaire pour la v3)

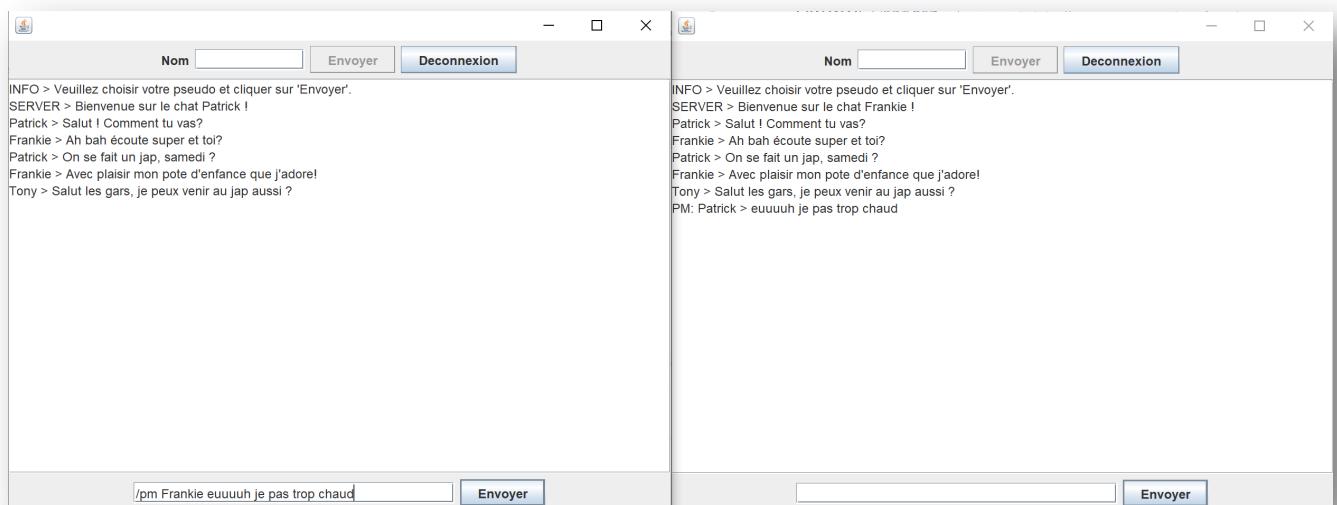
On lance le serveur comme sur les autres version. Ci-dessous, nous pouvons voir deux clients lancés, il peut y avoir une multitudes d'autres clients.



Ajoutons un troisième utilisateur !



Dans cette situation Patrick peut envoyer un message privé à Frankie pour que Tony ne puisse pas le lire, en faisant **/PM [username] [message]**.



Il y a d'autres fonctionnalités de ce type que nous avons implémenté comme **/shout**, protocole qui permet de passer tous les caractères en UPPERCASE.

Pour voir tous les protocoles et leurs fonctionnalités, il vous suffit de taper **/help**.

Tony > help  
INFO > /pm pseudo message : Envoyer un message privé  
INFO > /shout message : Envoyer un message en CAPSLOCK  
INFO > /help : Afficher l'aide  
INFO > /exit : Déconnecter la session

Pour vous déconnecter correctement nous avons implémenté **Deconnexion** le bouton qui permet de correctement éteindre le programme client en action.

## Conclusion

Dans le projet nous avons progressé sur plusieurs aspects. Tout d'abord la notion de socket en Java, et la communication inter-processus. Une fois établie, nous avons pu implémenter dans la V1 toutes les demandes du cahier des charges (n-clients, multithread, etc.).

Une fois le projet réussi, nous nous sommes concentrés sur des fonctionnalités supplémentaires que l'on peut tester avec la commande **/help**. Enfin nous avons pu intégrer le GUI.

La V3 est la version définitive du projet, nous avons encore amélioré le multithreading avec des commandes d'attente et de réveil que nous n'avions pas réussi à intégrer dans les versions précédentes.

Nous espérons que vous allez pouvoir tester et apprécier notre application.