Compte Rendu : Projet Multitâches

Introduction

Dans le cadre de notre projet Multitâches, nous avons réalisé un serveur et un client communiquant entre eux à l'aide de sockets montés sur le serveur local via le port 8080.

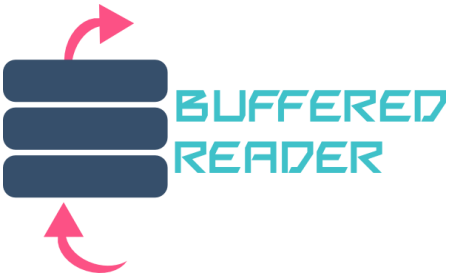
Le projet a été réalisé par M. Ansari Antoine et M. Vovard Hugo. Il est disponible à l'adresse web :

https://github.com/hvovar39/ProjetMutlitache
lien de téléchargement : https://github.com/hvovar39/ProjetMutlitache/archive/main.zip

Le serveur est écrit en Java. Il implémente un serveur socket sur le port 8080. Il permet à la manière de what's app ou messenger de faire communiquer les gens. Ceci à une échelle plus petite bien sûr car le but n'était pas de créer le nouveau réseau social du moment mais bel et bien de manipuler des sockets et des threads pour permettre une communication entre deux machines. En effet, le serveur n'est accessible que par réseau local en utilisant le port 8080. Les clients associés doivent donc être lancés sur la même machine pour être fonctionnels. Nous sommes donc parvenus à un résultat fonctionnel et efficace qui permet la communication avec les serveurs et la transmission d'infos est effective.

Le client est lui aussi codé en java. Il permet d'orienter l'utilisateur sur ce qu'il doit entrer pour ainsi faire le lien entre l'utilisateur et le serveur gérant l'envoi de messages et la connexion. Nous sommes parvenus à un résultat fonctionnel et correct bien que l'absence d'interface graphique rendant le projet moins « user-friendly » est dommageable.

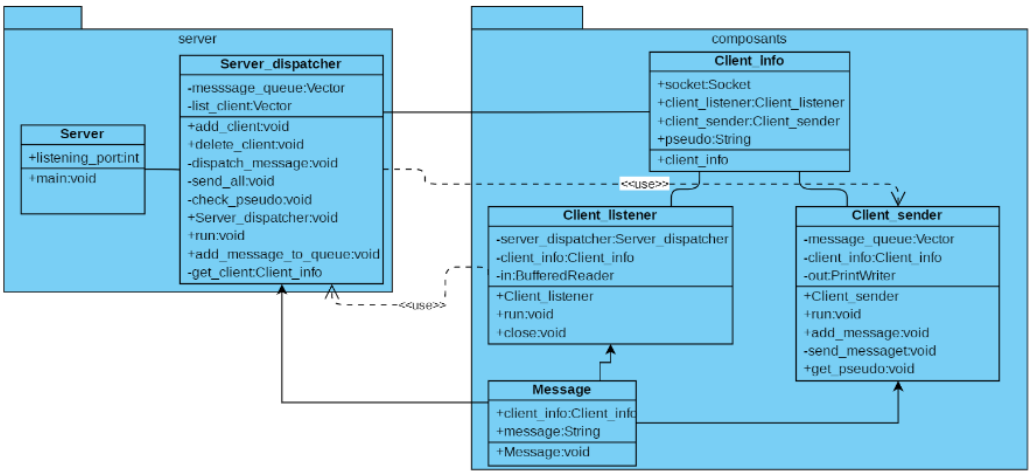


Partie 1 : Modélisation

Tous les documents contenus dans cette partie du rapport sont également disponibles dans le dossier DOC à la racine du projet.

a) Modélisation de la partie serveur



L'architecture du serveur peut être retrouvée ci-dessus, nous nous sommes inspirés du diagramme indiqué dans l'énoncé en le modifiant un peu à notre manière.

La partie **gauche** du diagramme contient le « main », ainsi que le dispatcher.

Sever : Ici le fichier « main » permet de tout simplement récupérer les connexions.

Server_dispatcher : Ici, le dispatcher permet de gérer les clients, les 2 threads associés à chaque client, les groupes, ainsi que l'aiguillage des messages.

La partie **droite** du diagramme contient les composants spécifiques.

Client_info : Ici, chaque instance correspond à un client connecté. Chaque client est donc associé de façon unique à une socket, un pseudo, un listener et un sender.

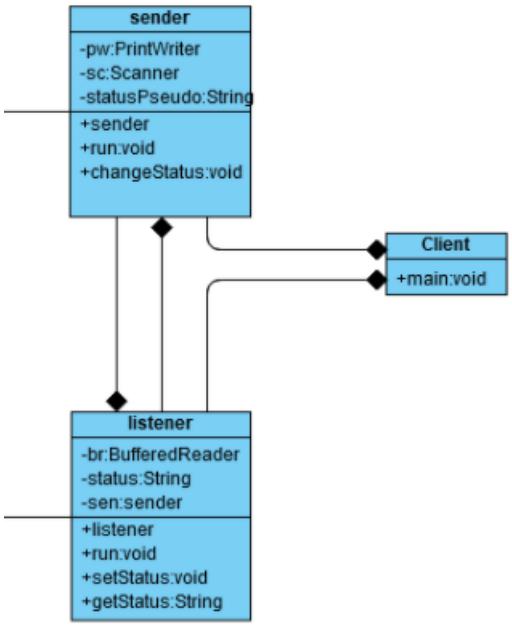
Client_listener : Gère la réception des messages.

Client_sender : Gère l'envoie des messages.

Message : classe spécifique permettant d'associer un "message" (String) au client expéditeur.

Team : classe spécifique permettant de stocker une liste de client appartenant à un groupe.

b) Modélisation de la partie client



L'architecture du client peut être retrouvée ci-dessus. L'architecture est basique mais efficace. Deux threads sont associés au client qui a pour but de lancer ces threads ainsi que le socket. Nous nous sommes inspirés du schéma proposé dans l'énoncé mais en y ajoutant cette fois un thread listener qui facilite l'écoute des messages envoyés par le serveur.

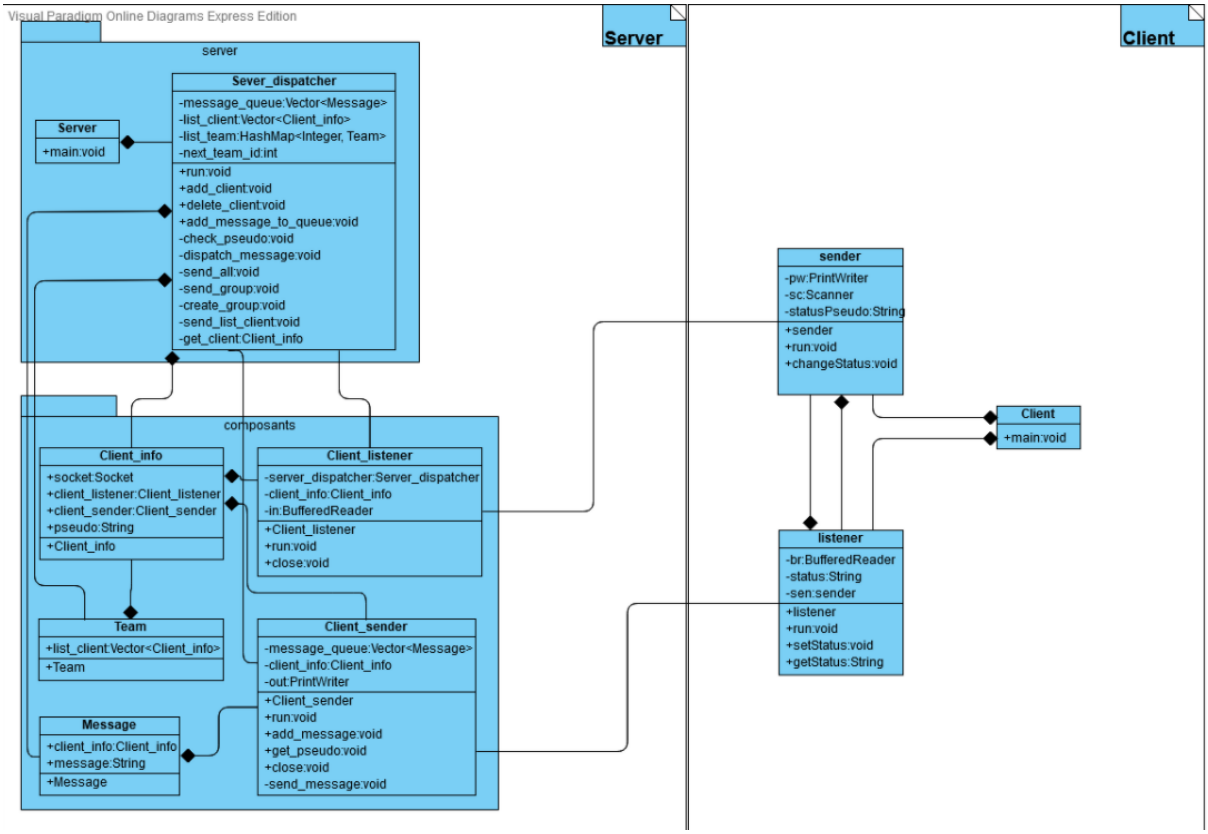
La partie client contient le client, ainsi que les threads Sender et Listener.

Le **client** (client.java) permet de lancer le serveur socket, ainsi que les threads dont il initialise les objets BufferReader pour lire les entrées seveur ainsi que PrintWriter pour envoyer des messages au serveur.

Le **Listener** (listener.java) permet d'écouter les sorties serveur et d'informer l'utilisateur de ce qu'il reçoit.

Le **Sender** (sender.java) permet d'écouter les entrées client et d'informer le serveur des entrées utilisateur.
Une grande partie de ses fonctions réside dans le guidage de l'utilisateur dans la partie envoie au serveur.

c) Modélisation générale du projet combiné



Voici ci-dessus le diagramme général du projet, on voit bien ici le lien entre la partie serveur et la partie client via les threads listener et sender comme indiqué au-dessus. L'architecture est efficace et nous avons mis l'accent sur l'optimisation du programme pour ne pas créer de fonctions trop redondantes ou inutiles.

Ce lien entre le serveur et le client est permis via la création et l'utilisation d'un socket. L'utilisation des classes BufferReader (pour lire les sorties du serveur) et de PrintWriter (pour envoyer des entrées au serveur) permet également cette liaison via les threads Sender et Listener.

Partie 2 : Fonctionnalités

a) Serveur

```
88          //Attends la fin des thread sender et listener associé à ci puis enlève ci de la liste
89          public void delete_client (Client_info ci) {
90                  synchronized (this.list_client) {
91                          try {
92                                  this.list_client.remove(ci);
93                                  ci.client_listener.close();
94                                  this.wait();
95                                  ci.client_sender.close();
96                                  ci.socket.close();
97                          }catch (Exception e) {
98                                  System.out.println("ERROR fermeture : "+ e + "Un problem est survenu lors de la fermeture de la socket associé"
99                                                  + " au client "+ ci.pseudo);
00                          }
01                  }
```

Code du dispatcher

```
35          public void close() {
36                  try {
37                          synchronized (this) {
38                                  this.wait();
39                                  this.running = false;
40                                  this.notify();
41                                  this.in.close();
42                                  this.server_dispatcher.notify();
43                          }
```

Code du listener

Cette partie du code permet de supprimer proprement un client. Pour cela, des méthodes sont disponible dans le listener et le sender. Pour s'assurer que la fermeture des éléments se fait dans le bon ordre, une synchronisation à l'aide de wait et de notify est nécessaire.

```
22              while (true) {
23                      synchronized (this.message_queue) {
24                              if (!this.message_queue.isEmpty()) {
25                                      msg = this.message_queue.firstElement();
26                                      this.message_queue.remove(0);
27                                      System.out.println("From "+ msg.client_info.pseudo +" :\n" + msg.message);
28
29                                      String[] tab = msg.message.split(";");
30
31                                      //Cas du choix du pseudo
32                                      if (tab[0].equals("pseudo"))
33                                              this.check_pseudo(msg);
34
35                                      //Cas de gestion
36                                      else if (tab[0].equals("gestion")) {
37                                              if (tab[1].equals("close"))
38                                                      this.delete_client(msg.client_info);
39                                              else if (tab[1].equals("list"))
40                                                      this.send_list_client(msg);
41                                              else
42                                                      msg.client_info.client_sender.add_message(new Message (null, "gestion;infos;"+ msg.client_info.pseudo));
43                                      }
44
45                                      //Cas de création d'un groupe
46                                      else if (tab[0].equals("creategroupe"))
47                                              this.create_group(msg);
48
49                                      //Cas d'un message de groupe
50                                      else if (tab[0].equals("groupe"))
51                                              this.send_groupe(msg);
53                                      //Cas d'un message public
54                                      else if (tab[0].equals(""))
55                                              send_all(msg);
56
57                                      //Cas d'un message privé
58                                      else
59                                              this.dispatch_message(msg);
```

La boucle d'aiguillage du run du dispatcher permet de récupérer l'information du type de message (gestion, groupe, privé, public) et d'agir en conséquence.

```
25                  while (this.running) {
26                          synchronized (this.message_queue) {
27                                  if (!this.message_queue.isEmpty()) {
28                                          System.out.println(this.message_queue);
29                                          msg = this.message_queue.firstElement();
30                                          this.message_queue.remove(0);
31
32                                          //Cas du choix du pseudo
33                                          if (msg.message.equals("pseudo")) {
34                                                  this.get_pseudo(flag_pseudo);
35                                                  flag_pseudo --;
36                                          }
37                                          else if (msg.message.equals("pseudook")) {
38                                                  flag_pseudo = 1;
39                                                  this.get_pseudo(flag_pseudo);
40                                          }
41
42                                          //Cas général
43                                          else if (!msg.message.equals(""))
44                                                  send_message (msg);
```

Code du sender
La boucle d'envoie (dans le run du Client_sender) permet d'accéder à sa queue de message et de les envoyer au client lié. Le seul cas particulier ici est celui du choix de pseudo. Pour le reste des messages, le formatage est effectué avant l'ajout à la queue.

b) Client

Le client java utilise une structure en 3 fichiers. Un client, un listener et un sender.

Le client implémente le socket et lance les threads présent dans les deux autres fichiers.

```
//Initialisation
Scanner sc = new Scanner(System.in);
Socket client = new Socket( host: "localhost", port: 8080);

// Flux qui permet de recevoir
BufferedReader br = new BufferedReader( new InputStreamReader (client.getInputStream()));
//Flux qui permet d'envoyer
PrintWriter pw = new PrintWriter(new BufferedWriter(new OutputStreamWriter(client.getOutputStream())), autoFlush: true);

//On initialise les threads en créant les objets thread sender et listener
sender sen = new sender(pw, sc);
//On associe sender à listener pour pouvoir avertir en cas de création d'un pseudo déjà utilisé
listener lis = new listener (br, sen);

//On lance les threads ici
lis.start();
sen.start();
```

On peut voir que la création du listener prend le sender en paramètre. La raison est que les deux sont liés et le listener va modifier une partie du sender dans le cadre de l'authentification (pseudo erreur).

```
while (true) {
    try {
        s = br.readLine();
        sen.changeStatus(s);
        if (s.equals("pseudo;ok")) {
            while (true) {
                s = br.readLine();
                //Permet de vérifier les réponses serveur via la nomenclature
                String[] message = s.split( regex: ";");
                if (message[0].equals("private"))
                    System.out.println(message[1] + " vous a envoyé un message : " + message[2]);
                if (message[0].equals("groupe"))
                {
                    if(message.length==2)
                        System.out.println("Felicitations, votre groupe a été créé et a pour ID : " + message[1]);
                    System.out.println(message[2] + " a envoyé un message au groupe " + message[1] + " : " + message[3]);
                }
                if (message[0].equals("gestion"))
                {
                    if (message[1].equals("infos"))
                        System.out.println("Vous êtes connectés sous le pseudo : "+message[2] + " (entrer pour continuer...)");
                    if (message[1].equals("list"))
                    {
                        System.out.println("Les personnes suivantes sont connéctées sur le serveur : (entrer pour continuer...)
                        for (int i = 2 ; i<message.length;i++) {
                            System.out.println(message[i]);
                        }
                    }
                }
                else {
                    //Affiche les messages public uniquement
                    if (message.length==2)
                        System.out.println("L'utilisateur " + message[0] +  " a envoyé un message public : " + message[1]);
```

Code du listener

Ici on écoute donc les réponses du serveur en utilisant la classe bufferedReader implémentée dans le client.java. Grace à la nomenclature, on peut vérifier les réponses du serveur et donc afficher des messages en conséquence. On voit également comme précisé précédemment qu'au début on appelle la méthode change status du sender pour indiquer si l'authentification est fonctionnelle. (Voir partie suivante)

```
String pseudo;
do {
    System.out.println("Veuillez entrer un pseudo pour vous connecter\n");
    pseudo = sc.nextLine();
    s = "pseudo;" + pseudo;
    pw.println(s);
    System.out.println("Bienvenue " + pseudo + ", entrez pour continuer...");
    sc.nextLine();
    if (this.statusPseudo.equals("pseudo;err"))
        System.out.println("Oops le pseudo est déjà utilisé, veuillez en entrer un nouveau");
} while (!this.statusPseudo.equals("pseudo;ok"));
while (true) {
    String choice;
    do {
        System.out.println("Voulez vous envoyer des messages ou gérer votre session ?\n(Veuillez entrer message ou gestion)
        choice = sc.nextLine();
        if (choice.equals("quit!"))
            pw.println("gestion;close");
    } while (!choice.equals("message") && !choice.equals("gestion"));
    if (choice.equals("message")) {
        String mess;
        System.out.println("Bienvenue dans le chat\n-------------------------------------");
        System.out.println("Entrez action :\nmessage privé : entrez private\nmessage public : entrez public\nmessage de grc
        String stat = sc.nextLine();
        switch (stat) {
            case "private":
                System.out.println("Veuillez entrer le pseudo du destinataire");
                s = "private;" + sc.nextLine() + ";";
                while (true) {
                    System.out.println("Veuillez entrer le message (entrez 'quit!' pour changer d'option)");
                    mess = sc.nextLine();
                    if (mess.equals("quit!"))
```

```
        }
    }
    else {
        String stat;
        do {
            System.out.println("Bienvenue dans l'onglet de gestion de session\n-------------------------------------")
            System.out.println("Entrez action :\nPour obtenir vos infos de session : entrez infos\nPour obtenir la list
            stat = sc.nextLine();
        } while (!stat.equals("close") && !stat.equals("infos") && !stat.equals("list"));
        s = "gestion;" + stat;
        pw.println(s);
        sc.nextLine();
```

Code du sender (en partie)

Le sender est implémenté pour guider l'utilisateur dans ses choix d'entrée. L'architecture du code est transparente et classique pour un client. Les entrées utilisateurs sont prises en compte et les nomenclature (pseudo ; /gestion ;) y sont ajoutés automatiquement pour rendre le projet plus « user-friendly ».

```
private PrintWriter pw;
private Scanner sc;
protected String statusPseudo;
```

Le sender possède en plus du scanner et du printwriter, un statuspseudo pour indiquer si le pseudo entré est déjà utilisé ou non.

```java
//Fonction pour le changement du statut du pseudo (pseudo;err si pseudo déjà utilisé)
public void changeStatus(String newStat) {
    this.statusPseudo = newStat;
}
```

Ici la méthode utilisée par le listener pour changer le status en pseudo;err en cas de problème.

c) Protocole de communication entre le serveur et le client

Le serveur et le client communiquent via un protocole formaté de messages afin de rendre l'exécution du programme plus efficace et plus claire.

Ci-dessus voici la nomenclature de notre protocole de communication entre le serveur et le client.

Création d'un groupe

- From Client to Server : **creategroupe;pseudo1;pseudo2;pseudo3;...;message**
- From Server to Client : **groupe;idgroupe**

Messages

Messages publics

- From Client to Server : **;message**
- From Server to Client: **pseudo_sender;message**

Messages privés

- From Client to Server : **private;pseudo_reciever;message**
- From Server to Client : **private;pseudo_sender;message**

Messages de groupe

- From Client to Server : **groupe;idgroupe;message**
- From Server to Client : **groupe;idgroupe;pseudo_sender;message**

Gestions

Fermeture connection

- From Client to Server : **gestion;close**

Infos clients

- From Client to Server : **gestion;infos**
- From server to Client : **gestion;infos;pseudo**

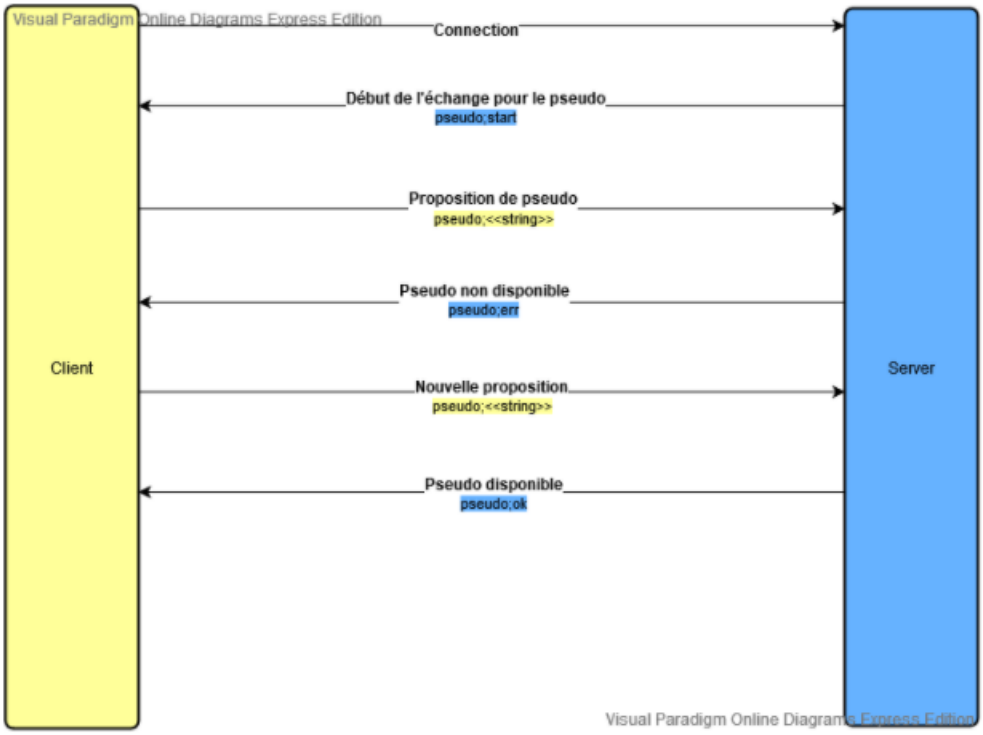
Liste des connectées

- From Client to Server : **gestion;list**
- From Server to Client : **gestion;list;pseudo1;pseudo2;pseudo3;...**

Les fonctions et les données sont ici séparées par un point-virgule. Cela a rendu la tache plus facile du coté serveur mais surtout client pour agir en fonction de la nomenclature des messages reçus. (voir exemple ci-dessous)

```
String[] message = s.split( regex: ";");
if (message[0].equals("private"))
    System.out.println(message[1] + " vous a envoyé un message : " + message[2]);
if (message[0].equals("groupe"))
{
```

Pour la partie authentification, nous avons également créé un protocole de communication qui sert notamment à vérifier si un pseudo n'existe pas déjà dans la couche métier. (exemple ci-dessous)

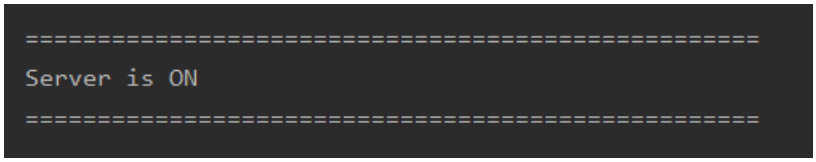


Cela permet d'indiquer au client si l'utilisateur a entré un pseudo valable ou non. (D'où la création de la méthode changeStatus décrit au-dessus.

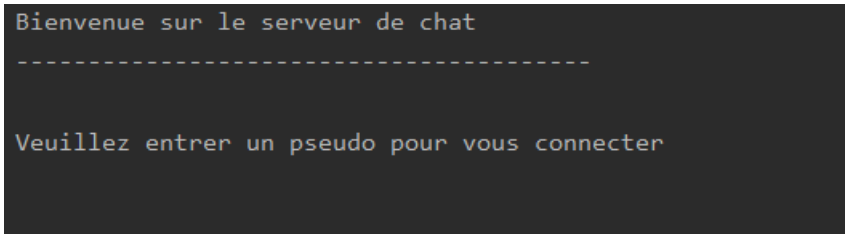
```
do {
    System.out.println("Veuillez entrer un pseudo pour vous connecter\n");
    pseudo = sc.nextLine();
    s = "pseudo;" + pseudo;
    pw.println(s);
    System.out.println("Bienvenue " + pseudo + ", entrez pour continuer...");
    sc.nextLine();
    if (this.statusPseudo.equals("pseudo;err"))
        System.out.println("Oops le pseudo est déjà utilisé, veuillez en entrer un nouveau");
} while (!this.statusPseudo.equals("pseudo;ok"));
```

Partie 3 : Exécution

Pour faire fonctionner le projet, il suffit de lancer un serveur. Une fois le serveur lancé, il affichera le message suivant.



C'est à ce moment là que les clients sont exécutables. Au lancement, les clients afficheront le message suivant pour permettre aux utilisateurs de se connecter.



Pour lancer les fichiers, il suffit d'exécuter les fonctions main java via l'invité de commande (installer java au préalable), dans le fichier Server.java situé dans src/server dans la partie ChatServeur et dans le fichier Client.java situé dans src/ dans la partie ChatClient.

Partie 4 : Conclusion

Nous avons réussi dans ce projet à mettre en œuvre un serveur, ainsi qu'un client exécutable en parallèle par plusieurs utilisateurs se connectant à l'unique serveur et communiquant entre eux.

Nous avons comme demandé réussi à mettre en place le serveur ainsi que le dispatcher qui permet de gérer les clients ainsi que l'aiguillage des messages.

Comme indiqué dans les paragraphes au-dessus, des processus de communication inter-processus ont été utilisés comme des fonctions de synchronize ainsi que wait et notify. Cependant il est à noter que l'utilisation de ces fonctions n'a pas été forcément partout nécessaire au vu de l'architecture de notre code.

Logiquement afin de faire communiquer le serveur et les clients une communication par socket a été mise en place simplement via le serveur local sur le port 8080.

Nous avons également implémenté une application couche métier qui permet de stocker le pseudo des utilisateurs se connectant à notre serveur.

Avec plus de temps, nous aurions probablement pu intégrer une interface graphique dont l'ébauche est disponible (FXML), mais le temps nous a rattrapé. Le client est tout de même fonctionnel et les threads permettent le bon envoi des données aux utilisateurs et la bonne réception de leurs entrées.

Nous sommes satisfaits de l'avancée du projet et de l'ensemble de ses fonctionnalités bien que gourmands de perfection, nous aurions aimé pouvoir mettre en place une interface graphique ainsi qu'un protocole de communication.