

RAPPORT SR03

PROJET 1

APPLICATION DE CHAT MULTI-THREAD EN JAVA

PRÉSENTATION GÉNÉRALE DU PROJET

Le projet consiste à concevoir une application de messagerie permettant à plusieurs utilisateurs de communiquer entre eux. Chaque utilisateur sera identifié par un pseudo qui se devra d'être unique. Nous avons pris le parti de limiter le nombre d'utilisateurs simultanés à 20. Le projet est codé en Java et doit être composé d'un serveur auquel se connectent des clients (les utilisateurs) par l'intermédiaires de sockets. Le programme doit utiliser des threads : 1 côté serveur pour intercepter les messages envoyés et 2 côté client pour intercepter les messages envoyés par l'utilisateur et intercepter les messages envoyés par le serveur.

En premier lieu il faut lancer le serveur. Si le serveur parvient à se lancer correctement, un message "Connexion au serveur réussie" apparaît sur la console. Dans le cas contraire, une erreur apparaît. Il faut ensuite que les utilisateurs se connectent. Pour cela, il faut connecter un client au serveur. De même un message indique que la connexion a réussi. Puis il est demandé à l'utilisateur de rentrer un pseudo. Le programme vérifie que le pseudo n'est pas vide (retour à la ligne ou blank) et qu'il n'est pas déjà pris par un utilisateur déjà connecté. Si ce n'est pas le cas, l'utilisateur peut rejoindre la conversation. Un message lui stipule qu'il faut rentrer "exit" pour quitter.

Une fois l'utilisateur connecté il peut engager la discussion en écrivant dans l'invite de commande Pour chaque utilisateur, une nouvelle console à travers laquelle il communiquera s'ouvre. Une fois un nouvel utilisateur connecté, un message s'affiche sur les consoles des autres utilisateurs pour les avertir de l'arrivée du nouveau venu dans la discussion.

Lorsqu'un utilisateur écrit un message, celui-ci apparaît en couleur sur sa console (si vous faites tourner l'application sur Eclipse) et est précédé de son pseudo sur les consoles de tous les autres utilisateurs. Comme stipulé précédemment, pour quitter la conversation l'utilisateur doit taper "exit" dans la console. Un message lui indique ensuite qu'il a quitté la conversation ("au revoir [pseudo de l'utilisateur]"). Sur les consoles de tous les utilisateurs encore connectés s'affiche un message les prévenant de son départ. Les utilisateurs encore connectés peuvent bien sûr continuer à discuter. Les messages qu'ils échangeront ne seront pas visibles de l'utilisateur déconnecté puisque la communication est rompue entre lui et le serveur et donc entre lui et les autres utilisateurs. Lorsqu'il ne reste qu'un seul utilisateur connecté, celui-ci peut continuer à écrire dans la console mais les messages ne seront envoyés à personne.

EXPLICATION DU CODE

Server.java

Variables :

serverSocket : serveur de connexion qui va attendre que les clients se connectent par l'intermédiaire de sockets de communication

clientSocket : serveur de communication qui se connecte au serveur de connexion et va permettre de dialoguer avec le client.

maxClients : nous avons décidé pour des raisons pratiques de limiter à 20 le nombre d'utilisateurs simultanés. Cela permet d'éviter que le programme ne crash à cause d'un trop grand nombre de connexion (c'est un petit programme, il n'a pas la carrure d'une grosse messagerie instantanée) et par ailleurs cela permet d'utiliser de manière simple des tableaux puisque leur dimension est connue à l'avance.

clients : tableau qui stocke les objets sockets qui se connectent au serveur. Le tableau stocke en réalité les threads qui vont intercepter les messages envoyés par le client.

portNumber : nous avons choisi de nous connecter au port 3333.

Le socket de connexion se connecte au port 3333. Nous lançons ensuite une boucle infinie pour qu'il continue d'accepter les demandes entrantes. Quand une demande arrive, il l'accepte et la stocke dans clientSocket. Puis, nous allons l'ajouter au tableau clients et lancer un thread pour intercepter les messages en provenance du client. Nous lançons également un thread qui va surveiller que ce client ne s'arrête pas de manière inattendue. Si notre tableau contient déjà 20 clients, nous affichons un message pour prévenir que la conversation est pleine et ne peut accepter de nouveaux utilisateurs auquel cas on ferme le socket.

Client.java

Variables :

clientSocket : le socket de communication qui va chercher à se connecter au serveur.

out : le stream de sortie.

in : le stream d'entrée.

inputLine : les messages entrants en provenance du serveur.

closed : booléen qui passe à true quand le socket doit être fermé.

On ouvre le socket sur le localhost au port 3333 (comme le serveur). On ouvre également inputLine qui va lire tous les messages entrants, out qui va écrire sur le socket et in qui va lire dessus.

Nous nous assurons que tout a été initialisé correctement pour pouvoir écrire des données sur le socket. Puis nous ouvrons un thread pour pouvoir lire les données venant du serveur. Tant que la variable closed est à false, nous récupérons via inputLine tous les messages en provenance du serveur et nous les écrivons sur le

socket via la variable out. Tant que nous ne recevons pas du serveur le message “au revoir [pseudo de l'utilisateur]” qui indique que l'utilisateur s'est déconnecté, nous continuons de lire les données en provenance du serveur. Quand c'est le cas, nous passons la variable closed à true et nous fermons les deux streams (in et out) et le socket.

ClientThread

Variables :

clientName : le pseudo de l'utilisateur passé au socket

out : le stream de sortie.

in : le stream d'entrée.

clientSocket : le socket de communication qui va chercher à se connecter au serveur.

clients : tableau qui stocke les objets sockets qui se connectent au serveur. Le tableau stocke en réalité les threads qui vont intercepter les messages envoyés par le client.

maxClients : le nombre maximum de connexions simultanées acceptées (20).

ClientThread est utilisé par le serveur pour intercepter les messages en provenance du client comme évoqué précédemment. En premier lieu, nous initialisons toutes les variables avec les données reçues par le serveur lors de la connexion d'un client. Puis, nous allons ouvrir une boucle while dont l'utilisateur pourra sortir en donnant un pseudo répondant aux critères fournis dans la boucle. Nous nous assurons que le pseudo fourni n'est pas vide (blank ou retour à la ligne) puis nous parcourons les pseudos des autres utilisateurs en parcourant les clientName du tableau clients. Si le pseudo n'a pas déjà été utilisé, nous sortons de la boucle. Via le stream out, nous indiquons à l'utilisateur qu'il a rejoint la discussion puis on crée un bloc de synchronisation via synchronized. On va ajouter le nouvel utilisateur (son nom) à la liste des clients puis on va de façon synchronisée afficher sur tous les autres clients un message pour indiquer la connexion d'un nouvel utilisateur. Nous pouvons alors lancer la conversation. Nous allons créer une variable line qui va lire les messages sur le socket via le stream d'entrée in. De la même manière, nous allons via synchronized afficher les messages qui passent par line chez tous les autres utilisateurs. Nous allons faire cela dans une boucle while qui sera interrompue seulement quand le message passé dans le socket sera “exit”. A ce moment-là, encore une fois de la même manière, nous allons afficher chez tous les utilisateurs que l'utilisateur X s'est déconnecté. Nous allons stopper le thread qui était chargé de surveiller les interruptions inattendues du client X afin qu'il n'interprète pas cette déconnexion comme non prévue. Sur la console de l'utilisateur X, nous allons afficher un message d'au revoir.

Il faut maintenant effacer le thread. Dans le tableau client, nous mettons la case correspondante à null. Elle pourra être attribuée à un nouvel utilisateur qui se connectera ultérieurement. Comme dans le code du client, nous fermons les deux streams (in et out) et le socket.

Nous avons également implémenté dans `ClientThread` la fonction `preventUnexpectedInterruption`. Cette fonction va envoyer à chaque client un message pour lui indiquer qu'un autre client a été déconnecté de manière inattendue. Il va en outre supprimer ledit client du tableau `clients`.

UnexpectedClientInterruption

Variables :

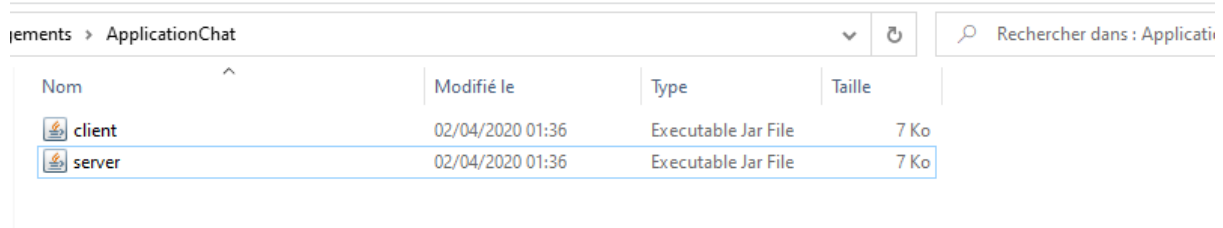
`client` : le client surveillé par le thread.

`userHasLeft` : booléen qui passe à `true` si le client a été arrêté par l'utilisateur.

On lance une boucle infinie pour surveiller l'activité du client tant qu'il tourne. Si le client meurt (méthode `client.isAlive()` renvoie `false`) on vérifie si cela est du fait de l'utilisateur ou s'il s'agit d'un arrêt inattendu. Pour cela, on vérifie la valeur de la variable `userHasLeft`. Si le client est arrêté par l'utilisateur la fonction `stopThread` aura passé à `true` cette variable. On va alors lancer la fonction `preventUnexpectedInterruption` contenu dans `ClientThread` pour supprimer le client et prévenir les autres utilisateurs de cette déconnexion.

DÉMONSTRATION

Dans cette démonstration, nous allons lancer le programme à partir de l'invite de commande Windows et non à partir de l'IDE pour des raisons pratiques (ne pas avoir à switcher entre les différentes consoles depuis l'IDE). Il est bien sûr possible de lancer l'application directement depuis Eclipse (ce qui permet en prime d'avoir les messages de la console courante en vert pour les distinguer des messages entrants).

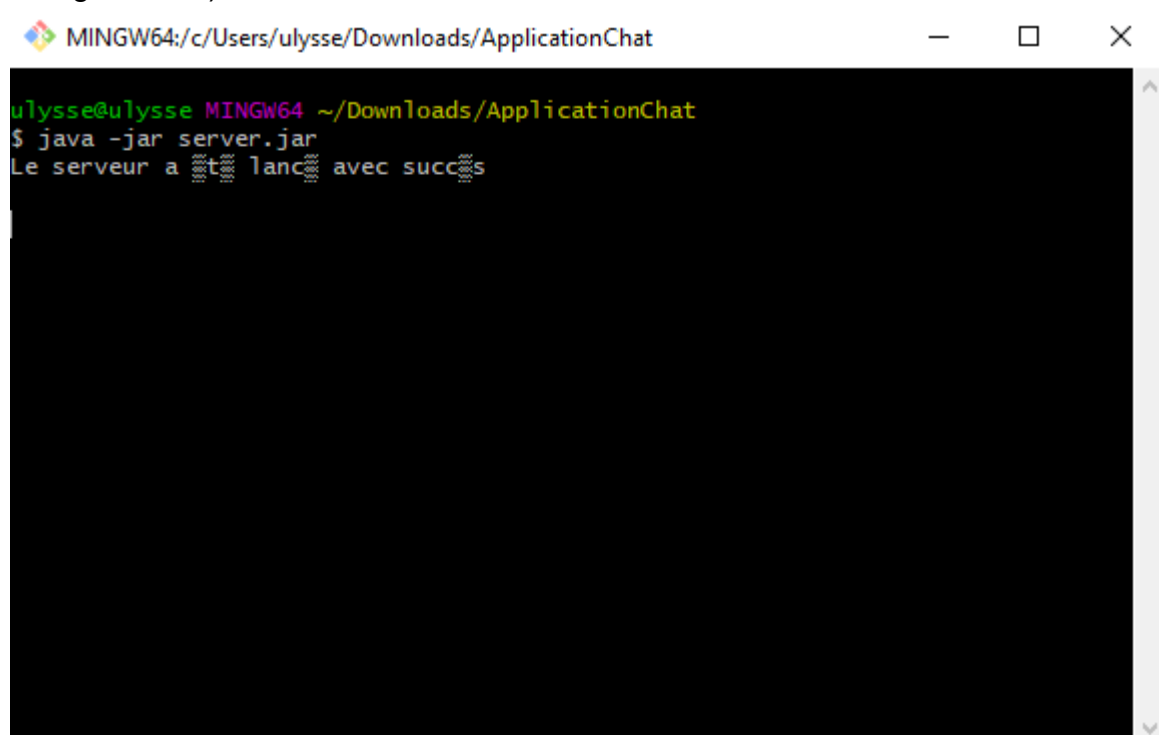


Resources > ApplicationChat				Rechercher dans : Applicati
Nom	Modifié le	Type	Taille	
client	02/04/2020 01:36	Executable Jar File	7 Ko	
server	02/04/2020 01:36	Executable Jar File	7 Ko	

Nous avons 2 exécutables. Un pour le client et un pour le serveur. Nous allons d'abord ouvrir un terminal pour lancer le serveur et y taper la commande suivante :

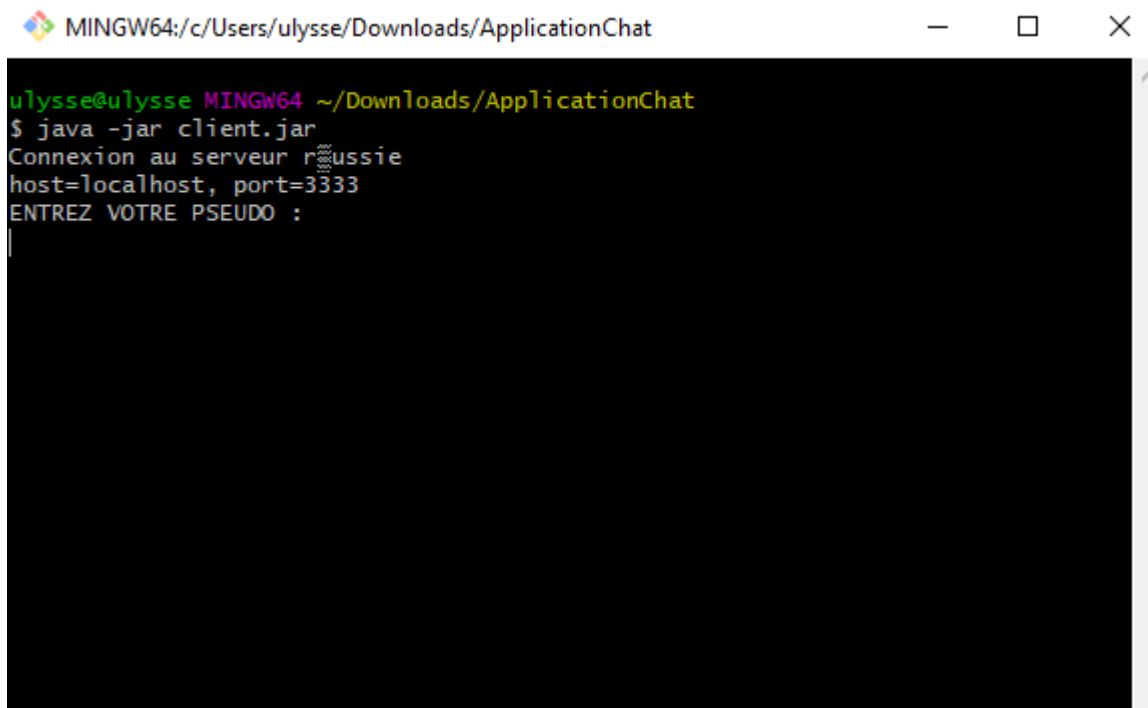
```
java -jar server.jar
```

On voit apparaître le message "le serveur a été lancé avec succès" (UTF-8 n'est malheureusement pas pris en charge par la console, ce qui cause des problèmes d'affichages pour certains caractères, lancer depuis Eclipse pour avoir la prise en charge UTF-8).



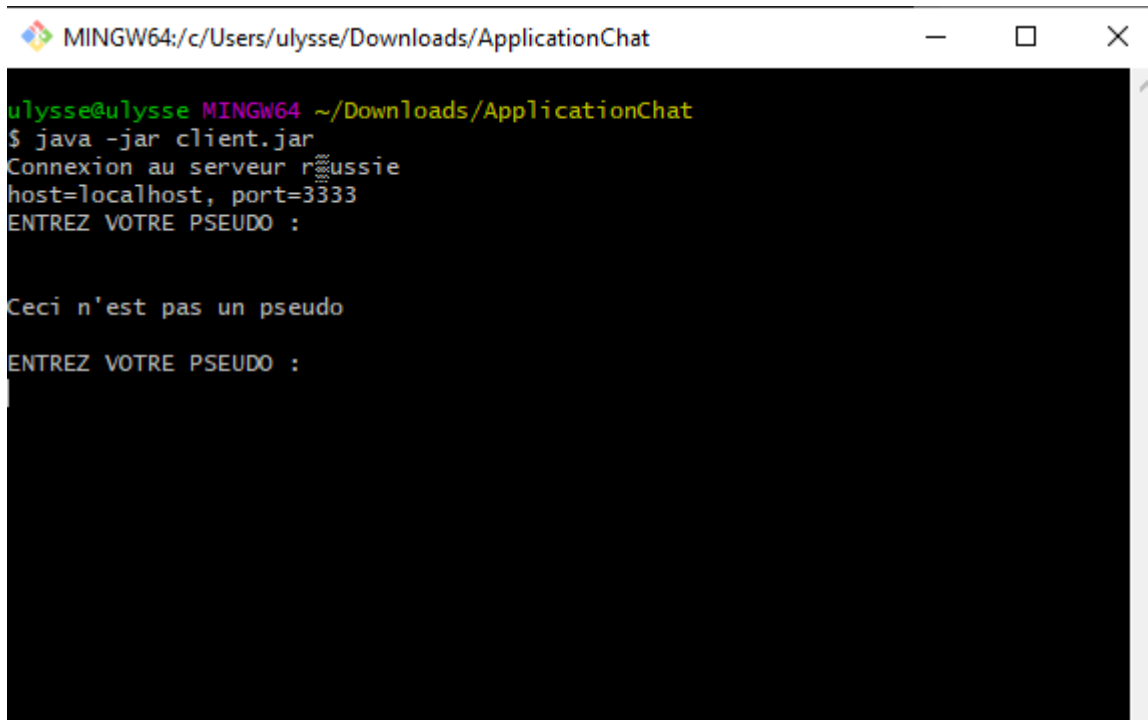
```
MINGW64:/c/Users/ulyse/Downloads/ApplicationChat
ulyse@ulyse MINGW64 ~/Downloads/ApplicationChat
$ java -jar server.jar
Le serveur a été lancé avec succès
```

On va maintenant lancer le premier client. Pour cela on va ouvrir un nouveau terminal et entrer la commande `java -jar client.jar`. L'application nous demande de rentrer un pseudo.



```
ulyse@ulyse MINGW64 ~/Downloads/ApplicationChat
$ java -jar client.jar
Connexion au serveur russe
host=localhost, port=3333
ENTREZ VOTRE PSEUDO :
|
```

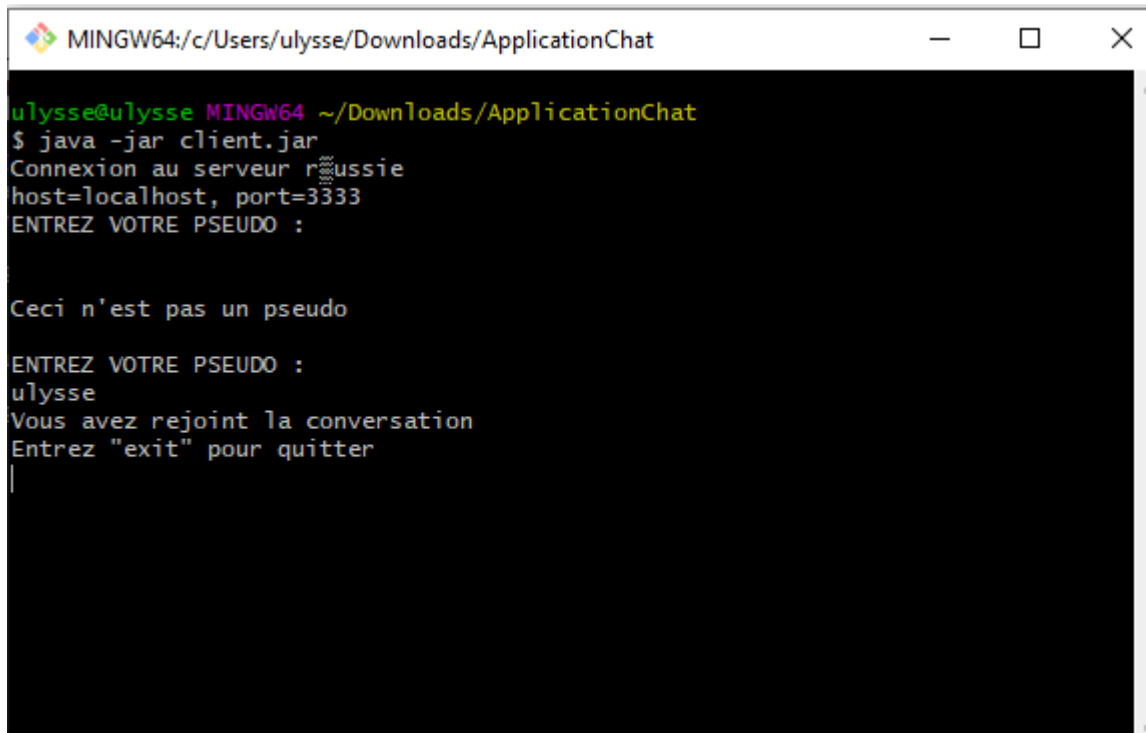
Si j'essaie d'entrer un pseudo vide, l'application me demande de rentrer un pseudo valide.



```
ulyse@ulyse MINGW64 ~/Downloads/ApplicationChat
$ java -jar client.jar
Connexion au serveur russe
host=localhost, port=3333
ENTREZ VOTRE PSEUDO :

Ceci n'est pas un pseudo
ENTREZ VOTRE PSEUDO :
|
```

Une fois mon pseudo accepté, je suis maintenant dans le salon de discussion.

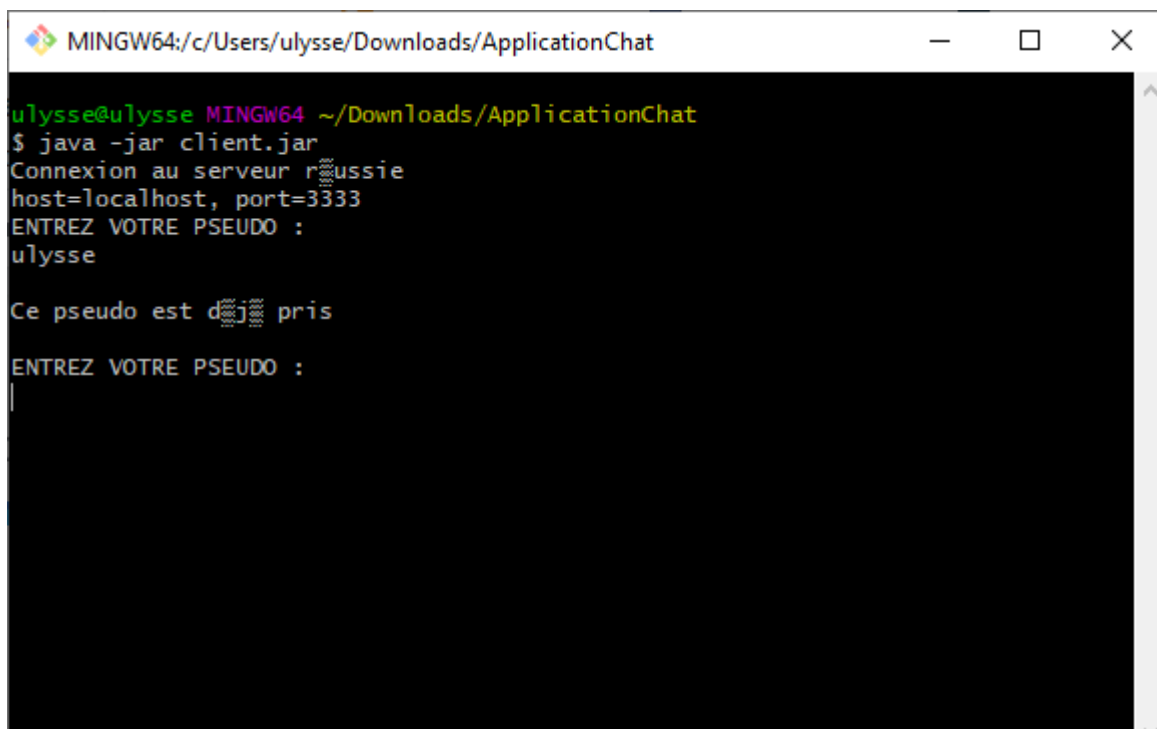


```
MINGW64:/c/Users/ulyse/Downloads/ApplicationChat
ulyse@ulyse MINGW64 ~/Downloads/ApplicationChat
$ java -jar client.jar
Connexion au serveur russia
host=localhost, port=3333
ENTREZ VOTRE PSEUDO :

Ceci n'est pas un pseudo

ENTREZ VOTRE PSEUDO :
ulyse
Vous avez rejoint la conversation
Entrez "exit" pour quitter
|
```

Nous allons pour les besoins de cette démonstration faire discuter entre eux 3 personnes et donc ouvrir deux autres terminaux et les connecter à la conversation en rentrant un pseudo. Si le pseudo est déjà pris, l'application me demande d'en choisir un nouveau.



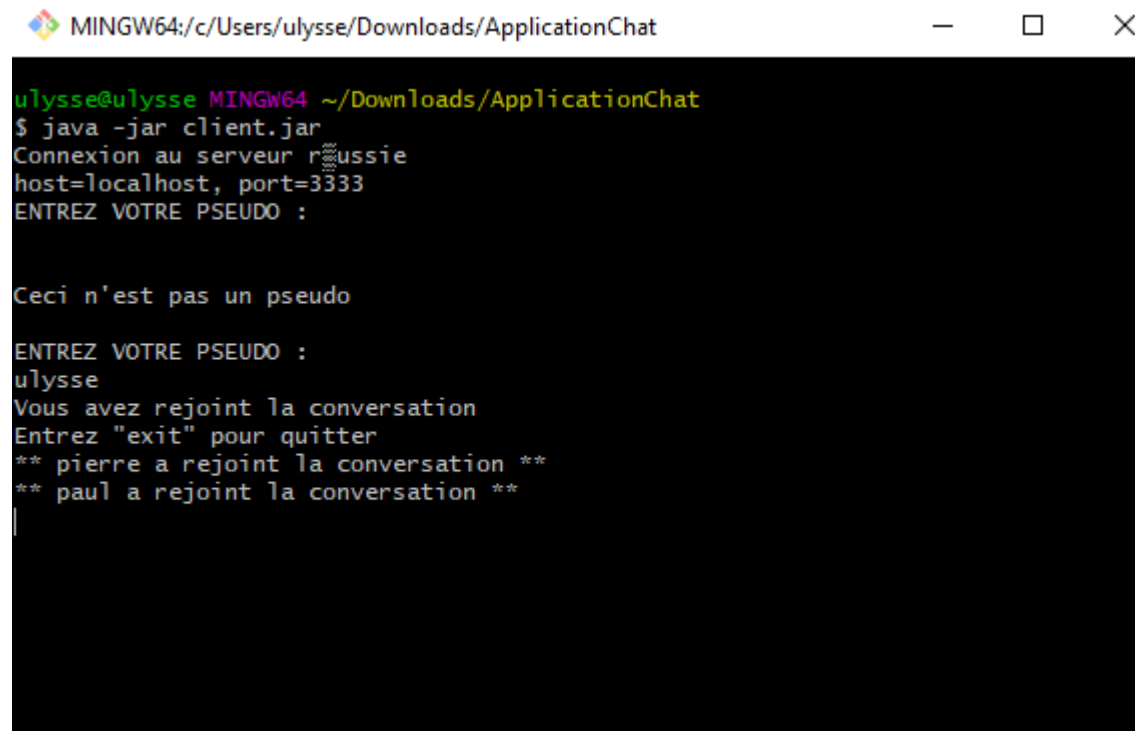
```
MINGW64:/c/Users/ulyse/Downloads/ApplicationChat
ulyse@ulyse MINGW64 ~/Downloads/ApplicationChat
$ java -jar client.jar
Connexion au serveur russia
host=localhost, port=3333
ENTREZ VOTRE PSEUDO :
ulyse

Ce pseudo est déjà pris

ENTREZ VOTRE PSEUDO :
|
```

Quand de nouveaux utilisateurs se connectent, les utilisateurs déjà connectés en sont informés.

Utilisateur 1

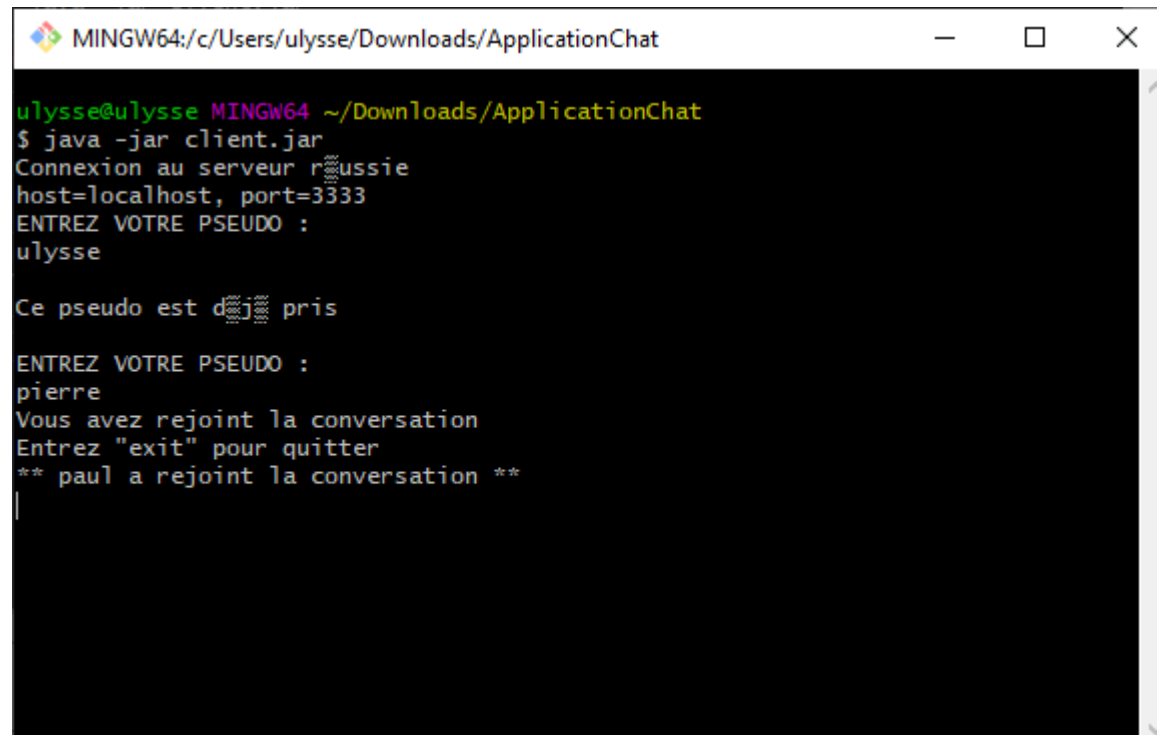
A screenshot of a terminal window titled "MINGW64:/c/Users/ulyse/Downloads/ApplicationChat". The prompt is "ulyse@ulyse MINGW64 ~/Downloads/ApplicationChat". The user enters the command "\$ java -jar client.jar". The output shows a successful connection to a server at localhost:3333. The user is prompted to enter a pseudo and enters "ulyse". A message "Ceci n'est pas un pseudo" is displayed. The user enters "ulyse" again, and a message "Vous avez rejoint la conversation" is shown. The terminal then displays two status messages: "** pierre a rejoint la conversation **" and "** paul a rejoint la conversation **".

```
ulyse@ulyse MINGW64 ~/Downloads/ApplicationChat
$ java -jar client.jar
Connexion au serveur réussie
host=localhost, port=3333
ENTREZ VOTRE PSEUDO :

Ceci n'est pas un pseudo

ENTREZ VOTRE PSEUDO :
ulyse
Vous avez rejoint la conversation
Entrez "exit" pour quitter
** pierre a rejoint la conversation **
** paul a rejoint la conversation **
```

Utilisateur 2

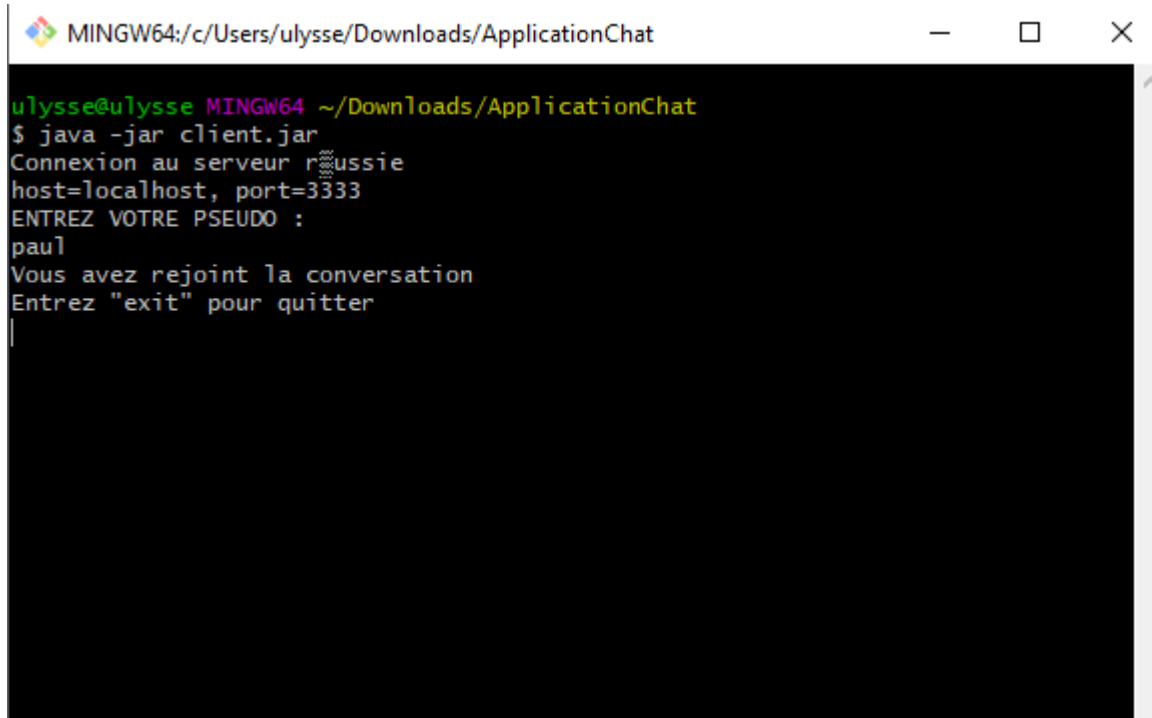
A screenshot of a terminal window titled "MINGW64:/c/Users/ulyse/Downloads/ApplicationChat". The prompt is "ulyse@ulyse MINGW64 ~/Downloads/ApplicationChat". The user enters the command "\$ java -jar client.jar". The output shows a successful connection to a server at localhost:3333. The user is prompted to enter a pseudo and enters "ulyse". A message "Ce pseudo est déjà pris" is displayed. The user enters "pierre", and a message "Vous avez rejoint la conversation" is shown. The terminal then displays a status message: "** paul a rejoint la conversation **".

```
ulyse@ulyse MINGW64 ~/Downloads/ApplicationChat
$ java -jar client.jar
Connexion au serveur réussie
host=localhost, port=3333
ENTREZ VOTRE PSEUDO :
ulyse

Ce pseudo est déjà pris

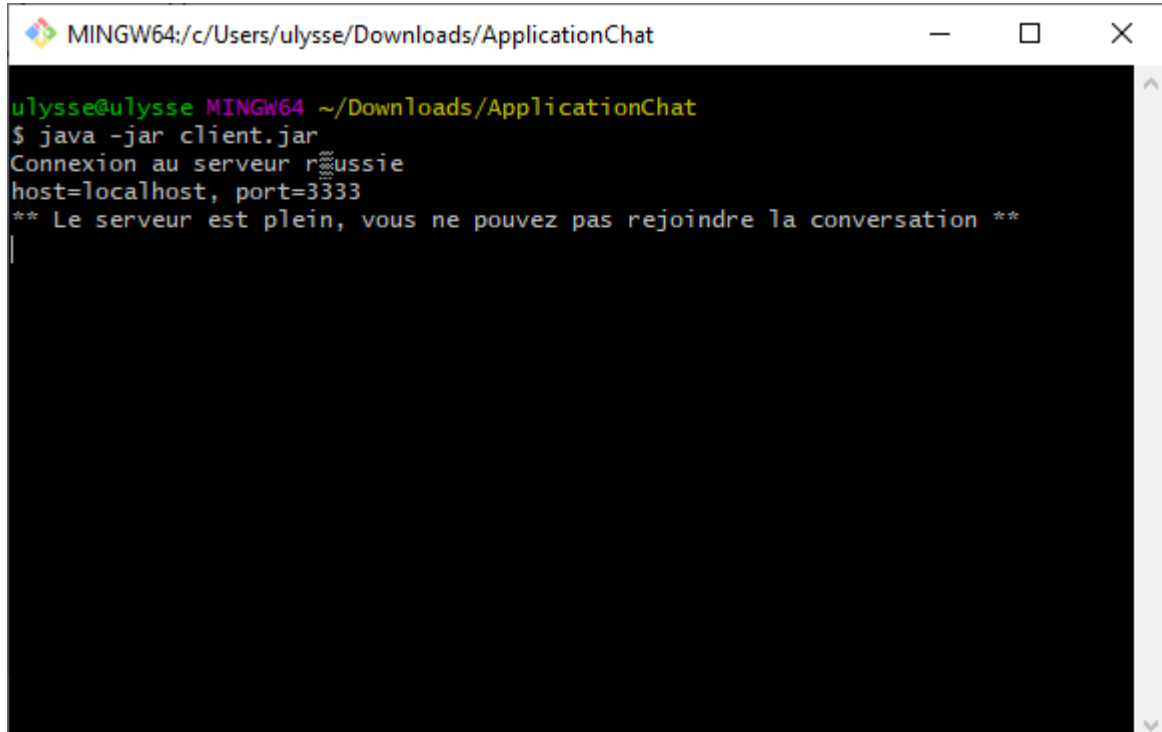
ENTREZ VOTRE PSEUDO :
pierre
Vous avez rejoint la conversation
Entrez "exit" pour quitter
** paul a rejoint la conversation **
```


Utilisateur 3



```
MINGW64:/c/Users/ulyse/Downloads/ApplicationChat
ulyse@ulyse MINGW64 ~/Downloads/ApplicationChat
$ java -jar client.jar
Connexion au serveur réussie
host=localhost, port=3333
ENTREZ VOTRE PSEUDO :
paul
Vous avez rejoint la conversation
Entrez "exit" pour quitter
|
```

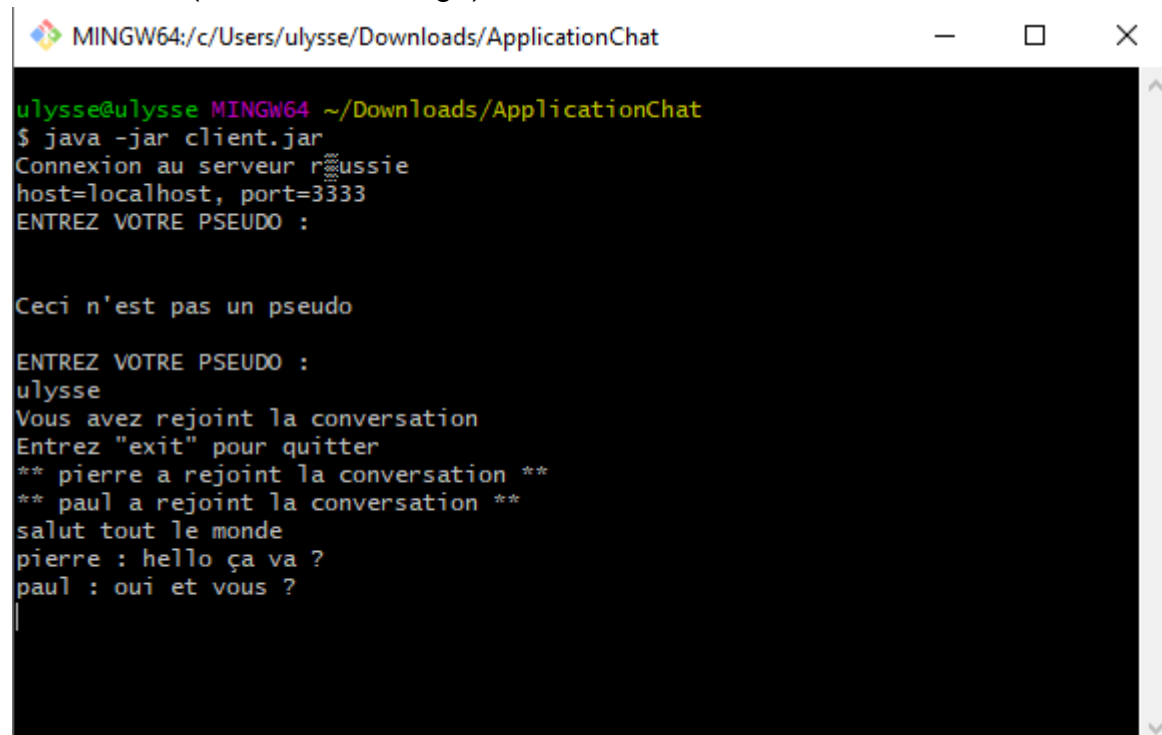
Pour les besoins de la démonstration, nous avons réduit à 3 le nombre d'utilisateurs simultanés autorisé (normalement 20). Si un 4ème utilisateur tente de se connecter voici ce qui apparaît sur la console.



```
MINGW64:/c/Users/ulyse/Downloads/ApplicationChat
ulyse@ulyse MINGW64 ~/Downloads/ApplicationChat
$ java -jar client.jar
Connexion au serveur réussie
host=localhost, port=3333
** Le serveur est plein, vous ne pouvez pas rejoindre la conversation **
|
```

Nous allons maintenant voir ce qui se passe quand nos utilisateurs interagissent entre eux.

Utilisateur 1 (début de l'échange) :



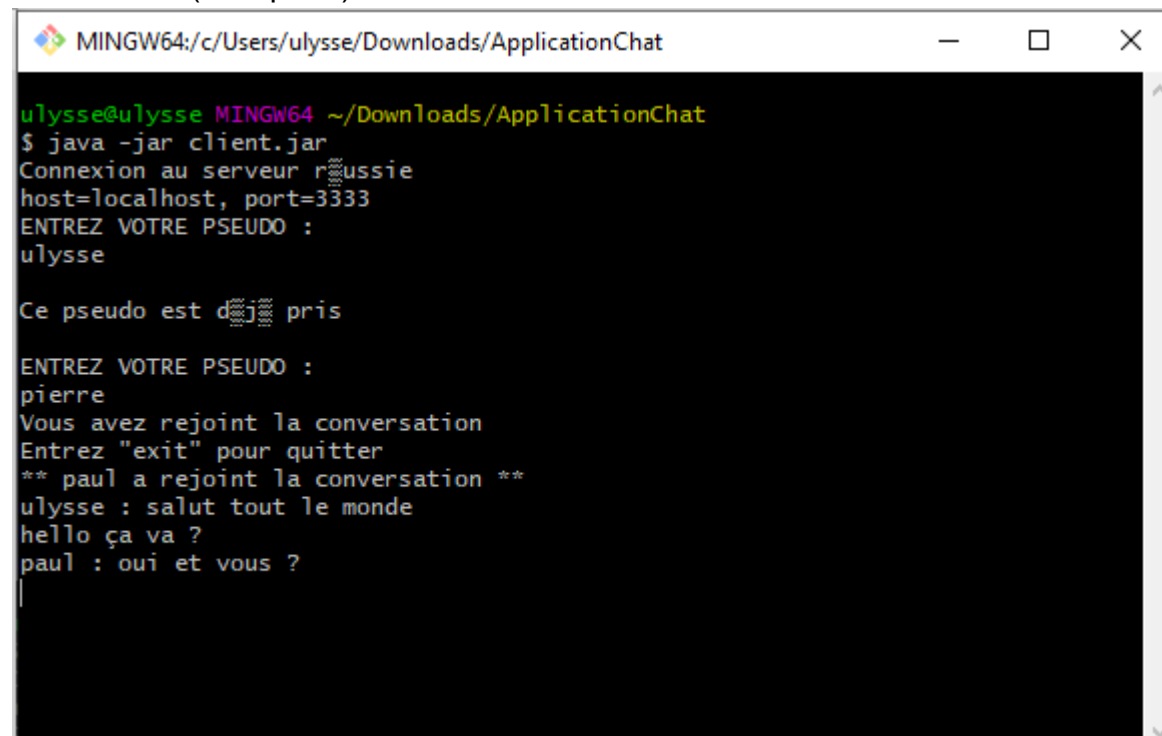
```
MINGW64:/c/Users/ulysse/Downloads/ApplicationChat

ulysse@ulysse MINGW64 ~/Downloads/ApplicationChat
$ java -jar client.jar
Connexion au serveur réussie
host=localhost, port=3333
ENTREZ VOTRE PSEUDO :

Ceci n'est pas un pseudo

ENTREZ VOTRE PSEUDO :
ulysse
Vous avez rejoint la conversation
Entrez "exit" pour quitter
** pierre a rejoint la conversation **
** paul a rejoint la conversation **
salut tout le monde
pierre : hello ça va ?
paul : oui et vous ?
|
```

Utilisateur 2 (lui répond) :



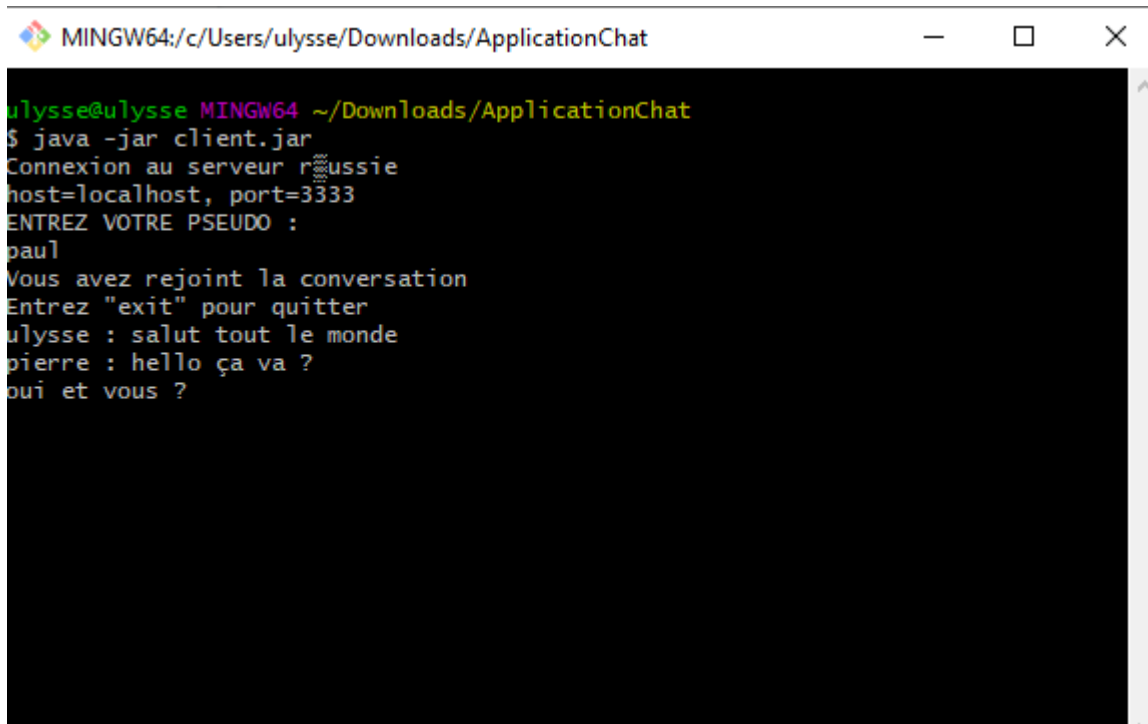
```
MINGW64:/c/Users/ulysse/Downloads/ApplicationChat

ulysse@ulysse MINGW64 ~/Downloads/ApplicationChat
$ java -jar client.jar
Connexion au serveur réussie
host=localhost, port=3333
ENTREZ VOTRE PSEUDO :
ulysse

Ce pseudo est déjà pris

ENTREZ VOTRE PSEUDO :
pierre
Vous avez rejoint la conversation
Entrez "exit" pour quitter
** paul a rejoint la conversation **
ulysse : salut tout le monde
hello ça va ?
paul : oui et vous ?
|
```

Utilisateur 3 (leur répond) :

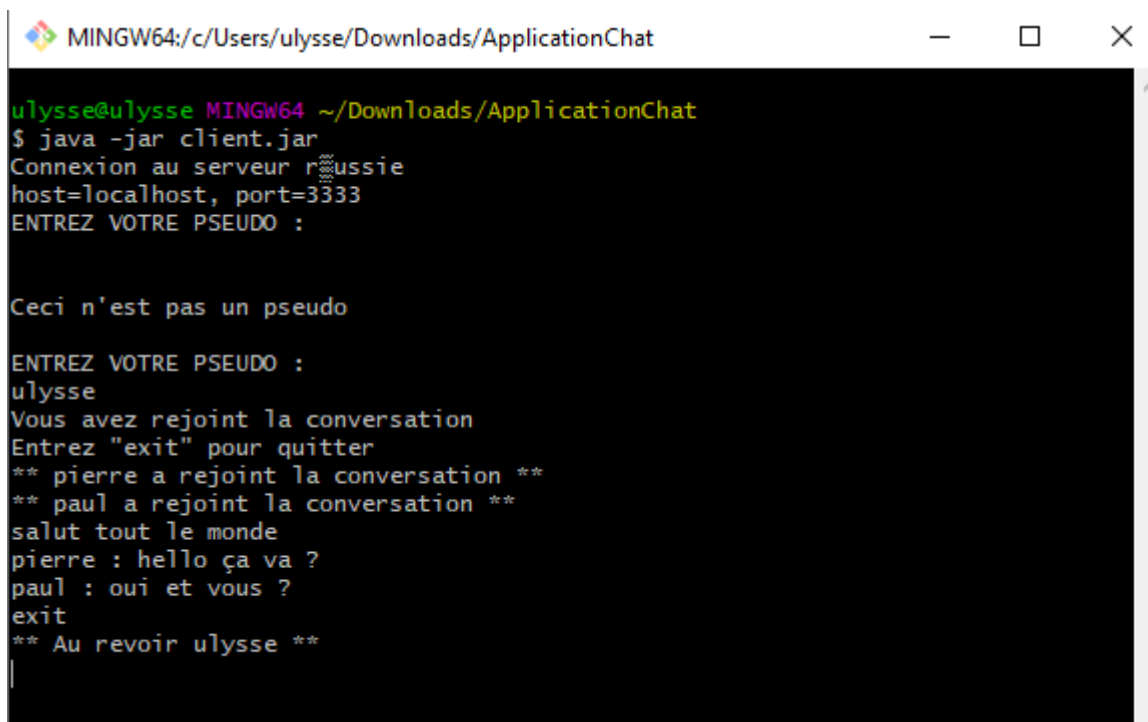


```
MINGW64:/c/Users/ulysses/Downloads/ApplicationChat
ulysses@ulysses MINGW64 ~/Downloads/ApplicationChat
$ java -jar client.jar
Connexion au serveur réussie
host=localhost, port=3333
ENTREZ VOTRE PSEUDO :
paul
Vous avez rejoint la conversation
Entrez "exit" pour quitter
ulysses : salut tout le monde
pierre : hello ça va ?
oui et vous ?
```

On voit bien que les 3 messages s'affichent sur les 3 consoles. Lorsque le message vient d'un utilisateur extérieur son nom est également affiché (ce n'est pas le cas pour l'utilisateur courant afin de pouvoir faire le distinguo entre ses messages et les messages des autres utilisateurs).

Si maintenant l'utilisateur 1 quitte la conversation avec la commande "exit" voici ce qui apparaît sur les consoles.

Utilisateur 1 :

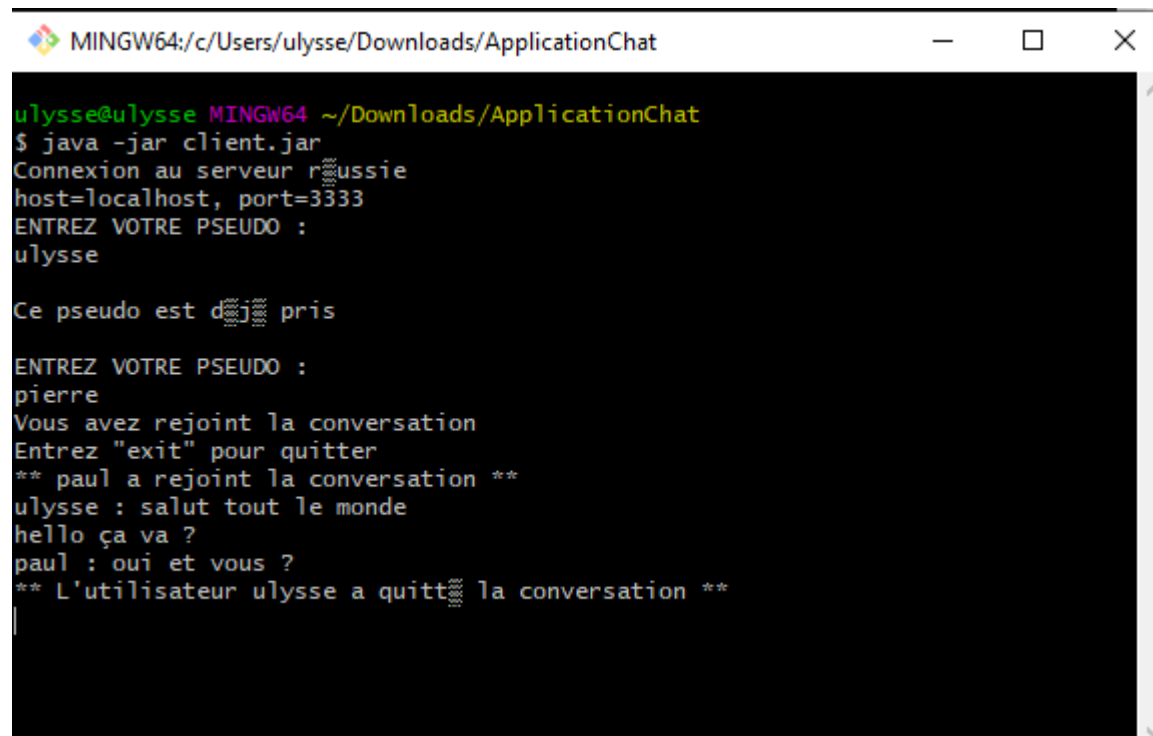


```
MINGW64:/c/Users/ulysses/Downloads/ApplicationChat
ulysses@ulysses MINGW64 ~/Downloads/ApplicationChat
$ java -jar client.jar
Connexion au serveur réussie
host=localhost, port=3333
ENTREZ VOTRE PSEUDO :

Ceci n'est pas un pseudo

ENTREZ VOTRE PSEUDO :
ulysses
Vous avez rejoint la conversation
Entrez "exit" pour quitter
** pierre a rejoint la conversation **
** paul a rejoint la conversation **
salut tout le monde
pierre : hello ça va ?
paul : oui et vous ?
exit
** Au revoir ulysses **
```

Utilisateur 2 :



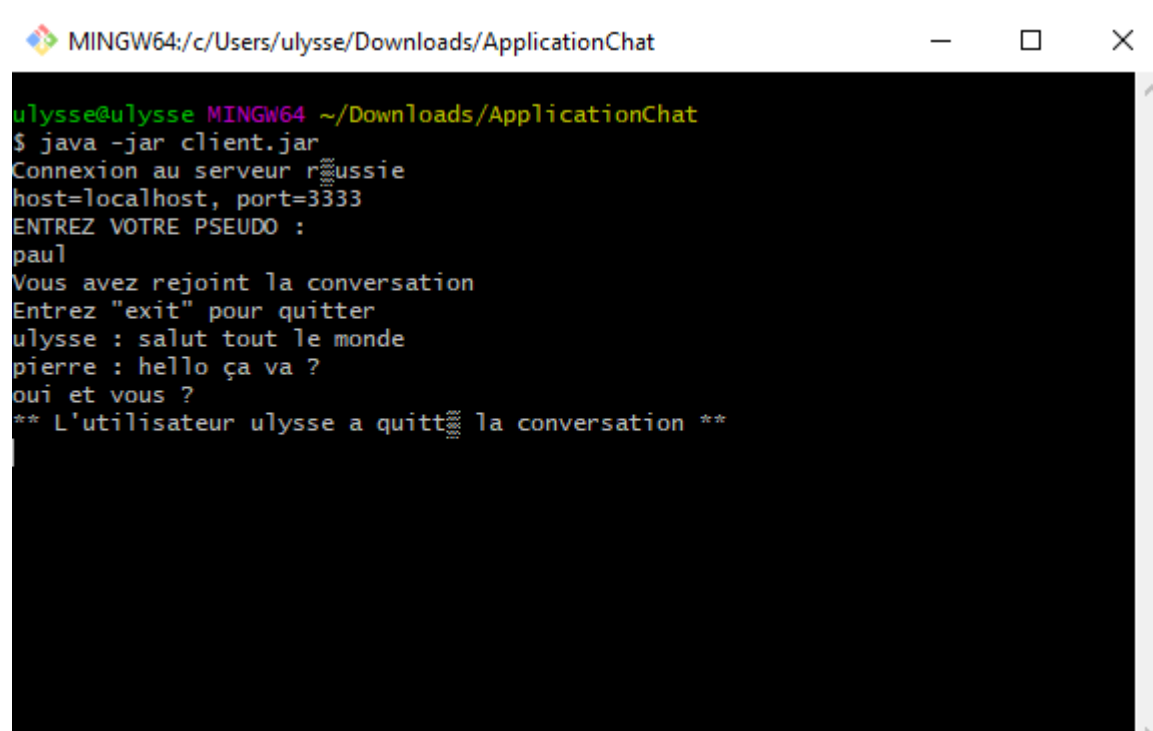
```
MINGW64:/c/Users/ulysses/Downloads/ApplicationChat

ulysses@ulysses MINGW64 ~/Downloads/ApplicationChat
$ java -jar client.jar
Connexion au serveur réussie
host=localhost, port=3333
ENTREZ VOTRE PSEUDO :
ulysses

Ce pseudo est déjà pris

ENTREZ VOTRE PSEUDO :
pierre
Vous avez rejoint la conversation
Entrez "exit" pour quitter
** paul a rejoint la conversation **
ulysses : salut tout le monde
hello ça va ?
paul : oui et vous ?
** L'utilisateur ulysses a quitté la conversation **
```

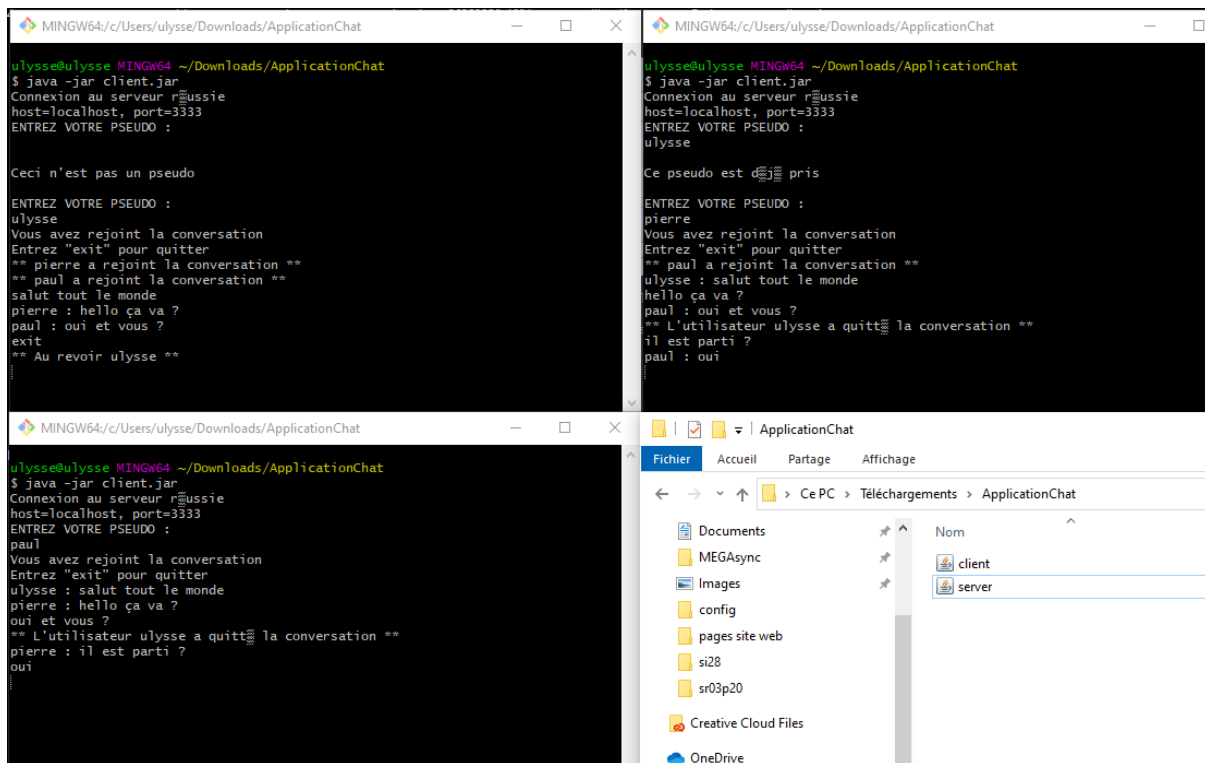
Utilisateur 3 :



```
MINGW64:/c/Users/ulysses/Downloads/ApplicationChat

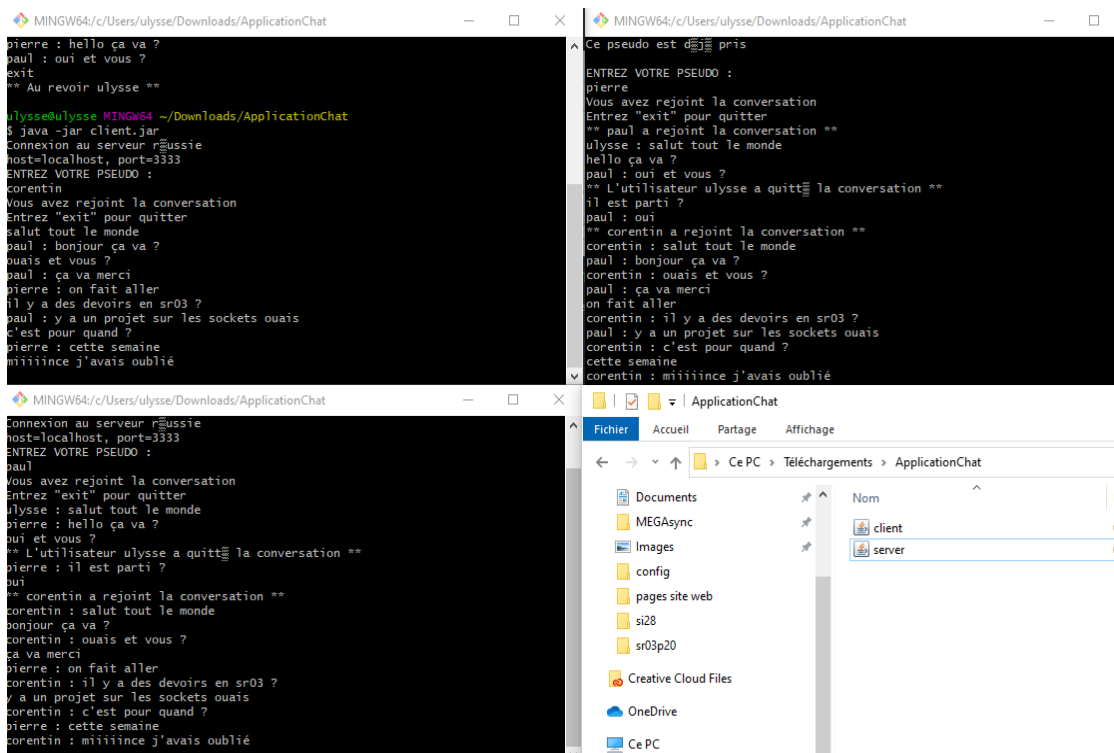
ulysses@ulysses MINGW64 ~/Downloads/ApplicationChat
$ java -jar client.jar
Connexion au serveur réussie
host=localhost, port=3333
ENTREZ VOTRE PSEUDO :
paul
Vous avez rejoint la conversation
Entrez "exit" pour quitter
ulysses : salut tout le monde
pierre : hello ça va ?
oui et vous ?
** L'utilisateur ulysses a quitté la conversation **
```

Les utilisateurs restants peuvent continuer leur discussion qui n'apparaît plus que sur leurs consoles.

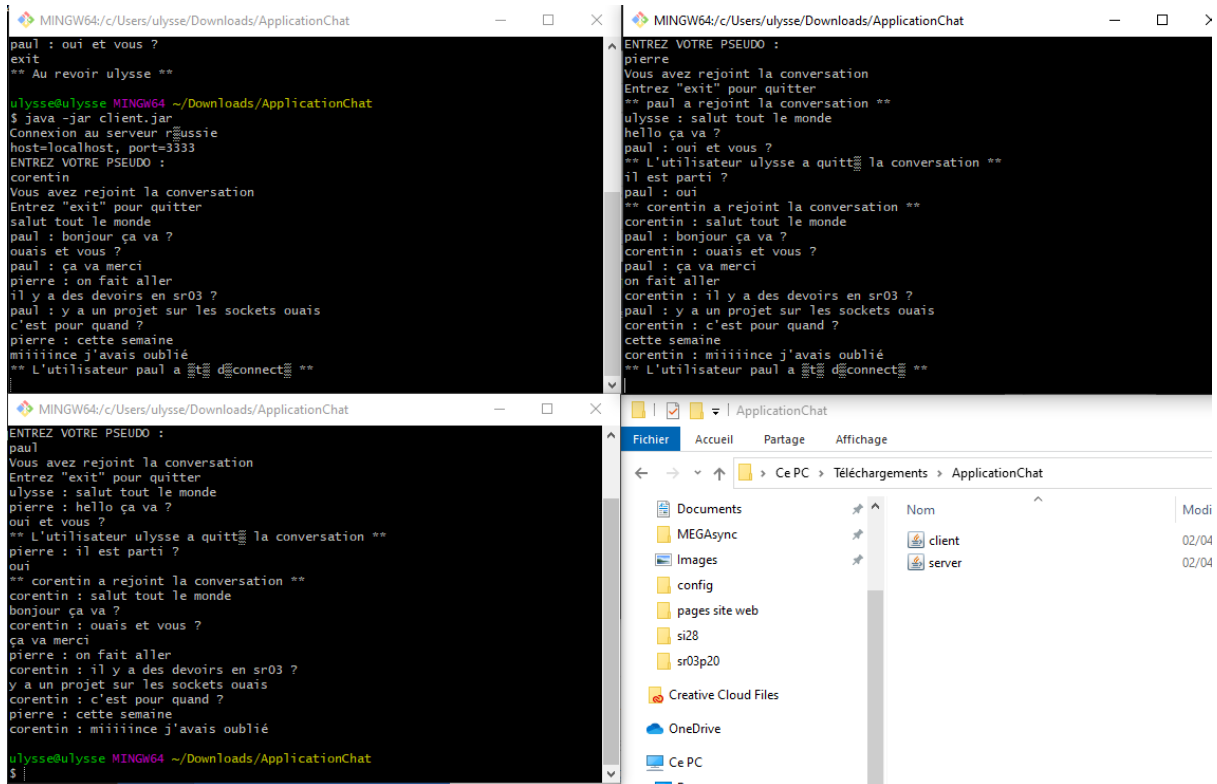


Une place ayant été libéré, un autre utilisateur peut s'y connecter avec un pseudo différent ou alors le pseudo de l'utilisateur qui vient de se déconnecter et qui est de nouveau libre.

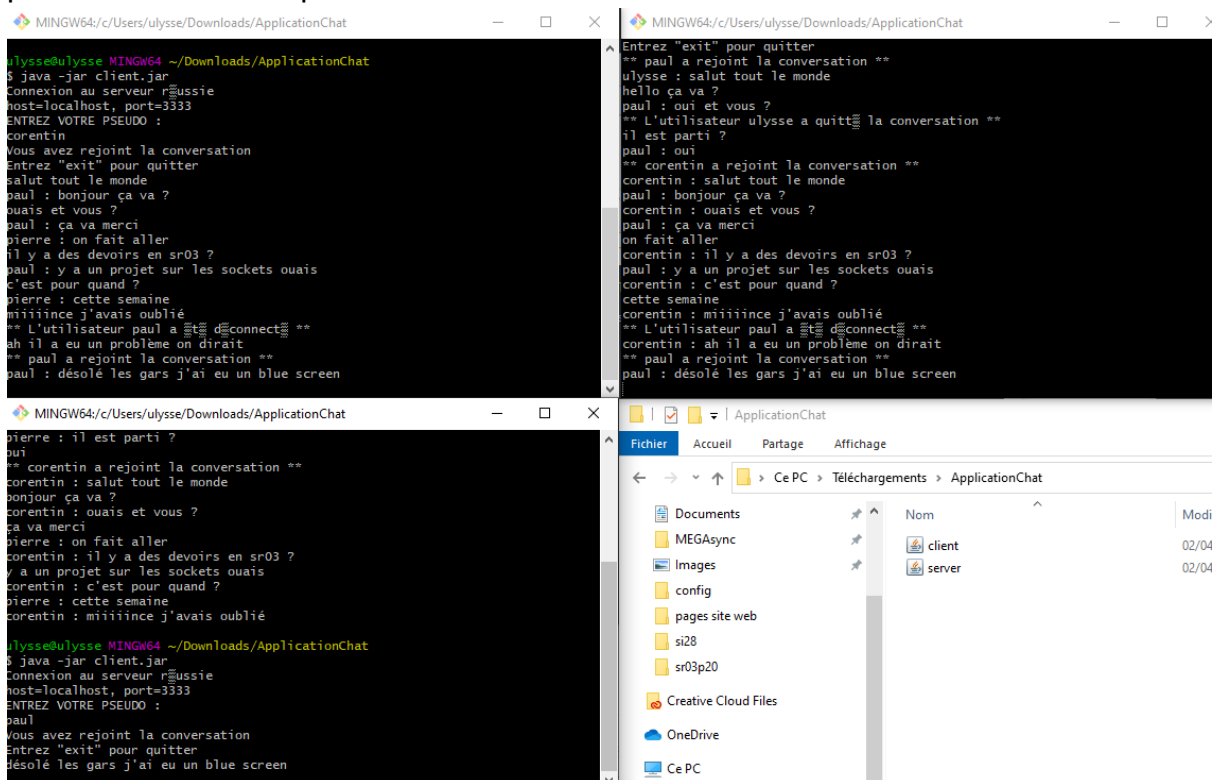
Le nouvel utilisateur n'a bien sûr pas connaissance des messages envoyés avant sa connexion.



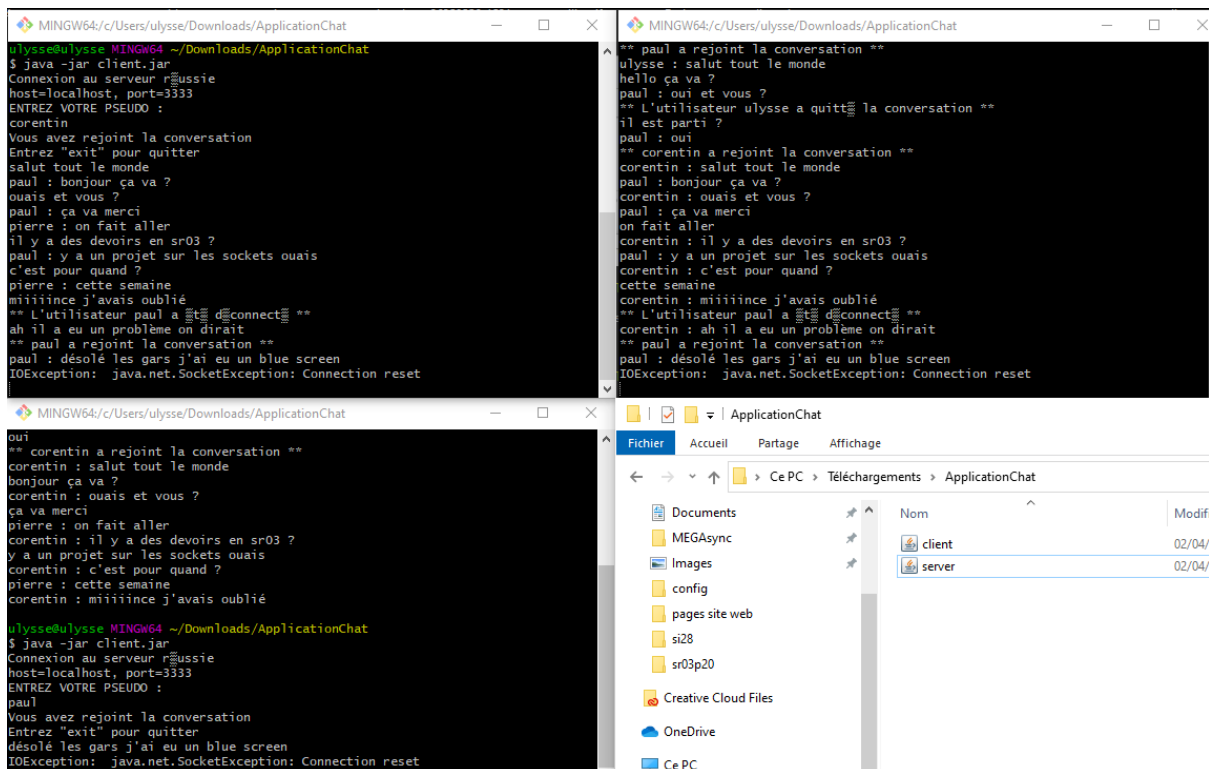
Si un utilisateur est déconnecté de manière imprévue, un message est envoyé aux autres utilisateurs pour les en avertir.



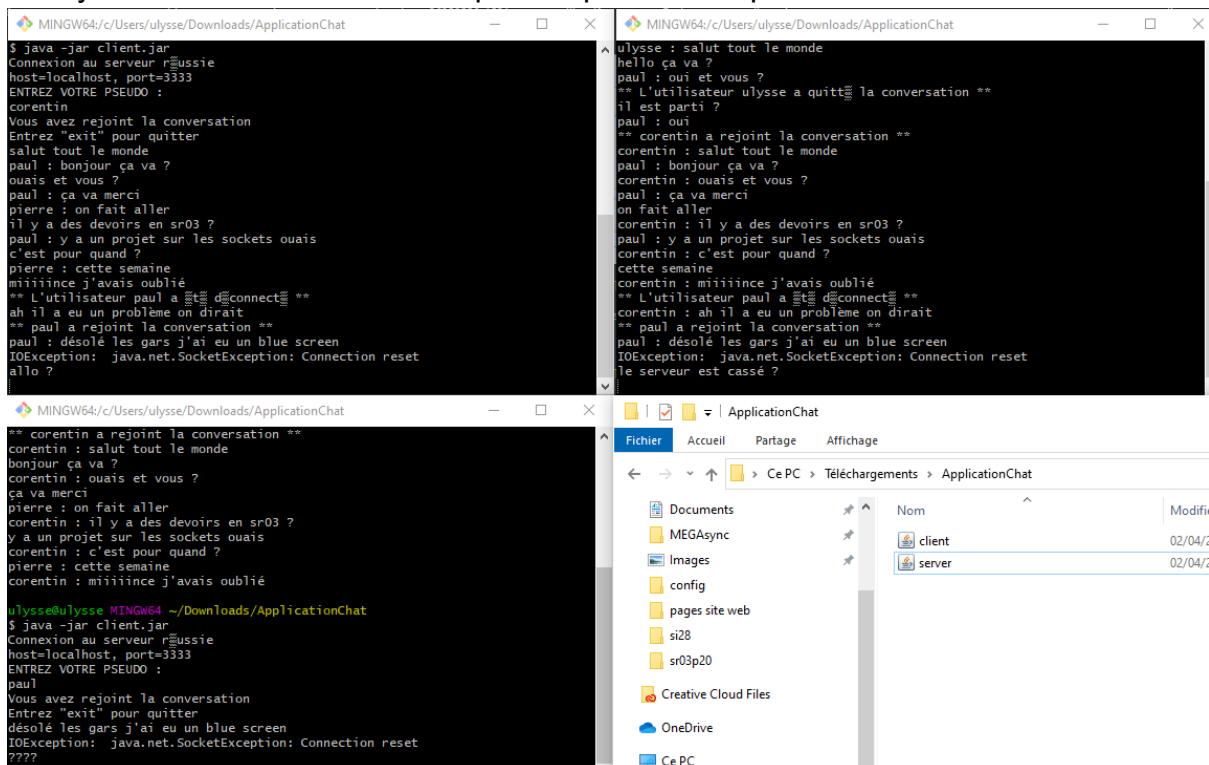
Comme pour une déconnexion classique, son socket et son pseudo sont libérés et peuvent être utilisés par un autre utilisateur.



Si en revanche c'est le serveur qui se déconnecte, une erreur va être affichée sur les consoles des utilisateurs.



Si les utilisateurs tentent de communiquer, leurs messages ne seront bien sûr pas envoyés. Relancer le serveur ne permet pas de récupérer les sessions en cours.



QUESTIONS :

Comment garantir qu'un pseudo est unique ?

Au moment de créer un pseudo on va rentrer dans une boucle qui nécessitera que le pseudo soit unique pour en sortir. Quand l'utilisateur rentre un pseudo, on parcourt le tableau clients pour vérifier les pseudos des autres clients. Si un pseudo est identique à un pseudo déjà pris, un message apparaît et l'utilisateur reste dans la boucle et doit rentrer un autre pseudo. Si on a parcouru tout le tableau sans trouver de doublon, (l'indice *i* est alors égale à la taille du tableau), on sort de la boucle while avec un break.

Comment vous faites pour gérer le cas d'une déconnexion de client sans que le serveur soit prévenu et vis-versa ?

On implémente un thread de surveillance pour chaque client. On le lance en même temps que ClientThread. Si le client s'arrête (client.isAlive() à false) et que cet arrêt n'est pas dû à une demande de l'utilisateur (userHasLeft à false) on prévient les autres utilisateurs à travers la fonction preventUnexpectedInterruption de la classe ClientThread et on supprime ledit client du tableau des clients pour libérer sa place. Si le serveur se déconnecte les clients reçoivent une alerte via les exceptions.