Projet : Application de chat multi-thread en Java

I) Contexte

Le projet est de développer une application console de chat entre plusieurs clients, tous connectés au même serveur. Les différents clients peuvent choisir un pseudonyme, envoyer des messages (que tous les autres clients connectés à ce moment là recevront), et recevoir les messages que les autres clients connectés envoient.

Le serveur, une fois lancé, va stocker les données des différents clients. Le principe d'un serveur, est d'être en permanence en « écoute » des requêtes qui lui sont soumises par des clients. Il traite ensuite les requêtes en fonction de leur nature.

Un client, lors de sa première connexion, choisit un pseudonyme, qui est stocké par le serveur. Il envoie des messages au serveur, qui les renvoie aux autres clients (c'est les requêtes que le client envoie au serveur).

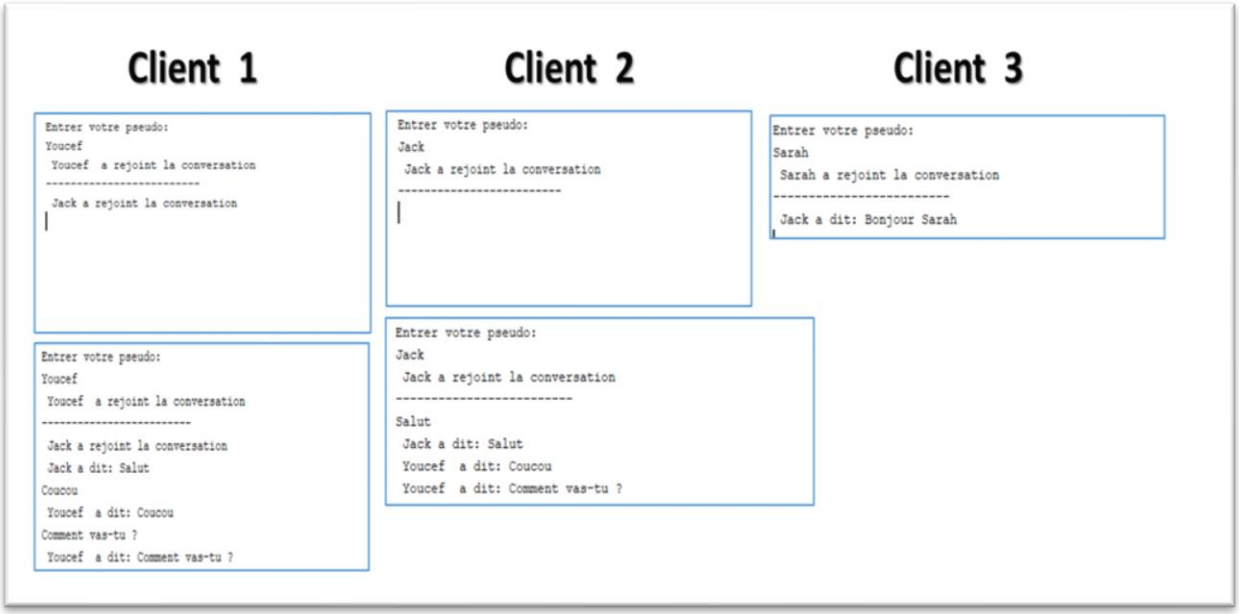
Nous allons utiliser différents concepts comme les Sockets et les Threads.

Un socket est, par définition, une extrémité d'un lien de communication bidirectionnel (nous auront donc recours à des sockets server, et des sockets clients). Un socket est associé à un numéro de port.

Un thread permet à un programme d'effectuer des tâches en « arrière-plan », sans interrompre le programme dans sa globalité.

Plusieurs objectifs à remplir dans ce projet :

- Le serveur accepte bien les clients qui se connecte à son port
- Les clients peuvent choisir un pseudonyme qui est enregistré
- Ce pseudonyme est unique
- Les clients sont notifiés par les connexions / déconnexions des clients, ce qui implique que les connexions / déconnexions sont gérées par le serveur
- Déconnexion d'un client amorcée par l'entrée d'un mot clé « exit » de sa part
- Gérer le cas d'une déconnexion du client sans que le serveur soit prévenu et vice-versa (c'est-à-dire, si le client se déconnecte sans envoyer « exit »)



Exemple d'utilisation théorique :

1) Le serveur est déjà démarré, client 1 se connecte et entre le pseudo Yuheng, puis entre le message « salut ».
2) Client 2 se connecte et entre le nom Louis, il ne verra pas le message « salut » de Yuheng, qui lui est averti de la connexion de Louis. De là s'en suit un dialogue entre les 2 clients.
3) Louis se déconnecte, Yuheng, qui est toujours là, est notifié, et peut continuer à discuter (avec lui-même comme il est seul, mais n'importe qui peut encore se connecter).

II) Conception et développement

Nous avons choisi d'utiliser le format de donnée DataInputStream et DataOutputStream pour les requêtes entrantes et sortantes, car le format est facile à utiliser, notamment pour les chaînes de caractères. Les méthodes readUTF() et writeUTF() permettent d'écrire sur ou de lire un socket.

L'implémentation du système suivant nécessite aussi la mise en place de setters et getters notamment pour avoir un code plus lisible et facile d'accès.

Nous avons développé l'application depuis l'environnement Eclipse Java EE.

a) Serveur

Le serveur sera supporté par un serverSocket, son rôle est d'accepter et d'établir une connexion avec une client sur un port défini (numéro de port élevé de préférence pour éviter des problèmes de sécurité, concurrence), ici 20000. Le but est simple : le serveur doit toujours être dans un état d'écoute à une potentielle nouvelle requête d'un client.

Dès lors qu'un nouveau client est accepté, un thread MessageRecepteur s'en occupe (permettant au serveur de continuer de tourner pendant ce temps-là).

La première étape est le pseudonyme, rentré par le client, et testé par le serveur pour vérifier son unicité grâce à la fonction memeNom, qui parcourt simplement la Hashmap en vérifiant les noms des autres utilisateurs. Une fois un pseudonyme valide entré, le client est ajouté à la liste de clients dans la Hashmap. Les clients connectés sont stockés dans une Hashmap (qui permet de stocker des paires de variables, ici on stockera les informations de la socket client ET le nom du client).

Le thread associé à notre client rentre alors dans une boucle while(true) dans laquelle le thread vérifie si le message est « exit », dans ce cas le client est déconnecté proprement, sinon le message est renvoyé à tous les autres clients. Le client est notifié des départs / arrivés des autres clients. La Hashmap est bien sûr mise à jour dès lors qu'un client s'est déconnecté.

b) Client

Chaque client sera supporté par un socket, différent de serverSocket car doit avoir un nom d'hôte (ici localhost), et un numéro de port sur lequel le client essayer de se connecter (le numéro doit être le même que celui du serverSocket ouvert de l'autre côté.

Deux threads vont être créés : un pour envoyer les messages, et un pour recevoir les messages aux autres clients.

MessageEnvoyer est un thread qui va lire ce que le client écrit et renvoyer le message aux autres clients, sauf si celui-ci est « exit », qui entamerait la procédure de déconnection.

MessageRecepteur est un thread qui affiche au client tous les messages envoyés par les clients connectés au serveur.

Quand un client se déconnecte, il faut bien penser à terminer les 2 threads MessageEnvoyer et MessageRecepteur puis le socket client.

L'utilisation de deux threads distincts permet au client de pouvoir écrire un message tout en recevant des messages venant des autres clients (parallélisation des tâches grâce aux threads)

c) Déconnexion inattendue

La déconnexion du client sans que le serveur soit prévenu par « exit » déclenche une erreur, qui sera gérée dans la partie catch. Ainsi le serveur va retirer le client dans la partie « catch »

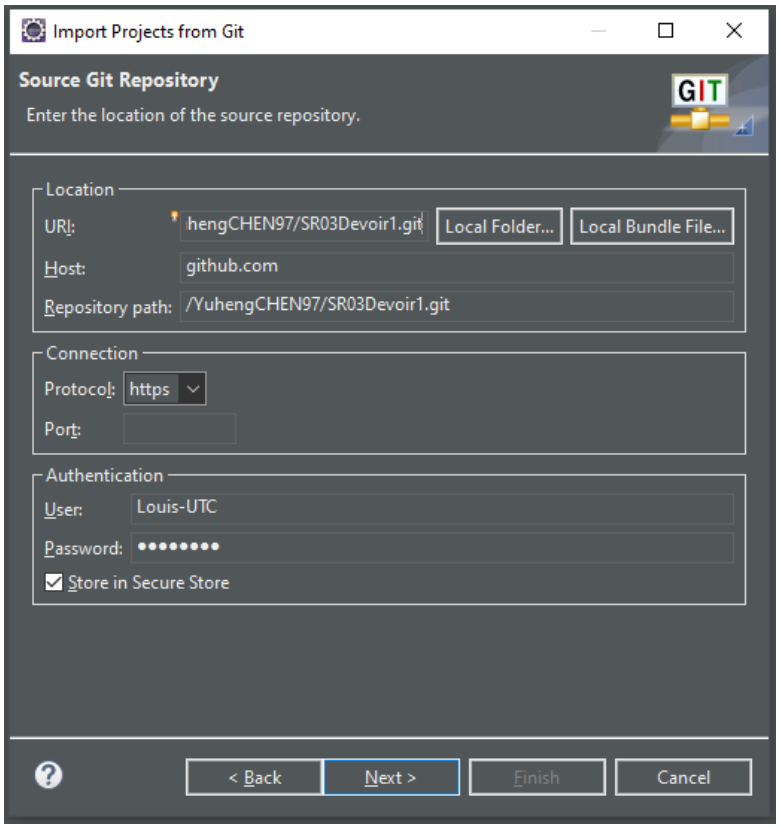
Pour le client, si le serveur se déconnecte, avec une erreur par exemple, le client va arrêter le thread en question (car la variable quitter prendra la valeur true, amenant à un break de la boucle while(true)).

III) Scénarios

a) Récupération, installation et utilisation de l'application

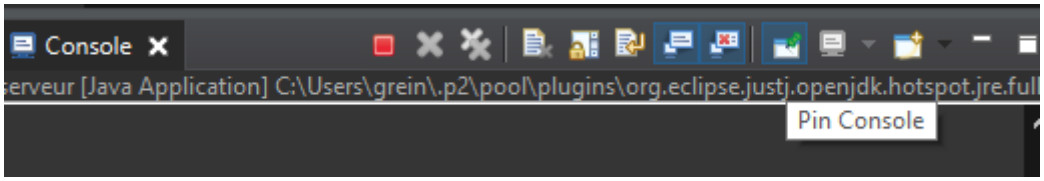
Installation de l'environnement de développement « Eclipse IDE for Enterprise Java and Web Developers » via : https://www.eclipse.org/downloads/packages/installer.

Une fois Eclipse lancé, allez dans File -> Import -> Git -> Projects from Git -> Clone URI



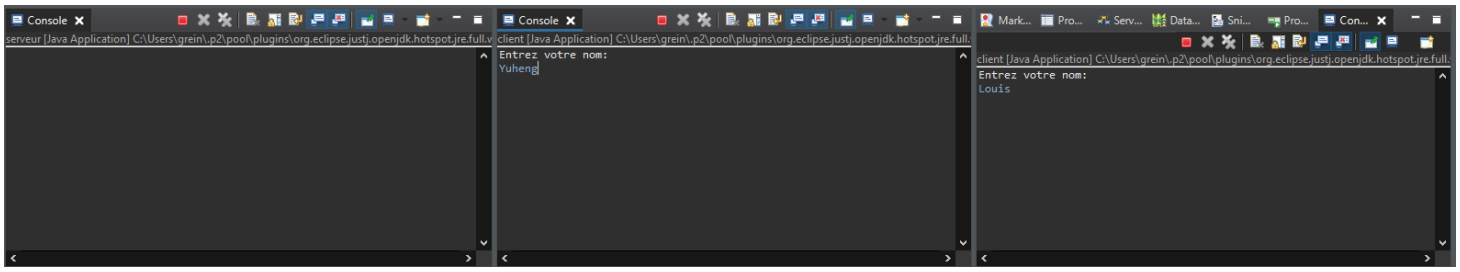
Récupération de l'URI via : https://github.com/YuhengCHEN97/SR03Devoir1 que vous pouvez copier-coller dans URI, puis connectez-vous avec votre compte GitHub (au lieu de Louis-UTC …) puis terminez le clonage du projet.

Ouvrez ensuite le fichier serveur.java, clic-droit dessus, « Run as Java Application », cela va vous ouvrir une console. Cliquez sur « Pin console » (pour que cette console reste sur l'affichage du serveur). La target runtime du serveur est Apache Tomacat v8.0



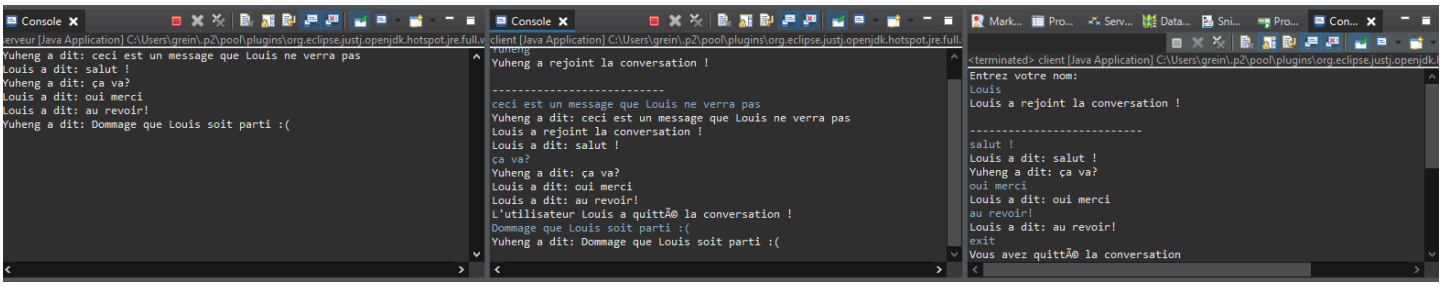
Puis ouvrez deux autres consoles (pour connecter 2 clients) avec le bouton à droite de Pin Console.

Ouvrez le fichier client.java, puis répétez l'opération autant de fois que vous voudrez connecter de clients : cliquez sur « Run as Java Application ». Avec le bouton à droite de Pin Console, vous pouvez choisir d'afficher tel ou tel client sur la console, et vous pouvez verrouiller votre choix en cliquant sur Pin Console.



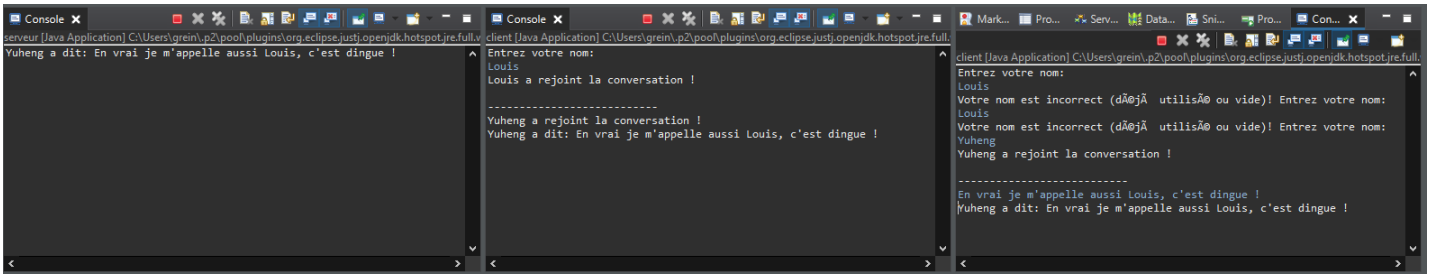
Nous voilà avec 3 consoles (dans l'ordre) : le serveur, client 1 (Yuheng), client 2 (Louis).

b) Scénario idéal



Voici le scénario idéal, le serveur est lancé, Yuheng rejoint le chat, écrit un message (que Louis ne verra pas), Louis se connecte, ils communiquent, Louis s'en va, Yuheng est notifié, mais il peut continuer à discuter, et d'autres clients peuvent venir.

c) Même pseudo



Tant que le pseudo n'est pas différent de tous les autres présents dans la Hashmap, le client ne peut pas entrer dans le chat.