Etienne LEBARILLIER

Mathis DELAUNAY

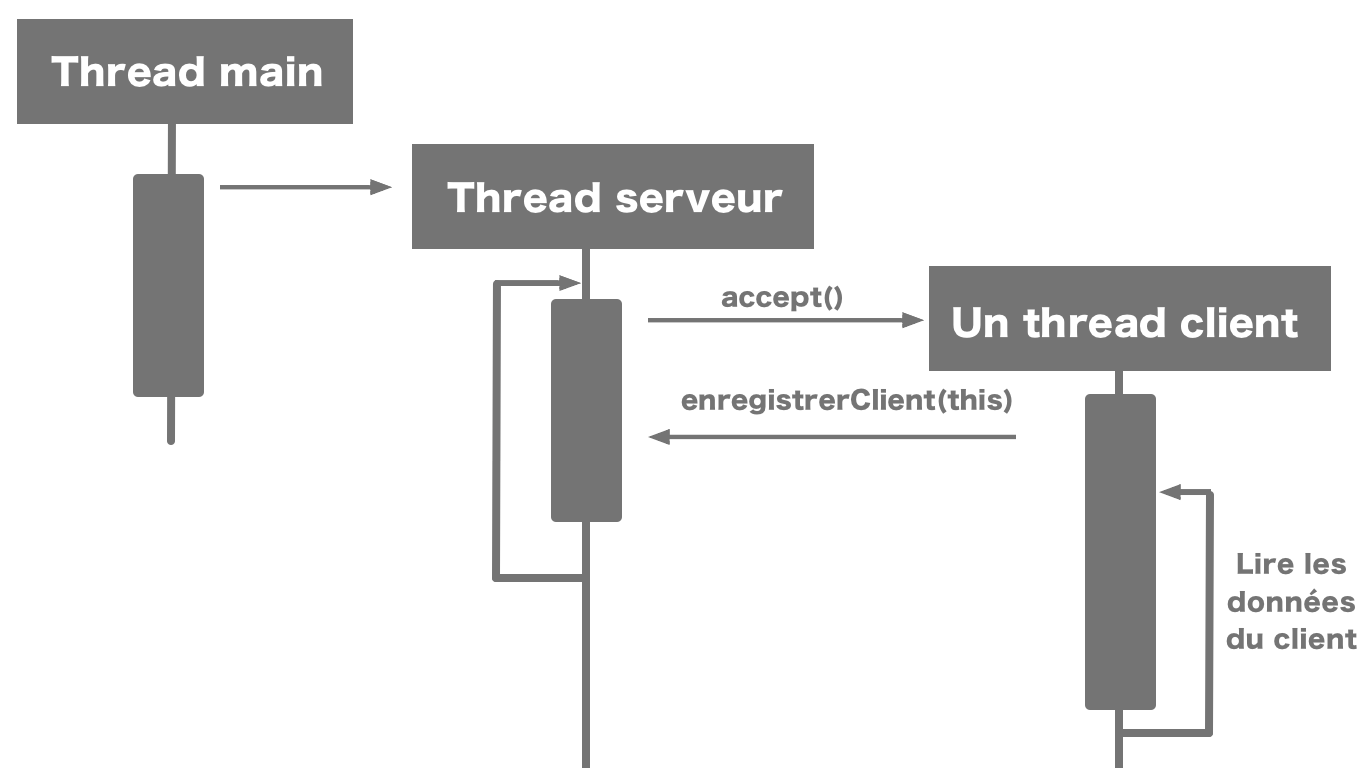
**Projet chat réseau en JAVA**

Le but du projet est de créer un serveur de chat avec le protocole TCP. Ce protocole est dit connecté contrairement à UDP qui ne se préoccupe pas si les paquets ont bien été transmis.

**Les Thread**

Il est impératif pour ce projet d’utiliser les Thread. En effet, le serveur doit pouvoir gérer ses clients, et en même temps pouvoir en accepter de nouveaux. De plus, sans l’utilisation de Thread, soit notre application ne pourrait être connectée qu’à un seul client, soit notre serveur serai bloquant sur la fonction accept et ne pourrait donc communiquer avec tous ses clients.

Les Threads vont donc résoudre notre problème. En effet, ils permettent l’exécution simultanée de plusieurs processus partageants les mêmes variables locales. Ainsi, nous pourrons dédier un thread pour la gestion de l’acceptation de nouveaux client (La classe ServeurChat), et d’autres thread, qui attendront une réponse de chaque client connecté au serveur (La classe Client). Nous avons décidé d’implémenter l’interface Runnable pour nos classes. En effet, pour l’amélioration de notre code ou de futures mises à jour, il est préférable de traiter une interface pour permettre l’héritage de nos classes. Voici donc le fonctionnement global des Thread :



On peut donc voir qu’après la méthode accept du Serveur, celui-ci créé un client threadé qui va lire ce que l’utilisateur écrit. Nous mettons aussi le serveur en Thread car cela libère le main. Le thread serveur va donc boucler sur le accept et autoriser la connexion de nouveaux clients. Chaque utilisateur aura son thread client dédié pour attendre une réponse de sa part.

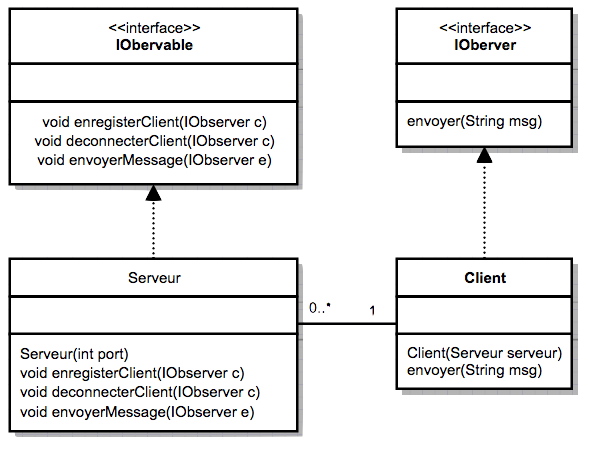
**Les sockets**

Nous allons maintenant parler de la technologie utilisée pour le réseau. Etant donné que nous utilisons le protocole de couche 4 nommé TCP, nous allons utiliser les bibliothèques de JAVA nommées *java.net.Socket* et *java.net.ServerSocket.* Ainsi, notre classe serveur va détenir sa socket sur le port précisé en paramètre de son constructeur grâce à la bibliothèque *ServerSocket* qui permet toute la gestion d’une socket serveur. Ensuite, tous les clients qui auront chacun leur socket qui sera passée en paramètre de leur constructeur, après l’avoir obtenue lors du accept côté serveur. Pour les clients, nous utilisons la bibliothèque *Socket* car nous avons simplement besoin de manipuler des sockets et non pas de gérer l’écoute sur un port particulier etc. Chaque client aura donc accès à son buffer en écriture et en lecture pour communiquer avec l’utilisateur. Pour cela, il aura simplement besoin d’appeler des méthodes de sa socket nommées *getInputStream*() et *getOutputStream*(). Pour plus de facilitées pour le client, il pourra transformer les flux en buffer pour faciliter l’écriture et la lecture. Étant donné que la bibliothèque des flux fonctionne avec le patron Décorateur, le client aura simplement à encapsuler ses flux dans les différents constructeurs nommés *InputStreamReader* puis *BufferedReader* pour la lecture et *OutputStreamWriter* puis *BufferedWriter* pour l’écriture.

**Le patron Observateur**

Pour notre projet nous avons décidé d’opter pour le patron de comportement nommé Observateur. En effet, les différents observateurs seront les clients qui observeront le serveur. Ils devront s’enregistrer auprès du serveur dès qu’ils seront créés pour que celui-ci les connaisse.

Voici à quoi ressemblerai notre Projet suivant le pattern Observateur (Ce schéma ne représente que les méthodes utiles pour le patron, pas le code complet) :



Nous avons donc bien la méthode « enregistrerClient » qui permettra aux clients de l’utiliser ainsi : serveur.enregistrerClient(this) pour s’enregistrer auprès du serveur. La méthode « deconnecterClient » qui permettra aux clients de ne plus être observateur du serveur, et ainsi cesser la communication dans notre cas. Nous avons la méthode « envoyerMessage » qui va pouvoir notifier tous ses observateurs de l’arrivée d’un nouveau message. Ainsi, cette fonction appellera les méthodes « envoyer » de chaque observateur pour leur envoyer le message. Cependant, « envoyerMessage » prend en paramètre un observateur. C’est une exception qui permet de ne pas envoyer de message à celui qui l’a envoyé. En effet, le client aura simplement à appeler « envoyerMessage » ainsi :

serveur.envoyerMessage(this) pour ne pas recevoir le message qu’il vient d’envoyer.

**La synchronisation**

Nous pouvons voir que nous n’avons pas besoin de passer de message dans envoyerMessage. En effet, les messages à envoyer sont stockés dans une autre classe. Cette classe se nomme BoiteAuxLettres et permet de stocker le message à envoyer. La classe détient un attribut de type String privé, qui ne pourra être accédé que par les méthodes get et put. Quand la méthode get sera exécutée, si la variable contenant le texte à envoyer est vide, alors on endormira le processus qui a tenter d’acceder au message. Si la méthode put est exécutée alors que la variable contient quelque chose, le processus courant sera endormi. En effet, l’accès à cette variable est une section critique, car deux Threads ne doivent pas accéder à la variable en même temps et il ne faut pas perdre le message à envoyer. Ainsi, il suffit d’endormir les Threads qui tentent d’accéder à la variable alors qu’ils n’ont pas le droit, puis de les réveiller une fois qu’ils le peuvent quand l’état de la variable a changé. Cela assure donc une certaine synchronisation.

De plus, une synchronisation est aussi faite sur le serveur. En effet, avec la méthode envoyerMessage(IObserver o), c’est une section critique car le message doit être envoyer à tout le monde dans le même ordre. C’est pourquoi nous devons rendre cette méthode synchronized.