Rapport Devoir 1 : Application de chat multi-thread en Java

Tobias SAVARY

Marius BUREAU

11 mars 2024

Année universitaire 2023/2024

Table des matières

Contexte

Le projet consiste à développer une application de chat Client/Serveur en utilisant des sockets qui permet d'organiser une discussion publique avec un ensemble de participants. La discussion dans cette application sera affichée dans la console.

L'application reposera sur le fonctionnement d'un processus serveur, fonctionnant sur la machine côté serveur et d'un nombre X de processus clients, fonctionnant sur les machines utilisateurs.

Pour se connecter au serveur, le client aura besoin du programme "Client" ainsi que des programmes threads associés. Il aura aussi besoin de connaître le port sur lequel l'application (côté serveur) est en train de tourner, ainsi que de son adresse IP. Cela implique qu'aujourd'hui l'application ne peut fonctionner qu'entre des clients se situant dans le même réseau que le serveur.

Concernant l'implémentation, il a fallu trouver un moyen de gérer les différents clients en simultané. Pour cela, deux options étaient possibles :

— La création de processus fils

— La création de threads

La solution qui a été retenue a été celle des threads, car cela permet de ne pas être bloqué dans le traitement d'un message en cours. Aussi, afin de ne pas être en conflit lors de la réception et l'envoi de message pour un client, il a été décidé que cela se ferait indépendamment l'un de l'autre (un thread pour la réception de messages et un thread pour l'envoi de messages).

Lors de l'analyse du fonctionnement du serveur, nous avons remarqué que le serveur effectue des actions uniquement lors du traitement de message envoyé par un client. Il n'est donc pas nécessaire d'utiliser deux threads par client comme pour le programme client. Ici, nous utiliserons uniquement un thread par client, qui va attendre la lecture de données sur son socket et effectuera des actions (diffusion de message par exemple) en lien avec ce qui a été lues. Le serveur effectuera une boucle infinie afin d'accepter les nouveaux clients et d'effectuer la gestion et la diffusion de messages reçus.

Exemples de cas d'utilisation pratiques :

— Un client se connecte au serveur et envoi un message.
— Plusieurs clients se connectent au serveur et discutent ensemble.
— Un client se déconnecte du serveur en envoyant le message "exit".
— Un client se déconnecte du serveur sans prévenir (par exemple en fermant la fenêtre du terminal).
— Le serveur s'arrête et tous les clients connectés sont déconnectés.
— Un client essaye de se connecter avec un pseudonyme déjà utilisé par un autre client.

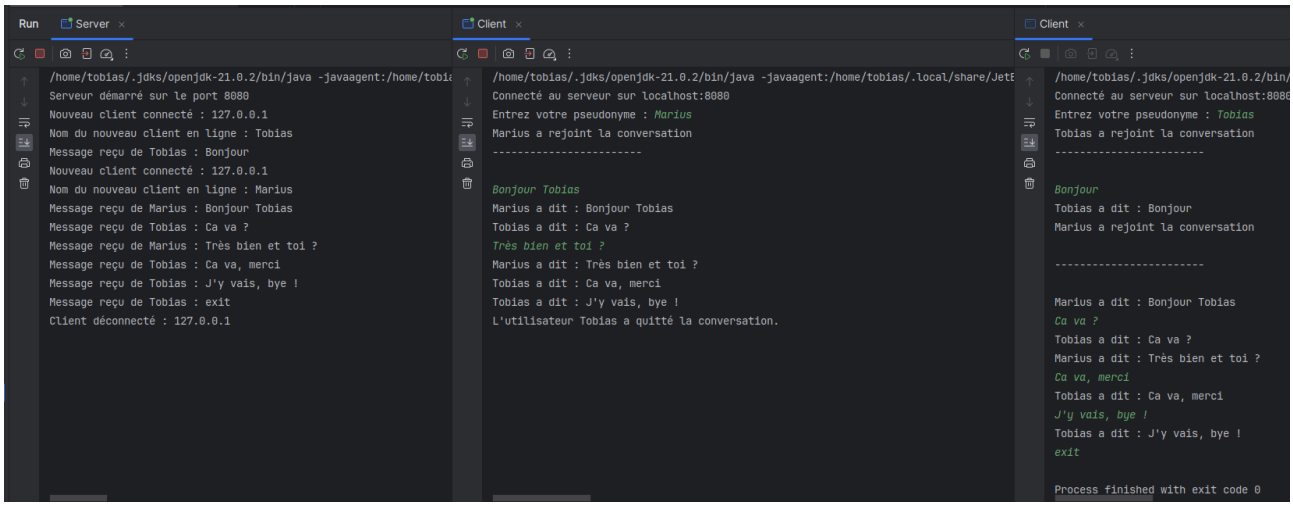

Figure 1 – Exemple d'utilisation Normale

Figure 2 – Deconnexion Client

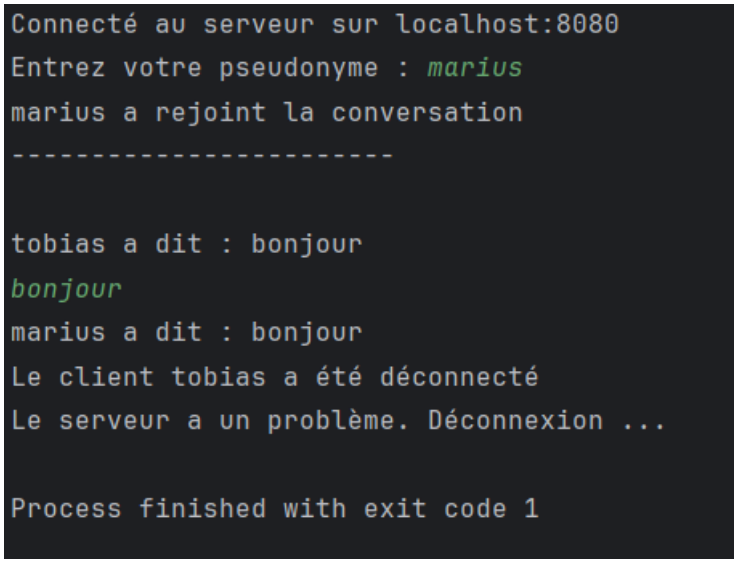

Figure 3 – Deconnexion Serveur

Scénarios

Récupération du code source du projet.
Cloner le repository GitLab suivant : https://gitlab.utc.fr/savaryto/UTC_SR03.git

Pour installer et lancer l'application, il faut :

Utilisation en local :

1. Ouvrir le projet dans IntelliJ IDEA.
2. Lancer une instance de la classe Server.
3. Lancer une ou plusieurs instances de la classe Client.

Utilisation en réseau :

1. Ouvrir le projet dans IntelliJ IDEA.
2. Modifier l'adresse IP et le port dans la classe Client pour qu'ils correspondent à ceux du serveur.
3. Lancer une ou plusieurs instances de la classe Server.

Diagrammes des cas d'utilisation

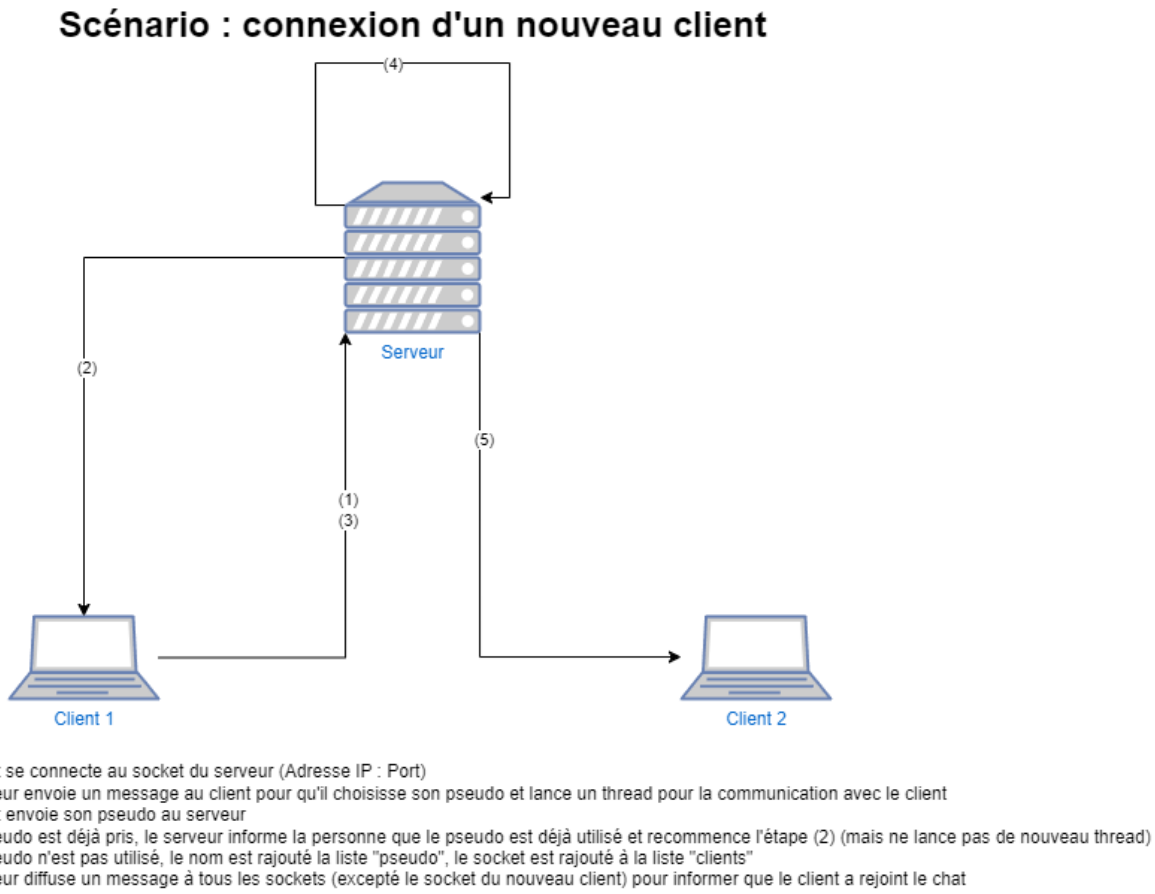

Figure 4 – Diagramme de connexion d'un nouveau client

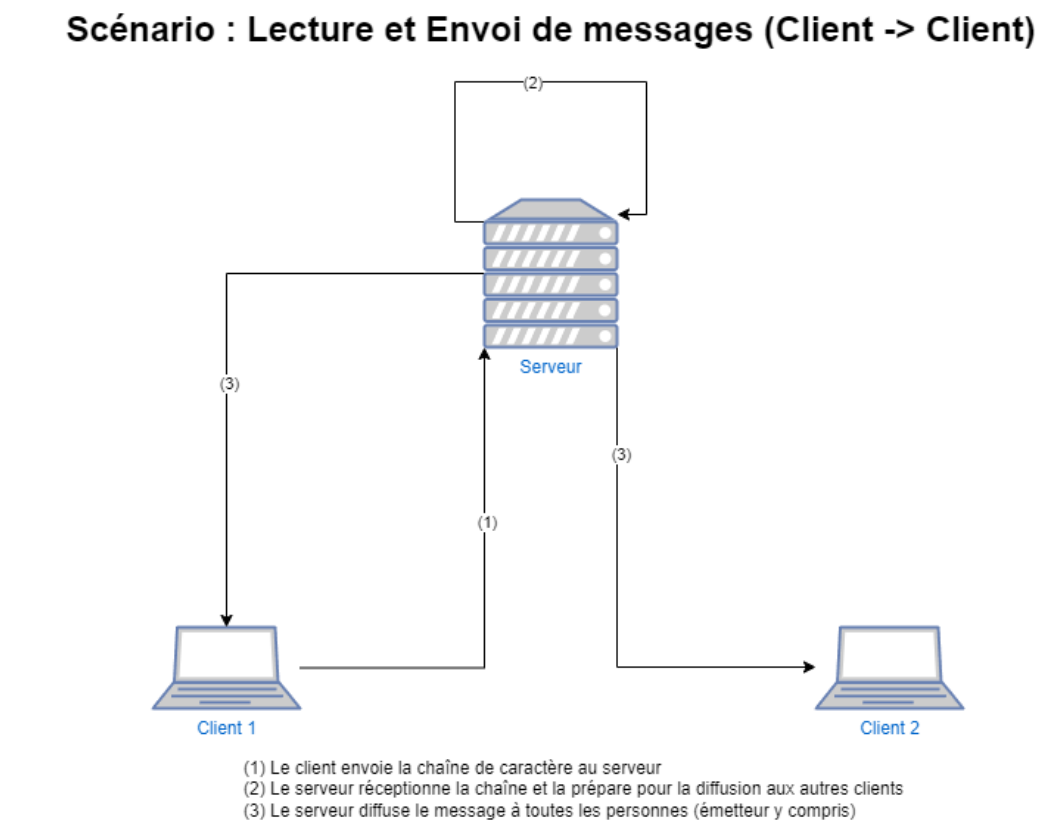

Figure 5 – Diagramme de lecture et d'envoie de message

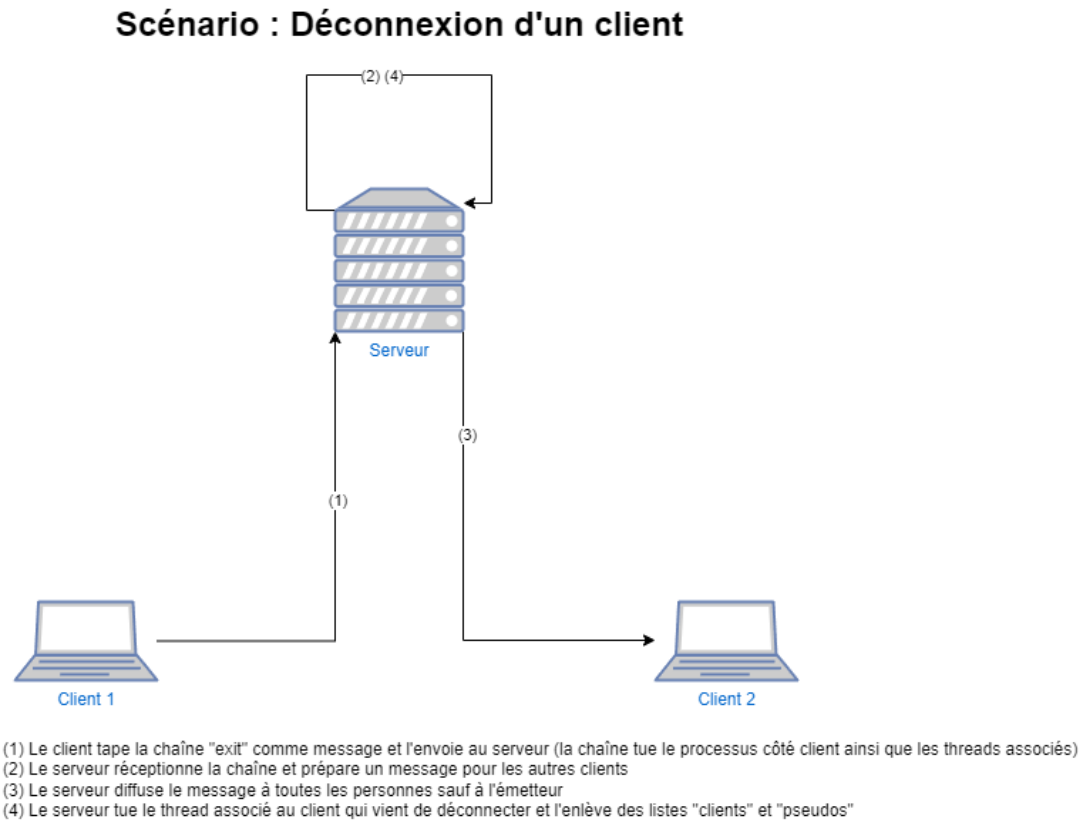

Figure 6 – Diagramme de déconnexion d'un client

Conception et développement

Du côté serveur, l'application exécute une boucle infinie pour continuer à accepter les demandes entrantes et stocke les objets socket nouvellement connectés dans un tableau "clients". Un thread est lancé pour intercepter tout message envoyé dans chaque objet socket récemment stocké. Lorsque le serveur reçoit un message dans l'un des sockets de communication, il récupère ce message puis il le diffuse sur l'ensemble des objets socket stockés dans le tableau "clients". Dans le cas où le serveur reçoit un message "exit" dans l'un des objets socket correspondant à l'un des clients, il diffuse le message suivant "l'utilisateur X a quitté la conversation" sur le reste des clients connectés et libère le socket, les flux d'E/S et le thread associé au client.

Du côté client, l'application effectue une connexion auprès du serveur, envoie son pseudonyme à travers le socket de communication créé, et implémente deux threads. Le premier est utilisé pour intercepter les messages venant du serveur et le deuxième est utilisé pour récupérer les messages saisis par l'utilisateur et de les transmettre au serveur.

Gestion des pseudos uniques

Pour garantir qu'un pseudonyme est unique, nous avons utilisé une liste statique "pseudo" dans la classe Server. Lorsqu'un nouveau client se connecte, le serveur vérifie si le pseudo est déjà présent dans la liste. Si c'est le cas, le serveur envoie un message au client pour lui demander de choisir un autre pseudo. Sinon, le serveur ajoute le pseudonyme à la liste et envoie un message de bienvenue au client.

Voici le code correspondant dans la classe Server :

```
private static final List<String> pseudo = new ArrayList<>();

public static boolean addPseudo(String userName, Socket socket) {
    if (pseudo.contains(userName)) {
        return false;
    } else {
        clients.add(socket);
        pseudo.add(userName);
        return true;
    }
}
```

Dans la méthode run() de la classe MessageReceptor, nous avons ajouté le code suivant pour gérer l'ajout d'un nouveau client :

```
// If the first message, send the username to all clients
if (!isAdded) {
    setUserName(new String(buffer, 0, length));
    isAdded = Server.addPseudo(getUserName(), clientSocket);
    if (isAdded) {
        System.out.println("Nom du nouveau client en ligne : " + getUserName());
        Server.broadcastMessage((getUserName() + " a rejoint la conversation\n"));
        Server.broadcastMessage("------------------------\n");
    } else {
        clientSocket.getOutputStream().write(
            "Le pseudo est déjà utilisé. Veuillez choisir un autre pseudo :".getBytes());
    }
    continue;
}
```

Gestion des deconnexions inattendues

Client

Pour gérer le cas d'une déconnexion de client sans que le serveur soit prévenu, nous avons utilisé une variable statique "isRunning" dans la classe MessageReceptor. Lorsqu'un client se déconnecte, le serveur arrête le thread correspondant en mettant la variable "isRunning" à false. De plus, nous avons ajouté un gestionnaire d'exception SocketException dans la méthode run() de la classe MessageReceptor pour détecter les déconnexions inattendues et arrêter le thread correspondant.

Voici le code correspondant dans la classe MessageReceptor :

```
private boolean isRunning = true;

public void setRunning(boolean running) {
    isRunning = running;
}

public void stopThread() {
    setRunning(false);
    try {
        this.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

@Override
public void run() {
    while (isRunning()) {
        // code pour lire les messages entrants
    }
}
```

Dans la méthode run() de la classe MessageReceptor, nous avons ajouté le code suivant pour gérer la déconnexion d'un client :

```
} catch (SocketException s) {
    System.out.println("Connexion rénitialisé par le client");
    try {
        Server.broadcastMessage("Le client " + getUserName() + " a été déconnecté");
        Server.removeClient(clientSocket, getUserName());
        clientSocket.close();
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}
```

Serveur

Pour gérer le cas d'une déconnexion du serveur sans que les clients soient prévenus, nous avons utilisé une variable statique "isRunning" dans la classe Server. Lorsque le serveur s'arrête, il met la variable "isRunning" à false et arrête tous les threads correspondants aux clients connectés. De plus, nous avons ajouté un gestionnaire d'exception SocketException dans la méthode run() de la classe IncomingMessageHandler pour détecter les déconnexions inattendues du serveur et arrêter le thread correspondant.

Voici le code correspondant dans la classe IncomingMessageHandler :

```
private boolean isRunning = true;

public void setRunning(boolean running) {
    isRunning = running;
}

public void stopThread() {
    setRunning(false);
    try {
        socket.shutdownInput();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

@Override
public void run() {
    while (isRunning() && !socket.isClosed()) {
        // code pour lire les messages entrants
    }
}
```

Dans la méthode run() de la classe IncomingMessageHandler, nous avons ajouté le code suivant pour gérer la déconnexion du serveur :

```
} catch (SocketException s) {
    System.out.println("Le serveur a un problème. Déconnexion ...");
    try {
        getSocket().close();
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
    System.exit(1);
}
```

Conclusion

Dans ce projet, nous avons implémenté une application de chat multi-thread en Java qui permet d'organiser une discussion publique entre un ensemble de participants. Nous avons utilisé des sockets pour la communication entre le serveur et les clients, et des threads pour gérer les messages entrants et sortants. Nous avons également implémenté des fonctionnalités pour garantir l'unicité des pseudos et gérer les déconnexions inattendues.