CEG3585 - Laboratoire 2

Introduction :

L'architecture client-serveur a deux composants principaux. Le premier est un client qui va envoyer des requêtes pour certaines destinations. Le deuxième est le serveur qui va recevoir les requêtes des clients et répondrais à leurs requêtes en renvoyant les informations demandées.

Le client va d'abord essayer de se connecter au serveur par un port réseau. Le serveur qui attend la connexion d'un client sur ses ports réseau va ensuite ouvrir un socket afin de communiquer avec le client. Suite à la connexion, le serveur va passer les données demandées au client. Pour que ces interactions se produisent sans problème, les clients et les serveurs doivent suivent le même protocole pour leur couche transport.

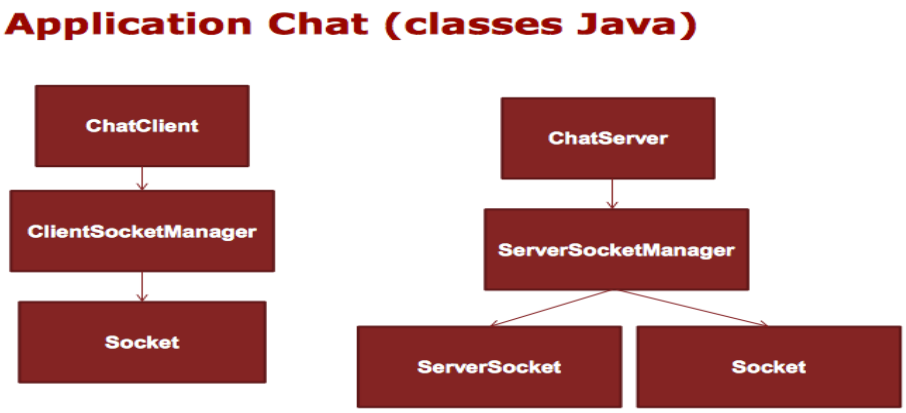
L'architecture client-serveur présente plusieurs avantages à la communication des données. Premièrement, puisque les ressources sont localisées au niveau du serveur, l'implémentation est plus simple que d'autres architecture de systèmes distribués. Aussi puisque l'architecture est centralisée au serveur la mise à jour des données est plus simple puisqu'il faut juste modifier les données à un endroit.

Les désavantages de cette architecture se présente quand plusieurs client essayent de se connecter en même temps, cela pourrait causer des bris dans le serveur qui ne peut pas supporter un trop gros nombre de requêtes. De plus, si le serveur devient indisponible alors les clients ne peuvent plus recevoir des réponses à leurs requêtes. Finalement les systèmes qui implémentent les architectures client-serveur peuvent être très coûteux dans leur installation et leur entretien.

Architecture de l'application chat :

Dans ce laboratoire, une application de chat qui utilise l'architecture client-serveur est introduite. L'application est constituée de 4 classes, une classe pour gérer la partie client (chatClient), une pour gérer les sockets du client (ClientSocketManager), une pour le serveur (ChatServer) et une pour gérer les sockets du serveur

(ServerSocketManager). Dans le contexte du laboratoire, votre tâche consiste à compléter les codes « ClientSocketManager.java » et « ServerSocketManager.java ». Les autres classes sont déjà complétées.



Configuration :

L'application chat est séparée entre serveur et client. À l'aide de ChatClient, le client doit se connecter au serveur à l'aide de l'adresse IP du serveur, le port utilisé par le serveur et un nom d'utilisateur. Ceci est fait à l'aide d'un GUI. Par défaut, le port est 4444, mais le client peut spécifié le port. L'adresse IP et le port sert à configurer le ClientSocketManager, une classe qui gère la connexion au serveur à l'aide d'un Socket. Le ClientSocketManager crée un Socket à l'aide des paramètres qui lui ont été fournis. L'adresse IP locale et le port local sont configurés automatiquement. Après la connexion, il est possible de lire et d'écrire au serveur.

Le serveur écoute sur le port 4444 par défaut. Le serveur peut accepter la connexion de jusqu'à **dix clients** selon son implémentation. Les connexions sont gérées par ServerSocketManager.

Le ServerSocketManager gère un ServerSocket. Ceci permet d'accepter des clients tant que la limite n'est pas dépassée et permet la communication entre le serveur et ses clients. Le serveur peut donc vérifier si les clients se sont bien connectés et peut débuter la réception des messages. Les messages reçus par le serveur sont envoyés aux clients et les clients affichent les messages aux utilisateurs.

Organigramme du code

Côté client

- Connexion au serveur
- Toutes les 10ms, le client vérifie l'état du serveur
- Vérifie le type de la trame reçu (SEL, POL)
    - Si SEL
        - Ajoute le message à la zone de texte du client
    - Si POL
        - Vérifie si le buffer est vide
        - Si non vide, envoie un message de type "ACK [nom] says: [message]" au serveur.
        - Sinon, envoie "NAC"

Côté serveur

- Vérifie si un nouveau client s'est connecté
- Si la limite de client est atteinte, aucun client ne peut être accepté.
- Autrement, écouter le socket pour accepter un client.
    - Si un client est accepté,
        - Assigner un id et créer un objet d'entrée et un objet de sortie.
        - Mettre à jour le nombre de clients.
        - Écrire "[client] joined" dans le tableau de messages à l'index associé au client.
    - Autrement, il y a timeout.
- Pour tous les clients connectés, envoie "POL"
- Lecture des sockets de chaque client
    - Si le message commence par ACK, le message est ajouté dans un tableau en attendant d'être distribué.
    - Si le message commence par NAC, rien n'est fait avec le message.
    - Autrement, le message inconnu est imprimé à la console du serveur avec l'id du client.
- Envoie tous les messages reçus à tous les clients connectés avec le préfixe "SEL"
    - Le serveur réinitialise le tableau.
- Le serveur vérifie s'il y a des connexions qui ont été fermées.
    - Pour chaque connexions fermées associées à un socket nonnul, affecter la valeur nulle au socket, aux objets d'entrée et de sortie associées.

Méthodes des classes java

Class ClientSocketManager

Méthode connect(string, int)

Dans cette méthode, il faut initialiser le reader s_in et le writer s_out. Pour faire ceci, il faut créer un nouveau Socket mySocket qui a été initialisé avec les valeurs de IP et de port de la destination. Les variables du IP et du port de la source doivent également être initialisés avec des méthodes get sur le Socket mySocket.

Méthode close()

La méthode close() sert à fermer la collection du Socket. Pour faire ceci, il faut initialiser les variables du IP de destination à null, et des ports de la source et destination à -1.

Méthode read()

Cette méthode crée un String stream qui contient le message reçu de la connexion. Ceci est fait en lisant s_in, ce qui a été reçu par le BufferedReader. S'il y a une erreur, il faut assumer que la connexion est fermée et donc appeler close() pour s'assurer que la méthode est fermée et finalement stream est initialiser à la valeur null. Cette méthode retourne le stream.

Méthode write()

Cette méthode écrit un String stream a la connexion. Ceci est fait en faisant un println au PrintWriter s_out.

Class ServerSocketManager

Constructeur ServerSocketManager

Ce constructeur prend comme input un portNumber qui est utiliser lors de la création du ServerSocket serverSocket. Ce ServerSocket doit être initialiser avec une valeur d'attente de 1000ms grâce à la méthode setSoTimeout().

Méthode listenOnSocket()

Cette méthode crée un nouveau socket newClient, printwriter out et bufferedreader in. Si MAXCLIENTS est atteint, donc un nouveau client ne peut pas être accepté. Sinon, le serverSocket lance la méthode accept(). Dans le cas où une nouvelle connexion a eu lieu, ceci devient l'initialisation du newClient. Ce newClient reçoit un temps d'attente de 0ms. Les valeurs out et in sont initialisées. Ce nouveau client est finalement ajouté au groupe. Cette méthode retourne le clientid, ce qui l'identification du nouveau client.

Méthode readClient()

Pour cette méthode, vous avez un String stream qui prendra comme valeur ce qui est écrit dans le socket identifier par le clientid si le client est connecté. S'il y a une erreur, il faut assumer que la connexion est fermée et donc clients[clientid], s_out[clientid],

s_in[clientid] prend la valeur null. Le compteur clientCount est ensuite décrémenter. Le stream est retourné.

Méthode writeClient()

Cette méthode écrit le string au socket. Si le clientid est valide, on écrit en utilisant un println le stream dans son s_out.