

SR03 Devoir 1 : Application de chat multi-thread en Java

Contexte

Présentation de sujet

Dans le cadre de la formation de l'UV SR03, nous avons appris le socket. Il s'agit d'une technologie de transfert d'informations entre des programmes qui exécutent sur un réseau. Ce devoir vise à réaliser une application de chat en Java permettant de créer une salle de discussion dans laquelle plusieurs utilisateurs peuvent communiquer ensemble à l'aide de sockets.

Concepts

Après avoir recherché des technologies à implémenter, nous avons décidé de baser sur les quatre concepts suivants afin de réaliser notre application.

Application client/serveur

L'application est développée afin de créer un environnement client/serveur qui est un mode de communication à travers un réseau entre plusieurs programmes : l'un, qualifié de client, envoie des requêtes ; l'autre ou les autres, qualifiés de serveurs, attendent les requêtes des clients et y répondent.

Communication par Socket

D'après [Oracle](#) : «Un socket est un point d'extrémité d'une liaison de communication bidirectionnelle entre deux programmes exécutés sur le réseau. Un socket est lié à un numéro de port afin que la couche TCP puisse identifier l'application à laquelle les données sont destinées à être envoyées.»

Cette définition explique également l'utilisation de socket dans ce devoir. Nous avons deux types de socket : le socket de connexion et le socket de communication.

- Socket de connexion : utilisé par le serveur et sert à établir une adresse que les clients peuvent utiliser pour trouver et se connecter au serveur
- Socket de communication : utilisé par le serveur et le client sert à se dialoguer

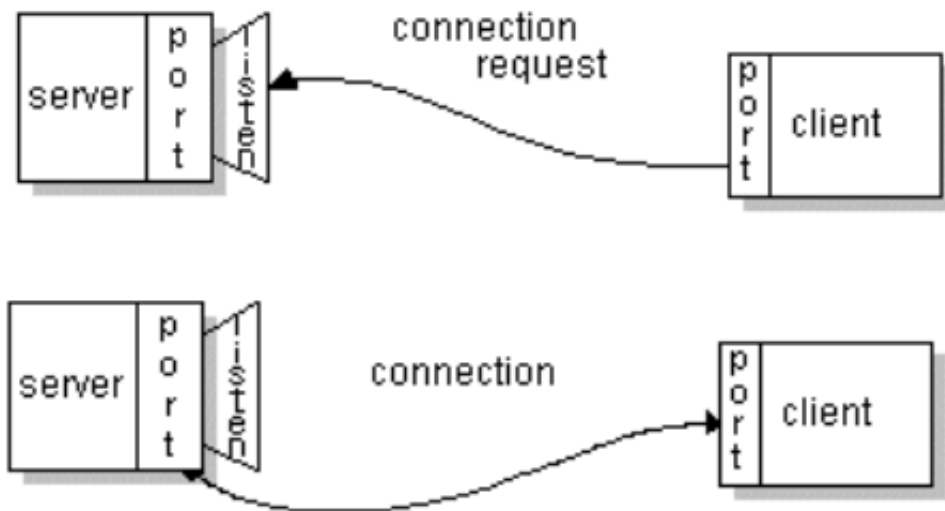


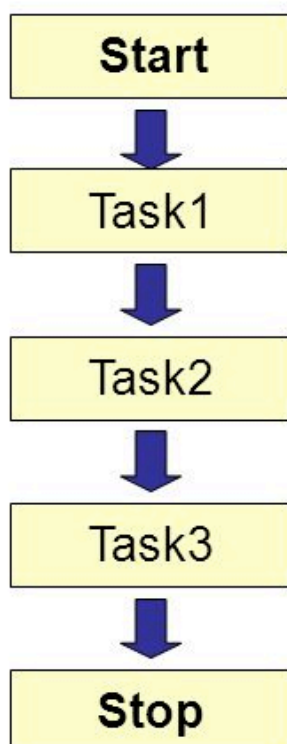
Figure 1 : Client demande une connexion au serveur et le socket de communication établie sur chaque programme

Thread

Un thread désigne un « fil d'exécution » dans le programme. Il s'agit d'une suite linéaire et continue d'instructions qui sont exécutées séquentielles les unes après les autres. En effet, le langage Java est multi-thread, il permet d'utiliser facilement des threads.

Dans ce devoir, l'utilisation des threads permet à l'application d'être capable de traiter plusieurs tâches simultanément, par exemple : le programme de client puisse envoyer un message en recevant des autres messages, ou le serveur intercepte nombreux messages en provenance de différents clients en même temps.

A typical program



Multi-Thread

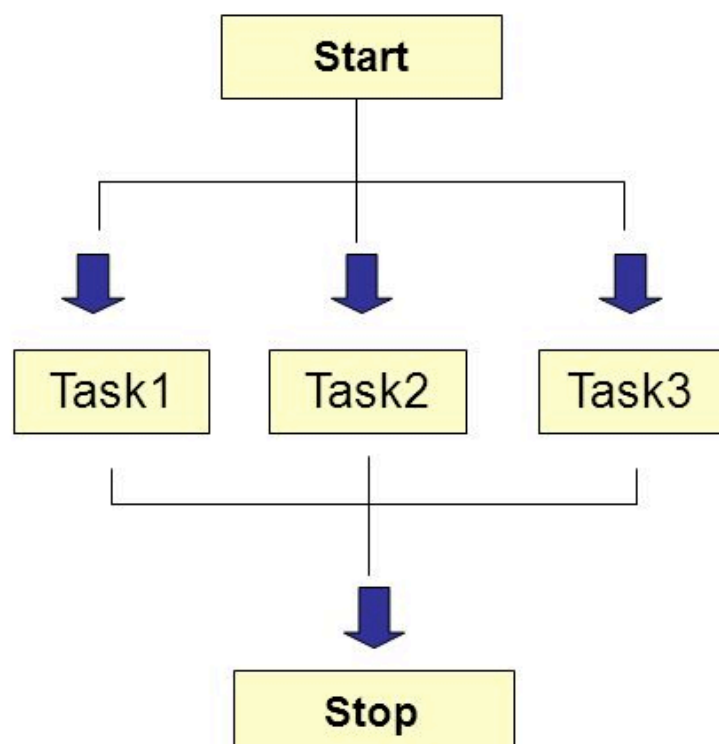


Figure 2 : La différence de l'exécution en utilisant multi-thread et sans thread

Mécanisme de HeartBeat

Comme ce que nous avons mentionné dans la partie de **Communication par socket**, la problématique c'est que les deux parties garderont leur socket ouvert indéfiniment jusqu'au moment où le client envoie un message de «exit» au serveur. Cela laisse ouverte la possibilité qu'un côté puisse fermer son socket intentionnellement ou en raison d'une erreur, sans en informer l'autre côté. Afin de détecter ce scénario et de libérer les connexions périmées, le mécanisme de **Heartbeat** sera une solution pertinente sur la connexion de TCP.

Grosso modo, les clients envoient des notifications périodiquement que l'application gestionnaire peut recevoir pour informer au serveur de l'état de sa connexion et dès que le serveur reçoit ce message, il va renvoyer un acquittement au client qui signifie que le serveur a bien reçu l'état de sa connexion et lui informe également de l'état du serveur.

Par ailleurs, puisqu'il n'y a qu'un seul serveur dans notre projet, afin de réduire la pression sur le serveur, nous avons choisi de laisser le client envoyer activement des paquets cardiaques.

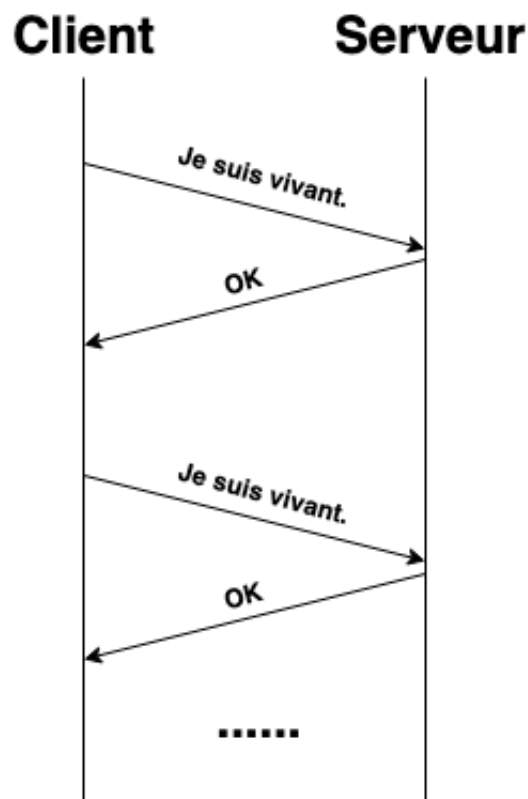


Figure 3 : Scénario de Heartbeat

Au côté serveur, si la connexion est silencieuse pendant *heartbeat_time* secondes, le serveur continuera d'attendre un moment, s'il n'y a toujours pas de message du client, cela signifie qu'il y a un problème avec le client. Dans ce cas-là le serveur se déconnecte activement et affiche un message d'erreur « Le client est en panne ».

Et au côté client, la manière à détecter des déconnexions imprévues est similaire que cela du serveur.

Objectifs à atteindre

- Établir une application permettant la transmission de messages entre le client et le serveur
- Être capable de diffuser le message du serveur à tous ses clients actifs
- Utiliser des threads différents pour gérer la communication de multi-utilisateurs
- Gérer la déconnexion à l'aide de le mécanisme Heartbeat
- Gérer l'unicité de client

Cas d'utilisation théorique

Cette partie décrit le déroulement provisoire de l'application lorsqu'un utilisateur la démarre.

Tout d'abord, l'utilisateur devrait prendre un pseudonyme unique à utiliser pendant la session de discussion. Une fois que son pseudonyme est validé, les autres utilisateurs seront notifiés de sa connexion.

Ensuite, l'utilisateur pourra écrire son message et appuyer Entrée à l'envoyer. Le message s'affichera sur la console des autres utilisateurs sous la forme `<pseudonyme_expéditeur> a dit : <le message>` alors que l'utilisateur ne voit que la ligne de message qu'il a saisi à envoyer. Il pourra également consulter des messages provenant d'autres utilisateurs sur sa console.

Pour quitter la session, l'utilisateur envoie un message "exit" et les autres utilisateurs seront informés de la déconnexion à travers du serveur.

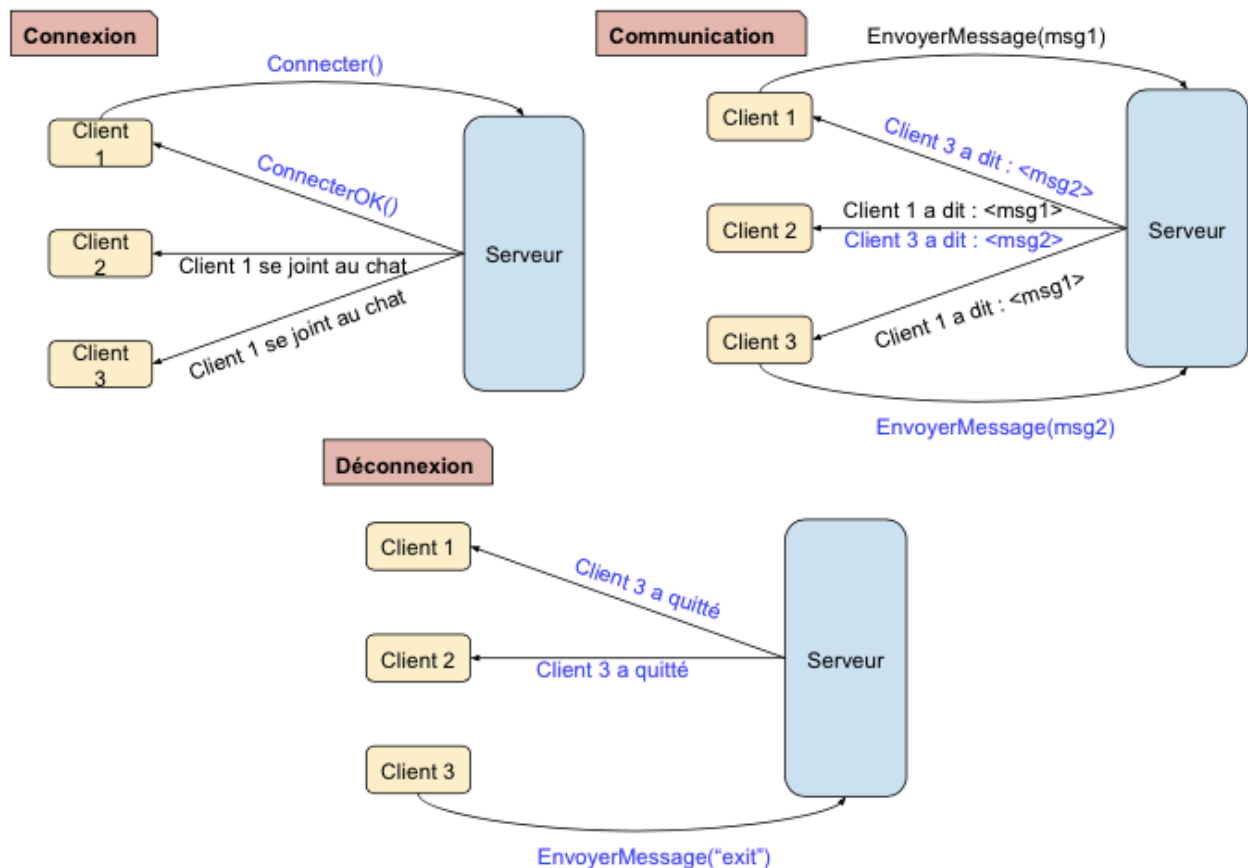


Figure 4 : Schéma d'utilisation théorique

Développement

Après avoir déterminé des fonctionnalités à implémenter, nous avons passé à la phase de réalisation. Dans cette partie, nous allons expliquer l'architecture de code, comment nous avons organisé des packages, des classes et ainsi deux structures de données importantes dans le projet. Ensuite, nous allons détailler la phase de programmation pour expliquer comment nous avons réalisé la communication de multi-utilisateurs et la gestion de déconnexion.

Architecture de code

D'après le TD1 sur le Socket, nous avons décidé de séparer le code en deux projets : `Client-chat` et `Server-chat`. Ce travail nous permet de bien organiser les ressources de code et de différencier explicitement le rôle du client et celui du serveur dans l'application.

L'architecture du côté client

```
├── clients
│   ├── Client.java
│   └── exception
│       └── PanneServeurException.java
├── message
│   ├── HBMessage.java
│   └── TextMessage.java
└── thread
    ├── HeartbeatAgent.java
    ├── HeartbeatListener.java
    ├── ReceiveMessage.java
    └── SendMessage.java
```

Nous avons 4 packages et totalement 8 classes.

Package client

Dans le package **client**, il contient d'une classe `Client` qui permet d'instancier un client, ainsi de ses propres threads lors de son démarrage de l'application. La classe s'en charge également la connexion au serveur.

Package thread

Le package **thread** comprend de 4 classes héritées de la classe Thread en Java. Les threads servent à la transmission des messages et d'état de processus entre le client et le serveur.

- `ReceiveMessage`
Un thread qui reçoit tous des objets de message venant de client et qui charge d'afficher des `TextMessages`.
- `SendMessage`
Un thread qui s'occupe d'envoyer des objets de `TextMessage`.
- `HeartbeatAgent`
Un thread qui permet d'envoyer de `HBMessage` au serveur toutes les

`DEFAULT_SAMPLING_PERIOD` secondes.

- HeartbeatListener

Un thread qui collecte tous des HBMessages et évaluer l'état du client.

Package exception

On crée une classe d'exception `PanneServeurException` afin de générer un message d'erreur personnalisé qui avertit au client que le serveur a échoué.

Package message

Ce package contient des classes dédiées au mécanisme HeartBeat. Étant donné que les messages de Heartbeat doivent être envoyés tout le temps, nous devons les simplifier autant que possible afin de réduire la pression sur le serveur et le client durant le traitement des paquets de Heartbeat. C'est pour cette raison que nous avons créé une classe spécifique `HBMessage`.

- Classe HBMessage qui ne contient que son temps de création

```
public class HBMessage implements Serializable {
    public String toString() {
        return new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").format(new
Date());
    }
}
```

- Classe TextMessage qui contient un champ `msg`

```
public class TextMessage implements Serializable{
    private String msg = "";

    public TextMessage(String msg) {
        this.msg = msg;
    }
    public String getMsg() {
        return this.msg;
    }
}
```

L'architecture du côté serveur



Le projet `server-chat` se compose de 4 packages et 6 classes. Sa structure est presque similaire à celle du client, car il contient également les packages **exception** et **message** qui ont les mêmes fonctionnalités que ceux du client.

La différence se trouve au package **server** et **thread**.

Package server

Le package contient la classe `Server` qui permet de créer un socket de connexion et de passer le socket de communication avec chaque client dans un thread séparé.

Package thread

- `MessageReceptor` est responsable de l'envoi et de la réception des objets de message au chaque client, selon une liste de clients actifs.
- `HeartbeatListener` collecte tous des HBMessages et évaluer l'état du client.

Structures de données

Hashtable

Pour faciliter la gestion des clients actifs dans la session chat en cours, nous avons créé une liste `listClient` en type **Hashtable**, une table de hachage. Cette structure de données stocke les informations de client sous la forme `[clé : valeur]` où `clé` est l'objet de thread `MessageReceptor` de client et `valeur` est le pseudonyme du client. Cela nous permet au serveur de gérer la communication de multi-utilisateurs et d'avoir l'unicité de pseudonyme.

Voici les méthodes de Hashtable qu'on utilisait

Description	Méthode	Retourner
Ajouter un élément	<code>put(clé, valeur)</code>	V
Vérifier si la liste est vide	<code>isEmpty()</code>	boolean
Vérifier si la liste contient une clé associée à la valeur donnée	<code>containsValue(Object value)</code>	boolean
Retourner la liste des clés	<code>keySet()</code>	Set

Queue

Puisqu'on doit chercher une structure de donnée qui nous permet de savoir quand le client/serveur s'est déconnecté, c'est-à-dire qu'auquel moment que le client/serveur n'a reçu plus le HBmessage et donc à l'aide de la caractéristique **FIFO** de queue qui nous permet de simuler l'ordre du temps. Les dernières nouvelles de HBMessage qui sont stockées dans la file d'attente vont être dépilées plus tard. Une fois la file d'attente devient vide, il est probable que l'autre côté n'ait pas envoyé de HBmessage depuis longtemps. On arrive à juger de l'état de l'autre côté, qui est basé sur des HBmessages reçus.

Voici les méthodes de queue fournis par java qu'on utilisait.

	throw Exception	retourner false/null
Ajouter des éléments à la fin de la queue	add(E e)	boolean offer(E e)
Retourner l'élément de tête et l'enlever	E remove()	E poll()
Retourner l'élément de tête et le garder	E element()	E peek()

Programmation

Dans la suite, nous allons détailler consécutivement les fonctionnalités de l'application réalisées en trois phases principales.

Implémentation de communication de socket

Côté du serveur

La classe **server** contient la méthode `main()` pour exécuter le programme Server-chat.

Tout d'abord, nous générons un socket de connexion au port prédéfini à l'aide de la commande `new ServerSocket(PORT_NUMBER)`.

```
// Le socket de connexion au port 1234
try {
    serverSocket = new ServerSocket(PORT_NUMBER);
    System.out.println("Le socket de connexion a été créé au port " +
PORT_NUMBER);
} catch (Exception e) {
    System.out.println("Échoué à créer un socket de connexion.");
}
```

Lorsque le socket de connexion est créé, il rentre dans une boucle infinie pour qu'il reste toujours ouvert à attendre la demande de connexion venant de client. Si la demande est acceptée, un socket de communication sera créé par la méthode `accept()` de l'objet socket de connexion ou bien le socket de client.

Troisièmement, nous créons un nouveau thread **MessageReceptor** en utilisant la commande `new MessageReceptor(socketClient, listClient)` et puis nous démarrons le thread pour qu'il commence son traitement des messages.

```
int numeroClient = 0;
while (true) {
    client = serverSocket.accept();

    // Créer un thread MessageReceptor pour le nouveau client
    MessageReceptor newMessageReceptor = new MessageReceptor(client,
listClient);
    listClient.put(newMessageReceptor, "");
    newMessageReceptor.start();

    System.out.println("Client " + numeroClient + " se connecte !");

    numeroClient++;
}
```

Côté du client

La classe **client** contient la méthode `main()` pour exécuter le programme Client-chat.

Dès le démarrage, un socket client est créé pour envoyer une demande de connexion au port du serveur. Lorsque la connexion est établie, nous exécutons deux threads pour envoyer et recevoir les messages.

```
public static void main(String args[]) throws UnknownHostException,
IOException {
    // Établir la connexion
    Socket s = new Socket("localhost", ServerPort);
    System.out.println("Connected : " + s);

    // Récupérer input and out streams
    ObjectOutputStream outputStream = new
ObjectOutputStream(s.getOutputStream());
    ObjectInputStream inputStream = new ObjectInputStream(s.getInputStream());

    // Créer des thread pour traiter des messages
    SendMessage sendMessage = new SendMessage(outputStream);
    ReceiveMessage receiveMessage = new ReceiveMessage(s, inputStream);

    // Lancer les thread
    sendMessage.start();
    receiveMessage.start();
}
```

Traitement des messages par le thread

Utilisation de thread au côté serveur : MessageReceptor

La classe **MessageReceptor** hérite la classe Thread de Java. La classe sert à communiquer au client, suivre son état et diffuser le message du client aux autres clients.

Les composants de la classe **MessageReceptor** :

- Les attributs :

Type	Nom	Description
Hashtable<MessageReceptor, String>	listClient	La liste de clients actifs
Queue	hbMsgList	La liste de message HeartBeat
Socket	client	Le Socket de communication d'un client
String	clientName	Le pseudonyme du client
ObjectInputStream	inputStream	Le stream d'entrée
ObjectOutputStream	outputStream	Le stream de sortie
Boolean	closed	L'état de socket

- Les méthodes de services
 - `void hbMsgHandler()` : Envoyer l'acquittement de Heartbeat au client et
 - `void sendObject(MessageReceptor destination, Object obj)` : Envoyer l'objet de message à la destination
 - `void terminerSocket()` : Annoncer la déconnexion d'un client aux autres clients et terminer les I/O flux et le socket.
 - `void broadcast(String msg, String clientName)` : Diffuser le message `msg` de client `clientName` aux autres clients dans la liste de clients actifs.
- La méthode principale : **void run()**

La méthode `run()` s'exécute lorsque le thread est démarré. Elle effectue des tâches dans l'ordre :

- Demander le pseudonyme du client. Vérifier les données entrées pour que le pseudonyme ne contienne pas des caractères particuliers "@,!" et soit unique.
- Créer et lancer le thread **HeartbeatListener** pour commencer le suivi d'état vivant de programme client
- Annoncer le pseudonyme du client à tout client actif à l'aide de `sendObject()`

```
public void sendObject(MessageReceptor destination, Object obj) throws
IOException {
    destination.outputStream.writeObject(obj);
    destination.outputStream.flush();
}
```

- Commencer la communication continue entre le serveur et le client. Le serveur reçoit le

message de client, s'il est un message de HeartBeat, on l'ajoute à la liste `hbMsgList` par la méthode `hbMsgHandler()`. Sinon, il diffuse le message aux autres clients à l'aide de la méthode `broadcast()`. La communication se termine lorsque le serveur reçoit un message dont le contenu est "exit".

```
while (true) {
    //Récupérer le messages entrants
    Object obj = inputStream.readObject();
    if (obj instanceof HBMessage) {
        hbMsgHandler();
    } else {
        TextMessage receivedObj = (TextMessage) obj;
        String msg = receivedObj.getMsg();
        // Quitter la conversation
        if (msg.startsWith("exit")) {
            this.closed = true;
            hbListener.closeHbListener(true);
            break;
        } else {
            // Diffuser le message aux autres clients
            broadcast(msg, this.clientName);
        }
    }
}
```

- Une fois qu'il sort de la boucle, il va fermer le socket de communication avec le client en utilisant la méthode `terminerSocket()`.

```
public void terminerSocket() {
    try {
        System.out.println(this.clientName + " se déconnecte.");
        listClient.remove(this);

        synchronized (this) {
            if (!listClient.isEmpty()) {
                for (MessageReceptor client : listClient.keySet()) {
                    if (client != null && client != this && client.clientName
!= null) {
                        try {
                            this.sendObject(client, new TextMessage("*** " +
this.clientName + " a quitté la conversation ***"));
                        } catch (IOException e1) {
                            e1.printStackTrace();
                        }
                    }
                }
            }
        }
    }
}
```

```

// Fermer la connexion
this.inputStream.close();
    this.outputStream.close();
    this.client.close();
} catch (IOException e1) {
e1.printStackTrace();
}
}

```

Utilisation de thread au côté Client : ReceiveMessage & SendMessage

Le programme Chat de client utilise deux threads pour traiter la communication avec le serveur. Cela lui permet d'exécuter simultanément le processus de réception de celui d'envoi le message. Les deux threads sont deux classes héritées de la classe Thread en Java.

ReceiveMessage

La classe **ReceiveMessage** se charge la réception d'un objet HBMessage ou d'un message du serveur pendant la session de chat.

Les composants de la classe **ReceiveMessage** :

- Les attributs :

Type	Nom	Description
Socket	client	Le Socket de communication de client
String	receivedMsg	Le contenu de message reçu
ObjectInputStream	inputStream	Le stream d'entrée
Boolean	closed	L'état de socket
Queue	hbMsgList	La liste de message HeartBeat

- Les méthodes service :

- `void interpreterMessage()` : interpreter le message reçu. Si le message est un acquittement du serveur, on l'ajout à la list hbMsgList. Sinon, on récupère le contenu du message et l'affecte à la `receivedMsg`
- `void terminerSocket()` : mettre en pause de 20 millisecondes afin d'attendre que le serveur ferme complètement le socket et puis il ferme le flux d'entrée, ainsi que son propre socket

```

public void terminerSocket() {
    // Attendre que le serveur ferme la connexion
    try {
        this.sleep(20);
    } catch (InterruptedException e1) {

```

```

        e1.printStackTrace();
    }

    // Fermer la connexion
    try {
        this.inputStream.close();
        this.client.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

- La méthode principale : **void run()**

Une boucle tourne continûment pour recevoir des messages entrants. Le message reçu est pré-traité par `interpreterMessage()`. Le contenu de message reçu s'affiche sur la console. Le processus se répète jusqu'à quand l'objet reçoit le message indique "Vous avez quitté la conversation". Il quitte la boucle et ferme le socket par `terminerSocket()`

```

// Récupérer les messages jusqu'à quand il reçoit le message de fin
while (!this.closed) {
    try {
        interpreterMessage();
        // Recevoir le message entrant
        if (this.receivedMsg != null) {
            synchronized (this) {

                // Quitter la boucle si le socket a fermé, sinon afficher le message
                // sur la console
                if (this.receivedMsg.equals("Vous avez quitté la
conversation")) {
                    System.out.println("Bye!");
                    this.closed = true;
                    break;
                } else {
                    System.out.println(this.receivedMsg);
                    this.receivedMsg = null;
                }
            }
        }
    } catch (IOException ex) {
        terminerSocket();
        System.out.println(" Message reçu est erroné ");
        break;
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
}

```

SendMessage

La classe **SendMessage** est assez simple, elle crée et démarre un objet de la classe **HeartbeatAgent**. Le fonctionnement de cet objet sera présenté dans la partie suivante. Ensuite, le thread **SendMessage** récupère l'input sur la console et l'envoie au serveur à l'aide de la méthode `sendObject()` jusqu'à quand il finit à envoyer le message "exit". Lorsque le message de fin est envoyé, le thread ferme le socket en utilisant la méthode `terminerSocket()`.

Les méthodes service `sendObject()` ressemble à celle utilisée par **MessageReceptor**, alors que la méthode `terminerSocket()` des threads **SendMessage** et **ReceiveMessage** sont identiques.

- La méthode principale : **void run()**

```
public void run() {
    // Démarrer le heartbeatAgent dans le thread de sendMessage
    HeartbeatAgent heartbeatAgent = new HeartbeatAgent(outputStream,
this.closed);
    heartbeatAgent.setPriority(Thread.MIN_PRIORITY);
    heartbeatAgent.start();

    while (!this.closed) {

        synchronized (this) {
            // Récupérer le contenu du message
            String sendMsg = scn.nextLine();
            if (sendMsg != null) {
                try {
                    this.sendObject(new TextMessage(sendMsg));

                    // Quitter la boucle après avoir envoyer le message de fermeture du
socket
                    if (sendMsg.equals("exit")) {
                        // Fermer le flux de SendMessage
                        this.closed = true;
                        // Fermer le flux de heartbeatAgent
                        heartbeatAgent.closeHeartbeatAgent(true);
                        break;
                    }
                } catch (IOException ex) {
                    Logger.getLogger(SendMessage.class.getName()).log(Level.SEVERE, null,
ex);

                    System.out.println("Échoué à envoyer le message. ");
                    break;
                }
            }
        }

        // Fermer le flux
        if (this.closed) {
```

```

    terminerSocket();
}
}

```

Dans les threads de transmission de message, nous avons utilisé la méthode `synchronized(this){}` de Java afin d'assurer qu'il y a un seul thread qui utilise l'objet `this`.

Réalisation du mécanisme de HeartBeat

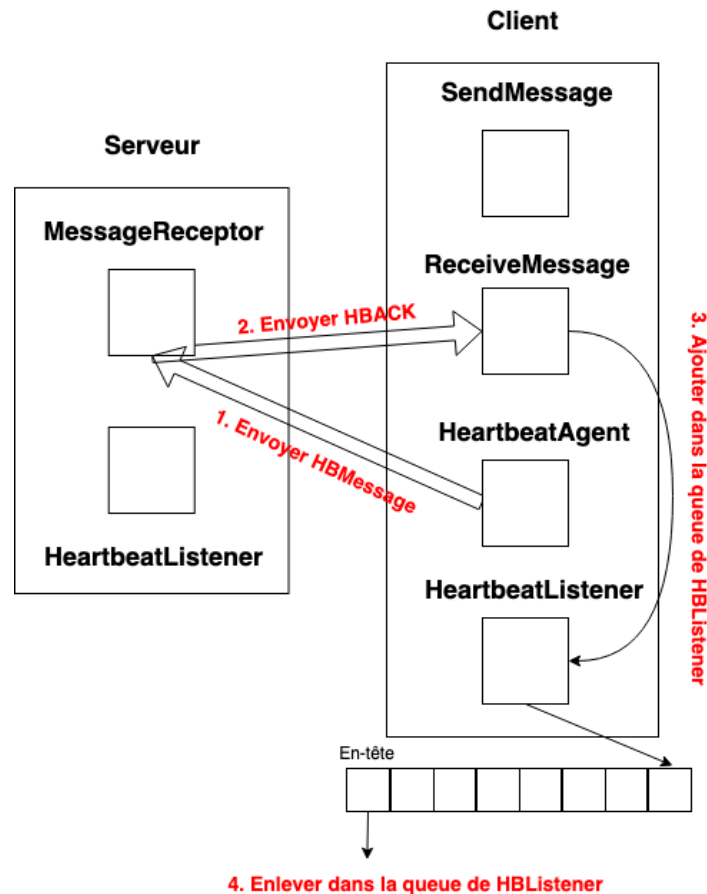


Figure 5 : Schéma de le mécanisme de HeartBeat

Nous vous avons expliqué le principe général du mécanisme de heartbeat. Pour le réaliser, il nous faut penser à quatre points.

1. Comment envoyer d'objet `HBMessage`

On crée une fonction commune qui permet d'envoyer un objet de `HBMessage` ou `TextMessage` en fonction du paramètre reçu à l'aide de `java.io.ObjectOutputStream`.

```

public void sendObject(Object obj) throws IOException {
    this.oos.writeObject(obj);
    this.oos.flush();
}

```

Donc on peut la mettre dans une boucle infinie du thread `HeartbeatAgent`.

```
// HeartbeatAgent.java
while (!this.closed) {
    this.sendObject(new HBMessage());
    Thread.sleep(DEFAULT_SAMPLING_PERIOD * 1000);
}
```

2. Comment distinguer les objets de message

Une fois que le thread reçoit un objet de message, il appliquera la fonction `interpretMessage()`. Nous atteignons à récupérer d'objet envoyé par l'autre côté à travers de la méthode `readObject()`, et puis on le détecte par l'opérateur `instanceof` qui nous permet de savoir si cet objet est un type donné.

- Si oui, on l'ajoute dans notre queue de `HeartbeatListener`.
- Sinon, il s'agit que cet objet est plutôt un objet de `TextMessage`.

```
public void interpretMessage() throws ClassNotFoundException, IOException {
    Object obj = this.inputStream.readObject();
    if (obj instanceof HBMessage) {
        hbMsgList.add("ACK");
    } else {
        TextMessage receivedObj = (TextMessage) obj;
        this.receivedMsg = receivedObj.getMsg();
    }
}
```

3. Comment juger l'état de l'autre côté à partir des HBMessage reçus

On sait le fait que le message reçu précédemment sera d'abord ajouté à la file d'attente, et la file lira et supprimera l'élément en-tête toutes les `DEFAULT_READ_PERIOD / 1000` secondes. C'est-à-dire qu'une fois la file d'attente vide, cela signifie que, récemment, le thread n'a pas reçu de message, ce qui nécessite de lui rappeler d'attirer l'attention. Dans ce cas-là, nous pouvons attendre un intervalle de temps (`DEFAULT_CHECK_PERIOD`) pour confirmer que le thread ne recevoir plus de nouveaux messages. Donc, on arrive à conclure que l'autre côté avait du problème sur la connexion. Ensuite, on retournera une erreur à travers d'une exception personnalisée et aussi terminera ce socket.

```
//HeartbeatListener.java
...
public void checkClientAlive() throws PanneClientException {
    // Si la queue de HBMsg n'est pas vide -> cette client est active
    try {
        if (hbMsgList.peek() != null) {
            System.out.println("HBListener:\t" + hbMsgList.remove() + " is alive.");
            Thread.sleep(DEFAULT_READ_PERIOD);
        } else {
```



```

        Thread.sleep(DEFAULT_CHECK_PERIOD);
        hbMsgList.element();
        //Si la file d'attente est toujours vide, une erreur sera générée
        lors de l'exécution de cette méthode
    }

    } catch (Exception e) {
        // TODO: handle exception
        throw new PanneClientException("Le client \"" + clientName + "\" est
en panne.");
    }

}

public void run() {
    while (!this.closed) {
        try {
            checkClientAlive();
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
            //terminer ce thread
            this.setHbListenerClosed(true);
        }
    }
}
}

```

4. Comment terminer le thread HbListener

Par la suite, nous allons combiner le client et le serveur pour expliquer comment le client se déconnecter.

Afin de permettre aux clients de quitter le programme de chat en toute sécurité, nous avons conçu le processus suivant. Il aussi bien conclut le processus de la déconnexion avec socket.

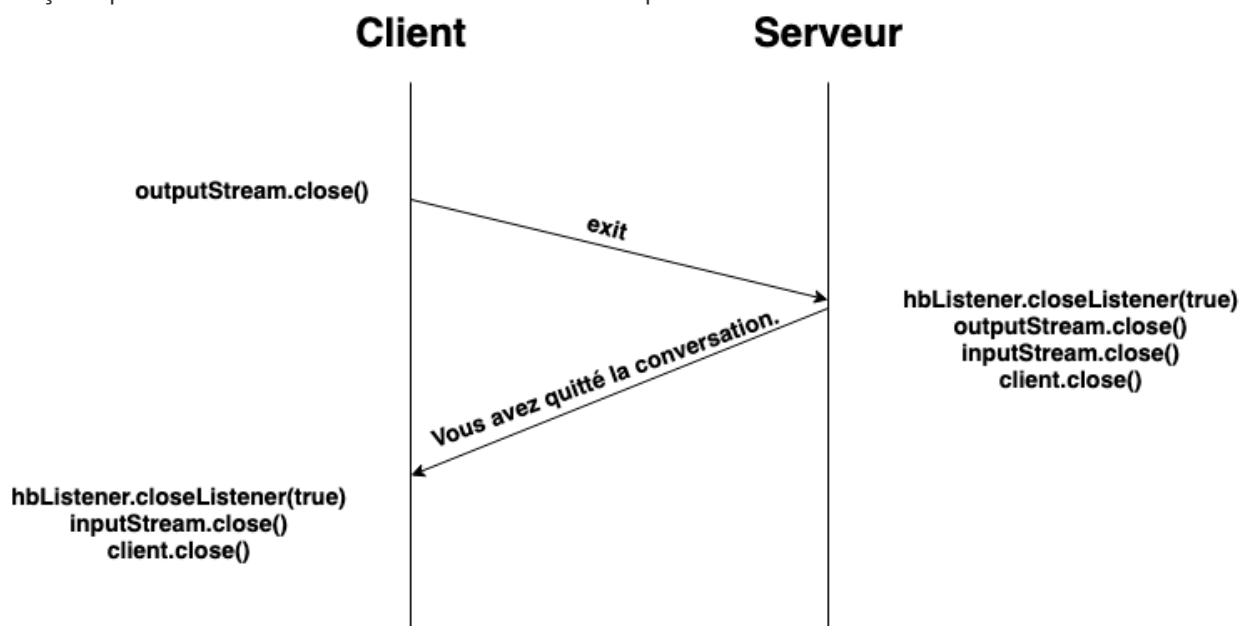


Figure 7 : Schéma de la déconnexion par un message "exit"

Nous avons construit une interface `closeHBLListener(true)` dans le thread `HeartbeatListener` pour fermer ce thread, afin d'éviter le cas que le client ne soit confondu par le serveur avec la sortie inattendue même s'il quitte normalement.

Résultat

Finalement, nous avons réussi à implémenter une application qui valide tous les objectifs fixés que nous avons mentionnés précédemment. Ce devoir nous avons permis de pratiquer la programmation en Java, de renforcer nos connaissances sur le thread et la communication de socket. Nous avons également appris comment commenter le code et générer une documentation en Javadoc.

Nous avons également pu répondre aux questions proposées dans le sujet.

Comment garantir qu'un pseudonyme est unique ?

- Utiliser une HashTable pour stocker le socket de client et son pseudonyme.

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

- Implémenter le mécanisme HeartBeat afin de suivre l'état de programme.

Utilisations pratiques

Nous pouvons imaginer le scénario suivant, un total de trois utilisateurs `a`, `b` et `c` discutent en même temps. Après de la salutation, on imagine trois cas.

1. Le client `a` quitté normalement le chat avec "exit".
2. Le client `b` force de quitter le programme.
3. Lorsque seul le client `c` est laissé, nous forçons l'arrêt du serveur, donc `c` est obligé de se déconnecter.

Cas général : Le client `a` quitte normalement le chat avec "exit"

La console du `serveur` pendant la session de chat

```
Console Client2.java ObjectOutputStream.class Socket.class
<terminated> Server [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_241.jdk/Contents/Home/bin/java (Apr 10, 2020, 2:33:34 PM)
Le socket de connexion a été créé au port 1234
Client 0 se connecte !
Pseudo de nouveau client : a
HBListener:      is alive.
Client 1 se connecte !
HBListener:      a is alive.
Pseudo de nouveau client : b
HBListener:      is alive.
Client 2 se connecte !
Pseudo de nouveau client : c
HBListener:      is alive.
HBListener:      a is alive.
HBListener:      b is alive.
HBListener:      c is alive.
HBListener:      a is alive.
Broadcast message a été envoyé par a
HBListener:      b is alive.
HBListener:      c is alive.
HBListener:      a is alive.
HBListener:      b is alive.
HBListener:      c is alive.
Broadcast message a été envoyé par c
```

Le client `a` quitté le chat tranquillement

```
bin — haida@LucieLYU — .ient-chat/bin — zsh — 65x18
→ bin git:(chat3) * java clients.Client
Connected : Socket[addr=localhost/127.0.0.1,port=1234,localport=65200]
Running HeartbeatAgent
Entrer votre pseudonyme :
a
a a rejoint la conversation. Tapez 'exit' pour se déconnecter

-----

b a rejoint la conversation
c a rejoint la conversation
Bonjour
c a dit : Bonjour a!
b a dit : Bonjour, a et b!
exit
Bye!
→ bin git:(chat3) █
```

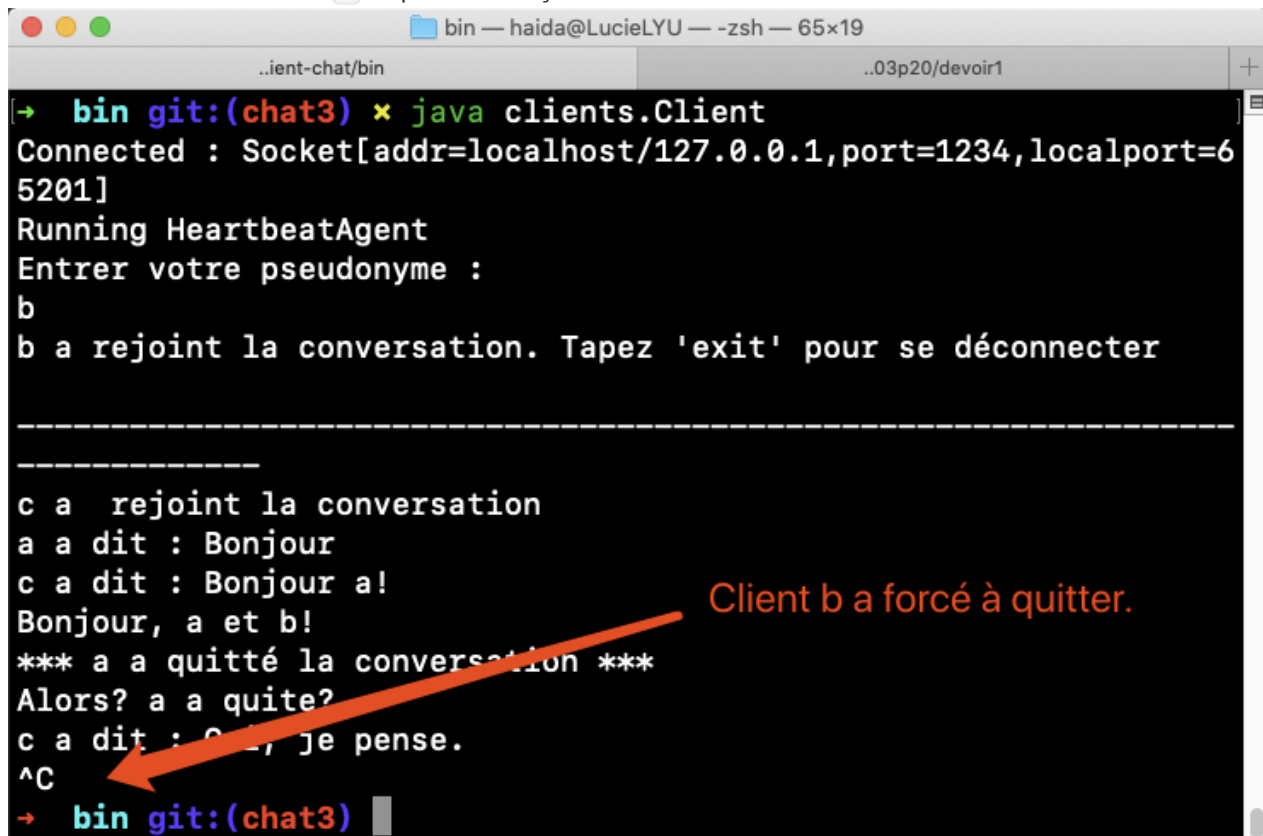
La console du `serveur` bien s'affiche la déconnexion du client a

```
Console Client2.java ObjectOutputStream.class Socket.class
<terminated> Server [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_241.jdk/Contents/Home/bin/java (Apr 10, 2020, 2:33:34 PM)
HBListener:      c is alive.
HBListener:      a is alive.
HBListener:      b is alive.
Broadcast message a été envoyé par b
HBListener:      c is alive.
HBListener:      a is alive.
HBListener:      b is alive.
HBListener:      c is alive.
a se déconnecte.
HBListener:      b is alive.
HBListener:      c is alive.
```

Client a bien se déconnecte.

Déconnexion de client sans que le serveur soit prévenu

À ce moment-là, le client `b` a quitté de façon inattendue



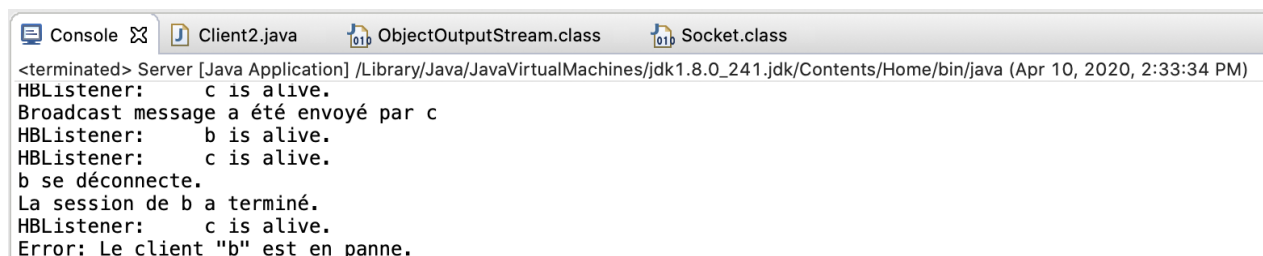
```
bin — haida@LucieLYU — zsh — 65x19
..ient-chat/bin ..03p20/devoir1
[→ bin git:(chat3) × java clients.Client
Connected : Socket[addr=localhost/127.0.0.1,port=1234,localport=65201]
Running HeartbeatAgent
Entrer votre pseudonyme :
b
b a rejoint la conversation. Tapez 'exit' pour se déconnecter

-----

c a rejoint la conversation
a a dit : Bonjour
c a dit : Bonjour a!
Bonjour, a et b!
*** a a quitté la conversation ***
Alors? a a quite?
c a dit : Oui, je pense.
^C
→ bin git:(chat3) █
```

La console du `serveur` s'affiche deux messages d'erreur :

- Le premier est généré par le `InputStream` de socket
- Le second vient de notre mécanisme de `HeartBeat`, qui bien détecte la déconnexion imprévue du client



```
Console Client2.java ObjectOutputStream.class Socket.class
<terminated> Server [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_241.jdk/Contents/Home/bin/java (Apr 10, 2020, 2:33:34 PM)
HBListener: c is alive.
Broadcast message a été envoyé par c
HBListener: b is alive.
HBListener: c is alive.
b se déconnecte.
La session de b a terminé.
HBListener: c is alive.
Error: Le client "b" est en panne.
```

Et le retrait inattendu de le client `b` n'affectera pas la communication des autres client.

Déconnexion de serveur sans que le client soit prévenu

Maintenant, si nous arrêtons le `serveur` sans arrêter l'utilisateur `c`, la console du client `c` affichera également deux messages d'erreur. Le second est fourni par le mécanisme de `HeartBeat`, et le programme client se fermera automatiquement

```
bin — haida@LucieLYU — .ient-chat/bin — zsh — 65x21
→ bin git:(chat3) ✖ java clients.Client
Connected : Socket[addr=localhost/127.0.0.1,port=1234,localport=65202]
Running HeartbeatAgent
Entrer votre pseudonyme :
c
c a rejoint la conversation. Tapez 'exit' pour se déconnecter

-----
a a dit : Bonjour
Bonjour a!
b a dit : Bonjour, a et b!
*** a a quitté la conversation ***
b a dit : Alors? a a quite?
Oui, je pense.
*** b a quitté la conversation ***
b est en panne
Message reçu est erroné
Error: Le serveur est en panne.
→ bin git:(chat3) □
```

Installation

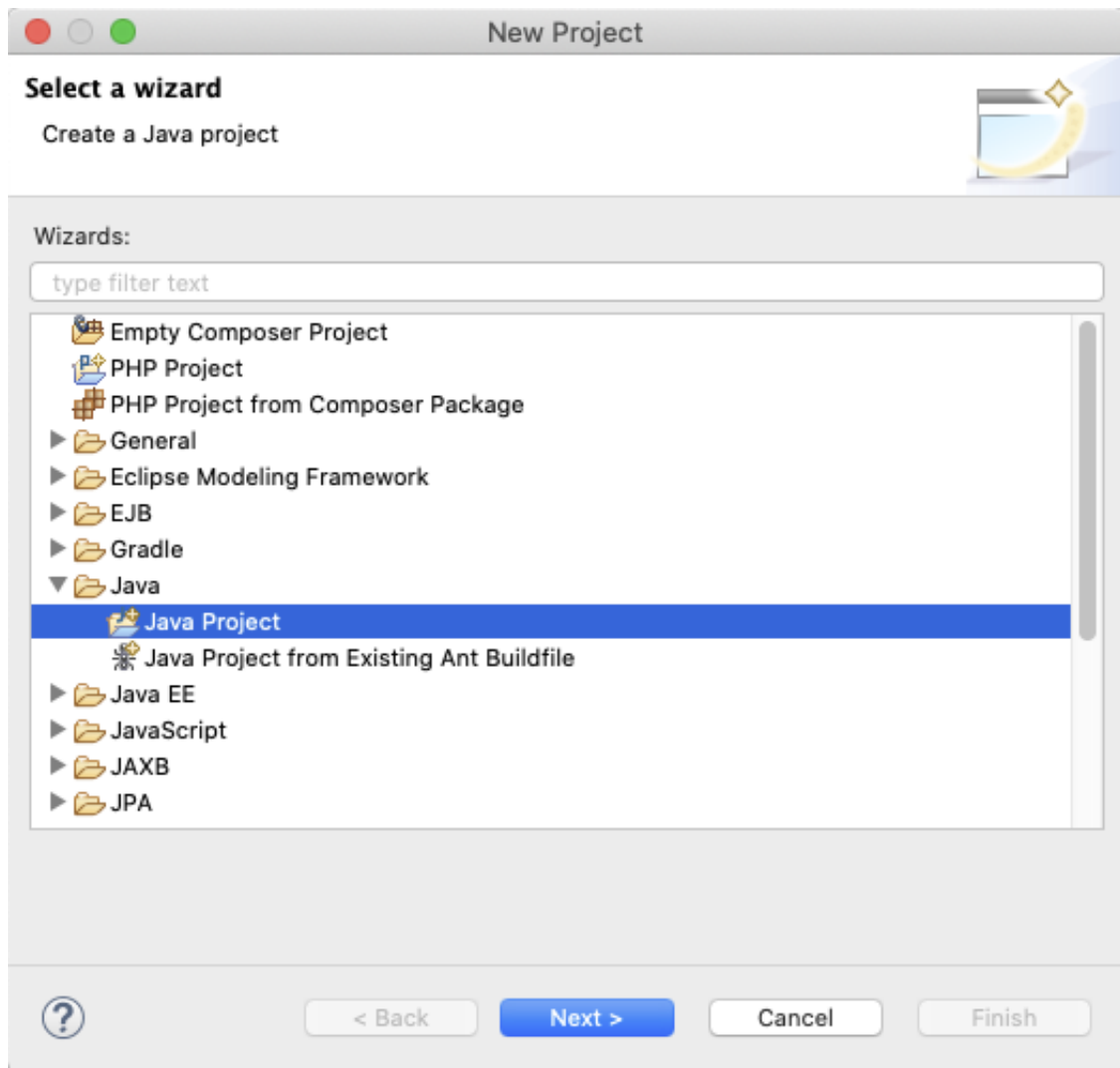
Les instructions suivantes se présentent l'installation, ainsi que l'exécution du projet.

Prérequis

- [Eclipse IDE 2019-12](#)
- [Java SE 8](#)

Import de projet en Eclipse

- Clôner les projets sur [gitlab](#)
- Aller à `File -> New -> Project`
- Choisir `Java Project` et cliquer `Next>`



- Décrocher `Use default location` -> choisir le chemin de notre projet `client-chat` et `serveur-chat` -> cliquer `Next>`

New Java Project

Create a Java Project

Create a Java project in the workspace or in an external location.

Project name:

☐ Use default location

Location:

JRE

☒ Use an execution environment JRE:

☐ Use a project specific JRE:

☐ Use default JRE (currently 'Java SE 8 [1.8.0_241]') [Configure JREs...](#)

Project layout


☐ Use project folder as root for sources and class files


☒ Create separate folders for sources and class files [Configure default...](#)

Working sets

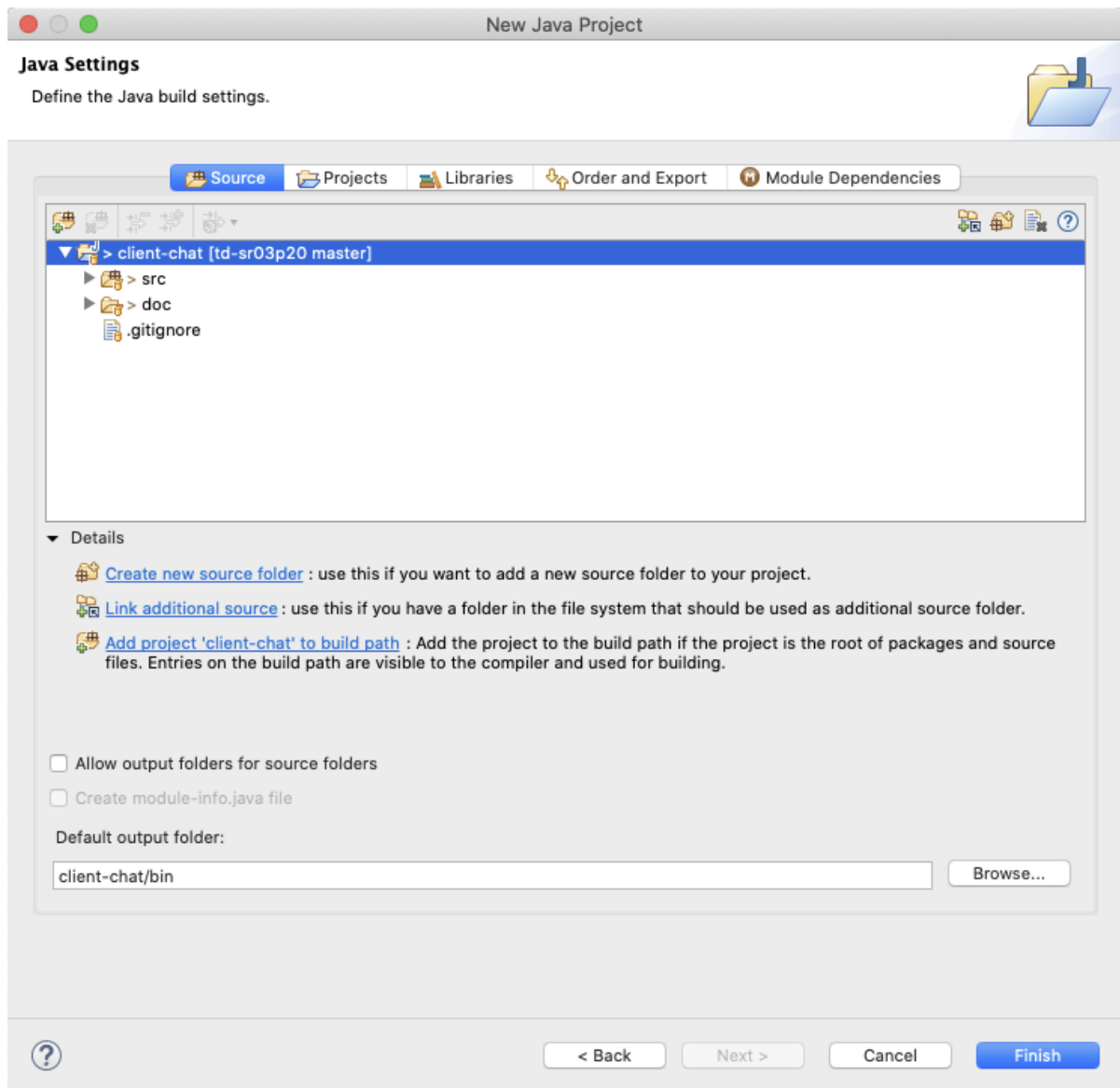
☐ Add project to working sets

Working sets:

 The wizard will automatically configure the JRE and the project layout based on the existing source.



- Rappelez-vous le chemin ci-dessous dans `Default output folder` et cliquer `Finish`



Execution de l'application

Sur le terminal

Pour lancer le serveur

- Aller au dossier de output de projet serveur-chat

```
cd /serveur-chat/bin
# Il faut remplacer le chemin de output si c'est différent que chez vous.
```

- Lancer la classe de serveur

```
java server.Server
```

Pour lancer le client

- Aller au dossier de output de projet client-chat

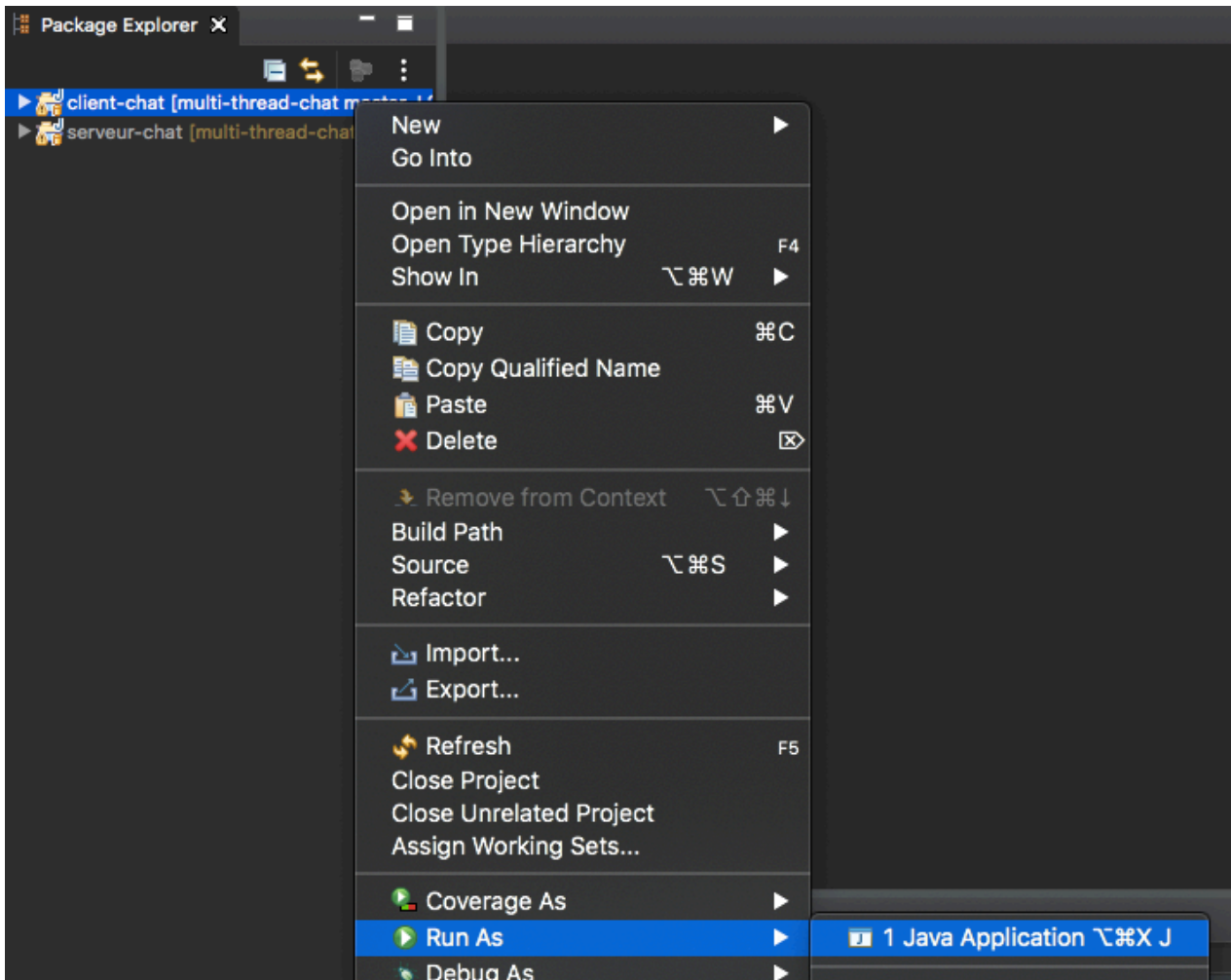

```
cd /client-chat/bin
# Il faut remplacer le chemin de output si c'est différent que chez vous.
```

- Lancer la client

```
java client.Client
```

Sur Eclipse

- Cliquer droite au nom du projet -> choisir Run As -> Java Application



Consulter la documentation

- Ouvrir le fichier `index.html` se trouvant dans le répertoire `/multi-thread-chat/client-chat/doc/` sur le navigateur (remplacer `client-chat` par `server-chat` pour la documentation du serveur)

File

/multi-thread-chat/client-chat/doc/index.html

★

Bookmarks

Ent

Basic HTML and H...

Journal

codeGuide

GoogleVN

GI05

TN10

W

All Classes

Packages

clients

exception

message

thread

All Classes

Client

HBMessage

HeartbeatAgent

HeartbeatListener

PanneServeurException

ReceiveMessage

SendMessage

TextMessage

OVERVIEW

PACKAGE

CLASS

USE

TREE

DEPRECATED

INDEX

HELP

PREV

NEXT

FRAMES

NO FRAMES

Packages

Package	Description
clients	
exception	
message	
thread	

OVERVIEW

PACKAGE

CLASS

USE

TREE

DEPRECATED

INDEX

HELP

PREV

NEXT

FRAMES

NO FRAMES