Cours 8 Modèle client-serveur

Programmation objets, web et mobiles en Java
Licence 3 Professionnelle - Multimédia

Pierre Talbot (talbot@ircam.fr)

UPMC/IRCAM

11 janvier 2018

Le menu

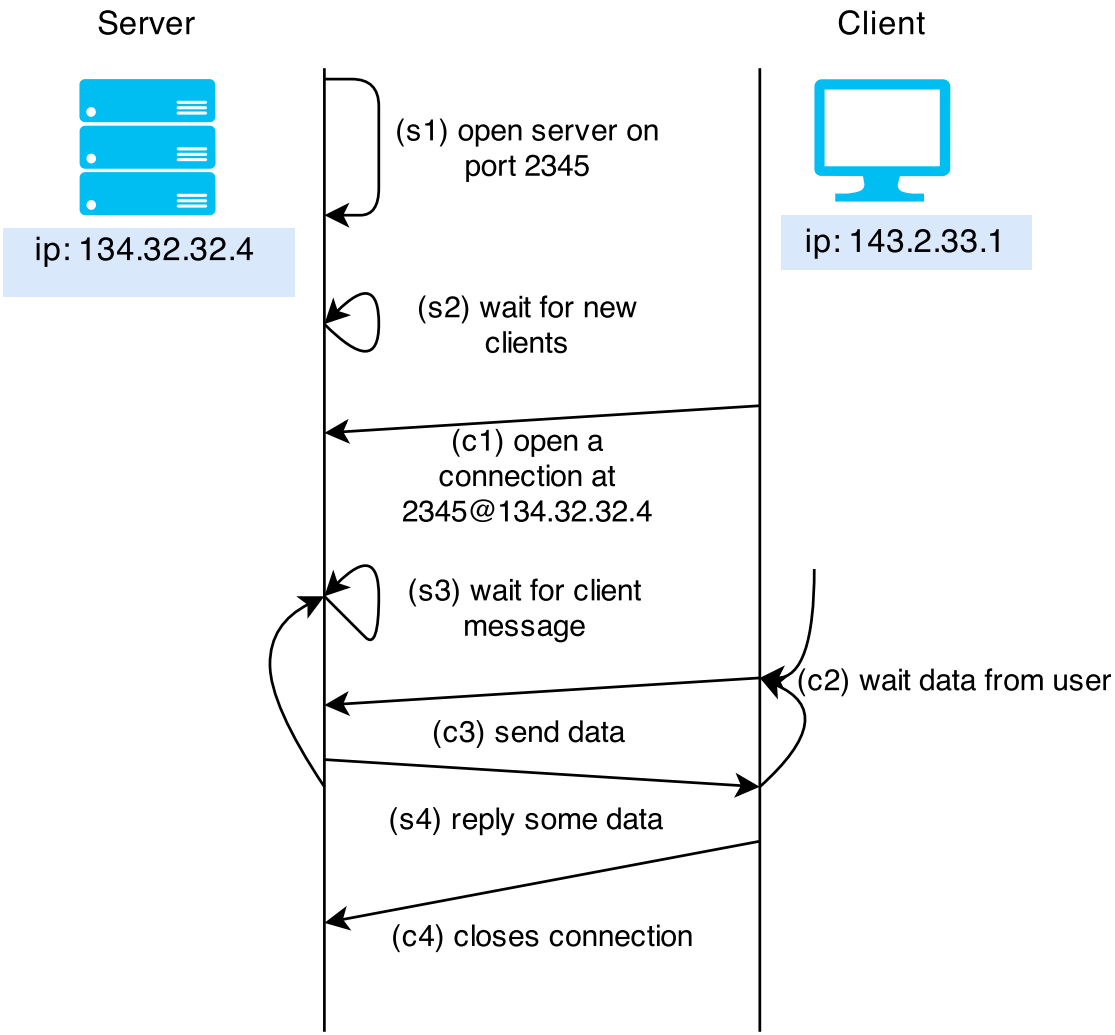
Quelques bases

- Chaque machine est identifiée avec une adresse IP.
- On ouvre un canal de communication sur un port particulier (80 pour `http`).
- Ceux de 0 à 1023 sont réservés pour certains protocoles reconnus.

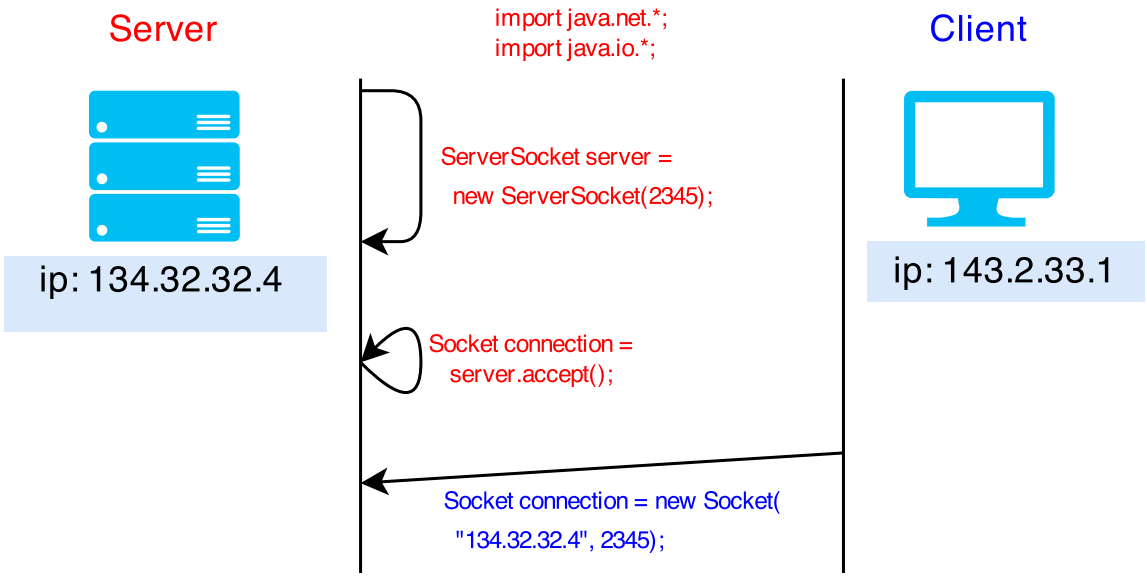
Modèle client-serveur

- Un serveur écoute les demandes des clients sur un port particulier.
- Un client se connecte au serveur avec les coordonnées (ip, port).
- Le serveur peut agir comme un intermédiaire pour la communication inter-clients.
- C'est un modèle centralisé car le serveur est le noyau, sans lui, les clients deviennent inutiles (à l'opposé des modèles pair à pair).

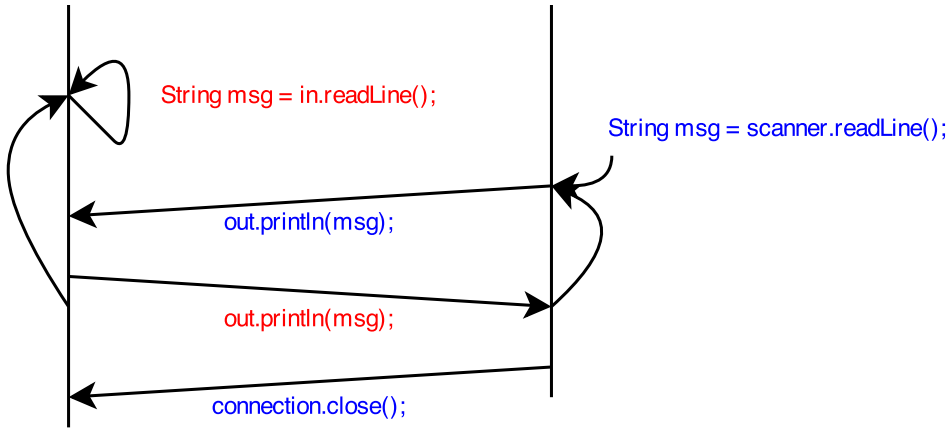
Scénario de communication client-serveur



Client-serveur en Java




```
PrinterWriter out =
  new PrinterWriter(connection.getOutputStream(), true);
BufferedReader in =
  new BufferedReader(new InputStreamReader(connection.getInputStream()));
```



Le menu

Ressources en Java

- Certains objets doivent libérer des ressources à la fin de leur vie.
- Par exemple, un fichier ou un *socket* doivent être fermés.

Exemple

```
Socket socket = new Socket(hostName, portNumber);
// ... code
socket.close();
```

Problème ?

Ressources en Java

- Certains objets doivent libérer des ressources à la fin de leur vie.
- Par exemple, un fichier ou un *socket* doivent être fermés.

Exemple

```
Socket socket = new Socket(hostName, portNumber);
// ... code
socket.close();
```

Problème ? Le *socket* n'est jamais fermé si une exception est lancée dans 'code'.

Ressources en Java

Jusqu'en Java 6 il fallait englober le code dans un *try-catch* et utiliser la clause *finally* pour fermer les ressources.

Exemple

```
Socket socket = new Socket(hostName, portNumber);
try {
  // ... code
} catch (Exception e) {
} finally {
  socket.close();
}
```

Problème : il est facile d'oublier de libérer les ressources utilisées.

try-with-resources

En Java 7, l'ajout de l'instruction *try-with-resources* permet la fermeture automatique des ressources.

Exemple

```
try (Socket socket = new Socket(hostName, portNumber))
{
  // ... code
}
```

- Le bloc `catch` n'est pas obligatoire, le `try` n'est plus réservé qu'aux traitements des exceptions.
- Une ressource est une classe qui implémente `Closeable`.
- La méthode `close()` est alors appelée automatiquement.

try-with-resources

On peut initialiser plusieurs ressources dans un `try`.

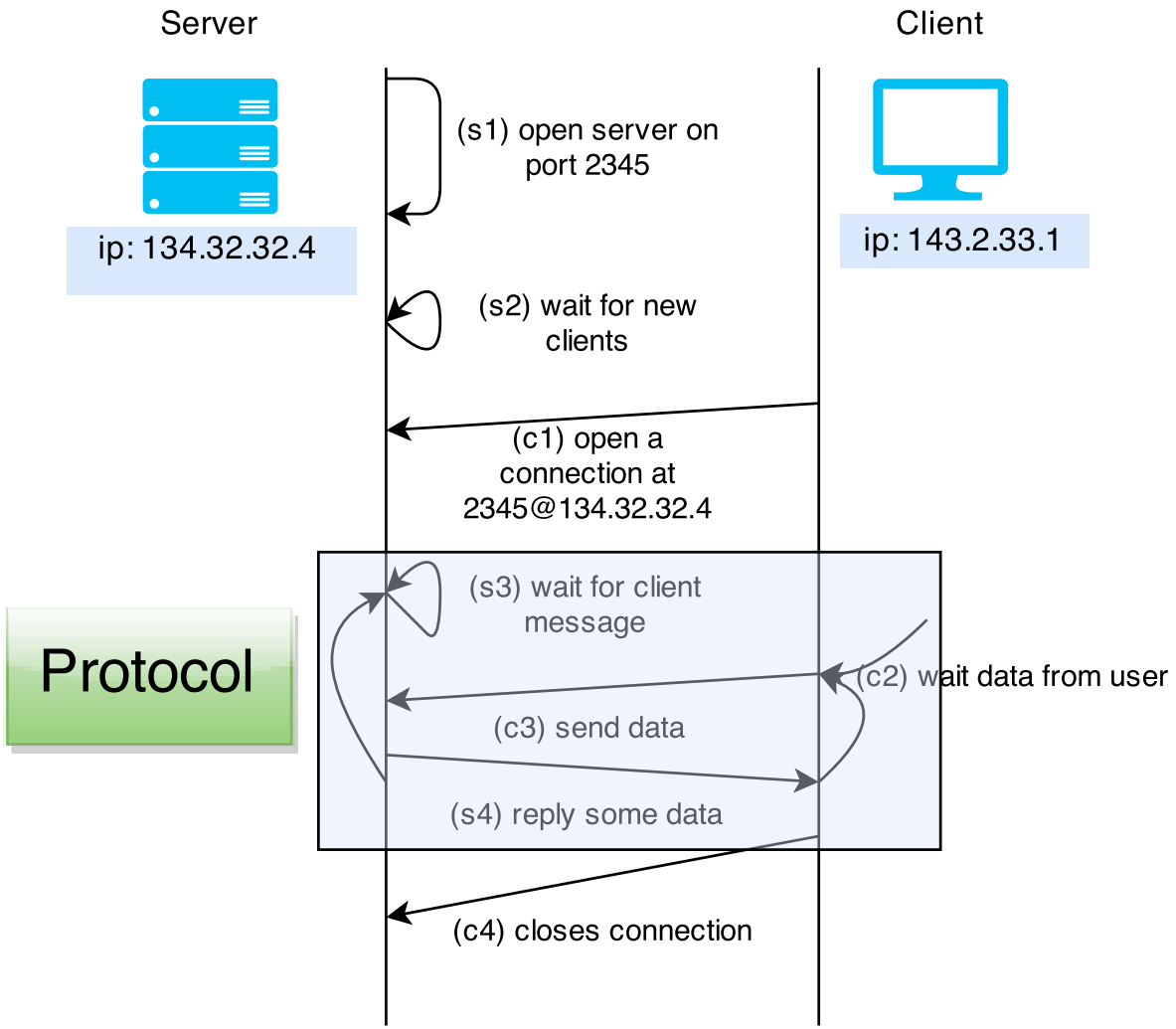
Exemple

```
try (
  Socket socket = new Socket(hostName, portNumber);
  PrintWriter out = new PrintWriter(socket.getOutputStream())
)
{
  // ... code
}
```

Noter que `socket` sera bien fermé si le constructeur de `PrintWriter` lance une exception.

Le menu

Protocole


Server
Client
ip: 134.32.32.4
ip: 143.2.33.1
(s1) open server on port 2345
(s2) wait for new clients
(c1) open a connection at 2345@134.32.32.4
Protocol
(s3) wait for client message
(c2) wait data from user
(c3) send data
(s4) reply some data
(c4) closes connection

Protocole

- La manière dont s'enchaine les messages définit le protocole.
- Il spécifie la manière dont le serveur et les clients communiquent.
- S'il est bien documenté, on peut implémenter un client sans consulter le code du serveur.

Exemple

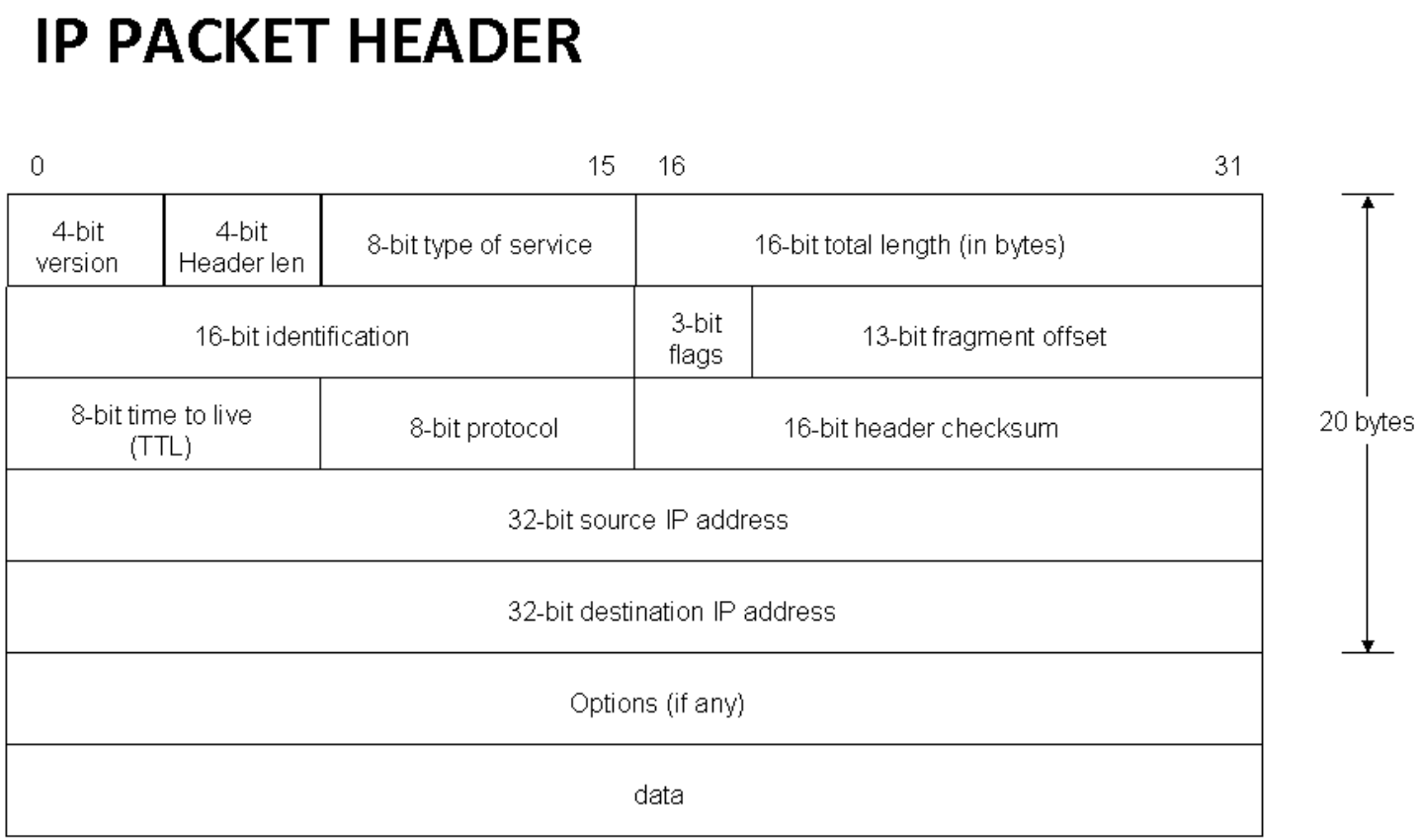
1. Le client envoie un pseudo et mot de passe.
2. Le serveur regarde si c'est correct et envoie ok, sinon ko.
3. Le client repart en (1) si c'est ko. Sinon il demande ses informations de profil.
4. Le serveur envoie les informations.
5. ...

Format de protocole

- Deux grandes familles :
  1. Binaire : Les données sont structurées et interprétées suivant la taille en octets des différents champs.
  2. Texte : Les données sont des chaines de caractères pouvant suivre un format de haut niveau (XML, JSON, . . .).

Procotole au format binaire

Par exemple, les protocoles réseaux sont spécifiés avec un format binaire.



Procotole au format binaire

- L'avantage est que la taille d'un paquet est minimisée.
- Néanmoins ils sont peu lisibles, plus difficiles à implémenter et inadéquats pour l'interopérabilité.
- **À n'utiliser que si vous avez essayé et montré que le format texte était trop lent**.

Protocole au format texte

Exemple tiré du serveur d'*add-ons* du jeu *Battle for Wesnoth* (`http://hyc.io/wesnoth/umcd.pdf`).

Requête de suppression d'un add-on

- Format :

```
[request_umc_delete]
  id = ID
  password = PASSWORD
[/request_umc_delete]
```

- Fields description :

ID	The ID of the UMC we want to delete.
PASSWORD	The password of the UMC.

Protocole au format texte

Réponse du serveur

- An error packet can be sent for the common reasons (see 2.4.2) but also because :
  1. The password is wrong.
- In case of success, a packet with no field is sent.

```
[request_umc_delete]
[/request_umc_delete]
```

Exercice : Protocole avec JSON

- Encapsuler le message dans un paquet au format JSON.
- Pour se faire spécifier un protocole très simple.

Exemple

```
{
  name: "request_umc_delete",
  id: 132,
  password: "UTE6542162143ECUSACE"
}
```

Exemple de spécification JSON : `https://github.com/ptal/online-broker/wiki/Online-broker-API`

Librairie JSON

Dépendance Maven (à ajouter dans pom.xml)

```
<dependency>
  <groupId>org.json</groupId>
  <artifactId>json</artifactId>
  <version>20141113</version>
</dependency>
```

Exemple

```
import org.json.simple.JSONObject;
//...
JSONObject obj = new JSONObject();
obj.put("name","request_umc_delete");
obj.put("id", new Integer(132));
obj.put("password", "UTE6542162143ECUSACE");
StringWriter out = new StringWriter();
obj.writeJSONString(out);
String jsonText = out.toString();
JSONObject sameObj = new JSONObject(jsonText);
```

Le menu

Serveur multi-clients

Problématique

- Comment est-ce que le serveur peut gérer plusieurs clients simultanément ?
- On voudrait effectuer plusieurs actions concurrentes :
  1. Accepter des nouveaux clients.
  2. Attendre les messages des clients déjà connectés.
- Le problème est que ces deux actions sont bloquantes, on peut soit faire l'une ou l'autre.

N+1 programmes

L'intuition est faire :

- 1 programme qui accepte les nouveaux clients.
- N programmes qui communiquent avec les N clients connectés.

Générer autant de processus est très lourd pour le système et consomme trop de ressources. On va utiliser des processus dit "légers" : les *threads*.

Le menu

Threads

Il existe deux façons de créer des *threads*, en héritant de `Thread` ou en implémentant `Runnable`.

```
class Connection extends Thread {
  Socket socket;
  Connection(Socket socket) {
    this.socket = socket;
  }

  public void run() {
    // code communicating with the client
    ...
    socket.close();
  }
}
...
Connection connection = new Connection(socket);
connection.start();
```

Runnable

Il peut être judicieux de ne pas utiliser l'héritage si la classe doit hériter d'autre chose. Dès lors, on utilise l'interface `Runnable`.

```
class Connection implements Runnable {
  Socket socket;
  Connection(Socket socket) {
    this.socket = socket;
  }

  public void run() {
    // code communicating with the client
    ...
    socket.close();
  }
}
...
Thread t = new Thread(new Connection(socket));
t.start();
```

Le menu

Arrêter proprement le serveur

Pour arrêter le serveur, il faut pouvoir signaler aux différents sous-processus qu'on veut arrêter.

```
class Connection extends Thread {
  ...
  public void interrupt() {
    super.interrupt();
    try {
      socket.close();
    } catch (IOException e) {} // quietly close
  }
  public void run() {
    try {
      ...
    }
    catch (InterruptedIOException e) {
      Thread.currentThread().interrupt();
    }
    catch (IOException e) {
    }
    socket.close();
  }
}
...
Connection connection = new Connection(socket);
...
connection.interrupt();
```

Arrêter proprement le serveur

- Pour arrêter toutes les connexions, il faut les avoir enregistrées dans un tableau.
- La méthode `join` d'un *thread* permet d'attendre la fin de son exécution.

```
ArrayList<Connection> connections = new ArrayList<Connection>();
...
for(Connection c : connections) {
  c.interrupt();
}
for(Connection c : connections) {
  c.join();
}
```

Le menu

Chat en ligne

- À chaque fois que le serveur reçoit un message, celui-ci le diffuse à tous les clients connectés.
- On crée une classe `ClientRoom` contenant toutes les connexions.

```
class ClientRoom {
  ArrayList<Connection> connections;
  ...
  public void broadcast_msg(String msg) {
    for(Connection c : connections) {
      c.send(msg);
    }
  }
}
```

Deux threads pour le client

Vu que le client peut maintenant recevoir et envoyer des messages, il faut utiliser un *thread* pour chaque.

```
class Client implements Runnable {
  MessageReader msg_reader;
  public Client(...) {
    msg_reader = new MessageReader(in);
    msg_reader.start();
  }

  public void run() {
    while ((userInput = stdin.nextLine()) != null) {
      out.println(userInput);
    }
  }
}
```

Deux threads pour le client

Vu que le client peut maintenant recevoir et envoyer des messages, il faut utiliser un *thread* pour chaque.

```
class Client implements Runnable {
  MessageReader msg_reader;
  public Client(...) {
    msg_reader = new MessageReader(in);
    msg_reader.start();
  }

  public void run() {
    while ((userInput = stdin.nextLine()) != null) {
      out.println(userInput);
    }
  }
}
```

Améliorer ce code pour que le programme quitte quand l'utilisateur tape "\quit".

Le menu

Race conditions

- Les *threads* sont difficiles à utiliser quand il s'agit de partager la mémoire.
- En effet, deux *threads* peuvent écrire au même instant dans une case mémoire.
- La première écriture ne sera pas prise en compte.
- C'est notamment catastrophique pour les structures de données, car elles deviennent incohérentes.
- Exemple simple de *race condition* au tableau.

Synchronized

Le mot-clé *synchronized* permet qu'une méthode ne soit exécutée que par un seul thread à la fois.

```
public class Singleton {
  private static volatile Singleton instance = null;

  public static Singleton getInstance() {
    if (instance == null) {
      instance = new Singleton();
    }
    return instance;
  }
}
```

Quel est le soucis dans un contexte multi-thread ?

Synchronized

Cette solution permet d'éviter que `instance` soit initialisé deux fois.

```
public class Singleton {
  private static volatile Singleton instance = null;

  public static synchronized Singleton getInstance() {
    if (instance == null) {
      instance = new Singleton();
    }
    return instance;
  }
}
```

Le problème c'est qu'on verrouille la méthode à chaque fois, alors que le problème peut juste se poser à l'initialisation.

Synchronized(object)

- On peut verrouiller l'accès à un bloc grâce à la synchronisation sur un objet ou une classe.
- Le premier `if` est une optimisation pour éviter de retomber dans les travers de la méthode précédente.
- Le deuxième `if` est nécessaire pour éviter que deux instances soient créées.

```
public class Singleton {
  private static volatile Singleton instance = null;

  public static Singleton getInstance() {
    if (instance == null) {
      synchronized (Singleton.class) {
        if (instance == null) {
          instance = new Singleton();
        }
      }
    }
    return instance;
  }
}
```

Retour au serveur

- Il faut ajouter des `synchronized` aux bons endroits.
- Quelles sont les ressources que partagent les différents *threads* ?
- Principalement la liste des connections et lors du *broadcast*.
- On ne veut pas que les messages des différents clients arrivent mélangés (début de la phrase d'un, puis phrase de l'autre, puis fin de phrase).

Finalement, améliorer le serveur pour qu'il efface un client de la liste quand celui-ci se déconnecte ou qu'il envoie "\quit".

Le menu

Concurrence vs Parallélisme

Il ne faut pas confondre les deux. La concurrence est un domaine plus général qui n'implique pas forcément plusieurs unités de calcul. Définition de `http://docs.oracle.com/cd/E19455-01/806-5257/6je9h032b/index.html` :

- *Parallelism* : A condition that arises when at least two threads are executing simultaneously.
- *Concurrency* : A condition that exists when at least two threads are making progress. A more generalized form of parallelism that can include time-slicing as a form of virtual parallelism.

Mot de la fin

- Éviter de partager les ressources avec plusieurs *threads*.
- Renseignez-vous sur la concurrence par *passage de messages*.
- Utiliser les *sockets* sur la boucle locale (*localhost*) comme medium de communication inter-thread n'est **pas bête**.