

Cours n°3

Sockets TCP et UDP en Java

Safa YAHY
safa.yahi@univ-amu.fr

Sockets en Java

- Java permet d'écrire facilement et rapidement des applications réseaux.
- Le code consacré au réseau est concis.
- Offre une indépendance vis-à-vis du matériel et du logiciel.

Entrées/Sorties en Java

Entrées/Sortie en Java

Lecture

- InputStream => octets
- InputStreamReader => caractères
- BufferedReader => texte bufferisé
- BufferedInputStream => octets bufferisés
-

Ecriture

- OutputStream => octets
- OutputStreamWriter => caractères
- BufferedWriter => texte bufferisé
- BufferedOutputStream => octets bufferisés
-

Classe InputStream

- **InputStream** est la classe d'entrée de base.
- Elle permet de lire un ou plusieurs octets.
 - **int read()** : lit l'octet suivant depuis le flux d'entrée.
 - **int read(byte[] b)** : lit un certain nombre d'octets et les sauvegarde dans *b*.
 - **int read(byte[] b, int off, int len)** : lit jusqu'à *len* octets et les sauvegarde dans le *b*: le premier caractère lu est sauvegardé à la position *off*.
- Quelques sous-classes : FileInputStream, ObjectInputStream, DataInputStream.

Classe InputStream

- Ces méthodes `read(...)` sont bloquantes jusqu'à ce qu'il y ait des données disponibles, on atteint EOF ou il y a une exception.
- Ces fonctions retournent -1 quand on atteint EOF.
- Sinon `read()` retourne l'octet lu (0-255) alors que les deux autres retournent le nombre d'octets lus.

Classe InputStreamReader

- La classe **InputStreamReader** (dérive de Reader) lit des octets “bruts” depuis le stream d'octets sous-jacent et les décode en des caractères selon l'encodage indiqué.
- Pour avoir un objet InputStreamReader, on peut utiliser :
`public InputStreamReader(InputStream in)` ou
`public InputStreamReader(InputStream in, String charsetName)`
- **Méthodes de lecture**
 - `int read()` : retourne un caractère
 - `int read(char[] cbuf, int off, int len)`: un retourne un ensemble de caractères

Classe `BufferedReader`

- La classe **BufferedReader** lit du texte à partir d'un flux de caractères et le met dans un tampon (de caractères) afin d'améliorer les `read()`.
- un `read()` sur un `BufferedReader` se fait depuis le tampon. Lorsque le tampon se vide, il est rempli avec le maximum de données possible.
- Sans la bufferisation, chaque demande de lecture, nécessite de lire des octets depuis le flux d'octets et de les convertir en des caractères.
- Pour avoir un objet `BufferedReader`, on peut utiliser :

`public BufferedReader(Reader in)`

Classe BufferedReader

Méthodes de lecture :

- **int read()** : lit un seul caractère
- **int read(char[] cbuf, int off, int len)** : lit un ensemble de caractères et les range dans une portion d'un tableau: le but ici est de lire un maximum de caractères en répétant des read() jusqu'à ce qu'on :
 - lise le nombre voulu d'octets
 - ou atteigne EOF
 - ou la méthode ready() retourne false (le prochain read() pourrait être bloquant).
- **String readLine()** : lit une ligne (retourne null si on atteint EOF).

Exemple 1

Lecture depuis l'entrée standard qui est représentée par System.in de la classe InputStream

```
BufferedReader in = new BufferedReader(new  
InputStreamReader(System.in));
```

```
System.out.println(in.readLine());
```

Exemple 2

Exemple : *Lecture depuis un fichier*

```
BufferedReader in = new BufferedReader (new  
InputStreamReader(new FileInputStream("fichier-test")));
```

```
String buf;
```

```
while ((buf = in.readLine()) != null)  
    System.out.println(buf);
```

Classe OutputStream

- **OutputStream** est la classe de sortie de base. Elle permet d'écrire un ou plusieurs octets.
 - **void write(int b)** : écrit un octet.
 - **void write(byte[] b)** : écrit le tableau d'octets *b*
 - **void write(byte[] b, int off, int len)** : écrit *len* octets depuis *b* à partir de la position *off*.
- Quelques sous-classes : FileOutputStream, ObjectOutputStream, DataOutputStream.

Classe OutputStreamWriter

- ♦ La classe **OutputStreamWriter** reçoit des **caractères** d'un processus et les traduit en des octets selon l'encodage en question. Elle écrit ces octets sur le stream associé.
- ♦ Exemples de constructeurs :
 - ♦ `public OutputStreamWriter(OutputStream in)`
 - ♦ `public OutputStreamWriter(OutputStream in, charsetName cs)`
- ♦ Méthodes d'écriture :
 - ♦ `void write(int c)`
 - ♦ `void write(char[] cbuf, int off, int len)`
 - ♦ `void write(String str, int off, int len)`

Entrées/Sortie en Java

- A chaque fois que l'on invoque une méthode d'écriture sur un `OutputStreamWriter`, les caractères écrits sont convertis en octets et mis dans un tampon avant d'être écrits sur l'`OutputStream` associé.
- Pour purger ce tampon, on utilise la méthode `void flush()`
- Les caractères passés à `write()` ne sont pas bufferisés.

Classe BufferedWriter

- Quand on écrit sur un **BufferedWriter**, les caractères sont placés dans un tampon et transférés vers le flux de sortie sous-jacent quand il faut.
- Cette classe fournit la méthode `newLine()` qui permet de rajouter le “bon” saut de ligne.
- Les mêmes méthodes d'écriture que la classe `OutputStreamWriter`.
- La méthode `flush()` est utilisée pour vider le buffer.

Exemple

Ecriture sur un fichier

```
BufferedWriter out = new BufferedWriter(new  
OutputStreamWriter(new  
FileOutputStream("mon_fichier")));  
out.write("Ma première ligne");  
out.flush();
```


Bufferisation dans les flux d'octets

- **BufferedInputStream** permet d'améliorer la lecture depuis un `InputStream`.
- Elle possède un tampon (un tableau d'octets *buf*). Une lecture dans ce cas tente d'abord d'avoir des données depuis le tampon. S'il n'y a plus de données, le stream recopie le maximum de données possibles depuis la source
- Constructeurs :
 - `Public BufferedInputStream(InputStream in)`
 - `Public BufferedInputStream(InputStream in, int size)`

Bufferisation dans les flux d'octets

- ♦ BufferedOutputStream stocke les octets reçus dans un tampon (un tableau d'octets) puis les écrit d'un seul coup sur le stream d'octets sous-jacent lorsqu'il faut vider le buffer.
- ♦ Constructeurs :
 - ♦ `Public BufferedOutputStream(OutputStream out)`
 - ♦ `Public BufferedOutputStream(OutputStream out, int size)`
- ♦ Il importe d'appeler `flush()` pour envoyer les données du stream quand il le faut.

Schéma d'un client UDP

- ♦ Le client crée une socket de la classe **DatagramSocket** :
 - ♦ Cette socket est non connectée.
 - ♦ Elle est liée à un point de terminaison (@IP, port) local quelconque défini en général par le système.
- ♦ Le client (connaissant l'@ IP et le port du serveur) échange des données avec le serveur en utilisant les méthodes **send** et **receive** qui prennent comme paramètre un objet de la classe **DatagramPacket**.
- ♦ Le client ferme la socket en utilisant la méthode **close**.

Sockets TCP en Java

Classes Socket & ServerSocket

Classes de sockets en Java

Trois classes sont utilisées :

- Socket : sockets TCP actives
- ServerSocket : sockets TCP passive
- DatagramSocket : sockets UDP.

Client TCP : classe Socket

Shéma général d'un client TCP :

- ♦ `new Socket()`
- ♦ `connect()`
- ♦ `getInputStream()` / `getOutputStream()`
ensuite `read()/write()` selon le protocole applicatif
- ♦ `close()`

Connexion du client au serveur

- Appel explicite à connect()
 - `new socket()`
 - `connect(SocketAddress endpoint)`

On utilise InetSocketAddress qui dérive de SocketAddress

- `new socket(String host, int port)`

Crée une socket et la connecte au point de terminaison spécifié par le couple (host, port).

- `new socket(InetAddress address, int port)`

Crée une socket et la connecte au point de terminaison spécifié par le couple (address, port).

Echange des données

Pour échanger des données, le client (ou le serveur)

- ouvre le flux d'entrée associé à sa socket en utilisant :
`InputStream getInputStream()`
- ouvre le flux de sortie associé à sa socket en utilisant :
`OutputStream getOutputStream()`
- Reçoit les données du serveur en effectuant des lectures sur le flux d'entrée et envoie des données par des écritures sur le flux de sortie.

Déconnexion du serveur

A la fin, le client :

- ferme les flux d'entrée et de sortie
- ferme la socket via la méthode `close()`
- Les flux d'entrée et de sortie associés à une socket doivent être fermés avant de fermer la socket

Exemple d'un client TCP daytime

```
String ligne;  
Socket sockClient = new Socket();  
sockClient.connect(new InetSocketAddress("192.168.1.76", 50013));  
  
BufferedReader br = new BufferedReader(new InputStreamReader(  
    sockClient.getInputStream()));  
  
while ((ligne = br.readLine()) != null)  
    System.out.println(ligne);  
  
br.close();  
sockClient.close();
```

Serveur TCP

Schéma général d'un serveur

- `new ServerSocket()`
- `bind()`
- `accept()`
- `getInputStream()` / `getOutputStream()`
ensuite `read()/write()` (ordre et nombre selon le protocole applicatif)
- `close()`

ServerSocket

En java, la classe `SocketServer` peut être utilisée par le serveur pour écouter les demandes de connexion.

- ♦ `new ServerSocket()` : crée une socket non liée

- ♦ `void bind(SocketAddress endpoint)`

lie la socket au point de terminaison spécifié.

- ♦ Ou bien

- ♦ `new ServerSocket()`

- ♦ `void bind(SocketAddress endpoint, int backlog)`

lie la socket au point de terminaison spécifié, la taille maximale de la file d'attente des clients correspond à `backlog`.

bind() implicite

- `new ServerSocket(int port)`

crée une socket de serveur liée au port spécifié.

- `new ServerSocket(int port, int backlog)`

crée une socket de serveur liée au port spécifié. La taille maximale de la file d'attente des connexions entrantes correspond au paramètre backlog.

- `new ServerSocket(int port, int backlog, InetAddress bindAddr)`

crée une socket de serveur liée au port et à l'adresse IP spécifiés. La taille maximale de la file d'attente des connexions entrantes correspond au paramètre backlog.

ServerSocket

- Dans les constructeurs 1 et 2, l'adresse IP du point de terminaison local de la ServerSocket n'est pas précisée.
- =>L'adresse considérée dans ce cas est l'adresse non spécifiée (contenant uniquement des zéro : 0.0.0.0).
- Cette adresse (appelée parfois appelée “wildcard”) permet au serveur de recevoir des messages sur n'importe quelle interface réseau de la machine sur laquelle il s'exécute (loopback, Ethernet, wifi, etc) et ce en utilisant n'importe quelle adresse de cette interface (dans le cas où l'interface possède plusieurs adresses).

ServerSocket

- **Socket accept()** : écouter et accepter une connexion en créant une nouvelle socket de communication avec le client.

C'est une méthode est bloquante jusqu'à ce qu'il y ait une demande de connexion.

- **void close()** : ferme la socket du serveur.

Exemple d'un serveur daytime TCP

```
Socket client; BufferedWriter out; BufferedReader in;  
ServerSocket serveur = new ServerSocket(port);  
for (int i = 1; i <= nbClients; i++) {  
    client = serveur.accept();  
    out = new BufferedWriter(new OutputStreamWriter(  
        client.getOutputStream()));  
    String chaineDate = new GregorianCalendar().getTime().toString();  
    out.write(chaineDate);  
    out.newLine();  
    out.flush(); // le flush n'est pas indispensable ici  
    client.close();  
}  
serveur.close();
```


Sockets UDP en Java : Classe DatagramSocket

Schéma client/serveur UDP en Java

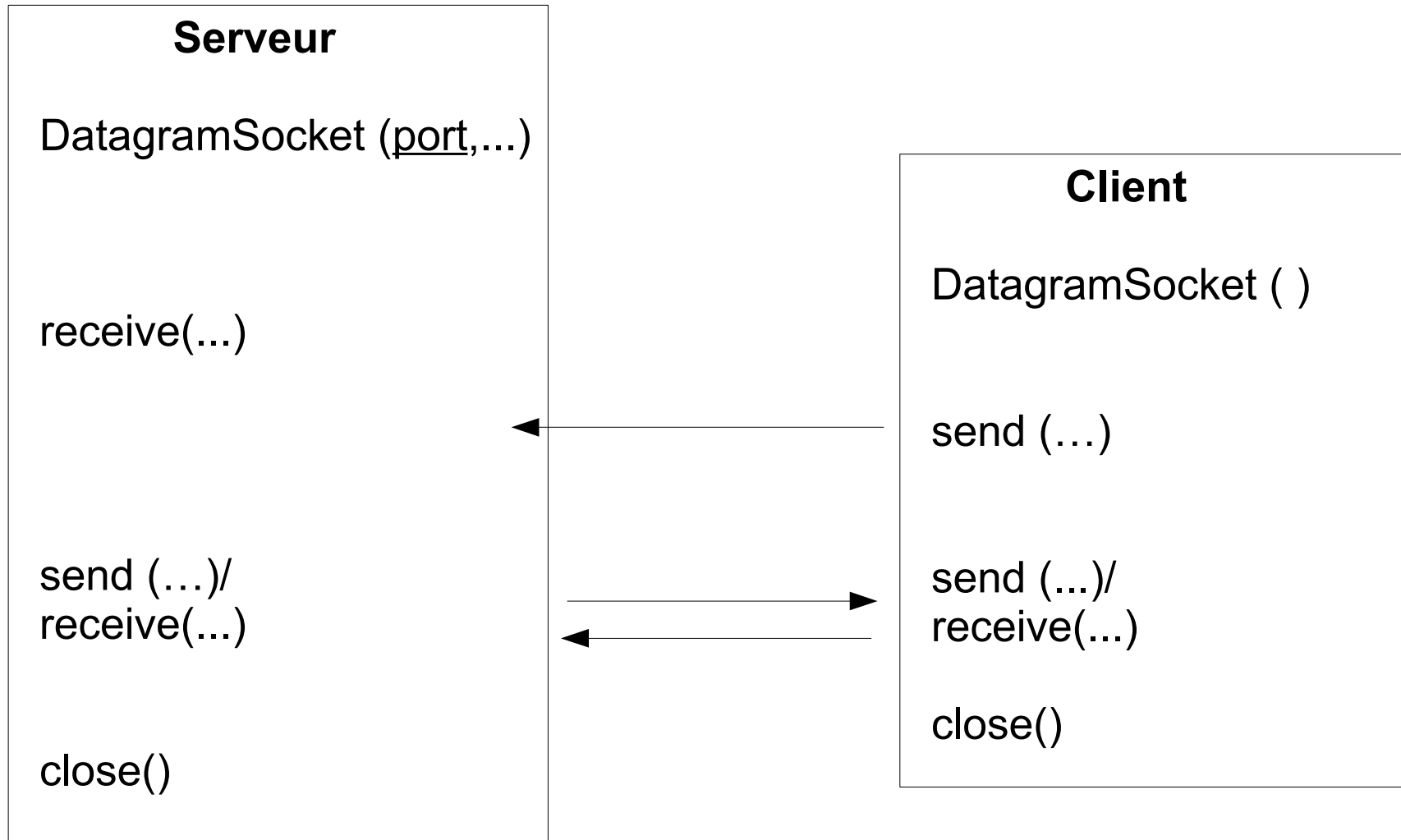


Schéma d'un serveur UDP

- ♦ Le serveur crée une socket d'écoute de la classe **DatagramSocket** :
 - ♦ Cette socket est non connectée.
 - ♦ Elle est liée à point de terminaison (@IP, port) local qui doit être connu par le client.
- ♦ Le serveur échange des données avec le client en utilisant les méthodes **send** et **receive** qui prennent comme paramètre un objet de la classe **DatagramPacket**.
- ♦ Le serveur ferme la socket en utilisant la méthode **close**.

Remarques

- Le client initie la communication avec le serveur par un `send(...)`.
- Le serveur commence par un `receive(...)`.
- => le serveur obtient ainsi l'adresse du client.

Classe DatagramSocket

Constructeurs pour le serveur

Appel explicite à bind()

- ♦ `new DatagramSocket(null)`
- ♦ `public void bind(SocketAddress addr)`

D'autres possibilités :

- ♦ `public DatagramSocket(int port)` // équivalent à ce qui précède.
- ♦ `public DatagramSocket(int port, InetAddress laddr)`
- ♦ `public DatagramSocket(SocketAddress bindaddr)`

Constructeurs pour le client

- ♦ public DatagramSocket()

Ce constructeur sans paramètres crée une datagramme socket liée à n'importe quel port local libre et à l'adresse wildcard.

- ♦ Notez que dans le cas du multicast, le serveur peut utiliser le constructeur du client et vice versa.

InetAddress

- La classe **InetAddress** représente une adresse IP.
- Plusieurs méthodes dans InetAddress nous permettent d'avoir un object InetAddress telles que :
`public static InetAddress getByName(String host)` où host est soit un nom de machine soit une représentation textuelle d'une adresse IP.
- D'autres méthodes:
- `public String getHostAddress() :`
- `public String getHostName()`
- `public boolean isMulticastAddress()`
- etc

Méthodes send et receive

- ♦ **public void send(DatagramPacket p)**

envoie un DatagramPacket depuis la socket en question.
Le DatagramPacket comprend les données à envoyer, leur taille, l'adresse IP et le port de la destination.

- ♦ **public void receive(DatagramPacket p)**

reçoit un DatagramPacket via cette socket.

Le DatagramPacket contient les données reçues, leur taille, l'adresse IP et le port de l'émetteur.

=> méthode bloquante jusqu'à la reception d'un message.

Classe DatagramPacket : constructeurs

- ♦ **public DatagramPacket(byte[] buf, int length)**

En général, crée un DatagramPacket pour recevoir des paquets d'une taille = length avec $\text{length} \leq \text{buf.length}$

Si l'émetteur envoie un message d'une taille plus importante que length, alors ce message sera tronqué.

- ♦ **public DatagramPacket(byte[] buf, int length, InetAddress address, int port)**

Crée, en général, un DatagramPacket pour envoyer des paquets d'une taille = length au point de terminaison distant spécifié.

Classe DatagramPacket : modification du DatagramPacket après création

- **void setData(byte[] buf)**

définit les données du paquet

- **void setLength(int length)**

définit la longueur du paquet

- **void setAddress(InetAddress iaddr)** et **void setPort(int iport)**

spécifient respectivement l'adresse IP et le port de la machine vers laquelle le datagramme sera envoyé.

Classe DatagramPacket : avoir les informations d'un DatagramPacket

- **byte[] getData()**

retourne les données du paquet

- **int getLength()**

retourne la taille des données à envoyer ou la taille des données reçues.

- **InetAddress getAddress()** et **int getPort()**

retournent respectivement l'adresse IP et le port de la machine vers laquelle le datagramme sera envoyé ou depuis laquelle il a été envoyé.

String vers tableau d'octets

- ♦ Pour convertir une chaîne de caractères en un tableau d'octets, on peut utiliser la méthode `byte[] getByte()` de la classe `String`.
- ♦ Pour convertir un tableau d'octets en une chaîne de caractères, on peut utiliser le constructeur
 - ♦ `public String (byte[] tab)`
 - ♦ `public String (byte[] tab, int offset, int length)`

Exemple client UDP daytime

?

Exemple serveur UDP daytime

?

Multicast

Multicast

- ♦ Exemples d'applications:
 - ♦ Jeux en ligne
 - ♦ Visio-conférences
 - ♦ Vidéo streaming, etc.
- ♦ Un même message est envoyé à un groupe de destinataires.
- ♦ Est-ce que l'émetteur envoie le message plusieurs fois (un message par destinataire) ?
- ♦ Réponse : NON
- ♦ L'émetteur envoie le message une seule fois et la couche réseau le propage comme il faut pour atteindre les destinataires.

Multicast

- ♦ Le multicast permet d'envoyer un message à un groupe de destinataires en un seul envoi.
- ♦ Un groupe multicast a une adresse IP de multicast :
 - ♦ En IPv4 : la classe D [224.0.0.0 à 239.255.255.255].
 - ♦ En IPv6 : les adresses qui commencent par ff (en héra).
- ♦ 224.0.0.0 – 224.0.0.255 : adresses de liaison locale réservée (protocoles de routage).
- ♦ 224.0.1.0 à 238.255.255.255 : adresses d'étendue globale.
- ♦ 239.0.0.0/8 : adresses d'étendue administrative.

Multicast

La classe **MulticastSocket** permet de communiquer en mode multicast. Elle dérive de DatagramSocket (le multicast se fait en UDP).

- ♦ Pour joindre un groupe multicast :
 - ♦ On crée une MulticastSocket avec le port désiré via le constructeur : **public MulticastSocket(int port)**
 - ♦ On utilise ensuite la méthode
public void joinGroup(InetAddress mcastaddr)
- ♦ Pour quitter un groupe multicast, on utilise la méthode
public void leaveGroup(InetAddress mcastaddr)

Exemple : serveur

```
String msg = "Coucou";  
DatagramSocket socket = new DatagramSocket();  
InetAddress groupe = InetAddress.getByName("239.0.0.20");  
while (true) {  
    DatagramPacket packet = new DatagramPacket(msg.getBytes(),  
        msg.length(),groupe, 50111);  
    socket.send(packet);  
    Thread.sleep(3000);  
}  
socket.close()
```

Exemple : client

```
MulticastSocket socket = new MulticastSocket(50111);  
  
InetAddress groupe = InetAddress.getByName("239.0.0.20");  
  
socket.joinGroup(groupe);  
  
byte buf[ ] = new byte[256];  
  
DatagramPacket packet = new DatagramPacket(buf, buf.length);  
  
socket.receive(packet);  
  
System.out.println(new String(packet.getData()), 0, packet.getLength());  
  
socket.leaveGroup(groupe);  
socket.close();
```

Exemple : ce qui se passe

- Un serveur qui envoie le message “Coucou” au groupe ayant pour adresse 239.0.0.20 et utilisant le port 50111 chaque 3 secondes.
- Un client qui joint le groupe multicast, attend le message, l'affiche une reçu et quitte le groupe.

Exemple : remarques

- Le client n'a pas besoin d'envoyer un premier message au serveur pour lui indiquer son adresse et son port.
- Le serveur connaît l'adresse de multicast dont fait partie le client et aussi le port associé.
- Le serveur possède un port quelconque, c'est lui qui initie le dialogue.

Threads

- Un programme peut comprendre un ensemble de tâches qui peuvent s'effectuer en parallèle.
- En programmation concurrente, on a deux possibilités :
 - Les processus
 - Les threads
- Les threads (appelés parfois processus légers) requièrent moins de ressources que les processus.
- Un thread est défini dans un processus et partage ses ressources => gérer l'accès simultané aux ressources partagés.

Première façon pour créer un thread en java

- Créer une classe de tâche, on implémente pour cela l'interface `Runnable` et on définit sa méthode `run()` qui décrit la tâche.
- On instancie une tâche `myTask`
- Pour créer un `Thread` pour une tâche on utilise :
`Thread monThread = new Thread(myTask)`
et on appelle la méthode `start()` qui appelle `run()`.

Deuxième façon pour créer un thread en java

- Définir une nouvelle classe qui dérive de Thread.
- Définir la méthode run() qui décrit la tâche effectuée par le thread associé.
- Pour lancer le thread : créer un objet de la nouvelle classe et invoquer la méthode start() sur cet objet (appel implicite à run()).

Serveur multi tâches

...

```
while(estActif) {  
    client = serveur.accept();  
    ThreadServeur thread = new ThreadServeur(client);  
    thread.start();  
}
```

...

```
public ThreadServer extends Thread{  
  
    ...  
  
    public void run() {  
        // Le traitement effectué par ThreadServeur }  
    }
```

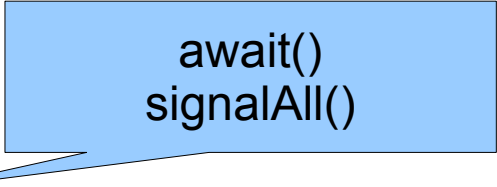
Synchronized pour l'exclusion mutuelle

- Le mécanisme “synchronized” permet la gestion des accès concurrents à des ressources partagées.
- Quand un thread exécute une méthode déclarée “synchronized” d'un objet, tous les autres threads qui invoquent des méthodes synchronized du même objet se bloquent jusqu'à la fin de la première méthode.
- Pour améliorer la concurrence, on peut synchroniser juste une partie d'une méthode en mettant un verou sur un objet particulier avec “synchronized statement” :

```
synchronized(objet){  
    statements;}
```

Lock pour l'exclusion mutuelle

- On peut utiliser des veroux explicites pour l'exclusion mutuelle avec la classe **ReentrantLock** qui admet les méthodes :
 - lock(),
 - unlock et
 - newCondition() : retourne un objet **Condition**
- Ceci permet la coordination entre plusieurs threads.
- Un objet Condition permet par exemple à un thread de se bloquer jusqu'à ce qu'une condition soit vérifiée (changement d'état d'un objet A). D'autres threads (qui travaillent sur la le même objet A) vont envoyer des notif au premier. Ainsi, il peut voir si sa condition est vérifiée.



await()
signalAll()

Pool de threads

- Dans ce qui précède, on crée un nouveau thread par client.
- Cependant, pour un serveur très sollicité cela est coûteux : créer de nouveaux threads implique une charge en plus.
- Alternative : créer un nombre fixe de threads au lancement du serveur et les réutiliser. On parle de **pool de threads**.
- Quand un client arrive, il est traité par un thread du pool, lorsque ce thread finit de traiter le client, il retourne au pool pour attendre d'autres clients.
- Quand un client arrive alors que le pool ne contient pas de thread libre, le client est placé en général en file d'attente.

Pool de threads

Les pools de threads permettent de :

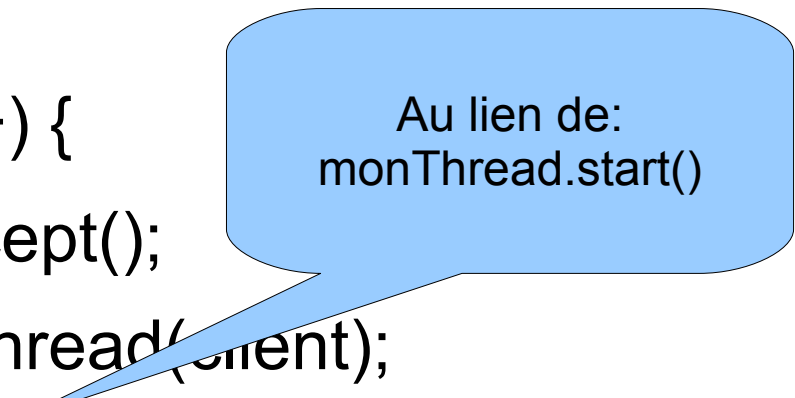
- réduire le temps de création d'un nouveau thread à chaque demande de connexion.
- contrôler et de gérer les ressources système et équilibrer la charge entre plusieurs serveurs.
- améliorer la sécurité (éviter certaines attaques DOS).

Pool de threads en Java

```
ExecutorService pool =
```

```
Executors.newFixedThreadPool(nbThreads);
```

```
for(int i=1; i<= nbClients; i++) {  
    client = serverSocket.accept();  
    monThread = new monThread(client);  
    pool.execute(monThread);  
}
```



Au lieu de:
monThread.start()

Pool de threads en java

- Séparer la création et la gestion de threads de leurs tâches => objets Executor.
- L'interface **Executor** permet d'exécuter des instances Runnable selon une certaine stratégie.
- L'interface **ExecutorService** dérive de Executor et permet par exemple à une tâche d'être arrêtée. Elle permet aussi d'utiliser des instances **Callable** (comme des Runnable mais elle retournent une valeur).
- Les instances de ExecutorService sont générées grâce aux méthodes Fabrique de **Executors**. Par exemple : `newFixedThreadPool()`, `newCachedThreadPool()`, etc.