Polytech Marseille

RAPPELS
Threads, Concurrence et synchronisation, Sockets
(illustration avec Java)

Informatique - Formation par alternance HUGo - 5A

Ce document fait reprend de façon rapide les parties "synthèse du cours" des sujets évoqués en programmation distribuée l'année universitaire précédente.

1 Threads

Une **activité** est une suite d'instructions (l'exécution est ***séquentielle***). Un système est **multitâche** s'il peut faire coexister plusieurs activités au sein d'un même environnement.
Un **processus** est une activité seule au sein d'un environnement restreint (mémoire, ...). Au contraire, un **thread** (aussi appelé "processus léger") partage tout un environnement mémoire avec d'autres threads.

Autrement dit, un **processus** est l'instance d'exécution d'un programme, au sein d'un environnement restreint (mémoire, ...). Un processus peut contenir un ou plusieurs **threads**. Ces threads s'exécutent en quasi-simultanéité au sein d'un même processus (ou simultanément sur un processeur multicœurs [1]). A la différence des processus, les threads partagent le même espace mémoire. La communication inter-threads occasionne peu de surcharge. Le passage contextuel d'un thread à un autre est peu coûteux. Le multitâche de threads surcharge moins que le multitâche de processus, d'où l'intérêt de la programmation dite "multithread".

Java est multithread. La JVM [2] a un **mécanisme interne de gestion des threads** dont l'**ordonnanceur**. Notez que le multitâche de processus est hors contrôle de l'environnement d'exécution Java.

Les **états** d'un thread sont :
- en train de s'exécuter (*running*), il a été choisi ("élu") par l'ordonnanceur,
- prêt à s'exécuter (*runnable*), dès que le CPU est disponible,
- suspendu (*suspended*), ce qui stoppe temporairement son activité,
- poursuivre l'exécution (*resumed*), là où il a été suspendu,
- bloqué (*blocked*) en attendant une ressource.

Un thread a un niveau de **priorité** (nous en reparlerons plus loin avec la méthode `setPriority`). Un thread peut se terminer à tout moment, ce qui arrête son exécution immédiatement. Un thread terminé ne peut pas être remis en route.

Un thread peut :
- soit volontairement laisser le contrôle. Ceci est réalisé en passant le contrôle explicitement, dormant ou bloquant en E/S [3]. Le thread prêt à s'exécuter et de plus haute priorité est alors élu.
- soit être supplanté par une autre de plus haute priorité. Si c'est systématiquement le cas, on parle de multitâche préemptif.

1. Un processeur ayant 4 cœurs (ou "core") exécute jusqu'à 4 threads en même temps.
2. Machine virtuelle Java
3. Entrée/Sortie

L'**ordonnanceur de la JVM** n'est pas tenu d'être équitable. Les threads de basses priorités ne sont exécutés que si les threads de hautes priorités sont bloqués.

L'existence de plusieurs threads au sein d'un même environnement induit nécessairement des **problèmes** de :
- Ordonnancement car les exécutions sont ***non-déterminismes***.
- Partage des données, il faut alors ***synchroniser des actions*** sur ces données partagées. Nous reverrons cela dans la partie du cours traitant de la concurrence.

Parmi les **threads importants** de Java, il y a : le thread ***Main***, le gestionnaire d'horloge (***Clock handler***), le thread oisif (***Idle thread***), le ramasse-miettes (***Garbage collector***), le thread de finalisation (***Finalizer thread***), les threads liés aux composants graphiques et aux événements du système de fenêtrage.

Nous allons maintenant aborder la **création de threads en Java**. Il existe deux possibilités de créer un thread :
- par ***héritage*** de la classe `Thread`
- par ***implémentation*** de l'interface `Runnable`.

Il existe deux manières de faire car Java n'autorise pas l'héritage multiple. Une classe ne peut être l'extension que d'une seule classe. Par contre, elle peut implémenter plusieurs interfaces.

Exemple **par héritage** de la classe `Thread` :

```
class UnThread extends Thread {
        ...

        public UnThread() { /* constructeur */
        }

        public void run() {
                /* activité du thread */
                /* instructions exécutées par le thread une fois démarré */
                ...
        }
}
```

On crée et on démarre le thread de cette façon :

```
...
UnThread thr = new UnThread(); // création du Thread
thr.start(); // démarrage du thread (de l'activité)
...
```

Exemple **par implémentation** de l'interface `Runnable` :

```
class UnRunnable implements Runnable {
        ...

        public UnRunnable() { /* constructeur */
        }

        public void run() {
                /* activité de la tâche */
                /* instructions exécutées une fois la tâche démarrée */
                ...
        }
}
```

On crée et on démarre le thread de cette façon :

```
...
UnRunnable unRunnable = new UnRunnable(); // création de la tâche
Thread thr = new Thread(unRunnable); // création du Thread effectuant la tâche
thr.start(); // démarrage du thread (de l'activité)
...
```

Notez qu'il ne faut pas confondre l'activité (le *thread*) avec la structure de données (le `Thread` qui est la classe JAVA).

Voici quelques **méthodes de la classe** `Thread` :
Informations sur les threads :

- `string` **`getName`**`()` : retourne le nom du thread.
- `void` **`setName`**`(String name)` : donne le nom `name` au thread. Lance une "SecurityException".
- `static int activeCount()` : renvoie le nombre de threads actuellement exécutés.
- `static int enumerate(Thread[] tarray)` : stocke l'ensemble des Threads du même groupe dans le tableau et renvoie le nombre de Threads. Lève une "SecurityException".
- `static Thread` **`currentThread`**`()` : renvoie le Thread correspondant au thread en train d'être exécuté.

Méthodes liées à l'ordonnancement :

- `void setPriority(int)` : fixe la priorité du thread. Cette priorité est entre MIN_PRIORITY et MAX_PRIORITY (qui sont souvent 1 et 10). La priorité par défaut est NORM_PRIORITY (généralement 5). Lève une "IllegalArgumentException" et une "SecurityException".
- `int getPriority()` : renvoie la priorité du thread.
- `static void yield()` : provoque une "pause" du thread en cours et un réordonnancement. Attention l'ordonnanceur de la JVM peut ne pas tenir compte des priorités.

Manipulation de threads :

- `static void` **`sleep`**`(long millis)` et `static void sleep(long millis, int nanos)` : provoque la mise en attente du thread. Lève une "InterruptedException".
- `void` **`join`**`()` ou `void join(long millis)` ou `void join(long millis, int nanos)` : provoque l'attente de l'arrêt du thread (sans ou avec délai). Lève une "InterruptedException".

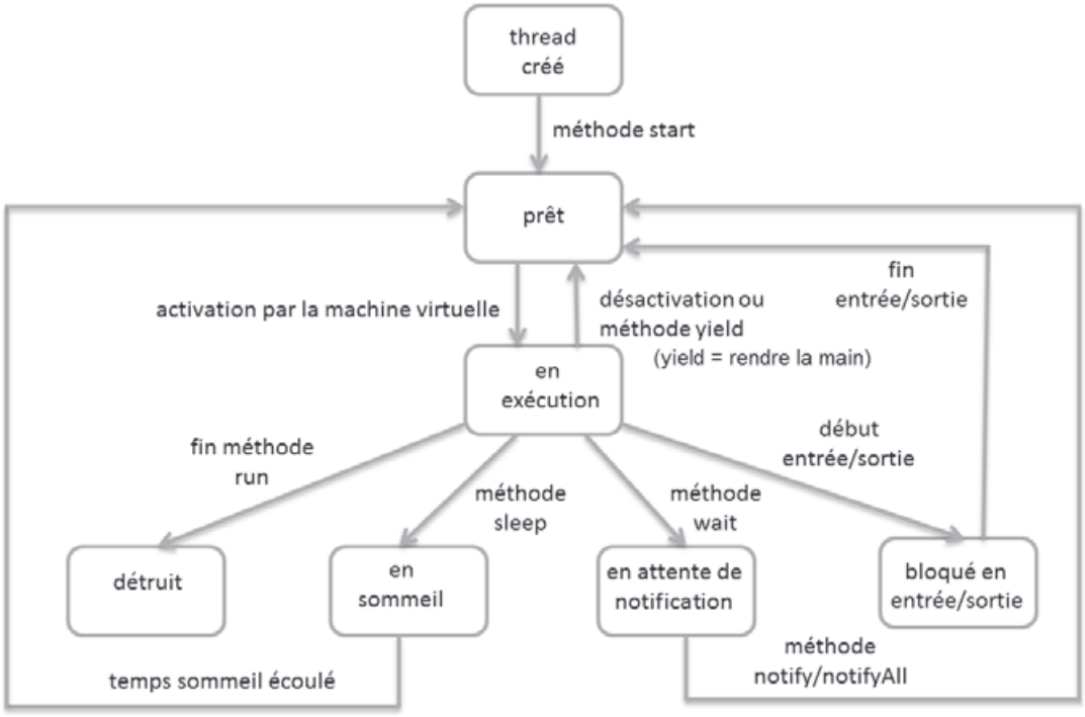
Plusieurs de ces méthodes sont associées aux états possibles d'un thread dans la figure présentée plus loin.

Notez bien qu'à la fin de la méthode `run()`, le thread termine et meurt.

Enfin, la programmation multithread est délicate car :
- les exécutions sont non-déterminisme (l'entrelacement des instructions des threads est différent d'une exécution à l'autre),
- certaines erreurs n'apparaissent que sous hautes charges (i.e., en production),
- les optimisations impliquent des comportements parfois non intuitifs,
- il existe des variations dans l'implémentation des machines virtuelles (notamment au niveau de l'ordonnanceur de la JVM,
- certaines ressources peuvent être partagées entre des threads (besoin de mécanismes de synchronisation).

À ne pas oublier

→ démarrage d'un thread avec `start()` (et non avec `run()`)
→ activité du thread : instructions écrites dans la méthode `run()`
→ nommage d'un thread (`setName()` et `getName()`)
→ récupération du thread courant (`Thread.currentThread()`)
→ attente de la fin d'exécution d'un thread (`join()`)
→ à la fin du `run()`, le thread s'arrête / meurt.
→ états d'un thread :



2 Concurrence et synchronisation

L'utilisation de plusieurs **threads** (la programmation multithread) peut être source de problèmes notamment si les threads partagent une variable. Si l'accès à cette variable est seulement en lecture (i.e., accès en "lecture seule") alors il n'y a pas de problème particulier. Les threads ne font que récupérer la valeur de la variable. Par contre, si un ou plusieurs threads modifient la valeur de la variable (i.e., accès en écriture) alors il y aura des problèmes de **concurrence** d'accès. Le concepteur/développeur (vous) doit gérer cela et mettre en place une solution de **synchronisation** des accès.

Étant donnée du code, on peut y définir des portions particulièrement importantes (comme la manipulation d'une variable partagée) que l'on dénote par **sections critiques**.

```
// Du code
...
/* Section Critique */
... // utilisation et modification d'une ou plusieurs variables partagées
/* Fin de la Section Critique */
// Encore du Code
...
```

Plusieurs sections critiques dépendantes ne doivent jamais exécuter leur code simultanément (par plusieurs threads différents) : on dit qu'elles doivent être en **exclusion mutuelle**.

Pour mettre en place l'exclusion mutuelle, il faut utiliser des verrous. Lorsqu'un thread entre dans une section critique, il demande le verrou. S'il l'obtient, il peut alors exécuter le code. S'il ne l'obtient pas, parce qu'un autre thread l'a déjà pris, il est alors bloqué en attendant de l'obtenir.

Une section critique est vue comme une **opération atomique** (une seule opération indivisible) par une autre section critique utilisant le même verrou.

2.1 Exclusion mutuelle : Sémaphores

Depuis Java 1.5, il existe la classe `Semaphore` présente dans le package `java.util.concurrent`.

Rappelons qu'un sémaphore encapsule un entier (avec une contrainte de positivité) et deux opérations atomiques d'incrémentation et de décrémentation :
- variable entière (toujours positive ou nulle).
- opération P (en Java, méthode `acquire()`) : décrémente le compteur s'il est strictement positif ; bloque le thread appelant P s'il est nul en attendant de pouvoir le décrémenter.
- opération V (en Java, méthode `release()`) : incrémente le compteur.

La valeur de l'entier correspond au nombre d'accès autorisés.

Exemple :

```
Semaphore sem = new Semaphore(1);
// sémaphore initialisé à 1 : un seul thread à la fois en section critique
 try {
     sem.acquire(); // demande un accès (reste bloqué ici
                    // si pas aucun accès disponible)
     //section critique
     sem.release(); // rend un accès
 }
 catch (InterruptedException e) {
     e.printStackTrace();
 }
```

2.2 Exclusion mutuelle : Moniteurs

Un **moniteur** est un mécanisme utilisé comme un verrou d'exclusion mutuelle. Lorsqu'un thread acquiert un verrou, on dit qu'il entre dans le moniteur. Les autres threads voulant le même verrou sont dits en attente du moniteur. On utilise les moniteurs via les **blocs synchronisés** (mot-clé **synchronized**).

Un moniteur peut être vu comme ***une boîte ne pouvant contenir qu'un seul thread***. Une fois qu'un thread y est entré, les autres doivent attendre qu'il en sorte. Il n'y a pas de classe "Monitor". **Chaque objet a son propre moniteur implicite** dans lequel un thread entre automatiquement lorsqu'il exécute un bloc synchronisé.

```
synchronized(unObjet) {
     //section critique (accès à un seul thread à la fois)
}
```

`unObjet` représente un verrou, qui peut être un objet Java quelconque. Attention cependant, il vaut mieux utiliser `final`, pour être sûr que la référence vers l'objet n'est pas modifiée (et représente donc le même verrou).

Cas particulier : lorsque l'objet servant de verrou pour la synchronisation est `this` et qu'il englobe tout le code d'une méthode, on peut mettre le mot-clé `synchronized` dans la signature de la méthode.

Les deux morceaux de code ci-dessous pour `methode()` sont strictement équivalents :

```
synchronized void methode() {
   //section critique (accès à un seul thread à la fois)
}
```

est équivalent à

```
void methode() { // aucune instruction avant "synchronized"
  synchronized(this) {
      //section critique (accès à un seul thread à la fois)
  } // aucune instruction après "synchronized"
}
```

Un moniteur Java est **réentrant**, c'est-à-dire qu'un thread peut réacquérir un verrou qu'il possède déjà. En pratique, cela signifie qu'une méthode avec le mot-clé `synchronized` peut appeler une autre méthode du même objet qui possède aussi le mot-clé `synchronized` sans créer un interblocage.

2.3 Synchronisation coopérative : Moniteurs

Il peut être nécessaire d'effectuer une **synchronisation coopérative** entre threads.

Un mécanisme de **communication inter-threads** via les moniteurs permet d'éviter une scrutation entre les threads. Il n'y a donc pas de perte des cycles CPU à attendre.

Ce mécanisme est accessible à travers 3 méthodes :
- `wait()` : fait sortir le thread appelant du moniteur et le met en sommeil jusqu'à ce qu'un autre thread entre dans le même moniteur et appelle `notify()`
- `notify()` : réveille le premier thread ayant appelé `wait()` sur le même objet
- `notifyAll()` : réveille tous les threads ayant appelé `wait()` sur le même objet ; le thread de priorité la plus élevée s'exécutera en premier.

Il existe également la forme `final void wait(long timeout)` qui permet d'attendre un nombre donné de millisecondes.

Ces méthodes sont implantées dans `Object` de sorte que **toutes les classes en disposent**.

Attention : elles **ne peuvent être appelées qu'à l'intérieur d'un bloc synchronisé (ou d'une méthode synchronisée)**.

Remarque : au réveil (par "notify()" ou "notifyAll()"), l'ordre FIFO n'est pas garanti. On ne sait pas quel thread "endormi" (i.e., présent dans la file d'attente du moniteur) va être réveillé.

3 Sockets

3.1 Modèle client/serveur

Dans le modèle client/serveur, un client (le programme client) demande l'exécution d'un service à un serveur (le programme serveur), il envoie alors une requête. Le serveur réalise le service (par exemple un certain calcul) puis envoie la réponse au client. Le client et le serveur sont en général localisés sur des machines différentes.

Il ne faut pas mélanger la notion de programme serveur avec la machine serveur (ordinateur) car une machine peut exécuter plusieurs programmes serveurs en même temps. Ces derniers écoutent chacun un port différent. Nous pouvons faire la même remarque pour la notion de programme client et de machine client. Pour nous, le terme "serveur" et "client" désigneront respectivement un programme serveur et un programme client.

Notons qu'il y a une indépendance interface-réalisation. Le client sait comment utiliser le service mais il ne sait pas comment le service est implémenté. Le serveur peut changer l'implémentation du service sans affecter l'interface, autrement dit, sans changer la façon d'utiliser ce service.

Les notions de serveur et de client sont relatives car un serveur peut faire appel à un autre serveur (il est alors le client). C'est le cas dans les architectures 3-tiers où, par exemple, un serveur web fait appel à un serveur de bases de données. Dans les réseaux P2P ("peer-to-peer"), chaque nœud du réseau est à la fois client et serveur.

3.2 Mise en œuvre

Nous pouvons mettre en œuvre le modèle client/serveur :
- soit par des opérations de bas niveau avec les primitives du système de communication du système d'exploitation (OS [4]) : utilisation de **sockets**,
- soit par des opérations de haut niveau d'abstraction en utilisant un **intergiciel** (en anglais, Middleware) comme Java RMI et CORBA. Ces derniers s'appuient sur les primitives de l'OS.

4. Operating System

3.3 Généralités sur les sockets

Une socket (que nous pouvons traduire par une "prise") est un point de terminaison dans une communication bidirectionnelle entre deux programmes fonctionnant en réseau. Cela offre un mécanisme de communication (de bas niveau) permettant d'utiliser les protocoles de transport TCP [5] et UDP [6].

Le concept de socket a été introduit dans Unix dans les années 80 par Bill Joy (Projet "Berkeley Software Distribution", BSD). C'est un standard aujourd'hui (Linux, Windows, ...). De plus, les sockets sont utilisables depuis de très nombreux langages de programmation (C, Java, Python, ...).

Il y a deux réalisations possibles : le mode connecté ou le mode non connecté.

Avec le mode connecté (protocole **TCP**), il y a une ouverture d'une liaison, une suite d'échanges, puis une fermeture de la liaison. Le serveur préserve son état entre deux requêtes (il conserve des informations sur les requêtes des clients). Les garanties de TCP sont l'ordre, le contrôle de flux et la fiabilité. Ce mode est adapté aux échanges ayant une certaine durée (plusieurs messages).
$\rightarrow$ analogie avec le téléphone (communication vocale).

Avec le mode non connecté (protocole **UDP**), les requêtes successives sont indépendantes. Il n'y a pas de préservation de l'état entre les requêtes. Le client doit indiquer son adresse à chaque requête (il n'y a pas de liaison permanente). Il n'y a pas de garanties particulières (perte de messages, désordre, ...). Ce mode est adapté aux échanges brefs (réponse en un message).
$\rightarrow$ analogie avec le courrier postale ou les sms.

Les points communs entre les deux modes sont :
- Le client est à l'initiative de la communication.
- Le serveur doit être à l'écoute.
- Le client doit connaître les informations du serveur : (**adresse IP, numéro de port**).

Sur une machine, le numéro de port (ou plus simplement "port") permet de distinguer un programme fonctionnant en réseau d'un autre. Il y a généralement plusieurs serveurs en cours d'exécution sur une machine, chacun utilisant un port différent.

Certains numéros de port sont réservés et ne peuvent pas être utilisés, par exemple, 7 pour ICMP, 21 pour FTP, 22 pour SSH, 23 pour Telnet, 25 pour SMTP, 53 pour DNS, 80 pour HTTP, 110 pour POP, 123 pour NTP, 143 pour IMAP, 161 pour SNMP. **N'utilisez pas un port inférieur à 1024**. Il est aussi utile de connaitre les ports utilisés par défaut par les serveurs les plus courants : 3306 pour MySQL, 1521 pour Oracle, 5432 pour PostgreSQL, 8080 pour Apache Tomcat, 27017 pour MongoDB, ...

Les modes de gestion de requêtes d'un serveur sont :
- Mode **séquentiel** ou **itératif** : un seul client servi à la fois,
- Mode **parallèle** ou **concurrent** ou "**multi-threadé**" : plusieurs clients sont servis en même temps.

Les modes d'échange de données entre clients et serveurs sont :
- Mode **ligne** (chaînes de caractères),
- Mode **bloc** d'octets (bytes).

Il est important de noter que lors de la conception d'une application avec les sockets, il est très important de définir le **protocole applicatif** (dialogue entre client et serveur, ***format des messages échangés***). Par exemple, pour une application de gestion distante de comptes bancaires, un client envoie "`Toto;titi;3;100;`" au serveur pour demander une opération de dépôt (ayant le code 3) de 100 euros sur le compte de "Toto" et le mot de passe est "titi". Le serveur répond "`Toto:*:0:250;`" où le code retour "0" indique que l'opération s'est bien passée et le nouveau solde du compte s'élève

5. Transmission Control Protocol
6. User Datagram Protocol

à 250 euros. Le format des messages a donc été défini comme suit. Les requêtes sont de la forme "`nomCompte;motDePasse;codeOperation;montant;`" et les réponses correspondent au format "`nomCompte:*:codeRetour:nouveauSolde;`". Le protocole est simple : le client se connecte au serveur, puis répète envoi d'une demande d'opération et réception de la réponse, et enfin se déconnecte du serveur.

Il faut faire attention aux différentes architectures possibles entre les machines qui dialoguent (problèmes d'"endianisme", etc.), elles pourraient ne pas se comprendre.

Enfin, il faut prendre en compte les éventuels problèmes de perte de messages, de déséquencement des messages (désordre à la réception), . . .si on utilise le protocole UDP.

Les classes importantes pour la réalisation, en Java, d'une application client-serveur en mode connecté (TCP) sont : ***Socket*** *(utilisée par le client et le serveur)* et ***ServerSocket*** *(utilisée par le serveur).*

Les classes importantes pour la réalisation, en Java, d'une application client-serveur en mode non-connecté (UDP) sont : ***DatagramSocket*** et ***DatagramPacket***.

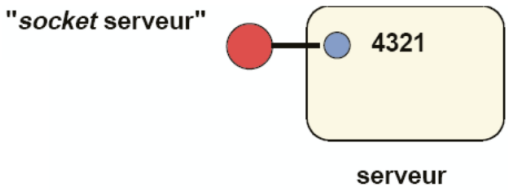
La classe Java représentant une adresse IP est *InetAddress*. On récupère l'adresse IP d'une machine à partir de son nom avec la méthode ***getByName(String hostname)***.

3.4 Sockets TCP (mode connecté)

En mode connecté (TCP), nous avons le déroulement suivant.

1) Le serveur crée une "socket serveur" (associée à un port, par exemple le 4321) et se met en attente de connexion d'un client (bloquant tant que pas de client).

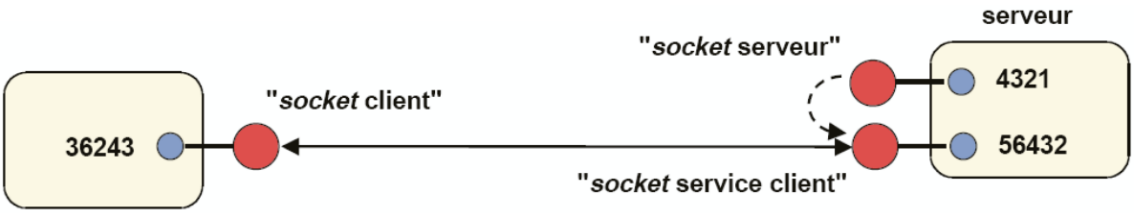
```
ServerSocket sock = new ServerSocket(4321);
Socket sockservice = sock.accept();
```



2) Le client se connecte à la "socket serveur". Par cela, il doit connaître l'adresse IP (supposons que c'est 139.168.1.1) et le port correspondant au service désiré (ici, 4321).

```
Socket s = new Socket("139.168.1.1", 4321);
```

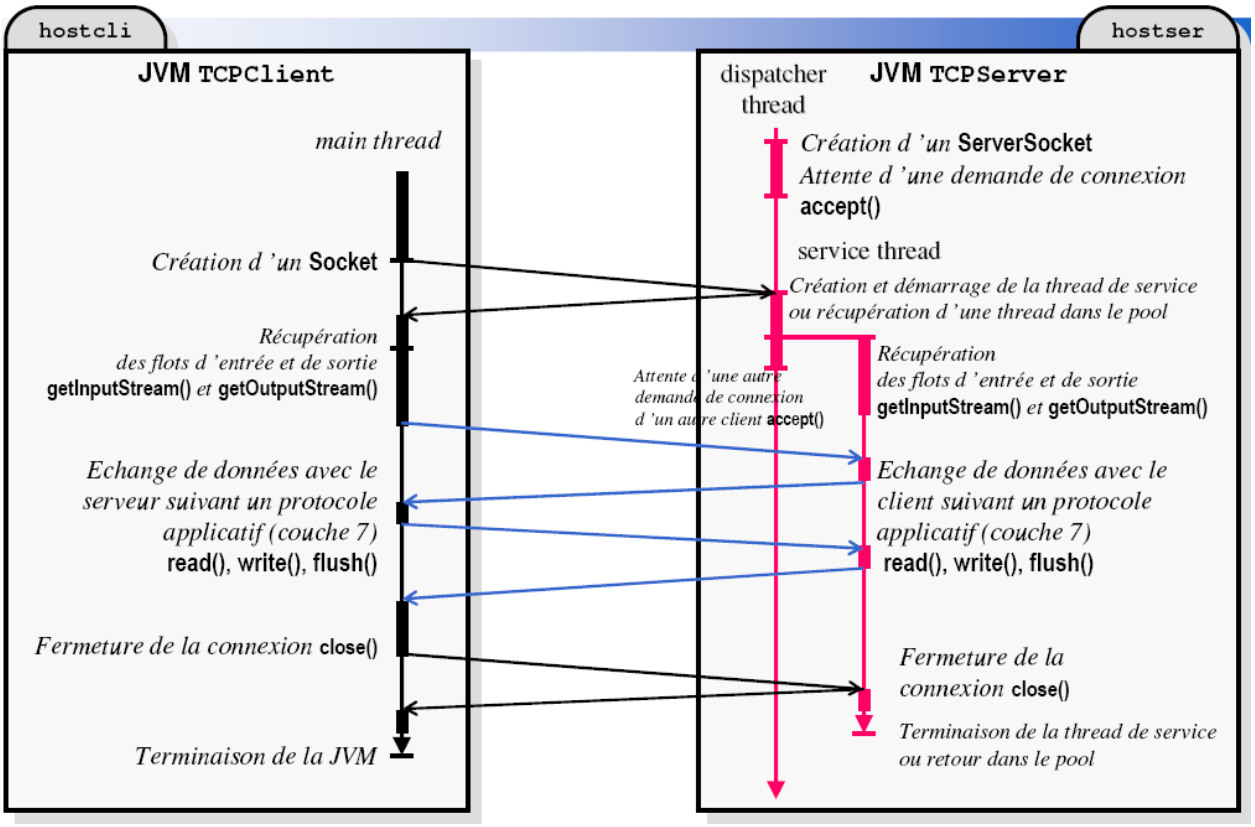
Deux sockets sont alors créées : une "socket client" côté client (associée par exemple au port 36243) et une "socket service client" côté serveur résultant du `accept()` (associée par exemple au port 56432). Ces sockets sont connectées entre elles.



3) Le client et le serveur communiquent alors via les deux sockets : l'interface est celle des fichiers ("read()", "write()"). La "socket serveur" peut accepter de nouvelles connexions (car la socket associée au port 4321 est de nouveau libre).

Le client récupère le flot d'entrée et celui de sortie de la socket (`s.getInputStream()` et `s.getOutputStream()`) et crée des flux de plus haut niveau pour faciliter leur utilisation (par exemple `BufferedReader` et `PrintWriter`;
`in = new BufferedReader(new InputStreamReader(s.getInputStream()));`
`out = new PrintStream(s.getOutputStream())`).
Pour recevoir un message, le client lit sur le flot d'entrée de la socket (`in.readLine()`).
Pour envoyer un message, le client écrit sur le flot de sortie de la socket (`out.println(...)`).
A la fin, une fois tous les échanges terminés, le client ferme la socket (`s.close()`).
Remarque : la lecture sur la socket (par exemple avec `readLine()`) est bloquante tant que rien n'a été reçu. Le fonctionnement est similaire du côté serveur (du thread de service si le serveur est multi-threadé).

Schéma récapitulatif



3.5 Sockets UDP (mode non connecté)

En mode non-connecté (protocole UDP), on ne parle pas vraiment de client et de serveur mais plus d'émetteur et de récepteur. L'émetteur envoie un message (un datagramme) au récepteur.

Voici le déroulement général.

Côté récepteur

Le récepteur crée un DatagramSocket en indiquant le port d'écoute :
`DatagramSocket ds = new DatagramSocket(4321);`
Il construit un DatagramPacket en indiquant la taille max d'un message :
`DatagramPacket paquetRecu = new DatagramPacket(...);`
Il se met ensuite en attente de réception d'un datagramme :
`ds.receive(paquetRecu);`
La méthode `receive` est bloquante tant qu'un datagramme n'a pas été reçu.

Côté émetteur

L'émetteur crée un DatagramSocket :

```
DatagramSocket ds2 = new DatagramSocket();
```

Il crée ensuite un DatagramPacket en indiquant le message, la taille de ce message, l'adresse IP du récepteur et le numéro de port du récepteur :

```
DatagramPacket paquetAenvoyer = new DatagramPacket(...);
```

Enfin, il envoie le datagramme :

```
ds2.send(paquetAenvoyer);
```

Remarques

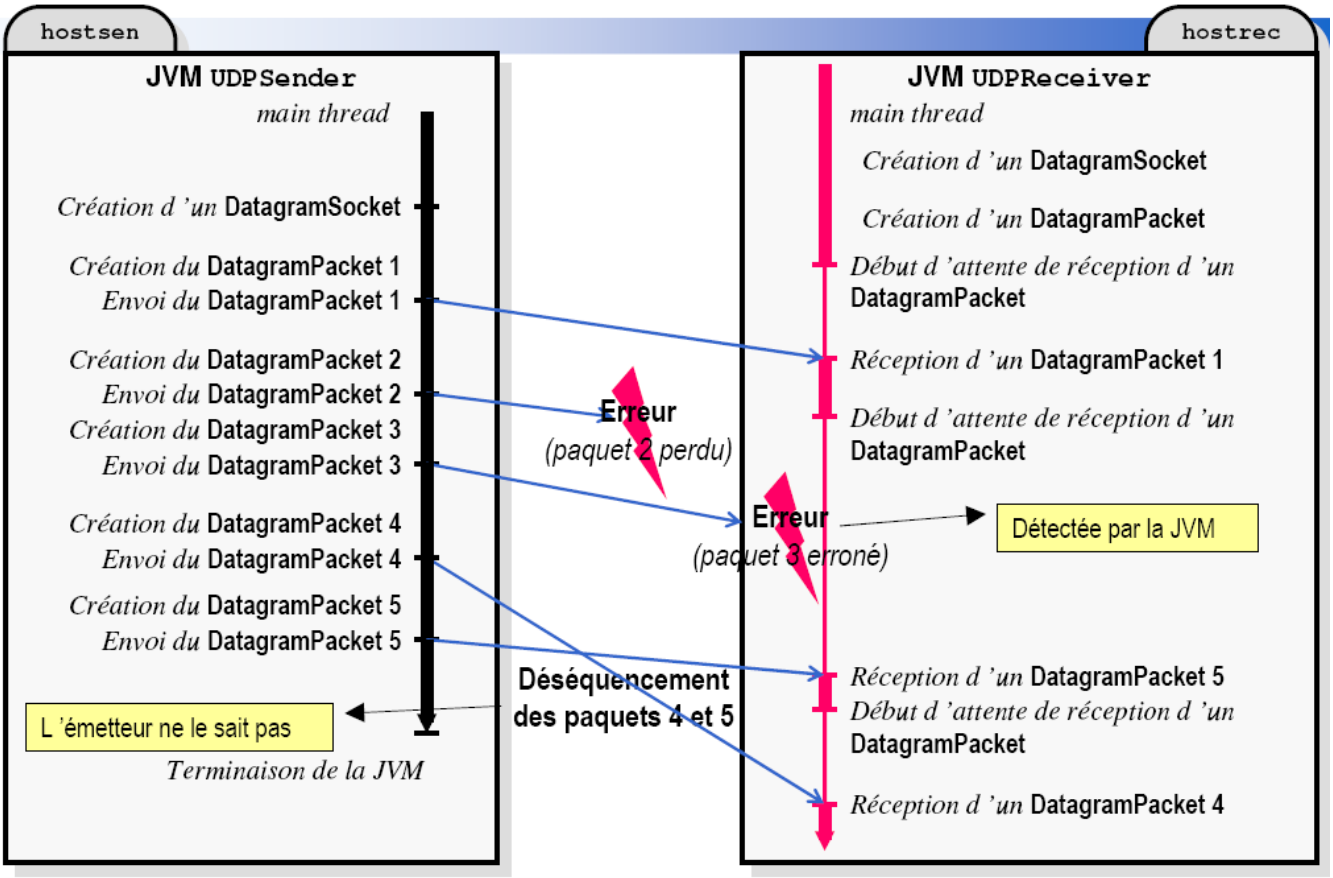
Le récepteur peut récupérer l'adresse IP et le port de l'émetteur car ils sont indiqués dans le datagramme reçu (`getAdress()` et `getPort()`). Il peut donc répondre si besoin.

Si le récepteur répond alors les rôles s'inverse et le fonctionnement reste identique.

Attention, ici les messages peuvent être perdus (sans être averti). De plus, les messages peuvent être reçu dans le désordre au niveau du récepteur (2 messages envoyés par un même émetteur peuvent arriver dans un désordre sur un même récepteur).

Ces problèmes sont à la charge du développeur (mécanismes de "timeout", d'acquittement, de numérotation dans les messages, ...).

Schéma récapitulatif



3.6 Sérialisation

C'est un procédé introduit dans le JDK version 1.1 qui permet de rendre un objet persistant. Cet objet est mis sous une forme sous laquelle il pourra être reconstitué à l'identique (ce sont les valeurs de ses attributs qui sont sauvegardés). Ainsi il pourra être stocké sur un disque dur ou transmis au travers d'un réseau pour le créer dans une autre JVM.

Au travers de ce mécanisme, Java fourni une façon facile, transparente et standard de réaliser cette opération. Ceci permet de facilement mettre en place un mécanisme de persistance. Il est de ce fait inutile de créer un format particulier pour sauvegarder et relire un objet.
Le format utilisé est indépendant du système d'exploitation. Ainsi, un objet sérialisé sur un système peut être réutilisé par un autre système pour récréer l'objet.

L'ajout d'un attribut à l'objet est automatiquement pris en compte lors de la sérialisation.

Attention : toutefois, la dé-sérialisation de l'objet doit se faire avec la même (version de la) classe qui a été utilisée pour la sérialisation. La sérialisation peut s'appliquer facilement à tous les objets.

Envoi d'un objet sérialisé

Étant donné que la sérialisation permet de sauvegarder l'état interne d'un objet (i.e. ses attributs) puis de le récupérer dans un flux, on peut donc envoyer les informations d'un objet par réseau.

Tout d'abord, l'objet que l'on souhaite sérialiser doit implémenter l'interface `java.io.Serializable` (ou du moins, une de ses classes mères doit l'implémenter). Cette interface ne définit aucune méthode mais permet simplement de marquer une classe comme pouvant être sérialisée.

Si l'on tente de sérialiser un objet qui n'implémente pas l'interface `Serializable`, une exception `NotSerializableException` est levée.

Nous supposons que nous avons instancié une socket puis avons déclaré un `BufferedOutputStream` récupérant le flux de la socket en sortie. La sérialisation d'un objet est effectué lors de l'appel de la méthode `writeObject()` sur un objet implémentant `ObjectOutput` (par exemple `ObjectOutputStream`) en passant en paramètre l'objet à sérialiser. Il faut passer en paramètre du constructeur d'`ObjectOutputStream` le flux sur lequel on envoie l'objet sérialisé.

Réception d'un objet sérialisé

La réception d'un objet sérialisé fonctionne quasiment de la même façon que pour l'envoi. On a une socket et un `BufferedInputStream` qui prend le flux entrant de la socket.

La dé-sérialisation d'un objet est effectuée en appelant la méthode `readObject()` sur un objet implémentant `ObjectInput` (comme par exemple `ObjectInputStream`).

Mot-clé `Transcient`

Le contenu des attributs est visible dans le flux dans lequel est sérialisé l'objet. Il est ainsi possible pour toute personne ayant accès au flux de voir le contenu de chaque attribut même si ceux si sont `private`. Ceci peut poser des problèmes de sécurité surtout si les données sont sensibles.

Java introduit le mot clé `transient` qui précise que l'attribut qu'il qualifie ne doit pas être inclus dans un processus de sérialisation et donc de dé-sérialisation.

Exemple :

```
private transient String codeSecret;
```

Lors de la dé-serialisation, les champs `transient` sont initialisés avec la valeur `null`. Ceci peut poser des problèmes à l'objet qui doit gérer cette état pour éviter d'avoir des exceptions de type `NullPointerException`.