Programmation Réseau

Java NIO

E/S non-bloquantes

Jean-Baptiste.Yunes@univ-paris-diderot.fr

UFR Informatique

2012-2013

Entrées/Sorties non-bloquantes

- On considère en premier chef que les opérations d'entrées/sorties sont **bloquantes** :
  - tenter une telle opération met le demandeur en attente jusqu'à ce que l'opération puisse être réalisée (ou se révèle impossible)
- la logique est assez facile à gérer
  - le synchronisme est fort

Le relevé de compteur ou la livraison de colis

- vous savez que le préposé doit passer le 12
  - le 12 au matin (précautionneux vous vous levez à 6:00), puis...
  - vous attendez
  - jusqu’à ce qu’il vienne...
  - s’il ne vient pas, votre journée est gâchée, votre vie est fichue
- dans la vie on finit par abandonner

Le relevé de compteur ou la livraison de colis

- vous savez que le préposé doit passer le 12
  - le 12 au matin (précautionneux vous vous levez à 6:00), puis...
  - vous attendez
  - jusqu'à ce qu'il vienne...
  - s'il ne vient pas, votre journée est gâchée, votre vie est fichue
- dans la vie on finit par abandonner

Entrées/Sorties non-bloquantes

- Une **entrée/sortie non-bloquante** correspond à la situation dans laquelle on souhaite **ne pas attendre** si l'entrée/sortie ne peut être réalisée
  - pour faire autre chose, quitte à revenir tenter plus tard

Le boulanger ou le boucher

- vous voulez du pain ou du steack haché
  - vous allez chez le boulanger ou le boucher, mais il n'y en a pas ou plus
  - vous **n'attendez pas** car on ne peut vous dire quand le boulanger ou le boucher se décidera à en refaire, et vous avez autre chose à faire...
  - alors vous partez faire cette autre chose
  - et puis vous revenez plus tard...et recommencez
- cette fois votre **opération** n'est **pas bloquante** : vous n'êtes simplement pas servi et on vous laisse repartir

Asynchronous vs Non-blocking

- Il faut normalement distinguer
  - la requête
  - la réalisation de la requête
- l’asynchronisme autorise à ne pas être bloqué pendant la réalisation
- la non-réalisation non-bloquante qui autorise à ne pas être bloqué en attente de la possible réalisation

Le remplissage de la baignoire

- vous ouvrez le robinet
  - il y a de l'eau, vous laissez couler
    - en regardant bêtement jusqu'à ce que ça finisse (en sirotant un délicieux cocktail)

bloquant synchrone

    - vous laissez couler mais vous partez immédiatement faire le repassage en attendant

asynchrone non-bloquant

  - il n'y a pas d'eau
    - vous attendez qu'elle revienne

bloquant

    - vous passez à autre chose (en refermant)

non-bloquant

Non Blocking IOs : un problème informatique

- serveur à double entrée… ???
  - virtual hosting par exemple
    - selon l'adresse internet à travers laquelle vous vous adressez à lui, le serveur répond différemment
  - ainsi vous pouvez avoir sur une **même machine**, un seul serveur web qui rend des choses générales depuis le réseau externe (interface externe) et des choses privées depuis l'intérieur (interface interne)

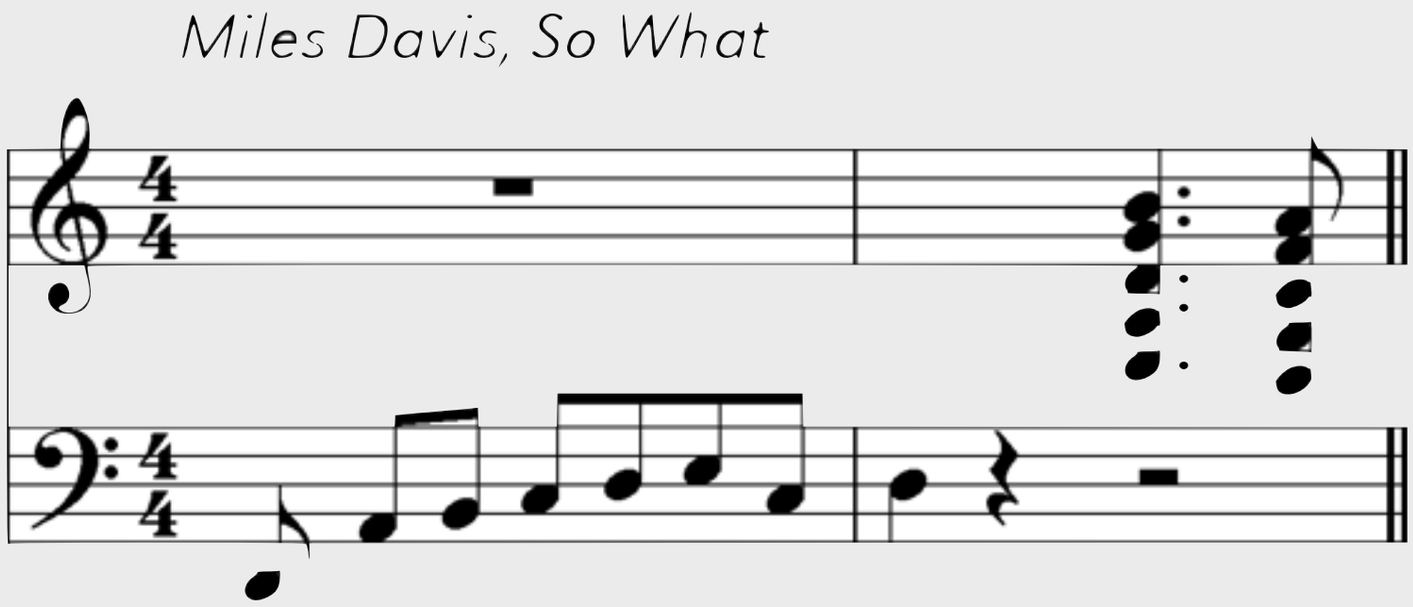
Non Blocking IOs : un problème informatique

- le problème est :
  - comment attendre sur deux choses à la fois ?
    - si je fais

      ```
      attendre_sur_canal_1()
      attendre_sur_canal_2()
      ```

    - en mode bloquant, si rien n'arrive sur le canal 1, rien de ce qui arrive sur le canal 2 n'est traité, il risque même de se remplir...
  - la stratégie symétrique a le défaut symétrique!

Non Blocking IOs : un problème informatique



Non Blocking IOs : un problème informatique

- la stratégie « moi les Threads, ça m'connaît »
  - un Thread par canal + des entrées/sorties **bloquantes**
    - le multithreading résoud le problème à sa façon en introduisant de l'asynchronisme (intrinsèque à la concurrence)
  - bien entendu, on déplace le problème puisqu'il faut maintenant gérer la concurrence
    - ce n'est pas forcément mieux...

Non Blocking IOs : un problème informatique

- la stratégie « moi on m'la fait pas! »
  - rappelons qu'au moins pour la lecture avec
    - `int InputStream.available()` renvoie le nombre d'octets pouvant être lus
    - `boolean Reader.ready()` indique s'il est possible de lire quelque chose (au moins 1 caractère)
  - pour l'écriture nada!
- bien mais un peu faible...

- la stratégie, pas besoin de remplacer le problème par un autre, peut-être que mon environnement m'offre quelque chose d'assez riche...

- les entrées/sorties de `java.io.*` et `java.net.*` sont bloquantes

- le package `java.nio` a été créé pour satisfaire les besoins d'entrées/sorties non-bloquantes (entre autres)

- mais aussi pour garantir une certaine efficacité aux opérations d'entrées/sorties

Les canaux (Channels)

- le premier concept important (ici) est celui de Channel de `java.nio.channels`, sans entrer dans les détails il en existe essentiellement trois déclinaisons intéressantes (toujours pour ce cours) :

- Channel est une interface avec deux méthodes
  - `close()`
  - `boolean isOpen()`

- Il en existe essentiellement trois déclinaisons intéressantes (toujours pour ce cours) :
  - `ServerSocketChannel`
  - `DatagramChannel`
  - `SocketChannel`
- et dont on devine bien les rôles respectifs

- `DatagramChannel` et `SocketChannel` sont (entre autres) des `ByteChannel`
- c'est-à-dire que peut les utiliser pour lire ou écrire
  - i.e. ce sont des `ReadableByteChannel` et `WriteableByteChannel`
- par conséquent ils sont très similaires à des `Socket` ou des `DatagramSocket`

- `ServerSocketChannel` est lui similaire à ServerSocket
  - on ne peut pas y lire ou y écrire
    - mais on peut faire `accept()`

- aucune de ces classes ne possède de constructeur directement accessible, il faut passer par une méthode usine :

```
DatagramChannel DatagramChannel.open();

ServerSocketChannel ServerSocketChannel.open();

SocketChannel SocketChannel.open();
```

- qui créent un objet correspondant

- les canaux correspondant sont créés mais pas connectés, il faut donc, si nécessaire, récupérer la `Socket`, la `ServerSocket` ou la `DatagramSocket` sous-jacente :

```
DatagramSocket socket();
ServerSocket socket();
Socket socket();
```

  - qui renvoient un objet du bon type
  - et effectuer l'association à l'aide des fonctions habituelles

- pour des raisons pratiques il existe dans chaque types de Sockets correspondant une méthode permettant de retrouver le Channel associé s'il existe :

```
*Channel getChannel();
```

by the Courtesy of RATP

il faut noter que l'on peut alors modifier le caractère bloquant ou non de ces objets via la méthode (héritée de `SelectableChannel`) via

```
SelectableChannel configureBlocking(boolean);
```

- on peut aussi tester ce caractère bloquant via

```
boolean isBlocking();
```

la modification du caractère bloquant ou non ne doit être faite que si le `Channel` n'est pas encore dans un `Selector` (voir plus loin)

Les sélecteurs (`Selectors`)

- outre les `Channels` et leur mode bloquant ou non, le second point important est qu'il est possible de demander à tout moment si certaines opérations d'un ensemble prédéterminé sont possibles
  - cette opération s'appelle la sélection et s'effectue via un sélecteur (`Selector`)
- la création d'un `Selector` s'effectue grâce à

```
Selector();
```

- un `Selector` permet donc de sélectionner un certain nombre d'opérations réalisables parmi un ensemble d'opérations souhaitées

  - toute opération souhaitée sur un `Channel` doit être préalablement enregistrée dans un sélecteur à l'aide de la méthode héritée de `SelectableChannel` :

```
SelectionKey register(Selector,int);
```

Attention, on ne peut enregistrer un sélecteur que s'il est en mode non-bloquant!

```
SelectionKey register(Selector,int);
```

- où :
  - le second argument est l'ensemble des opérations que l'on souhaite « surveiller » sur le Channel
  - le retour est la clé de sélection qui pourra être utilisée lors d'une future opération de sélection pour déterminer ce qu'il est possible de réaliser à ce moment là

- les opérations souhaitées doivent être compatibles avec le `Channel` choisi et sont toujours parmi les 4 suivantes (et qui peuvent être combinées par addition) :

```
SelectionKey.OP_ACCCEPT,
SelectionKey.OP_READ,
SelectionKey.OP_WRITE,
SelectionKey.OP_CONNECT
```

- On remarquera que chaque type de `Channel` fournit l'ensemble des opérations souhaitables possibles via

```
int validOps();
```

- une fois le sélecteur configuré, on peut soit :
  - attendre (donc en mode bloquant) qu'une des opérations soit réalisable par appel à

    ```
    int select();
    ```

  - tester (donc en mode non bloquant) si l'une des opérations est réalisable par appel à

    ```
    int selectNow();
    ```

- le retour indique le nombre de clés sur lesquelles des opérations sont réalisables

- la liste des clés concernées peut être extraite par appel à
  `Set<SelectionKey> selectedKeys();`

- un itérateur peut ensuite être employé pour parcourir cette liste afin de retrouver pour chaque clé :
  - les opérations possibles via
    `int readyOps();`
  - et le canal concerné via
    `SelectableChannel channel();`

Important : lors du parcours, il est essentiel de penser à supprimer la clé de sélection via l'itérateur si un traitement correspondant a été réalisé (ou va l'être)

Non Blocking IOs

- dans l’exemple qui suit, on met en place de quoi permettre à un serveur d’attendre des connexions entrantes sur deux ports en même temps

Non Blocking IOs

```
public static ServerSocketChannel
                createServerChannel(Selector s,int port) {
  try {
    // créé un canal serveur
    ServerSocketChannel ssc = ServerSocketChannel.open();
    // positionne-le en mode non-bloquant
    ssc.configureBlocking(false);
    // on configure la Socket sous-jacente
    ServerSocket ss = ssc.socket();
    ss.bind(new InetSocketAddress(port));
    // on configure le sélecteur
    ssc.register(s,ssc.validOps());
    return ssc;
  } catch(Exception e) {
    e.printStackTrace(); System.exit(1);
  }
  return null;
}
```

```
public static void main(String []args) {
   try {
      // Get a selector
      Selector s = Selector.open();
      // Create the non-blocking channels, bind and register them
      ServerSocketChannel ssc1 =
                  createServerChannel(s,50123);
      ServerSocketChannel ssc2 =
                  createServerChannel(s,50124);
…
```

Non Blocking IOs

```
// Catch incoming connection requests
while (true) {
  System.out.println("Waiting for incoming connections");
  s.select();
  Iterator<SelectionKey> it = s.selectedKeys().iterator();
  while (it.hasNext()) {
    // Get one key, and remove it from the set
    SelectionKey sk = it.next(); it.remove();
    // Get the channel
    ServerSocketChannel ssc =
                          (ServerSocketChannel)sk.channel();
    // accept
    Socket sock = ssc.socket().accept();
    // Launch a server with this service's Socket
    new Server(sock,ssc.socket().getLocalPort()).start();
    }
  }
 } catch(Exception e) { e.printStackTrace(); System.exit(1); }
}
```