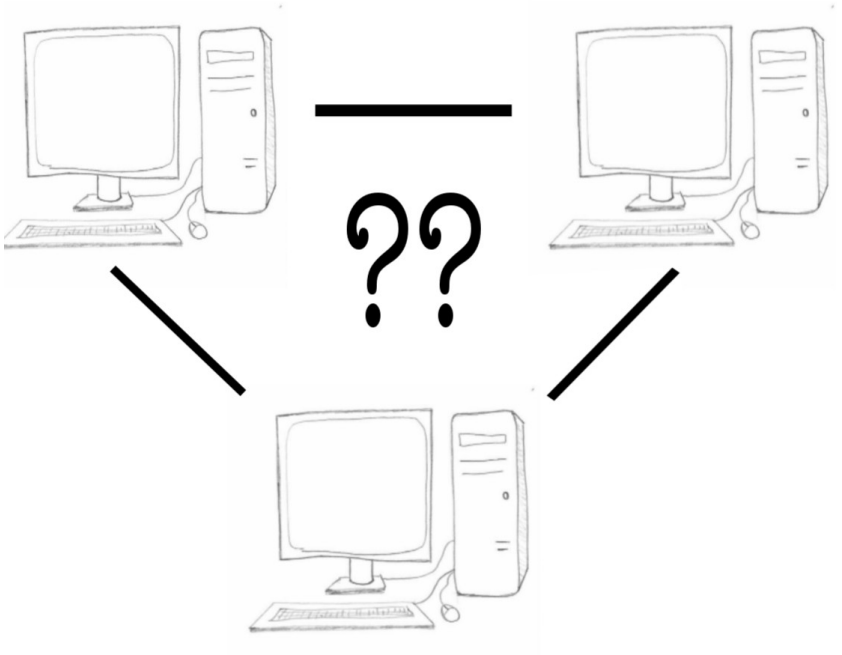
PROGRAMMATION RÉSEAU

Arnaud Sangnier
sangnier@irif.fr

Entrées-Sorties Non Bloquantes



E/S Bloquantes

- Jusqu'à présent :
  - Les méthodes de communication sur les socket étaient bloquantes
  - Tenter une opération met le demandeur en attente jusqu'à ce que l'opération puisse être réalisée (ou se révèle impossible)
    - Par exemple :
      - Attente d'un message sur une socket
      - Attente d'une connexion (dans le cas TCP)
  - **Avantages :**
    - Assez facile à gérer car il existe une synchronisation forte entre les différentes actions
  - **Inconvénients :**
    - Quand on est bloqué, on ne peut rien faire d'autre

Problème

- Exemple de problème
  - Vous attendez un colis chez vous
  - Le colis doit arriver dans la journée
  - Si vous sortez de chez vous, vous ratez le colis
  - Si jamais le colis n'arrive jamais, votre journée est gâchée
- Une entrée/sortie **non-bloquante** correspond à la situation dans laquelle on souhaite ne pas attendre si l'entrée/sortie ne peut pas être réalisée
  - Pourquoi faire ?
    - Pour faire autre chose, quitte à revenir essayer plus tard
    - Ou pour tester plusieurs choses en même temps

Difficulté

- Les difficultés dans l'utilisation d'E/S non-bloquantes sont multiples
  - Quand revient-on tester de nouveau ?
    - Exemple de mauvaise utilisation :
      - On fait une boucle qui teste toujours si il y a quelque chose (et qui ne fait rien d'autre)
        - -> Mieux vaut dormir en attendant un événement que faire de l'attente active
  - Que veut-on faire pendant ce temps ?
    - On verra que souvent ce que l'on va faire, c'est quand même bloqué mais dans l'attente de différents événements

Le boucher et le boulanger

- Vous devez acheter du pain et de la viande
- Il y a du monde chez le boucher et le boulanger
- Si vous attendez longtemps chez le boucher, peut être le boulanger sera fermé ensuite
- De plus, si vous attendez encore plus chez le boucher, peut-être celui-ci ne vous servira pas (il devra fermer)
- De même si vous attendez longtemps chez le boulanger
- Si on pouvait attendre chez les deux et aller chez le premier libre, on serait sûr d'avoir quelque chose à manger

Un exemple plus pratique

- On souhaite faire un programme qui écoute sur deux ports UDP en même temps (port1 et port2)
- Le problème est donc comment attendre sur deux choses à la fois
- Si l'on fait :
  - attendre sur le port 1 une donnée
  - puis, attendre sur le port 2 une donnée
- En mode bloquant, si rien n'arrive sur le port 1, on ne pourra jamais accédé à ce qui arrive sur le port 2, le canal de communication risque même de se remplir
- Si on choisit la solution symétrique (attente sur port2 puis sur port1), le même problème risque de se poser

Quelles solutions ? (1)

- On peut faire une solution multithreadé
- On crée deux threads
  - Le premier thread attend une donnée sur le port 1
  - Le deuxième thread attend une donnée sur le port 2
- Le parallélisme résout ainsi le problème de blocage que l'on avait avant
- **Avantages** :
  - On sait déjà le faire
- **Inconvénients :**
  - On repousse le problème à gérer correctement la concurrence

Exemple

```
public class AttenteUDP extends Thread{

    int port;

    public AttenteUDP(int p){
        port=p;
    }

    public void run(){
        try{
            DatagramSocket ds=new DatagramSocket(port);
            byte[]data=new byte[100];
            DatagramPacket paquet=new DatagramPacket(data,data.length);
            while(true){
                ds.receive(paquet);
                String st=new
String(paquet.getData(),0,paquet.getLength());
                System.out.println("PORT :"+port+", MESS :"+st);
            }
        } catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

Exemple

```
public class DoubleRead{
    public static void main(String[] args){
        try{
            AttenteUDP au1=new AttenteUDP(4343);
            AttenteUDP au2=new AttenteUDP(4344);
            au1.start();
            au2.start();
        } catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

Quelles solutions ? (2)

- Existe-t-il une solution proposée par Java pour ne pas remplacer le problème de l'attente multiple par un problème de gestion de la concurrence ?
- Y a-t-il une méthodes pour attendre plusieurs événements en même temps ?
- Les entrées/sorties de **java.io.*** et **java.net.*** sont bloquantes
- La librairie java.nio.* a été créée pour satisfaire les besoins d'entrées/sorties non bloquantes (entre autres)
- Mais aussi pour garantir une certaine efficacité sur l'accès aux entrées/sorties
- Nous allons voir comment utiliser cette librairie sur notre exemple

Les canaux (Channel)

- La première interface importante est l'interface Channel de la librairie **java.nio.channels**
- La plupart des classes que nous utiliserons pour nos entrées/sorties seront des Channel
- Cette interface ne contient que de deux méthodes
  - **boolean isOpen()**
  - **void close() throws IOException**
- Ce qui va nous intéresser, ce sont les déclinaisons de cette classe, en particulier
  - **SocketChannel**, **DatagramChannel** et **ServerSocketChannel**
- Ces classes vont substituées les classes :
  - **Socket**, **DatagramSocket** et **ServerSocket**

Les canaux (Channel)

- Pour les trois classes de canaux qui nous intéressent : **SocketChannel**, **DatagramChannel** et **ServerSocketChannel**
  - Il n'y aura pas de constructeur
  - Pour obtenir un nouveau canal, on utilisera la méthode statique **open()** des classes correspondantes
  - De plus pour ces classes, on peut dire si on souhaite que les objets soient bloquant ou non
    - on utilise la méthode **SelectableChannel configureBlocking(boolean block)**
  - On peut aussi tester si un canal est bloquant ou non
    - **boolean isBlocking()**

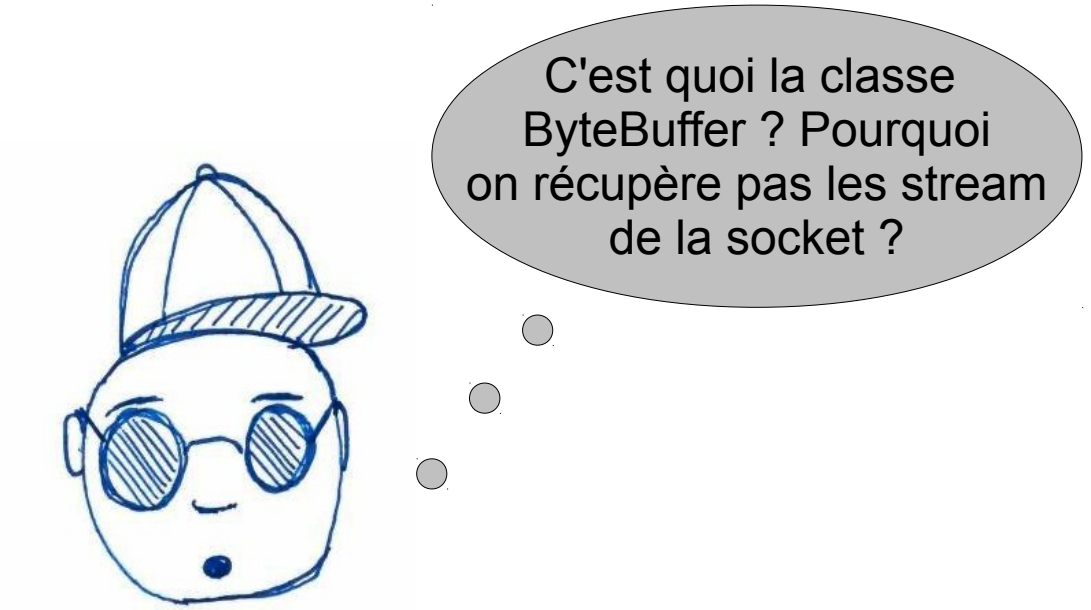
SocketChannel

- La classe **SocketChannel** va correspondre à une Socket en TCP
- On peut l'utiliser pour ce connecter à une machine
  - **public abstract boolean connect(SocketAddress r) throws IOException**
  - Attention si la **SocketChannel** est déclarée non-bloquante, on peut sortir du connect sans être connecté (la valeur renvoyée est false)
  - La méthode **boolean finishConnect()** permet de tester si la connexton est finie
  - Pour contourner ce problème, on peut faire :

```
sc.configureBlocking(false);
sc.connect(new InetSocketAddress("localhost", 5555));
while(! sc.finishConnect() ){
    //wait, or do something else...
}
```

- On peut l'utiliser pour lire et écrire des données
  - **abstract int read(ByteBuffer dst)**
  - **abstract int write(ByteBuffer src)**
- On peut récupérer la **Socket** associée avec la méthode **socket()**

Une nouvelle classe pour les E/S ?



- En fait, vous pouvez récupérer la socket pour faire des opérations comme **bind** ou changer des options
- Si le canal est non bloquant, vous ne pouvez plus écrire et lire sur la socket comme avant
- Il faut passer par des **read** et **write** de **ByteBuffer**

DatagramChannel

- La classe **DatragramChannel** va correspondre à une socket UDP
- On peut l'utiliser pour envoyer des paquets, pour cela on utilisera
  - **abstract DatagramChannel connect(SocketAddress remote)**
  - Attention même si on est en UDP on utilise connect pour préciser le destinataire
  - Ici on n'a pas de DatagramPacket
- Mais aussi pour écouter sur un port donné
  - **abstract DatagramChannel bind(SocketAddress local)**
  - On peut aussi là récupérer la DatagramSocket associée et l'utiliser pour faire le **bind**
    - On utilise la méthode **socket()**
- On peut l'utiliser pour lire et écrire des données
  - **abstract int read(ByteBuffer dst)**
  - **abstract int write(ByteBuffer src)**

ServerSocketChannel

- La classe ServerSocketChannel est associée à ServerSocket
- Elle permet de faire des attentes de connexions en TCP
- On peut l'associer à un port local
  - **ServerSocketChannel bind(SocketAddress local)**
- On peut ensuite ensuite accepter des connexions
  - **SocketChannel accept()**

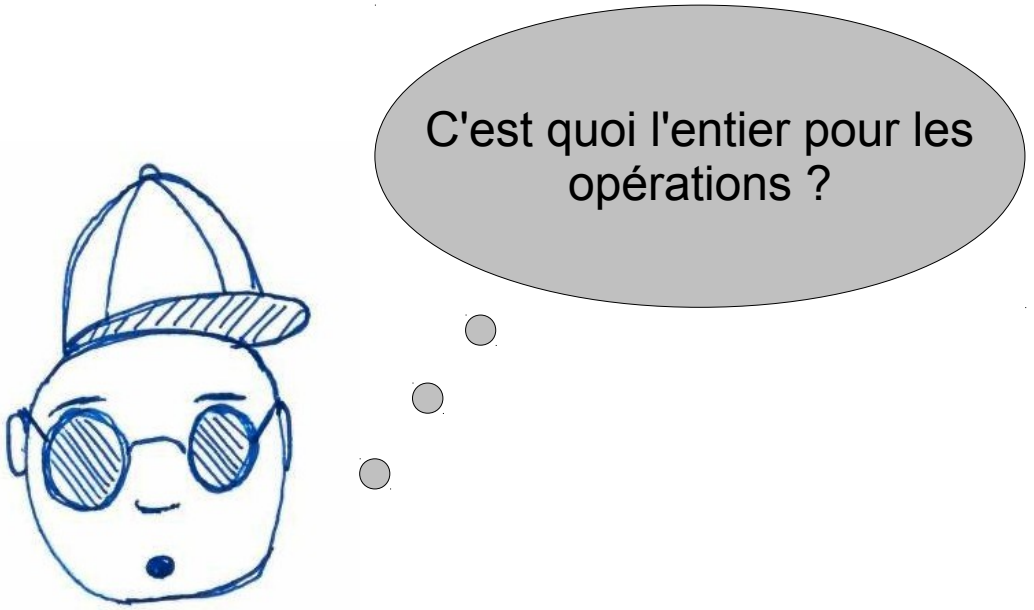
Comment utiliser ces canaux

- On va voir comment éviter de bloquer en lecture ou sur une attente de connexion particulière grâces aux canaux
- Le principe est le suivant
  - On créé un canal correspondant à ce que l'on souhaite faire
  - On déclare ce canal non bloquant
  - On enregistre le canal dans un sélecteur en indiquant les opérations à surveiller
  - On attend que le sélecteur nous dise qu'une opération est disponible
  - On réalise les opérations disponibles
  - On revient attendre sur le sélecteur

Les sélecteurs

- Les sélecteurs servent à observer à tout moment si des opérations sur des canaux sont possibles ou non
  - Cette opération s'appelle la sélection
- Pour cela on utilise la classe **Selector**
- Pour créer un sélecteur, on a un constructeur simple
  - **Selector()**
- Pour observer des événement sur un canal à l'aide d'un sélecteur, il faut enregistrer le canal ainsi que les opérations à observer
- On utilise la méthode de la classe **SelectableChannel**
  - **public final SelectionKey register(Selector sel,int ops) throws ClosedChannelException**
  - renvoie une clef correspondant à l'enregistrement
- **ATTENTION : LE CANAL DOIT ETRE NON BLOQUANT**

Comment ?



- Les opérations précisées doivent être compatibles avec le canal qui s'enregistre et sont parmi les quatre suivantes (que l'on peut combiner grâce à l'addition)
  - **SelectionKey.OP_ACCCEPT, SelectionKey.OP_READ, SelectionKey.OP_WRITE, SelectionKey.OP_CONNECT**

La sélection

- Une fois que l'on a enregistrée tous les canaux que l'on souhaite écouter, on peut passer à la sélection
- Deux options (deux méthodes du sélecteur) :

1) On attend (en mode bloquant) qu'une opération soit réalisable
   - **int select()**
2) On teste si une opération est réalisable (non bloquant) avec
   - **int selectNow()**
   - retourne le nombre de clés sur lesquelles des opérations sont possibles

Récupération des canaux

- Une fois sortie de la sélection, on peut récupérer les clefs correspondantes grâce à
  - **abstract Set<SelectionKey> selectedKeys()**
- Cette méthode renvoie un ensemble de SelectionKey que l'on peut parcourir grâce à un itérateur
- **ATTENTION :** Lorsque l'on traite une clef lors de l'itération, il faut retirer cette clef de l'ensemble avec remove par exemple
- Pour chaque **SelectionKey**, on peut
  - tester l'opération possible
    - **boolean isAcceptable(), boolean isReadable(),** ...
  - Récupérer le canal correspondant
    - **SelectableChannel channel()**

Exemple

```
public class DoubleRead2{
    public static void main(String[] args){
        try{
            Selector sel=Selector.open();
            DatagramChannel dsc1=DatagramChannel.open();
            DatagramChannel dsc2=DatagramChannel.open();
            dsc1.configureBlocking(false);
            dsc2.configureBlocking(false);

            dsc1.bind(new InetSocketAddress(4344));
            dsc2.bind(new InetSocketAddress(4343));

            dsc1.register(sel,SelectionKey.OP_READ);
            dsc2.register(sel,SelectionKey.OP_READ);
            ByteBuffer buff=ByteBuffer.allocate(100);
```

Exemple (suite)

```
while(true){
                System.out.println("Waiting for messages");
                sel.select();
                Iterator<SelectionKey> it=sel.selectedKeys().iterator();
                while(it.hasNext()){
                    SelectionKey sk=it.next();
                    it.remove();
                    if(sk.isReadable() && sk.channel()==dsc1){

                        System.out.println("Message UDP 4344 recu");
                        dsc1.receive(buff);
                        String st=new String(buff.array(),0,buff.array().length);
                        buff.clear();
                        System.out.println("Message :"+st);
                    } else if (sk.isReadable() && sk.channel()==dsc2){
                        System.out.println("Message UDP 4344 recu");
                        dsc2.receive(buff);
                        String st=new String(buff.array(),0,buff.array().length);
                        System.out.println("Message :"+st);
                    } else{
                        System.out.println("Que s'est il passe");

                    }
                }
            }

        } catch(Exception e){
            e.printStackTrace();
        }
```