Université Lille 1
Master mention Informatique – M1

Construction d'applications réparties

I. Applications réparties en mode message

Romain.Rouvoy@univ-lille.fr

Plan

1. Programmation réseau
   - 1.1 Notions générales
   - 1.2 TCP
   - 1.3 UDP
   - 1.4 Multicast IP
2. Programmation concurrente

1.1 Notions générales

Protocoles de transport réseaux

Protocoles permettant de **transférer des données de bout en bout**

- **s'appuient sur les protocoles rsx inférieur** (IP) pour routage, transfert noeud à noeud...
- **servent de socles pour les protocoles applicatifs** (RPC, HTTP, FTP, DNS...)

- API associées pour pouvoir envoyer/recevoir des données

UDP	mécanisme d'envoi de messages
TCP	flux **bi-directionnel** de communication
Multicast-IP	envoi de message à un groupe de destinataire

1.1 Notions générales

Caractéristiques des protocoles de transport réseaux

2 primitives de communications

- `send` **envoi** d'un message dans un *buffer* distant
- `receive` **lecture** d'un message à partir d'un *buffer* local

Propriétés associées :

fiabilité : est-ce que les messages sont garantis sans erreur ?
ordre : est-ce que les messages arrivent dans le même ordre que celui de leur émission ?
contrôle de flux : est-ce que la vitesse d'émission est contrôlée ?
connexion : les échanges de données sont-ils organisés en cx ?

1.1 Notions générales

Caractéristiques des protocoles de transport réseaux

2 modes pour les primitives `send` et `receive`

- bloquants
- non-bloquants

En Java

- `send` **bloquant** (jusqu'à envoi complet du message)
- `receive` **bloquant** (jusqu'à ce qu'il y ait un message à lire)

2 modes x 2 primitives = 4 combinaisons

Bloquant	+ souple
Non-bloquant	programme + simple à écrire

→ receive bloquant + *multi-threading* ≈ receive non-bloquant

1.1 Notions générales

Adressage

Classe `java.net.InetAddress`

Création

```
InetAddress host = InetAddress.getLocalHost();
InetAddress host = InetAddress.getByName("www.univ-lille.fr");
InetAddress[] host = InetAddress.getAllByName("www.univ-lille.fr");
```

Méthodes principales

- adresse symbolique : `String getHostName()`
- adresse IP : `String getHostAddress()`
- adresse binaire : `byte[] getAddress()`

Propriétés du protocole TCP

Connexions TCP

demande **d'ouverture par un client**
acception explicite de la demande **par le serveur**
⇨ au delà échange en mode **bi-directionnel**
⇨ au delà distinction rôle client/serveur "artificielle"

fermeture de connexion à l'initiative du client ou du serveur
⇨ vis-à-vis notifié de la fermeture

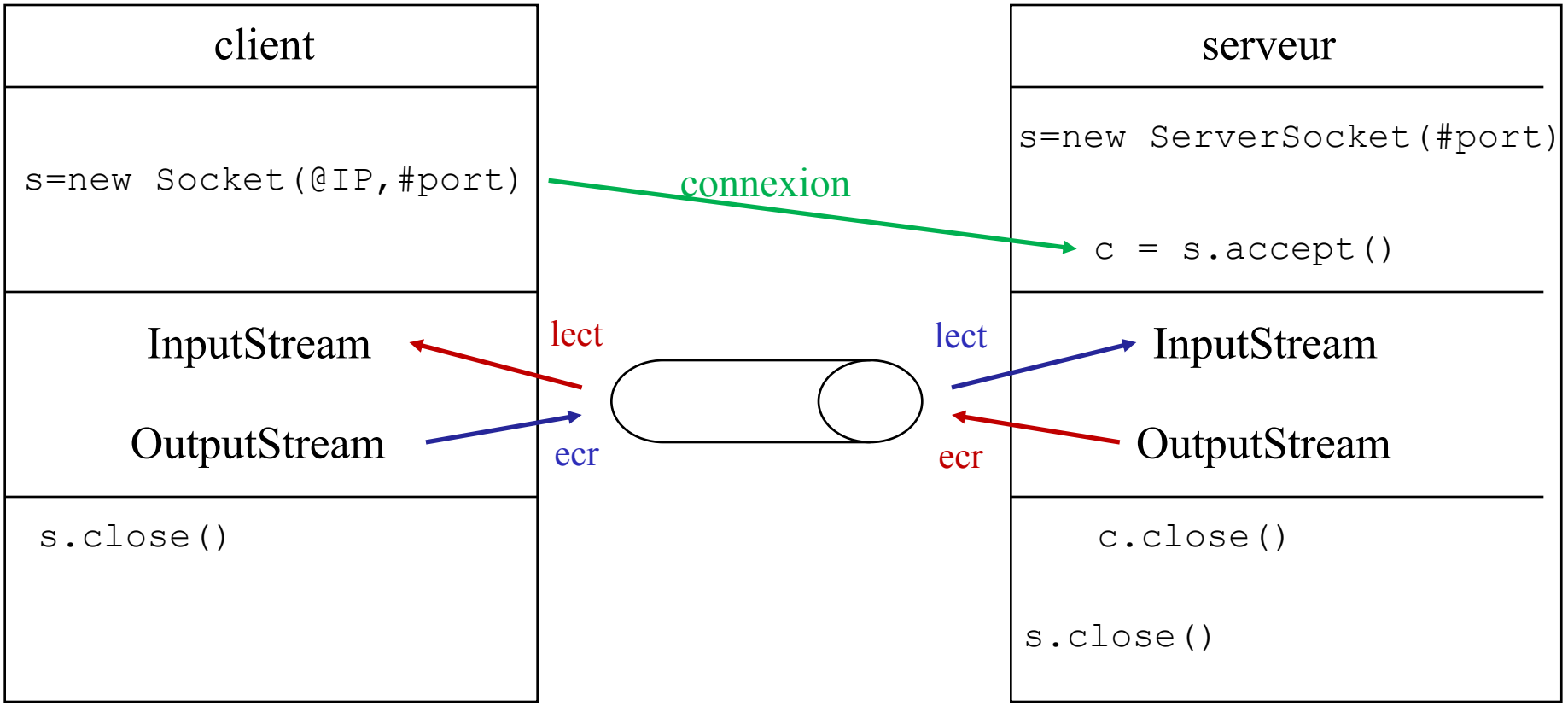
Pas de mécanisme de gestion de panne

trop de pertes de messages ou réseau trop encombré
→ connexion perdue

Utilisation

nombreux protocoles applicatifs : HTTP, **FTP**, Telnet, SMTP, POP...

Fonctionnement



Fonctionnement

1. serveur crée une *(server)socket* et **attend une demande de connexion**
2. client envoie une demande de connexion
3. serveur accepte connexion
4. dialogue client/serveur en mode flux
5. fermeture de connexion à l'initiative du client ou du serveur

API `java.net.Socket`

Constructeur : adresse + nº port

```
Socket s = new Socket("www.univ-lille.fr",80);
Socket s = new Socket(inetAddress,8080);
```

Méthodes principales

- adresse IP : `InetAddress getInetAddress(), getLocalAddress()`
- port : `int getPort(), getLocalPort()`
- flux in : `InputStream getInputStream()`
- flux out : `OutputStream getOutputStream()`
- fermeture : `close()`

1.2 TCP

API `java.net.Socket`

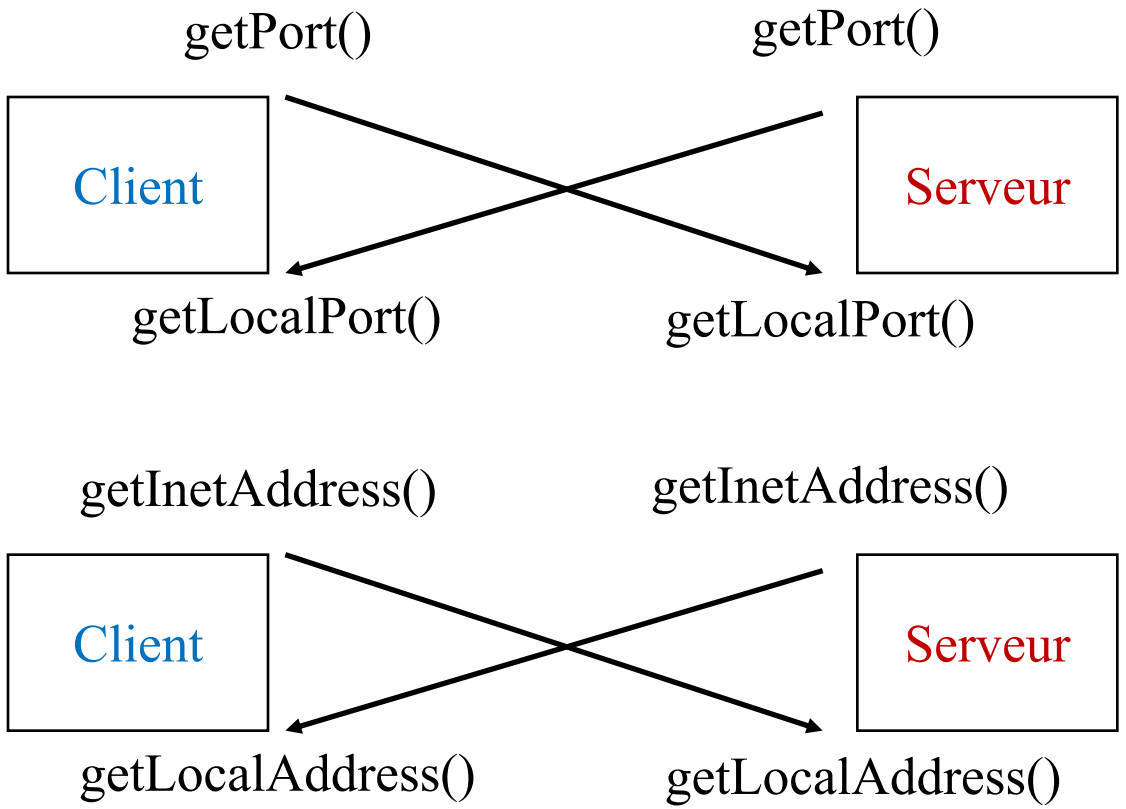
Options TCP : `timeOut`, `soLinger`, `tcpNoDelay`, `keepalive`

`setSoTimeout(int)`

`int = 0`	read bloquant (à l'∞) tant qu'il n'y a pas de données à lire
`int > 0`	délai max. de blocage sur un read passé ce délai, exception `SocketTimeoutException` levée (la socket reste néanmoins opérationnelle)

API `java.net.Socket`

Symétrie des valeurs retournées lorsque 2 *sockets* sont connectées



1.2 TCP

API `java.net.ServerSocket`

Constructeur : n° port

```
ServerSocket s = new ServerSocket(8080);
```

Méthodes principales

- adresse IP : `InetAddress getInetAddress()`
- port : `int getLocalPort()`

- attente de connexion : `Socket accept()`
- fermeture : `void close()`

Options TCP : timeOut, receiveBufferSize

1.2 TCP

API `java.net.ServerSocket`

Méthode `accept()` bloquante par défaut

→ schéma de programmation *dispatcheur*
- 1 *thread dispatcheur* écoute sur un port (et ne fait que ça)
- dès qu'une connexion arrive le travail est délégué à un autre *thread*

⇨ le thread *dispatcheur* ne fait "que" des appels à `accept()`

→ setSoTimeout(int)

int = 0 accept bloquant (à l'∞) tant qu'il n'y a pas de connexion à accepter

int > 0 délai max de blocage sur un `accept`
passé ce délai, exception `SocketTimeoutException`

→ `java.nio` à partir JDK 1.4 : `ServerSocket.getChannel()`

retourne une instance de `ServerSocketChannel`

qui implance une méthode `accept()` non bloquante

Personnalisation du fonctionnement des *sockets*

1. Modification des données transmises
2. Utilisation d'une *Factory*
3. Sous-classage

Modification des données transmises

Besoin : compression, chiffrage, signature, audit...

Solution

construire de nouveaux flux d'entrée/sortie à partir de

```
in = aSocket.getInputStream()
out = aSocket.getOutputStream()
```

ex :

```
zin = new GZIPInputStream(in)
zout = new GZIPOutputStream(out)
```

→ lire/écrire les données avec `zin` et `zout`

Personnalisation du fonctionnement des *sockets*

Utilisation d'une *Factory* (instance chargée de créer d'autres instances)

Besoin
- contrôle des param. (port, options TCP) des *sockets* créées par `new Socket()`
- redirection automatique de *sockets* pour franchir des *firewalls*

Solution

appel de la méthode

```
static Socket.setSocketFactory(SocketImplFactory)
```

en fournissant une instance implantant l'interface

```
interface SocketImplFactory {
    SocketImpl createSocket();
}
```

- un seul appel par programme
- idem `static` **`ServerSocket.`**`setSocketFactory(SocketImplFactory)`

Personnalisation du fonctionnement des *sockets*

Sous-classage

Dériver `Socket` et `ServerSocket`
Redéfinir `accept()`, `getInputStream()`, `getOutputStream()`...

1.3 UDP

Propriétés du protocole UDP

Taille des messages

limitée par l'implantation d'IP sous-jacente (en général 64 K)

Perte de messages

possible

rq : ok pour certaines applications (*streaming* audio, vidéo...)

Pas de contrôle de flux

Ordre des messages non garanti

Pas de connexion

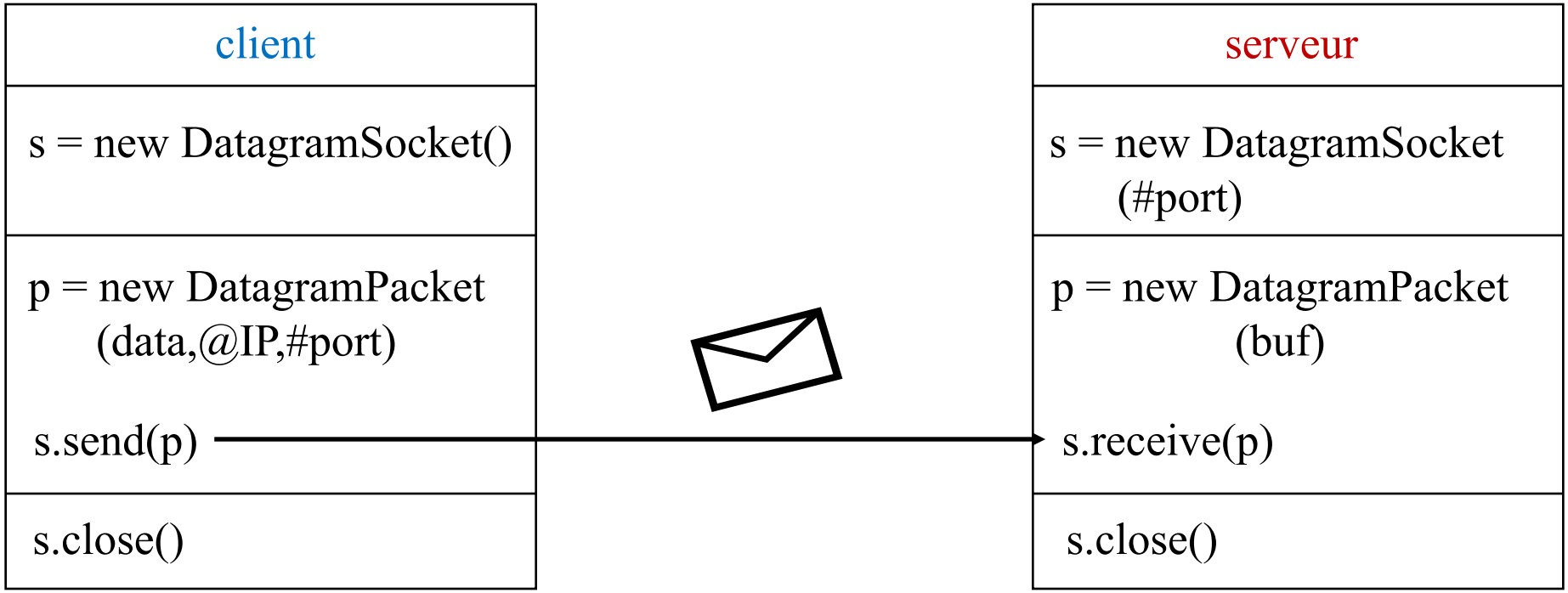
⇨ **+ performant que TCP**

⇨ **- de qualité de service** (fiabilisation peut être "reprogrammée" au niveau applicatif)

Utilisation : NFS, DNS, TFTP, *streaming*, jeux en réseau

Fonctionnement

- serveur crée une socket UDP
- serveur attend la réception d'un message
- client envoie message



1.3 UDP

API `java.net.DatagramSocket`

Constructeur	`DatagramSocket(port)`	*socket* UDP sur #port
	`DatagramSocket()`	*socket* UDP sur port qqconque
	(+ ...)	
`send(DatagramPacket)`		envoi
`receive(DatagramPacket)`		réception

Options UDP : timeOut...

Rq : possibilité de "connecter" une socket UDP à une (@IP,#port)
→ `connect(InetAddress, int)`
→ pas de connection réelle, juste un contrôle pour restreindre les send/receive

2 solutions pour asynchronisme `receive()`
→ *setSoTimeout*
→ `java.nio` à partir JDK 1.4

1.3 UDP

API `java.net.DatagramPacket`

Constructeur `DatagramPacket( byte[] buf, int length )`
`DatagramPacket( byte[] buf, int length, InetAddress, port )`
(+ ...)

`getPort()` port de l'émetteur pour une réception
port du récepteur pour une émission

`getAddress()` *idem* adresse

`getData()` les données reçues ou à envoyer

`getLength()` *idem* taille

Personnalisation du fonctionnement des *sockets* UDP

- Factory
- Sous-classage

⇨ même principe que pour les *sockets* TCP

1.4 Multicast IP

Multicast IP

Diffusion de messages vers un groupe de destinataires

- messages émis sur une @
- messages reçus par tous les récepteurs "écoutant" sur cette @
- +sieurs émetteurs possibles vers la même @
- les récepteurs peuvent rejoindre/quitter le groupe à tout instant

- @ IP de classe D (de 224.0.0.1 à 239.255.255.255)

⇨ @ indépendante de la localisation φ des émetteurs/récepteurs

Même propriétés que UDP

taille des messages limitée à 64 K
pas de contrôle de flux
pas de connexion

perte de messages possible
ordre des messages non garanti

1.4 Multicast IP

Utilisation

MBone, jeux en réseaux... + nbreuses applications

Caractéristique **intéressante** de Multicast IP

- indépendance entre service et localisation φ (⇨choix @ IP classe D)
- possibilité de doubler/multiplier les instances de service
  ⇨ tolérance aux pannes, réponse de la + rapide...

Certaines @ classe D sont assignées
(voir `http://www.iana.org/assignments/multicast-addresses`)

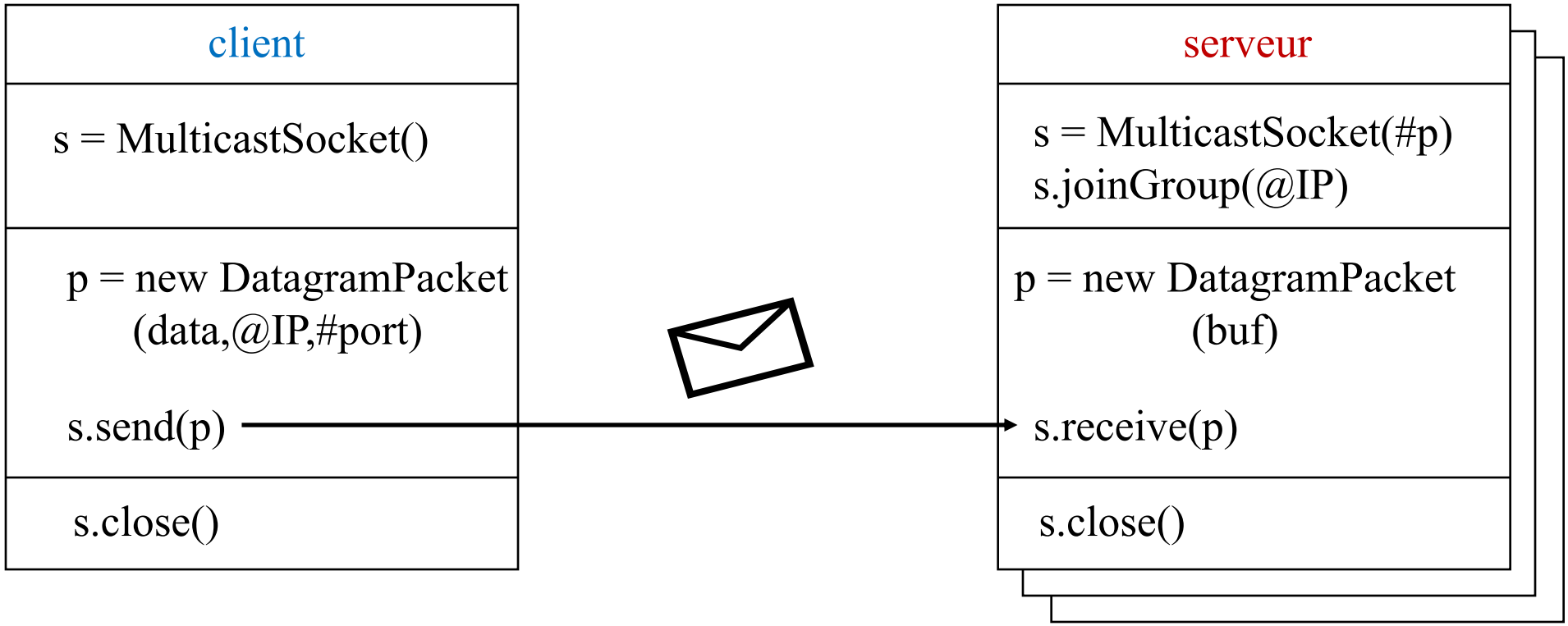
Ex :

```
224.0.6.000-224.0.6.127      Cornell ISIS Project
224.0.18.000-224.0.18.255    Dow Jones
224.0.19.000-224.0.19.063    Walt Disney Company
```

1.4 Multicast IP

Fonctionnement

- serveur(s) créent une socket MulticatIP
- chaque serveur rejoint le groupe de diffusion
- client envoie message



1.4 Multicast IP

API `java.net.MulticastSocket`

Constructeur	`MulticastSocket(port)`	sur #port
	`MulticastSocket()`	sur port qqconque
	(+ ...)	
`send(DatagramPacket)`		envoi
`receive(DatagramPacket)`		réception
`joinGroup(InetAddress)`		se lier à un groupe
`leaveGroup(InetAddress)`		quitter un groupe

Ø de la diffusion contrôlable avec TTL `setTimeToLive(int)`

⇨ # de routeurs que le paquet peut traverser avant d'être arrêté

= 0	aucun (le paquet ne quitte pas la machine)
= 1	même sous-réseau
= 128	monde entier

⇨ diffusions souvent bloquées par les routeurs malgré TTL

1.4 Multicast IP

Autres mécanismes de diffusion sur groupe

Construction de protocoles fiables au-dessus de MulticastIP

- Jgroups `http://www.cs.unibo.it/projects/jgroup/`
- LRMP `http://webcanal.inria.fr/lrmp/index.html`
- JavaGroups `http://sourceforge.net/projects/javagroups/`
- ...

Exemples de propriétés fournies

- fragmentation/recomposition messages > 64 K
- ordre garanti des messages
- notifications d'arrivées, de retraits de membres
- organisation arborescente des nœuds de diffusion

2. Programmation concurrente

2.1 Introduction

Threads Java

Possibilité de programmer des traitements concurrents au sein d'une JVM

⇨ simplifie la programmation dans de nbreux cas
- programmation événementielle (ex. IHM)
- I/O non bloquantes
- *timers*, déclenchements périodiques
- servir +sieurs clients simultanément (serveur Web, BD...)

⇨ meilleure utilisation des capacités (CPU) de la machine
- utilisation des temps morts

2.1 Introduction

Threads Java

Processus vs *threads* (= processus légers)

- processus : espaces mémoire séparés (API `java.lang.Runtime.exec(..)`)
- *threads* : espace mémoire partagé
  (seules les piles d'exécution des *threads* sont ≠)

⇨ + efficace

⇨ - robuste

- le plantage d'un *thread* peut perturber les autres
- le plantage d'un processus n'a pas (normalement) d'incidence sur les autres

Approches mixtes : +sieurs processus ayant +sieurs *threads* chacun

Java

- 1 JVM = 1 processus (au départ)
- mécanisme de *threads* intégré à la JVM (vers *threads* noyau ou librairie)

Modèle de programmation

Ecriture d'une classe

- héritant de `java.lang.Thread`
- ou implantant l'interface `java.lang.Runnable`

```
interface Runnable {
    public void run();
}
```

Dans les 2 cas, les instructions du *thread* sont définies dans la méthode `run()`

```
public void run() {        // seule signature possible
    ...
}
```

Rq: Java 8 notation fonctionnelle

```
() -> { ... }
```

Modèle de programmation

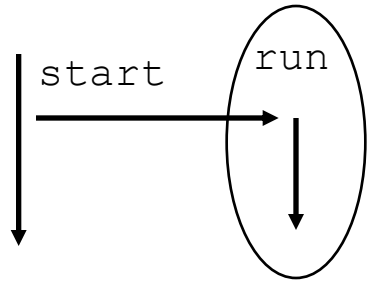
Lancement d'un thread : appel à la méthode `start()`

```
public class MonThread extends Thread {
    public void run() { ... }
}
```

Code lanceur

```
MonThread mthread = new MonThread();
mthread.start();
```

⇨ exécution concurrente
du code lanceur et du *thread*



Remarque : la méthode `main()` est associée automatiquement à un *thread*

Modèle de programmation

2è possibilité : utilisation du constructeur **`public`** `Thread(Runnable)`

Programme lanceur

```
public class MonThread2 implements Runnable {
    public void run() { ... }
}

MonThread2 foo = new MonThread2();
Thread mthread = new Thread(foo);
mthread.start();
```

Remarques

- création d'autant de *threads* que nécessaire (même classe ou classes ≠)
- appel de `start()` une fois et une seule pour chaque *thread*
- un *thread* meurt lorsque sa méthode `run()` se termine
- !! on appelle jamais directement `run()` (`start()` le fait) !!

Modèle de programmation

Remarques

- pas de passage de paramètre au *thread* via la méthode `start()`

  ⇨ définir des variables d'instance
  ⇨ les initialiser lors de la construction

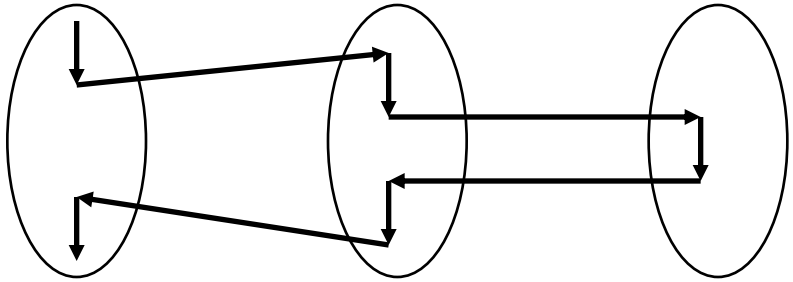
```
public class Foo implements Runnable {

    int p1;
    Object p2;

    public Foo(int p1,Object p2) { this.p1=p1; this.p2=p2; }
    public void run() { ... p1 ... p2 ... }
}

new Thread(new Foo(12,aRef)).start();
```

2.3 Synchronisation

Modèle d'objets multi-threadé passifs

En Java : *threads* ⊥ objets

- *threads* non liés à des objets particuliers
- exécutent des traitements sur éventuellement +sieurs objets
- sont eux-même des objets



"autonomie" possible pour un objet (≈ notion d'agent)
⇨ "auto"-*thread*

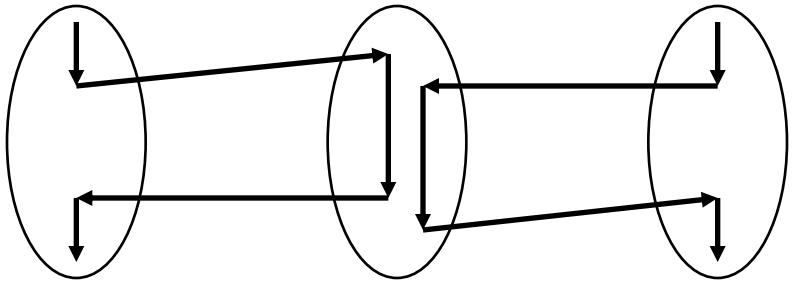
```
public class Foo implements Runnable {
    public Foo() { new Thread(this).start(); }
    public void run() { ... }
}
```

⇨ la construction d'un objet lui assigne des instructions à exécuter

Modèle d'objets multi-threadé passifs

2 ou +sieurs *threads* peuvent exécuter la **même méthode** simultanément

⇨ 2 flux d'exécutions distincts (2 piles)
⇨ 1 même espace mémoire partagé (les champs de l'objet)



Exclusion mutuelle

Besoin : code d'une méthode section critique

⇨ au plus 1 *thread* simultanément exécute le code de cette méthode pour cet objet
⇨ utilisation du mot clé `synchronized`

```
public synchronized void ecrire(...) { ... }
```

⇨ si 1 *thread* exécute la méthode, les autres restent bloqués à l'entrée

⇨ dès qu'il termine, le 1er *thread* resté bloqué est libéré
⇨ les autres restent bloqués

Exclusion mutuelle

Autre besoin : bloc de code ($\in$ à une méthode) section critique

⇨ au plus 1 *thread* simultanément exécute le code de cette méthode pour cet objet
⇨ utilisation du mot clé `synchronized`

```
public void ecrire2(...) {
    ...
    synchronized(objet) { ... }   // section critique
    ...
}
```

`objet` : objet de référence pour assurer l'exclusion mutuelle (en général **`this`**)

Chaque objet Java est associé à un verrou
`synchronized` = demande de prise du verrou

2.4 Exclusion mutuelle

Exclusion mutuelle

Le contrôle de concurrence s'effectue au niveau de l'objet

⇨ +sieurs exécutions d'une même méth. `synchronized` dans des objets ≠ possibles

⇨ si +sieurs méthodes `synchronized` ≠ dans 1 même objet
au plus 1 *thread* dans **toutes les méthodes** `synchronized` de l'objet
⇨ les autres méthodes (non `synchronized`) sont tjrs exécutables concurrement

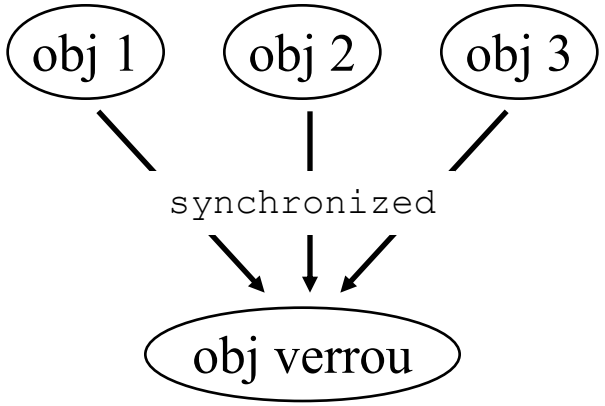
Remarques

- JVM garantit atomicité d'accès au byte, char, short, int, float, réf. d'objet
  !! pas long, ni double !!
- coût
  appel méthode `synchronized` ≈ 4 fois + long qu'appel méthode "normal"
  ⇨ à utiliser à bon escient

Exclusion mutuelle

Autre besoin : exclusion mutuelle "à +sieurs"
i.e. + méthodes et/ou blocs de codes dans des obj. ≠ en exclusion entre eux

⇨ choix d'un objet de référence pour l'exclusion
⇨ tous les autres se "`synchronized`" sur lui



2.5 Autres politiques

Autres politiques de synchronisation

Ex : lecteurs/écrivain, producteur(s)/consommateur(s)

⇨ utilisation des méthodes `wait()` et `notify()` de la classe `java.lang.Object`
⇨ disponibles sur tout objet Java

`wait()` : met en attente le *thread* en cours d'exécution
`notify()` : réactive un *thread* mis en attente par `wait()`
si pas de *thread* en attente, RAS

!! ces méthodes nécessitent un accès exclusif à l'**objet exécutant** !!
⇨ à utiliser avec méthode `synchronized` ou bloc `synchronized(this)`

```
synchronized(this) { wait(); }
synchronized(this) { notify(); }
```

sinon exception "`current thread not owner`"
⇨ + `try/catch(InterruptedException)`

2.5 Autres politiques

Méthode `wait()`

Fonctionnement

Entrée dans `synchronized`
- acquisition de l'accès exclusif à l'objet (`synchronized`)

`wait()`
- mise en attente du *thread*
- relachement de l'accès exclusif
- attente d'un appel à `notify()` par un autre *thread*
- attente de la réacquisition de l'accès exclusif
- reprise de l'accès exclusif

Sortie du `synchronized`
- relachement de l'accès exclusif à l'objet

2.5 Autres politiques

Méthode `notify()`

Réactivation d'un thread en attente sur un `wait()`

Si +sieurs *threads*

- spec JVM : pas de garantie sur le *thread* réactivé
- en pratique les implantations de la JVM réactivent le 1er endormi

`notify()` pas suffisant pour certaines politiques de synchronisation notamment lorsque compétition pour l'accès à une ressource

- 2 *threads* testent une condition (faux pour les 2) → `wait()`
- 1 3ème *thread* fait `notify()`
- le *thread* réactivé teste la condition (tjrs faux) → `wait()`

→ les 2 *threads* sont bloqués

→ `notifyAll()` réactive **tous les *threads*** bloqués sur `wait()`

2.5 Autres politiques

Politique lecteurs/écrivain

soit 1 seul écrivain, soit plusieurs lecteurs

- demande de lecture : bloquer si écriture en cours ⇨ booléen écrivain
- demande d'écriture : bloquer si écriture ou lecture en cours ⇨ compteur lecteurs

réveil des bloqués en fin d'écriture et en fin de lecture

```
boolean ecrivain = false;
int lecteurs = 0;
```

2.5 Autres politiques

Politique lecteurs/écrivain

- demande de lecture : bloquer si écriture en cours
- réveil des bloqués en fin de lecture

```
// Avant lecture
synchronized(this) {
    while (ecrivain) { wait(); }
    lecteurs++;
}

// ...
// On lit
// ...

// Après lecture
synchronized(this) {
    lecteurs--;
    notifyAll();
}
```

2.5 Autres politiques

Politique lecteurs/écrivain

- demande d'écriture : bloquer si écriture ou lecture en cours
- réveil des bloqués en fin d'écriture

```
// Avant écriture
synchronized(this) {
    while (ecrivain || lecteurs>0) { wait(); }
    ecrivain = true;
}

// ...
// On écrit
// ...

// Après écriture
synchronized(this) {
    ecrivain = false;
    notifyAll();
}
```

2.5 Autres politiques

Politique producteurs/consommateurs

1 ou +sieurs producteurs, 1 ou +sieurs consommateurs, zone tampon de taille fixe

- demande de production : bloquer si tampon plein
- demande de consommation : bloquer si tampon vide

réveil des bloqués en fin de production et en fin de consommation

```
int max = ...           // taille du tampon
tampon = ...            // tableau de taille max
int taille = 0;         // # d'éléments en cours dans le tampon
```

2.5 Autres politiques

Politique producteurs/consommateurs

- demande de production : bloquer si tampon plein
- réveil des bloqués en fin de production

```
// Avant production
synchronized(this) {
    while (taille == max) { wait(); }
    taille++;
}

// On produit (maj du tampon)

// Après production
synchronized(this) {
    notifyAll();
}
```

2.5 Autres politiques

Politique producteurs/consommateurs

- demande de consommation : bloquer si tampon vide
- réveil des bloqués en fin de consommation

```
// Avant consommation
synchronized(this) {
    while (taille == 0) { wait(); }
    taille--;
}

// On consomme (maj du tampon)


// Après consommation
synchronized(this) {
    notifyAll();
}
```

2.5 Autres politiques

Schéma général de synchronisation

- bloquer (éventuellement) lors de l'entrée
- réveil des bloqués en fin

```
synchronized(this) {
  while (!condition) {
    try { wait(); } catch(InterruptedException ie) {}
  }
}

// ...

synchronized(this) {
    try { notifyAll(); } catch(InterruptedException ie) {}
}
```

2.5 Autres politiques

Implantation d'une classe sémaphore

```
public class Semaphore {

    private int nbThreadsAutorises;

    public Semaphore( int init ) {
        nbThreadsAutorises = init;
    }

    public synchronized void p() {
        while ( nbThreadsAutorises <= 0 ) {
            try { wait(); } catch(InterruptedException ie) {}
        }
        nbThreadsAutorises --;
    }

    public synchronized void v() {
        nbThreadsAutorises ++;
        try { notify(); } catch(InterruptedException ie) {}
    }
}
```

2.6 Compléments

API

`wait(timeout)` mise en attente au max `timeout` ms

`static Thread.sleep(m)` endormissement du *thread* courant `m` ms

`static Thread Thread.currentThread()`
retourne un objet `Thread` représentant le *thread* courant

`Thread.interrupt()` lève une exception `InterruptedException`

`Thread.join()` attend la fin d'un *thread* (ie fin méthode `run`)

D'autres méthodes `stop`, `suspend`, `resume`
ont été "dépréciées" depuis JDK 1.2
car sources d'interblocage

→ utiliser `interrupt()`
→ utiliser des tests explicites dans la méthode `run` pour orienter "la vie" d'un *thread*

```
public void run() { while(goon==true) {...} }
```

Pool de thread

Serveurs concurrents avec autant de *threads* que de requêtes
⇨ concurrence "débridée"
⇨ risque d'écroulement du serveur

Pool de thread : limite le nb de *threads* à disposition du serveur

Pool fixe : nb cst de *threads*
Pb : dimensionnement

Pool dynamique
- le nb de *threads* s'adapte à l'activité
- il reste encadré : [borne sup , borne inf]

Optimisation : disposer de *threads* en attente (mais pas trop)
- encadrer le nb de *threads* en attente

2.6 Compléments

I/O asynchrone

But : pouvoir lire des données sur un flux sans rester bloquer en cas d'absence

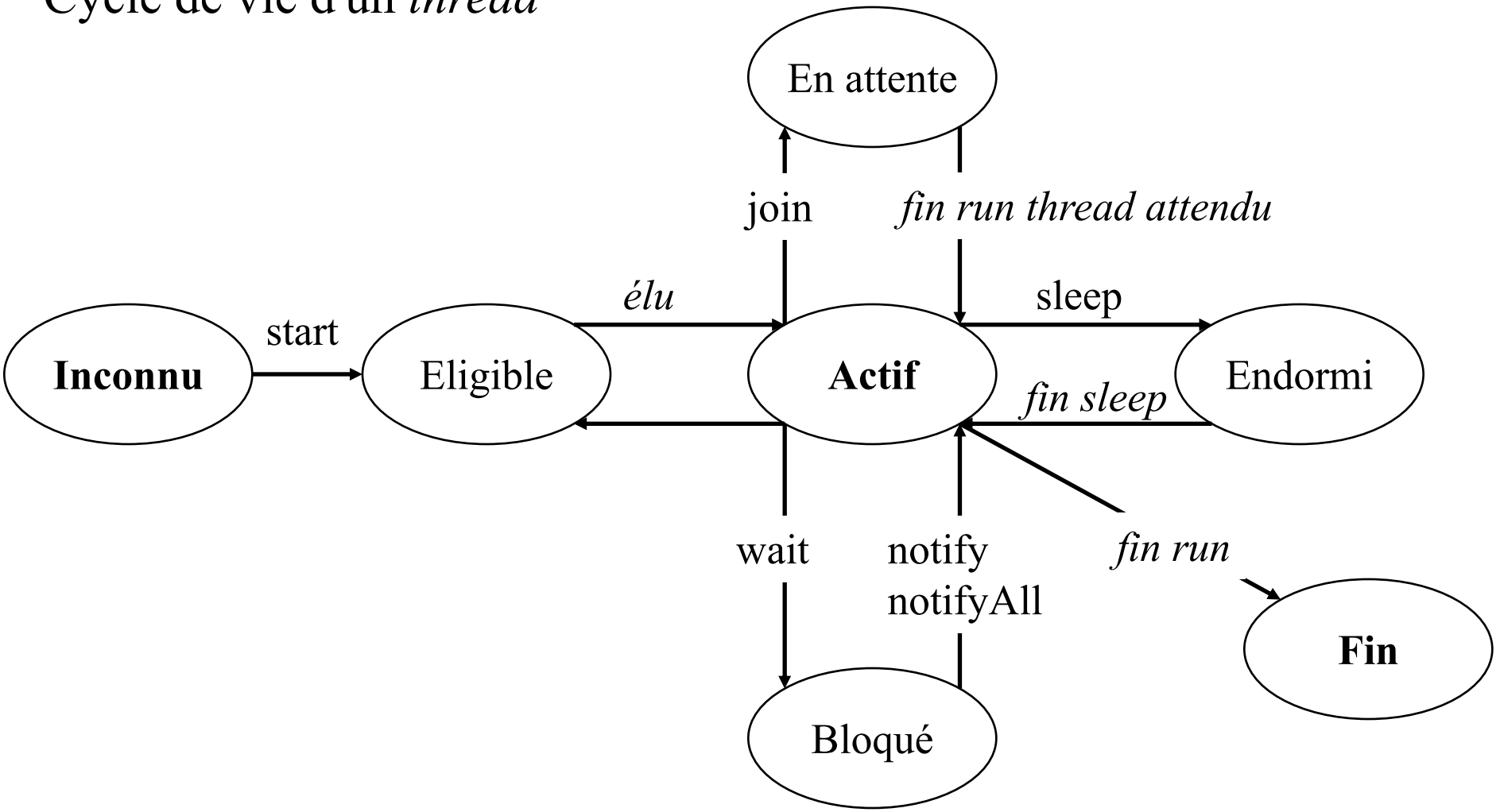
Solution
- un *thread* qui lit en permanence et stocke les données dans un buffer
- une méthode read qui lit dans le buffer

```
public class AsyncInputStream
  extends FilterInputStream implements Runnable {

  int[] buffer = ...   // zone tampon pour les données lues

  AsyncInputStream( InputStream is ) {
    super(is); new Thread(this).start(); }
```

```
public void run() {
  int b = is.read();
  while( b != -1 ) {
    // stocker b dans buffer
    b = is.read(); } }
```

```
public int read() {
  return ...
  // 1ère donnée dispo dans buffer
}
```

Cycle de vie d'un *thread*



2.6 Compléments

Priorités

Permet d'associer des niveaux d'importance (de 1 à 10) aux *threads*
Par défaut 5

Spec JVM : !! aucune garantie sur la politique d'ordonnancement !!

Groupes de *threads*

Permet de grouper des *threads* pour les traiter globalement

- gestion des priorités
- interruption

Organisation hiérarchique avec liens de parenté entre les groupes