Master mention Informatique
M1

Construction
d'Applications Réparties

Lionel Seinturier
Lionel.Seinturier@univ-lille1.fr
2014/2015

Objectifs du cours

Appréhender la conception d'applications réparties
- motivations et concepts
- architectures et exemples
- problèmes et solutions

Comprendre les solutions
- Internet et sockets Java
- Web Services, REST, SOAP
- Objets répartis en Java (RMI)
- Composants Java EE (JSP, servlet, EJB)

Maîtriser par la pratique (importance des TP)
- Java, REST, RMI, Java EE

Sommaire

Organisation

- début cours : 12 janvier
- début TD : semaine du 19/1
- début TP : semaine du 26/1

Enseignants TD/TP
- J.-C. Bach, C. Ballabriga, L. Duchien, G. Lipari, M. Monperrus

http://www.fil.univ-lille1.fr/portail/
- M1S2 > CAR

http://moodle.univ-lille1.fr/course/view.php?id=346
- clé d'inscription : car

Organisation du contrôle

Un examen note : 60%

4 sujets de TPs note : 40%

- chaque sujet de TP dure 3 (ou 2) séances
- utilise le langage Java
- démonstration de chaque TP en séance
- les TPs sont relevés et notés
  - relève par l'application PROF uniquement (mail refusé)
  - dates de remise fermes
  - binôme ∈ même groupe (pas de changement de binôme)

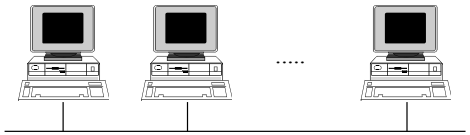
I. Introduction aux applications réparties

Lionel Seinturier

Lionel.Seinturier@univ-lille1.fr

Introduction

Problématique



Permettre à un programme de s'exécuter sur **plusieurs machines**
(PC, *mainframe*, *laptop*, PDA, …) reliées par un réseau
- à large échelle (Internet)
- local (intranet)

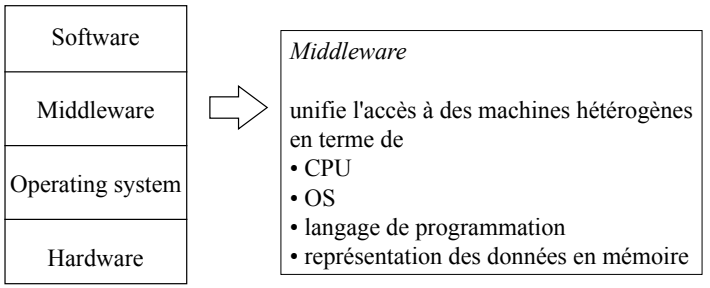
∩ de plusieurs domaines de l'informatique

- système d'exploitation - réseau - langage de programmation	- système d'exploitation répartis - librairies de programmation réseau - langages de programmation étendus

Introduction

Vocabulaire

- application répartie
- exécutées par des plates-formes dite *middleware* (en français intergiciel)

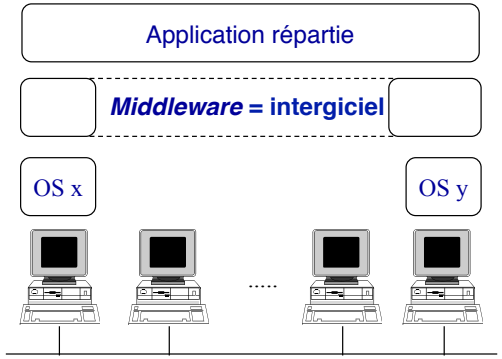


Introduction

Middleware

• masque hétérogénéité des machines et des systèmes
• masque répartition des traitements et données
• fournit une interface aux applications (modèle de programmation + API)

« *Middleware is everywhere* »
© IBM



Introduction

Middleware

En général, le middleware

• n'est pas visible par l'utilisateur final
• est un outil pour le développeur d'applications
• se retrouve enfoui dans les applications

Middleware permet de mettre en oeuvre des serveurs

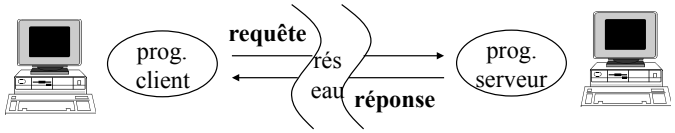
• à finalité fixe : serveur Web, serveur de fichiers, serveur de BD, ...
• effectuant des traitements quelconque : RMI, Java EE, .NET, Web Services, ...

Introduction

Problématique

Nombreux paradigmes de communication associés

⇨ le principal : interaction **requête/réponse** ou **client/serveur**



Interaction client/serveur

= 1 requête + 1 réponse
= demande d'exécution d'un traitement à distance et réponse
≈ appel procédural étendu au cas où appelant et appelé ne sont pas situés sur la même machine

Introduction

Problématique

Deuxième paradigme

⇨ interaction **par messagerie** (MOM : *Message-Oriented Middleware*)

Attention ≠ envoi message sur socket
Protocoles de niveau applicatif (pas transport) + propriétés (ex transactionnelles)

MOM : comm. **asynchrone** (fonctionnement client et serveur découplés)
Interaction client/serveur comm. **synchrone**

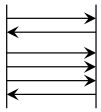
Introduction

Evolution du middleware

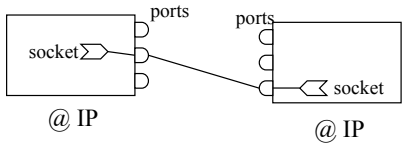
socket C, RPC, RPC objet, bus logiciel (CORBA)

Envoi de messages par socket

- primitives send & receive
- conception des progs cl et serv en fonction messages attendus et à envoyer



- socket : API C au-dessus des protocoles TCP & UDP
- fiabilité, ordre, ctrl de flux, connexion
- send/receive bloquant/non bloquant



Introduction

Remote Procedure Call (RPC)

- appel d'une procédure sur une machine distante
- groupement de 2 messages : appel & retour
- adressage : @IP + nom fonction

- définition des signatures des procédures
- compilateur de souches client et serveur

Exemple : RPC Sun

```
struct bichaine { char s1[80]; char s2[80]; };

program CALCUL {
        version V1 {
                int multpar2(int) = 1;
                string concat(struct bichaine) = 2;
                void plus_un() = 3;
        } = 1;
} = 0x21234567;
```


Introduction

RPC Objet

- mise en commun concepts RPC et prog objet
- appel d'une méthode sur un objet distant
- éventuellement +sieurs objets par machine
- adressage serveur de noms + nom logique

Exemple : Java RMI

```
import java.rmi.Remote;
import java.rmi.RemoteException;

interface CompteInterf extends Remote {
    public String getTitulaire() throws RemoteException;
    public float solde() throws RemoteException;
    public void deposer( float montant ) throws RemoteException;
    public void retirer( float montant ) throws RemoteException;
    public List historique() throws RemoteException;
}
```

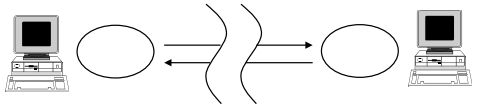

1. Modèle de programmation

1. Modèle de programmation

Modèle de programmation

2 programmes

- 1 programme client
- 1 programme serveur



⇨ **processus distincts**
⇨ **mémoires distinctes**
⇨ machines distinctes (sauf si répartition logique)

Selon le contexte 1 programme peut être client et serveur

- 1 programme client
- 1 **programme serveur** qui pour rendre le service **est client** d'un 3è programme
- 1 programme serveur

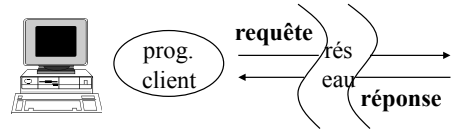
⇨ être client, être serveur, n'est pas immuable
mais **dépend de l'interaction considérée**

1. Modèle de programmation

Modèle de programmation

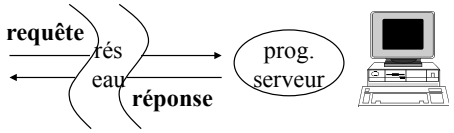
Point de vue du client

1. envoie une requête
2. attend une réponse



Point de vue du serveur

1. attend une requête
2. **effectue un traitement et produit une réponse**
3. envoie la réponse au client



mais le serveur doit aussi pouvoir traiter
les requêtes de **plusieurs clients**

1.1 Côté serveur

Plusieurs clients simultanément

1 bis. sélection de la requête à traiter
(FIFO ou avec priorité)

Plusieurs mises en oeuvre possibles pour le traitement de la requête

- 1 activité unique
- 1 activité par requête
- 1 *pool* d'activités

rq : activité = processus ou *thread*

1.1 Côté serveur

Plusieurs clients simultanément - 1 processus unique

```
while (true) { 1 2 3 }
```

Plusieurs clients envoient des requêtes simultanément
mais le serveur n'en traite qu'une à la fois

→ simple
→ pas de risque de conflit de concurrence
→ suffisant dans certains cas
(ex. 1 requête toutes les 10s qui demande 2s de traitement)

1.1 Côté serveur

Plusieurs clients simultanément - 1 processus par requête

Chaque arrivée de requête
déclenche la création d'un processus


```
while (true) { 1 fork() }
p1     p2     p3     ...
2 3    2 3    2 3    ...
```

→ les clients sont servis + rapidement
→ conflits éventuels en cas d'accès simultanés à une ressource partagée (ex. fichier)

Problème : une concurrence "débridée" peut écrouler la machine
→ restreindre le nombre de processus traitants

1.1 Côté serveur

Plusieurs clients simultanément - *pool* de processus

• *pool* fixe
• *pool* dynamique

Pool fixe

• 1 nombre constant de processus
• 1 processus qui reçoit les requêtes et les dispatche aux processus du *pool*
• si aucun processus n'est libre, les requêtes sont mises en attente

Avantage
- pas de risque d'écroulement
  (pour peu que le nombre de processus soit correctement dimensionné)

Inconvénients
- un *pool* de processus inactifs consomme des ressources inutilement
- les pointes de traffic sont mal gérées

1.1 Côté serveur

Plusieurs clients simultanément - *pool* de processus

Pool dynamique

Toujours avoir des processus prêt sans surcharger
inutilement la machine

→ le nombre de processus varie
→ mais le nombre de processus prêts est toujours compris entre 2 bornes
→ mélange des politiques 1 proc/req et *pool* fixe

• nb max de proc (ex. 150)
• nb max de proc inactifs (ex. 20) : au delà on détruit les proc
• nb min de proc inactifs (ex. 5) : en deça on crée de nouveaux proc
• nb proc créés au départ (ex. 15)

rq : solution retenue par Apache

1.2 Communications

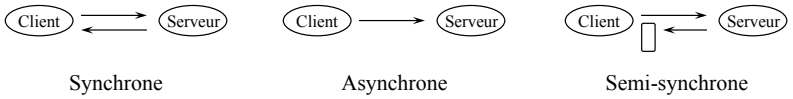
Modèle de programmation

1.1 Côté serveur

1.2 Communications client/serveur
- Modes
- Concepts
- Notion de connexion
- Gestion d'états
- Représentation des données
- Passage de paramètres
- Traitement des pannes

1.2 Communications

Modes de communication

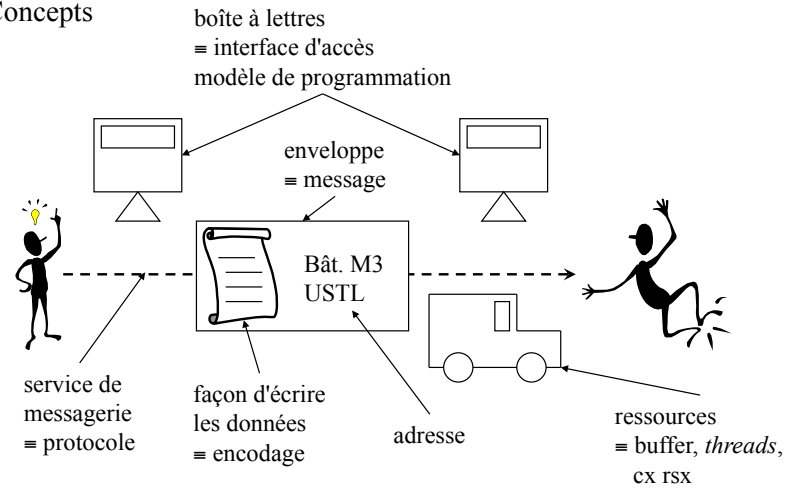


Synchrone Asynchrone Semi-synchrone

- **Synchrone** le client attend la réponse pour continuer son exécution
- **Asynchrone** le client n'attend pas de réponse et continue tout de suite
- **Semi-synchrone**
  le client continue son exécution après l'envoi et
  récupère le résultat ultérieurement
  - à futur explicite — le résultat est stocké dans une boîte à lettres (BAL) le client consulte cette BAL pour récupérer le res.
  - à futur implicite — le résultat est fourni automatiquement au client par ex. via une variable du prog. client

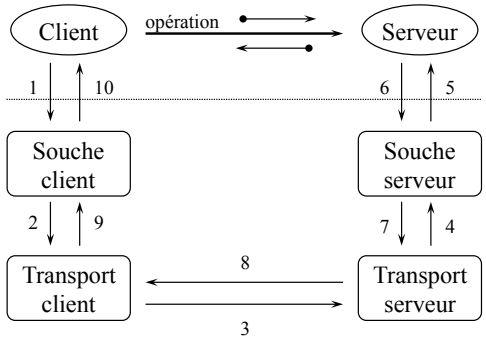
1.2 Communications

Concepts



1.2 Communications

Mise en oeuvre des concepts



1. Appel local
2. Préparation du message d'appel
2. Envoi
4. «Upcall» vers la souche serveur
5. Décodage du message
6..10 : Chemin inverse

Vocabulaire français souche ou talon
Vocabulaire anglais souche cl. = *stub* ou *proxy*, souche serv. = *skeleton*

1.2 Communications

Connexion

Problématique

Délimitation des communications entre un client et un serveur

Mode non connecté (le + simple)
- les messages sont envoyés "librement"
- example : NFS

Mode connecté
- les messages sont
  - précédés d'une ouverture de connexion
  - suivis d'une fermeture de connexion
- facilite la gestion d'état
- permet un meilleur contrôle des clients
- ex : FTP, Telnet, SMTP, POP, JDBC, HTTP

Rq : la cx est le + souvent liée au transport (TCP) plutôt qu'au protocole applicatif lui-même

Gestion d'états du protocole de communication

Problématique

2 requêtes successives d'un même client sont-elles indépendantes ?
→ faut-il sauvegarder des infos entre 2 requêtes successives d'un même client ?

Mode sans état (le + simple)

- pas d'info sauvegardée
- les requêtes successives d'un même client sont indépendantes
- ex : NFS, HTTP

Types de requêtes envisageables
- demande d'écriture du **k-ième** bloc de données d'un fichier

Types de requêtes non envisageables
- demande d'écriture du bloc **suivant**

Gestion d'états du protocole de communication

Mode avec état

- les requêtes successives s'exécutent
  en fonction de l'état laissé par les appels précédents
  → sauvegarde de l'état

Notion proche : session cliente dans un serveur Web
Suivi de l'activité d'un client entre le moment où il arrive et celui où il quitte le site

Pb : y a-t-il un mécanisme explicite qui indique au serveur que le client part ?
- si oui (ex. déconnexion notifiée au serveur) alors ok
- si non
  pb : aucun sens de conserver *ad vitam* les données de la session
  heuristique : la session expire au bout d'un délai fixé
  inconv. : un client très lent peut revenir après expiration
  → (re)commencement d'une nouvelle session

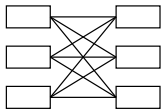
Représentation des données

Problématique

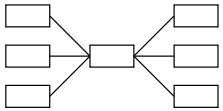
Comm. entre machines avec des formats de représentation de données ≠
→ pas le même codage (*big endian* vs *little endian*)
→ pas la même façon de stocker les types (entiers 32 bits vs 64 bits, ...)

2 solutions

On prévoit tous les cas de conversions possibles
(n^2 convertisseurs)



On prévoit un format pivot et on
effectue 2 conversions (2n convertisseurs)



Nbreux formats pivots : ASN.1, Sun XDR, sérialisation Java, CORBA CDR, ...

Passage de paramètres

Problématique

Client et serveur ont des espaces mémoire ≠
→ passage par valeur ok
→ passage par référence
  pas possible directement
  une ref. du client n'a aucun sens pour le serveur (et vice-versa)

Solution : mécanisme copie/restauration

1. copie de la valeur référencée dans la requête
2. le serveur travaille sur la copie
3. le serveur renvoie la nouvelle valeur avec la réponse
4. le client met à jour la réf. avec la nouvelle valeur

Passage de paramètres

Mais copie/restauration pas parfait

Problème du double incrément

```
void m(&x, &y) { x++; y++; }
a=0; m(a,a);
```

résultat attendu a=2, résultat obtenu a=1 !!

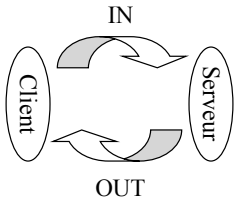
⇨ détecter les doubles (multiples) référencements
⇨ pas facile dans le cas général

Problème en cas de mises à jour concurrentes de la valeur référencée

Passage de paramètres

Définitions couramment adoptée (au lieu de valeur/référence)

- mode IN (entrée)	passage par valeur avec la requête
- mode OUT (sortie)	passage par valeur avec la réponse
- mode IN/OUT (entrée/sortie)	copie/restauration de la valeur



• IN si le serv. modifie la valeur, le cl. ne "voit" pas cette modif.
• OUT si le cl. transmet une valeur, le serv. ne la "voit" pas

Traitement des pannes

Dans la majorité des cas

• symptôme : absence de réponse
• cause inconnue : réseau ? client ? serveur ?

Techniques **logicielles** de détection des pannes

• *heart beat*	périodiquement le serveur signale son activité au client
• *pinging*	périodiquement le client sonde le serveur qui répond

➢ les résultats ne sont jamais sûr à 100%
➢ périodicité délicate à régler
➢ impossibilité de distinguer une “vraie” panne d’un ralentissement dû à surcharge
➢ pas une vraie détection, possibilité de fausse détection
➢ on parle de **suspicion de panne**

Traitement des pannes

Comportement possible en présence de pannes

client envoie requête
si panne signalée par détecteur
alors signaler la panne au client

➢ la requête s’exécute (si pas de panne), sinon elle ne s’exécute pas
➢ comportement dit “au plus 1 fois” (0 fois ou 1 fois)

Traitement des pannes

2ème comportement possible en présence de pannes : “au moins 1 fois”
(1 fois ou n fois)

client envoie requête
tant que résultat non reçu
  attendre délai // éventuellement attente interrompue par détecteur
  renvoyer requête

- tentatives de réémissions pour compenser les pertes de message
- en cas de fausse détection de panne
  le message est reçu +sieurs fois ⇨ le traitement s’exécute +sieurs fois

ok si idempotent
i.e. plusieurs exécutions du même traitement ne doivent pas poser problème

idempotent	x:=5	lire_fichier(bloc k)	ecrire_fichier(bloc k)
¬idempotent	x++	lire_fichier_bloc_suivant()	

Traitement des pannes

3ème comportement possible en présence de pannes

idem “au moins 1 fois”
+ numérotation des messages
en vue de détecter (côté serveur) les réémissions

- le traitement s’exécute exactement 1 fois

Université Lille 1
Master mention Informatique – M1

Construction d'applications réparties

II. Applications réparties en mode message

Lionel.Seinturier@univ-lille1.fr

Plan

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 (aussi appelé synchrone)
- non bloquants (aussi appelé asynchrone)

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.lifl.fr");
InetAddress[] host = InetAddress.getAllByName("www.lifl.fr");
```

Méthodes principales

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

1.2 TCP

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

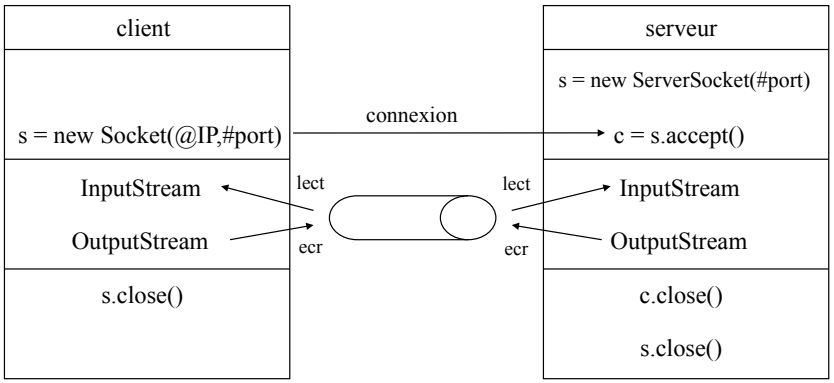
trop de pertes de messages ou réseau trop encombré
→ connexion perdue

Utilisation

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

1.2 TCP

Fonctionnement



Fonctionnement

- serveur crée une *socket* et attend une demande de connexion
- client envoie une demande de connexion
- serveur accepte connexion
- dialogue client/serveur en mode flux
- fermeture de connexion à l'initiative du client ou du serveur

API `java.net.Socket`

Constructeur : adresse + n° port

```
Socket s = new Socket("www.lifl.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()`

API `java.net.Socket`

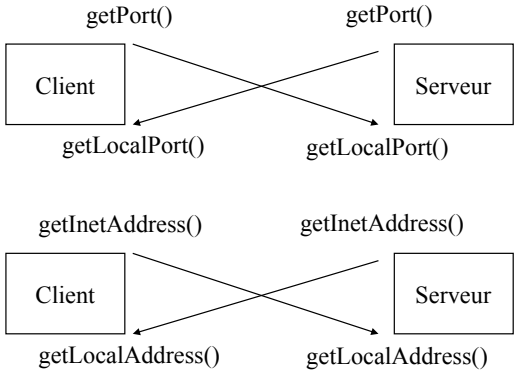
Options TCP : timeOut, soLinger, tcpNoDelay, keepAlive

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

1.2 TCP

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`

1.2 TCP

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)`

1.2 TCP

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

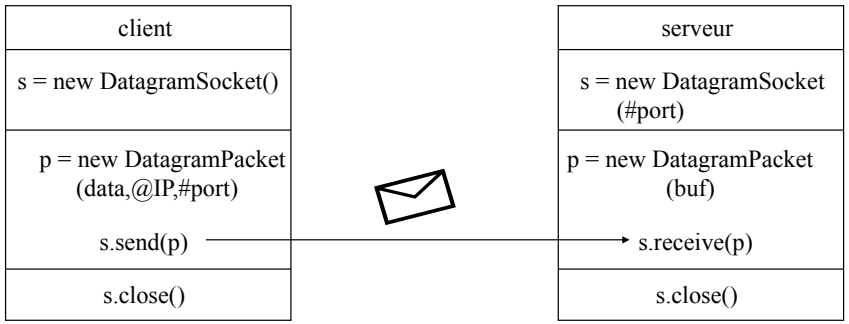
⇨ + performant que TCP
⇨ - de qualité de service (fiabilisation peut être "reprogrammée" au niveau applicatif)

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

1.3 UDP

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	perte de messages possible
pas de contrôle de flux	ordre des messages non garanti
pas de connexion	

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, ...

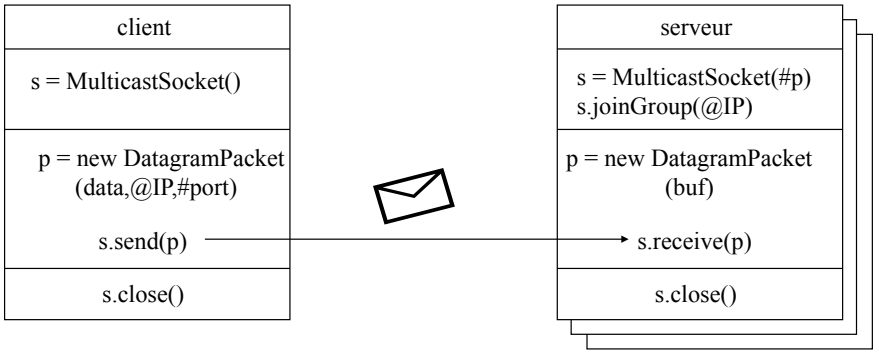
Certaines @ classe D sont assignées
(voir `http://www.iana.org/assignments/multicast-addresses`)

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 noeuds 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)

2.2 Modèle de programmation

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 def. dans la méthode `run()`

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


2.2 Modèle de programmation

Modèle de programmation

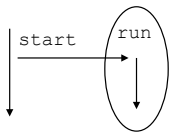
Lancement d'un thread : appel à la méthode `start()`

```
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*

2.2 Modèle de programmation

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) !!

2.2 Modèle de programmation

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

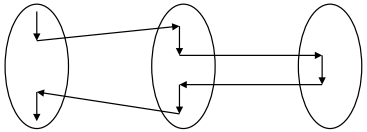
```
public class Foo implements Runnable {
    int p1;
    Object p2;
    public Foo(int p1,Object p2) {this.p1=p1; this.p2=p2; }
    public void run() { ... p1 ... p2 ... }
}

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() { ... }
}
```

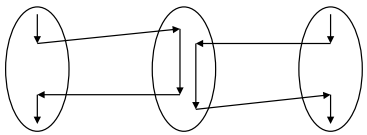
⇨ la construction d'un objet lui assigne des instructions à exécuter

2.3 Synchronisation

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)



2.4 Exclusion mutuelle

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

2.4 Exclusion mutuelle

Exclusion mutuelle

Autre besoin : bloc de code (∈ 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

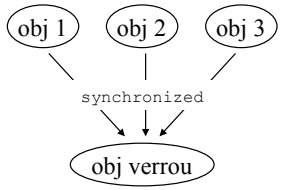
- JVM garantit atomicité d'accès au byte, char, short, int, float, réf. d'objet
  !! pas long, ni double !!
- coût
  appel méthode `synchronized` ≈ 4 fois + long qu'appel méthode "normal"
  ⇨ à utiliser à bon escient

2.4 Exclusion mutuelle

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) {...} }
```


2.6 Compléments

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

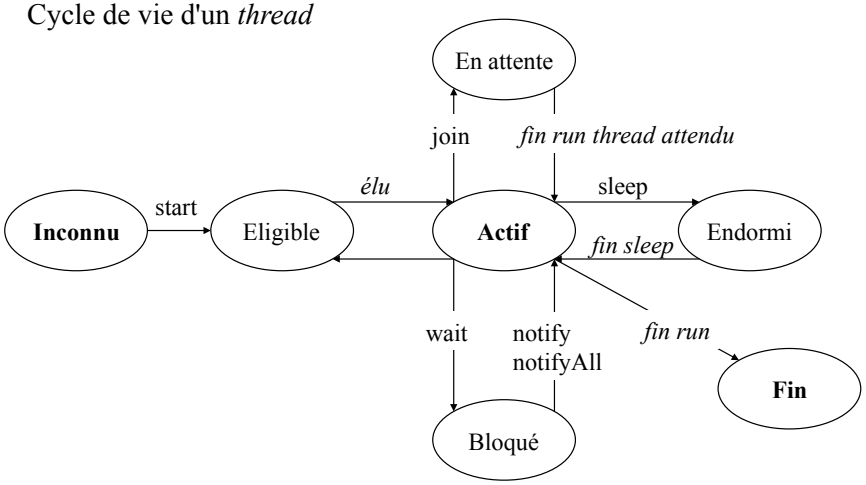
```
public class AsyncInputStream
  extends FilterInputStream implements Runnable {
  int[] buffer = ...   // zone tampon pour les données lues
  AsyncInputStream( InputStream is ) {
    super(is); new Thread(this).start(); }
```

```
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
}
```


2.6 Compléments

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