USTL – Master mention Informatique – M1

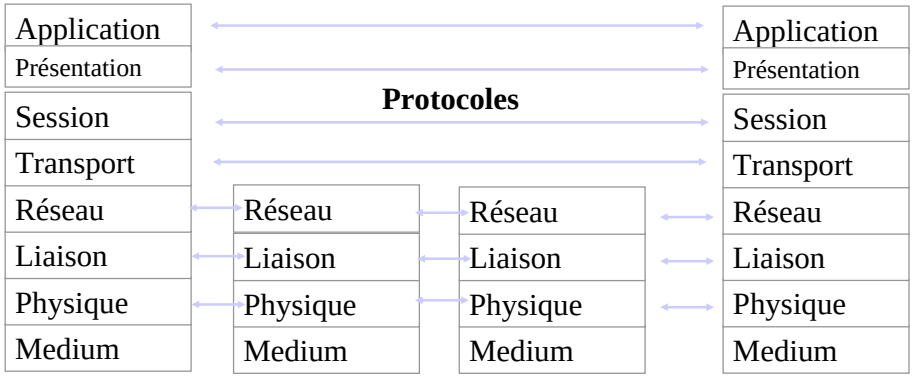
Construction d'applications réparties

II. Applications réparties en mode message

Lionel.Seinturier@lifl.fr

Plan

1.1 Réseau

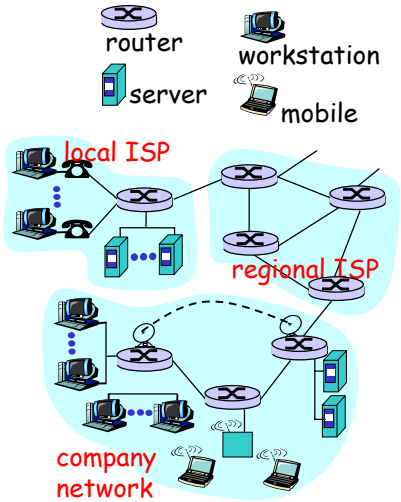


Internet

Des millions de machines
- Stations de travail, serveurs, portable, PDA, …
- qui exécutent des app. réparties

Des liaisons
- filaire, radio, satellite

Des routeurs
- transmettent les paquets (datagrammes) à travers le réseau



Internet

Application : applis supportées par le réseau
- http, smtp, ftp

Transport : transfert de bout en bout
- tcp, udp

Réseau : routage des datagrammes de la source vers la destination
- ip, protocoles de routage

Liaison : transfert de données entre éléments voisins
- ppp, ethernet

Physique : bits sur le câble

Application
Transport
Réseau
Liaison
Physique

Notion de protocole

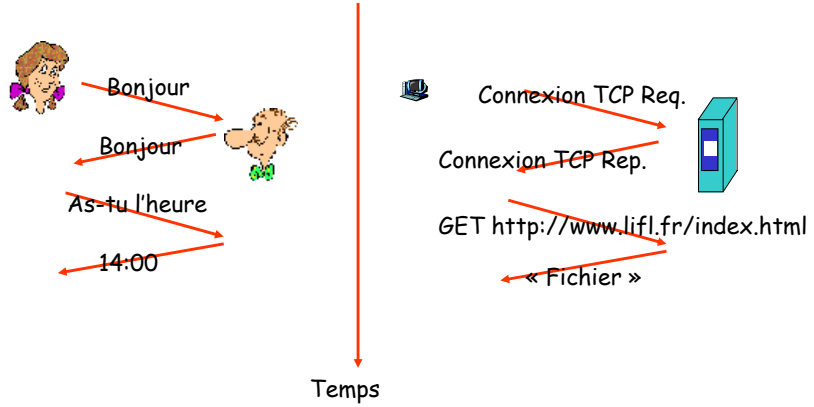
Des protocoles humains
- Quelle heure est-il ?
- J'ai une question…
- Introductions

Des protocoles réseaux
- Des machines plutôt que des humains
- Toutes les activités de communication dans l'internet sont gouvernées par des protocoles

- Envois de messages spécifiques
- Actions spécifiques exécutées lorsque des messages sont reçus, etc.

Les protocoles definissent le format, l'ordre des messages envoyés et reçus à travers des entités réseaux, et les actions à entreprendre lors de la réception des messages

Notion de protocole



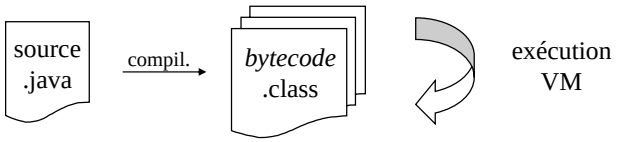
Autres protocoles humains ?

1.2 Java

Objectifs de Java

• un langage simple, OO, portable
• philosophie WORA ("Write Once, Run Anywhere")
1 source s'exécute **à l'identique** sur +sieurs CPU/OS
(Solaris, Linux, Win xx, MacOS) pourvu d'une **machine virtuelle** (VM)

Principe



∀ plate-forme, à partir d'un .java

- .class obtenu par compilation identique
- .class s'exécute à l'identique

Contrepartie

.class - performant code natif (.exe)
mais ...

1.2 Java

Historique

1991 **J. Gosling**, B. Joy, A. Van Hoff : OAK (électronique grand public)
1993 JDK 1.0, lien avec Web (applet)
1996 Java Beans
1997 JDK 1.1, Enterprise Java Beans, JavaCard, Java OS
1998 Java 2 (JDK 1.2, 1.3, 1.4)
2004 Java 5 (JDK 1.5) : annotations, types génériques, *enum*, *auto-boxing*

En cours de développement

2006 Java 6 (beta 2) : amélioration API, intégration SGBD
2007 ? Java 7 : VM multi-langage Java, JavaScript, PHP, Python, …

Nombreuses technologies pour les applications client/serveur

JSP, servlet, JDBC, JMS, JavaIDL, JavaMail, RMI, JCE, JAAS, JNDI, JTS, JTA, JavaCard, JMX, JMI, ...

1.2 Java

Disponibilité

3 éditions

Java SE (Standard Edition)	"le" JDK (compilateur + VM + librairies)
Java EE (Enterprise Ed.)	serveur pour des applications d'entreprise client/serveur (nécessite Java SE)
Java ME (Micro Ed.)	VM + librairies pour des systèmes embarqués (PDA, tel., ...)

Outils

- IDE Eclipse, JBuilder, VisualAge, JDeveloper, Forte, NetBeans, ...
- VM Jikes RVM, Kaffe, kissme, LaTTe, SableVM, ...
- compilateur Jikes, GCJ, KJC, Espresso, Manta, …
- manipulation *bytecode* ASM, BCEL, BAT, CFParse, Jasmin, Javassist, JMangler, …
- ... nombreux autres

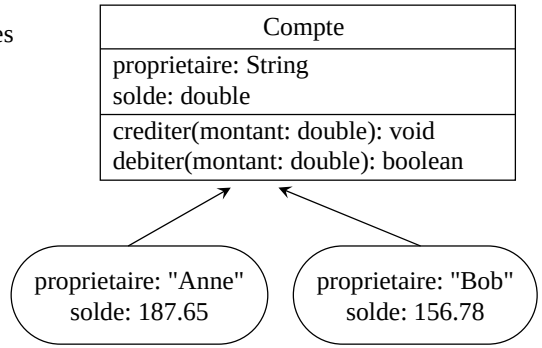
1.2 Java

Concepts de base

classe/objet/méthode/donnée – héritage – interface – exception

Classe : élément logiciel modélisant un type d'entité du problème à modéliser
comprend
- des données ≈ variables
- des méthodes (code) ≈ procédures

Objet : élément logiciel créé
à partir d'une classe
créer un objet = instancier
objet = instance



1.2 Java

Concepts de base

classe/objet/méthode/donnée – héritage – interface – exception

```
public class Compte {
   private String proprietaire;
   private double solde;

   public void crediter( double montant ) { solde+=montant; }
   public boolean debiter( double montant ) {
        if ( solde >= montant }
                { solde-=montant; return true; }
        else return false;
   }
}

public class TestCompte {
   public static void main( String[] args ) {
        Compte bob = new Compte();    // instanciation
        bob.crediter(156.78);
}  }
```

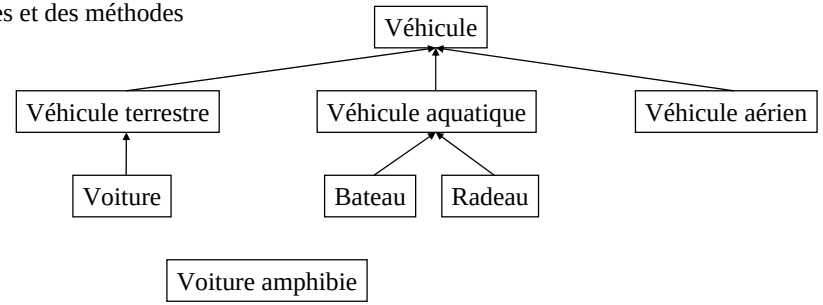
1.2 Java

Concepts de base

classe/objet/méthode/donnée – héritage – interface – exception

Héritage : relatation de spécialisation/généralisation entre classes
⇨ hiérarchie de classes

La classe fille hérite des
données et des méthodes

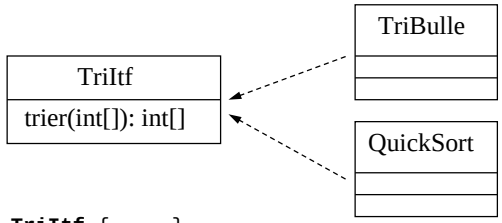


1.2 Java

Concepts de base

classe/objet/méthode/donnée – héritage – interface – exception

Interface : point d'accès
= ensemble de déclaration de méthodes
⇨ pas de code
⇨ permet de découpler la définition des services autorisés
de la façon dont ils sont implémentés




```
public interface TriItf {
  int[] trier( int[] tab );
}

public class TriBulle implements TriItf { ... }
public class QuickSort implements TriItf { ... }
```


1.2 Java

Concepts de base

classe/objet/méthode/donnée – héritage – interface – exception

Vocabulaire : implémenter (implanter) = associer du code aux définitions de l'interface

- 1 interface peut être implémentée par +sieurs classes
- 1 classe peut implémenter +sieurs interfaces

≠ interface – implémentation fondamentale en client/serveur
⇨ masquer la façon dont est réalisé un service

liens avec la notion d'API (*Application Programming Interface*)

1.2 Java

Concepts de base

classe/objet/méthode/donnée – héritage – interface – exception

Mécanisme de retour anticipé d'une méthode pour signaler un comportement anormal (division par zéro, disque plein, pb réseau, …)

```
public class Serveur {
   public void m() {
        // ...
        throw new Exception("Erreur");      // levée de l'exception
        // ...
}  }

public class Client {
   public static void main( String[] args ) {
        Serveur s = new Serveur();
        try { s.m(); }               // récupération de l'exception
        catch( Exception e ) { System.out.println("problème"); }
}  }
```

1.2 Java

FAQ

Qu'est-ce qu'un CLASSPATH ?
liste de répertoires dans lesquel la MV Java recherche les .class

Que fait la commande : `javac -d classes src/monpackage/MaClasse.java`
compile le fichier `src/monpackage/MaClasse.java`
et place le .class dans le répertoire `classes`

Que fait la commande suivante : `java monpackage.MaClasse`
exécute le programme `monpackage.MaClasse`

Qu'est-ce qu'une classe ? une méthode ? un champ ?
classe = entité du langage modélisant un concept
méthode = procédure contenant du code et associée à une classe
champ = variable associée à une classe

A quoi correspond la ligne suivante : `public class MaClasseImpl extends Base`
définit la classe `MaClasseImpl` qui hérite de la classe `Base`

1.2 Java

FAQ

Que fait la ligne de code suivante : `String salutation = "hello "+"world";`
crée une chaîne `salutation` qui contient *hello world*

Comment affiche-t-on un message à l'écran ?
```
System.out.println("message ...");
```

Comment programme-t-on en Java une boucle `for` ? un `if` ?
```
for ( int i=0 ; i<255 ; i ++ ) { ... }
if (cond) { ... } else { ... }
```

Que fait l'instruction `return` ? l'instruction `System.exit(1)` ?
`return` : fin de la méthode courante
`System.exit(1)` : fin du programme

Qu'est-ce qu'une exception ?
mécanisme de retour anticipé d'une méthode pour signaler un comportement anormal (division par zéro, disque plein, pb réseau, …)

2. Programmation réseau

2.1 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

2.1 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 ?

2.1 Notions générales

Caractéristiques des protocoles de transport réseaux

2 modes

- synchrone les primitives sont bloquantes
- asynchrone les primitives sont non bloquantes

Exemple

- send sync et receive sync (Java)
  send reste bloqué jusqu'à l'envoi complet du message
  receive reste bloqué jusqu'à ce qu'il y ait un message à lire

	sync	async
send		
receive		

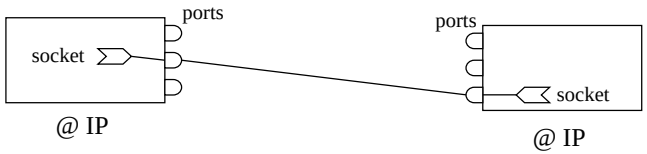
Asynchrone + souple
Synchrone programme + simple à écrire

→ receive synchrone + multi-threading ≈ receive asynchrone

2.1 Notions générales

Adressage

socket = abstraction des extrémités servant à communiquer
chaque socket *associée* à @ IP + n° port local



1 récepteur pour chaque port/machine
éventuellement +sieurs émetteurs vers le même port/machine

2.1 Notions générales

Adressage

Chaque carte rsx (connectée de façon fixe et directe) = 1 point d'accès au rsx
= 1 @ IP permanente

- +sieurs @ IP par machine
- @ IP = 32 bits (128 pour IPv6)

4 classes d'adresses IP

- A : 128 rsx, 16 M @ par réseau
- B : 16 K rsx, 64 K @ par réseau
- C : 2 M rsx, 256 @ par réseau
- D : @ réservées pour la diffusion (Multicast IP)

Correspondance @ symbolique ↔ @ IP assurée par DNS
ex. : www.lifl.fr ↔ 132.227.60.13

2.1 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()`

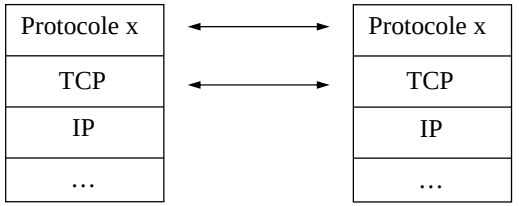
2.1 Notions générales

Problématique de la construction de protocoles applicatifs

TCP, UDP : socles pour la construction de protocoles de + haut niveau

Définition de protocole

- message + paramètres
- format des messages
- enchaînement des messages
- cas d'erreur : message, format, paramètres, enchaînement



!! distinction entre niveaux !!
⇨ utilisation des services du protocole sous-jacent
ex. : ouverture de connexion TCP

2.1 Notions générales

Connexion

Problématique

Les communications entre un client et un serveur sont-elles précédées d'une ouverture de cx ? (et suivies d'une fermeture)

Mode non connecté (le + simple)

- les messages sont envoyés "librement"

Mode connecté

Avantages
- facilite la gestion d'état
- meilleur contrôle des clients (arrivées & départs)

2.1 Notions générales

Connexion

niveau transport et/ou applicatif

	Transport (niveau 4)	Applicatif (niveau 7)
Connecté	TCP	FTP, Telnet, SMTP, POP, JDBC
Non connecté	UDP	HTTP, NFS, DNS, TFTP

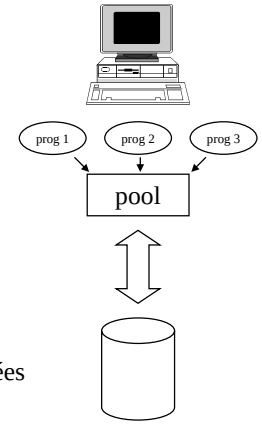
- purement non connecté : NFS+UDP
- purement connecté : FTP+TCP
- mixte : HTTP+TCP

Rq : HTTP
- cx lié au transport (TCP)
- pas à HTTP lui-même
- mécanisme session

⇨ arrivées & départs

2.1 Notions générales

Pool de connexion



Connexions réseau coûteuse
• en ressources occupées
• en temps mis pour ouvrir/fermer les connexions

But du *pool* (en français réserve)
mutualiser les connexions entre 2 machines
ex. : *pool* de connexions vers un SGBD

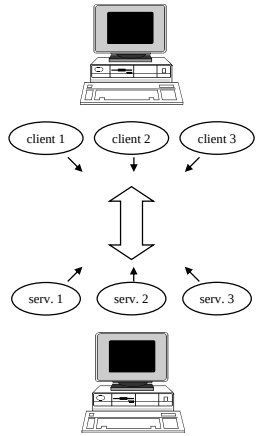
• tous les programmes n'ont pas forcément besoin de toutes les connexions en même temps
• le *pool* peut restreindre ou adapter le # de connexions simultanées

Si les progs ont besoin de plusieurs connexions

⇨ politique de gestion de ressources partagées
⇨ problèmes "classiques" : réservation, interblocage, …

2.1 Notions générales

Multiplexage de communications



Même objectif que le *pool*
1 seule connexion partagée entre plusieurs programmes c/s

Pb : distinguer les ≠ flux de données
⇨ les encadrer par un protocole de multiplexage

Mécanisme pouvant être couplé avec le *pool*

2.1 Notions générales

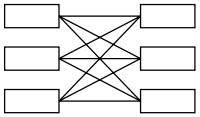
Représentation des données

Problématique

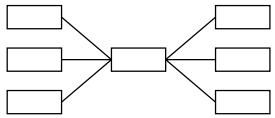
Communications entre machines avec des formats de représentation de données ≠
→ pas le même codage (*big endian* vs *little endian*)
→ pas la même façon de stocker les types (entiers 32 bits vs 64 bits, ...)

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, ...

2.1 Notions générales

Traitement des pannes

3 types

• client
• serveur
• réseau - panne de rsx local en général détectable (ex. brin Ethernet)
- panne rsx large échelle (Internet) non signalée généralement

Dans la majorité des cas

• symptôme : absence de réponse
• cause (rsx, serveur planté, serveur très lent) **inconnue**
• solution
- choisir un autre serveur : pas toujours possible
- abandonner : pas forcément satisfaisant
- réessayer plus tard (en espérant un retour en service) ⇨ mieux

2.1 Notions générales

Traitement des pannes

Problématique

Comportement de l'interaction c/s satisfaisant en présence de ré émissions
(≈ appel de méthode local)
sans que cela soit trop lourd à gérer

⇨ Notion de **sémantique d'invocation**

- au moins 1 fois garantie traitement demandé exécuté au moins 1 fois [1..n]
- au plus 1 fois [0..1]
- exactement 1 fois
  - le plus satisfaisant
  - le plus lourd

2.1 Notions générales

Traitement des pannes

Au moins 1 fois

Il faut que le traitement demandé soit idempotent
ie +sieurs exécutions du même traitement ne doivent pas poser problèmes

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

Algorithme

client envoie requête
tant que résultat non reçu
    attendre délai
    renvoyer requête

2.1 Notions générales

Traitement des pannes

Au plus 1 fois

Algorithme

client envoie requête
si au bout d'un délai d'attente pas de résultat
alors signaler la panne au client

Exactement 1 fois (le + dur à assurer)

On ne sait pas si l'absence de réponse est due
- à une perte de message sur le réseau
- à une panne de serveur
- à un serveur très lent

⇨ ré émissions (requête et/ou réponse)
⇨ + détection des ré émissions

2.1 Notions générales

Traitement des pannes

Exactement 1 fois

ex : client renvoie sa requête à l'expiration du délai

- serveur planté définitivement
  en général, pas de réessai ad vitam → abandon au bout de 3 envois
- serveur planté mais redémarre
  est-ce que la requête a quand même été reçue ?
  → si oui continuer l'exécution
  est-ce que la réponse avait été envoyée mais est perdue ?
  → si oui renvoyer la réponse, si non exécuter
- serveur très lent
  donc requête bien reçue et traitement commencé
  → continuer l'exécution
- réseau coupé définitivement (idem serveur planté définitivement)
- réseau coupé mais reprend (idem serveur planté mais redémarre)

2.1 Notions générales

Détection des pannes

Mécanisme complémentaire pour détecter des pannes
en cours d'exécution d'un traitement

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

2.2 TCP

Propriétés du protocole TCP

Taille des messages
- quelconque
- envoi en général bufférisé
- vidage autoritaire des buffers possible (dépend OS)

Perte de messages
- acquittements (masqués au programmeur) de messages envoyés
- timeout de ré-émission des messages en cas de non récept. de l'acq.
- (mécanisme de fenêtre)

Contrôle de flux
- éviter qu'un émetteur trop rapide fasse "déborder" le buffer du récepteur
- blocage de l'émetteur si nécessaire

Ordre des messages
- garantie que l'ordre d'émission = ordre de réception
- garantie de non duplication des messages

2.2 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, ...

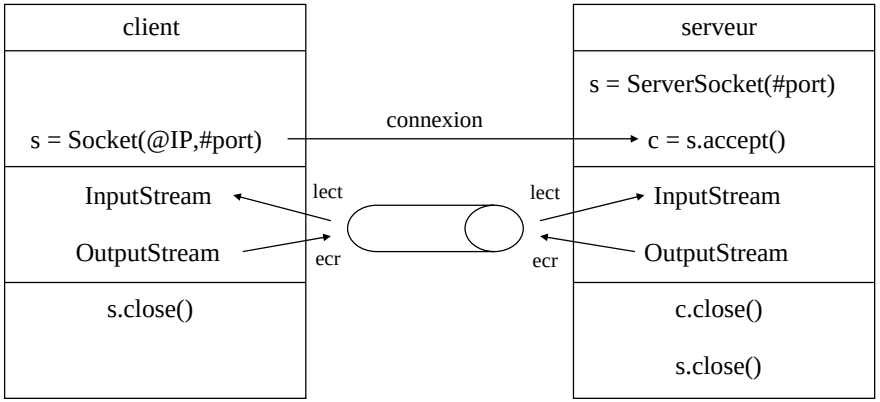
2.2 TCP

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

2.2 TCP

Fonctionnement



2.2 TCP

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()`

2.2 TCP

API `java.net.Socket`

Options TCP : timeOut, soLinger, tcpNoDelay, keepAlive

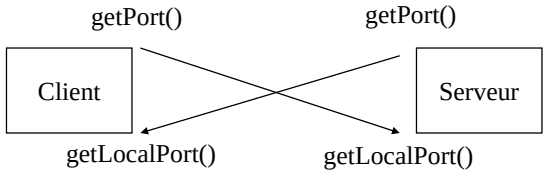
setSoTimeout(int)

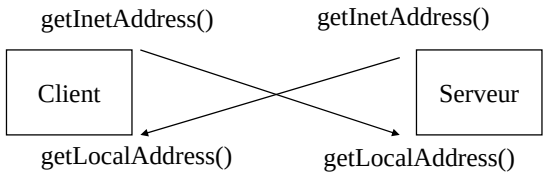
int = 0	read bloquant (à l'∞) tant qu'il n'y a pas de données à lire
int > 0	délai max. de blocage sur un read passé ce délai, exception `SocketTimeoutException` levée (la socket reste néanmoins opérationnelle)

2.2 TCP

API `java.net.Socket`

Symétrie des valeurs retournées lorsque 2 *sockets* sont connectées





2.2 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

2.2 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 lev`

→ `java.nio` à partir JDK 1.4 : `ServerSocket.getChannel()`
retourne une instance de `ServerSocketChannel`
qui implance une méthode `accept()` non bloquante

2.2 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`

2.2 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)`

2.2 TCP

Personnalisation du fonctionnement des *sockets*

Sous-classage

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

2.3 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 : pas 1 pb pour certaines applications (*streaming* audio, vidéo, ...)

Pas de contrôle de flux

Ordre des messages non garanti

Pas de connexion

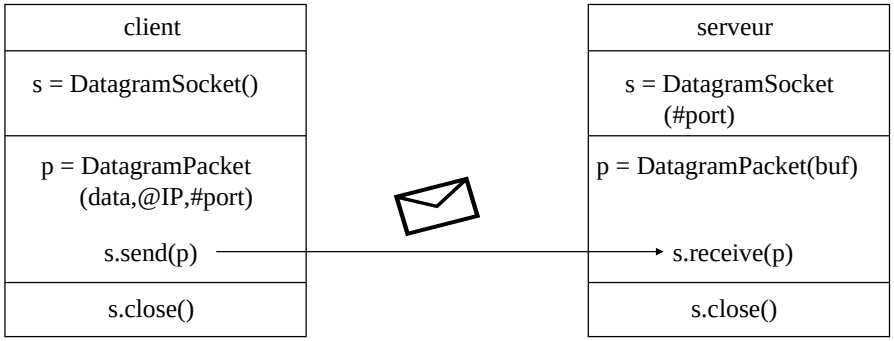
⇨ + performant que TCP
⇨ - de qualité de service (fiabilisation peut être "reprogrammée" au niv. applic.)

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

2.3 UDP

Fonctionnement

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



2.3 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

2.3 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

2.4 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	

2.4 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, ...

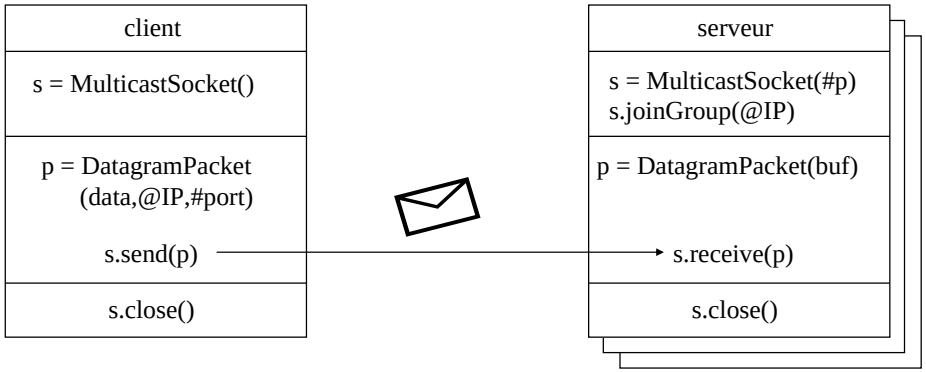
Certaines @ classe D sont assignées
(voir `http://www.iana.org/assignments/multicast-addresses`)

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
```


2.4 Multicast IP

Fonctionnement

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



API `java.net.MulticastSocket`

Constructeur	MulticastSocket(port) MulticastSocket() (+ ...)	sur #port 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

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

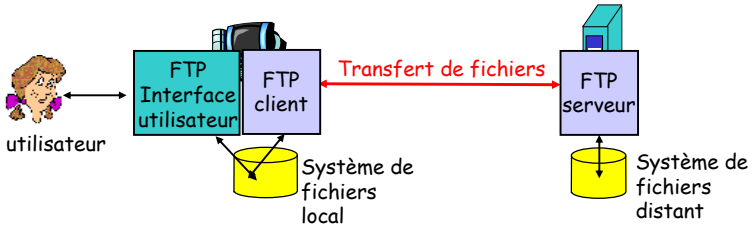
- fragmentation/recomposition messages > 64 K
- ordre garanti des messages
- notifications d'arrivées, de retraits de membres
- organisation arborescente des noeuds de diffusion

2.5 Les applis C/S Internet

FTP : File Transfer Protocol
SMTP : Simple Mail Transfer Protocol
DNS : Domain Name System

et HTTP ?... dans le cours suivant...

2.5.1 ftp : le protocole de tranfert de fichiers



Transfert de fichiers de/vers une machine hôte
Modèle client/serveur
- *client:* côté qui initie le transfert
- *server:* hôte distant

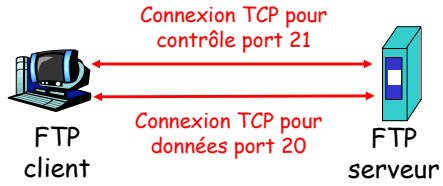
ftp: RFC 959
ftp serveur: port 21

2.5.1 ftp: contrôle et données sur des connections séparées

Le client ftp contacte le serveur ftp sur le port 21, en indiquant qu'il s'agit d'une connexion TCP

Deux canaux en // sont ouverts:

- contrôle: échange de commandes, réponses entre client et server.
- Données : transfert de données de/vers le serveur



Le serveur ftp maintient un état vis-à-vis du répertoire courant du client, authentification

2.5.1 commandes & réponses ftp

Les commandes:

Emises sous forme ASCII sur le canal de contrôle

USER *username*

PASS *password*

LIST retourne la liste de fichiers du répertoire courant

RETR filename demande (gets) de fichier

STOR filename stocke (puts) un fichier sur la machine distante

Les codes retour

Code d'état et texte

331 Username OK, password required

125 data connection already open; transfer starting

425 Can't open data connection

452 Error writing file

2.5.2 Courrier électronique

Trois composants majeurs:

Agents utilisateurs

Serveurs de messagerie

Le protocole : SMTP

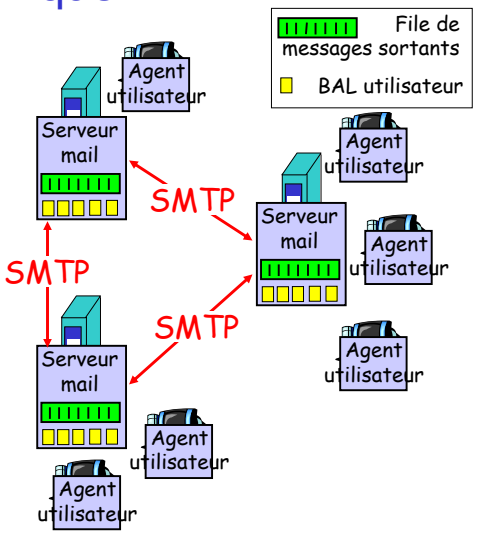
L'agent utilisateur

Un lecteur de message

Composition, lecture, édition de messages électroniques

e.g., Eudora, Outlook, elm, Netscape Messenger

Gère les messages entrants et sortants stockés sur le serveur



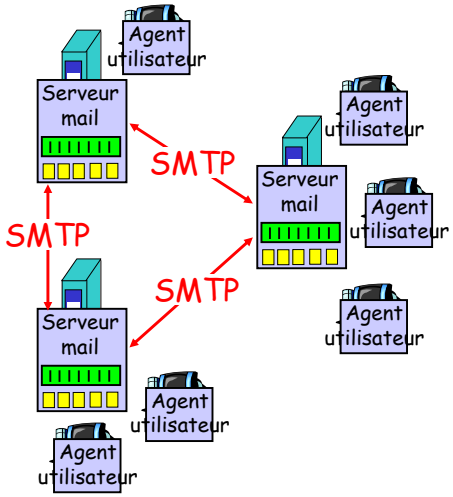
2.5.2 Courrier éléctronique : les serveur de messagerie

Serveurs de messagerie

BAL contient les messages entrants (à lire) pour l'utilisateur

File de messages (messages sortants) à envoyer

Protocole smtp entre serveurs de messagerie pour envoi de messages



2.5.2 Courrier électronique: SMTP [RFC 821]

utilise tcp pour tranférer des messages entre serveurs de messagerie sur le port 25

Transfert direct: serveur émetteur vers serveur récepteur

trois phases

- connexion
- transfert de messages
- déconnexion

Interaction commande/réponse interaction

- commande: texte ASCII
- réponse: code état et phrase

messages doivent être en 7-bit ASCII

2.5.2 Une simple interaction smtp

```
S: 220 hamburger.edu
C: HELO crepes.fr
S: 250  Hello crepes.fr, pleased to meet you
C: MAIL FROM: <alice@crepes.fr>
S: 250 alice@crepes.fr... Sender ok
C: RCPT TO: <bob@hamburger.edu>
S: 250 bob@hamburger.edu ... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: Do you like ketchup?
C:   How about pickles?
C: .
S: 250 Message accepted for delivery
C: QUIT
S: 221 hamburger.edu closing connection
```


2.5.2 Le format des messages

SMTP (RFC 821): protocole pour l'échange de messages

+ RFC 822: standard définissant le format des messages :

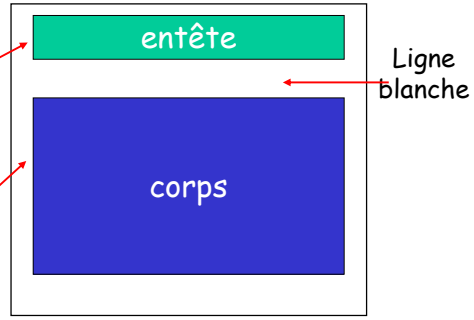
entête, e.g.,

- To:
- From:
- Subject:

different des commandes smtp!

corps

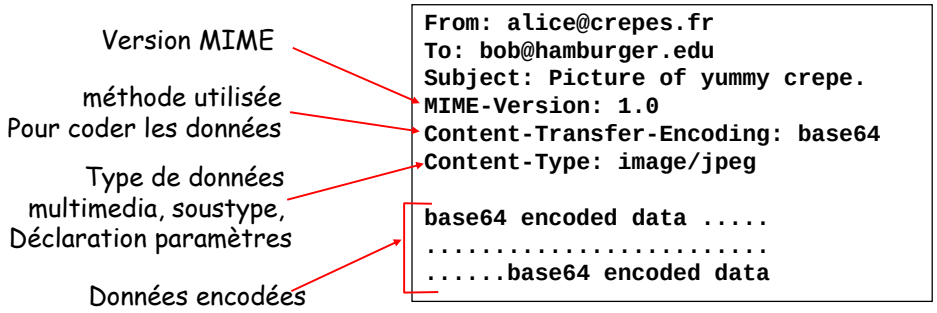
- le "message", caractères ASCII seulement



2.5.2 format des messages : extensions multimedia

MIME: extension multimedia, RFC 2045, 2056

Lignes supplémentaires dans l'entête du message pour déclarer que le type du contenu est de type MIME



2.5.2 types MIME

`Content-Type: paramètres type/subtype;`

Texte
Exemples de sous-types **`plain`, `html`**

Image
Exemples de sous-types **`jpeg`, `gif`**

Audio
Exemples de sous-types : **`basic`** (8-bite mu-law encoded), **`32kadpcm`** (32 kbps coding)

Video
Exemples de sous-types **`mpeg`, `quicktime`**

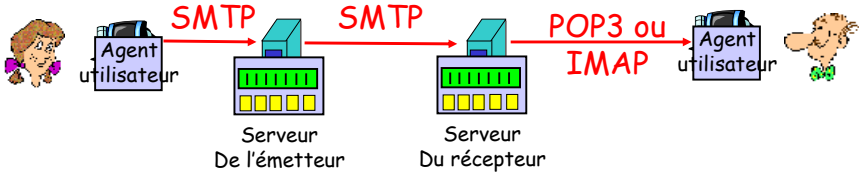
Application
Autres données qui doivent être traitées directement par un lecteur

Exemples de sous-types **`msword`, `octet-stream`**

2.5.2 Exemple “Multipart”

```
From: alice@crepes.fr
To: bob@hamburger.edu
Subject: Picture of yummy crepe.
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary=98766789

--98766789
Content-Transfer-Encoding: quoted-printable
Content-Type: text/plain

Dear Bob,
Please find a picture of a crepe.
--98766789
Content-Transfer-Encoding: base64
Content-Type: image/jpeg

base64 encoded data .....
.........................
......base64 encoded data
--98766789--
```


2.5.2 Protocoles d’accès



SMTP: envoie les messages vers le serveur du récepteur

Protocole d’accès : aller chercher les msgs sur le serveur

- POP: Post Office Protocol [RFC 1939]
  - identification (agent <-->server) et chargement
- IMAP: Internet Mail Access Protocol [RFC 1730]
  - Plus de caractéristiques
  - manipulation des msgs stockés sur le serveur
- HTTP: Hotmail , Yahoo! Mail, etc.

2.5.2 Protocole POP3

Phase d’authorisation
Commandes client :

- **`user:`** nom
- **`pass:`** password

Réponses serveur

- **`+OK`**
- **`-ERR`**

Phase de transaction
client:

`list:` liste les no de messages

`retr:` recherche message par no

`dele:` efface

`quit`

```
S: +OK POP3 server ready
C: user alice
S: +OK
C: pass hungry
S: +OK user successfully logged on
```

```
C: list
S: 1 498
S: 2 912
S: .
C: retr 1
S: <message 1 contents>
S: .
C: dele 1
C: retr 2
S: <message 1 contents>
S: .
C: dele 2
C: quit
S: +OK POP3 server signing off
```

2.5.3 DNS: Domain Name System

Personne: Plusieurs identificateurs :

- no secu, nom, no passeport

Hôtes, routeurs Internet:

- Adresse IP (32 bit) – utilisée pour acheminer les datagrammes
- "nom", e.g., chomolungma.lifl.fr – utilisé par les humains

Q: correspondance entre noms et ad. IP ?

Domain Name System:

Base de données répartie réalisée par une hiérarchie de serveurs de noms

Protocole de la couche application : hôte, routeurs, serveurs de noms doivent *résoudre* les noms (traduction nom/adresse) pour communiquer

note: fonction coeur de l'Internet

2.5.3 les serveurs de noms du DNS

Pourquoi ne pas centraliser le DNS?

point sensible (panne)

volume du trafic

base de données centralisée distante

maintenance

Pas de passage à l'échelle!

Pas de serveurs qui connaissent toutes les traductions nom/IP

Serveur de noms local

- Chaque domaine a un serveur de noms local (dit par défaut)
- Une requête DNS est dirigée en premier vers le serveur local

Serveur de noms faisant autorité:

- Pour un hôte: celui qui stocke l'adresse et le nom de l'hôte
- Peut exécuter la traduction nom/adresse pour ce nom d'hôte

2.5.3 Les serveurs racines DNS

contacté par le serveur de noms local qui ne peut pas résoudre un nom

Serveur racine:

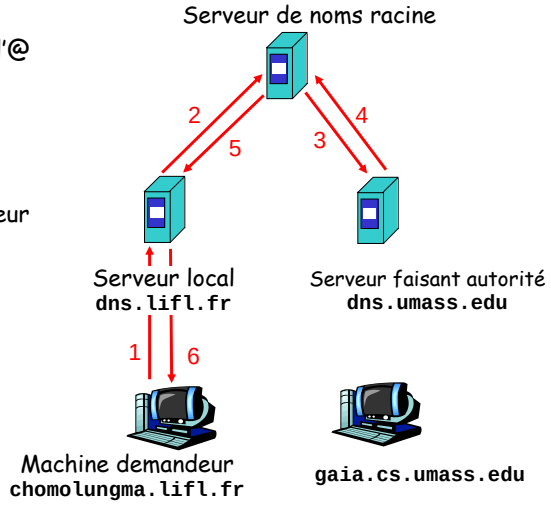
- Contacte le serveur de noms faisant autorité si la traduction n'est pas connue
- Fait la traduction
- Renvoie la traduction au serveur de noms local

13 serveurs racines dans le monde

2.5.3 un exemple simple DNS

chomolungma.lifl.fr veut l'@ IP de **gaia.cs.umass.edu**

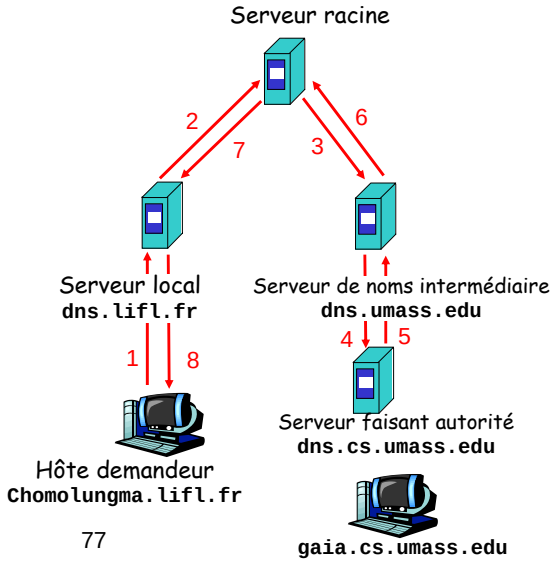
1. contacte son serveur DNS local **dns.lifl.fr**
2. **dns.lifl.fr** contacte le serveur racine, si nécessaire
3. Le serveur racine contacte le serveur faisant autorité, **dns.umass.edu,** si nécessaire



2.5.3 un exemple DNS

Serveur racine:
Peut ne pas connaître le serveur de noms faisant autorité

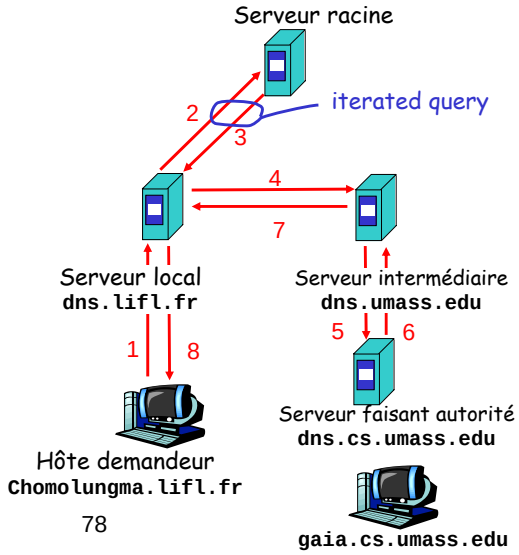
Peut connaître *un serveur de noms intermédiaire:* qu'il contacte pour trouver le serveur de noms faisant autorité



2.5.3 DNS: requêtes itératives

Requête récursive
Déplace la résolution de nom vers le serveur contacté
Charge ? Etat ?

Requête itérative
Le serveur contacté donne en réponse le serveur suivant à contacter
"je ne connais pas ce nom, mais essayez avec ce serveur"



2.5.3 DNS : cache et mise à jour des données

Dès qu'un serveur de noms apprend une traduction @IP/nom, il la stocke dans son cache

- Les entrées du cache sont associées à un timer et disparaissent après un certain temps

Les mécanismes de mises à jour/notification sont définies dans

- RFC 2136
- http://www.ietf.org/html.charters/dnsind-charter.html

2.5.3 les enregistrements DNS

DNS: BD répartie qui gère des ressources (RR)

Format RR : `(name, value, type, ttl)`

Type=A
- **name** est un nom d'hôte
- **value** est une @ IP

Type=NS
- **name** est un nom de domaine (e.g. foo.com)
- **value** est une @ IP du serveur faisant autorité pour ce domaine

Type=CNAME
- **name** est le nom alias pour certains noms canoniques
  e.g. `www.ibm.com` est en réalité `servereast.backup2.ibm.com`
- **value** est le nom canonique

Type=MX
- **value** et le nom du serveur de messagerie associé à **name**

2.5.3 Le protocole DNS, les messages

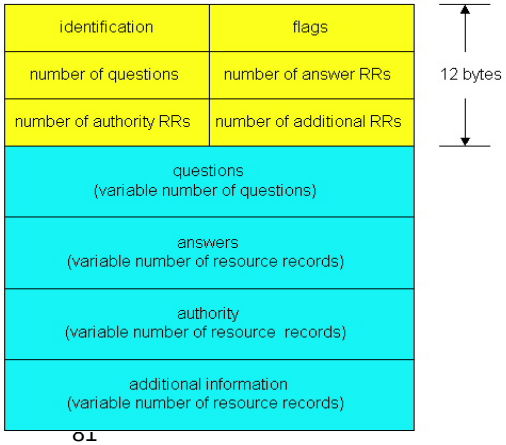
Le protocole DNS : les requêtes et les réponses ont le même format

Entête msg

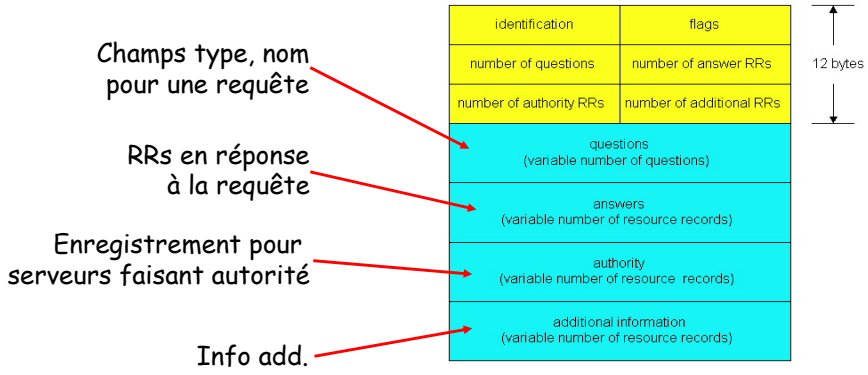
identification: 16 bits # pour requête, les réponses aux requêtes utilisent le même #

flags:

- requête ou réponse
- souhaite récursion
- recursion disponible
- réponse par serveur faisant autorité



2.5.3 Le protocole DNS les messages



3. Programmation concurrente

3.1 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

3.1 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)

3.2 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
    ...
}
```


3.2 Modèle de programmation

Modèle de programmation

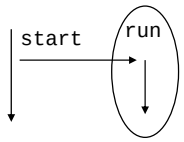
Lancement d'un thread : appel à la méthode `start()`

```
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*

3.2 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) !!

3.2 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

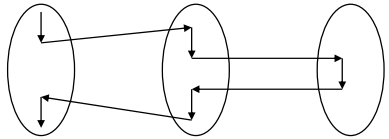
```
public class Foo implements Runnable {

    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();
```


3.3 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() { ... }
}
```

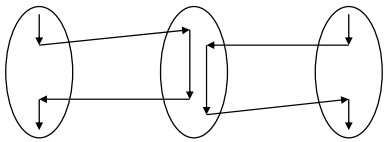
⇨ la construction d'un objet lui assigne des instructions à exécuter

3.3 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)



3.4 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

3.4 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

3.4 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 !!

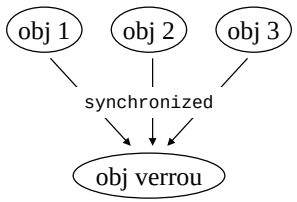
• coût
appel méthode `synchronized` ≈ 4 fois + long qu'appel méthode "normal"
⇨ à utiliser à bon escient

3.4 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



3.5 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)`

3.5 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

3.5 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()`

3.5 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;
```


3.5 Autres politiques

Politique lecteurs/écrivain

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

```
void demandeLecture(...) throws InterruptedException {
  synchronized(this) {
    while (ecrivain) { wait(); }
    lecteurs++;
} }

// On lit

void finLecture(...) throws InterruptedException {
  synchronized(this) {
    lecteurs--;
    notifyAll();
} }
```

3.5 Autres politiques

Politique lecteurs/écrivain

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

```
void demandeEcriture(...) throws InterruptedException {
  synchronized(this) {
    while (ecrivain || lecteurs>0) { wait(); }
    ecrivain = true;
} }

// On écrit

void finEcriture(...) throws InterruptedException {
  synchronized(this) {
    ecrivain = false;
    notifyAll();
} }
```


3.5 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
```


3.5 Autres politiques

Politique producteurs/consommateurs

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

```
void demandeProd(...) throws InterruptedException {
  synchronized(this) {
    while (taille == max) { wait(); }
} }

// En exclusion mutuelle
//   on produit (maj du tampon)
//   taille++

void finProd(...) throws InterruptedException {
  synchronized(this) {
    notifyAll();
} }
```


3.5 Autres politiques

Politique producteurs/consommateurs

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

```
void demandeCons(...) throws InterruptedException {
  synchronized(this) {
    while (taille == 0) { wait(); }
} }

// En exclusion mutuelle
//   on consomme (maj du tampon)
//   taille--

void finCons(...) throws InterruptedException {
  synchronized(this) {
    notifyAll();
} }
```

3.5 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) {}
}
```


3.5 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) {}
    }
}
```


3.6 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) {...} }
```


3.6 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

3.6 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

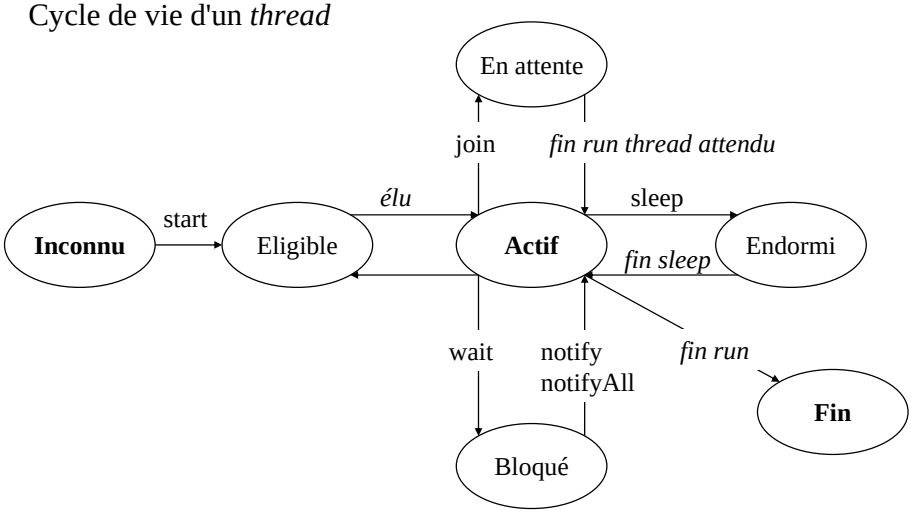
```
public class AsyncInputStream
  extends FilterInputStream implements Runnable {
  int[] buffer = ...  // zone tampon pour les données lues
  AsyncInputStream( InputStream is ) {
    super(is); new Thread(this).start(); }
```

```
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
}
```


3.6 Compléments

Cycle de vie d'un *thread*



3.6 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 !!
En pratique (ex JDK 1.3 Win 98)

- entre *threads* de même priorité : tourniquet
- si 1 *thread* de priorité + élevée : tourniquet 2 niveaux
- si 2 *threads* de priorité + élevée
  les *threads* de priorité basse ne sont pas ordonnancés !!

3.6 Compléments

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