

Applications réseaux

Filière Informatique et Réseaux

<http://igm.univ-mlv.fr/~duris/RESEAU/JAVA/IR2/>

Etienne.Duris@univ-paris-est.fr

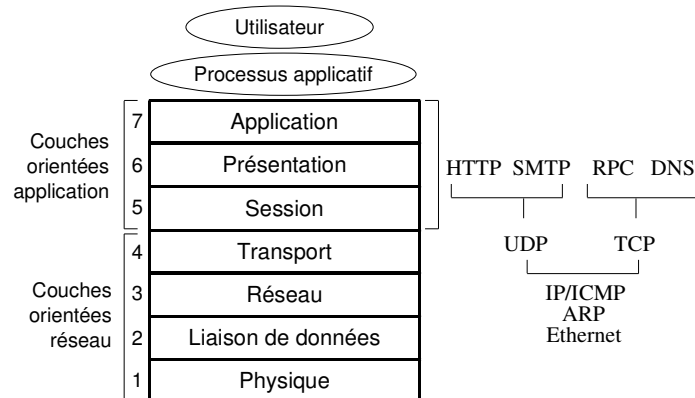
UFR Ingénieurs 2000

Université Paris-Est Marne-la-Vallée

Contexte

- On s'intéresse aux applications de type Internet
 - Interconnexion des réseaux utilisant le protocole Internet (IP)
 - La couche IP sert à **adresser**, **acheminer**, **router** les paquets IP
 - Pour échanger des informations (être utilisé), le protocole Internet de la **couche réseau (3)** requiert un protocole « compagnon » de la **couche transport (4)**
 - UDP (*User Datagram Protocol*) ou
 - TCP (*Transmission Control Protocol*)
 - La grande majorité des applications (DNS, SMTP, FTP, HTTP, etc.) reposent sur cette suite dite « TCP/IP »
- L'accès aux « couches basses » (1 et 2) est plus délicat
 - Dépendant du protocole, du système d'exploitation (libpcap)

Open System Interconnection (OSI)



Standards et normes

- Adoptés par l'IAB (*Internet Architecture Board*)
 - IRTF (*Internet Research Task Force*) : long terme
 - IETF (*Internet Engineering Task Force*) : court terme
- Distribués par l'INTERNIC (*INTERNet Network Information Center*)
 - sous la forme de documents appelés **RFC** (*Request For Comments*)
 - www.rfc-editor.org
 - www.irtf.org
 - www.ietf.org

Gestion des adresses Internet

- Les noms et numéros sont gérés par
 - IANA (*Internet Assigned Numbers Authority*)
 - www.iana.org
 - ICANN (*Internet Corporation for Assigned Names and Numbers*)
 - www.icann.org
- Délèguent leurs fonctions au niveau régional
 - RIPE NCC (Réseaux IP Européens, Network Coordination Center) pour une partie de l'Europe de l'Est et l'Afrique
 - Délègue, au niveau national, à des LocalIR (*Internet Registry*). GEANT (europe) RENATER (france)
 - Délègue à l'administrateur (ex. Université)

Les noms IANA et adresses numériques

- Noms composés de labels séparés par des points « . »
 - Chaque label fait au plus 63 caractères, au plus 255 caractères au total
 - Majuscules et minuscules sont indifférenciées
- Organisés hiérarchiquement
 - Domaine « univ-mlv » est un sous-domaine du domaine « fr »
 - Quand une machine recherche une @IP associée à un nom,
 - Elle fait appel à un serveur DNS local.
 - S'il n'a pas la réponse localement, il fait appel au serveur DNS du domaine du nom recherché ou, au pire, au serveur racine.
- DNS (*Domain Name System*) est une base de données distribuée
 - Implémentation BIND (*Berkeley Internet Name Domain*)
 - Fichiers `/etc/hosts` et `/etc/resolv.conf`
 - Outils **dig** ou **nslookup**

Adresses Internet

- Adresses **universelles logiques** (non physiques)
 - 4 octets en IP v4, 16 octets en IP v6
- Cinq classes d'adresses en IP v4: A, B, C, D ou E
 - Déterminée à partir des 4 bits de poids fort du premier octet
 - **0xxx** : adresse de classe **A**
 - **10xx** : adresse de classe **B**
 - **110x** : adresse de classe **C**
 - **1110** : adresse de classe **D**
 - **1111** : adresse de classe **E**

Classes d'adresses

- Chaque adresse comporte deux parties
 - Le début identifie un réseau
 - La fin identifie une machine dans ce réseau
- La classe indiquait historiquement les limites de ces deux parties
 - Classe A : 1 octet pour le réseau et 3 pour la machine
 - Classe B : 2 octets pour le réseau et 2 pour la machine
 - Classe C : 3 octets pour le réseau et 1 pour la machine
 - En fonction de sa classe, une adresse de réseau peut « contenir » plus ou moins d'adresses de machines
 - 16 millions en classe A, 65000 en classe B et 256 en classe C

Exemple de classe d'adresse

- Pour représenter l'adresse du réseau, tous les bits correspondant à l'adresse de la machine sont mis à zéro (convention)
- Exemple: l'adresse 192.55.6.2 est de classe C
 - 11000000.00110101.00000110.00000010
 - L'adresse du réseau est donc 192.55.6.0
 - L'adresse de la machine dans ce réseau est « 2 »
- Les adresses de classe D sont des adresses de groupe (**multicast**): de 224.0.0.0 à 239.255.255.255
- Les adresses de classe E sont réservées

Masques de réseau

- Le découpage par classes est trop rigide
- La notion de masque de réseau permet de « couper » les deux parties n'importe où (ailleurs qu'aux octets « ronds »)
- Il s'agit d'une adresse composée d'un nombre de bits à 1 au début, et tout le reste à 0
- Remplace et affine la notion de classe
- Par exemple, 10.65.67.12 / 255.224.0.0 ou encore 10.65.67.12 / 11

Masque	11111111.11100000.00000000.00000000	255.224.0.0
Adresse	00001010.01000001.01000011.00001100	10.65.67.12
Réseau	00001010.01000000.00000000.00000000	10.64.0.0
Machine	00000000.00000001.01000011.00001100	0.1.67.12

Composition de réseaux

- La notion de masque permet de
 - « découper » un réseau en sous-réseaux, c'est le **subnetting**.
Par exemple, les réseaux 10.32.0.0/11 et 10.64.0.0/11 peuvent coexister: leurs adresses ne se recouvrent pas.
 - « agréger » plusieurs sous-réseaux en un seul, c'est le **supernetting**.
Par exemple, l'adresse de réseau 193.55.86.34/22 regroupe 4 réseaux de classe C
- Très utile pour hiérarchiser un réseau (routage)
- L'utilisation des masques de réseau relève de la RFC 1519: **CIDR** (*Classless Inter Domain Routing*)

Adresses réservées

- L'adresse du **réseau** lui-même
 - Tout à zéro pour ce qui concerne les bits « machine »
- L'adresse de **broadcast**
 - Broadcast réseau local (tout à 1): 255.255.255.255
 - Broadcast dirigé: tout à 1 pour ce qui concerne les bits « machine » (souvent bloqué par les routeurs)
- L'adresse « **non spécifiée** »: 0.0.0.0 (*wildcard* ou *anylocal*)
 - Elle représente la machine locale avant qu'elle ait une adresse
- L'adresse de **loopback**: 127.0.0.1
- Plages d'adresses réservées non affectées (RFC 1918)
 - 10.0.0.0/8 et 127.0.0.0/8 pour la classe A
 - 169.254.0.0/16 et 172.16.0.0/16 pour la classe B
 - 192.168.0.0/16 pour la classe C

Adresses logiques, adresses physiques

- Sur même réseau physique, pas besoin de routeur
 - Point à point: aucune ambiguïté
 - Diffusion: nécessité de traduction @physique <--> @logique
- Différents mécanismes possibles
 - Association statique entre les 2 adresses
 - Table d'association (DHCP)
Dynamic Host Configuration Protocol, RFC 2131
 - Calcul (e.g. multicast Ethernet ou partie d'adresse en IP v6)
 - Calcul ou recherche dynamique
 - Évite de maintenir « à la main » les tables
 - Protocoles de résolution:
ARP **Address Resolution Protocol** (RFC 826)
ou NetBios (Windows)

Address Resolution Protocol

- Chaque machine possède une table ARP
 - Contient des « entrées » associant @IP à @MAC, dont la durée de vie est relativement courte (< 20 min)
 - Commande **arp -a**
 - Pour joindre une machine d'@IPB donnée, une machine d'@IPA recherche si elle dispose de l'information dans sa table
 - Si oui, elle utilise l'adresse physique associée
 - Si non, elle diffuse (broadcast physique) une « requête ARP » dans tout le réseau physique qui dit en substance:
« Qui possède l'adresse logique @IPB ? »
 - Seule la machine ayant cette adresse (si elle existe) répond à la machine A (sur son adresse physique @MacA)
 - Les deux machines ont renseigné leurs tables ARP

ARP (format de trame)

0	8	16	24
Type de réseau (1 pour Ethernet)		Type de protocole (0800 pour Internet)	
Taille @phy. (Eth:6)		Taille @log. (IP:4)	Type opération (1,2: ARP; 3,4: RARP)
Octets 1, 2, 3 et 4 de l'adresse physique de l'émetteur (ex: Ethernet)			
Octets 5, 6 @phys émetteur		Octets 1 et 2 de l'@logique de l'émetteur	
Octets 3 et 4 de l'@ logique de l'émetteur		Octets 1, 2 de l'@ physique du récepteur	
Octets 3, 4, 5 et 6 de l'adresse physique du récepteur (ex: Ethernet)			
Adresse logique du récepteur (ex: IP)			

ARP (suite)

- ARP non sécurisé
 - Une machine ayant accès au réseau peut se faire passer pour une autre
 - *ARP cache poisoning*, attaque *Man In the Middle (MiM)*
 - Les broadcasts générés inondent tout le LAN
 - RARP (*Reverse ARP*)
 - Obtenir une @ IP (dynamique) à partir d'une @MAC
 - Plus utilisé dans la pratique
- Remplacé dans la pratique par DHCP (*Dynamic Host Configuration Protocol*), RFC 2131
 - basé sur UDP, ports 67 et 68

Protocole Internet

- RFC 791 (Jon Postel) et STD 0005
 - Acheminement de datagrammes d'un point à un autre du réseau, repérés par des adresses IP
- Principe de **commutation de paquet**
 - Chaque datagramme est « routé » indépendamment des autres (**plusieurs chemins possibles, non conservation de l'ordre**)
 - Lorsqu'un datagramme est transmis, rien n'assure qu'il arrivera un jour: **non fiable** ou **au mieux** (*best effort*)
- Traversée de plusieurs réseaux physiques différents
 - possibilité de **fragmentation** des paquets

Format des datagrammes (IP v4)

0	4	8	12	16	20	24	28
Version	Taille (mots de 32 bits)	Service (TOS)	Taille totale, en-tête compris (en octets)				
Identificateur				Marq.	Décalage du fragment		
Durée de vie	Protocole (6: TCP, 17: UDP, etc...)			Somme de contrôle			
Adresse IP émetteur							
Adresse IP destinataire							
[Options éventuelles...							
...options éventuelles...							
...options éventuelles]					Bourrage (pour compléter à un mot de 32)		

	0	1	2	3	4	5	6	7
Type Of Service	Précédence	Délai	Débit	Sureté	Coût	0		

Fragmentation

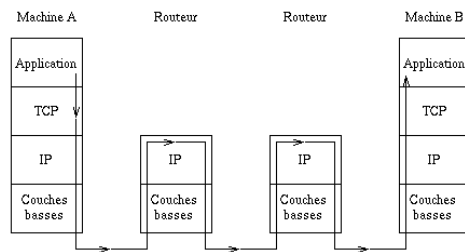
- MTU (*Maximum Transfert Unit*): taille maximum d'un datagramme IP transporté par un réseau physique
 - Ex: 1500 pour Ethernet, 4470 pour FDDI
- Il peut être nécessaire de « découper » un datagramme pour adapter sa taille au MTU du réseau traversé
- L'en-tête des différents fragments d'un même datagramme est presque le même
- Le champ **Identificateur** est le même pour tous les fragments (avec l'adresse source, il est un id. « unique »)
- **Taille totale** et **Somme de contrôle** changent, mais aussi **Décalage du fragment** et **Marq.**

Fragmentation (suite)

- **Décalage du fragment**:
 - Position du (début des données du) fragment dans le datagramme original, en multiple de 8 octets
 - Le premier fragment est à la position 0
- **Marq.** : champ constitué de trois bits
 - Le premier est réservé à 0
 - **DF** (*Don't Fragment bit*): s'il vaut 1, il faut jeter ce datagramme plutôt que de le fragmenter
 - Permet de faire de la découverte de *Path MTU*
 - **MF** (*More Fragment bit*): s'il vaut 1, le fragment contenu dans ce datagramme n'est pas le dernier du datagramme original

Routing

- De **proche en proche**, en fonction de l'**adresse Internet du destinataire** et des **tables de routages** des différentes machines rencontrées:
aucune connaissance de la topologie globale



Tables de routage

- Chaque entrée de la table spécifie
 - Une adresse
 - Un masque pour cette adresse (routeurs CIDR)
 - Une interface de sortie, ou l'@ du prochain routeur
- Un exemple:

Destination	Gateway	Genmask	Iface
127.0.0.0	*	255.0.0.0	lo
192.55.6.0	*	255.255.255.0	eth0
default	192.55.6.1 (Routeur)		0.0.0.0
- Commandes **netstat -r** (consulter), **route** (construire)

Principe du routage

- Filtrage par le plus long préfixe**
- Pour router un datagramme d'adresse destination ipA
 - Pour chaque entrée dans la table (route), faire un **ET** binaire entre ipA et le masque de la route
 - Si valeur obtenue est égale à l'adresse, alors route valide
- Une adresse peut être filtrée par **plusieurs entrées**
- Plus l'adresse est **précise**, meilleure est la route
 - Possibilité d'**ordonner** les entrées
 - Notion de **coût** éventuelle
- Route par défaut**: filtre toutes les adresses

Principe du routage (suite)

- Une fois la meilleure route trouvée, elle indique
 - Soit une interface de réseau physique (machine dest)
 - Table ou requête ARP pour déterminer l'@MAC correspondante
 - Soit l'adresse d'un routeur
 - Table ou requête ARP pour déterminer l'@MAC correspondante
 - Datagramme envoyé au routeur pour le "saut" suivant
 - Interface du routeur doit pouvoir être atteinte récursivement

Mise à jour et partage des informations de routage

- Peut se faire statiquement, à la main (**route**)
- Protocoles de mise à jour dynamique
 - Il est impossible que tous les routeurs d'Internet s'échangent leurs informations de routage
 - Il sont organisés en AS (*Autonomous Systems*)
 - A l'intérieur les protocoles sont dits IGP (*Interior Gateway Protocol*). Ex: RIP, OSPF, EIGRP
 - Entre-eux, les AS s'échangent des informations *via* des EGP (*Exterior Gateway Protocol*). Ex: EGP, BGP

ICMP (*Internet Control Message Protocol*)

- Permet d'envoyer des informations de contrôle à l'expéditeur d'un datagramme (encapsulées dans IP)
 - utilisé par les commandes **ping** ou **traceroute**.
 - Différents types de datagrammes
 1. **Echo Reply**
 2. Destination Unreachable
 3. Source Quench
 4. **Redirect (change route)**
 5. **Echo Request**
 6. **Time Exceeded to Datagram**
 7. Parameter Problem on a Datagram
 8. Timestamp Request
 9. Timestamp Reply
 10. Information Request (obsolete)
 11. Information Reply (obsolete)
 12. Address Mask Request
 13. Address Mask Reply

Quelques mots sur IP v6

- RFC 2373 et RFC 2460. Pourquoi?
 - Nécessité de pouvoir attribuer plus d'adresses
 - Volonté d'être plus efficace dans le routage (nombre d'entrées)
 - Pouvoir intégrer facilement de nouveaux services
 - Impérieuse nécessité d'une transition progressive
- Points clés:
 - Adresses et allocation des préfixes (id interface)
 - Découverte des voisins (attribution d'adresse + routage)
 - Format de datagramme plus gros, mais simplifié
 - En-têtes d'extensions

Les adresses IP v6

- 128 bits au lieu de 32 en IP v4 (16 octets au lieu de 4)
 - Puissance d'adressage astronomique (seule une petite partie est prévue pour utilisation immédiate)
- Représentation
 - 8 groupes de 4 symboles hexadécimaux séparés par « : »
 - Ex (www.6bone.net):
3ffe:0b00:0c18:0001:0000:0000:0000:0010
 - Les 0 en tête de groupe ne sont pas forcément représentés
Exemple: **3ffe:b00:c18:1:0:0:0:10**
 - Contraction possible d'une (unique) suite contiguë de groupes de 16 bits à 0. Exemple : **3ffe:b00:c18:1::10**

Le protocole IP et Java

- API d'accès: `java.net.*`;
- Adresses IP représentées par les instances de la classe `InetAddress`, ou plus précisément de l'une des deux sous-classes
 - `Inet4Address` pour le protocole IPv4 (32 bits)
 - `Inet6Address` pour le protocole IPv6 (128 bits)
- *A priori*, chaque instance représente 2 informations
 - un tableau d'octets (4 ou 16): adresse numérique (*address*)
 - un nom éventuel (DNS, NIS): adresse « littérale » (*host*)
 - Si ce nom existe ou si la résolution de nom inverse a déjà été faite

Représentation des adresses

- L'adresse « littérale » n'est pas nécessairement résolue (peut être remplacée par la chaîne vide)
 - méthode `String toString()` ⇒
`www.6bone.net/131.243.129.43` ou encore
`www.6bone.net/3ffe:b00:c18:1:0:0:0:10`
 - méthodes `String getHostName()` ⇒ « `www.6bone.net` » ou
`String getCanonicalHostName()` ⇒ « `6bone.net` »
 - méthode `String.getHostAddress()`
⇒ « `131.243.129.43` » (adresse numérique sous forme de String)
 - méthode `byte[] getAddress()`
⇒ {131, 243, 129, 43} (attention, en Java, le type `byte` comme tous les types primitifs numériques est *signé*!)

Récupération d'une instance (méthodes statiques)

- Adresse IP de la machine locale
 - `InetAddress.getLocalHost()` (éventuellement adresse de loopback)
- Étant donnée une adresse littérale ou numérique
 - `InetAddress.getByName(String host)`
 - `InetAddress.getAllByName(String host)` // tableau
- Étant donné un tableau d'octets (pas de résolution DNS)
 - `InetAddress.getByAddress(byte[] addr)` // non bloquant
- Étant donné un nom et un tableau d'octets (idem)
 - `InetAddress.getByAddress(String host, byte[] addr)`
 - création, aucun système de nom interrogé pour vérifier la validité

Exemples de représentations

```
➤ InetAddress ia1 = InetAddress.getByName("java.sun.com");
System.out.println(ia1.toString()); // java.sun.com/192.18.97.71
System.out.println(ia1.getHostName()); // java.sun.com
System.out.println(ia1.getCanonicalHostName()); // flres.java.sun.COM
System.out.println(ia1.getHostAddress()); // 192.18.97.71

➤ InetAddress ia2 = InetAddress.getByName("192.18.97.71");
System.out.println(ia2.toString()); // /192.18.97.71
System.out.println(ia2.getHostName()); // flres.java.sun.COM
System.out.println(ia2.getCanonicalHostName()); // flres.java.sun.COM
System.out.println(ia2.getHostAddress()); // 192.18.97.71

➤ byte[] b = ia2.getAddress(); // affiche b[0]=-64
for(int i=0; i<b.length; i++) { // b[1]=18
    System.out.println("b["+i+"]="+b[i]); // b[2]=97
} // b[3]=71
```


Exemples de représentation (suite)

- ```
InetAddress [] ias = InetAddress.getAllByName("www.w3.org");
for(int i = 0; i < ias.length; i++) // affiche: www.w3.org/18.29.1.34
System.out.println(ias[i]); // www.w3.org/18.29.1.35
 // www.w3.org/18.7.14.127
```
- Toutes ces méthodes peuvent lever des exceptions de classe `UnknownHostException`
  - si le format est incorrect
  - si les arguments de sont pas corrects
  - si la résolution de nom n'aboutit pas
- ```
InetAddress bidon =
InetAddress.getByAddress("nimporte.quoi", new byte[]{1,2,3,4});
System.out.println(bidon); // affiche : nimporte.quoi/1.2.3.4
bidon = InetAddress.getByAddress("nimporte.quoi");
// lève java.net.UnknownHostException
```

Observations sur les adresses

méthode	IP v4	IP v6	adresses concernées
<code>isAnyLocalAddress()</code>	0.0.0.0/8	::0	non spécifiée
<code>isLoopbackAddress()</code>	127.0.0.0/8	::1	Loopback
<code>isLinkLocalAddress()</code>	169.254.0.0/16	fe80::/16	locale au lien (a)
<code>isSiteLocalAddress()</code>	10.0.0.0/8 172.16.0.0/12 192.168.0.0/16	fec0::/16	locale au site (b)
<code>isMulticastAddress()</code>	224.0.0.0/4	ff00::/8	Multicast
<code>isMCGlobal()</code>	de 224.0.1.0 à 238.255.255.255	ffxe::/16	multicast portée globale
<code>isMCOrgLocal()</code>	239.192.0.0/14 (a)	ffx8::/16	mult. port. org.
<code>isMCSiteLocal()</code>	239.255.0.0/16 (a)	ffx5::/16	mult. port. site
<code>isMCLinkLocal()</code>	224.0.0.0/24 (a)	ffx2::/16	mult. port. lien
<code>isMCNodeLocal()</code>	aucune (b)	ffx1::/16	mult. port. noeud

(a) sous réserve de valeurs de TTL adéquates.

(b) Seul un champ TTL à 0 donne une portée locale en IP v4.

Test de connectivité

- Depuis jdk1.5, possibilité de faire une sorte de ping
 - Depuis une instance de la classe `InetAddress`, teste si l'adresse qu'elle représente est « atteignable » (*reachable*)
 - Implantation au mieux, mais firewall ou config de serveur peuvent faire en sorte que ça échoue (retourne false) alors que l'@IP est atteignable sur certains port
 - Typiquement, tente un envoi de ECHO REQUEST en ICMP (si privilège ok), ou alors tente d'établir une connexion TCP avec le port 7 (Echo)
 - `public boolean isReachable(int timeout) throws IOException`
 - Au delà de `timeout` millisecondes, levée de l'exception
 - `public boolean isReachable(NetworkInterface netif, int ttl, int timeout) throws IOException`
 - Permet de spécifier l'interface de sortie (null pour n'importe laquelle)
 - et le nombre de sauts maximum des paquets (0 pour défaut)

Test de connectivité : exemples

```
public class NameResolution {
    public static void main(String[] args) throws IOException {
        InetAddress inet = InetAddress.getByAddress(args[0]);
        System.out.println(inet);
        System.out.println("Test de connectivité...");
        if (inet.isReachable(500))
            System.out.println(" réussi.");
        else
            System.out.println(" échoué.");
    }
}
```

- **User:** tentative connexion TCP

- Accepte ou refuse, mais en général répond

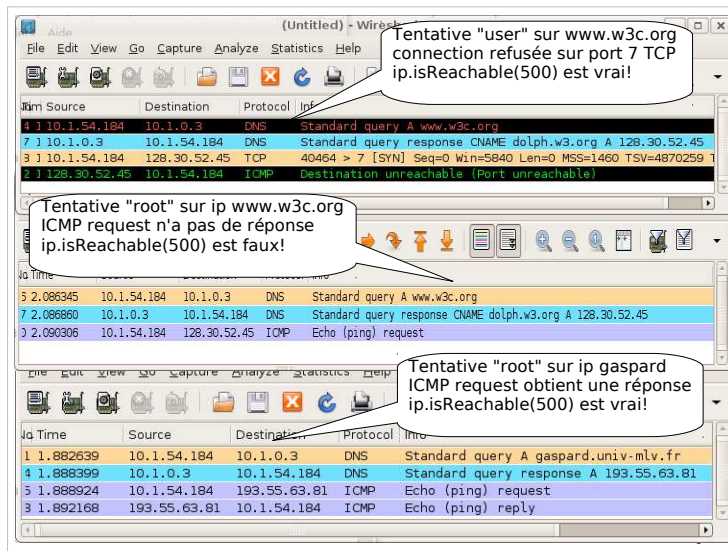
- **Root:** ping (ICMP)

- Si ne réponds pas aux ICMP Request, échec
- Si répond, succès

```
User$> java NameResolution www.w3c.org
www.w3c.org/128.30.52.45
Test de connectivité... réussi.
```

```
Root# java NameResolution www.w3c.org
www.w3c.org/128.30.52.45
Test de connectivité... échoué.
```

```
Root# java NameResolution gaspard
gaspard.univ-mlv.fr/193.55.63.81
Test de connectivité... réussi.
```



Interfaces de réseau

- Une interface de réseau est représentée par un objet de la classe `java.net.NetworkInterface`
 - un nom (lo, eth0, etc..)
 - une liste d'adresses IP associées à cette interface
- Les méthodes **statiques** permettant d'obtenir des objets de cette classe:
 - `java.util.Enumeration<NetworkInterface> getNetworkInterfaces()`
 - `NetworkInterface getByname(String name)`
 - `NetworkInterface getByInetAddress(InetAddress addr)`
- Peuvent lever des exceptions `SocketException`

Exemple de représentation

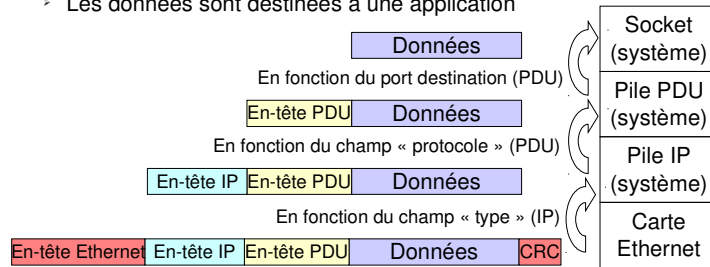
- ```
for(Enumeration<NetworkInterface> e =
 NetworkInterface.getNetworkInterfaces();
 e.hasMoreElements();)
 System.out.println(e.nextElement());
// affiche:
// name:eth0 (eth0) index: 2 addresses:
// /193.12.34.56;
//
// name:lo (lo) index: 1 addresses:
// /127.0.0.1;
```
- `getInetAddresses()` sur une instance renvoie une énumération des adresses IP associées.
    - Principalement utilisé pour spécifier un attachement multicast sur une interface donnée.

## Les adresses de socket

- Une socket identifie un point d'attachement à une machine. Selon le protocole, des classes sont dédiées pour représenter ces objets
  - `java.net.DatagramSocket` pour UDP
  - `java.net.Socket` pour TCP
- Les adresses de sockets identifient ces points d'attachement indépendamment du protocole
  - `java.net.SocketAddress` (classe abstraite)
  - a priori* indépendant du protocole réseau, mais la seule classe concrète est dédiée aux adresses Internet (IP)
    - `java.net.InetSocketAddress`

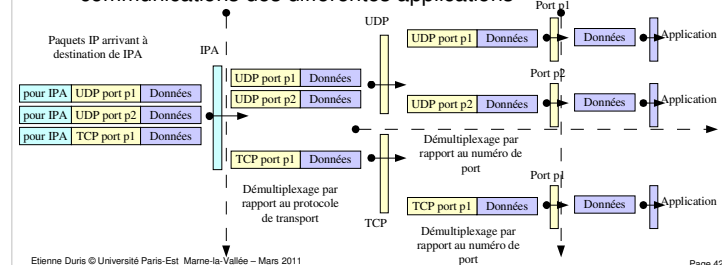
## Encapsulation des données

- Une trame Ethernet est adressée à une adresse MAC
- Un paquet IP (couche 3) est destiné à une adresse IP
- Un PDU (couche 4 – UDP ou TCP) est destiné à un port
- Les données sont destinées à une application



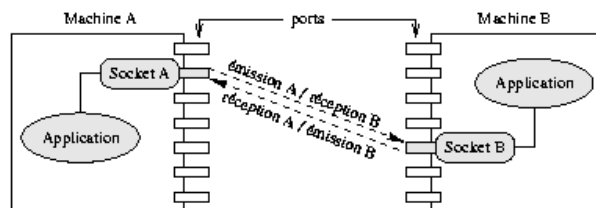
## Multiplexage / démultiplexage

- Les applications se lient à des **sockets** pour accéder au réseau
- Ces sockets système sont identifiées par une **adresse IP** et un **numéro de port**
- Elles permettent de multiplexer/démultiplexer les communications des différentes applications



## Le rôle des sockets

- Pour UDP et TCP, les sockets jouent le même rôle et sont identifiées par une adresse IP et un numéro de port.
- Au moins 2 sockets impliquées dans une communication



## Adresses de loopback et non spécifiée

- Loopback: ne correspond à aucun réseau physique
  - **127.0.0.1** en IP v4
  - **::1** en IP v6
  - nom en général associé: **localhost**
- Adresse non spécifiée (*wildcard*): correspond à « n'importe quelle » adresse (*anylocal*)
  - **0.0.0.0** en IP v4
  - **::0** en IP v6
    - En émission, elle vaut **l'une des** adresses IP de la machine locale
    - En réception, elle filtre **toutes** les adresses IP de la machine locale

## Classe InetSocketAddress

- `java.net.InetSocketAddress`
  - adresse IP (`InetAddress`)
  - numéro de port (`int`)
- Trois constructeurs acceptant comme arguments
  - `InetSocketAddress(InetAddress addr, int port)`
    - Si `addr` est null, la socket est liée à l'adresse wildcard (non spécifiée)
  - `InetSocketAddress(int port)`
    - l'adresse IP est l'@ wildcard
  - `InetSocketAddress(String hostName, int port)`
    - Si `hostName` non résolu, l'adresse de socket est marquée comme étant non résolue
    - Peut être testé grâce à la méthode `isUnresolved()`

## Adresse de socket (suite)

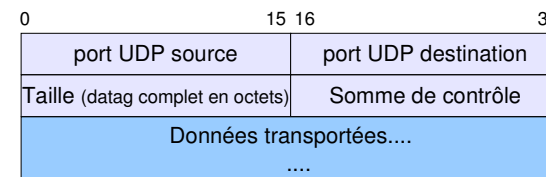
- Tous lèvent une `IllegalArgumentException` si le numéro de port est en dehors de [0..65535]
- Si le port spécifié vaut 0, un port éphémère sera choisi lors de l'attachement de la socket.
- La méthode `toString()` affiche `<@ip>:<n° port>`
- ```
InetSocketAddress sa1 =
    new InetSocketAddress("un.truc.zarb",50);
System.out.println(sa1+(sa1.isUnresolved()?" non résolue":"résolue"));
// Affiche : un.truc.zarb:50 non résolue
InetSocketAddress sa2 =
    new InetSocketAddress("java.sun.com",80);
System.out.println(sa2 + (sa2.isUnresolved()?" non résolue":" résolue"));
// Affiche : java.sun.com/192.18.97.71:80 résolue
```

Le protocole UDP

- User Datagram Protocol (RFC 768)
 - acheminement de datagrammes au dessus de IP
 - pas de fiabilité supplémentaire assurée
 - assure la préservation des limites de chaque datagramme
- Le multiplexage/démultiplexage au niveau des machines se fait via la notion de port
 - certains ports sont affectés à des services particuliers
 - RFC 1700 ou www.iana.org/assignments/port-numbers
 - En général, les n° de port inférieurs à 1024 sont réservés

Format

- Taille max des données transportées: $(2^{16}-1 \cdot 8) \sim 64\text{Ko}$
- Checksum optionnel en IP v4, obligatoire en IP v6
 - ajout (pour le calcul) d'un pseudo-en-tête avec @ dest et src



Exemple d'encapsulation dans une trame Ethernet:



Accès Java à UDP

- Classiquement (avant jdk 1.4), l'accès se fait grâce à deux classes et permet de manipuler des tableaux d'octets. Il faut les « **interpréter** » en fonction de l'application: encodage de caractères, format, etc.
- `java.net.DatagramSocket`
 - représente une socket d'attachement à un port UDP
 - Il en faut une sur chaque machine pour communiquer
- `java.net.DatagramPacket` qui représente deux choses:
 - Les données qui transitent:
 - un tableau d'octets,
 - l'indice de début et
 - le nombre d'octets
 - La machine distante:
 - son adresse IP
 - son port

Créer un DatagramSocket

- Représente un objet permettant d'envoyer ou recevoir des datagrammes UDP
- Différents constructeurs acceptant des arguments:
 - `InetSocketAddress` (@IP+port, éventuellement wildcard)
 - Si null, socket non attachée. À attacher plus tard avec `bind()`
 - `port + InetAddress`
 - même chose si `InetAddress` null
 - `port` seul (attachée à l'adresse *wildcard*)
 - aucun argument
 - (attachée à l'adresse *wildcard* et à un port libre de la machine)
- Peut lever `SocketException` (problème d'attachement)

Observations sur DatagramSocket

- `getLocalPort()`, `getLocalAddress()` et `getLocalSocketAddress()`
 - retournent les infos sur la socket attachée localement
 - Numéro de port, `InetAddress` et `InetSocketAddress` locaux
- `bind(SocketAddress)`
 - attache la socket si elle ne l'est pas déjà
 - `isBound()` retourne `false`
- `close()`
 - ferme la socket (libère les ressources système associées)
- `isClosed()`
 - Permet de savoir si la socket est fermée

Créer un DatagramPacket

- Représente un objet spécifiant les données qui doivent transiter ainsi que l'interlocuteur
- Plusieurs constructeurs existent, qui spécifient:
 - les données
 - `byte[]` buffer
 - `int` offset
 - `int` length
 - L'interlocuteur distant
 - soit une `InetSocketAddress`
 - soit une `InetAddress` et un numéro de port (`int`)

Un objet, deux usages

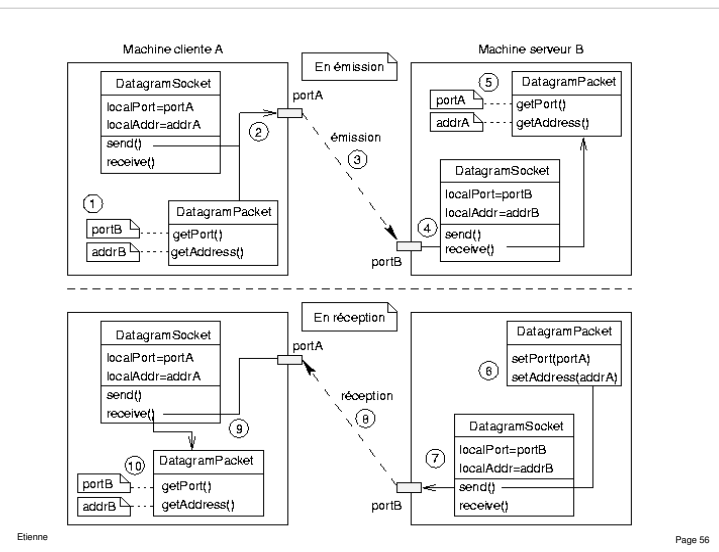
- En émission, le `DatagramPacket` spécifie
 - les données à envoyer
 - la machine (et le port) vers qui les envoyer
- En réception, le `DatagramPacket` spécifie
 - la zone de données permettant de recevoir
 - mettra à jour, lors de la réception, la machine et le port depuis lesquels ces données ont été reçues
- Un même objet peut servir aux deux usages

Émission, réception

- Un `DatagramPacket` est fourni à (et pris en charge par) un `DatagramSocket`
- Pour émettre:
 - `datagramSocket.send(datagramPacket);`
- Pour recevoir:
 - `datagramSocket.receive(datagramPacket);`
 - Appel bloquant tant que rien n'est reçu
- Peuvent lever différentes `IOException`

Observations sur DatagramPacket

- Différentes méthodes d'accès aux champs
 - `[set/get]Address()`, `[set/get]Port()`, `getSocketAddress()`
 - Concerne la machine distante (qui a émis ou va recevoir)
 - `[set/get]Data()`, `getOffset()`, `setLength()`
 - permet de spécifier les données ou les bornes dans le tableau d'octets
- `getLength()` comportement « normal » en émission mais:
- `getLength()` avant réception: **taille de la zone** de stockage
- `getLength()` après réception: **nombre d'octets reçus**



Précisions

- Les données reçues au delà de la taille de la zone de stockage sont perdues
- En pratique, les plateformes limitent le plus souvent à 8K
 - Peuvent se contenter d'accepter 576 octets, en-tête IP comprise
- Le système peut fixer des tailles des tampons de réception et d'émission (pour plusieurs paquets)
 - On peut demander à les changer, sans garantie de succès
 - `[get/set]ReceiveBufferSize()` et `[get/set]SendBufferSize()`
- Risque de perte de datagramme:
 - on peut vouloir limiter l'attente en réception
 - `socket.setSoTimeout(int milliseconds)` interrompt l'attente de réception au delà de `milliseconds`
 - lève alors une exception `SocketTimeoutException`

Exemple d'émission (client)

```
➤ // socket sur un port libre et sur l'adresse wildcard
   locale
   DatagramSocket socket = new DatagramSocket();
   // tableau d'octets correspondant à la String "Hello"
   byte[] buf = "Hello".getBytes("ASCII");
   // création d'un datagramme contenant ces données
   // et destiné au port 3333 de la machine de nom serveur
   DatagramPacket packet =
       new DatagramPacket(buf, buf.length,
                           InetAddress.getByName("serveur"),
                           3333);
   // envoi du datagramme via la socket
   socket.send(packet);
```

Exemple de réception (client)

```
➤ // (sur la même socket qui a envoyé "Hello")

   // allocation et mise en place d'un buffer pour la réception
   byte[] receiveBuffer = new byte[1024];
   packet.setData(receiveBuffer);
   System.out.println(packet.getLength());
   // affiche: 1024 (c'est la taille de la zone de stockage)

   // mise en attente de réception
   socket.receive(packet);
   System.out.println(packet.getLength());
   // affiche le nombre d'octets effectivement reçus (<= 1024)

   // construction d'une String correspondant aux octets reçus
   String s = new String(receiveBuffer, 0,
                         packet.getLength(), "ASCII");
   System.out.println(s);
   // Quelle est la taille de la zone de stockage ici?
```

Exemple de réception (serveur)

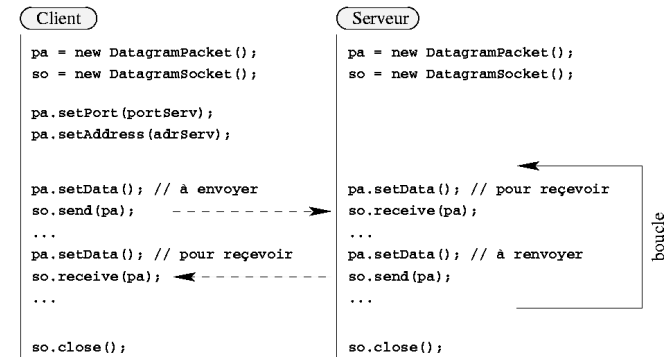
- Dans le cas d'un client, le choix du numéro de port d'attachement de la socket n'a pas d'importance
 - il sera connu par le serveur à la réception du datagramme émis par le client
- En revanche, dans le cas d'un serveur, il doit pouvoir être communiqué aux clients, afin qu'ils puissent l'interroger
 - nécessité d'attacher la socket d'écoute (d'attente des clients) à un port et une adresse spécifique et connue
 - utiliser un constructeur de `DatagramSocket` le spécifiant
 - risque que le port ne soit pas libre => `SocketException`

Réception serveur (suite)

```
> // socket d'attente de client, attachée au port 3333
DatagramSocket socket = new DatagramSocket(3333);
// datagramme pour la réception avec allocation de buffer
byte[] buf = new byte[1024];
DatagramPacket packet = new
DatagramPacket(buf,buf.length);
byte[] msg = "You're welcome!".getBytes("ASCII");
// message d'accueil

while (true) {
    socket.receive(packet); // attente de réception bloquante
    // place les données à envoyer
    // (@ip et port distant sont déjà ok)
    packet.setData(msg);
    socket.send(packet); // envoie la réponse
    packet.setData(buf,0,buf.length); // replace la zone de
    // réception
}
}
```

Modèle client-serveur



La pseudo-connexion

- > Dans le cas d'un serveur qui doit échanger plusieurs datagrammes consécutifs avec un client
 - > possibilité de ne considérer **que** ce client durant ce qu'on appelle une « **pseudo-connexion** »
 - > les sockets du serveur et du client sont dites « paires »
 - > `connect(InetAddress, int)` ou `connect(SocketAddress)` pour établir la pseudo-connexion
 - > `disconnect()` pour terminer la pseudo-connexion
 - > pendant ce temps, tous les datagrammes en provenance d'autres ports/adresses IP sont ignorés
 - > informations sur la pseudo-connexion: `isConnected()`, `getInetAddress()`, `getPort()`, `getRemoteSocketAddress()`

Communication en diffusion

- > D'un émetteur vers un groupe ou un ensemble de récepteurs
 - > le **broadcast** utilise les mêmes classes `DatagramSocket` et `DatagramPacket` que pour la communication *unicast*, avec une adresse destination de *broadcast*
 - > le **multicast** utilise la classe `DatagramPacket` pour les datagrammes mais la classe `MulticastSocket`, spécifique pour les sockets
- > Efficace lorsque le protocole de réseau sous-jacent offre la diffusion (ex. Ethernet):
 - > un seul datagramme peut être utilisé

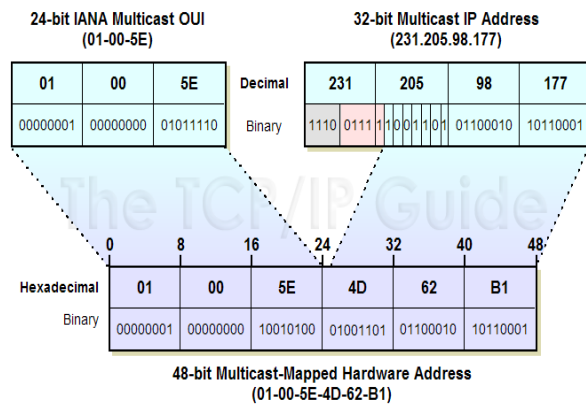
Le broadcast

- La classe `DatagramSocket` dispose de méthodes `[set/get]Broadcast()`
 - autorisation pour la socket d'émettre des *broadcasts*
 - `SO_BROADCAST true` par défaut
- En réception, une socket ne peut recevoir des *broadcasts* que si elle est attachée à l'adresse non spécifiée (*wildcard*)

Le multicast

- On s'adresse à un ensemble de machines (ou applications) ayant explicitement adhéré au groupe
 - adresses de classe D en IP v4: 224.0.0.0/4
 - adresses du réseau FF00::/8 en IP v6
 - les machines ayant rejoint un tel groupe acceptent les datagrammes à destination de l'adresse correspondante
- Traduction d'adresse IP multicast vers IP Ethernet
 - IP v4: l'@ Ethernet est le résultat du OU binaire entre les 23 bits de poids faible de l'adresse IP v4 et `01:00:5E:01:24:0C`
 - IP v6: l'adresse ffx:xxx:xxx:xxx:xxx:xxx:yyy:yyy produit l'adresse Ethernet `33:33:yy:yy:yy:yy`

Mapping multicast IPv4



java.net.MulticastSocket

- Hérite de `DatagramSocket`
- Trois constructeurs de `MulticastSocket`
 - sans arguments: attache à un port libre et à l'@ wildcard
 - numéro de port: attache à ce port et à l'@ wildcard
 - SocketAddress: attache à cette adresse de socket, ou bien n'attache pas la socket si l'argument vaut null
- Les constructeurs appellent la méthode `setReuseAddress(true)`
 - autorise plusieurs sockets à s'attacher à la même adresse
 - MÊME PORT** pour tous les membres du groupe

MulticastSocket en émission

- En émission, `MulticastSocket` s'utilise comme `DatagramSocket`
 - possibilité de spécifier une adresse IP source pour l'émission des datagrammes: `[set/get]Interface()`
 - possibilité de spécifier une interface de réseau pour l'émission: `[set/get]NetworkInterface()`
 - pour émettre vers le groupe, l'adresse de destination et le numéro du port dans le `DatagramPacket` doivent être ceux du groupe

MulticastSocket en émission (suite)

- On peut spécifier une « durée de vie » pour le datagramme: plus exactement, un nombre de sauts
 - 0=émetteur, 1=réseau local, 16= site, 32=région, 48=pays, 64=continent, 128=monde
 - méthodes `[set/get]TimeToLive()`
 - vaut 1 par défaut ⇒ propagé par aucun routeur
- Envoi des datagrammes multicast sur loopback
 - `[set/get]LoopbackMode()` peut être refusé par le système
- Suffit pour envoyer vers le groupe, pas pour recevoir.

MulticastSocket en réception

- Il faut explicitement « rejoindre » le groupe
 - `joinGroup()` avec l'adresse IP multicast du groupe en argument (existe aussi avec `SocketAddress` et/ou `NetworkInterface`)
 - permet alors à la socket de recevoir tous les datagrammes destinés à cette adresse multicast
 - peut rejoindre plusieurs groupes de multicast
 - `leaveGroup()` permet de quitter le groupe d'adresse spécifiée en argument

Le protocole TCP

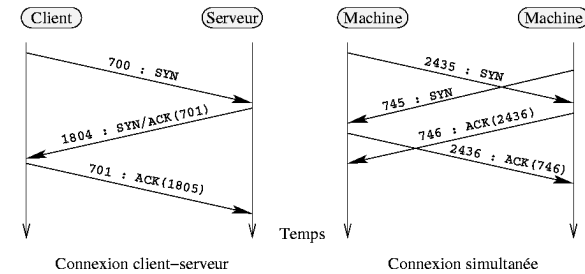
- Transmission Control Protocol, RFC 793
 - Communication
 - par flots,
 - fiable,
 - mode connecté
 - full duplex
 - Données bufferisées, encapsulées dans datagrammes IP
 - flot découpé en segments (~536 octets)
 - Mécanismes de contrôle de flot
 - ex: contrôle de congestion
 - assez lourd d'implantation (beaucoup plus qu'UDP)

Principe des segments

- La fiabilité est obtenue par un mécanisme d'**acquiescement des segments**
 - À l'émission d'un segment, une alarme est amorcée
 - Elle n'est désamorcée que si l'acquiescement correspondant est reçu
 - Si elle expire, le segment est réémis
- Chaque segment possède un **numéro de séquence**
 - préserver l'ordre, éviter les doublons
 - les acquiescements sont identifiés par un **marqueur ACK**
 - transport dans un même segment des données et de l'acquiescement des données précédentes: **piggybacking**

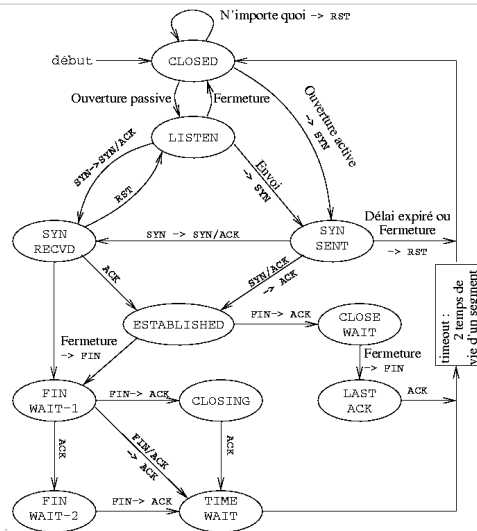
La connexion

- Three Way Handshake:
 - SYN --- SYN/ACK --- ACK

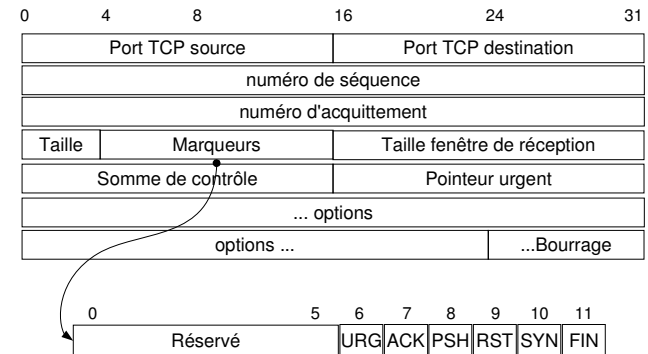


- Refus de connexion: SYN --- RST/ACK

Diagramme d'états



Format de trame TCP (en-tête)



Format (suite)

- Champ Taille
 - en nombre de mots de 32 bits, la [taille de l'en-tête](#)
- Marqueurs
 - URG données urgentes (utilisation conjointe pointeur urgent)
 - ACK acquittement
 - PSH force l'émission immédiate (w.r.t. temporisation par défaut)
 - RST refus de connexion
 - SYN synchronisation pour la connexion
 - FIN terminaison de la connexion
- Somme de contrôle (comme IP v4 avec un pseudo en-tête)
- Options
 - Exemple: taille max. de segment, estampillage temporel

Sockets en Java

- [java.net.ServerSocket](#)
 - représente l'objet socket en attente de connexion des clients (du côté serveur)
- [java.net.Socket](#)
 - représente une connexion TCP,
 - tant du côté client (instigateur de la connexion par la création d'un objet [Socket](#))
 - que du côté serveur (l'acceptation par une [ServerSocket](#) retourne un objet [Socket](#))
- L'attachement se fait, comme en UDP, à une adresse IP et un numéro de port

Socket du côté client

- Construire l'objet [Socket](#), éventuellement l'attacher localement, puis demander sa connexion à la socket d'un serveur
 - [Socket\(\)](#),
 - puis [bind\(SocketAddress bindPoint\)](#) pour attachement local, et
 - [connect\(SocketAddress\)](#) ou [connect\(SocketAddress, int\)](#) pour établir la connexion (int = timeout éventuel)
 - [Socket\(InetAddress addr, int port\)](#)
 - [Socket\(String host, int port\)](#)
 - [Socket\(InetAddress addr, int port, InetAddress localAddr, int localPort\)](#)
 - [Socket\(String host, int port, InetAddress localAddr, int localPort\)](#)

Les arguments de l'attachement

- Si [bind\(\)](#) avec une [SocketAddress](#) qui vaut [null](#)
 - choisit une adresse valide et un port libre
- Possibilités d'échec de l'attachement local
 - [BindException](#) (port demandé déjà occupé)
 - [SocketException](#) (la socket est déjà attachée)
 - peut être testé avec [isBound\(\)](#)
- Lors du [connect\(\)](#),
 - la socket est automatiquement attachée (si pas déjà fait)
 - initiation du *three way handshake* ([ConnectException](#))
 - méthode bloquante qui retourne normalement si succès

Utiliser la connexion

- Une fois l'objet socket construit, attaché et connecté, la connexion est établie
 - `InputStream getInputStream()` donne un flot de lecture des données arrivant sur la connexion
 - `OutputStream getOutputStream()` donne un flot d'écriture sur la connexion
 - les méthodes de lecture (`read()`) sont bloquantes
 - possibilité de limiter l'attente en lecture par `setSoTimeout(int milliseconds)`
 - au delà de cette durée, `read()` lève une `SocketTimeoutException`

Observations sur les sockets clientes

- Adresse IP et port d'attachement local
 - `getLocalAddress()`, `getLocalPort()`, `getLocalSocketAddress()`
- Adresse IP et port auxquels la socket est connectée
 - `getInetAddress()`, `getPort()`, `getRemoteSocketAddress()`
- Taille des zones tampons
 - `[set/get]SendBufferSize()`, `[set/get]ReceiveBufferSize()`
- Fermeture de la connexion: `close()`, `isClosed()`
 - la fermeture de l'un des flots ferme les deux sens de communication et la socket

Fermeture de socket

- Possibilité de ne fermer qu'un seul sens (*half-closed*)
 - `socket.shutdownOutput()` `isOutputShutdown()`
 - initie le *three way handshake* de déconnexion après l'émission de toutes les données présente dans le buffer d'émission
 - lève une `IOException` à chaque tentative d'écriture
 - `socket.shutdownInput()` `isInputShutdown()`
 - effet local: indication de fin de flot à chaque tentative de lecture
 - effacement après acquittement de toute donnée reçue
 - `close()` non bloquante,
 - mais socket reste ouverte tant qu'il y a des données
 - `setSoLinger(boolean on, int linger_sec)`
 - peut rendre la méthode `close()` bloquante, avec un timeout

Exemple socket cliente

```
➤ // Création de l'objet socket et connexion
Socket s = new Socket(serverName, port);
// Affichage des extrémités de la connexion
System.out.println("Connexion établie entre " +
    s.getLocalSocketAddress() + " et " +
    s.getRemoteSocketAddress());
// Récupération d'un flot en écriture et envoi de données
PrintStream ps = new PrintStream(
    s.getOutputStream(), "ASCII");
ps.println("Hello!");
// Récupération d'un flot en lecture et réception de données
// (1 ligne)
BufferedReader br = new BufferedReader(
    new InputStreamReader(s.getInputStream(), "ASCII"));
String line = br.readLine();
System.out.println("Données reçues : " + line);
s.close();
```

Configurations des sockets clientes

- Gestion données urgentes
 - `sendUrgentData(int octet)` côté émetteur
 - `setOOBInline(boolean on)` Si `true`, remplace les données urgentes dans le flot normal côté récepteur (éliminées par défaut, `false`)
- Emission forcée
 - `setTcpNoDelay(boolean on)` pour ne pas temporiser l'émission (débraye l'algorithme de Nagle qui remplit les segments au max.)
- Classe de trafic (*Type Of Service* ou *Flow Label*)
 - `[set/get]TrafficClass()` permet de manipuler: bits 2 (*coût monétaire faible*), 3 (*haute fiabilité*), 4 (*haut débit*) et 5 (*faible délai*). Ex: `s.setTrafficClass(0x02 | 0x10);`
- Etat de la connexion
 - `[set/get]KeepAlive()`: (détection de coupure) `false` par défaut

Socket du côté serveur

- `java.net.ServerSocket`
 - permet d'**attendre** les connexions des clients qui, lorsqu'elles sont établies, sont manipulées par un objet de la classe `Socket`
- Différents constructeurs
 - `ServerSocket()`, puis `bind(SocketAddress sockAddr)` ou `bind(SocketAddress sockAddr, int nbPendantes)`
 - Attachement local de la socket TCP permettant éventuellement de fournir le nombre de connexions pendantes (début du *three way handshake*, mais pas terminé, ou bien connexion pas encore prise en compte par l'application)

D'autres constructeurs

- `ServerSocket(int port)`,
`ServerSocket(int port, int nbPen)` ou
`ServerSocket(int port, int nbPen, InetAddress addr)`
- Si port 0, attachement à un port libre
 - nécessité de le divulguer pour que les clients puissent établir des connexions
- Observations sur l'attachement local:
 - `getInetAddress()`, `getLocalPort()` et `getLocalSocketAddress()`

Acceptation de connexion

- `Socket s = serverSock.accept();`
 - méthode bloquante, sauf si `setSoTimeout()` a été spécifié avec une valeur en millisecondes non nulle
 - L'objet `Socket` retourné est dit « **socket de service** » et représente la connexion établie (comme du côté client)
 - On peut récupérer les adresses et ports des deux côtés de la connexion grâce à cet objet socket de service
 - Si plusieurs sockets sont retournées par la méthode `accept()` du même objet `ServerSocket`, ils sont attachés au même port et à la même adresse IP locale
=> démultiplexage sur (*ipCli, portCli, ipSer, portSer*)

Exemple de serveur (pas très utile)

```
// Création et attachement d'un objet ServerSocket
ServerSocket ss = new ServerSocket(3333);
while (true) {
    Socket s = ss.accept(); // mise en attente de connexion
    // Récupération des flots de lecture et d'écriture
    BufferedReader br = new BufferedReader(
        new InputStreamReader(s.getInputStream(), "ASCII"));
    PrintStream ps = new PrintStream(
        s.getOutputStream(), true, "ASCII");
    System.out.println("Reçu: "+br.readLine());
    // Affiche le "Hello" reçu et envoie
    // inlassablement le même message
    ps.println("You're welcome!");
    s.close();
    // Ferme la socket "de service", pour en accepter une autre
}
```

Paramétrage de socket serveur

- Paramètres destinés à configurer les socket clientes acceptées
 - `[set/get]ReceiveBufferSize()`,
 - `[set/get]ReuseAddress()`
- Modifier l'implantation des sockets
 - `setSocketFactory(SocketImplFactory fac)`

Vue schématique d'une communication

