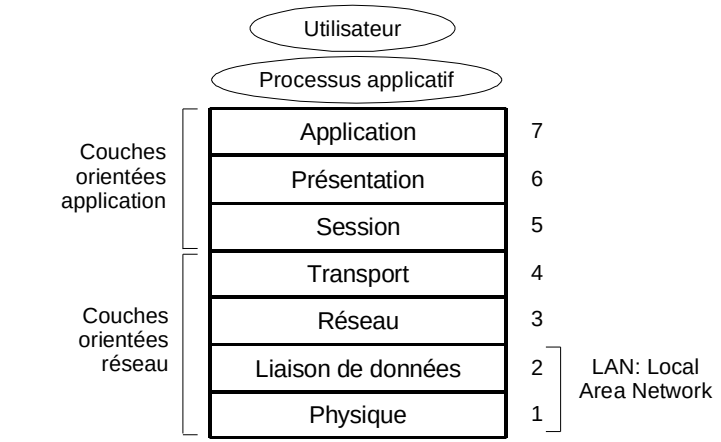
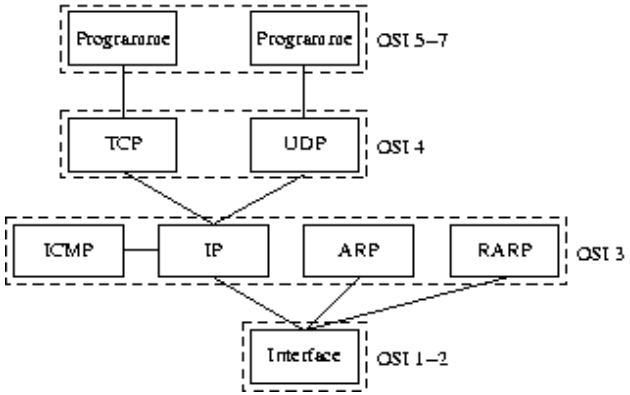
Le protocole Internet

- Réseau Internet
  - interconnexion des réseaux communiquant en utilisant le protocole Internet (IP)
  - Pour être utilisé, le protocole Internet de la couche réseau (3) requiert un protocole « compagnon » de la couche transport (4)
    - UDP (*User Datagram Protocol*)
    - TCP (*Transmission Control Protocol*)
  - Internet, Intranet et Extranet utilisent le même protocole
    - indépendant des fournisseurs,
    - propose des adresses logiques universelles,
    - la documentation est largement diffusée et souvent gratuite

Open System Interconnection (OSI)

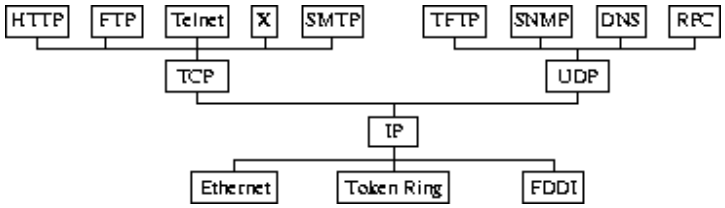


OSI et TCP/IP



Quelques protocoles au dessus d'IP

- De nombreux protocoles de la couche application sont supportés par UDP et / ou TCP



Historique

- 1969. Début de DARPAnet, réseau militaire USA
- 1972. Première démonstration de ARPAnet. Début des specifications des protocoles
- 1982. Premières interfaces de programmation: sockets Unix BSD
- 1983. TCP remplace NCP
- 1986. Mise en place du réseau NSFnet
- 1989. Naissance du protocole HTTP et du langage HTML
- 1992. Mise en place des réseaux EBONE et RENATER
- 1993. Premier véritable navigateur: Mosaic
- 1994. Les entreprises se connectent
- 1996. Début du 6bone, réseau mondial IP v6 (G6 en France)
- 2000. IP v6 disponible chez les constructeurs
- 2002. Plus de 500 millions d'internautes dans le monde
- 2004. Ajout des adresses IPv6 dans les serveurs DNS racines

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 Registery*). Exemple: RENATER
      - Délègue à l'administrateur (ex. Université)

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 indique 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
  - **1100**0000.00110101.00000110.00000010
  - 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.
  - « aggréger » 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

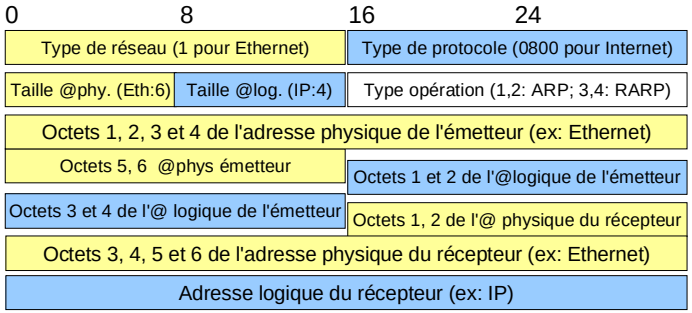
Adresses logiques, adresses physiques

- Sur même réseau physique, pas besoin de routeur
  - Point à point: aucune ambiguité
  - 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
  - Continent 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)



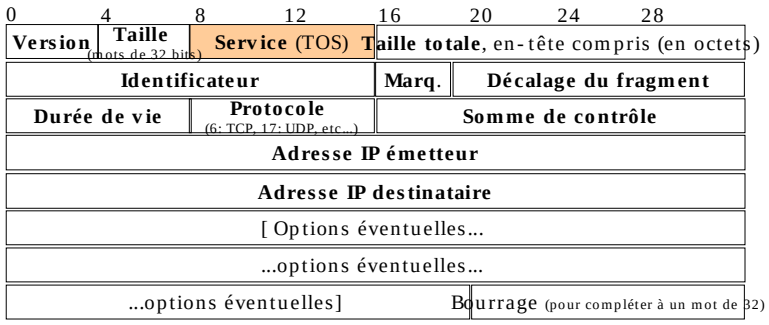
ARP (suite)

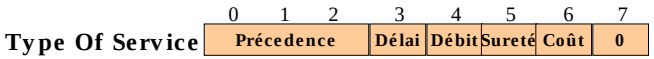
- ARP non sécurisé
  - Une machine ayant accès au réseau peut se faire passer pour une autre
    - *ARP cache poisoning*, attaque *Man In the Middle (MiM)*
  - Les broadcasts générés innondent tout le LAN
  - RARP (*Reverse APR*)
    - Obtenir une @ IP (dynamique) à partir d'une @MAC
  - Plus utilisé 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 RFC 815
  - 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)





Fragmentation

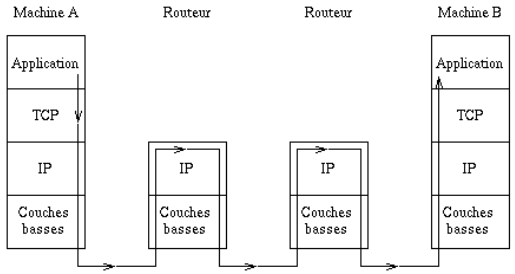
- MTU (*Maximum Transfert Unit*): taille maximum d'un datagramme IP transporté par un réseau physique
  - Ex: 1500 pour Ethernet, 4470 pour FDDI
  - Il peut être nécessaire de « découper » un datagramme pour adapter sa taille au MTU du réseau traversé
  - L'en-tête des différents fragments d'un même 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

Routage

- 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 Sytems*)
  - A l'interieur 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 Exceed 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
```


Noms et adresses Internet

- Les noms tels que etudiant.univ-mlv.fr ou java.sun.com ne sont pas indispensables, mais seulement pratiques à mémoriser
- Différents espaces de noms
  - IANA, mondial, standard, publique: le plus répandu
  - Autres: NetBios, AlterNIC, privés...
- Différents moyen d'associer 1 nom à 1 adresse IP
  - Fichier statique (`hosts`, `lmhosts`)
  - Interrogation dynamique « à la ARP » (NetBios)
  - Serveur de nom: DNS, WINS

Les noms IANA

- Composés de labels séparés par des points « . »
  - Chaque label fait au plus 63 caractères
  - Le nom complet fait au plus 255 caractères
  - Majuscules et minuscules sont indifférenciées
- Organisés hierarchiquement
  - 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.
- Commandes `host` ou `dig` permettent d'interroger le serveur DNS

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 primtifs 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("n.importe.quoi", new byte[]{1,2,3,4});
  System.out.println(bidon);       // affiche :   n.importe.quoi/1.2.3.4
  bidon = InetAddress.getByName("n.importe.quoi");
                                   // lève java.net.UnknownHostException
```

Observations sur les adresses

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

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<InetAddress>
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;
```

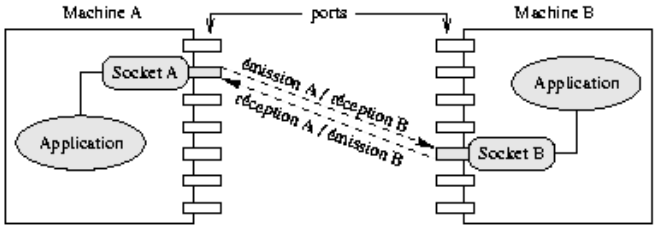
- getInetAddresses() sur une instance renvoit une énumération des adresses IP associées.
  - Principalement utilisé pour spécifier un attachement multicast sur une interface donnée.

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

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)

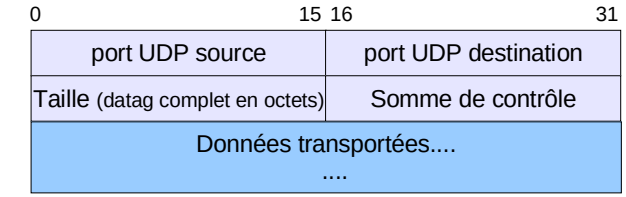
- Tous lèvent une IllegalArgumentException si le numéro de port est en dehors de [0..65535]
- Si le port spécifié vaut 0, un port éphémère sera choisi lors de l'attachement de la socket.
- La méthode toString() affiche **<@ip>:<n°port>**
-
```
InetSocketAddress sa1 =
    new InetSocketAddress("un.truc.zarb",50);
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}-1-8)~64Ko
- 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:

En-tête Ethernet	En-tête IP	En-tête UDP	Données transportées	CRC Ethernet

Accès Java à UDP

- Classiquement (avant 1.4), l'accès se fait grâce à deux classes
- 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

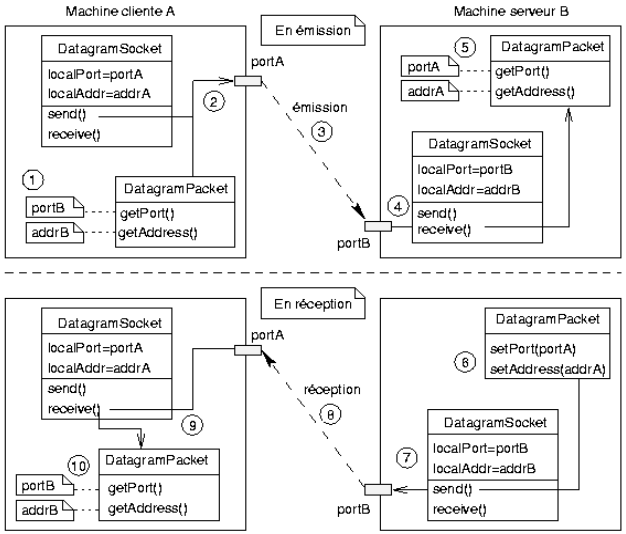
- En émission, le DatagramPacket spécifie
  - les données à envoyer
  - la machine (et le port) vers qui les envoyer
- En réception, le DatagramPacket spécifie
  - la zone de données permettant de recevoir
  - mettra à jour, lors de la réception, la machine et le port depuis lesquels ces données ont 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
- Le système peut fixer des tailles des tampons de réception et d'émission.
  - 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)

```
// 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)

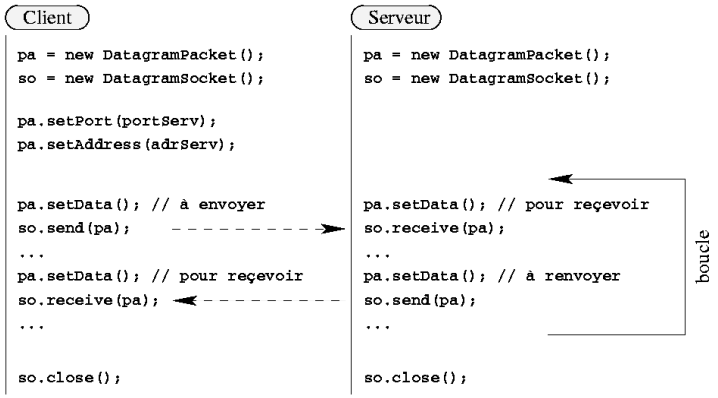
- Dans le cas d'un client, le choix du numéro de port d'attachement de la socket n'a pas d'importance
  - il sera connu par le serveur à la réception du datagramme émis par le client
- En revanche, dans le cas d'un serveur, il doit pouvoir être communiqué aux clients, afin qu'ils puissent l'interroger
  - 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()

Accès à UDP *via* les canaux

- Depuis jdk 1.4, java.nio.channels.DatagramChannel
- Canal vers une socket UDP
  - On peut créer une java.net.DatagramSocket à partir d'un DatagramChannel, mais pas le contraire
    - Si une socket UDP su a été créée à partir d'un canal, on peut récupérer ce canal par su.getChannel(). Sinon, cette méthode retourne null.
  - DatagramChannel.open() crée et retourne un canal associé à une socket UDP (non attachée)
  - DatagramChannel n'est pas une abstraction complète des sockets UDP: pour les opérations précises (binding, etc...) on récupère l'objet DatagramSocket sous-jacent

DatagramChannel

- Par défaut, un canal dc récupéré par Datagram-Channel.open() est bloquant. Il peut être configuré non bloquant.
  - dc.socket() récupère alors la DatagramSocket correspondante
  - Elle n'est pas attachée. On peut faire bind(SocketAddress) sur cette socket
- Les méthodes send() et receive() sont accessibles depuis le canal
  - Elles manipulent des ByteBuffer et receive() retourne un objet SocketAddress identifiant l'émetteur des données reçues
- On doit faire une pseudo-connexion pour pouvoir utiliser les méthodes read() et write() avec des ByteBuffer, plus classiques sur les canaux (interlocuteur implicite pour ces méthodes)

Envoi sur un DatagramChannel

- int send(ByteBuffer src, SocketAddress target)
- Provoque l'envoi des données restantes du tampon src vers target
- Semblable à un write() du point de vue du canal
- Si **canal bloquant**, la méthode retourne lorsque tous les octets ont été émis (leur nombre est retourné)
- Si **canal non bloquant**, la méthode émet tous les octets ou aucun
- Si une autre écriture est en cours sur la socket par une autre thread, l'invocation de cette méthode bloque jusqu'à ce que la première opération soit terminée

Réception depuis un DatagramChannel

- SocketAddress receive(ByteBuffer dst)
- Par défaut, méthode bloquante tant que rien n'est reçu par la socket
- Au plus dst.remaining() octets peuvent être reçus
  - Le reste est tronqué
- Retourne l'adresse de socket (IP+port) de l'émetteur des données
- Si **canal non bloquant**, soit tout est reçu, soit le tampon n'est pas modifié et la méthode retourne null
- Bug: on ne peut pas limiter le temps d'attente en lecture, comme c'est le cas avec DatagramSocket.setSoTimeout()
  - Dans ce cas, c'est ignoré par le flot de lecture sur le canal
  - Si besoin, il faut le faire « à la main »

Exemple de client UDP avec canaux : envoi

```
// Récupération de l'adresse IP et du port
InetAddress server = InetAddress.getByName(args[0]);
int port = Integer.parseInt(args[1]);
InetSocketAddress isa = new InetSocketAddress(server, port);
// Création d'un objet canal UDP non attaché
DatagramChannel dc = DatagramChannel.open();
// Les données à envoyer doivent être dans un ByteBuffer
ByteBuffer bb = ByteBuffer.wrap("Hello".getBytes("ASCII"));
// L'attachement de la socket UDP sous-jacente est implicite
dc.send(bb,isa);
dc.close();
```


Exemple de client UDP avec canaux : réception

```
// zone de données pour réception
ByteBuffer bb = ByteBuffer.wrap(new byte[512]);
// Création d'un objet canal UDP non attaché
DatagramChannel dc = DatagramChannel.open();
// La réception attache implicitement le canal UDP et retourne
// la socket de l'émetteur
SocketAddress sender = dc.receive(bb);
bb.flip();
System.out.println(bb.remaining() +
                   " octets reçus de " + sender + ":");
System.out.println(new String(bb.array(),0,
                              bb.remaining(), "ASCII"));
dc.close();
```

Exemple de serveur UDP avec canaux

```
// Création d'un objet canal UDP (non attaché)
DatagramChannel dc = DatagramChannel.open();
InetSocketAddress isa = new InetSocketAddress(port);
// attachement (explicite) de la socket sous-jacente
// au port d'écoute
dc.socket().bind(isa);
ByteBuffer bb = ByteBuffer.allocateDirect(512);
while (true) {
    SocketAddress sender = dc.receive(bb);  // réception
    bb.flip();
    dc.send(bb,sender);          // réponse à l'émetteur
    bb.clear();
}
```


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
- Pas encore d'accès via les canaux pour la diffusion (jdk1.7?)

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 ffxx:xxxx:xxxx:xxxx:xxxx:xxxx:yyyy:yyyy produit l'adresse Ethernet 33:33:yy:yy:yy:yy

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

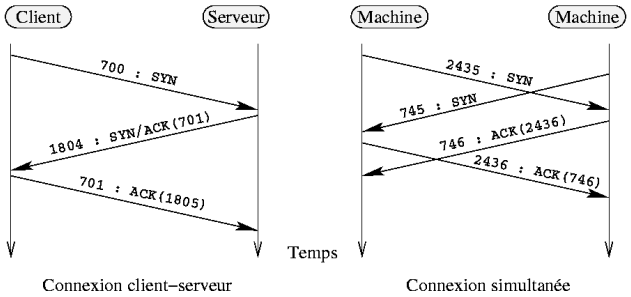
- Transmission Control Protocol, RFC 793
  - Communication
    - par flots,
    - fiable,
    - mode connecté
    - full duplex
  - Données bufferisées, encapsulées dans datagrammes IP
    - flot découpé en segments (~536 octets)
  - Mécanismes de contrôle de flot
    - ex: contrôle de congestion
  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

Principe des segments

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

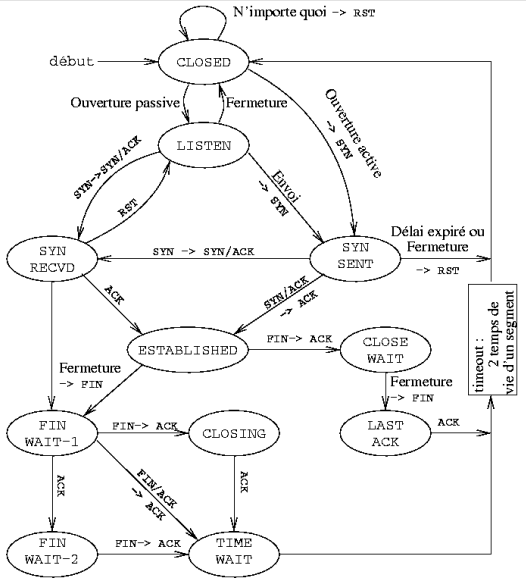
La connexion

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

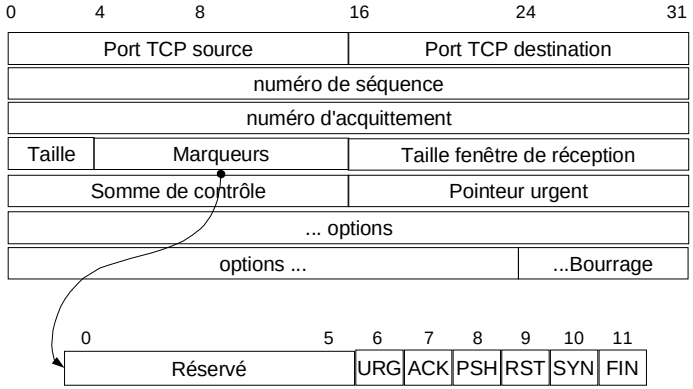


  - Refus de connexion: SYN --- RST/ACK

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)
  - inititation 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();
```


Quelques configurations des sockets clientes

- Gestion données urgentes
  - sendUrgentData(int octet) côté émetteur
  - setOOBInline(boolean on) Si true, replace 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)***

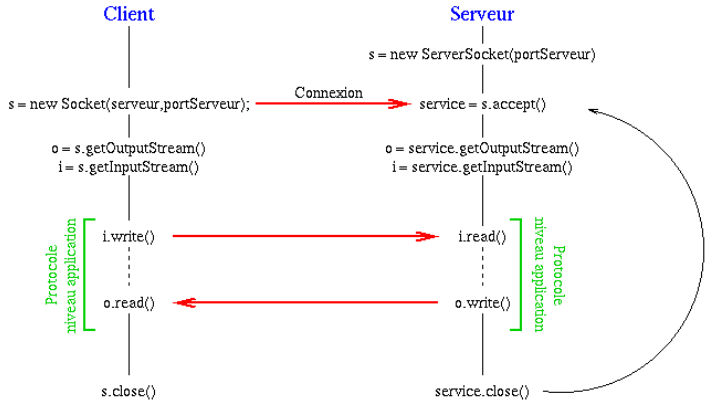
Exemple de serveur (pas très utile)

```
// 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



Canaux vers les sockets TCP

- java.nio.channels.ServerSocketChannel et SocketChannel
  - Acceptation de connexion et représentation des connexions
- SocketChannel représente un canal sur une socket TCP
  - Objet canal de socket TCP (non attaché) retourné par open()
  - Attachement éventuel de la socket sous-jacente Socket socket() Par un appel à bind() sur cet objet Socket
- ServerSocketChannel
  - Objet canal d'écoute de connexions entrantes retourné par open()
  - N'est que l'abstraction d'un canal sur une ServerSocket, récupérable par la méthode socket()
  - Il faut attacher cette ServerSocket par bind() avant de pouvoir accepter des connexions (sinon, NotYetBoundException)

ServerSocketChannel

- Appeler accept() sur le canal pour accepter des connexions (la server socket sous-jacente doit être attachée)
  - Si le canal est **bloquant**, l'appel bloque
    - Retourne un SocketChannel quand la connexion est acceptée ou
    - Lève une IOException si une erreur d'entrée-sortie arrive
  - Si le canal est **non bloquant**, retourne immédiatement
    - null s'il n'y a pas de connexion pendante
  - Quel que soit le mode (bloquant / non bloquant) du ServerSocketChannel le SocketChannel retourné est **initialement en mode bloquant**

SocketChannel

- boolean connect(SocketAddress remote) sur le canal
- Si SocketChannel en mode **bloquant**, l'appel à connect() bloque et retourne true quand la connexion est établie, ou lève IOException
- Si SocketChannel en mode **non bloquant**, l'appel à connect()
  - Peut retourner true immédiatement (connexion locale, par exemple)
  - Retourne false le plus souvent: il faudra plus tard appeler finishConnect()
- Tant que le canal est non connecté, les opérations d'entrée/sortie lèvent NotYetConnectedException (peut être testé par isConnected())
- Un SocketChannel reste connecté jusqu'à ce qu'il soit fermé
  - Possibilité de faire des half-closed et des fermetures asynchrones
  - AsynchronousCloseException (si canal fermé alors qu'une opération d'écriture en cours était bloquée, ou pendant l'opération de connection)
  - ClosedByInterruptException (si autre thread interromp la thread courante lorsqu'elle effectue une opération, IO, connection, etc.)

Établissement de connexion

- Méthode finishConnect()
  - Si connexion a échoué, lève IOException
  - Si pas de connexion initiée, lève NoConnectionPendingException
  - Si connexion déjà établie, retourne immédiatement true
  - Si la connexion n'est pas encore établie
    - Si mode **non bloquant**, retourne false
    - Si mode **bloquant**, l'appel bloque jusqu'au succès ou à l'échec de la connexion (retourne true ou lève une exception)
  - Si cette méthode est invoquée lorsque des opérations de lecture ou d'écriture sur ce canal sont appelés
    - Ces derniers sont bloqués jusqu'à ce que cette méthode retourne
  - Si cette méthode lève une exception (la connexion échoue)
    - Alors le canal est fermé

Communication sur canal de socket TCP

- Une fois la connexion établie, le canal de socket se comporte comme un canal en lecture et écriture
  - extends AbstractSelectableChannel et implements ByteChannel, ScatteringByteChannel, GatheringByteChannel
- Méthodes read() et write()
  - En mode **bloquant**, write() assure que tous les octets seront écrits mais read() n'assure pas que le tampon sera rempli (au moins un octet lu ou détection de fin de connexion: retourne -1)
  - En mode **non bloquant**, lecture comme écriture peuvent ne rien faire et retourner 0.
- Fermeture de socket par close() entraine la fermeture du canal

Les sélecteurs

- Objets utilisés avec les canaux configurés en mode **non bloquant** qui héritent de SelectableChannel
  - Ces canaux peuvent être enregistrés auprès d'un sélecteur après avoir été configurés non bloquant (configureBlocking(false))
  - java.nio.channels.Selector : les instances sont créées par appel à la méthode statique open() et fermés par close()

Les sélecteurs (suite)

- Enregistrement d'un canal auprès d'un sélecteur
  - Se fait par un appel, sur le canal à enregistrer, de:
  - SelectionKey register(Selector sel, int ops) ou SelectionKey register(Selector sel, int ops, Object att)
    - ops représente les opérations "intéressantes" pour ce sélecteur
    - SelectionKey retourné représente la **clé de sélection** de ce canal, qui permet de connaître, outre le sélecteur:
      - channel() renvoie le canal lui même
      - interestOps() renvoie les opérations « intéressantes »
      - attachment() renvoie l'objet att éventuellement attaché

Autour des sélecteurs

- keys() appelé sur un sélecteur retourne toutes ses clés
  - SelectionKey keyFor(Selector sel) sur un canal donne la clé de sélection du sélecteur pour ce canal
- On peut enregistrer plusieurs fois un même canal auprès d'un sélecteur (il met la clé à jour)
  - Il est plus élégant de modifier sa clé de sélection
    - En argument de interestOps() ou de attach() sur cette clé de sélection
- Les opérations intéressantes sont exprimées par les bits d'un entier (faire des OU binaires (|))
    - SelectionKey.OP_READ, SelectionKey.OP_WRITE, SelectionKey.OP_CONNECT, SelectionKey.OP_ACCEPT
  - Étant donné un canal, validOps() renvoit ses opérations « valides »

Utilisation des sélecteurs

- Appel à la méthode select()
  - Entraîne l'attente passive d'événements intéressants pour les canaux enregistrés
  - Dès qu'une de ces opérations peut être effectuée, la méthode retourne le nombre de canaux séléctionnés
  - Elle ajoute également les clés de sélection de ces canaux à l'ensemble retourné par selectedKeys() sur le selecteur
    - Il suffit de le parcourir avec un itérateur
  - C'est à l'utilisateur de retirer les clés de sélection correspondant aux canaux sélectionnés qu'il a « utilisé »
    - méthode remove() de l'ensemble ou de l'itérateur ou
    - méthode clear() de l'ensemble qui les retire toutes
- Si une clé est intéressée par plusieurs opérations
  - readyOps() donne celles qui sont prêtes
  - raccourcis isAcceptable(), isConnectable(), isReadable() et isWritable()

Sélection bloquante ou non bloquante

- Les méthodes select() ou select(long timeout) sont bloquantes
  - Elles ne retournent qu'après que
    - un canal soit sélectionné ou
    - la méthode wakeup() soit appelée ou
    - la thread courante soit interrompue ou
    - le timeout ait expiré
- La méthode selectNow() est non bloquante
  - Retourne 0 si aucun canal n'est sélectionné
- La méthode wakeup() permet d'interrompre une opération de sélection bloquante, depuis un autre processus léger
- L'annulation de l'enregistrement d'un canal auprès d'un sélecteur peut se faire par cancel() sur la clé de sélection. Elle est aussi réalisée implicitement à la fermeture du canal.

Fonctionnement des sélecteurs

- Un sélecteur maintient 3 ensembles de clés de sélection
  - ***key set***
    - Ensemble des clés de tous les canaux enregistrés dans ce sélecteur.
    - Méthode keys()
  - ***selected-key set***
    - Ensemble des clés dont les canaux ont été identifiés comme **prêts**
      - pour au moins une des opérations spécifiées dans l'ensemble des opérations de cette clé
      - à l'occasion d'**une opération de sélection précédente**
    - Méthode selectedKeys()
  - ***canceled-key set***
    - Ensemble des clés qui ont été annulées mais dont les canaux n'ont pas encore été désenregistrés de ce sélecteur
    - Cet ensemble n'est pas directement accessible

Opération de sélection

- 1. Chaque clé de ***canceled-key set*** est supprimée
  - De chacun des ensembles dans lesquels elle est présente
  - À la fin de 1. *canceled-key set* est vide
- 2. L'OS est interrogé
  - Pour savoir quels canaux sont prêts à réaliser une opération qui est décrite par sa clé comme "d'intérêt" au début de l'opération de sélection
  - Pour chacun des canaux prêts à faire quelque chose
    - Soit sa clé n'était pas déjà dans *selected-key set*, alors elle y est ajoutée avec un ensemble d'opérations prêtes reflétant l'état du canal
    - Soit sa clé était déjà dans *selected-key set*, alors son ensemble d'opérations prêtes est mis à jour par un OU binaire avec les anciennes
- 3. Si, pendant l'étape 2, des clés ont été ajoutée dans *canceled-key set*, alors elles sont traitées comme dans 1.

Sélection et concurrence

- Les sélecteurs sont "thread-safe", mais pas leurs ensembles
- Une opération de sélection synchronise:
  - 1. sur le sélecteur
  - 2. sur le *key-set*
  - 3. sur le *selected-key set*
  - Et sur le *canceled-key set* pendant les étapes 1. et 3. de la sélection
- Un thread bloqué sur select() ou select(long timeout) peut être interrompu par un autre thread de trois manières:
  - En appelant wakeup() sur le sélecteur (select() retourne)
  - En appelant close() sur le sélecteur (select() retourne)
  - En appelant interrupt() sur le thread
    - Pose le statut d'interruption et appelle la méthode wakeup()

Exemples

- Serveur TCP de mise en majuscule utilisant des canaux
  - Interrogeable avec netcat ou telnet
  - Différentes versions
- Itératif avec canaux bloquants
  - 1 thread
- Concurrent avec canaux bloquants
  - N threads
- Acceptation de nouvelles connexions bloquante, mais lecture non bloquante sur toutes les connexions établies
  - 2 threads
- Acceptation et lectures non bloquantes
  - 1 thread
  - D'abord sans, puis avec mécanisme d'attachement de gestionnaire