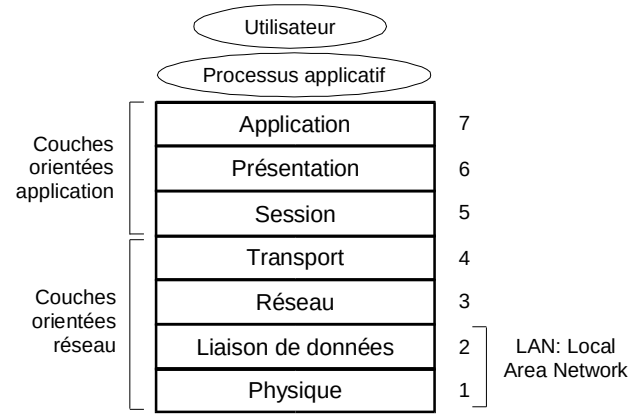
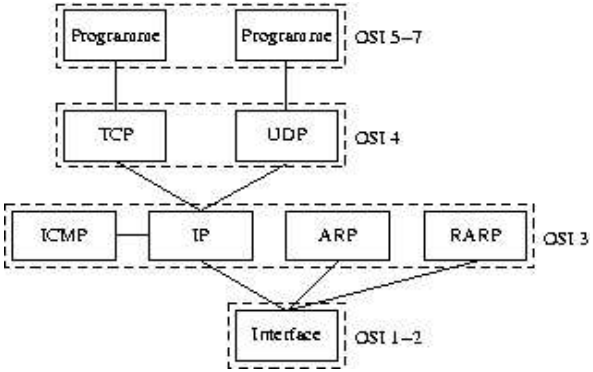
Le protocole Internet

- « 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)

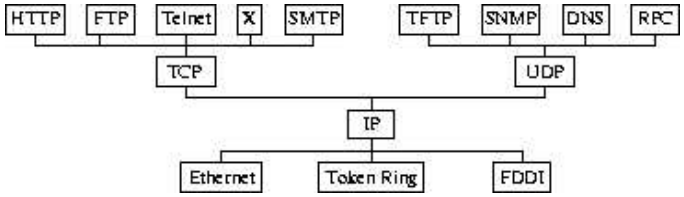


OSI et TCP/IP



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

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`**

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 hierarchiser un réseau (routage)
- L'utilisation des masques de réseau relève de la RFC 1519: **CIDR** (*Classless Inter Domain Routing*)

Adresses réservées
(ne peuvent pas être affectées à une machine)

- 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

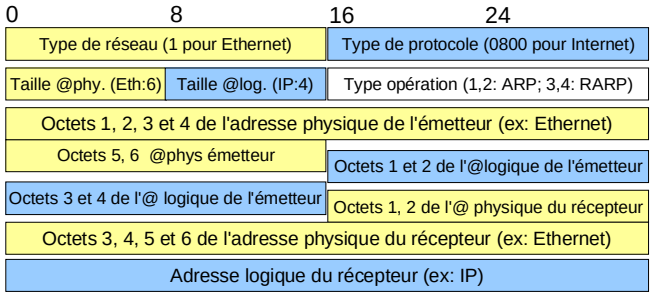
Adresses logiques, adresses physiques

- 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'adressse 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 @IPB donnée, une machine @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 répond à @IPA (si elle existe)
    - Les deux machines ont renseigné leurs tables ARP

ARP (format de trame)



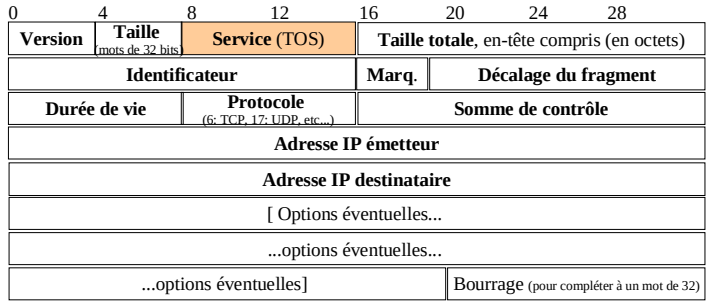
ARP (suite)

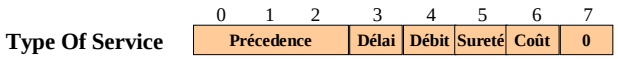
- ARP non sécurisé
  - Une machine ayant accès au réseau peut se faire passer pour une autre
    - *ARP cache poisoning*, attaque *Man In the Middle (MiM)*
- Les broadcasts générés « innondent » tout le LAN
- RARP (*Reverse APR*)
  - Obtenir une @ IP (dynamique) à partir d'une @MAC
  - Plus utilisé dans la pratique
- Remplacé dans la pratique par DHCP (*Dynamic Host Configuration Protocol*)
  - 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 diff.
  - possibilité de **fragmentation** des paquets

Format des datagrammes (IP v4)





Fragmentation

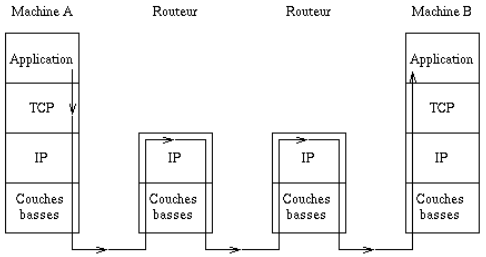
- MTU (*Maximum Transfert Unit*)
  - Taille maximum d'un datagramme IP transporté par un réseau physique
    - Ex: 1500 pour Ethernet, 4470 pour FDDI
  - Il peut être nécessaire de « découper » un datagramme pour adapter sa taille au MTU du réseau traversé
  - L'en-tête des différents fragments d'un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

Mais

```
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	*IPv4*
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	*IPv6*
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.

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 getNetworkInterfaces()
  - NetworkInterface getByName(String name)
  - NetworkInterface getByInetAddress(InetAddress addr)
- Peuvent lever des exceptions SocketException

Exemple de représentation

```
for (Enumeration 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;
```

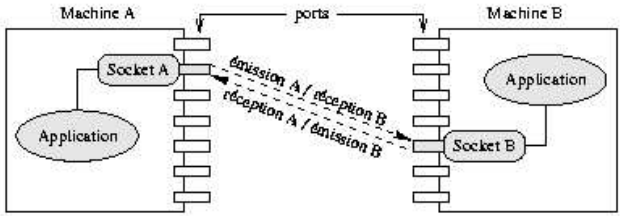
- La méthode getInetAddresses() sur une instance permet d'obtenir une énumération des adresses IP associées à une interface réseau.
- Principalement utilisé pour spécifier un attachement multicast sur une interface donnée.

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).

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>**

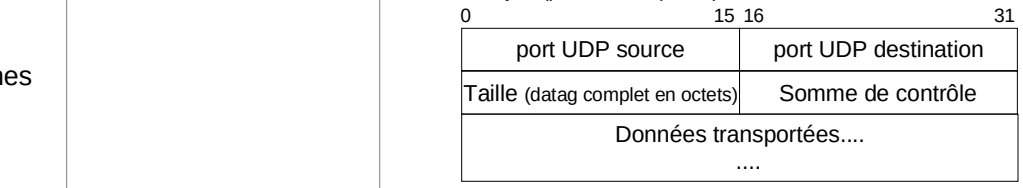
```
InetSocketAddress sa1 =
    new InetSocketAddress("un.truc.zarb",50);
System.out.println(sa1+(sa1.isUnresolved()?" non résolue":"résolue"));
    // Affiche : un.truc.zarb:50 non résolue
InetSocketAddress sa2 =
    new InetSocketAddress("java.sun.com",80);
System.out.println(sa2 + (sa2.isUnresolved()?" 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:

Accès Java à UDP

- 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

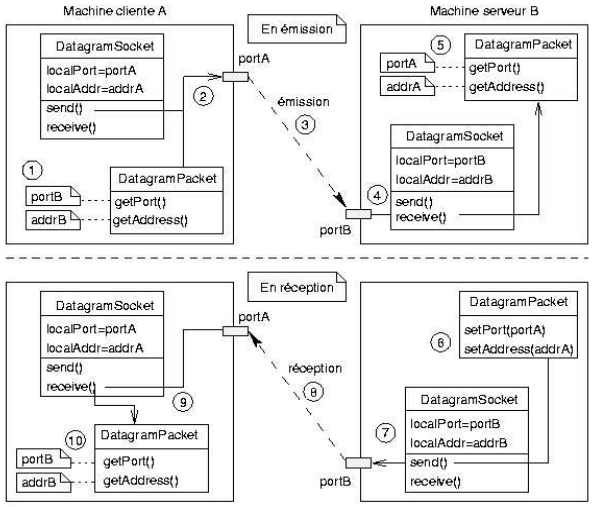
- En émission, le DatagramPacket spécifie
  - les données à envoyer
  - la machine (et le port) vers qui les envoyer
- En réception, le DatagramPacket spécifie
  - la zone de données permettant de recevoir
  - mettra à jour, lors de la réception, la machine et le port depuis lesquels ces données ont 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();
// 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());
System.out.println(s);
// Attention: quelle est la taille de la zone de stockage ici?
```


Exemple de réception (serveur)

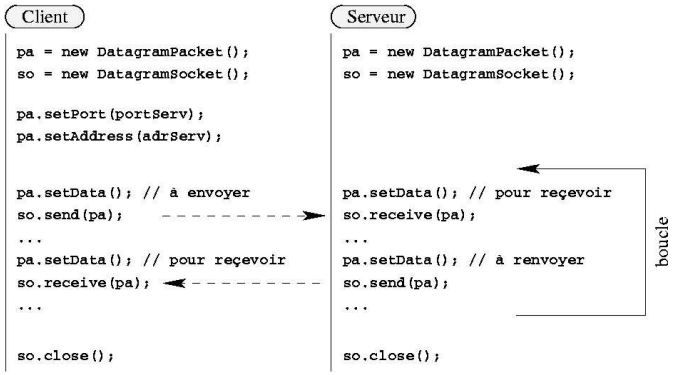
- Dans le cas d'un client, le choix du numéro de port d'attachement de la socket n'a pas d'importance
  - il sera connu par le serveur à la réception du datagramme émis par le client
- En revanche, dans le cas d'un serveur, il doit pouvoir être communiqué aux clients, afin qu'ils puissent l'interroger
  - 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();  // 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, 0, msg.length);
  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 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, mais 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

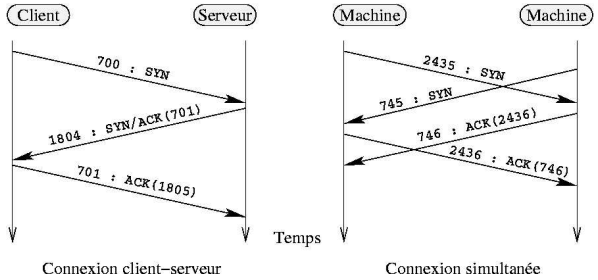
- Transmission Control Protocol, RFC 793
  - Communication
    - par flots,
    - fiable,
    - mode connecté
    - full duplex
  - Données bufferisées, encapsulées dans datagrammes IP
    - flot découpé en segments (~536 octets)
  - Mécanismes de contrôle de flot
    - ex: contrôle de congestion
  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

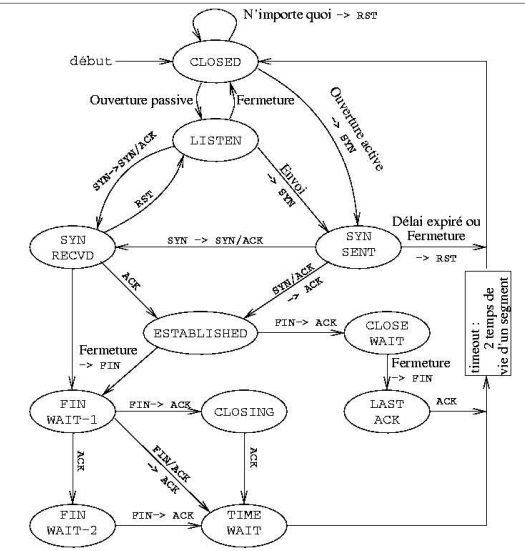
La connexion

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

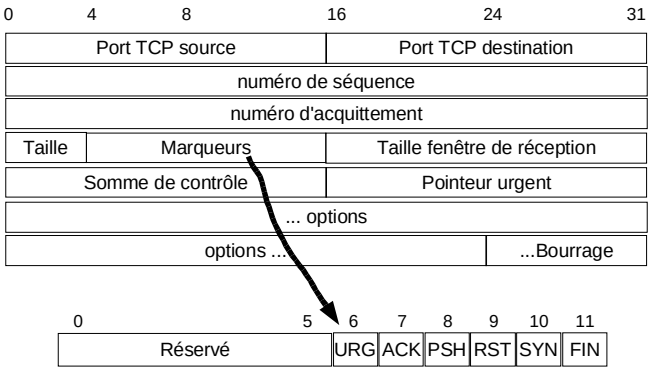


  - Refus de connexion: SYN --- RST/ACK

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
  - (ex. 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, éventuel[t] 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 commmunication 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());
ps.println("Hello!");
// Récupération d'un flot en lecture et réception de données (1 ligne)
BufferedReader br =
  new BufferedReader(new InputStreamReader(s.getInputStream()));
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 rempli 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(): par défaut false (détection de coupure)

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()

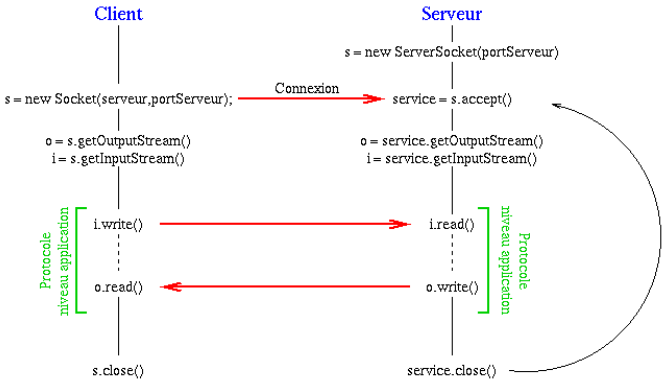
Acceptation de connexion

- Socket s = serverSock.accept();
  - méthode bloquante, sauf si setSoTimeout() a été spécifié avec une valeur en milli secondes 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)***

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



Typage des ressources Internet

- Types MIME (*Multi-purpose Internet Mail Extension*)
  - RFC 2046. Composé par un type et un sous-type
  - Les types principaux sont les suivants
    - text
    - image
    - audio
    - video
    - message
    - multipart
    - application
  - Ex: text/html, image/gif, application/octet-stream, etc.

Types MIME (2)

- Possibilité de spécifier un paramètre du sous-type
  - Après le sous-type et un point-virgule
  - Par exemple, codage des caractères pour le type texte
    - text/html; charset=iso-8859-1
- Pour les documents composites contenant différents types MIME
  - **multipart** avec **boundary** pour délimiter les parties
    - Sous-types:
    - mixed (par défaut),
    - alternative (la même information dans différents formats)
- La syntaxe des types MIME est ouverte

Les URI

- ***Uniform Resource Identifier*** (RFC 2396)
  - URL (Locator): localisation sur le réseau
  - URN (Name): indépendant de la localisation (site mirroir)
- URI identifie une ressource à partir d'un **schéma**
  - http, ftp, mailto, etc.
  - Attention: tous les URI ne sont pas nécessairement associés à des protocoles de communication
    - Exemple: URN:ISBN:2-7117-8689-7
  - Syntaxe générale:
    - **[<schéma>:]<partie dépendante du schéma>[#fragment]**

Syntaxe dans les URI

- Le schéma
  - doit commencer par une lettre minuscule, puis être composé de lettres minuscules, de chiffres et des signes « + », « . » et « - »
- La partie dépendante du schéma
  - Composée de caractères ASCII ou de caractères codés en hexadécimal de la forme « **%xx** »
  - Certains caractères peuvent avoir une interprétation spécifique (ils sont en général traduits automatiquement, si besoin, par les navigateurs. Il s'agit de l'espace et de
    " # < > ` | % { } \ ^ [ ] + ; / ? : @ = &

URI hierarchiques

- Certains URI sont dits « **hierarchiques** » (par opposition à « **opaques** »): leur partie dépendante du schéma peut être décomposée comme suit:
- **[<schéma>:][//<authority>][<path>][?<query>][#<fragment>]**
  - **<authority>**: autorité responsable du nommage de la ressource
  - **<path>**: chemin d'accès à la ressource: segments séparés par des « **/** » et suivis d'un ou plusieurs paramètres séparés par des « **;** »
  - **<query>**: chaîne de caractères interprétée par les deux champs précédents
  - Si l'autorité est liée à un serveur, elle a la forme suivante: **[<user-info>@]<machine>[:<port>]**
  - Si port n'est pas spécifié, port par défaut associé au schéma

Schémas http et https

- Schéma **`http`**
  - http://<machine>:<port>/<path>?<query>
  - Port par défaut **80**
  - <path> peut être interprété comme un chemin d'accès dans un système de fichiers, mais pas forcément
  - <query> informations supplémentaires (paramètres pour l'exécution d'une commande, par exemple)
- Schéma **`https`**
  - Protocole HTTP au dessus du protocole SSL
  - Même format, mais port par défaut **443**

Schéma ftp

- Transférer un fichier ou lister un répertoire
  - **`ftp://<user>:<passwd>@<machine>:<port>/<r1>/<r2>/...`**
    `.../name;type=<typecode>`
  - Si ni **`<user>`** ni **`<passwd>`** ne sont spécifiés, équivaut à ftp anonyme avec l'adresse e-mail en passwd
  - Port par défaut **21**
  - Suite **`<r1>/<r2>/.../name`** correspond au chemin d'accès relatif au répertoire de **`<user>`**.
  - **`;type=<typecode>`** est optionnel (typecode vaut « d » pour lister un répertoire, « i » pour binaire, « a » pour ASCII)
  - Exemples : **`ftp://ftp.inria.fr/rfc`**
    `ftp://duris:monmo2pas@igm.univ-mlv.fr/W3/img.gif;type=i`

Schéma mailto

- Désigne les boîtes aux lettres, courrier électronique
  - RFC 822
  - Historiquement URL, mais qqfois considéré comme URN
  - **`mailto:<adr1>,<adr2>?<header1>=<value1>&<header2>=<value2>`**
  - **`<adr i>`** sont les adresses destinataires du courrier
  - **`<header i>=<value i>`** sont des en-têtes à placer dans le courrier. Si **`<header>`** vaut **`body`**, c'est le corps du message
  - Seuls les en-têtes **`body`** et **`subject`** doivent être reconnus
  - Exemples:
    `mailto:roussel@univ-mlv.fr`
    `mailto:roussel@univ-mlv.fr,bedon@univ-mlv.fr?subject=test&body=test`

Schéma telnet

- Terminal interactif
- **`telnet://<user>:<passwd>@<machine>:<port>/`**
- Port par défaut **23**
- Ouvre un terminal sur la machine spécifiée, avec comme nom d'utilisateur **`<user>`**, à condition que le mot de passe soit valide

Schéma file

- Accès à un système de fichier (en général local)
- **`file://<machine>/<path>`**
- On n'utilise pas, en général, la partie machine, mais le « / » du path doit, en principe, être présent
- **`file:///home/duris/COURS/JAVA/http/progJavaHTTP.sdd`**
- Même pour des fichiers Windows, les séparateurs sont des « / »
- **`file:///C:/Program%20Files/`**

Les URI relatifs

- Concerne les URI hierarchiques
  - Parler d'une ressource « relativement » à un URI de base. Forme la + complète d'un URI relatif:
  - **`//<net_loc>/<path1>/.../<pathn>;<params>?<query>`**
  - Chacun des préfixes peut être omis et sera remplacé par l'URI de base (s'il est moins précis)
  - **`http://igm.univ-mlv.fr/~roussel/test/index.html`**
  - **`toto.html`** signifie alors **`http://igm.univ-mlv.fr/~roussel/test/toto.html`**
  - **`/~duris/index.html`** signifie alors **`http://igm.univ-mlv.fr/~duris/index.html`**
  - **`//etudiant.univ-mlv.fr`** signifie alors **`http://etudiant.univ-mlv.fr/`**

Le protocole HTTP

- (*Hyper Text Transfert Protocol*) RFC 2616 (version 1.1)
- Protocole **sans état**, **rapide**, **léger**, de niveau application permettant le **transfert de données** au dessus d'une **connexion** de type TCP (sûr)
  - Utilise les URLs
- Port par défaut **80**
- Basé sur le paradigme de **requête / réponse**
  - Chaque requête et chaque réponse est un « **message HTTP** »
- Largement utilisé sur le World Wide Web

Principe Requête/Réponse

- Connexion TCP établie entre un client et le port 80 (en général) du serveur HTTP
- Envoi par le client d'un message HTTP, dit *requête*
- Traitement de cette requête et envoi par le serveur d'un message, dit *réponse*, au client.
- En HTTP 0.9 et 1.0, la connexion est fermée par le serveur après chaque réponse
- En HTTP 1.1, il est possible de conserver la même connexion TCP pour plusieurs échanges de requêtes/réponses.

Format des messages HTTP

- **Ligne de début**
  Champ d'en-tête_1: valeur_1
  ...
  Champ d'en-tête_n: valeur_n
  ¶ *// ligne vide « \r\n »*
  [corps éventuel]
  - La présence de zéro, un ou plusieurs champs d'en-tête permettent de spécifier des paramètres
  - La présence d'un corps de message dépend de la ligne de début de ce message

Message de requête

- Constitution de la première ligne de requête
  - ***Méthode Ressource Version***
    Exemple: `GET /index.html HTTP/1.1`
    `Host: www.univ-mlv.fr`
- ***Méthode***: `GET, HEAD, POST, PUT, TRACE,...`
  - Représente le type d'opération à effectuer
- ***Ressource***: identifie la ressource recherchée
  - HTTP/1.1 exige que le champ « *Host* » soit spécifié
- ***Version***: identifie la version du protocole utilisée
  - Si rien n'est spécifié, la version HTTP 0.9 est considérée

Message de réponse

- Constitution de la ligne dite « **de statut** »
  - ***Version Code Message*** Exemple:

```
HTTP/1.1 200 OK
Server: Netscape-Enterprise/4.1
Date: Wed, 18 Sep 2002 09:30:24 GMT
...
Content-type: text/html
Content-length: 13298
¶
<!DOCTYPE HTML PUBLIC ....
<html><head>
...
</html>
```


Les codes de statut

- Organisés en 5 catégories
  - **1xx « informatifs »**: acceptés uniquement dans les versions 1.1. Réponse « intermédiaire » du serveur.
    Ex: `100 Continue`
  - **2xx « de succès »**: la réception, la compréhension ou l'acceptation de la requête est réussie. Ex: `200 OK`
  - **3xx « de redirection »**: d'autres actions sont requises pour traiter la requête. Ex: `301 Moved Permanently`
  - **4xx « erreurs du client »**: erreur de syntaxe ou de spécification. Ex: `404 Not Found`
  - **5xx « erreurs du serveur »**: serveur incapable de répondre. Ex: `501 Method Not Implemented`

Quelques champs d'en-tête

- Spécifiques (requêtes/réponses) ou bien généraux:
- **Server** (réponse): informations sur le serveur HTTP
- **Date** (général): estampille les messages HTTP
- **Last-modified** (serveur): date de dernière modification
- **ETag** (): identifiant de la ressource géré par le serveur
- **Content-type** (général): type le corps du message
- **Content-length** (général): nombre d'octets du corps
- **Accept: text/plain** (requête): ce qu'accepte le client
- **Accept-charset**, **Accept-language**, **Accept-encoding**
  Content-encoding, **Transfert-encoding...**

Quelques requêtes

- `GET`: récupérer une ressource
- `HEAD`: identique à `GET`, mais sans corps de msg
  - Peut servir à savoir si une ressource existe et si elle à changé, sans nécessairement la récupérer
- `TRACE`: écho de la requête
  - le serveur renvoit, dans le corps du message, le message qu'il a reçu du client (diagnostique)
- `OPTIONS`: demande d'information sur les méthodes
  - Le serveur renvoit toutes les méthodes qu'il implante dans le champ d'en-tête « `Allow` »

Connexion persistante ou "keep alive"

- Avant 1.1, champ **`Content-length`** pas obligatoire:
  - Comme le serveur fermait la connexion à la fin de la réponse, ce champ servait simplement à vérifier que OK
- Avec la **connexion persistante** (***keep alive***)
  - Serveur peut garder la connexion après la fin de réponse
  - Plus efficace (délais d'ouverture et fermeture de connexion)
  - Possibilité d'exploiter le full-duplex de la connexion TCP (plusieurs transactions peuvent se recouvrir)
- Toutes les réponses doivent contenir **`Content-length`** (sauf réponses sans corps de message ou `chunked`)
  - Indispensable pour déterminer la fin d'un message


```
% telnet www.w3.org 80
Trying 193.51.208.67...
Connected to www.w3.org.
Escape character is '^]'

GET /Overview HTTP/1.1
Host: www.w3.org

HTTP/1.1 200 OK
Date: Sun, 30 Jun 2002 15:33:43 GMT
Server: Apache/1.3.6 (Unix) PHP/3.0.9
Last-Modified: Sat, 29 Jun 2002 16:08:59 GMT
ETag: "2d2c96-2c69-3778ef9b"
Accept-Ranges: bytes
Content-Length: 11369
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC ...
<html lang="en"> ... le code du document html ...
</html>
```

La connexion n'est pas fermée...

```
HEAD /Overview HTTP/1.1
Host: www.w3.org

HTTP/1.1 200 OK
Date: Sun, 30 Jun 2002 15:33:58 GMT
Server: Apache/1.3.6 (Unix) PHP/3.0.9
Last-Modified: Sat, 29 Jun 2002 16:08:59 GMT
ETag: "2d2c96-2c69-3778ef9b"
Accept-Ranges: bytes
Content-Length: 11369
Content-Type: text/html; charset=iso-8859-1

OPTIONS / HTTP/1.1
Host: www.w3.org
Connection: close

HTTP/1.1 200 OK
Date: Sun, 30 Jun 2002 15:34:17 GMT
Server: Apache/1.3.6 (Unix) PHP/3.0.9
Content-Length: 0
Allow: GET, HEAD, POST, PUT, DELETE, CONNECT, OPTIONS, PATCH,
  PROPFIND, PROPPATCH, MKCOL, COPY, MOVE, LOCK, UNLOCK, TRACE
Connection: close

Connection closed by foreign host.
%
```

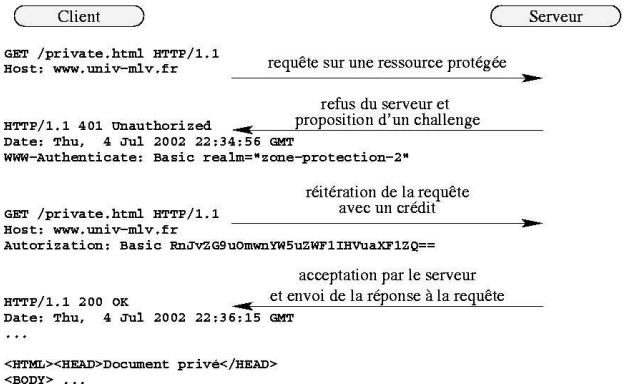

L'authentification

- Pour limiter l'accès de ressources à qq clients
  - Mécanisme de *challenge/crédit* pour l'authentification des requêtes
  - Utilise les champs **`WWW-Authenticate`** et **`Autorization`**
- Lors d'une requête sur une ressource protégée
  - Serveur répond **`401 Unauthorized`** avec, un champ **`WWW-Authenticate`** qui contient au moins un *challenge* applicable à la ressource concernée (une liste)
  - Un challenge est constitué d'un schéma (mot, ex: Basic), suivi de paires attribut=valeur. Exemple:
    `Basic realm= "zone-protection-2"`

L'authentification (suite)

- Le client doit alors réémettre sa requête avec un *crédit* dans le champ **`Autorization`**. Exemple:
  - **`Autorization: Basic blablabla`**
    où **`blablabla`** représente le codage en base 64 de la chaîne « login:passwd »
  - Attention, base 64 est réversible, et le crédit passe en clair
- Autre schéma: ***Digest***
  - Requiert une valeur d'autorisation obtenue par hachage cryptographique utilisant, en plus de user et passwd de client, des informations liées à la requête et un valeur aléatoire tirée par le serveur

Authentification (exemple)



Envoyer des données

- Une requête peut servir à envoyer des informations vers le serveur HTTP
  - Méthode POST, dans le corps du message, mais aussi
  - Méthode GET, dans l'URL sous la forme de « *query string* »: permet de transmettre quelques arguments
- Ces données sont principalement utilisées pour « générer » une ressource dynamiquement
  - Utilise des « programmes » satellites au serveur
  - CGI, Servlettes/JSP, PHP, ASP...

Les formulaires HTML (GET)

- Permet de définir des zones de saisie pour le client
  - <form enctype="application/x-www-form-urlencoded"
    action="http://www.serv.fr/path"
    method="**GET**">
    <input type="text" name="user" value="votre nom">
    <input type="submit" name="ok" value="Envoyer">
    </form>

  - Un click sur Envoyer crée une requête GET à l'URL suivant: http://www.serv.fr/path?user=Etienne+Duris

Les formulaires HTML (POST)

- <form action="http://www.serv.fr/path" method="**POST**">
  <input type="text" name="user" value="votre nom">
  <input type="submit" name="ok" value="Envoyer">
  </form>
- Même apparence dans un navigateur
- Un click sur Envoyer crée une requête POST
  - à l'URL http://www.serv.fr/path,
  - Avec comme champs d'en-tête
    Host: www.serv.fr
    Content-Type: application/x-www-form-urlencoded
    Content-Length: 14
  - dont le **corps du message** contient
    user=votre+nom

Formulaire plus riche (illustration)

```
<form enctype="application/x-www-form-urlencoded"
  action="http://www.serv.fr/path" method="GET">
  Zone de texte (nom) :
  <input type="text" size="30" name="user" value="votre nom"><hr>
  <textarea name="srctext" rows="6" cols="40">
   Tapez votre message ici.
  </textarea><hr>
  Zone de boutons (age) : Quel âge avez vous?
  moins de 18 ans <input type="radio" name="age" value="-18">
  de 19 à 35 ans <input type="radio" name="age" value="19-35">
  de 36 à 50 ans <input type="radio" name="age" value="36-50" checked>
  plus de 50 ans<input type="radio" name="age" value="50-"><hr>
  Zone de menu (disponibilité) : Vous êtes
  <select name="dispo">
    <option value="ok">disponible
    <option selected value="ko">indisponible
  </select><hr>
  <input type="hidden" name="client" value="a1234-567-8901">
  <input type="submit" name="choix1" value="Accepter">
  <input type="submit" name="choix2" value="Refuser">
  <input type="reset" value="Réinitialiser">
  <hr>
</form>
```


Corps de message « par morceaux »

- Lorsqu'une requête génère une réponse longue
  - On devrait attendre toute la génération de la réponse afin de connaître sa taille AVANT de créer l'en-tête de la réponse et son champ **`Content-Length`**
  - On peut alors « découper » la réponse en morceaux (***chunked***) et commencer à envoyer les premiers avant d'avoir fini de générer la totalité du document
  - Utilise le champ **`Transfert-Encoding: chunked`**
  - Il n'y a plus de champ d'en-tête **`Content-length`**
  - Chaque morceau est préfixé par sa taille
  - Le premier morceau de taille nul indique la fin du corps

Chunked (exemple)

```
GET /search?q=test HTTP/1.1
Host: www.google.fr

HTTP/1.1 200 OK
Server: GWS/2.0
Date: Mon, 23 Sep 2002 11:58:56 GMT
Transfer-Encoding: chunked
Content-Type: text/html

b3f
<html><head>
..... Premier morceau ...
3ce2
..... Deuxième morceau ...
</body></html>
0
```


Envoyer des fichiers

- Si le formulaire spécifie un nom de fichier, c'est le contenu du fichier qu'on désire envoyer
  - le type (enctype) ne doit alors plus être "**application/x-www-form-urlencoded**", mais "**multipart/form-data**", introduit en HTML 4 pour cela

Envoyer des fichiers (code)

```
<form enctype="multipart/form-data"
  action="/sendfile" method="POST">
  Fichier 1 : <input type="file"
              name="fichier1" size="20"><br><br>
  Fichier 2 : <input type="file"
              name="fichier2" size="20"><br><br>
  <input type="hidden" name="ID" value="roussel">
  <input type="submit" name="submit" value="Envoyer">
</form>
```

- Supposons que le fichier contienne le texte suivant:
  - Test de fichier
- Alors la requête générée par un click sur Envoyer sera la suivante...


```
POST /sendfile HTTP/1.1
Host: server
Content-Type: multipart/form-data; boundary=ABCDEF
Content-Length: 372

--ABCDEF
Content-Disposition: form-data; name="fichier1"; filename="test.txt"
Content-Type: text/plain

Test de fichier

--ABCDEF
Content-Disposition: form-data; name="fichier2"; filename=""
Content-Type: application/octet-stream


--ABCDEF
Content-Disposition: form-data; name="ID"

roussel
--ABCDEF
Content-Disposition: form-data; name="submit"

Envoyer
--ABCDEF--
```


Requête GET ou POST

- Les deux méthodes sont utilisables dans un formulaire
- Différences principales
  - POST transmet les données dans le corps
  - GET transmet les données dans l'URL
  - POST + Ésthétique (arguments n'apparaissent pas)
  - POST non limité en taille (les serveurs limitent une taille maximale pour les URLs). Dans ce cas, code de statut `414 Requested-URI Too Large`
  - Un formulaire GET peut être « rejoué » en conservant l'URL (bookmark du navigateur)

D'autres requêtes

- PUT permet d'envoyer au serveur HTTP des données (fichiers) qu'il devra stocker
  - Différent de POST, dont les données sont destinées à une ressource identifiée par l'URL (qui doit normalement interpréter les données)
  - Dans le cas de PUT, les données **sont** la ressource à placer à l'endroit spécifié par l'URL
  - Mise à jour (`200 OK`) ou création (`201 Created`)
- DELETE permet de supprimer des données
- Requierent le droit d'accès (authentification)

Les relais

- Modèle client/serveur trop simpliste
  - Beaucoup de machines intermédiaires qui jouent le rôle de serveur pour le client et de client pour le serveur
  - Peuvent également stocker temporairement des réponses pour les « rejouer » à la prochaine requête identique: ***proxy cache***
  - Un client peus spécifier explicitement (statiquement) qu'il veut utiliser un proxy
  - Nombreux champs d'en-tête spécifiques
  - **`Proxy-Authenticate, Proxy-Authorization, Via, Max-Forwards, Cache-Control (Pragma: no-cache), Vary...`**

Session Tracking

- HTTP protocole sans état. Comment savoir quelle était la précédente requête du même client?
- Idée: faire « sauvegarder un état » par le client, qu'il devra renvoyer dans la prochaine requête
  - Champs « **`hidden`** » des formulaires
  - **Réécriture des URL** des liens hypertextes conten us dans les documents retournés. Exemple:
    `<a href="http://serveur/request?user=duris">`
    Ne requiert aucune participation du client
  - **Cookies** (plus simple pour le serveur)
    introduit par Netscape, puis RFC 2109
    Requiert la participation du client

Cookies

- C'est une paire <clé>=<valeur>
  - Initialement fournie par le serveur dans une réponse
  - Stockée par le client (navigateur)
  - Puis retransmise par le client lorsqu'il émet des requêtes
- Spécifié par un champ d'en-tête dans la réponse
  - **`Set-Cookie: <clé>=<valeur> [; expires=<DATE>]`**
    `[; path=<PATH>]`
    `[; domain=<DOM>]`
    `[; secure]`
- Rejoué par les clients:
  - **`Cookie: <clé1>=<valeur1>; <clé2>=<valeur2> ...`**

Cookies (suite)

- Attributs optionnels
  - **expires**: pour la date de péremption. Au delà, le client ne doit pas conserver le cookie
  - **domain** et **path** servent à déterminer les URL pour lesquels ce cookie doit être utilisé
    (par défaut, ceux de l'URL de la requête sont utilisés)
    - Les cookies dont la valeur de domain est différente du domaine du serveur ayant généré la réponse doivent être ignorés
    - Les cookies trop larges (moins de deux « . ») ignorés
  - **secure**: cookie à n'utiliser que si connexion sécurisée
  - Si même path et même domain, le nouveau écrase l'ancien. Si path diffère, on stocke et renvoit les deux.