# Programmation réseau en Java

Michel Chilowicz

EIOC3


Version du 19/09/2013



## Plan

# À la découverte du paquetage *java.net*

## Composants essentiels

- Les adresses IP : classe *InetAddress*
- Les interfaces réseau : classe *NetworkInterface*
- Les sockets UDP : classes *DatagramSocket* et *MulticastSocket* (paquet UDP : classe *DatagramPacket*)
- Les sockets TCP : classes *ServerSocket* et *Socket*
- Connexions niveau applicatif : classe *URLConnection* (pour gérer par exemple une connexion HTTP)

## Implantations compatibles

- OpenJDK (implantation de référence du JDK)
- API Android



# Modèle OSI : ce qu'il est possible de faire (ou pas) avec l'API standard Java

1. Physique
2. Liaison : envoi des trames Ethernet brutes impossible
3. Réseau : test de connectivité ICMP possible, envoi de paquets IP bruts impossible (pas de support des rawsockets)
4. Transport : support de UDP, TCP, SCTP (non-officiel)
5. Session : support de TLS sur TCP (*SSLServerSocket*, *SSLSocket*)
6. Application : résolution DNS, connexions HTTP...

# Fonctionnement interne de l'API réseau

- Repose sur les appels systèmes POSIX : getaddrinfo, socket, recv, send, accept...
- Code compilé natif de liaison avec les appels systèmes pour chaque plate-forme supportée (Windows, Linux, FreeBSD...)
- Interfaçage avec le code natif avec *Java Native Interface* (JNI)

$\longrightarrow$ l'API Java est une surcouche pour les appels systèmes réseau



# Concurrence

Deux approches pour gérer de multiples communications réseau :

## Approche bloquante multi-thread

- Un flot de communication = 1 thread
- Opérations d'E/S bloquantes (main redonnée à une autre thread pendant le blocage)

## Approche non-bloquante évènementielle

- Une seule thread
- Utilisation d'un sélecteur indiquant les flots de communication disponibles
- Opérations d'E/S non-bloquantes : retour immédiat si rien ne peut être lu ou écrit
- Approche implantée depuis Java 1.4 dans *java.nio*

# Deux protocoles IP (Internet Protocol) co-existants sur le net

## IPv4

Première version historique décrite dans la RFC 791 en 1981
Espace d'adressage de taille $2^{32}$

## IPv6

Nouvelle version introduite en 1996 pour résoudre la pénurie d'adresses IPv4
Espace d'adressage de taille $2^{128}$
Version en cours d'adoption

Les adresses IPv4 et IPv6 sont attribuées par l'Internet Assigned Numbers Authority (IANA)



# `java.net.InetAddress` : une adresse IP en Java

- Deux classes dérivées pour les adresses IPv4 (Inet4Address) et IPv6 (Inet6Address)
- Pas de constructeurs pour initialiser une adresse IP mais des méthodes statiques :
  - Depuis une adresse sous forme textuelle : `InetAddress.getByName(String addr)`
  - Depuis une adresse sous forme octale : `InetAddress.getByAddress(byte[] addr)`
  - Si l'on souhaite obtenir l'adresse localhost : `InetAddress.getLocalHost()`

# Quelle est donc cette adresse mystérieuse ?

```
public static void d(String s) { System.out.println(s); }

public static void displayAddressInfo(byte[] addr)
{
        // D'abord on récupère un objet InetAddress
        InetAddress ia = InetAddress.getByAddress(addr);
        // On affiche l'adresse sous forme textuelle
        d("Adresse texte : " + ia.getHostAddress());
        // On rappelle l'adresse octale
        d("Adresse octale : " + Arrays.toString(ia.getAddress()));
        // Ensuite on affiche les informations associées
        d("Adresse joker ? " + ia.isAnyLocalAddress());
        d("Adresse de lien local ? " + ia.isLinkLocalAddress());
        d("Adresse de boucle locale ? " + ia.isLoopbackAddress());
        d("Adresse de réseau privé ? " + ia.isSiteLocalAddress());
        d("Adresse de multicast ? " + ia.isMulticastAddress());
        if (ia.isMulticastAddress())
        {
                // Testons la portée du multicast
                d("Portée globale ? " + ia.isMCGlobal());
                d("Portée organisationnelle ? " + ia.isMCOrgLocal());
                d("Portée site ? " + ia.isMCSiteLocal());
                d("Portée lien ? " + ia.isMCLinkLocal());
                d("Portée noeud ? " + ia.isMCNodeLocal());
        }
}
```



# Conversion d'une adresse IPv4 octale en texte

## Méthode paresseuse

```
public String convert(byte[] addr) throws UnknownHostException
{
        return InetAddress.getByAddress(addr).getHostAddress();
}
```

## Méthode plus laborieuse

```
public int toUnsigned(byte b) { return (int)b & 0xff; }
public String convert(byte[] addr) throws UnknownHostException
{
        return String.format("%d.%d.%d.%d",
                toUnsigned(addr[0]), toUnsigned(addr[1]),
                toUnsigned(addr[2]), toUnsigned(addr[3]));
}
```

# Conversion d'une adresse IPv4 texte en octale

## Méthode paresseuse

```
public byte[] convert(String textAddr) throws UnknownHostException
{
        return InetAddress.getByName(textAddr).getAddress();
}
```

## Méthode plus laborieuse

```
public byte[] convert(String textAddr)
{
        String[] split = textAddr.split("\\.");
        byte[] addr = new byte[split.length];
        for (int i = 0; i < addr.length; i++)
                addr[i] = (byte)Integer.parseInt(split[i]);
        return addr;
}
```



# Test de connectivité d'une adresse IP

La méthode `boolean isReachable(int timeout) throws IOException` d'une instance d'`InetAddress` permet de tester la connectivité de l'adresse.

## Comment ça marche ?

1. On envoie soit des messages *ICMP echo request* (ping), soit on tente d'ouvrir une connexion TCP sur le port 7 (service echo) de l'hôte
2. L'hôte est considéré accessible ssi une réponse parvient avant *timeout* ms

## Cas de réponses négatives

- Machine hors-ligne
- Machine ne répondant pas aux pings et sans service TCP echo
- Firewall filtrant rencontré sur le chemin de routage

# Domain Name System

- Mémoriser des adresses IPv4 est difficile, des adresses IPv6 une torture
  $\longrightarrow$ nécessité d'un système d'indirection liant des noms faciles à retenir à des adresses IP
  $\longrightarrow$ solution proposée : Domain Name System (DNS) avec la RFC 1035
- Système hiérarchique de nommage chapeauté par l'Internet Corporation for Assigned Names and Numbers (ICANN) $\longrightarrow$ chaque zone DNS délègue l'administration de sous-noms à d'autres autorités
- Exemple de délégation : `igm.univ-mlv.fr.`
  - Gestion de `.` par l'ICANN
  - Gestion de `.fr.` par l'AFNIC
  - Gestion de `.univ-mlv.fr.` par l'Université Paris-Est Marne-la-Vallée
  - Gestion de `.igm.univ-mlv.fr.` par l'Institut Gaspard-Monge
- Conversion d'un nom en adresse IP par résolution DNS
  - On interroge les serveurs des autorités du plus général au plus spécialisé
  - Serveurs cache permettant une résolution récursive (serveur DNS du FAI)
- Résolution inverse (@IP $\rightarrow$ nom)
  - Interrogation sur le domaine `z.y.x.w.in-addr.arpa` pour une adresse IPv4 w.x.y.z
  - Interrogation sur le domaine `X.ip6.arpa` pour une adresse IPv6 (X est la succession de chiffres hexadécimaux en ordre inverse séparés par des points)



# Enregistrements DNS

Couples de clé/valeur pour une ressource sur un serveur DNS :

- A : adresse IPv4
- AAAA : adresse IPv6
- CNAME : nom canonique (la ressource demandée est un alias)
- MX : serveur de courrier électronique gérant les message entrants
- SRV : serveur gérant un service spécifique du domaine
- NS : serveur de délégation d'une zone DNS
- SOA : information sur l'autorité gérant la zone DNS
- PTR : champ pointeur vers un nom canonique utilisé pour la résolution inverse
- TXT : champs de texte divers

## Remarques

Certains champs peuvent exister en multiples exemplaires (plusieurs adresses IP pour équilibrage de charge, serveurs de mails pour redondance)
Il existe d'autres champs spécialisés utilisés par des extensions de DNS (DNSSEC) ou d'autres protocoles annexes (authentification de emails avec SPF, empreinte SSH avec SSHFP...).

# Résolution de nom

## Exécutables utiles

- `host name` : affiche les adresses IP liées à *name* en utilisant le serveur et cache DNS système
- `dig @dnsserver name` : récupère les entrées DNS de *name* auprès du serveur *dnsserver*

## Appel système POSIX

`getaddrinfo()` (`netdb.h`)



# Adresse IP depuis un nom

- La méthode statique `InetAddress.getByName(String nom)` fonctionne aussi bien pour une @IP textuelle qu'un nom de domaine : elle retourne **une** `InetAddress` résolue (résolution bloquante).
- `InetAddress.getAllByName(String nom)` retourne **toutes** les adresses IP correspondant au nom de domaine donné sous la forme d'un tableau d'`InetAddress` résolues.
- `InetAddress.getByAddress(String nom, byte[] addr)` retourne une `InetAddress` associant le nom et l'adresse spécifiés sans vérification de concordance.

Une exception `UnknownHostException` est levée en cas de problème (nom d'hôte non résolvable).

## Méthode utiles d'`InetAddress`

- `String getHostName()` retourne le nom d'hôte (peut aussi permettre une résolution inverse)
- `String getCanonicalHostName()` permet d'obtenir le nom canonique (CNAME)

# Implantation d'un résolveur DNS en Java

```
public static void main(String[] args) throws UnknownHostException
{
        if (args.length < 1)
                throw new IndexOutOfBoundsException("A domain must be provided");
        String name = args[0];
        InetAddress[] addrs = InetAddress.getAllByName(name);
        for (InetAddress addr: addrs)
                System.out.println(addr.getHostAddress());
}
```



# *java.net.NetworkInterface*

- Existe depuis Java 1.4
- Organisation sous forme d'arbre d'interfaces
- Propriétés des interfaces :
  - interface de boucle locale (*isLooopback()*)
  - interface supportant la multi-diffusion (*supportsMulticast()*)
  - interface virtuelle (*isVirtual()*), e.g. interface de VLAN, de tunnel...
  - interface de liaison point à point (*isPointToPoint()*)

# Obtenir une interface réseau

Identification d'une interface réseau :

- par un nom d'interface : *getName()*, e.g. eth0, wlan0, ppp0...
- par des adresses physiques (adresse MAC Ethernet) : *getInterfaceAddresses()*
- par des adresses logiques (adresse IP) : *getInetAddresses()*

Méthodes statiques de récupération d'interface :

- par nom d'interface : *NetworkInterface.getByName(String)*
- par adresse IP : *NetworkInterface.getByInetAddress(InetAddress)*



# Présentation de UDP (User Datagram Protocol)

- Permet d'envoyer des paquets de données IP indépendants
- La source et la destination d'un paquet sont définis par une adresse IP et un numéro de port (socket)
- Protocole d'implantation simple (latence faible) mais aucune garantie sur la bonne réception du paquet (ainsi que sur le délai de livraison) : pas de contrôle de flux et de congestion
- Vulnérable à l'usurpation d'@IP source
- Supporte le multicast et le broadcast contrairement à TCP
- Utilisé pour les applications nécessitant une latence faible avec une tolérance aux pertes de paquets :
  - Envoi de petits messages informatifs sur un réseau local
  - Streaming audiovisuel
  - Voix sur IP
  - Jeux en réseau
  - ...

# Format du datagramme UDP

- Port source sur 16 bits
- Port destination sur 16 bits
- Longueur sur 16 bits (soit 65535 octets au maximum)
- Somme de contrôle sur 16 bits (calculée sur l'en-tête comprenant la fin de l'en-tête IP et les données)
- Champ de données



# Sockets

- Les piles IP implantent un mécanisme de socket (association d'adresse IP et de port de communication ouvert sur une machine)
- Réalisation d'une connexion en envoyant un paquet d'une socket vers une autre

## Sur les systèmes POSIX

- `int socket(int domain, int type, int protocol)` pour ouvrir une socket
- `int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen)` pour lier une socket à une adresse et un port
- `int send(int socket, const void *buf, size_t len, int flags)` pour envoyer un datagramme
- `ssize_t recv(int s, void *buf, size_t len, int flags)` pour extraire un datagramme de la file d'attente de réception
- `man 7 socket, ...` pour en savoir plus

## Les interfaces réseau

- Pour chaque interface réseau active d'une machine : au moins une @IP
- Une socket doit se fixer sur une ou plusieurs interfaces réseau (pour s'intéresser uniquement aux données provenant de cette interface) :
  - Choix d'une interface réseau par son @IP (e.g. 127.0.0.1 pour lo en IPv4)
  - Choix de toutes les interfaces réseau avec l'adresse joker (adresse nulle)



## Socket réseau UDP : classe DatagramSocket

Association d'une adresse et d'un port pour recevoir ou envoyer des datagrammes UDP
Constructeurs disponibles :

- `DatagramSocket()` : création d'une socket sur l'adresse joker sur un des ports disponibles
- `DatagramSocket(int port)` : permet de choisir le port d'attache, utilisation de l'adresse joker
- `DatagramSocket(int port, InetAddress addr)` : permet de s'attacher sur une adresse spécifique de la machine (n'écoute pas sur toutes les interfaces)

Ces constructeurs peuvent lever une `SocketException` si la socket ne peut être allouée.
Après utilisation, une socket doit être fermée avec `close()` pour libérer les ressources systèmes associées.

# Datagramme UDP : classe DatagramPacket

Cette classe permet de manipuler des datagrammes UDP.

- Ils peuvent être créés par des constructeurs utilisant un tableau d'octets (`byte[]`) soit pour recevoir des données, soit pour en envoyer.
- Pour recevoir des données, le tableau d'octets communiqué doit être de taille suffisante pour recevoir les données (sinon le paquet est tronqué !).
- Une instance peut être utilisée pour envoyer plusieurs paquets ou pour en recevoir. Possibilité de modifier :
  - L'adresse de destination avec `setAddress(InetAddress)` ainsi que le port avec `setPort(int)`.
  - Le tampon de données avec `setData(byte[] buffer, [int offset, int length])`.
- Les données peuvent être consultées avec `byte[] getData()` et `int getLength()` (taille du paquet reçu).



# Caractères et octets

- Les données sont échangées sur le réseau par séquence d'octets
- Pour transmettre une chaîne de caractères de A vers B :
  1. Codage de la chaîne en octets par A
  2. Transmission d'un paquet avec le tableau d'octets
  3. Réception du paquet avec tableau d'octets par B
  4. Décodage des octets en chaîne par B

## Des caractères vers les octets (et vice-versa)

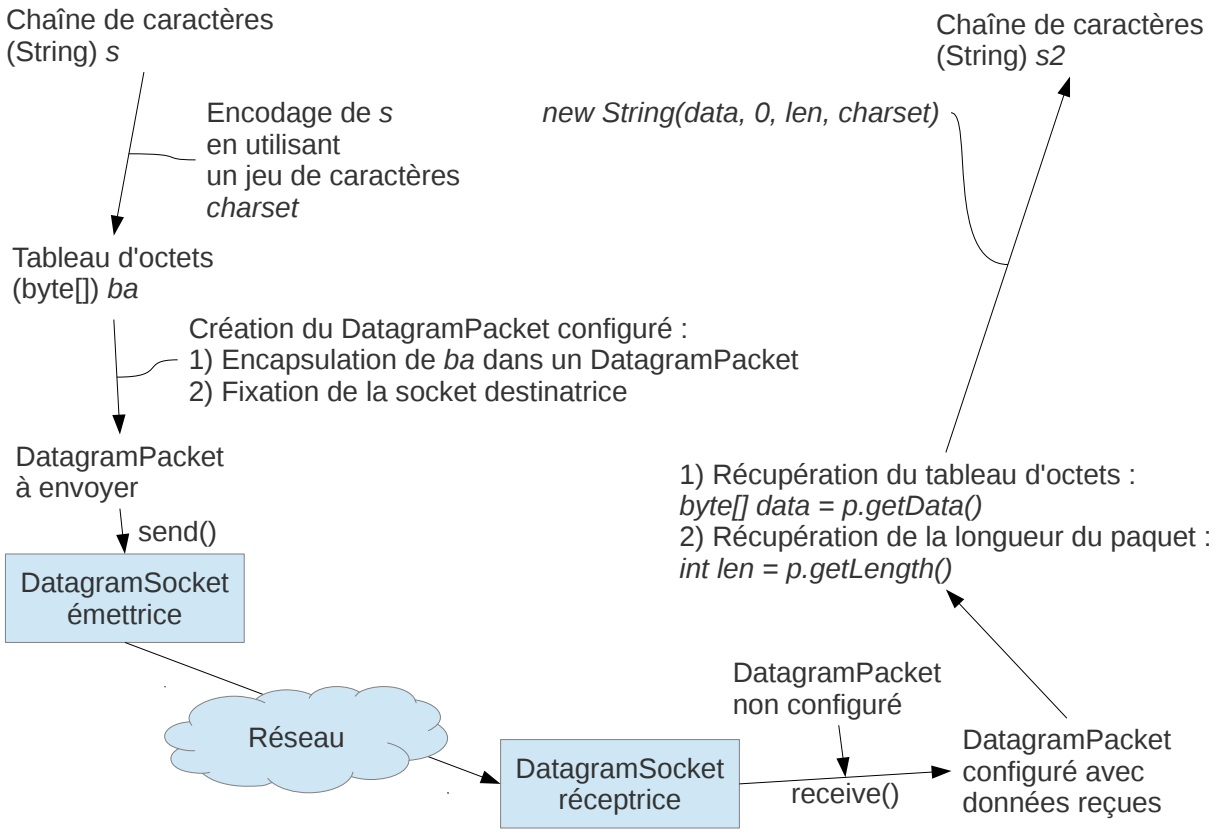
- Convention de conversion caractère-octet implantée par classe *Charset*
- Quelques jeux de caractères : ASCII, iso-8859-15, UTF-8, UTF-16LE, UTF-16BE, ...
- Un caractère peut être codé par un ou plusieurs octets
- De la chaîne aux octets : `byte[] String.toBytes(String charset)`
- Des octets à la chaîne : `new String(byte[] tableauOctets, int depart, int longueur, String charset)`
- Si le jeu de caractère n'est pas indiqué : utilisation du jeu par défaut (`static Charset Charset.defaultCharset()`)
- Pour obtenir tous les jeux disponibles : `static SortedMap<String,Charset> Charset.availableCharsets()`



## Envoyer et recevoir des paquets avec DatagramSocket

- Méthode `send(DatagramPacket)` pour envoyer une paquet : paquet ajouté dans la file d'attente d'émission
- Méthode `receive(DatagramPacket)` pour recevoir un paquet en file d'attente de réception
  - Appel bloquant jusqu'à la réception d'un paquet
  - `setSoTimeout(int timeoutInMillis)` fixe un délai limite de réception de paquet (au-delà, levée d'une `java.net.SocketTimeoutException`)
- Ces méthodes peuvent lever une `java.io.IOException` qui doit être gérée.
- La taille des files d'attente est consultable et configurable avec :
  - `int get{Send,Receive}BufferSize()`
  - et `void set{Send,Receive}BufferSize()`.

# Envoyer des paquets UDP contenant du texte






# Exemple Java : envoyer un paquet contenant l'heure à une plage d'adresses

```
import java.util.*; import java.io.*; import java.net.*;

public class TimeSender
{
        private final DatagramSocket socket;
        private final DatagramPacket packet;

        public TimeSender(int port) throws SocketException
        {
                this.socket = new DatagramSocket();
                this.packet = new DatagramPacket(new byte[0], 0); // Packet with empty array
                this.packet.setPort(port);
        }

        public void sendTimePacket(InetAddress a) throws IOException
        {
                // We reuse the same datagram packet but modify its content and addressee
                byte[] timeData = new Date().toString().getBytes();
                packet.setData(timeData);
                packet.setAddress(a);
                // this.packet.setPort is useless since the port does not change
                socket.send(packet); // May throw an IOException
        }

        public static InetAddress getSuccessor(InetAddress a)
        {
                // Exercise: write the implementation of getSuccessor(InetAddress) returning the successor of an IP address.
                // Example: the successor of 10.1.2.3 is 10.1.2.4 (for last byte 0 and 255 are forbidden)
        }

        public static void main(String[] args) throws IOException
        {
                if (args.length < 3) throw new IllegalArgumentException("Not enough arguments");
                InetAddress firstA = InetAddress.getByName(args[0]); // Start address
                InetAddress lastA = InetAddress.getByName(args[1]); // Stop address
                int port = Integer.parseInt(args[2]);
                TimeSender ts = new TimeSender(port);
                for (InetAddress currentA = firstA; ! currentA.equals(lastA); currentA = getSuccessor(currentA))
                        ts.sendTimePacket(currentA);

        }
}
```

## Quelques conseils pour l'échange de paquets

- Recycler des *DatagramPacket* déjà créés (en changeant les données hébergées)
- Ne pas créer des *DatagramSocket* inutiles, les libérer lorsqu'on n'en a plus besoin
- Toujours vérifier que la taille de réception est suffisante $\longrightarrow$ erreur classique : envoyer un paquet de $k$ octets puis recevoir sur ce même paquet sans changer de tableau ni de taille ; on est alors limité à $k$ octets pour la réception
- Ne pas oublier d'utiliser la taille des données reçues : la fin du tableau n'est pas exploitable
- Vérifier l'adresse IP/port émetteur d'un paquet reçu (même si cela n'apporte qu'une sécurité faible)



## Mécanisme de pseudo-connexion

- Possibilité de pseudo-connecter deux sockets UDP avec `void DatagramSocket.connect(InetAddress addr, int port)` : filtrage automatique des paquets émis et reçus
- Vérification de la connexion avec `boolean isConnected()`
- Déconnexion avec `void disconnect()`

# Quelques tests avec Netcat

Netcat : application simple pour communiquer en UDP (et aussi en TCP)

1. On lance le serveur sur le port N en écoute en UDP sur toutes les interfaces (adresse joker) : `nc -l -u N`
2. On exécute un client se connectant sur le port N de la machine : `nc -u nomMachine N`
3. Le client envoie chaque ligne de l'entrée standard dans un datagramme UDP : le serveur la reçoit et l'affiche sur sa sortie standard
4. Envoi de datagrammes du serveur vers le client également possible

Limitation pour le serveur netcat : communication avec un seul client



# Exemple Java : recevoir des datagrammes sur un port avec un délai limite

```
public class PacketReceiver
{
  public static final int BUFFER_LENGTH = 1024;

  public static void main(String[] args) throws IOException
  {
    if (args.length < 3) throw new IllegalArgumentException("Not enough arguments");
    InetAddress a = InetAddress.getByName(args[0]);
    int port = Integer.parseInt(args[1]);
    int timeout = Integer.parseInt(args[2]);
    DatagramSocket s = new DatagramSocket(port, a);
    try {
      s.setSoTimeout(timeout);
      DatagramPacket p = new DatagramPacket(new byte[BUFFER_LENGTH], BUFFER_LENGTH);
      for (boolean go = true; go; )
      {
        try {
          s.receive(p);
        } catch (SocketTimeoutException e)
        {
          System.err.println("Sorry, no packet received before the timeout of " + timeout + " ms");
          go = false;
        }
        if (go)
        {
          System.out.println("Has received data from " + p.getAddress() + ":" + p.getPort());
          System.out.println(new String(p.getData(), 0, p.getLength(), "UTF-8"));
          p.setLength(BUFFER_LENGTH); // Reset the length to the size of the array
        }
      }
    } finally
    {
      s.close(); // Don't forget to liberate the socket resource
    }
  }
}
```

# Multicast UDP

- Quelques rappels sur les adresses de multicast :
  - 224.0.0.0/4 en IPv4 (préfixe 1110)
  - FF00::0/8 en IPv6 (RFC 4291)
  - Certaines adresses à usage prédéfinis (groupe des hôtes, routeurs, uPnP...)

## Principe

- Les hôtes s'abonnent/se désabonnent à une adresse de multicast A.
- Une trame émise vers A est adressée à tous les membres du groupe.

## Avantages

- En théorie, mutualise les paquets échangés sur des segments de réseau contrairement à l'unicast.
- Évite d'envoyer des paquets à tous les hôtes comme en broadcast (broadcast incompatible avec un réseau étendu).



# Protocoles de multicast

- IGMP (Internet Group Management Protocol) permet de signaler à la passerelle du réseau local les membres de chaque groupe. Certains commutateurs de couche 2 l'utilisent.
- Sur Internet, un arbre de distribution multicast est construit avec PIM (Protocol Independent Multicast) en *Sparse Mode*, *Dense Mode*, mixte *Sparse-Dense* ou *Source Specific Mode*.

## java.net.MulticastSocket

- Version améliorée de `DatagramSocket` avec support du multicast
- Envoi classique d'un datagramme avec `send(DatagramPacket)` (à une IP unicast ou multicast)
- Réception de datagrammes en multicast requierant un abonnement préalable au groupe de multicast
  - `void joinGroup(InetAddress ia)` pour joindre un groupe
  - `void leaveGroup(InetAddress ia)` pour quitter un groupe
- Quelques paramètres utiles pour le multicast
  - `void setInterface(InetAddress ia)` ou `void setNetworkInterface(NetworkInterface netIf)` pour fixer l'interface utilisée pour la communication multicast
  - `void setTimeToLive(int ttl)` pour indiquer la portée des paquets multicast envoyés



## TCP (Transport Control Protocol)

Contrairement à UDP, TCP (RFC 793) présente les caractéristiques suivantes :

- Transmission unicast bidirectionnelle en mode connecté (flux)
- Mécanisme de retransmission des paquets non acquittés par le destinataire

Conséquences :

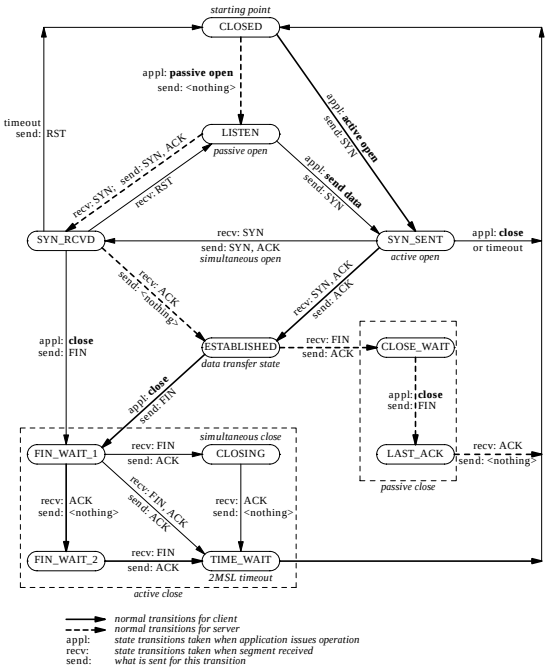
- Implantation plus lourde avec gestion d'états des correspondants
- Peu adapté au temps réel (décalage de réception dû à des paquets perdus)
- Peu performant sur des réseaux radio (réduction de débit en cas de problèmes transitoires)

# A mi-chemin entre UDP et TCP : SCTP (Stream Control Transmission Protocol)

- Transmission de messages (comme UDP) mais avec conservation d'ordre
- Fiabilité de l'acheminement avec gestion de congestion (comme TCP)
- Gestion de multi-flux
- Possibilités de multi-diffusion (comme UDP)
- Implanté en standard avec noyaux Linux et BSD récents
- Implantation Java intégrée depuis le JDK 1.7 : paquetage `com.sun.nio.sctp`



# Fonctionnement de TCP : les états (TCP/IP Illustrated, Stevens & Wright)



## Fonctionnement de TCP : l'envoi et l'acquittement de paquets

- Une fenêtre d'envoi de $n$ segments est utilisée : le correspondant doit envoyer au moins un acquittement tous les $n$ segments reçus
- Acquittement envoyé pour le segment $i$ : tous les segments jusqu'à $i$ ont été reçus
- Compte-à-rebours pour chaque segment envoyé : à l'expiration, le paquet est renvoyé
- Si trop de paquets perdus : congestion probable et réduction de la fenêtre d'envoi
- Problème : un paquet perdu n'est pas obligatoirement lié à la congestion de liens (réseaux sans fils)



## Format de segment TCP

- Port source (16 bits)
- Port destination (16 bits)
- Numéro de séquence (32 bits)
- Numéro d'acquittement (32 bits) du dernier segment reçu
- Taille de l'en-tête en mots de 32 bits (32 bits)
- Drapeaux divers (28 bits) : ECN (congestion), URG (données urgentes), ACK (accusé de réception), PSH (push), RST (reset), SYN (établissement de connexion), FIN (terminaison de la connexion)
- Taille de la fenêtre (16 bits) souhaitée en octets (au-delà un acquittement doit être envoyé)
- Somme de contrôle (16 bits), sur la fin de l'en-tête IP, l'en-tête TCP et les données
- Pointeur de données urgentes (16 bits)
- Options (complétées avec un bourrage pour une longueur d'en-tête multiple de 32 bits)

## Initialisation d'une socket client

En Java, socket TCP représentée par une instance de `java.net.Socket`
Constructeurs et méthodes peuvent lever une `IOException`

1. On souhaite écouter sur l'adresse joker et laisser le système choisir un port libre d'attache :
   - `Socket(InetAddress addr, int port)` : se connecte sur la socket distante *addr:port*
   - `Socket(String host, int port)` : équivaut à `Socket(InetAddress.getByName(host), port)`
2. On souhaite spécifier l'adresse de l'interface et le port local d'attache :
   - `Socket(InetAddress addr, int port, InetAddress localAddr, int localPort)`
3. On souhaite ne pas connecter la socket pour l'instant :
   1. `Socket()`
   2. ensuite, il faut attacher la socket localement avec `bind(SocketAddress sa)`,
   3. et puis se connecter (avec un temps limite si timeout $\neq 0$) à la socket distante avec `connect(SocketAddress remote, int timeoutInMillis)`



## Flots de Socket

Pour communiquer entres sockets TCP (locale et distante), on utilise des flots binaires.

- `InputStream getInputStream()` : flot où lire des données binaires venant du correspondant
- `OutputStream getOutputStream()` : flot où écrire des données à destination du correspondant

Quelques remarques (et rappels) :

- Les opérations d'E/S sur ces flots sont bloquantes.
- Les `IOException` doivent être gérées.
- Les écritures ne provoquent pas nécessairement l'envoi immédiat de paquet TCP (bufferisation possible)
  - Appeler `flush()` permet de demander l'envoi immédiat (utiliser *TCP No delay*))
- Fermer un flot ferme la socket sous-jacente

# Configuration de Socket

- `void setKeepAlive(boolean keepAlive)` : envoie régulièrement des paquets pour maintenir en vie la connexion
- `void setOOBInline(boolean oobinline)` : intègre les données urgentes Out-Of-Band dans le flux entrant (sinon elles sont ignorées)
  - `void sendUrgentData(int data)` : permet d'envoyer un octet urgent (8 bits de poids faible)
- `void setReceiveBufferSize(int size)` : fixe la taille du buffer en réception
- `void setSendBufferSize(int size)` : fixe la taille du buffer en émission
- `void setSoLinger(boolean on, int lingerTimeInSeconds)` : configure une fermeture bloquant jusqu'à confirmation de réception des dernières données envoyées ou timeout atteint
- `void setSoTimeout(int timeoutInMillis)` : configure un délai limite pour les lectures (`SocketTimeoutException` peut être levée)
- `void setTcpNoDelay(boolean on)` : permet de désactiver l'algorithme de Nagle (utile pour les protocoles interactifs : envoi immédiat de petits segments)
- `void setTrafficClass(int tc)` : configure l'octet Type-Of-Service pour les paquets (utile pour la QoS)
- `void setReuseAddress(boolean on)` : permet de réutiliser une adresse de socket locale récemment fermée (état TIME_WAIT) ; doit être appelé avant `bind`.

Remarque : existence de getters relatifs à ces méthodes (pour obtenir la configuration courante)



# Fermeture de Socket

- `shutdownInput()` : permet d'ignorer toutes les données entrant ultérieurement sur la socket $\longrightarrow$ la socket distante n'est pas informée et peut continuer d'envoyer des données
- `shutdownOutput()` : ferme le flot sortant de la socket
- `close()` : permet de libérer les ressources liées à la socket

# Exemple : un client TCP qui envoie un fichier

```
public static void main(String[] args)
{
        if (args.length < 2) throw new IllegalArgumentException("Not enough arguments");
        try ( Socket s = new Socket(args[0], Integer.parseInt(args[1]));
                InputStream is = new FileInputStream(args[2]);
                OutputStream os = new BufferedOutputStream(s.getOutputStream()) )
        {
                s.shutdownInput(); // We are not interested in the answer of the server
                transfer(is, os);
                os.flush();
        }
}
```



# Principe de fonctionnement d'une socket serveur

1. La socket serveur écoute les paquets entrants
2. À la réception d'un paquet SYN :
   - retour d'un paquet SYN/ACK à la socket client pour continuer le *3-way handshake*
   - retour d'un paquet RST pour refuser la connexion (si trop de connexions pendantes par exemple)
3. Le socket client confirme l'ouverture par un paquet ACK à la socket serveur
4. La socket serveur créé une socket de service pour communiquer avec la socket client et l'ajoute dans une file d'attente pour être récupérée par le programme serveur (par *accept*)

# java.net.ServerSocket

## Construction

- `ServerSocket(int port, int backlog, InetAddress addr)` : créé une socket serveur TCP écoutant sur l'IP locale spécifiée, le port indiqué avec le backlog (nombre maximal de connexions en file d'attente) donné (si addr non indiqué utilisation de l'adresse joker, si backlog non indiqué utilisation d'une valeur par défaut)
- `ServerSocket()` : créé une socket serveur localement non attachée
  - appel ultérieur de `bind(SocketAddress sAddr, int backlog)` nécessaire

## Acceptation de connexion

`Socket accept()` retourne la plus ancienne socket de service en attente.
Cette méthode est bloquante.
Timeout instaurable avec `setSoTimeout(int timeoutInMillis)`.



# Modèle itératif d'utilisation de ServerSocket

1. On construit une `ServerSocket`
2. On récupère une socket connectée avec le prochain client en attente avec `accept()`
3. On dialogue en utilisant la socket connectée au client
4. La communication terminée avec le client, on ferme la socket
5. On retourne à l'étape 2 pour récupérer le client suivant

Limitation du modèle : 1 seul client traité à la fois (sinon plusieurs threads nécessaires)

# Utilisation d'une socket de service

- Il s'agit d'une instance de `java.net.Socket` : même utilisation que côté client
  - `getInputStream()` pour obtenir un flot pour recevoir des données du client
  - `getOutputStream()` pour obtenir un flot pour envoyer des données au client
  - Utilisation possible des setters de configuration
- Comment connaître les coordonnées du client ?
  - `getInetAddress()` pour obtenir l'adresse IP
  - `getPort()` pour obtenir le port



# Exemple : un serveur TCP qui réceptionne des fichiers

```
public class FileReceiverServer
{
        // After one minute with no new-connection, the server stops
        public static final int ACCEPT_TIMEOUT = 60000;

        private final File directory;
        private final int port;

        public FileReceiverServer(File directory, int port)
        {
                this.directory = directory;
                this.port = port;
        }

        public void launch() throws IOException
        {
                try (ServerSocket ss = new ServerSocket(port))
                {
                        ss.setSoTimeout(ACCEPT_TIMEOUT);
                        while (true)
                        {
                                try (Socket s = ss.accept();
                                        OutputStream os = new BufferedOutputStream(new FileOutputStream(
                                                new File(directory, s.getAddress() + "_" + s.getPort())) )
                                {
                                        // Save the data in a file named with the address and port of the client
                                        transfer(s.getInputStream(), os);
                                } catch (SocketTimeoutException e)
                                {
                                        System.err.println("The server will shutdown because no new connections are available");
                                        break;
                                } catch (IOException e)
                                {
                                        e.printStackTrace();
                                        continue; // Treat the next incoming client
                                }
                        }
                }
        }

        public static void main(String[] args)
        {
                new FileReceiverServer(
                        new File(args[0]), Integer.parseInt(args[1])).launch();
        }
}
```