Programmation répartie avec les sockets

Patrice Torguet
IRIT
Université Paul Sabatier

Plan

- Problèmes liés au réseau
- Protocoles applicatifs
- Sockets BSD
- Protocoles de Transport
- Programmation en Java
- Conclusion

Caractéristiques des réseaux

- Latence
- Débit
- Fiabilité

Latence

- Durée nécessaire pour envoyer un bit de donnée d'un endroit à un autre dans le réseau (latency, lag, ping …)
- La latence diminue l'interactivité d'une application distribuée
- Raisons :
  - Délai du à la vitesse de la lumière (8.25 ms par fuseau horaire)
  - Délai ajouté par les ordinateurs (traitements)
  - Délai ajouté par le réseau (gestion des erreurs, de la congestion, commutation, traduction données en signal…)
- Autre notion : la gigue (variation de la latence)

Débit

- Nombre de bits qui peuvent être acheminés par le réseau en 1 seconde (throughput, bande passante, bandwidth)
- Dépends des types de câbles (ou du support de propagation) et des équipements réseaux

Fiabilité

- Un réseau peut perdre/détruire des données (congestion)
  - Retransmissions possibles par Transport
- Les données peuvent être modifiées (erreurs de transmission)
  - Retransmissions ou destruction (cf. plus haut)
- Si on veut de la fiabilité il faut gérer des acquittements (soit au niveau Transport soit au dessus - application en général)

Protocoles applicatifs

- Règles utilisées par deux applications pour communiquer
- Au niveau application :
  - Format des messages
  - Sémantique d'envoi/réception
  - Conduite à tenir en cas d'erreur
- cf. RFC proto app : HTTP, FTP, SMTP, POP...

Format des messages

- Indique ce que contiennent les messages échangés
- Côté émission : que doit on y mettre
- Côté réception : comment doit-on décortiquer un message reçu

Types de messages

- En général un protocole gère plusieurs types de messages
- Contrôle :
  - Connexion / Déconnexion / Acquittement / Synchronisation ...
- Information :
  - Ordres / Données / Codes d'erreurs...

Sémantique d'envoi/réception

- L'émetteur et le récepteur doivent se mettre d'accord sur ce que le récepteur peut déduire quand il reçoit un certain message
- Quelles actions le récepteur doit exécuter lors de la réception d'un certain message

Conduite à tenir en cas d'erreur

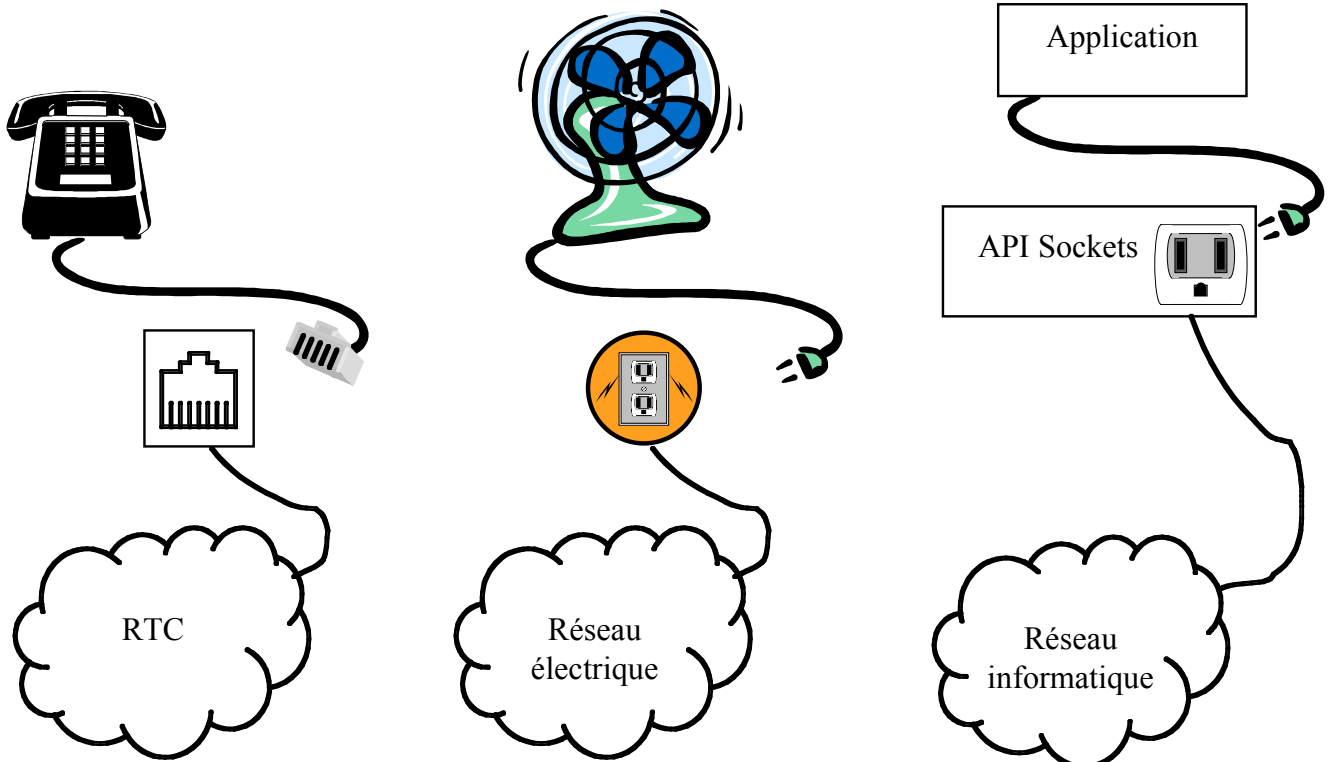
- Ensemble de scénarios d'erreur et ce que chaque application doit faire pour chaque cas

Les sockets BSD

- BSD (Berkeley Software Distribution) est un système d'exploitation de type UNIX développé par l'université de Berkeley depuis 1977
- Parmi les avancées introduites par ce système d'exploitation on trouve les sockets (version 4.3 de BSD - 1983)
- Le concept a été depuis repris par tous les systèmes d'exploitations

Sockets BSD

- Une socket est une abstraction correspondant à une extrémité d'un canal de communication
- Le nom socket (prise) vient d'une analogie avec les prises électriques et téléphoniques



Sockets BSD

- Les sockets sont aussi une API (interface de programmation) pour :
  - manipuler des informations liées à la communication (adresses source et destination)
  - créer un canal de communication si nécessaire
  - envoyer et recevoir des messages protocolaires applicatifs
  - contrôler et paramétrer la communication

Sockets BSD

- Les sockets BSD permettent des communications
  - internes à un système d'exploitation UNIX (domaine AF_UNIX)
  - à distance
    - en utilisant TCP/IP (domaine AF_INET)
    - ou d'autres piles de protocoles (ATM notamment)

Sockets BSD

- L'abstraction peut être
  - orientée réseau
    - pour TCP/IP par exemple une socket correspond à un quintuplet : @IP locale, port local, @IP distante, port distant, protocole de transport (TCP ou UDP)
  - orienté programmation
    - une socket est assimilée à un descripteur de fichier (comme les FIFO - pipes)

Protocoles de Transport

- Dans le domaine AF_INET on peut communiquer
  - en mode connecté (STREAM)
    - on utilise des connections TCP
  - en mode non connecté (DGRAM)
    - on utilise des datagrammes UDP indépendants
    - point à point ou multipoint (broadcast / multicast)
  - en mode “raw” (pour accéder aux couches basses - ICMP par exemple)

Protocoles de Transport

- Notion de port
  - Sur un même ordinateur plusieurs applications peuvent communiquer en même temps
  - Problème : comment savoir à quelle application on veut parler
  - Solution : chaque application est identifiée par un numéro unique (pour une machine donnée et pour un protocole donné) appelé numéro de port (entier sur 16 bits - 65535 ports différents - 0 n'est pas utilisé)

Protocoles de Transport

- Différents types de ports
  - ports systèmes ou bien connus (1-1023) - réservés au système d'exploitation (exemple 80 - serveur web)
  - ports utilisateurs ou enregistrés (1024- 49152) - ne doivent être utilisé que par des applications spécifiques (comme les premiers) enregistrées auprès de l'organisme IANA (Internet Assigned Numbers Authority - www.iana.org) (exemple 26000 - Quake)
  - ports privés ou dynamiques (les autres) - utilisés par les applications non encore enregistrées et par les clients TCP

IP : Internet Protocol

- Rappels :
  - gère l'adressage (@IP) et le routage dans Internet
  - gère la fragmentation pour satisfaire la taille max des trames des réseaux traversés : MTU
    - Attention : augmente la probabilité de perte (lors de la reconstruction si un seul fragment est perdu, le datagramme est détruit)
  - gère aussi le TTL : évite qu'un datagramme reste indéfiniment dans le réseau (en général ≈ nombre max de routeurs que le datagramme peut traverser)

Sockets STREAM / TCP

- Rappels sur TCP :
- offre un flot bidirectionnel d'octets entre 2 processus
  - Fiable (ni perte, ni duplication) et ordonné
  - "Connexion virtuelle" entre les 2 locuteurs (on peut donc détecter les ruptures de connexion)
  - Le plus utilisé aujourd'hui (mail, web, ftp...)
- 3 phases à programmer : connexion, dialogue, déconnexion

Sockets DGRAM / UDP

- Rappels sur UDP:
- Les données sont transférées dans des datagrammes UDP échangés indépendamment les uns des autres
  - Non fiable (pertes et duplications possibles) et non ordonné : best effort
  - Utilisé par les applications multimédia (audio, vidéo, jeux)
- Envoi/réception de messages en utilisant un socket

Sockets DGRAM / UDP

- Avantages
- Protocole plus simple (pas de gestion des connexions virtuelles, pas de gestion de la fiabilité) et donc moins coûteux pour chaque machine
- Protocole un tout petit peu plus rapide (pas de gestion de l'ordre et d'évitement de la congestion) : les messages sont envoyés directement (pas besoin d'attendre lorsque la fenêtre de réception est pleine) et donnés à l'application dès qu'ils sont reçus (pas de remise en ordre)

Sockets DGRAM / UDP

- Inconvénients
- Problèmes de fiabilité (perte de messages essentiellement dus à la congestion) : si on veut de la fiabilité, il faut la gérer au niveau application.

Sockets DGRAM / UDP

- Diffusion totale et restreinte
- Pour éviter d'envoyer le même message vers plusieurs destinations on peut utiliser la diffusion avec UDP
- Diffusion totale : vers toutes les machines d'un réseau local (@ IP de broadcast : id broadcast spécifique au réseau ou 255.255.255.255 IPv4)
  - Utile pour les applications restreintes à un réseau local mais pas généralisable sur Internet
  - Nécessite des traitements par toutes les machines du réseau local : coûteux

Sockets DGRAM / UDP

- Diffusion restreinte : vers un ensemble de machine qui peut varier dynamiquement
  - Les @IPv4 multicast : de 224.0.0.0 à 239.255.255.255
    - 224.* sont réservées
    - 239.* sont réservées ou ne sont pas routées à l'extérieur d'un réseau local
    - 225.0.0.0 à 238.255.255.255 pour une utilisation sur Internet
  - En IPv6 toutes les adresses commençant par FF.
    - ffXe::/16 sont utilisables sur Internet. Quelque soit X.
    - ffX5::/16 sont restreintes au réseau local. (Pour les tests en local).

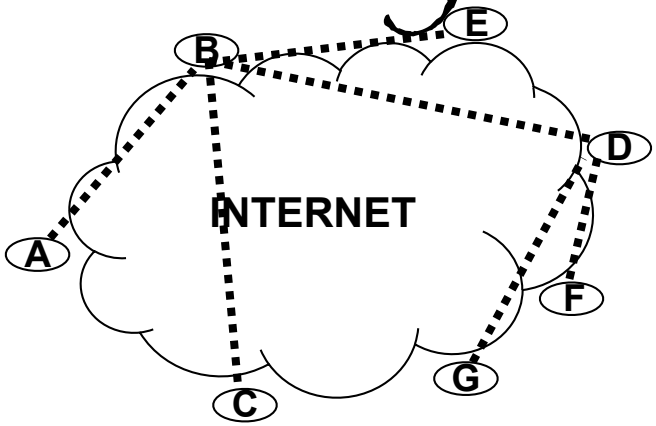
Sockets DGRAM / UDP

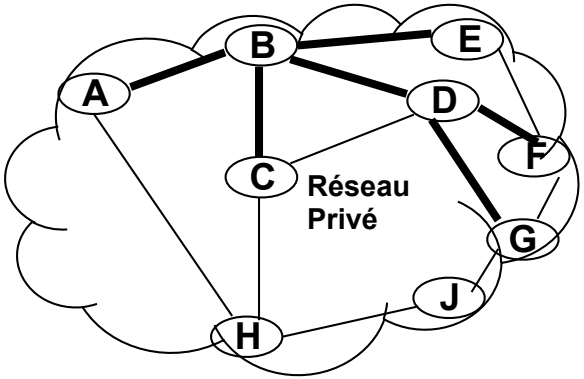
- Diffusion restreinte : principe
  - Une application sur une machine s'inscrit à une @ IP multicast
  - Les routeurs propagent les inscriptions et créent un arbre de diffusion (protocole IGMP)
  - Quand une application envoie un message vers une @ IP multicast, l'arbre de diffusion est utilisé et toutes les machines reçoivent le message (dans la limite du TTL utilisé)

Sockets DGRAM / UDP

- Diffusion restreinte : limitations
  - La plupart des routeurs le gèrent mais la plupart des administrateurs réseau le limitent au réseau local
  - Les FAI le réservent à certaines applications (ex. télé sur internet)
  - Solution : création de tunnels entre les réseaux locaux (le tunnel encapsule les datagrammes à destination d'@ IP mcast dans des datagrammes normaux voire même dans des segments TCP)
    - Overlay multicast et application level multicast

Overlay multicast





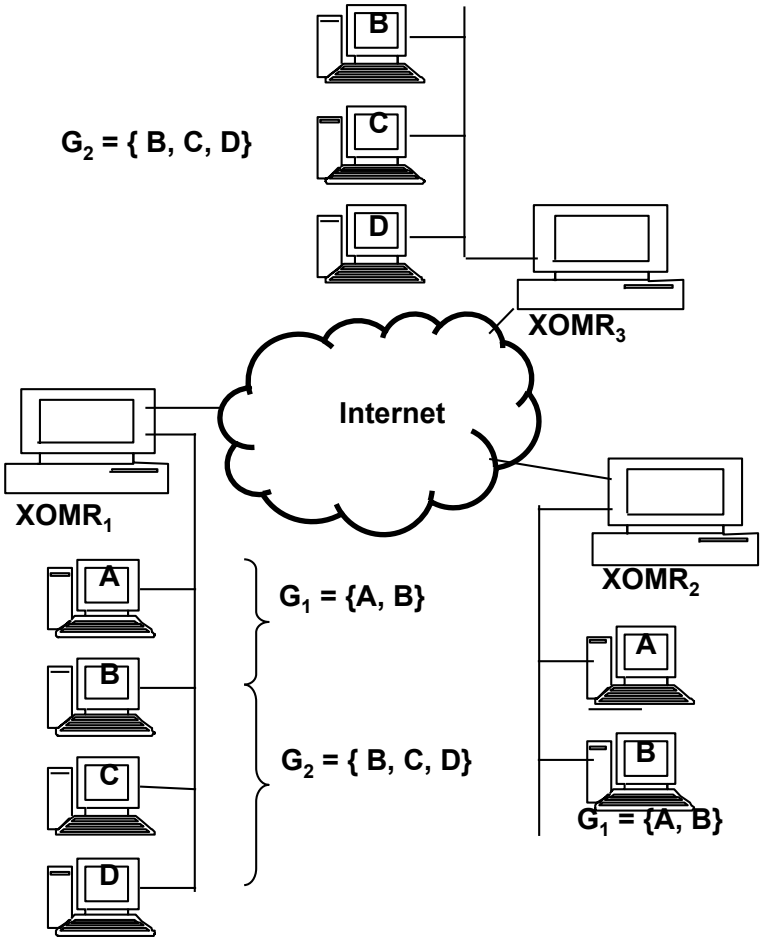
"Overlay" Multicast :

- Plusieurs vers plusieurs
  - Plusieurs émetteurs pour un même groupe
  - Arbres basés sur les sources
- Réseau Ouvert
  (indépendant des domaines de gestion)
- Peut être adapté à une application
  - Considérations de QoS de bout en bout
- Gestion des groupes efficaces

IP Multicast Standard :

- En général : 1 vers plusieurs
  - Un seul émetteur
  - Arbre basé sur une racine
- Réseau Privé
  (ne fonctionne que sur un seul domaine de gestion)
- Indépendant de l'application

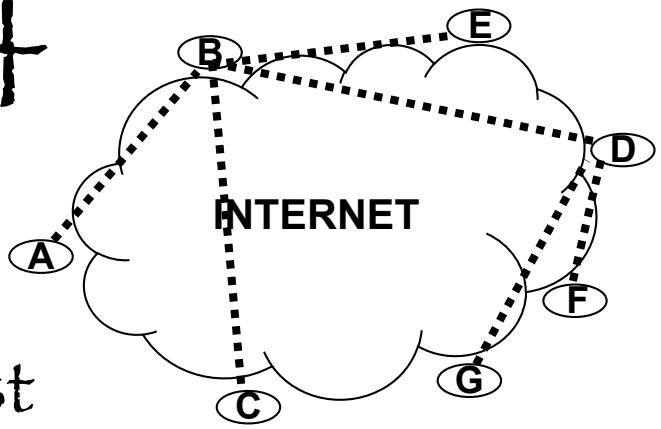
Overlay multicast : exemple XOM



Communications point à point (UDP ou TCP) entre les routeurs d'overlay (XOMR)

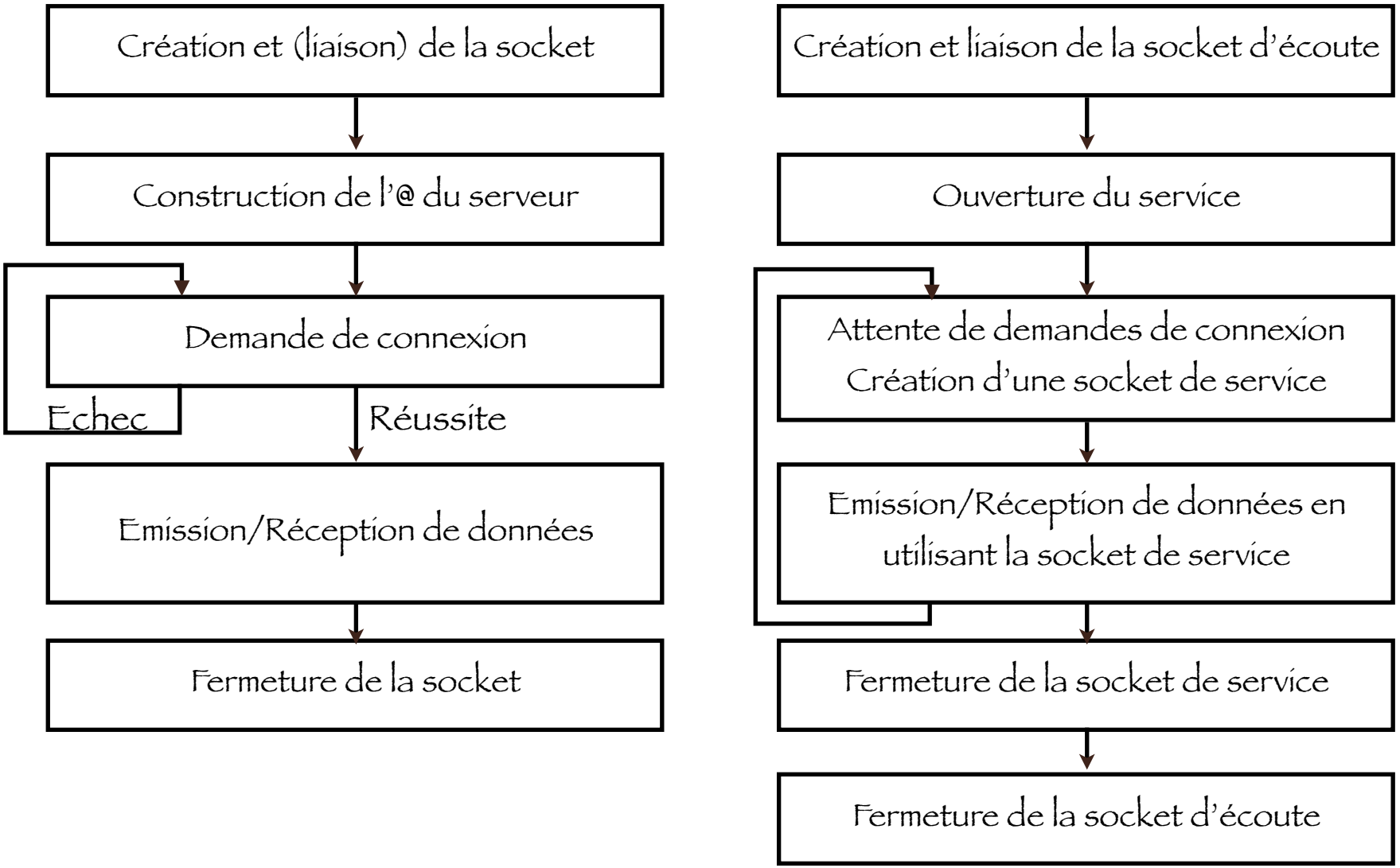
Multicast standard sur les LANs

Multicast applicatif

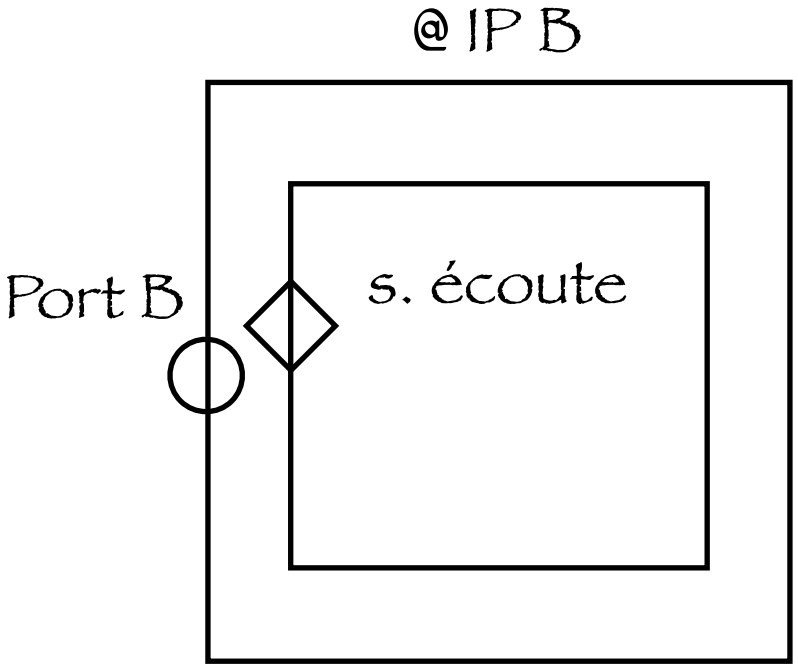


- Même principe que l'overlay mcast
- Mais l'overlay est géré par l'application (pas de "routeurs" d'overlay : A,B...G sont les applis).
- Avantage : on fait exactement ce que l'on veut (le routage peut être intelligent)
- Inconvénient : augmente la complexité de l'application

TCP et le Modèle client/serveur



Création sock écoute + bind



s. écoute

@ loc = @ IP B ou Any

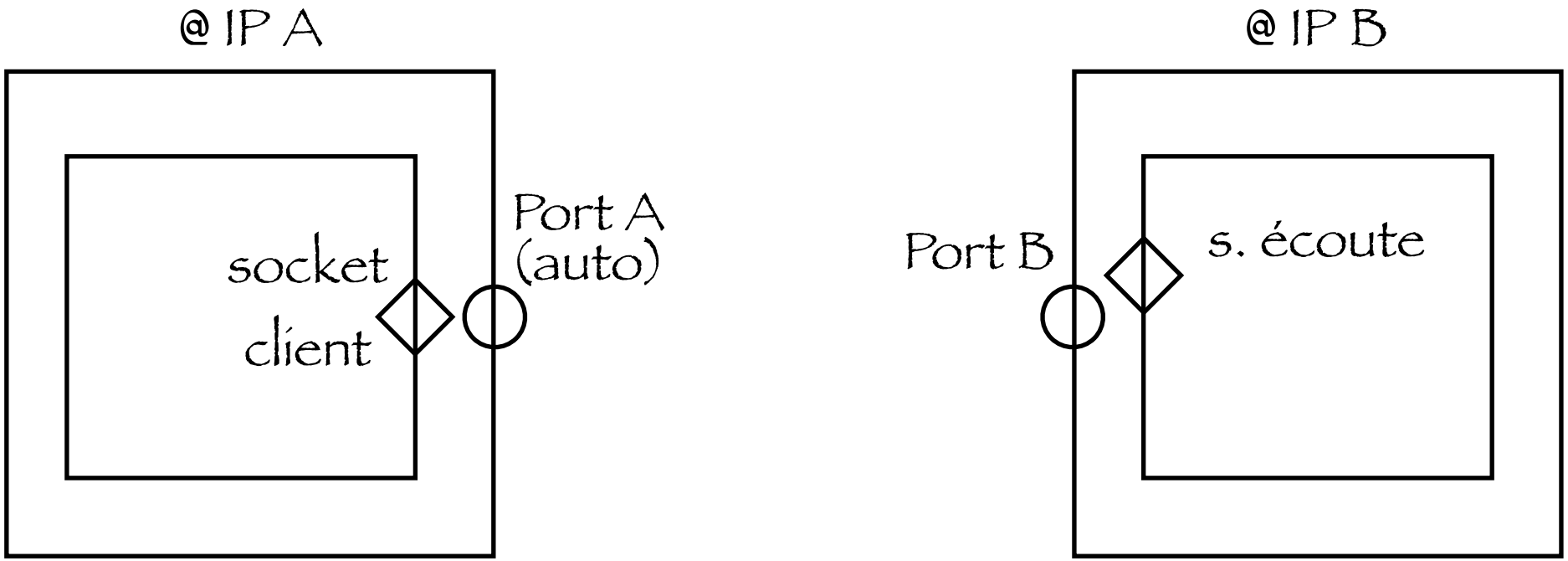
port loc = port B

@ dist = Any

port dist = 0

proto = TCP

Création sock client + bind



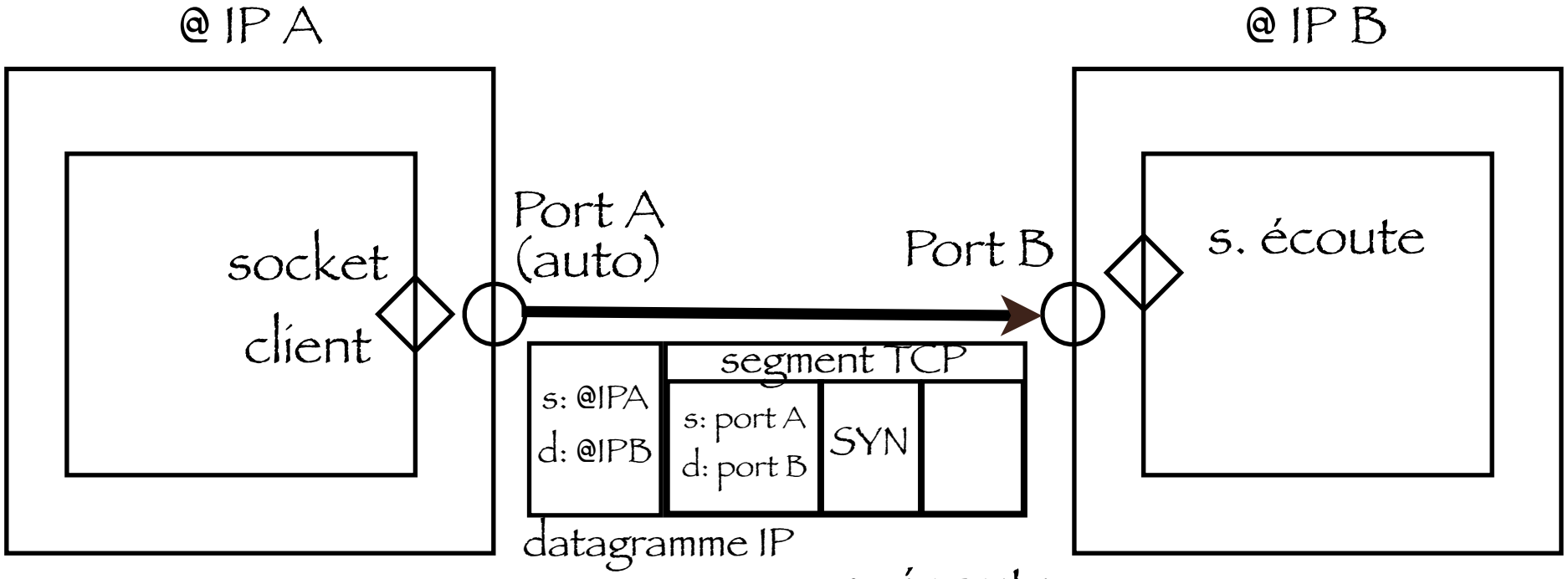
socket client

@ loc = @ IP A
port loc = port A
@ dist = Any
port dist = 0
proto = TCP

s. écoute

@ loc = @ IP B ou Any
port loc = port B
@ dist = Any
port dist = 0
proto = TCP

Demande de Connexion



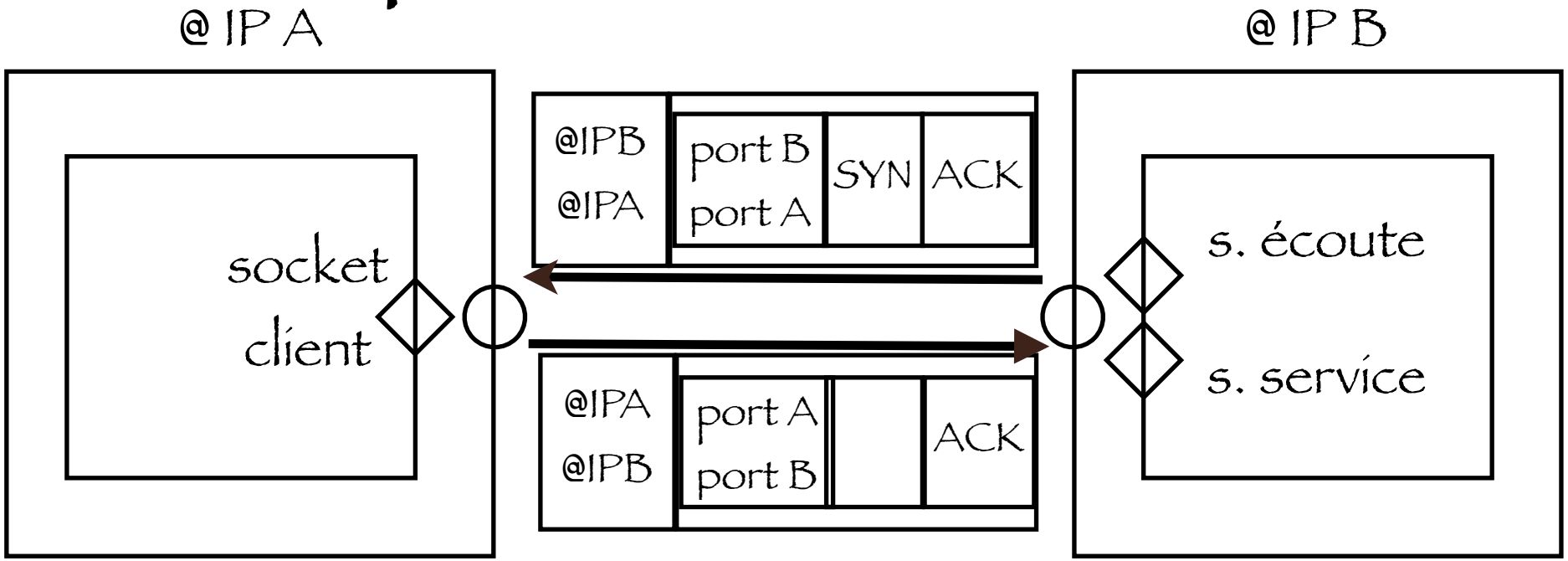
socket client

@ loc = @ IP A
port loc = port A
@ dist = @ IP B
port dist = port B
proto = TCP

s. écoute

@ loc = @ IP B ou Any
port loc = port B
@ dist = Any
port dist = 0
proto = TCP

Acceptation de connexion



socket client

@ loc = @ IP A
port loc = port A
@ dist = @ IP B
port dist = port B
proto = TCP

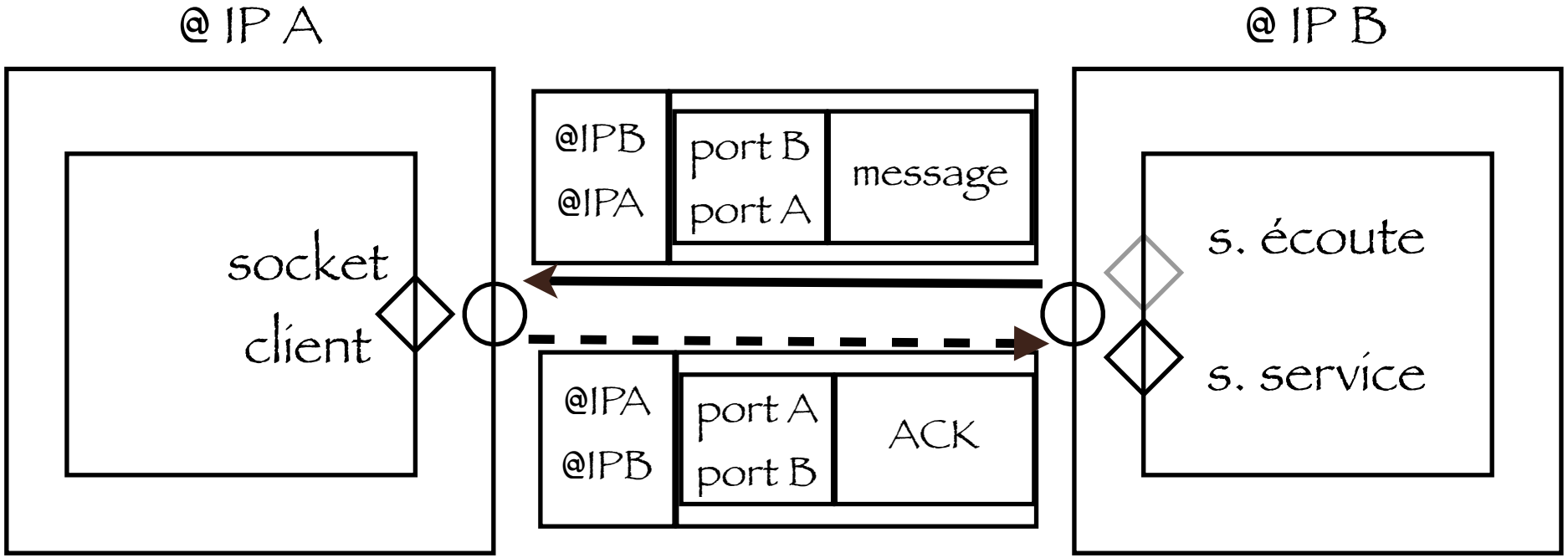
s. écoute

@ loc = @ IP B ou Any
port loc = port B
@ dist = Any
port dist = 0
proto = TCP

s. service

@ loc = @ IP B
port loc = port B
@ dist = @ IP A
port dist = port A
proto = TCP

Communication



socket client

@ loc = @ IP A
port loc = port A
@ dist = @ IP B
port dist = port B
proto = TCP

s. écoute

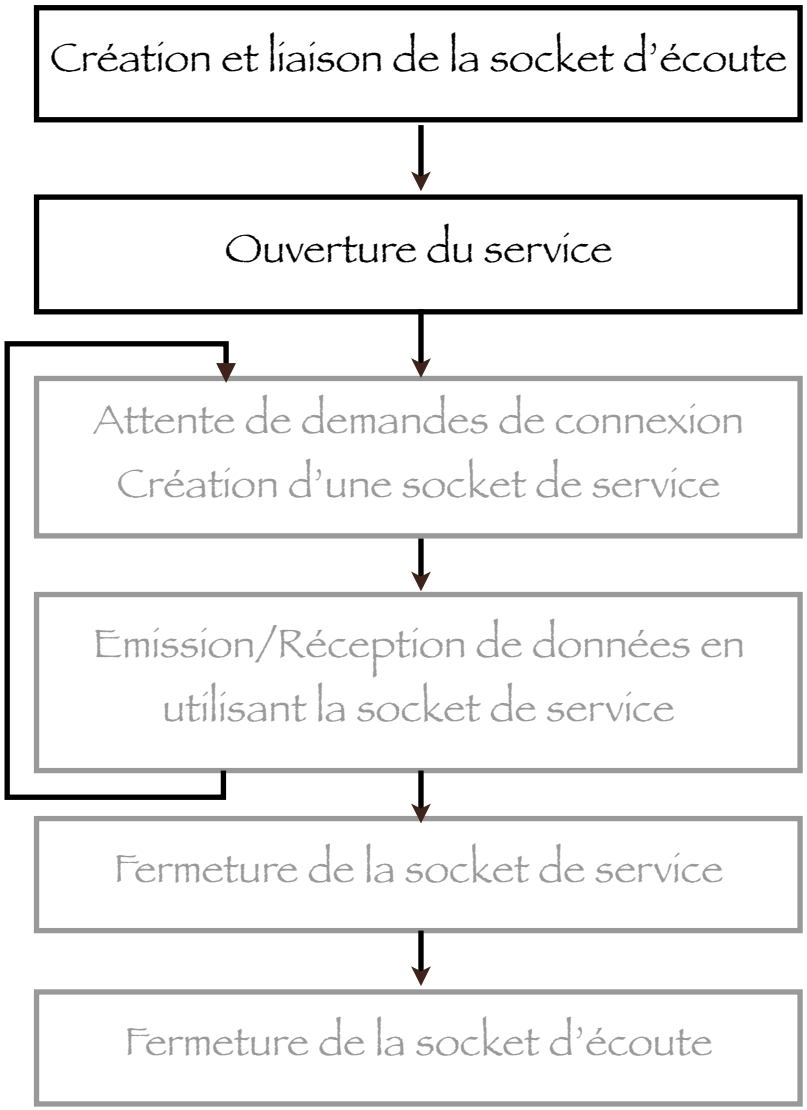
@ loc = @ IP B ou Any
port loc = port B
@ dist = Any
port dist = 0
proto = TCP

s. service

@ loc = @ IP B
port loc = port B
@ dist = @ IP A
port dist = port A
proto = TCP

Prog. en Java : TCP Serv

- Classe java.net.ServerSocket
  - abstraction des sockets d'écoute
  - utilisation du constructeur permettant de choisir le port
  - Remarque : on peut aussi utiliser d'autres constructeurs pour choisir l'@ IP de la machine et/ou la taille de la file d'attente (voir la doc java du package java.net)
  - Remarque 2 : on peut aussi utiliser le constructeur par défaut et la méthode bind qui prends en paramètre un InetSocketAddress : couple (@IP,port)

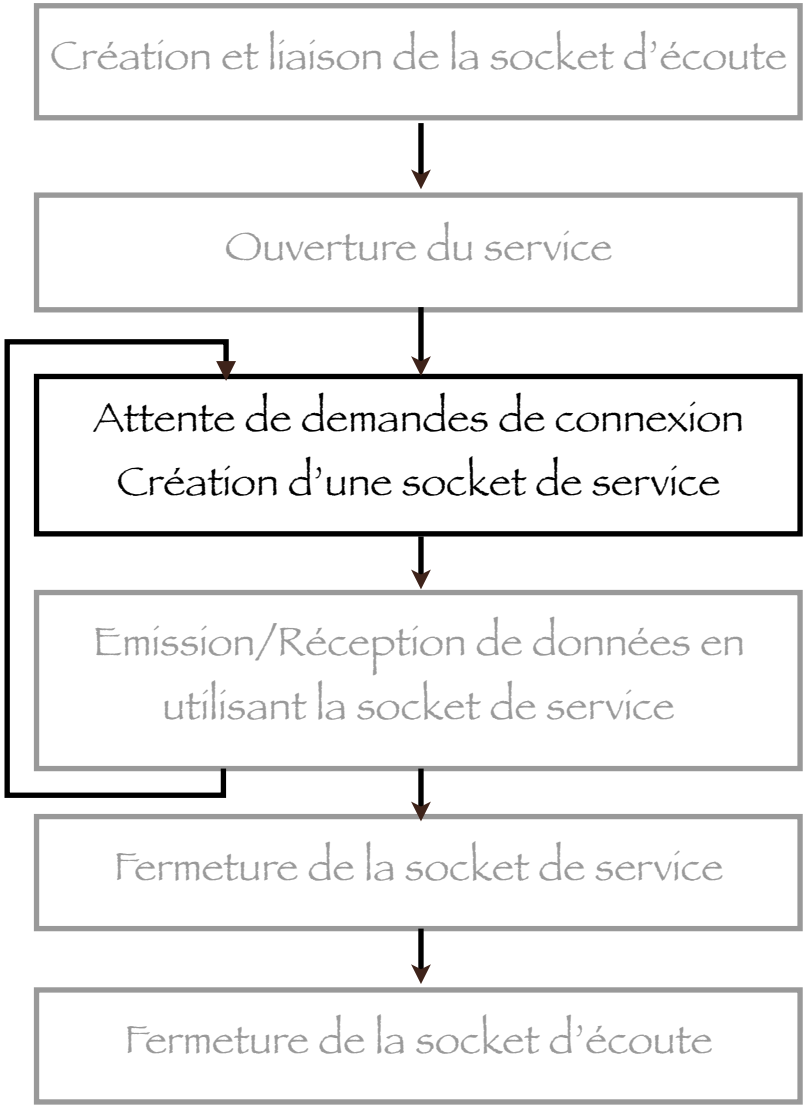


Prog. en Java : TCP Serv

```
import java.net.ServerSocket;
import java.net.Socket;
import java.io.IOException;
import java.io.DataInputStream;
import java.io.DataOutputStream;

ServerSocket sockEcoute;                // Déclaration du ServerSocket

//  Instanciation du ServerSocket en utilisant le constr. le plus simple (choix port)
try {
      sockEcoute = new ServerSocket(13214);
}
catch(IOException ioe) {
     System.out.println("Erreur de création du server socket: " + ioe.getMessage());
     return;
}
```

Prog. en Java : TCP Serv

- méthode accept
  - Attends une demande de connexion
  - Lors de la réception d'une demande => création d'une socket de service (objet de la classe Socket)
- Socket représente aussi bien les sockets de service que les sockets clients

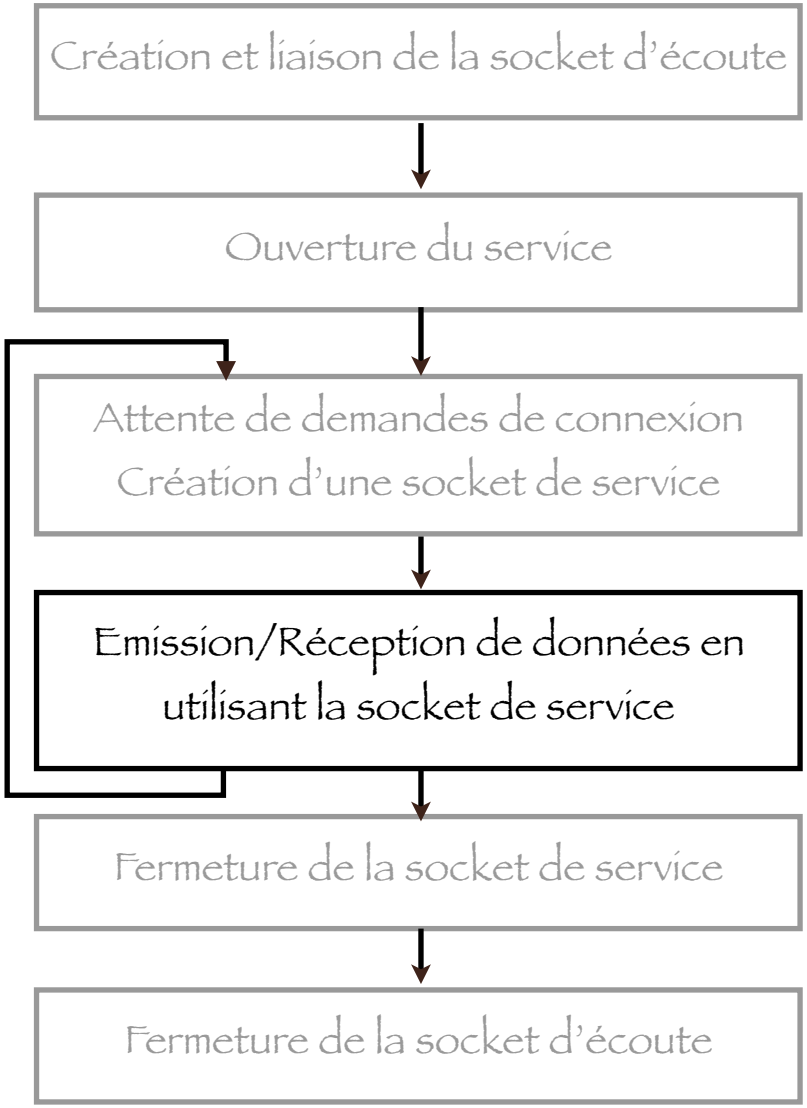


Prog. en Java : TCP Serv

```
Socket sockService;  // Declaration du socket de service

//  On appelle accept() sur le ServerSocket pour accepter les connections,
//  quand une connexion est reçue, un nouvel objet de la classe Socket est
//  renvoyé

while(true) {
    try {
        sockService = sockEcoute.accept();
    }
    catch(IOException ioe) {
        System.out.println("Erreur de accept : " + ioe.getMessage());
        break;
    }
    /* ... Traite la connexion avec le client ... */
}
```

Prog. en Java : TCP Serv

- Utilisation des classes d'E/S java (package java.io)
- Méthodes : getOutputStream et getInputStream de la classe Socket
- Retournent des flux d'E/S de base qu'on pourra encapsuler dans des flux plus complexes (BufferedReader, BufferedWriter, DataInputStream, DataOutputStream, ObjectInputStream, ObjectOutputStream...)



Prog. en Java : TCP Serv

Réception binaire

(ici en Big Endian)

```
try{
    //  Instancie un data input stream travaillant sur l'input stream de la socket
    DataInputStream iStream = new DataInputStream(sockService.getInputStream());

    //  Lit une chaîne de caractère et un entier sur le flux, et donc les reçoit du client
    String helloString = iStream.readUTF();
    int trois = iStream.readInt();
}
catch(IOException ioe) {
    System.out.println("Erreur de lecture : " + ioe.getMessage());
}
```

Prog. en Java : TCP Serv

Emission binaire

(ici en Big Endian)

```
try{
    //  Instancie un data output stream travaillant sur l'output stream de la socket
    DataOutputStream oStream = new DataOutputStream(
                                                    sockService.getOutputStream());

    //  écrit une chaîne et un flottant sur le flux, et donc les envoie au client
    oStream.writeUTF("Bonjour!");
    oStream.writeFloat(3.14f);
}

catch(IOException ioe) {
   System.out.println("Erreur d'écriture : " + ioe.getMessage());
}
```

Prog. en Java : TCP Serv

- Pour faire des émissions/réceptions en binaire Little Endian (par exemple pour communiquer avec du code C/C++ pour processeurs Intel) il existe plusieurs implémentations
- Exemple : Guava (Google Core Libraries for Java) https://code.google.com/p/guava-libraries/
  - LittleEndianDataInputStream
  - LittleEndianDataOutputStream

Prog. en Java : TCP Serv

Réception texte

```
try{
    //  Instancie un BufferedReader travaillant sur un InputStreamReader lié à
    // l'input stream de la socket
    BufferedReader reader = new BufferedReader (
                new InputStreamReader(sockService.getInputStream());

    //  Lit une ligne de caractères depuix le flux, et donc la reçoit du client
    String helloString = reader.readLine();
}
catch(IOException ioe) {
    System.out.println("Erreur de lecture : " + ioe.getMessage());
}
```

Prog. en Java : TCP Serv

Emission texte

```
try{
    //  Instancie un PrintStream travaillant sur l'output stream de la socket
    PrintStream pStream = new PrintStream(sockService.getOutputStream());

    //  écrit une ligne de caractères sur le flux, et donc l'envoie au client
    pStream.println("Bonjour!");
}

catch(IOException ioe) {
   System.out.println("Erreur d'écriture : " + ioe.getMessage());
}
```

Prog. en Java : TCP Serv

- Pour faire des émissions/réceptions d'objets en binaire (Big Endian) on utilise la sérialisation
- La classe de l'objet à émettre doit implémenter java.io.Serializable
- Attention : les attributs doivent eux aussi être sérialisables...
- La grande majorité des classes du JDK sont sérialisables

Prog. en Java : TCP Serv

Exemple de classe d'objet sérialisable

```
public class MaClasse implements Serializable {

        private int unEntier;
        private String uneChaine; // String est sérialisable

        public MaClasse() {
            unEntier = 0; uneChaine = "";
        }
        public int getUnEntier() {
            return unEntier;
        }
    ...
}
```

Prog. en Java : TCP Serv

Réception d'objet

```
try{
    //  Instancie un object input stream travaillant sur l'input stream de la socket
    ObjectInputStream oiStream = new ObjectInputStream(sockService.getInputStream());

    //  Lit un objet sur le flux, et donc le reçoit du client
    Object o = oiStream.readObject();
    if (o instanceof MaClasse) { // on vérifie qu'on puisse caster
        MaClasse monObjet = (MaClasse) o;
        ...
    }
}
catch(IOException ioe) {
    System.out.println("Erreur de lecture : " + ioe.getMessage());
}
```

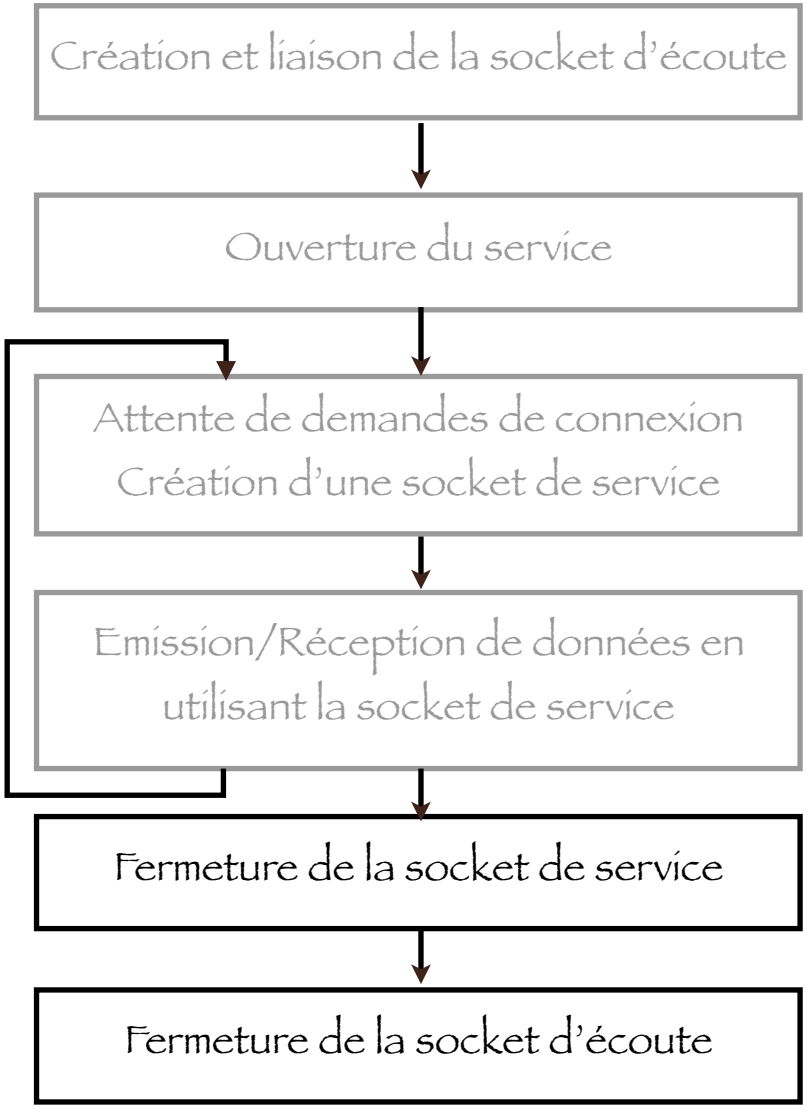
Prog. en Java : TCP Serv

Emission d'objet

```
try{
     //  Instancie un object output stream travaillant sur
     // l'output stream de la socket
     ObjectOutputStream ooStream = new ObjectOutputStream(
                                                    sockService.getOutputStream());

     //  écrit un objet sur le flux, et donc l'envoie au client
     MaClasse monObjet = new MaClasse();
     // peut générer une NotSerializableException
     ooStream.writeObject(monObjet);

} catch(IOException ioe) {
    System.out.println("Erreur d'écriture : " + ioe.getMessage());
}
```

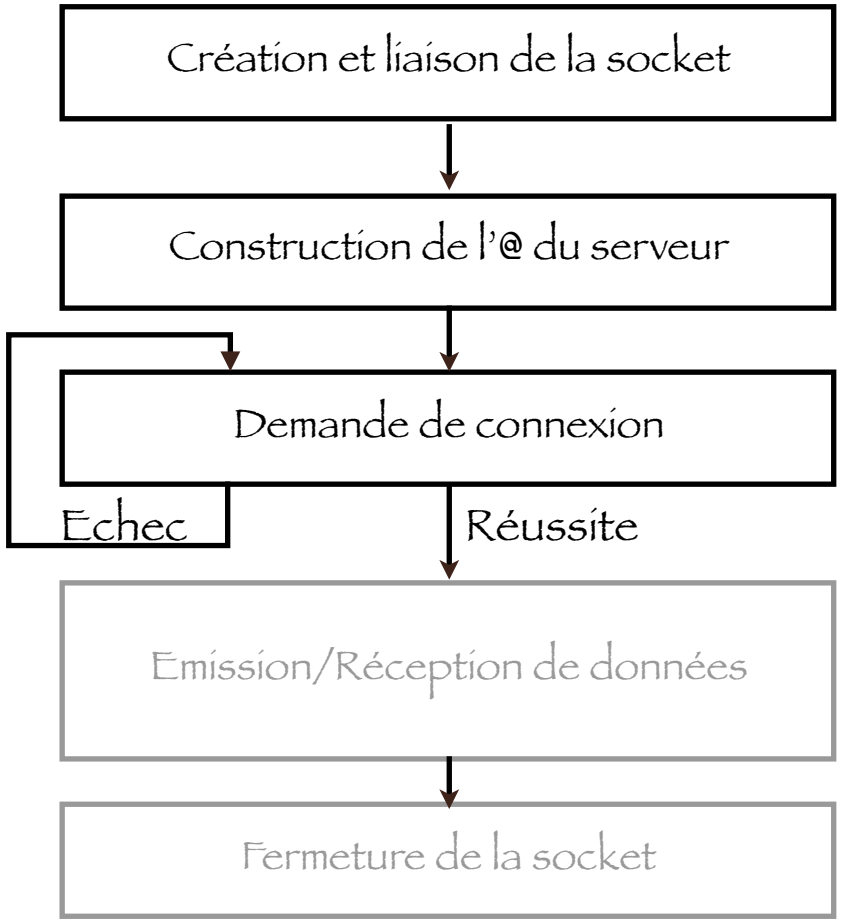
Prog. en Java : TCP Serv

- Méthode close des classes Socket et ServerSocket
  - code : sock.close() + try/catch IOException



Prog. en Java : TCP Client

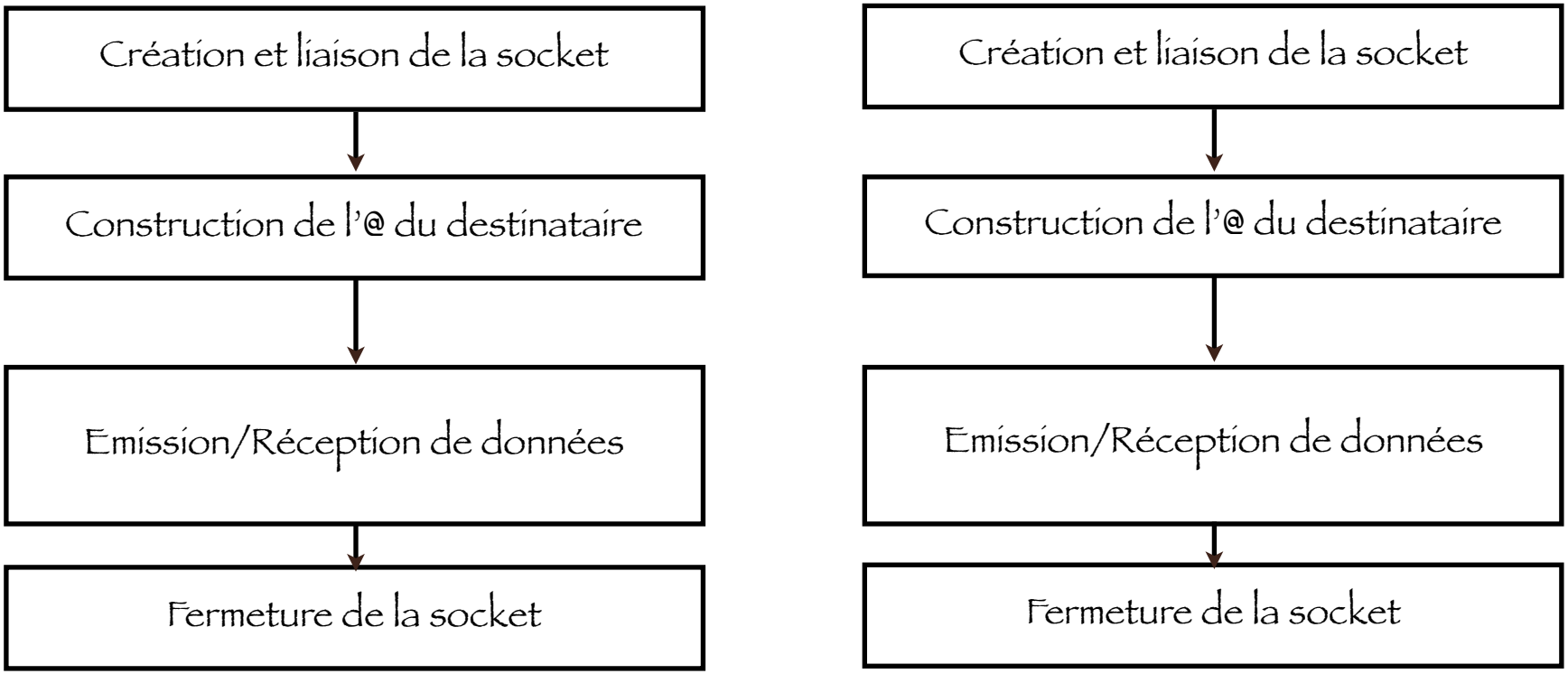
- Classe Socket
  - Utilisation d'un des constructeurs
  - Ils créent le socket, le lie et envoie la demande de conn.
  - Le plus utilisé permet de préciser l'adresse IP de destination ou le nom et le numéro de port



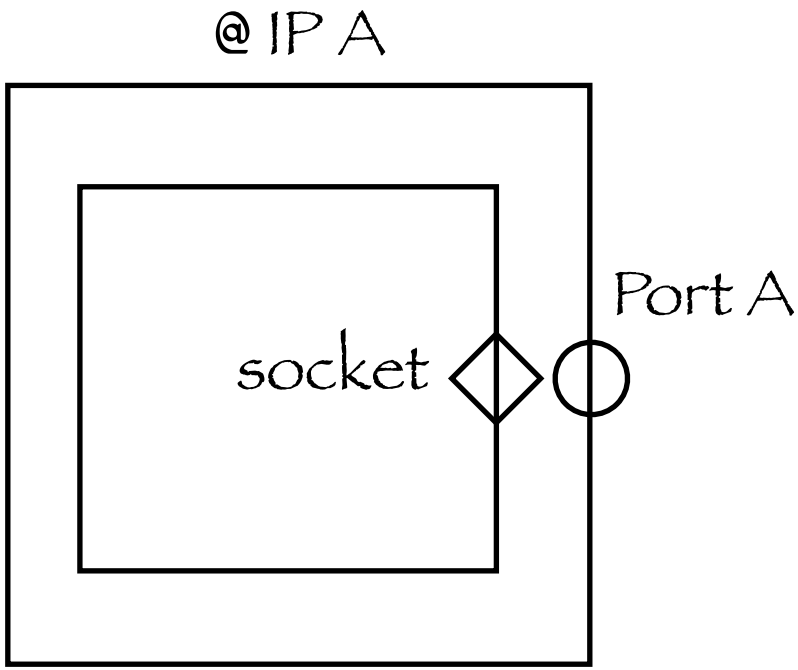
Prog. en Java : TCP Client

```
import java.net.Socket;
import java.io.IOException;
import java.io.DataInputStream;
import java.io.DataOutputStream;

Socket sock;                //  Déclaration du socket

//  Instanciation du socket en précisant le nom de machine et le port
try {
     sock = new Socket("marine.edu.ups-tlse.fr", 13214);
     // autres solutions :
     // sock = new Socket("10.5.4.1", 13214);
     // sock = new Socket("2001:cdba::3257:9652", 13214);
}
catch(IOException ioe) {
     System.out.println("Erreur de création ou de connexion : "
                              + ioe.getMessage());
     return;
}
```

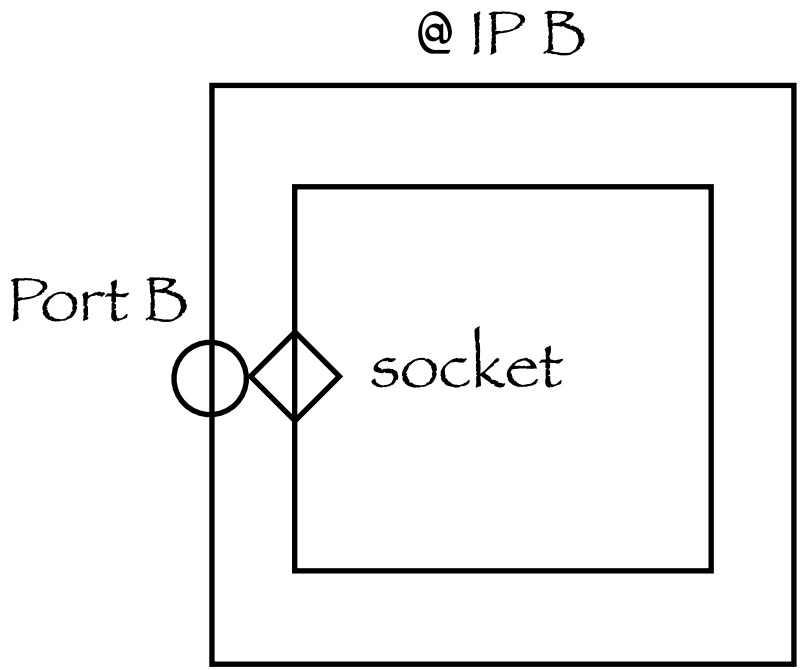
Emission/réception avec UDP



Création sockets + bind

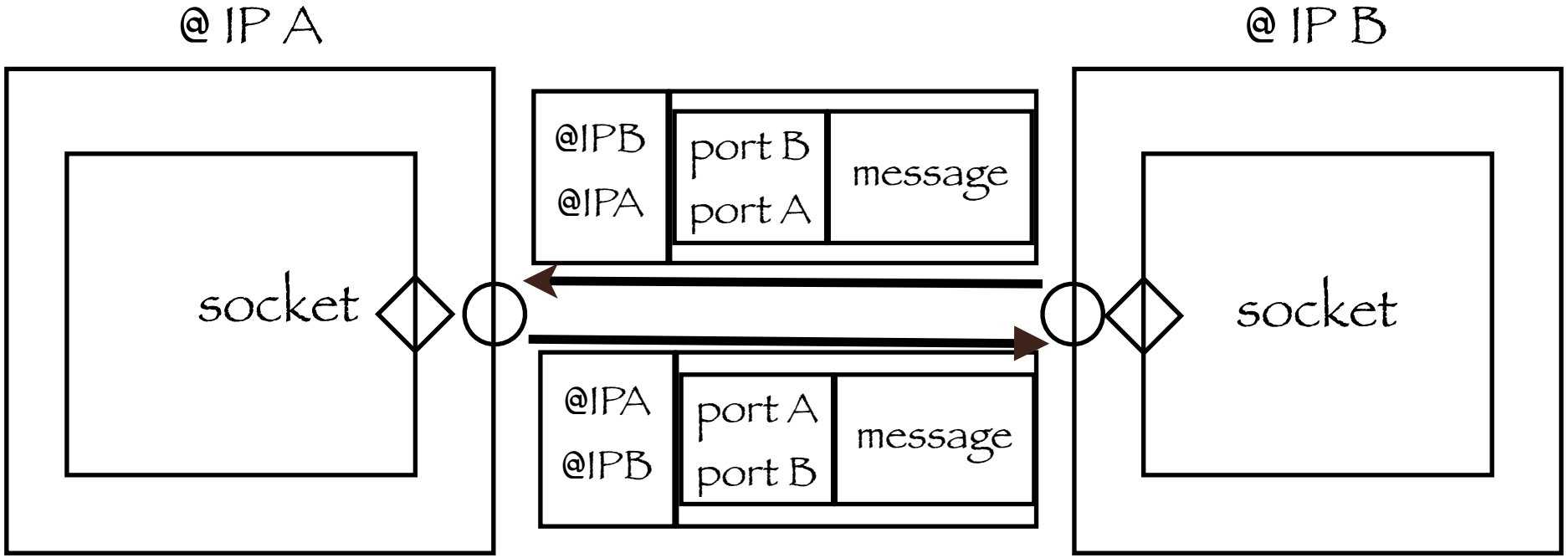


@ loc = @ IP A ou Any
port loc = port A
@ dist = Any
port dist = 0
proto = UDP



@ loc = @ IP B ou Any
port loc = port B
@ dist = Any
port dist = 0
proto = UDP

Communication



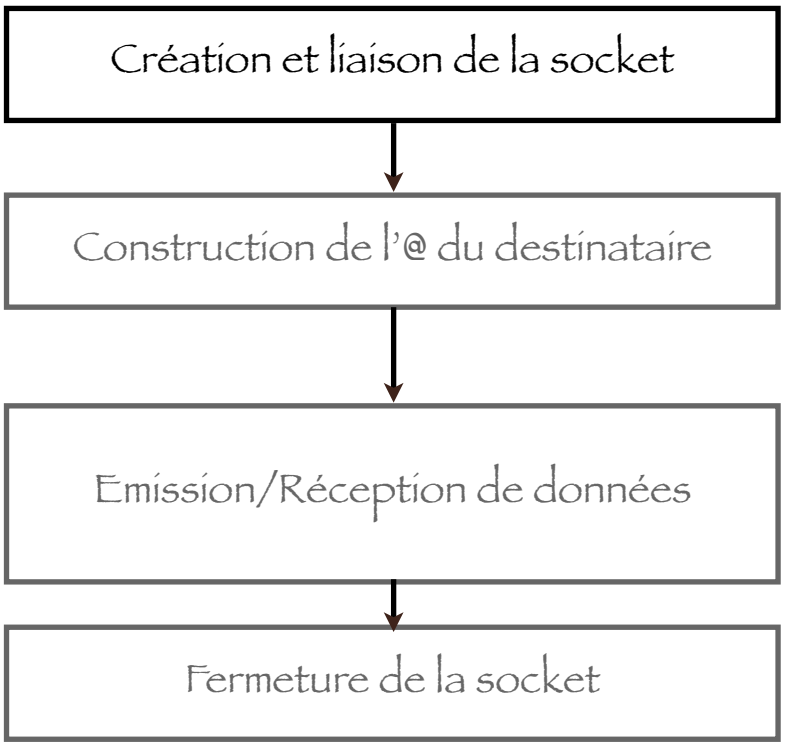
@ loc = @ IP A ou Any
port loc = port A
@ dist = Any
port dist = 0
proto = UDP

Pas de connexion,
en général on précise
la destination à chaque envoi

@ loc = @ IP B ou Any
port loc = port B
@ dist = Any
port dist = 0
proto = UDP

Prog. en Java : UDP

- Classe DatagramSocket
  - Crée une socket
  - Constructeurs :
    - défaut (socket lié à un port choisi par le système)
    - choix du port
    - choix du port et de l'adresse IP (cas où la machine a plusieurs @ IP)

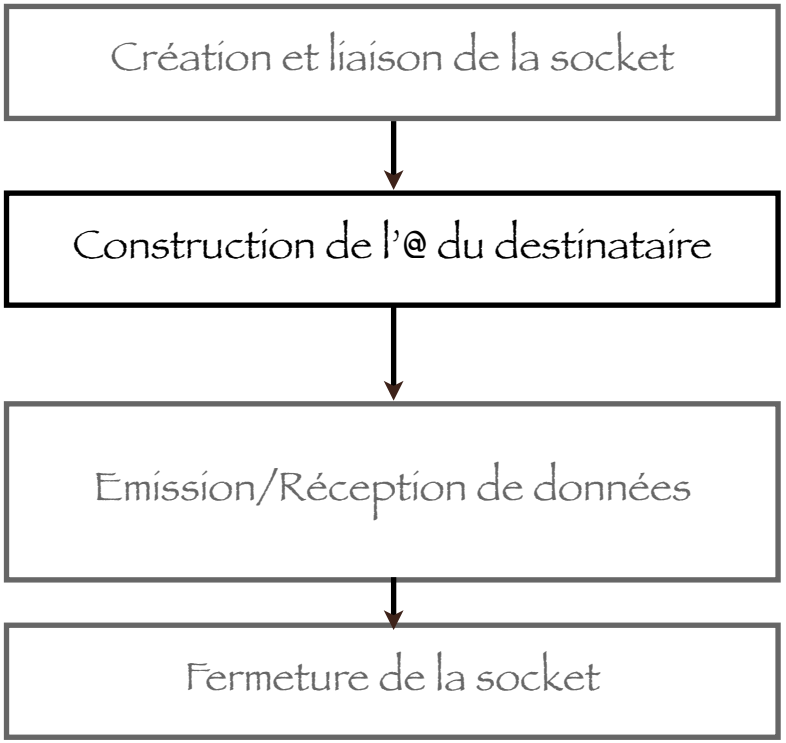


Prog. en Java : UDP

```
import java.net.DatagramSocket;
import java.io.IOException;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;

DatagramSocket sock;    // Déclaration d'un socket datagramme

try {
    sock = new DatagramSocket(13214);     // Lie au port UDP 13214
}

catch(IOException ioe) {
    System.out.println("Erreur création socket: " + ioe.getMessage());
    return;
}
```

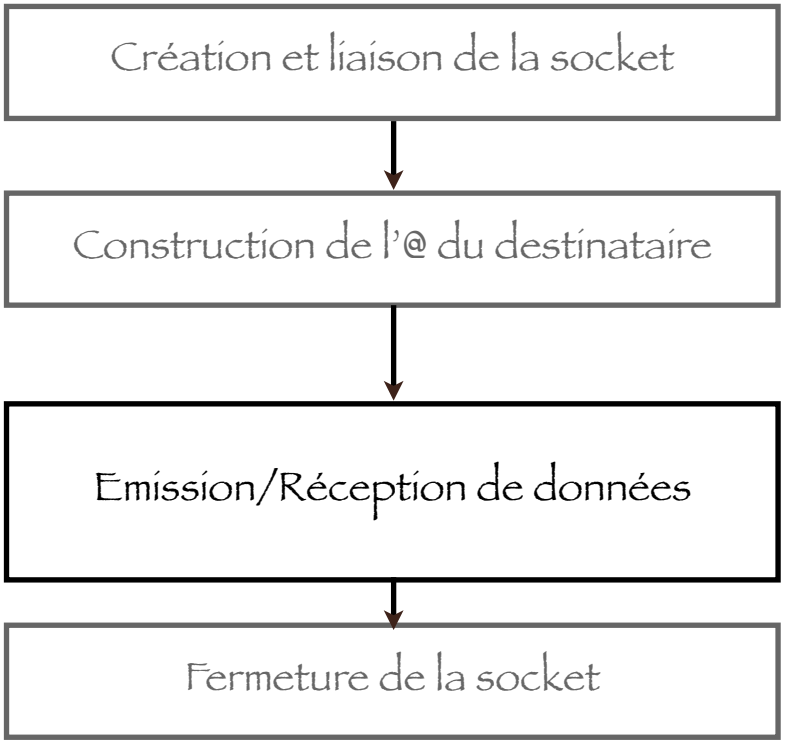
Prog. en Java : UDP

- Classe InetAddress
  - Représente les @IP v4 et v6 (2 classes dérivées Inet4Address et Inet6Address)
  - 3 méthodes statiques :
    - InetAddress getByName(String nom) : résolution de nom ou traduction d'adresse en notation standard
    - InetAddress getLocalHost() : adresse IP locale
    - InetAddress[] getAllByName(String nom) : renvoie toutes les adresses associées à un nom



Prog. en Java : UDP

- Classe DatagramPacket
  - Représente un datagramme UDP à émettre/recevoir
  - 2 constructeurs principaux
    - Emission : 4 params (données, longueur, @IP, port)
    - Réception : 2 params (tampon, longueur)
  - Méthodes pour récupérer : données, longueur (des données reçues effectivement), @IP et port distants, @IP et ports locaux (offset éventuel)
- Méthodes send/receive de DatagramSocket



Envoyer avec UDP

```
try{
    //  Construction de l'@IP et du port destination
    InetAddress destAddr = InetAddress.getByName("10.25.43.9"); // ou @IPv6 ou nom
    int destPort = 13214;
    //  Exemple d'utilisation avec un ByteArrayOutputStream
    ByteArrayOutputStream boStream = new ByteArrayOutputStream();
    DataOutputStream oStream = new DataOutputStream(boStream);
    oStream.writeUTF("Salut !");                    // Ecriture de données dans le flux
    oStream.writeInt(3);
    byte[] dataBytes = boStream.getByteArray(); // Conversion du flux en tableau d'octets
    DatagramPacket dgram =                          //  Construction du DatagramPacket
        new DatagramPacket(dataBytes, dataBytes.length, destAddr, destPort)
    sock.send(dgram);
}
catch(IOException ioe) {
    System.out.println("Erreur d'émission : " + ioe.getMessage());
}
```

Recevoir avec UDP

```
try{
    // Construction du tampon et de l'objet qui vont servir à recevoir
    byte[] buffer = new byte[255];
    DatagramPacket dgram = new DatagramPacket(buffer, buffer.length);

     // Attends puis reçoit un datagramme
     sock.receive(dgram);                          // Infos sur l'émetteur :
                                                   // dgram.getAddress() et dgram.getPort()

    //  Extrait les données
    ByteArrayInputStream biStream =
          new ByteArrayInputStream(buffer, 0, dgram.getLength());
    DataInputStream iStream = new DataInputStream(biStream);
    String salutString = iStream.readUTF();
    int trois = iStream.readInt();
}
catch(IOException ioe) {
    System.out.println("Erreur socket : " + ioe.getMessage());
}
```

Diffusion avec UDP

- Presque identique à UDP/IP unicast
  - Il suffit d'utiliser une addresse de diffusion comme adresse de destination.

InetAddress destAddr = InetAddress.getByName("255.255.255.255")

  - Note : un socket datagramme peut recevoir à la fois du traffic unicast et broadcast

Multicasting UDP

- Pour utiliser le multicast il faut :
  - Utiliser la classe MulticastSocket au lieu de DatagramSocket (MS hérite de DS)
  - Utiliser une adresse multicast comme destination
    - Exemple (pour IPv4) 225.0.0.1

```
InetAddress destAddr = InetAddress.getByName("225.0.0.1");
```

    - On peut aussi préciser le TTL pour limiter la portée du multicast

```
sock.setTimeToLive(1);
// puis on envoie le datagramme
sock.send(dgram);
```

Réception avec UDP multicast

- Pour recevoir il faut s'abonner à l'adresse IP multicast comme cela :

```
sock.joinGroup(InetAddress.getByName("225.0.0.1"));
// ou encore en IPv6
// sock.joinGroup(InetAddress.getByName("ff05::1"));
```

- Pour résilier l'abonnement :

```
sock.leaveGroup(InetAddress.getByName("225.0.0.1"));
// ou encore en IPv6
// sock.leaveGroup(InetAddress.getByName("ff05::1"));
```

Asynchronisme

- Attention à l'asynchronisme :
  - Recevoir est toujours une opération bloquante
  - Pour TCP, la connexion, l'attente (accept) et parfois l'émission sont bloquantes
- Solutions : les threads ou les opérations select de NIO (New IO)

Threads

- Serveur TCP
  - 1 thread bloqué sur l'accept
  - 1 thread par connexion active
- Client TCP ou utilisation d'UDP
  - 1 thread de réception
  - 1 thread applicatif (qui peut émettre)

Serveur TCP Echo avec Thread

```
public class EchoServerThreads implements Runnable {
    private Socket sss;
    public EchoServerThreads(Socket sss) {
        this.sss = sss;
    }
    public void run() { // traitement d'un client
        try {
            InputStream entreeSocket = sss.getInputStream();
            OutputStream sortieSocket = sss.getOutputStream();
            int b = 0;
            while (b != -1) {
                b = entreeSocket.read();
                sortieSocket.write(b);
            }
            sss.close();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
    public static void main(String args[]) throws IOException {
        ServerSocket ssg = new ServerSocket(1234);
        while (true) {
            Socket sss = ssg.accept();
            Thread t = new Thread(new EchoServerThreads(sss));
            t.start();
        }
    }
}
```

NIO

- Concept de sélecteur
  - Attente sur plusieurs flux d'E/S en parallèle
- En C/C++ : select (appel système)
- En Java : Selector du package NIO
  - Permet une attente d'E/S gérés par des classes dérivant de SelectableChannel

NIO

- Channel
  - Représente un canal bidirectionnel d'E/S
  - Permet des E/S par bloc (Stream et Reader/Writer fonctionnent eux par octet)
- Buffer
  - Représente un bloc d'information à lire/écrire sur un Channel

NIO

- Pour les sockets
  - ServerSocketChannel : canal associé à un ServerSocket
  - SocketChannel : canal associé à un Socket
  - DatagramChannel : canal associé à un DatagramSocket
- Utilisation de ByteBuffer : bloc d'octets

NIO

- ByteBuffer
  - création avec la méthode statique : allocate()

    ```
    ByteBuffer buffer = ByteBuffer.allocate(1024);
    ```

  - possibilité de réutiliser un tableau : méthode statique wrap (peu recommandé)
  - attributs
    - position : position courante de lecture/écriture
    - limit : partie utilisée du buffer après un flip
    - capacity : taille max du buffer

NIO

- ByteBuffer
  - écrire des données dans un buffer : utilisation d'une des méthodes put

```
buffer.put(b); // b est un byte ; position avance de 1
buffer.putInt(1); // position avance de 4
buffer.putChar(0,'a'); // écriture absolue
```

  - lire des données depuis un buffer : utilisation d'une des méthodes get

```
byte b = buffer.get(); // position avance de 1
int i = buffer.getInt(); // position avance de 4
buffer.getChar(0); // lecture absolue
```

NIO

- ByteBuffer
  - clear doit être appelée avant de remplir un buffer depuis un canal

```
buffer.clear(); // position <= 0, limit <= capacity
```

  - flip doit être appelée avant d'écrire un buffer sur un canal

```
buffer.flip(); // limit <= position, position <= 0
```

NIO

- ServerSocketChannel
  - création avec la méthode statique : open()

```
ServerSocketChannel ssc = ServerSocketChannel.open();
```

  - liaison à un port

```
ServerSocket ss = ssc.socket(); // socket serveur correspondant
InetSocketAddress address = new InetSocketAddress(1234);
ss.bind( address );
```

  - utilisation de la méthode accept() qui renvoie un SocketChannel

```
SocketChannel sc = ssc.accept();
```

  - fermeture avec la méthode close()

```
ssc.close();
```

NIO

- SocketChannel
  - création avec la méthode statique : open()

```
SocketChannel sc = SocketChannel.open(); // pour le client
```

  - connexion

```
InetSocketAddress address = new InetSocketAddress("localhost", 1234);
sc.connect(address); // connexion
```

  - lecture de données dans un buffer

```
sc.read(echoBuffer);
```

  - écriture de données d'un buffer

```
sc.write(echoBuffer);
```

  - fermeture avec la méthode close()

```
sc.close();
```

Serveur TCP Echo simple en NIO

```
public class NIOEchoSimple {

    static public void main( String args[] ) throws Exception {
        ByteBuffer echoBuffer = ByteBuffer.allocate( 1024 ); //  buffer de 1024 octets
        ServerSocketChannel ssc = ServerSocketChannel.open(); // canal socket serveur
        ServerSocket ss = ssc.socket(); // socket serveur correspondant
        InetSocketAddress address = new InetSocketAddress(1234);
        ss.bind( address ); // lie le socket au port 1234
        while (true) {
            // Attends puis accepte une nouvelle connexion
            SocketChannel sc = ssc.accept();
            int r = 0;
            while (r != -1) {
                echoBuffer.clear(); // prépare le buffer pour la lecture
                r = sc.read(echoBuffer); // lit depuis le canal dans le buffer
                if (r>0) { // si le client n'a pas fermé la connexion
                    echoBuffer.flip(); // prépare le buffer pour l'écriture
                    sc.write( echoBuffer ); // écrit le buffer sur le canal
                }
            }
            sc.close();
        }
    }
}
```

Client TCP Echo simple en NIO

```
public class NIOEchoClient {
    static public void main( String args[] ) throws Exception {
        ByteBuffer echoBuffer = ByteBuffer.allocate( 1024 ); //  buffer de 1024 octets
        SocketChannel sc = SocketChannel.open(); // canal socket
        InetSocketAddress address = new InetSocketAddress("localhost", 1234);
        sc.connect( address ); // connexion
        String s = "Hello World";
        echoBuffer.put(s.getBytes()); // place le message dans le buffer
        echoBuffer.flip(); // prépare le buffer pour l'écriture
        sc.write(echoBuffer); // écrit le buffer sur le canal
        echoBuffer.clear(); // prépare le buffer pour la lecture
        sc.read(echoBuffer); // lit depuis le canal dans le buffer

        System.out.println(new String(echoBuffer.array(), 0, echoBuffer.limit()));
        sc.close();
    }
}
```

NIO

- DatagramChannel
  - création avec la méthode statique : open()

```
DatagramChannel dc = DatagramChannel.open(); // pour le client
```

  - liaison à un port (pour recevoir)

```
DatagramSocket ds = dc.socket(); // socket correspondant
InetSocketAddress address = new InetSocketAddress(1234);
ss.bind(address);
```

  - réception de données dans un buffer

```
sc.receive(echoBuffer);
```

  - envoi de données d'un buffer vers une dest (@,port)

```
InetSocketAddress dest = new InetSocketAddress("localhost", 1234);
sc.send(echoBuffer, dest);
```

  - fermeture avec la méthode close()

```
sc.close();
```

Emetteur UDP en NIO

```
public class NIODGSend {
    static public void main( String args[] ) throws Exception {
        ByteBuffer echoBuffer = ByteBuffer.allocate( 1024 );
        DatagramChannel dc = DatagramChannel.open();
        InetSocketAddress dest = new InetSocketAddress("localhost", 1234);
        String s = "Hello World";
        echoBuffer.put(s.getBytes()); // place le message dans le buffer
        echoBuffer.flip(); // prépare le buffer pour l'écriture
        dc.send(echoBuffer, dest); // envoie le buffer vers dest
        dc.close();
    }

}
```

Récepteur UDP en NIO

```
public class NIODGRecv {
    static public void main( String args[] ) throws Exception {
        ByteBuffer echoBuffer = ByteBuffer.allocate( 1024 );
        DatagramChannel dc = DatagramChannel.open();
        DatagramSocket ds = dc.socket(); // socket associé
        InetSocketAddress address = new InetSocketAddress(1234);
        ds.bind(address); // lie le socket au port 1234
        echoBuffer.clear(); // prépare le buffer pour la lecture
        dc.receive(echoBuffer); // lit depuis le canal dans le buffer

        System.out.println(new String(echoBuffer.array(), 0, echoBuffer.limit()));
        dc.close();
    }

}
```

NIO

- Selector : E/S asynchrones
  - création avec la méthode statique : open()

```
Selector selector = Selector.open();
```

  - enregistrement de canal avec un type d'opération (accept, connect, read ou write)

```
ssc.configureBlocking( false ); // ssc est un ServerSocketChannel
SelectionKey cle = ssc.register(selector, SelectionKey.OP_ACCEPT);
sc.configureBlocking( false ); // sc est un SocketChannel ou un DatagramC
SelectionKey cle2 = ssc.register(selector, SelectionKey.OP_READ);
```

  - attente bloquante sur tous les canaux : select()

```
int num = selector.select(); // num contient le nombre de canaux prêts
int num = selector.select(500); // attends pendant 500 ms
```

NIO

- Selector : E/S asynchrones
  - récupération de l'ensemble des canaux prêts

```
Set<SelectionKey> selectedKeys = selector.selectedKeys();
Iterator<SelectionKey> it = selectedKeys.iterator();
while (it.hasNext()) {
 SelectionKey key = (SelectionKey)it.next();
 it.remove(); // important : il faut enlever la clé de l'itérateur
 // pour indiquer qu'on a traité l'opération d'E/S
 if (key.isAcceptable()) { ... }
 else if (key.isReadable()) { ... }
 else if (key.isWritable()) { ... }
 else if (key.isConnectable()) { ... }
}
```

Serveur TCP Echo async en NIO

```
public class NIOEcho {
    static public void main( String args[] ) throws Exception {
        ByteBuffer echoBuffer = ByteBuffer.allocate( 1024 );
        // Création du selector
        Selector selector = Selector.open();
        // Création du canal server socket
        ServerSocketChannel ssc = ServerSocketChannel.open();
        // devient non bloquant
        ssc.configureBlocking( false );
        // liaison sur le port 1234
        ServerSocket ss = ssc.socket();
        InetSocketAddress address = new InetSocketAddress( 1234 );
        ss.bind( address );
        // enregistrement dans le selecteur
        SelectionKey keyServ = ssc.register( selector, SelectionKey.OP_ACCEPT );
        while (true) {
            int num = selector.select(); // attente
            Set<SelectionKey> selectedKeys = selector.selectedKeys();
            Iterator<SelectionKey> it = selectedKeys.iterator();
```

Serveur TCP Echo async en NIO

```
   while (it.hasNext()) {
      SelectionKey key = (SelectionKey)it.next();
// enlève la clé de l'itérateur
it.remove();
      if (key.isAcceptable()) {
         // accepte la connexion
         SocketChannel sc = ssc.accept();
         sc.configureBlocking( false );
         // enregistrement du canal dans le selecteur
         SelectionKey newKey = sc.register( selector,
      SelectionKey.OP_READ );
         System.out.println( "Got connection from "+sc );
      } else if (key.isReadable()) {
         // Récupération du canal
         SocketChannel sc = (SocketChannel)key.channel();
```

Serveur TCP Echo async en NIO

```
                // boucle d'echo
                while (true) {
                    echoBuffer.clear();
                    int r = sc.read( echoBuffer );
                    if (r<=0) {
                        if (r < 0) {
                            // le client a fermé la connexion
            // => enlève le canal du sélecteur
                            sc.close();
                        }
                        break;
                    }
                    echoBuffer.flip();
                    sc.write( echoBuffer );
                }
            }
        }
    }
}
```

Conclusion

- Simple d'utilisation grâce à Java
- On peut utiliser la sérialisation pour envoyer des objets
- Asynchronisme possible grâce aux threads et à NIO
- Possibilité d'utiliser la cryptographie pour plus de sécurité : package javax.net.ssl