Systèmes Distribués

Licence Informatique 3ème année

Sockets TCP/UDP et leur mise en œuvre en C & Java

Eric Cariou

Université de Pau et des Pays de l'Adour
UFR Sciences Pau – Département Informatique

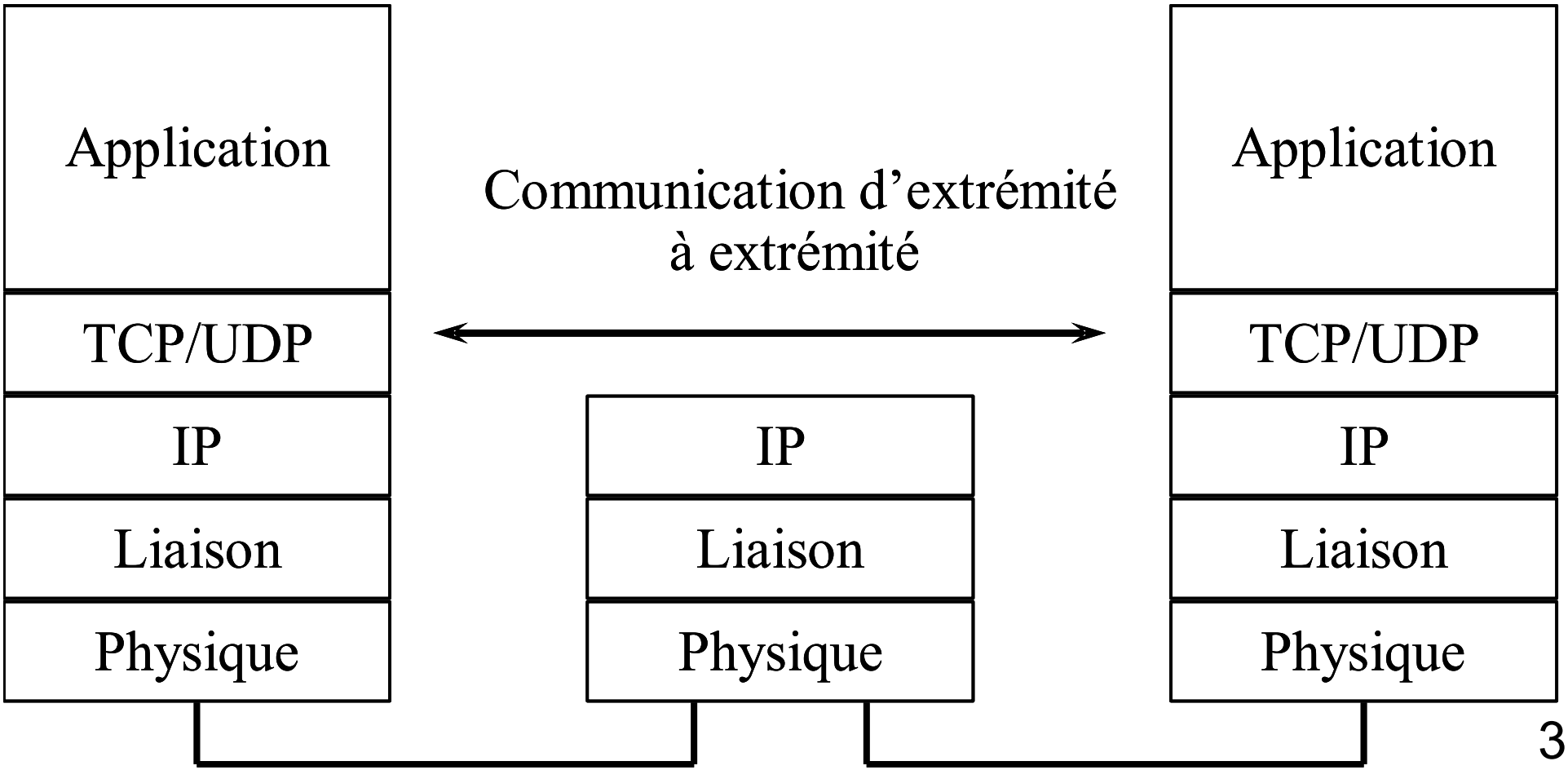
Eric.Cariou@univ-pau.fr

Plan

1. Présentation générale des sockets
2. Sockets UDP
3. Sockets TCP
4. Multicast UDP/IP

Rappel sur les réseaux

- TCP ou UDP
  - Communication entre systèmes aux extrémités
  - Pas de visibilité des systèmes intermédiaires



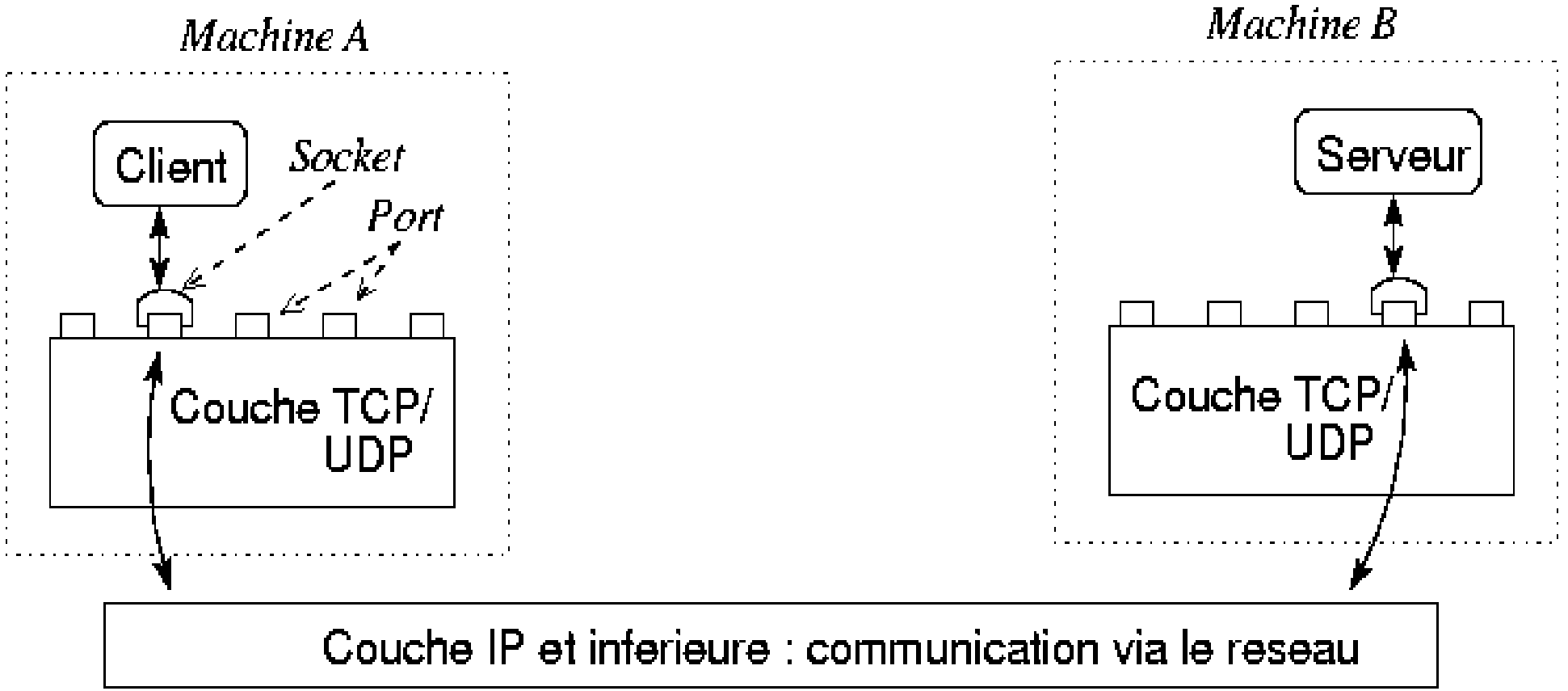
Adressage

- Adressage pour communication entre applications
  - Adresse « réseau » application = couple de 2 informations
    - Adresse IP : identifiant de la machine sur laquelle tourne l'appli
    - Numéro de port : identifiant local réseau de l'application
  - Couche réseau : adresse IP
    - Ex : 192.129.12.34
  - Couche transport : numéro de port TCP ou UDP
    - Ce numéro est en entier d'une valeur quelconque
      - Ports < 1024 : réservés pour les applications ou protocoles systèmes
      - Exemple : 80 = HTTP, 21 = FTP, ...
    - Sur un port : réception ou envoi de données
  - Adresse notée : *@IP:port* ou *nomMachine:port*
    - 192.129.12.34:80 : accès au serveur Web tournant sur la machine d'adresse IP 192.129.12.34

Sockets

- Socket : prise
  - Associée, liée localement à un port
    - C'est un point d'accès aux couches réseaux
  - Services d'émission et de réception de données sur la socket via le port
  - En mode connecté (TCP)
    - Connexion = tuyau entre 2 applications distantes
    - Une socket est un des deux bouts du tuyau
    - Chaque application a une socket locale pour gérer la communication à distance
  - Une socket peut-être liée
    - Sur un port précis à la demande du programme
    - Sur un port quelconque libre déterminé par le système
    - Par défaut, on ne peut lier qu'une socket par port

Sockets



- Une socket est
  - Un point d'accès aux couches réseau TCP/UDP
  - Liée localement à un port
    - Adressage de l'application sur le réseau : son couple @IP:port
- Elle permet la communication avec un port distant sur une machine distante : c'est-à-dire avec une application distante

Client/serveur avec sockets

- Il y a toujours différenciation entre une partie client et une partie serveur
  - Deux rôles distincts au niveau de la communication via TCP/UDP
  - Mais possibilité que les éléments communiquant jouent un autre rôle ou les 2 en même temps
- Différenciation pour plusieurs raisons
  - Identification : on doit connaître précisément la localisation d'un des 2 éléments communicants
    - Le coté serveur communique via une socket liée à un port précis : port d'écoute
    - L'adresse du serveur (@IP et port) est connue du client
  - Dissymétrie de la communication/connexion
    - Le client initie la connexion ou la communication

Sockets UDP

Sockets UDP : principe

- Mode datagramme
  - Envois de paquets de données (datagrammes)
  - Pas de connexion entre parties client et serveur
  - Pas de fiabilité ou de gestion de la communication
    - Un paquet peut ne pas arriver en étant perdu par le réseau et sans que l'émetteur en soit informé
    - Un paquet P2 envoyé après un paquet P1 peut arriver avant ce paquet P1 (selon la gestion des routes dans le réseau)
    - Un paquet envoyé à un destinataire non prêt à en recevoir (pas de socket sur le port) est détruit sans en informer l'émetteur
- Caractéristiques des primitives de communication
  - Émission de paquets est non bloquante
  - Réception de paquets est bloquante (sauf s'il y avait des paquets non lus)

Sockets UDP : principe

- Principe de communication
  - La partie serveur crée une socket et la lie à un port UDP particulier
  - La partie client crée une socket pour accéder à la couche UDP et la lie sur un port quelconque
  - Le serveur se met en attente de réception de paquet sur sa socket
  - Le client envoie un paquet via sa socket en précisant l'adresse du destinataire : couple @IP/port de la partie serveur
    - @IP de la machine sur laquelle tourne la partie serveur et numéro de port sur lequel est liée la socket de la partie serveur
  - Il est reçu par le serveur (sauf si problème réseau)
  - L'adresse du client (@IP et port) est précisée dans le paquet, le serveur peut alors lui répondre

Sockets UDP en C

Structures de données C

- Ensemble de structures de données pour manipulation des adresses des machines, des identifications réseau ...
- Adresse IP (v4) d'une machine
  - Fichier `<netinet/in.h>`
  - ```
    typedef uint32_t in_addr_t;
    struct in_addr  {
        in_addr_t s_addr;
      };
    ```
  - Les 4 octets d'une adresse IP sont codés par un entier non signé sur 32 bits
    - Ex. : 192.168.12.40 correspond à l'entier 0xC0A80C28
    ```

Conversions adresse IP

- Fichier `<arpa/inet.h>`
- Codage `in_addr` vers chaine de type "X.X.X.X"
  - `char *inet_ntoa(struct in_addr adresse)`
    - Renvoie une chaîne statique
- Codage chaîne "X.X.X.X" vers `in_addr` ou `long`
  - ```
    int inet_aton(const char *chaine,
                  struct in_addr *adresse)
    unsigned long inet_addr(char *chaine)
    ```
    - `inet_aton` pas disponible sur tous systèmes
    - `inet_addr` fonction standard
      - Retourne valeur `INADDR_NONE` en cas d'erreur
      - Attention : `INADDR_NONE` correspond aussi à une adresse de broadcast (255.255.255.255)
    ```

Structures de données C

- Identifiants d'une machine
  - Fichier `<netdb.h>`
  - ```
    struct hostent {
      char *h_name;          nom officiel,
      char **h_aliases;      liste des alias,
      int h_addrtype;        type d'adresse,
      int h_length;          longueur de l'adresse,
      char **h_addr_list;    liste des adresses
      #define h_addr h_addr_list[0]  première
                                        adresse
    };
    ```
  - Type d'adresse : internet (IP v4) par défaut
    - Valeur = AF_INET, longueur = 4 (en octets)
  - Une machine peut avoir plusieurs adresses IP et noms
    ```

Accès identifiants machines

- Accès aux identifiants locaux d'une machine
- Définition dans fichier `<unistd.h>`
  - `int gethostname(char *nom, size_t lg)`
    - Récupère le (premier) nom de la machine locale, placé dans `nom`
    - `lg` est la taille de l'espace mémoire référencé par `nom`
    - Retourne 0 si appel réussi, -1 sinon
  - `long gethostid()`
    - Retourne l'adresse IP de la machine locale sous forme d'un entier sur 4 octets

Accès identifiants machines

- Accès aux identifiants de machines distantes
- Définitions dans fichier `<netdb.h>`
  - `struct hostent *gethostbyname(char *nom)`
    - Retourne l'identifiant de la machine dont le nom est passé en paramètre ou NULL si aucune machine de ce nom trouvée
    - Recherche des infos dans l'ordre suivant
      1. Serveur de nom DNS
      2. Serveur NIS
      3. Fichier local `/etc/hosts`
    - Note
      - La structure retournée est placée à une adresse statique en mémoire
      - Un nouvel appel de `gethostbyname` écrase la valeur à cette adresse
      - Nécessité de copier la structure avec un `memcopy` ou un `bcopy` si on veut la conserver

Accès identifiants machines

- Accès identifiants machines distantes (suite)
  - `struct hostent *gethostbyaddr(`
    `char *adresse,` zone mémoire contenant l'adresse de la machine,
    `int longueur,` longeur de l'adresse,
    `int type)` type de l'adresse
    - Retourne les identifiants d'une machine à partir de son adresse
      - Retourne NULL si machine non trouvée
    - Si adresse IP
      - `longueur` = 4
      - `type = AF_INET`
      - `adresse` pointe vers une entier sur 4 octets codant l'adresse IP
    - Note
      - Là aussi, retourne une référence sur une zone mémoire statique écrasée à chaque nouvel appel

Identifiants d'une socket

- Identifiants d'une socket
  - Couple adresse IP/numéro de port dans le contexte IP
- Identifiant « général » abstrait
  - `struct sockaddr`
    - On ne l'utilise jamais directement mais une spécialisation selon le type de réseau ou de communication utilisé
  - Exemples de spécialisation
    - Fichier `<netinet/in.h>`
      - `struct sockaddr_in`
      - Contexte IP
    - Fichier `<sys/un.h>`
      - `struct sockaddr_un`
      - Contexte Unix (communication locale à la même machine via des sockets)

Identifiants d'une socket

- Identifiants socket TCP/UDP – IP
  - ```
    struct sockaddr_in {
        short sin_familly;      = AF_INET,
        u_short sin_port;       port associée à la socket,
        struct in_addr sin_adr; adresse de la machine,
        char sin_zero[8];       champ de 0 pour compléter
    };
    ```
- Opérations sur les sockets
  - Fichier `<sys/socket.h>`
    - Création de sockets
    - Liaison sur un port local
    - Envoi/réception de données
    - ...
    ```

Création et gestion de sockets

- Création d'une socket
  - `int socket(int domaine, int type, int protocole)`
  - Retourne un descripteur correspondant à la socket créée
    - Similaire à un descripteur de fichier
  - Paramètres
    - Domaine : utilisation de constantes parmi
      - AF_UNIX : domaine local Unix
      - AF_INET : domaine IP
    - Type : type de la socket selon protocoles sous-jacents, constantes parmi entre autres
      - SOCK_DGRAM : socket mode datagramme, non connecté et non fiable (UDP par défaut)
      - SOCK_STREAM : socket mode connecté et fiable (TCP par défaut)
      - SOCK_RAW : socket bas niveau (IP ou ICMP)
    - Protocole
      - On utilise généralement la valeur 0 qui sélectionne le protocole par défaut associé à chaque type de socket

Création et gestion de sockets

- Création d'une socket (suite)
  - En cas d'erreur lors de la création : retourne -1
  - Pour connaître le détail de l'erreur
    - Consulter la valeur de la variable `errno` ou afficher un message avec la fonction `perror`
    - Liste des erreurs dans le fichier `<errno.h>`
  - Si la socket n'est pas liée à un port donné via un `bind`
    - Lors du premier envoi d'un paquet, elle sera liée localement à un port quelconque disponible
- Fermeture d'une socket
  - `int close(int socket_desc)`
    - Ferme la socket dont le descripteur est passé en paramètre

Création et gestion de sockets

- Liaison d'une socket sur un port particulier
- `int bind(int sock_desc,` — descripteur de la socket,
  `struct sockaddr *adresse,` — adresse sur laquelle lier la socket,
  `socklen_t longueur_adresse)` — taille de l'adresse (int)
  - Retourne -1 en cas de problème
    - Problèmes les plus courants
      - Le port choisi a déjà une socket liée dessus (erreur EADDRINUSE)
      - Liaison non autorisée (ex : port < 1024) sur ce port (erreur EACCES)
  - Déclaration typique d'une adresse
    - De type `sockaddr_in` pour IP, avec champs positionnés comme suit
      - `sin_family = AF_INET`
      - `sin_addr.s_addr = INADDR_ANY` (ou une adresse IP locale)
        - `INADDR_ANY` : permet d'utiliser n'importe quelle IP de la machine si elle en a plusieurs et/ou évite de connaître l'adresse IP locale
      - `sin_port` = port sur lequel on veut lier la socket
        - Si 0 : n'importe quel port libre

Création et gestion de sockets

- Récupérer les informations sur une socket
  - `int getsockname(`
    `int descripteur,` descripteur de la socket,
    `struct sockaddr *adresse,` contiendra l'adresse mémoire de l'adresse réseau
    `int *longueur_adresse)`
    - Paramètre `longueur_adresse`
      - A l'appel : taille de l'espace réservé pour contenir l'adresse
      - Au retour : longueur effective de l'adresse

Représentations des nombres

- Selon le système/matériel, le codage binaire des nombres peut changer
  - *Big Endian* : octet de poids fort à gauche
  - *Little Endian* : octet de poids fort à droite
- Pour éviter une mauvaise interprétation des nombres
  - On les code en « mode réseau » lorsqu'on les utilise dans les structures ou fonctions d'accès au réseau et sockets
- 4 opérations assurent la traduction « mode réseau / local »
  - `u_short htons(u_short)` : entier court local vers réseau
  - `u_long htonl(u_long)` : entier long local vers réseau
  - `u_short ntohs(u_short)` : entier court réseau vers local
  - `u_long ntohl(u_long)` : entier long réseau vers local
  - Entiers courts : numéros de port
  - Entiers longs : adresses IP

Envoi/réception données en UDP

- Envoi de données sur socket
  - `int sendto(`

`int descripteur,`	descripteur de la socket qui émettra les données,
`void *message,`	pointeur vers la zone de données à émettre,
`int longueur,`	longueur des données,
`int option,`	0,
`struct sockaddr *adresse,`	adresse de la socket destinatrice,
`int longueur_adresse)`	longueur de l'adresse

    - Globalement, deux types d'informations à passer en paramètre
      - Lien vers les données à émettre
      - L'adresse identifiant le destinataire (couple @IP/port)
    - Retourne
      - Nombre de caractères (réellement) envoyés
      - -1 en cas de problème (détails avec `perror` ou `errno`)

Envoi/réception données en UDP

- Réception de données
  - `int recvfrom(`

```
    int descripteur,
    void *message,
    int longueur,
    int option,
    struct sockaddr *adresse,
    int *longueur_adresse)
```

descripteur de la socket qui attend les données,
pointeur vers la zone de données,
taille max de la zone données,
0 ou MSG_PEEK,
adresse de la socket émettrice,
longueur de l'adresse

  - Paramètres à fournir
    - Lien et taille de la zone de données qui contiendra les données reçues
    - `longueur_adresse` : longueur de l'adresse, à initialiser à l'appel, sera modifiée (avec la même valeur …) au retour

Envoi/réception données en UDP

- Réception de données (suite)
  - `recvfrom` retourne
    - Le nombre de caractères reçus
    - -1 en cas de problème
  - En retour via les pointeurs
    - Zone de données initialisée avec les données
    - Adresse : contient l'adresse de la socket émettrice (`longueur_adresse` contient la taille de cette adresse)
  - Paramètre `option = MSG_PEEK`
    - Le paquet de données reçu n'est pas retiré du tampon
    - A la prochaine réception de données, on lira les mêmes données
  - Réception de données est bloquante par défaut
    - Sauf si des paquets avaient été reçus et pas encore lus

Envoi/réception données en UDP

- Notes sur les tailles des données envoyées/reçues
  - A priori pas limite en taille pour les données circulant dans les paquets UDP, mais
    - Pour tenir dans un seul datagramme IP, le datagramme UDP ne doit pas contenir plus de 65467 octets de données
      - Un datagramme UDP est rarement envoyé via plusieurs datagrammes IP
    - Mais en pratique : il est conseillé de ne pas dépasser 8176 octets
      - Car la plupart des systèmes limitent à 8 Ko la taille des datagrammes UDP
    - Pour être certain de ne pas perdre de données : 512 octets max
    - Si datagramme UDP trop grand : les données sont tronquées
  - Si la taille de la zone de données en réception est plus petite que les données envoyées
    - Les données reçues sont généralement tronquées

Exemple de programme UDP

- Client / serveur basique en UDP
  - Client envoie une chaîne « bonjour » au serveur et attend une réponse
  - Serveur se met en attente de réception de données et renvoie la chaîne « bien recu » à l'émetteur
- Identification du serveur
  - Lancé sur la machine scinfe122
  - Écoute sur le port 4000

Exemple UDP : coté client

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <netdb.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <errno.h>

#define TAILLEBUF 20

int main() {

// identifiant de la machine serveur
struct hostent *serveur_host;
// adresse de la socket coté serveur
static struct sockaddr_in addr_serveur;
// taille de l'addresse socket
socklen_t lg;
```

Exemple UDP : coté client (suite)

- // descripteur de la socket locale

```
int sock;
```

// chaine à envoyer

```
char *msg = "bonjour";
```

// buffer de réception

```
char buffer[TAILLEBUF];
```

// chaine reçue en réponse

```
char *reponse;
```

// nombre d'octets lus ou envoyés

```
int nb_octets;
```

// création d'une socket UDP

```
sock = socket(AF_INET, SOCK_DGRAM, 0);
if (sock == -1)    {
    perror("erreur création socket");
    exit(1);
}
```

Exemple UDP : coté client (suite)

- // récupération identifiant du serveur

```
serveur_host = gethostbyname("scinfe122");
if (serveur_host==NULL) {
    perror("erreur adresse serveur");
    exit(1);
}
```

// création adresse socket destinatrice

```
bzero(&addr_serveur, sizeof(struct sockaddr_in));
addr_serveur.sin_family = AF_INET;
addr_serveur.sin_port = htons(4000);
memcpy(&addr_serveur.sin_addr.s_addr,
  serveur_host -> h_addr, serveur_host -> h_length);
```

Exemple UDP : coté client (fin)

- // on envoie le message "bonjour" au serveur

```
lg = sizeof(struct sockaddr_in);
nb_octets = sendto(sock, msg, strlen(msg)+1, 0,
                (struct sockaddr*)&addr_serveur, lg);
if (nb_octets == -1) {
    perror("erreur envoi message");
    exit(1); }
printf("paquet envoyé, nb_octets = %d\n",nb_octets);

// on attend la réponse du serveur
nb_octets = recvfrom(sock, buffer, TAILLEBUF, 0,
               (struct sockaddr*)&addr_serveur, &lg);
if (nb_octets == -1) {
    perror("erreur réponse serveur");
    exit(1); }
reponse = (char *)malloc(nb_octets * sizeof(char));
memcpy(reponse, buffer, nb_octets);
printf("reponse recue du serveur : %s\n",reponse);

// on ferme la socket
close(sock);
}
```

Exemple UDP : coté serveur

- // adresse de la socket locale
```
static struct sockaddr_in addr_local;
```
// adresse de la socket coté serveur
```
static struct sockaddr_in addr_client;
```
// identifiant du client
```
struct hostent *host_client;
```
// taille de l'addresse socket
```
socklen_t lg;
```
// descripteur de la socket locale
```
int sock;
```
// chaine à envoyer en réponse
```
char *reponse = "bien recu";
```
// buffer de réception
```
char buffer[TAILLEBUF];
```
// chaine reçue
```
char *chaine;
```
// nombre d'octets lus ou envoyés
```
int nb_octets;
```

Exemple UDP : coté serveur (suite)

- // création de la socket

```
sock = socket(AF_INET, SOCK_DGRAM, 0);
if (sock == -1) {
   perror("erreur création socket");
   exit(1);
}
```

// liaison de la socket sur le port local 4000

```
bzero(&addr_local, sizeof(struct sockaddr_in));
addr_local.sin_family = AF_INET;
addr_local.sin_port = htons(4000);
addr_local.sin_addr.s_addr=htonl(INADDR_ANY);

if( bind(sock, (struct sockaddr*)&addr_local,
         sizeof(addr_local))== -1 ) {
      perror("erreur bind");
      exit(1);
}
```

Exemple UDP : coté serveur (suite)

- // attente de données venant d'un client

```
lg = sizeof(struct sockaddr_in);
nb_octets = recvfrom(sock, buffer, TAILLEBUF, 0,
        (struct sockaddr *)&addr_client, &lg);
if (nb_octets == -1) {
    perror("erreur réception paquet");
    exit(1);
}
```

// récupère nom de la machine émettrice des données

```
host_client = gethostbyaddr(&(addr_client.sin_addr),
                            sizeof(long), AF_INET);
if (host_client == NULL) {
    perror("erreur gethostbyaddr");
    exit(1);
}
```

Exemple UDP : coté serveur (fin)

- // affichage message reçu et coordonnées émetteur

```
chaine = (char *)malloc(nb_octets * sizeof(char));
memcpy(chaine, buffer, nb_octets);
printf("recu message %s de la part de %s
    sur le port %d\n", chaine, host_client->h_name,
    ntohs(addr_client.sin_port));
```

// envoi de la réponse à l'émetteur

```
nb_octets = sendto(sock, reponse, strlen(reponse)+1,
    0,(struct sockaddr*)&addr_client, lg);
if (nb_octets == -1) {
      perror("erreur envoi réponse");
      exit(1);
}
```

// fermeture la socket

```
close(sock);
}
```

Notes sur ces exemples UDP

- Variables de type `sockaddr_in`
  - A déclarer en `static`
- Fonctions de copies zones mémoire
  - `memcpy(void *dest, void *source, size_t lg);`
    `memmove(void *dest, void *source, size_t lg);`
    - Copie `lg` octets d'une zone mémoire source vers une zone mémoire destination
    - `memmove` : à utiliser pour des copies de zones mémoires se recouvrant partiellement
- Initialisation de zone mémoire
  - `memset(void *zone_mem, int valeur, size_t lg)`
    - Initialise chacun des `lg` premiers octets de la zone avec `valeur`
  - `bzero(void *zone_mem, size_t lg)`
    - Initialise les `lg` premiers octets de la zone à 0

Sockets UDP en Java

Sockets UDP en Java

- Sockets UDP en Java
  - Pas de différences fondamentales de fonctionnement par rapport à leur utilisation en C
- Caractéristiques identiques
  - Mode non connecté, communication par datagramme
  - Pas de gestion de la taille des données envoyées
    - Possibilité de perte de données à l'émission et réception

Sockets UDP en Java

- Java intègre nativement les fonctionnalités de communication réseau au dessus de TCP-UDP/IP
  - Package `java.net`
- Classes utilisées pour communication via UDP
  - `InetAddress` : codage des adresses IP
  - `DatagramSocket` : socket mode non connecté (UDP)
  - `DatagramPacket` : paquet de données envoyé via une socket sans connexion (UDP)

Codage adresse IP

- Classe `InetAddress`
  - Constructeurs
    - Pas de constructeurs, on passe par des méthodes statiques pour créer un objet
  - Méthodes
    - `public static InetAddress getByName(String host) throws UnknownHostException`
      - Crée un objet `InetAddress` identifiant une machine dont le nom est passé en paramètre
      - L'exception est levée si le service de nom (DNS...) du système ne trouve pas de machine du nom passé en paramètre sur le réseau
      - Si précise une adresse IP sous forme de chaîne ("192.12.23.24") au lieu de son nom, le service de nom n'est pas utilisé
        - Une autre méthode permet de préciser l'adresse IP sous forme d'un tableau de 4 octets

Codage adresse IP

- Classe `InetAddress`
  - Méthodes (suite)
    - `public static InetAddress getLocalHost() throws UnknownHostException`
      - Retourne l'adresse IP de la machine sur laquelle tourne le programme, c'est-à-dire l'adresse IP locale
    - `public String getHostName()`
      - Retourne le nom de la machine dont l'adresse est codée par l'objet InetAddress

Datagramme

- Classe `DatagramPacket`
  - Structure des données en mode datagramme
  - Constructeurs
    - `public DatagramPacket(byte[] buf, int length)`
      - Création d'un paquet pour recevoir des données (sous forme d'un tableau d'octets)
      - Les données reçues seront placées dans `buf`
      - `length` précise la taille max de données à lire
        - Ne pas préciser une taille plus grande que celle du tableau
        - En général, `length` = taille de `buf`
      - Variante du constructeur : avec un offset pour ne pas commencer au début du tableau

Datagramme

- Classe `DatagramPacket`
  - Constructeurs (suite)
    - `public DatagramPacket(byte[] buf, int length, InetAddress address, int port)`
      - Création d'un paquet pour envoyer des données (sous forme d'un tableau d'octets)
      - `buf` : contient les données à envoyer
      - `length` : longueur des données à envoyer
        - Ne pas préciser une taille supérieure à celle de `buf`
      - `address` : adresse IP de la machine destinataire des données
      - `port` : numéro de port distant (sur la machine destinataire) où envoyer les données

Datagramme

- Classe `DatagramPacket`
  - Méthodes « get »
    - `InetAddress getAddress()`
      - Si paquet à envoyer : adresse de la machine destinataire
      - Si paquet reçu : adresse de la machine qui a envoyé le paquet
    - `int getPort()`
      - Si paquet à envoyer : port destinataire sur la machine distante
      - Si paquet reçu : port utilisé par le programme distant pour envoyer le paquet
    - `byte[] getData`
      - Données contenues dans le paquet
    - `int getLength()`
      - Si paquet à envoyer : longueur des données à envoyer
      - Si paquet reçu : longueur des données reçues

Datagramme

- Classe `DatagramPacket`
  - Méthodes « set »
    - `void setAddress(InetAdress adr)`
      - Positionne l'adresse IP de la machine destinataire du paquet
    - `void setPort(int port)`
      - Positionne le port destinataire du paquet pour la machine distante
    - `void setData(byte[] data)`
      - Positionne les données à envoyer
    - `int setLength(int length)`
      - Positionne la longueur des données à envoyer

Sockets mode datagramme (UDP)

- Classe `DatagramSocket`
  - Socket en mode datagramme
  - Constructeurs
    - `public DatagramSocket() throws SocketException`
      - Crée une nouvelle socket en la liant à un port quelconque libre
      - Exception levée en cas de problème (a priori il ne doit pas y en avoir)
    - `public DatagramSocket(int port) throws SocketException`
      - Crée une nouvelle socket en la liant au port local précisé par le paramètre `port`
      - Exception levée en cas de problème : notamment quand le port est déjà occupé

Sockets mode datagramme (UDP)

- Classe `DatagramSocket`
  - Méthodes d'émission/réception de paquet
    - `public void send(DatagramPacket p) throws IOException`
      - Envoie le paquet passé en paramètre. Le destinataire est identifié par le couple @IP/port précisé dans le paquet
      - Exception levée en cas de problème d'entrée/sortie
    - `public void receive(DatagramPacket p) throws IOException`
      - Reçoit un paquet de données
      - Bloquant tant qu'un paquet n'est pas reçu
      - Quand paquet arrive, les attributs de `p` sont modifiés
        - Les données reçues sont copiées dans le tableau passé en paramètre lors de la création de p et sa longueur est positionnée avec la taille des données reçues
        - Les attributs d'@IP et de port de p contiennent l'@IP et le port de la socket distante qui a émis le paquet

Sockets mode datagramme (UDP)

- Classe `DatagramSocket`
  - Autres méthodes
    - `public void close()`
      - Ferme la socket et libère le port à laquelle elle était liée
    - `public int getLocalPort()`
      - Retourne le port local sur lequel est liée la socket
  - Possibilité de créer un canal (mais toujours en mode non connecté)
    - Pour restreindre la communication avec un seul destinataire distant
    - Car par défaut peut recevoir sur la socket des paquets venant de n'importe où

Sockets mode datagramme (UDP)

- Classe `DatagramSocket`
  - Réception de données : via méthode `receive`
    - Méthode bloquante sans contrainte de temps : peut rester en attente indéfiniment si aucun paquet n'est jamais reçu
  - Possibilité de préciser un délai maximum d'attente
    - `public void setSoTimeout(int timeout) throws SocketException`
    - L'appel de la méthode `receive` sera bloquante pendant au plus `timeout` millisecondes
    - Une méthode `receive` se terminera alors de 2 façons
      - Elle retourne normalement si un paquet est reçu en moins du temps positionné par l'appel de `setSoTimeout`
      - L'exception `SocketTimeoutException` est levée pour indiquer que le délai s'est écoulé avant qu'un paquet ne soit reçu
        - `SocketTimeoutException` est une sous-classe de `IOException`

Sockets UDP Java – exemple coté client

```
InetAddress adr;
DatagramPacket packet;
DatagramSocket socket;

// adr contient l'@IP de la partie serveur
adr = InetAddress.getByName("scinfr222");

// données à envoyer : chaîne de caractères
byte[] data = (new String("youpi")).getBytes();

// création du paquet avec les données et en précisant l'adresse du serveur
// (@IP et port sur lequel il écoute : 7777)
packet = new DatagramPacket(data, data.length, adr, 7777);

// création d'une socket, sans la lier à un port particulier
socket = new DatagramSocket();

// envoi du paquet via la socket
socket.send(packet);
```

Sockets UDP Java – exemple coté serveur

```
DatagramSocket socket;
DatagramPacket packet;

// création d'une socket liée au port 7777
DatagramSocket socket = new DatagramSocket(7777);

// tableau de 15 octets qui contiendra les données reçues
byte[] data = new byte[15];

// création d'un paquet en utilisant le tableau d'octets
packet = new DatagramPacket(data, data.length);

// attente de la réception d'un paquet. Le paquet reçu est placé dans
// packet et ses données dans data.
socket.receive(packet);

// récupération et affichage des données (une chaîne de caractères)
String chaine = new String(packet.getData(), 0,
                           packet.getLength());
System.out.println(" recu : "+chaine);
```

Sockets UDP en Java – exemple suite

- La communication se fait souvent dans les 2 sens
  - Le serveur doit donc connaître la localisation du client
  - Elle est précisée dans le paquet qu'il reçoit du client
- Réponse au client, coté serveur
  - ```
    System.out.println(" ca vient de : "+
    packet.getAddress()+":"+ packet.getPort());

    // on met une nouvelle donnée dans le paquet
    // (qui contient donc le couple @IP/port de la socket coté client)
    String reponse = "bien recu";
    packet.setData(reponse.getBytes());
    packet.setLength(reponse.length());

    // on envoie le paquet au client
    socket.send(packet);
    ```
    ```

Sockets UDP en Java – exemple suite

- Réception réponse du serveur, coté client

```
// attente paquet envoyé sur la socket du client
socket.receive(packet);

// récupération et affichage de la donnée contenue dans le paquet
String chaine = new String(packet.getData(), 0,
                           packet.getLength());
System.out.println(" recu du serveur : "+chaine);
```

Critique sockets UDP

- Avantages
  - Simple à programmer (et à appréhender)
- Inconvénients
  - Pas fiable
  - Ne permet d'envoyer que des tableaux de byte
    - Si en C cela convient parfaitement au niveau de la manipulation de données, en Java, c'est de l'information de bas niveau non naturelle
    - En Java, il faut pouvoir envoyer des objets quelconques via des sockets

Structure des données échangées

- Format des données à transmettre
  - Très limité a priori : tableaux de byte
    - Et attention à la taille réservée : si le récepteur réserve un tableau trop petit par rapport à celui envoyé, une partie des données est perdue
  - Doit donc pouvoir convertir
    - Un objet quelconque en byte[] pour l'envoyer
    - Un byte[] en un objet d'un certain type après réception
  - Deux solutions
    - Dans chaque classe à transmettre : rajouter des méthodes qui font la conversion
      - Lourd et dommage de faire des tâches de si « bas-niveau » avec un langage évolué comme Java
    - Utiliser les flux Java pour conversion automatique

Conversion Object <-> byte[]

- Pour émettre et recevoir n'importe quel objet via des sockets UDP
  - En écriture : conversion de Object en byte[ ]

```
public byte[] fromObjectToByte(Object obj) {
  ByteArrayOutputStream byteStream =
                          new ByteArrayOutputStream();
  ObjectOutputStream objectStream =
              new ObjectOutputStream(byteStream);
  objectStream.writeObject(obj);
  return byteStream.toByteArray();
}
```

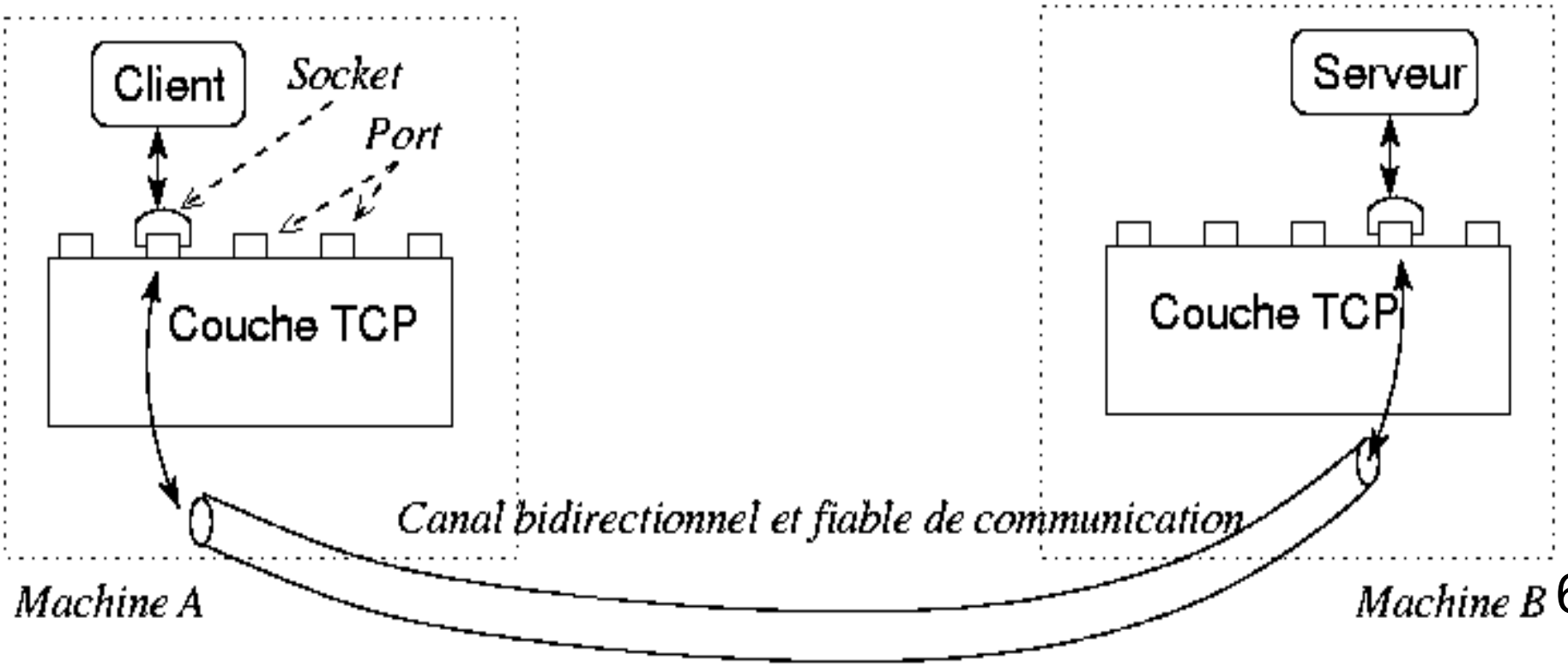
  - En lecture : conversion de byte[ ] en Object

```
public Object fromByteToObject(byte[] byteArray) {
  ByteArrayInputStream byteStream =
              new ByteArrayInputStream(byteArray);
  ObjectInputStream objectStream =
              new ObjectInputStream(byteStream);
  return objectStream.readObject();
}
```

Sockets TCP

Sockets TCP : principe

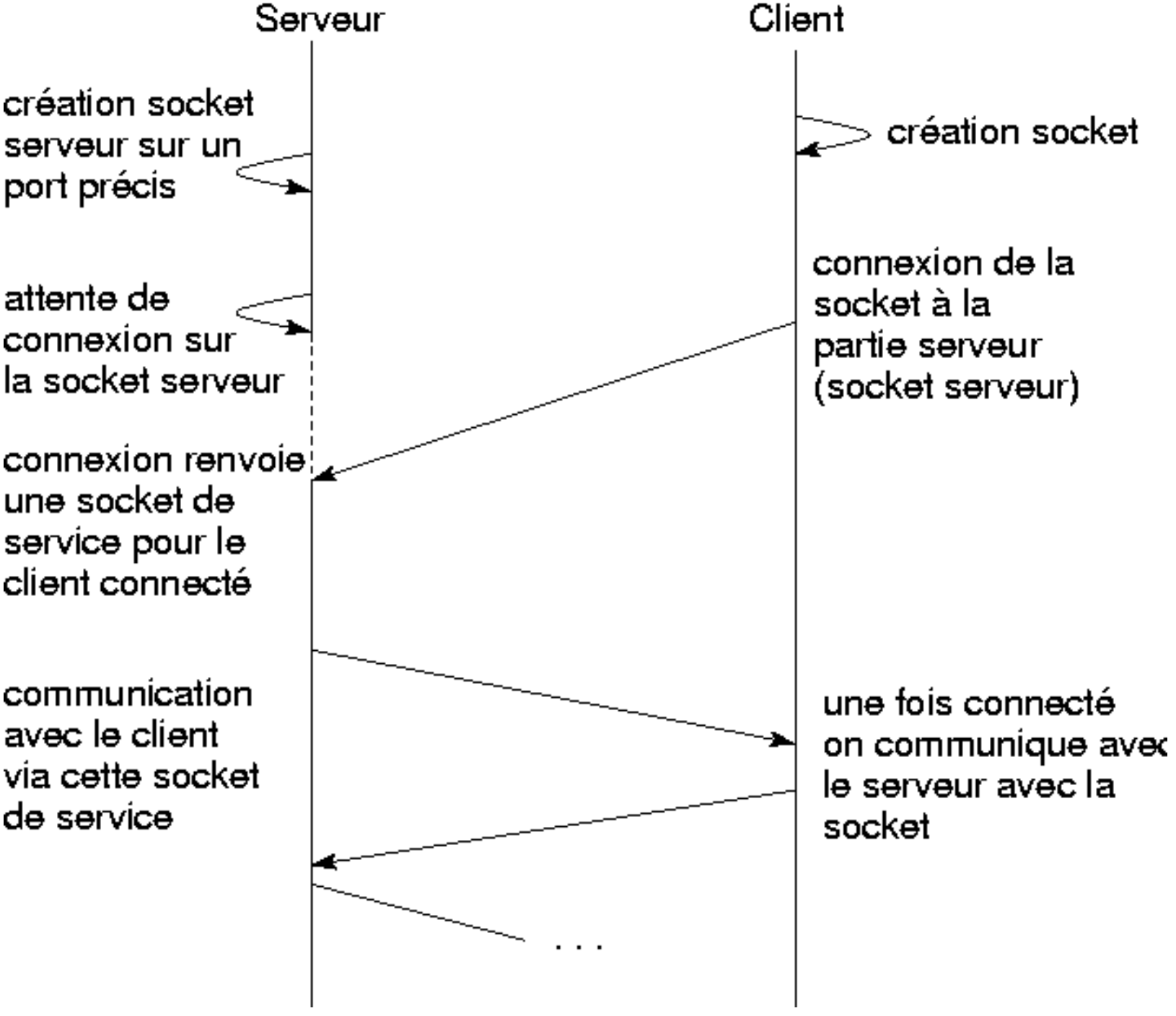
- Fonctionnement en mode connecté
  - Phase de connexion explicite entre client et serveur avant comm.
  - Données envoyées dans un « tuyau » et non pas par paquets
    - Flux (virtuels) de données
  - Fiable : la couche TCP assure que
    - Les données envoyées sont toutes reçues par la machine destinataire
    - Les données sont reçues dans l'ordre où elles ont été envoyées



Sockets TCP : principe

- Principe de communication
  - Le serveur lie une socket dite d'écoute sur un certain port bien précis et appelle un service d'attente de connexion de la part d'un client
  - Le client crée une socket liée à un port quelconque puis appelle un service pour ouvrir une connexion avec le serveur sur sa socket d'écoute
  - Du coté du serveur, le service d'attente de connexion retourne une socket de service (associée à un port quelconque)
    - C'est la socket qui permet de dialoguer avec ce client
    - Il y a une socket de service par client connecté
  - Comme avec sockets UDP : le client et le serveur communiquent en envoyant et recevant des données via leur socket

Sockets TCP : résumé communication



Sockets TCP en C

Socket TCP : mise en oeuvre en C

- Utilise les mêmes bases que pour UDP
  - Codage des différents identifiants
    - Structures `sockaddr_in`, `hostent`, `in_addr` et fonctions associées
  - Gestion des sockets : `socket()` et descripteur de fichier
- Besoin supplémentaire par rapport à UDP
  - Fonctions pour attendre des connexions côté serveur
  - Fonction pour se connecter à la socket d'écoute du serveur coté client
- Communication entre client/serveur après phase de connexion
  - Pas besoin de préciser identifiants du coté client à chaque envoi de données car on fonctionne en mode connecté
    - Pas d'usage de `recvfrom()` et `sendto()`
  - Utilise alors les fonctions système `read()` et `write()`

Sockets TCP : attente connexion

- Coté serveur
  - Fonction qui attend la connexion d'un client sur une socket d'écoute
    - Fichier `<sys/socket.h>`
    - ```
      int accept(int socket_ecoute,
                 struct sockaddr *addr_client,
                 int *lg_addr);
      ```
      - Paramètres
        - `socket_ecoute` : la socket d'écoute (à lier sur un port précis)
        - `addr_client` : contiendra l'adresse du client qui se connectera
        - `lg_addr` : contiendra la taille de la structure d'adresse
          - A initialiser avant d'appeler la fonction comme pour `recvfrom`
      - Retourne un descripteur de socket
        - La socket de service qui permet de communiquer avec le client qui vient de se connecter
        - Retourne -1 en cas d'erreur
      - Fonction bloquante jusqu'à l'arrivée d'une demande de connexion
      ```

Sockets TCP : init. connexion

- Coté serveur
  - Fonction pour configurer le nombre maximum de connexions pendantes
    - Nombre maximum de connexions en attente d'un `accept` à un instant donné (une fois atteint, les suivantes seront rejetées)
    - ```
      int listen(int descripteur,
                 int nb_con_pendantes);
      ```
    - Paramètres
      - `descripteur` : descripteur de la socket d'écoute à configurer
      - `nb_con_pendantes` : nombre max de connexions pendantes autorisées
    - Retourne 0 si exécution correcte, -1 en cas de problème
    - Le nombre de connexion pendantes précisé doit être inférieur à la valeur de la constante `SOMAXCONN` (fichier `<sys/socket.h>`)
      - Cette valeur dépend des systèmes d'exploitation
      - SOMAXCONN = 128 sous Linux
    - Fonction à appeler avant de faire les `accept`
      ```

Sockets TCP : ouverture connexion

- Coté client
  - Fonction pour ouvrir une connexion avec la partie serveur
    - `int connect(int descripteur, struct sockaddr *addr_serveur, int lg_addr);`
    - Paramètres
      - `descripteur` : descripteur de la socket coté client
      - `addr_serveur` : identifiant de la socket d'écoute coté serveur
      - `lg_adresse` : taille de l'adresse utilisée
    - Retourne 0 si tout se passe bien, -1 sinon
    - Fonction bloquante
      - Si le serveur ne fait pas l'`accept` de son coté

Sockets TCP : envoi/réception données

- Si connexion établie
  - « tuyau » de communication directe entre socket coté client et socket de service coté serveur
  - Pas besoin de préciser l'adresse du destinataire à chaque envoi de données ni de vérifier l'émetteur à la réception
- Fonctions de communication
  - On peut utiliser les fonctions standards pour communiquer
    - `write` pour émission
    - `read` pour réception
  - Si on veut gérer quelques options, fonctions spécialisées
    - `send` pour émission
    - `recv` pour réception

Sockets TCP : émission

- 2 fonctions pour l'émission de données
  - `ssize_t write(int descripteur, void *ptr_mem, size_t longueur);`
  - `ssize_t send(int descripteur, void *ptr_mem, size_t longueur, int option);`
  - Paramètres
    - `descripteur` : descripteur de la socket
    - `ptr_mem` : zone mémoire où sont les données à envoyer
    - `longeur` : nombre d'octets à envoyer
    - `option` : 0 si envoi normal ou `MSG_OOB` si envoi de données prioritaires
  - Retourne le nombre d'octets écrits ou -1 en cas de pb

Sockets TCP : réception

- 2 fonctions (bloquantes) pour réception de données
  - `ssize_t read(int descripteur, void *ptr_mem, size_t longueur);`
  - `ssize_t recv(int descripteur, void *ptr_mem, size_t longueur, int option);`
  - Paramètres
    - `descripteur` : descripteur de la socket
    - `ptr_mem` : zone mémoire où seront écrites les données reçues
    - `longueur` : taille de la zone mémoire
    - `option` : 3 valeurs possibles que l'on peut combiner (avec des ||)
      - 0 : réception normale
      - `MSG_OOB` : réception de données prioritaires
      - `MSG_PEEK` : lecture des données reçues mais sans les retirer du tampon
  - Retourne le nombre d'octets reçus ou -1 en cas de pb

Sockets TCP : exemple

- Même exemple que pour UDP
  - Serveur attend la connexion du client
  - Client envoie une chaîne au serveur
  - Serveur lui répond en lui renvoyant une chaîne
- Identification du serveur
  - Lancé sur la machine scinfe122
  - Écoute sur le port 4000

Exemple TCP : coté client

- // identification socket d'écoute du serveur

```
static struct sockaddr_in addr_serveur;
```

// identifiants de la machine où tourne le serveur

```
struct hostent *host_serveur;
```

// socket locale coté client

```
int sock;
```

// message à envoyer au serveur

```
char *message = "bonjour";
```

// chaîne où sera écrit le message reçu

```
char reponse[TAILLEBUF];
```

// nombre d'octets envoyés/reçus

```
int nb_octets;
```

// création socket TCP

```
sock = socket(AF_INET, SOCK_STREAM, 0);
if (sock == -1) {
    perror("creation socket");
    exit(1); }
```

Exemple TCP : coté client (suite)

- // récupération identifiants de la machine serveur

```
host_serveur = gethostbyname("scinfe122");
if (host_serveur==NULL)  {
    perror("erreur récupération adresse serveur\n");
    exit(1);
}
```

// création de l'identifiant de la socket d'écoute du serveur

```
bzero((char *) &addr_serveur,
sizeof(addr_serveur));
addr_serveur.sin_family = AF_INET;
addr_serveur.sin_port = htons(4000);
memcpy(&addr_serveur.sin_addr.s_addr,
    host_serveur->h_addr, host_serveur->h_length);
```

Exemple TCP : coté client (fin)

- // connexion de la socket client locale à la socket coté serveur

```
if (connect(sock,
            (struct sockaddr *)&addr_serveur,
            sizeof(struct sockaddr_in)) == -1) {
    perror("erreur connexion serveur");
    exit(1);
}
```

// connexion etablie, on envoie le message

```
nb_octets = write(sock, message, strlen(message)+1);
```

// on attend la réponse du serveur

```
nb_octets = read(sock, reponse, TAILLEBUF);
printf(" reponse recue : %s\n", reponse);
```

// on ferme la socket

```
close(sock);
```

Exemple TCP : coté serveur

- // adresse socket coté client
```
static struct sockaddr_in addr_client;
```
// adresse socket locale
```
static struct sockaddr_in addr_serveur;
```
// longueur adresse
```
int lg_addr;
```
// socket d'écoute et de service
```
int socket_ecoute, socket_service;
```
// buffer qui contiendra le message reçu
```
char message[TAILLEBUF];
```
// chaîne reçue du client
```
char *chaine_recue;
```
// chaîne renvoyée au client
```
char *reponse = "bien recu";
```
// nombre d'octets reçus ou envoyés
```
int nb_octets;
```

Exemple TCP : coté serveur (suite)

- // création socket TCP d'écoute

```
socket_ecoute = socket(AF_INET, SOCK_STREAM, 0);
if (socket_ecoute == -1) {
    perror("creation socket");
    exit(1); }
```

// liaison de la socket d'écoute sur le port 4000

```
bzero((char *) &addr_serveur, sizeof(addr_serveur));
addr_serveur.sin_family = AF_INET;
addr_serveur.sin_port = htons(4000);
addr_serveur.sin_addr.s_addr=htonl(INADDR_ANY);
if( bind(socket_ecoute,
         (struct sockaddr*)&addr_serveur,
         sizeof(addr_serveur))== -1 ) {
    perror("erreur bind socket écoute");
    exit(1);
}
```

Exemple TCP : coté serveur (suite)

- // configuration socket écoute : 5 connexions max en attente

```
if (listen(socket_ecoute, 5) == -1) {
    perror("erreur listen");
    exit(1);
}
```

// on attend la connexion du client

```
lg_addr = sizeof(struct sockaddr_in);
socket_service = accept(socket_ecoute,
        (struct sockaddr *)&addr_client,
&lg_addr);
if (socket_service == -1) {
    perror("erreur accept");
    exit(1);
}
```

Exemple TCP : coté serveur (fin)

- // la connexion est établie, on attend les données envoyées par le client

```
nb_octets = read(socket_service, message,
TAILLEBUF);
// affichage du message reçu
chaine_recue =
        (char *)malloc(nb_octets * sizeof(char));
memcpy(chaine_recue, message, nb_octets);
printf("recu message %s\n", chaine_recue);

// on envoie la réponse au client
write(socket_service, reponse, strlen(reponse)+1);

// on ferme les sockets
close(socket_service);
close(socket_ecoute);
```

Sockets TCP : gestion données

- En UDP : mode datagramme
  - Un paquet envoyé = un paquet reçu
  - Avec paquets tronqués si taille émission ou réception pas adaptée
- En TCP : mode connecté, communication par flux
  - Un bloc de données envoyé n'est pas forcément reçu en un seul bloc d'un coup
  - La couche TCP peut
    - Découper un bloc émis et le délivrer en plusieurs blocs
      - Plusieurs `read` renverront les données d'un seul `write`
    - Concaténer plusieurs blocs émis en un seul bloc reçu
      - Un `read` renverra les données de plusieurs `write`
  - Toujours vérifier la taille des données reçues, notamment lorsque l'on envoie des structures de données

Pseudo-connexion en UDP

- Peut utiliser la primitive `connect` en UDP
  - Réalise une pseudo connexion
    - La socket locale est configurée alors pour n'envoyer des paquets qu'à l'adresse de la socket passée en paramètre du `connect`
    - Peut alors utiliser les services `send/write` et `read/recv` à la place de `sendto` et `recvfrom`
  - Attention
    - Pas d'établissement de vraie connexion : juste une mémorisation de l'adresse destinataire pour simplifier émission
      - Si destinataire pas prêt à recevoir les données, elles seront perdues
      - N'offre donc pas non plus une communication fiable à la TCP
    - Une fois connectée, une socket UDP ne peut plus émettre des paquets vers une autre adresse que celle de la pseudo connexion

Pseudo-connexion en UDP

- Exemple de pseudo-connexion
  - La socket locale n'est utilisée que pour envoyer des données au port 4000 de la machine scinfe122
  - ...

```
sock = socket(AF_INET, SOCK_DGRAM, 0);
host = gethostbyname("scinfe122");
bzero((char *) &adresse, sizeof(adresse));
adresse.sin_family = AF_INET;
adresse.sin_port = htons(4000);
memcpy(&adresse.sin_addr.s_addr, host -> h_addr,
       host -> h_length);
// on fait la connexion
connect(sock, (struct sockaddr*)&adresse, lg);
// on peut utiliser alors la fonction write pour émission
write(sock, msg, lg_msg);
...
```

Informations sur la socket distante

- En TCP ou pseudo-connexion pour UDP
  - Possibilité de connaître les identifiants de la socket distante avec qui la socket locale est connectée
  - `int getpeername(int sock,`
    `(struct sockaddr *) adresse,`
    `socklen_t *lg_adresse);`
  - Paramètres
    - `sock` : descripteur de la socket locale
    - `adresse` : contiendra l'adresse de la socket distante
    - `lg_adresse` : contiendra la taille de l'adresse
      - A initialiser avant d'appeler la fonction

Sockets TCP en Java

Sockets TCP en Java

- Respecte le fonctionnement de base des sockets TCP, comme en C
  - Mode connecté
    - Connexion explicite du client au serveur
  - Communication fiable, pas de perte de données
- Particularité par rapport au sockets TCP/UDP en C et sockets UDP en Java
  - Les données échangées ne sont plus des tableaux d'octets
  - On utilise les flux Java
    - Chaque socket possède un flux d'entrée et un flux de sortie
    - Communication de haut niveau permettant d'envoyer facilement n'importe quel objet ou donnée via des sockets TCP

Sockets TCP en Java

- Classes du package `java.net` utilisées pour communication via TCP
  - `InetAddress` : codage des adresses IP
    - Même classe que celle décrite dans la partie UDP et usage identique
  - `Socket` : socket mode connecté
  - `ServerSocket` : socket d'attente de connexion du coté server

Socket en mode connecté

- Classe Socket
  - Socket mode connecté
  - Constructeurs
    - `public Socket(InetAddress address, int port) throws IOException`
      - Crée une socket locale et la connecte à un port distant d'une machine distante identifié par le couple `address/port`
      - Pas de service dédié de connexion, on se connecte à la partie serveur lors de l'instanciation de la socket
    - `public Socket(String address, int port) throws IOException,UnknownHostException`
      - Idem mais avec nom de la machine au lieu de son adresse IP codée
      - Lève l'exception `UnknownHostException` si le service de nom ne parvient pas à identifier la machine
    - Variante de ces 2 constructeurs pour préciser en plus un port local sur lequel sera liée la socket créée

Socket en mode connecté

- Classe Socket
  - Méthodes d'émission/réception de données
    - Contrairement aux sockets UDP, les sockets TCP n'offrent pas directement de services pour émettre/recevoir des données
    - On récupère les flux d'entrée/sorties associés à la socket
      - `OutputStream getOutputStream()`
        - Retourne le flux de sortie permettant d'envoyer des données via la socket
      - `InputStream getInputStream()`
        - Retourne le flux d'entrée permettant de recevoir des données via la socket
  - Fermeture d'une socket
    - `public close()`
      - Ferme la socket et rompt la connexion avec la machine distante

Socket en mode connecté

- Classe Socket
  - Méthodes « get »
    - `int getPort()`
      - Renvoie le port distant avec lequel est connecté la socket
    - `InetAddress getAddress()`
      - Renvoie l'adresse IP de la machine distante
    - `int getLocalPort()`
      - Renvoie le port local sur lequel est liée la socket
  - `public void setSoTimeout(int timeout) throws SocketException`
    - Positionne l'attente maximale en réception de données sur le flux d'entrée de la socket
      - Si temps dépassé lors d'une lecture : exception `SocketTimeoutException` est levée
      - Par défaut : temps infini en lecture sur le flux

Socket serveur

- Classe ServerSocket
  - Socket d'attente de connexion, coté serveur uniquement
  - Constructeurs
    - `public ServerSocket(int port) throws IOException`
      - Crée une socket d'écoute (d'attente de connexion de la part de clients)
      - La socket est liée au port dont le numéro est passé en paramètre
        - L'exception est levée notamment si ce port est déjà lié à une socket
  - Méthodes
    - `Socket accept() throws IOException`
      - Attente de connexion d'un client distant
      - Quand connexion est faite, retourne une socket permettant de communiquer avec le client : socket de service
    - `void setSoTimeout(int timeout) throws SocketException`
      - Positionne le temps maximum d'attente de connexion sur un accept
        - Si temps écoulé, l'accept lève l'exception `SocketTimeoutException`
      - Par défaut, attente infinie sur l'accept

Sockets TCP Java – exemple coté client

- Même exemple qu'avec UDP
  - Connexion d'un client à un serveur
  - Envoi d'une chaîne par le client et réponse sous forme d'une chaîne par le serveur
- Coté client

```
// adresse IP du serveur
InetAddress adr = InetAddress.getByName("scinfr222");

// ouverture de connexion avec le serveur sur le port 7777
Socket socket = new Socket(adr, 7777);
```

Sockets TCP Java – exemple coté client

- Coté client (suite)

```
// construction de flux objets à partir des flux de la socket
ObjectOutputStream output =
     new ObjectOutputStream(socket.getOutputStream());
ObjectInputStream input =
     new ObjectInputStream(socket.getInputStream());

// écriture d'une chaîne dans le flux de sortie : c'est-à-dire envoi de
// données au serveur
output.writeObject(new String("youpi"));

// attente de réception de données venant du serveur (avec le readObject)
// on sait qu'on attend une chaîne, on peut donc faire un cast directement
String chaine = (String)input.readObject();
System.out.println(" recu du serveur : "+chaine);
```

Sockets TCP Java – exemple coté serveur

- // serveur positionne sa socket d'écoute sur le port local 7777

```
ServerSocket serverSocket = new ServerSocket(7777);
```

// se met en attente de connexion de la part d'un client distant

```
Socket socket = serverSocket.accept();
```

// connexion acceptée : récupère les flux objets pour communiquer
// avec le client qui vient de se connecter

```
ObjectOutputStream output =
     new ObjectOutputStream(socket.getOutputStream());
ObjectInputStream input =
     new ObjectInputStream(socket.getInputStream());
```

// attente les données venant du client

```
String chaine = (String)input.readObject();
System.out.println(" recu : "+chaine);
```

Sockets TCP Java – exemple coté serveur

- Coté serveur (suite)

```
// affiche les coordonnées du client qui vient de se connecter
System.out.println(" ca vient de : "
    +socket.getInetAddress()+":"+socket.getPort());

// envoi d'une réponse au client
output.writeObject(new String("bien recu"));
```

- Quand manipule des flux d'objets
  - Souvent utile de vérifier le type de l'objet reçu pour faire un cast ensuite
    - Utilise `instanceof`
  - Exemple
    -
```
String chaine; Personne pers;
Object obj = input.readObject();
if (obj instanceof String) chaine = (String)obj;
if (obj instanceof Personne) pers = (Personne)obj;
```

Sockets TCP

- Critique sockets TCP
  - Avantages
    - Niveau d'abstraction plus élevé qu'avec UDP
      - Mode connecté avec phase de connexion explicite
      - Flux d'entrée/sortie avec la mise en œuvre Java
    - Fiable
  - Inconvénients
    - Plus difficile de gérer plusieurs clients en même temps
      - Nécessite du parallélisme avec des threads/processus
      - Mais oblige une bonne structuration coté serveur

Sockets UDP ou TCP ?

- Choix entre UDP et TCP
  - A priori simple
    - TCP est fiable et mieux structuré
  - Mais intérêt tout de même pour UDP dans certains cas
    - Si la fiabilité n'est pas essentielle
    - Si la connexion entre les 2 applications n'est pas utile voire est même contraignante
    - Exemple
      - Un thermomètre envoie toutes les 5 secondes la température de l'air ambiant à un afficheur distant
      - Pas grave de perdre une mesure de temps en temps
      - Le mode connexion oblige pour le thermomètre à tenter régulièrement de rouvrir une connexion si l'afficheur est planté : code plus complexe
        - Alors qu'en UDP, il suffit de relancer l'afficheur avec le thermomètre qui envoie ses données quoiqu'il arrive, que l'afficheur soit fonctionnel ou pas

Sockets UDP ou TCP ?

- Exemple de protocole utilisant UDP : NFS
  - Network File System (NFS)
    - Accès à un système de fichiers distant
  - A priori TCP mieux adapté car besoin de fiabilité lors des transferts des fichiers, mais
    - NFS est généralement utilisé au sein d'un réseau local
      - Peu de pertes de paquets
    - UDP est plus basique et donc plus rapide
      - TCP gère un protocole assurant dans n'importe quel contexte la fiabilité, ce qui implique de nombreux échanges supplémentaires entre les applications (envoi de messages de contrôle, d'acquittement...)
      - Peu de perte de paquet en UDP en local : peut directement gérer la fiabilité au niveau NFS ou applicatif et c'est moins coûteux en temps
    - Dans ce contexte, il n'est pas pénalisant d'utiliser UDP au lieu de TCP pour NFS
      - NFS fonctionne sur ces 2 couches

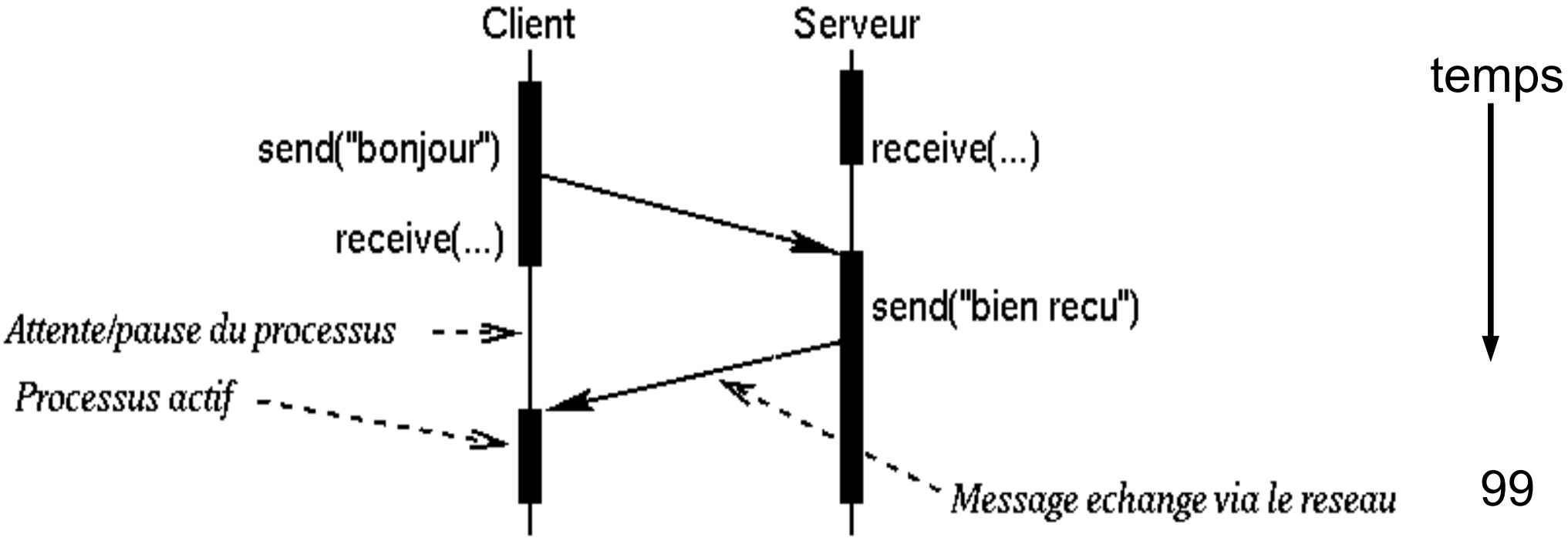
Gestion multi-clients

Application multi-clients

- Application client/serveur classique
  - Un serveur
  - Plusieurs clients
  - Le serveur doit pouvoir répondre aux requêtes des clients sans contrainte sur l'ordre d'arrivée des requêtes
- Contraintes à prendre à compte
  - Chaque élément (client ou serveur) s'exécute indépendamment des autres et en parallèle des autres

Concurrence

- Par principe, les éléments distants communicants sont actifs en parallèle
  - Plusieurs processus concurrents
  - Avec processus en pause lors d'attente de messages
- Exemple de flux d'exécution pour notre exemple de client/serveur précédent



Gestion plusieurs clients

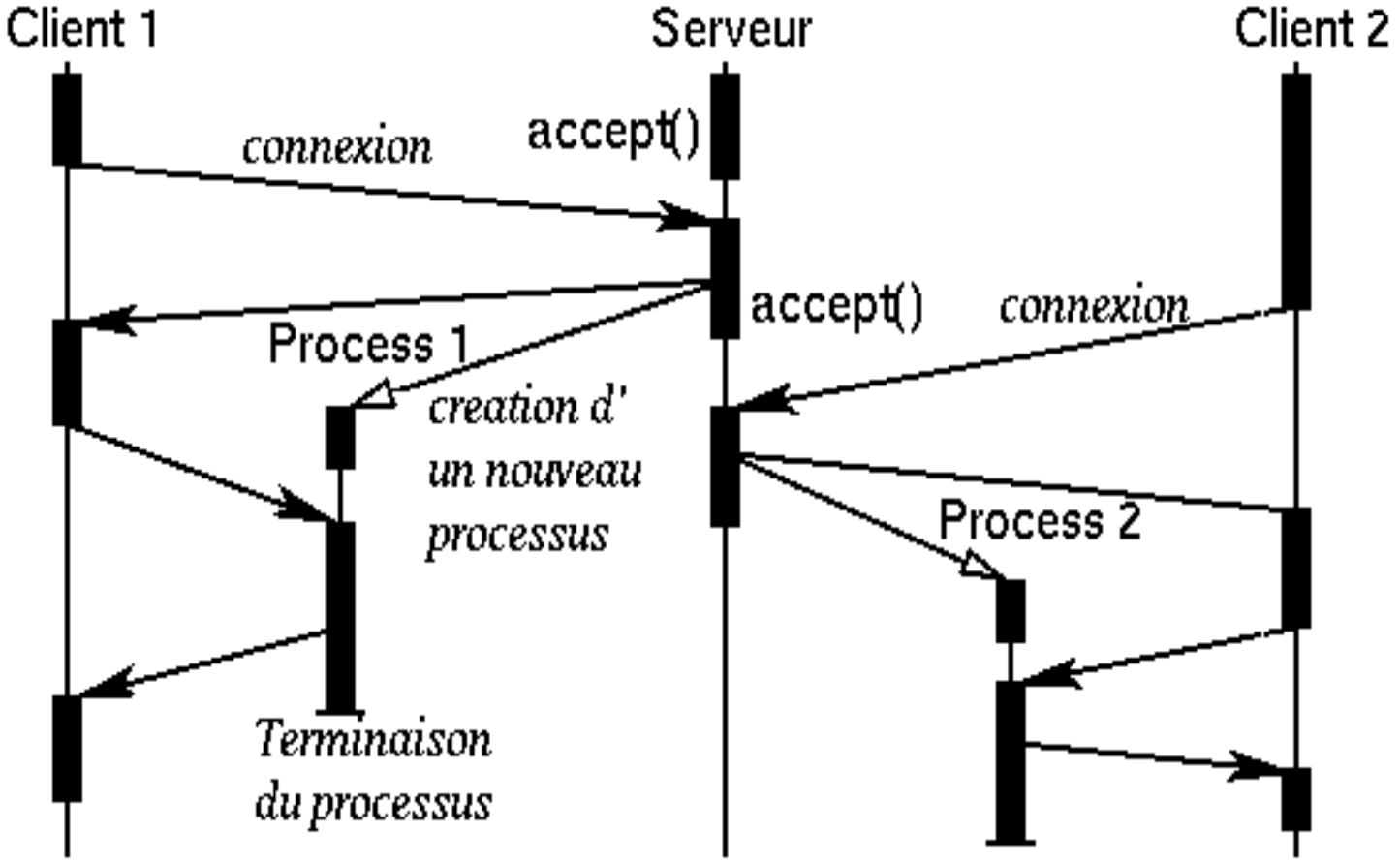
- Particularité coté serveur en TCP
  - Une socket d'écoute sert à attendre les connexions des clients
  - A la connexion d'un client, une socket de service est initialisée pour communiquer avec ce client
- Communication avec plusieurs clients pour le serveur
  - Envoi de données à un client
    - UDP : on précise l'adresse du client dans le paquet à envoyer
    - TCP : on utilise la socket correspondant au client
  - Réception de données venant d'un client quelconque
    - UDP : se met en attente d'un paquet et regarde de qui il vient
    - TCP : doit se mettre en attente de données sur toutes les sockets actives

Sockets TCP – gestion plusieurs clients

- Fonctionnement de TCP impose des contraintes
  - Lecture sur une socket : opération bloquante
    - Tant que des données ne sont pas reçues
  - Attente de connexion : opération bloquante
    - Jusqu'à la prochaine connexion d'un client distant
- Avec un seul flot d'exécution (processus/thread)
  - Si ne sait pas quel est l'ordonnancement des arrivées des données des clients ou de leur connexion au serveur
    - Impossible à gérer
- Donc nécessité de plusieurs processus ou threads
  - Un processus en attente de connexion sur le port d'écoute
  - Nouvelle connexion : un nouveau processus est créé pour gérer la communication avec le nouveau client

Sockets TCP – gestion plusieurs clients

- Boucle de fonctionnement général d'un serveur pour gérer plusieurs clients
  - while(true) {
    socketClient = acceptConnection();
    newProcessus(socketClient); }
  - Exemple avec 2 clients →



C – gestion plusieurs clients TCP

- Création d'un nouveau processus via un `fork()` à chaque connexion de client
  - Le processus principal fait les `accept()` en boucle et crée un nouveau processus fils pour la communication avec un nouveau client connecté
  -
```
while(1) {
    socket_service = accept(socket_ecoute,
          (struct sockaddr *)&addr_client,&lg_addr);
    if (fork() == 0) {
        // on est dans le fils
        close(socket_ecoute);
        // fonction qui gère la communication avec le client
        traiter_communication(socket_service);
        exit(0);
    }
    close(socket_service);
}
```

C – gestion des sockets

- Le fork() duplique toutes les données du père au fils créé
  - Y compris sa table des descripteurs de fichier
  - Une socket est un descripteur de fichier
- Le fils ferme la socket d'écoute
  - Par principe, il n'en a pas besoin, ce n'est pas son rôle de gérer les demandes de connexion
  - Ne ferme pas physiquement la socket d'écoute car le père a toujours son descripteur ouvert
- Le père ferme la socket de service qui vient d'être créée
  - Là aussi, par principe, le père n'en a pas besoin
  - Mais c'est surtout indispensable parce qu'au fil du temps sa table des descripteurs de fichier se remplit avec les connexions et une fois pleine, plus aucune connexion ne peut être acceptée

C – gestion processus

- Serveur multi-processus précédent
  - Les processus fils deviennent des zombis une fois terminés
  - Pour supprimer ce problème, exécuter avant la boucle
    - `signal(SIGCHLD, SIG_IGN);`
  - Peut aussi associer une fonction handler à SIGCHLD
    - Mais en général on ne fait rien à la terminaison du fils
- Pour que le serveur devienne un démon
  - Au départ du lancement du serveur, on crée un fils qui exécute un `setsid()` et on termine le processus principal
    - Le code du serveur (la boucle principale) est exécutée via ce fils

```
...
if ( fork() != 0 ) exit(0);
setsid();
...
while(1) {
     socket_service = accept(...)
     ...
}
```

Java – gestion plusieurs clients TCP

- Coté serveur
  - Crée un thread Java dédié à la communication avec le client qui vient de se connecter
  - Cf TD/TP

Multicast UDP/IP

Multicast

- On a vu comment faire communiquer des applications 1 à 1 via des sockets UDP ou TCP
- UDP offre un autre mode de communication : multicast
  - Plusieurs récepteurs pour une seule émission d'un paquet
- Broadcast, multicast
  - Broadcast (diffusion) : envoi de données à tous les éléments d'un réseau
  - Multicast : envoi de données à un sous-groupe de tous les éléments d'un réseau
- Multicast IP
  - Envoi d'un datagramme sur une adresse IP particulière
  - Plusieurs éléments lisent à cette adresse IP

Multicast

- Adresse IP multicast
  - Classe d'adresse IP entre 224.0.0.0 et 239.255.255.255
    - Classe D
    - Adresses entre 225.0.0.0 et 238.255.255.255 sont utilisables par un programme quelconque
      - Les autres sont réservées
  - Une adresse IP multicast n'identifie pas une machine sur un réseau mais un *groupe multicast*
- Socket UDP multicast
  - Avant envoi de paquet : on doit rejoindre un groupe
    - Identifié par un couple : @IP multicast/numéro port
  - Un paquet envoyé par un membre du groupe est reçu par tous les membres de ce groupe

Multicast

- Utilités du multicast UDP/IP
  - Évite d'avoir à créer X connexions et/ou d'envoyer X fois la même donnée à X machines différentes
  - En pratique
    - Utilisé pour diffuser des informations
    - Diffusion de flux vidéos à plusieurs récepteurs
      - Chaîne de télévision, diffusion d'une conférence
      - Le même flux est envoyé à tous au même moment
    - Pour récupérer des informations sur le réseau
      - 224.0.0.12 : pour localiser un serveur DHCP
- Limites
  - Non fiable et non connecté comme UDP

Sockets en C

Configuration des options des sockets & Broadcast, multicast UDP/IP

Configuration des sockets

- Pour utiliser le multicast en C
  - Doit passer par la configuration des sockets
- Configuration socket/couche IP
  - Accès aux options d'une socket
    - Lire l'état d'une option : `getsockopt`
      - `int getsockopt(int sock, int niveau, int option, void *valeur, socklen_t *longueur)`
    - Modifier l'état d'une option : `setsockopt`
      - `int setsockopt(int sock, int niveau, int option, void *valeur, socklen_t longeur)`
    - Ces fonctions retournent -1 si problème

Configuration des sockets

- Paramètres de `[get/set]sockopt`
  - `sock` : la socket que l'on veut gérer
  - `niveau` : le niveau du protocole choisi, valeurs entre autres parmi
    - SOL_SOCKET : la socket elle même
    - IPPROTO_IP : la couche IP
    - IPPROTO_TCP : la couche TCP
    - IPPROTO_UDP : la couche UDP
  - `option` : option choisie
    - Notamment pour le niveau SOL_SOCKET
      - SO_BROADCAST : autorisation de diffusion des paquets
      - SO_RVCBUF/SO_SNDBUF : taille des buffers de réception et d'émission
      - SO_REUSEADDR : autorise de lier plusieurs sockets au même port
      - SO_TYPE (avec get) : retourne le type de la socket (SOCK_DGRAM ...)
    - Notamment pour le niveau IP_PROTO
      - IP_[ADD/DROP]_MEMBERSHIP : inscription ou désinscription d'un groupe multicast
      - IP_MULTICAST_LOOP : paquet diffusé est reçu ou pas à son émetteur
      - IP_MULTICAST_TTL : TTL d'un paquet envoyé en multicast

Configuration des sockets

- Paramètres de `[get/set]sockopt` (suite)
  - `valeur` : données décrivant l'état de l'option
    - En général, on utilise un entier ayant pour valeur 0 ou 1
      - 0 : l'option n'est pas positionnée
      - 1 : l'option est positionnée
  - `longueur` : longueur du champ valeur
- Exemple pour faire de la diffusion (broadcast)
  - ```
    int autorisation, sock;
    sock = socket(AF_INET, SOCK_DGRAM, 0);
    autorisation = 1;
    setsockopt(SOL_SOCKET, SO_BROADCAST, &autorisation,
               sizeof(int));
    ```
  - On diffuse les paquets en utilisant l'adresse de diffusion du réseau
    - Adresse IP dont tous les bits codant la machine sont à 1
    - Ex pour machine 192.21.12.23 de classe C : adresse de diffusion = 192.21.12.255
    - Toutes les machines connectés au réseau recevront ce paquet
      - Attention à lire sur le même port que celui utilisé pour l'émission
    ```

Réalisation de multicast en C

- Pour décrire le groupe multicast, structure `ip_mreq`
  -
```
struct ip_mreq {
    struct in_addr imr_multiaddr;
    struct in_addr imr_interface;
};
```
    - `imr_multiaddr` : adresse IP multicast
    - `imr_interface` : adresse IP locale ou interface locale
      - On utilisera par défaut `INADDR_ANY`

Réalisation de multicast en C

- Pour initialiser une socket UDP en mode multicast, actions à effectuer
  1. Créer la socket UDP de manière normale
  2. Créer l'objet `ip_mreq`
  3. Associer cet objet `ip_mreq` à la socket avec l'option `IP_ADD_MEMBERSHIP`
     - Abonnement au groupe multicast
  4. Éventuellement appliquer l'option `SO_REUSEADDR`
     - Sinon on ne peut pas avoir 2 programmes utilisant le même groupe multicast sur la même machine à cause du bind réalisé
  5. Lier la socket au numéro de port du groupe
- Émission d'un paquet
  - On utilise le couple @IP du groupe/port du groupe

Multicast : exemple

- Adresse/port du groupe : 226.1.2.3:1234
  - Note : les erreurs ne sont pas gérées
-
```
int sock;
struct in_addr ip;
static struct sockaddr_in ad_multicast, adresse;
ip_mreq gr_multicast;

// création de la socket UDP
sock = socket(AF_INET, SOCK_DGRAM, 0);

// récupération adresse ip du groupe
inet_aton("226.1.2.3", &ip);

// création identificateur du groupe
gr_multicast.imr_multiaddr.s_addr = ip.s_addr;
gr_multicast.imr_interface.s_addr =
                                htons(INADDR_ANY);
```

Multicast : exemple

- // abonnement de la socket au groupe multicast

```
setsockopt(sock, IPPROTO_IP, IP_ADD_MEMBERSHIP,
           &gr_multicast, sizeof(struct ip_mreq));
```

// autorise de lier plusieurs sockets sur le port utilisé par cette
// socket, c'est-à-dire sur le port du groupe multicast

```
int reuse = 1;
setsockopt(sock, SOL_SOCKET, SO_REUSEADDR,
                 (int *)&reuse, sizeof(reuse));
```

// liaison de la socket au port du groupe multicast

```
bzero((char *) &adresse, sizeof(adresse));
ad_multicast.sin_family = AF_INET;
ad_multicast.sin_addr.s_addr = htons(INADDR_ANY);
ad_multicast.sin_port = htons(1234);
bind(sock, &adresse, sizeof(struct sockaddr_in));
```

Multicast : exemple

- // émission d'un paquet :
  // on l'envoie au couple @/port du groupe

```
static struct sockaddr_in adresse;
int longueur_adresse = sizeof(struct sockaddr_in);
bzero((char *) &adresse, sizeof(adresse));
adresse.sin_family = AF_INET;
adresse.sin_addr.s_addr = ip.s_addr;
adresse.sin_port = htons(1234);
sendto(sock, message, tailleMessage , 0,
   (struct sockaddr*)&adresse, longueur_adresse);
```

  // réception d'un paquet : avec `recvfrom` ou peut utiliser
  // aussi `recv` car ne recevra des paquets que venant du groupe

```
recv(sock, buffer, TAILLEBUFFER, 0);
```

Multicast en Java

Multicast UDP en Java

- Classe `java.net.MulticastSocket`
  - Spécialisation de `DatagramSocket`
  - Constructeurs : identiques à ceux de `DatagramSocket`
    - `public MulticastSocket() throws SocketException`
      - Crée une nouvelle socket en la liant à un port quelconque libre
      - Exception levée en cas de problème (a priori il doit pas y en avoir)
    - `public MulticastSocket(int port) throws SocketException`
      - Crée une nouvelle socket en la liant au port précisé par le paramètre `port` : c'est le port qui identifie le groupe de multicast
      - Exception levée en cas de problème

Multicast UDP en Java

- Classe `java.net.MulticastSocket` (suite)
  - Gestion des groupes
    - `public void joinGroup(InetAddress mcastaddr) throws IOException`
      - Rejoint le groupe dont l'adresse IP multicast est passée en paramètre
      - L'exception est levée en cas de problèmes, notamment si l'adresse IP n'est pas une adresse IP multicast valide
    - `public void leaveGroup(InetAddress mcastaddr) throws IOException`
      - Quitte un groupe de multicast
      - L'exception est levée si l'adresse IP n'est pas une adresse IP multicast valide
      - Pas d'exception levée ou de problème quand on quitte un groupe auquel on appartient pas

Multicast UDP en Java

- Classe `java.net.MulticastSocket` (suite)
  - Emission/réception de données
    - On utilise les services `send()` et `receive()` avec des paquets de type `DatagramPacket` tout comme avec une socket UDP standard
- Exemple, exécution dans l'ordre :
  - Connexion à un groupe
  - Envoi d'un paquet
  - Réception d'un paquet
  - Quitte le groupe

Multicast UDP en Java

- Exemple de communication via socket multicast UDP

```
// adresse IP multicast du groupe
InetAddress group = InetAddress.getByName("228.5.6.7");

// socket UDP multicast pour communiquer avec groupe 228.5.6.7:4000
MulticastSocket socket = new MulticastSocket(4000);

// données à envoyer
byte[] data = (new String("youpi")).getBytes();

// paquet à envoyer (en précisant le couple @IP/port du groupe)
DatagramPacket packet =
     new DatagramPacket(data, data.length, group, 4000);

// on joint le groupe
 socket.joinGroup(group);
```

Multicast UDP en Java

- Exemple (suite)

```
// on envoie le paquet
socket.send(packet);

// attend un paquet en réponse
 socket.receive(packet);

// traite le résultat
...
// quitte le groupe
socket.leaveGroup(group);
```

- Notes
  - Il est possible que le receive récupère le paquet que le send vient juste d'envoyer
  - Besoin d'un autre receive pour réponse venant d'un autre élément