

Réseaux IP : IHM

Licence Informatique 3^{ème} année

Sockets TCP/UDP et leur mise en œuvre en Java

Eric Cariou

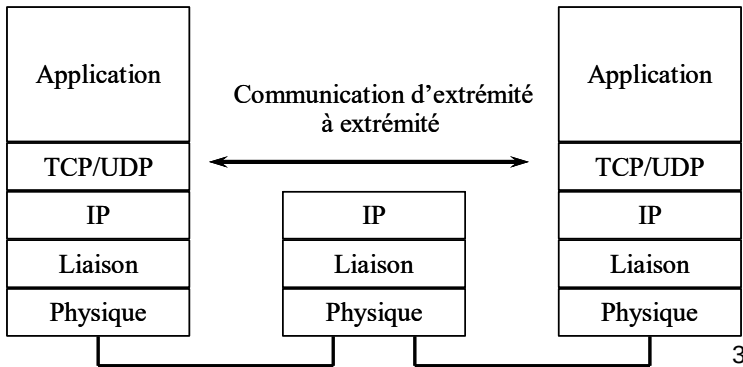
Université de Bretagne Occidentale
UFR Sciences & Techniques – Département Informatique

Eric.Cariou@univ-brest.fr

1

Rappel sur les réseaux

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



3

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

5

Plan

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

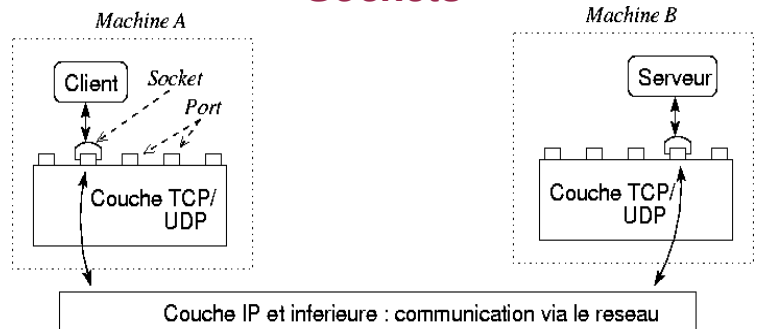
2

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

4

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

6

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

7

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)

9

Sockets UDP en Java

11

Sockets UDP

8

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

10

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)

12

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

13

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`

14

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

15

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`
 - ◆ Peut aussi préciser un offset pour ne pas lire les données au début de `buf`
 - ◆ `address` : adresse IP de la machine destinataire des données
 - ◆ `port` : numéro de port distant (sur la machine destinataire) où envoyer les données

16

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

17

Datagramme

◆ Classe `DatagramPacket`

◆ Méthodes « set »

- ◆ `void setAddress(InetAddress 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

18

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
 - ◆ Liaison réalisée à la première émission ou réception de données
 - ◆ 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`
 - ◆ Si 0 : lie (tout de suite) la socket à un port quelconque libre
 - ◆ Exception levée en cas de problème : notamment quand le port est déjà occupé

19

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

21

Sockets mode datagramme (UDP)

◆ Classe DatagramSocket

◆ Méthodes d'émission/réception de paquet

- ◆ `public void send(DatagramPacket pack) 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 avec la socket
- ◆ `public void receive(DatagramPacket pack) throws IOException`
 - ◆ Reçoit un paquet de données
 - ◆ Bloquant tant qu'un paquet n'est pas reçu
 - ◆ Quand paquet arrive, les attributs de `pack` sont modifiés
 - ◆ Les données reçues sont copiées dans le tableau passé en paramètre lors de la création de `pack` et sa longueur est positionnée avec la taille des données reçues
 - ◆ Les attributs d'@IP et de port de `pack` contiennent l'@IP et le port de la socket distante qui a émis le paquet

20

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`

22

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);
```

23

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());
System.out.println("recu : "+chaine);
```

24

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 reçu";  
packet.setData(reponse.getBytes());  
packet.setLength(reponse.length());
```

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

25

Critique sockets UDP

- ◆ Avantages
 - ◆ Simple à programmer (et à appréhender)
- ◆ Inconvénients
 - ◆ Pas fiable : ne sait pas si un paquet envoyé est reçu
 - ◆ Attention à la taille des données envoyées
 - ◆ Si le tableau d'octets envoyé ne tient pas dans 1 datagramme UDP, il est tronqué
 - ◆ La plupart des systèmes limitent la taille des datagrammes UDP à 8 Ko voire à 512 octets
 - ◆ En réception, si le tableau est plus petit que les données reçues, elles sont tronquées
 - ◆ 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

27

Conversion Object <-> byte[]

- ◆ Pour émettre et recevoir n'importe quel objet via des sockets UDP (voir cours sur les flux Java)

- ◆ 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();  
}
```

29

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());  
System.out.println(" reçu du serveur : "+chaine);
```

26

Structure des données échangées

- ◆ Format des données à transmettre
 - ◆ Très limité a priori pour du Java : tableaux de byte
 - ◆ 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
 - ◆ On envoie la copie binaire de l'objet Java en mémoire via les mécanismes de sérialisation
 - ◆ Voir cours de complément Java pour détails

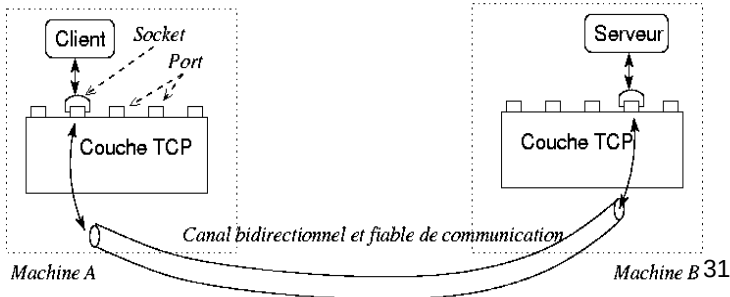
28

Sockets TCP

30

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

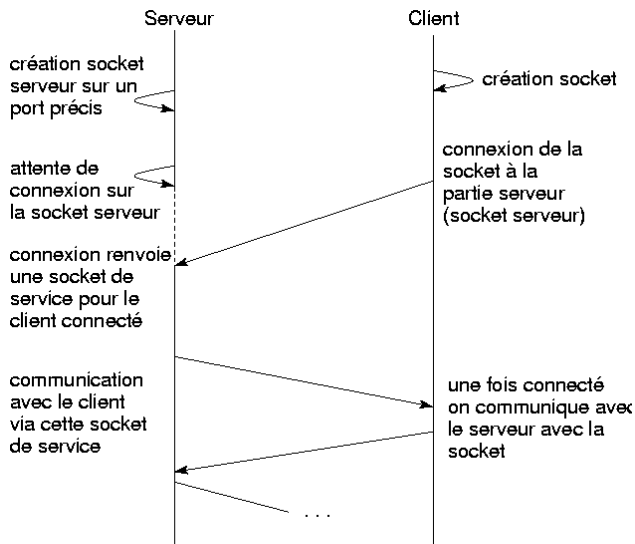


Sockets TCP : principe

- ◆ 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 côté 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

32

Sockets TCP : résumé communication



33

Sockets TCP en Java

34

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 (même si toujours possible)
 - ◆ 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

35

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 côté server

36

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 37

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 39

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
 - ◆ Libère le port local utilisé par la socket 38

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` 40

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(" reçu 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);
```

43

Sockets TCP

◆ Critique sockets TCP

◆ Avantages

- ◆ Niveau d'abstraction plus élevé qu'avec UDP
 - ◆ Mode connecté avec phase de connexion explicite
 - ◆ Fiable, pas de perte de données
 - ◆ 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 ou un sélecteur pour savoir quelle socket a reçu des données (ou qu'une demande de connexion est pendante)
 - ◆ Mais oblige une bonne structuration coté serveur

45

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'acquiescement...)
 - ◆ Peu de pertes de paquets 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

47

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;

44

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 trop 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, le thermomètre envoie ses données quoiqu'il arrive, que l'afficheur soit fonctionnel ou pas, ça ne change rien pour lui

46

Gestion multi-clients

48

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

49

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 puis regarde de qui il vient
 - ◆ TCP : doit se mettre en attente de données sur toutes les sockets actives

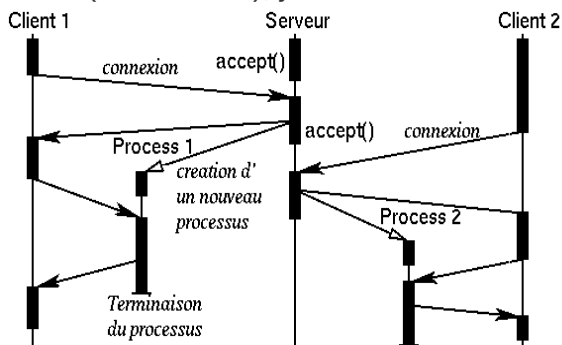
51

Sockets TCP – gestion plusieurs clients

- ◆ Boucle de fonctionnement général d'un serveur pour gérer plusieurs clients

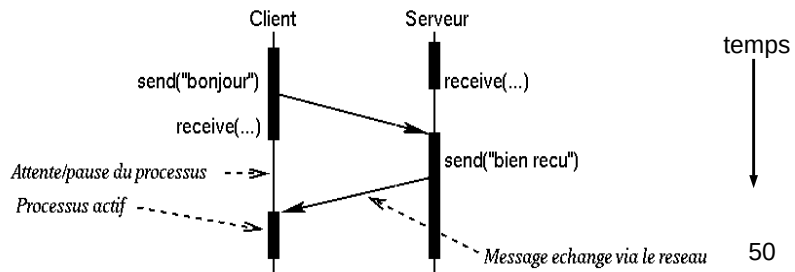
```
while(true) {
    socketClient = acceptConnection();
    newProcessus(socketClient);
}
```

- ◆ Exemple avec 2 clients →



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



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 (sauf avec un sélecteur)
- ◆ 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

52

Java – gestion plusieurs clients TCP

- ◆ Coté serveur
 - ◆ En Java, contrairement au C, les flots d'exécution parallèle ne sont que des threads
 - ◆ Pas de processus lourds comme avec le `fork()` du C
 - ◆ Dans la boucle d'acceptation de connexion
 - ◆ Crée un thread Java dédié à la communication avec le client qui vient de se connecter
 - ◆ Cf TD/TP

54

Java – gestion plusieurs clients TCP

- ◆ Alternative au multi-threads : utiliser un sélecteur
 - ◆ Classes du package `java.nio` : *New Input Output*
 - ◆ Associe plusieurs sockets à un sélecteur
 - ◆ Sockets de service ou la socket d'attente de connexions
 - ◆ On attend sur le sélecteur que quelque chose arrive
 - ◆ Données reçues par une socket
 - ◆ Connexion d'un nouveau client
 - ◆ On récupère la socket associée et on peut faire l'action (lecture ou acceptation de connexion) dessus
- ◆ Il n'est donc plus utile d'avoir un thread par socket de client
 - ◆ Moins coûteux en mémoire et en temps de bascule d'un thread à un autre
 - ◆ Mais plus compliqué d'avoir un contexte (données/session) spécifique à chaque client

55

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

57

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
 - ◆ Bien souvent filtré au-delà des réseaux locaux

59

Multicast UDP/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

56

58

Multicast en Java

60

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

61

Multicast UDP en Java

- ◆ Classe `java.net.MulticastSocket` (suite)
 - ◆ Nouvelles méthodes de gestion de groupes
 - ◆ Joint une interface réseau de la machine à un groupe multicast
 - ◆ Précise le numéro de port dans le groupe multicast rejoint via une adresse de socket
 - ◆ `public void joinGroup(SocketAddress mcastaddr, NetworkInterface netIf) throws IOException`
 - ◆ Rejoint le groupe multicast pour une interface réseau
 - ◆ L'exception est levée en cas de problèmes, notamment si l'adresse IP n'est pas une adresse IP multicast valide
 - ◆ `public void leaveGroup(SocketAddress mcastaddr, NetworkInterface netIf) throws IOException`
 - ◆ Quitte un groupe de multicast
 - ◆ L'exception est levée si l'adresse IP n'est pas une adresse IP multicast valide ou si on quitte un groupe auquel on n'appartient pas

63

Multicast UDP en Java

- ◆ Exemple de communication via socket multicast UDP

```
// socket UDP multicast liée au port local 4000
MulticastSocket socket = new MulticastSocket(4000);

// adresse IP multicast du groupe : 228.5.6.7
InetAddress ipAddr = InetAddress.getByAddress("228.5.6.7");

// adresse multicast de la socket : 228.5.6.7:4000
InetSocketAddress group =
    new InetSocketAddress(ipAddr, 4000);

// interface réseau de l'adresse locale par défaut
NetworkInterface nif =
    NetworkInterface.getByInetAddress(InetAddress.getLocalHost());

// lit l'interface réseau au groupe multicast
socket.joinGroup(group, nif);
```

65

Multicast UDP en Java

- ◆ Classe `java.net.MulticastSocket` (suite)
 - ◆ Gestion des groupes (anciennes méthodes, dépréciées)
 - ◆ `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 n'appartient pas

62

Multicast UDP en Java

- ◆ Classe `java.net.MulticastSocket` (suite)
 - ◆ Émission/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

64

Multicast UDP en Java

- ◆ Exemple (suite)

```
// 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, ipAddr, 4000);

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

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

// traite le résultat, fait d'autres envois/réceptions
...
// quitte le groupe pour l'interface réseau qui a été liée
socket.leaveGroup(group, nif);
```

66

Multicast UDP en Java

- ◆ Chaque participant au groupe doit avoir sa socket liée sur le port du groupe
 - ◆ Pour pouvoir recevoir les paquets à destination de ce port
 - ◆ Par défaut, on ne peut pas avoir 2 sockets sur le même port UDP sur la même machine
 - ◆ Sauf à paramétrer la socket : `socket.setReuseAddress(true);`
 - ◆ Ce code est exécuté dans les constructeurs de `MulticastSocket`
- ◆ Les paquets envoyés sont à destination de toutes les sockets associées au groupe
 - ◆ Une application reçoit donc les paquets qu'elle a envoyés
 - ◆ Pour ne plus recevoir ses propres paquets :
`socket.setOption(StandardSocketOptions.IP_MULTICAST_LOOP, false);`
 - ◆ Attention, réglage au niveau port local : s'il y a 2 applications sur la même machine, si on supprime l'auto-réception, aucune des 2 ne recevra un message envoyé par l'autre