Module 19 :
Application client/serveur

Sockets TCP/UDP et leur mise en œuvre en Java

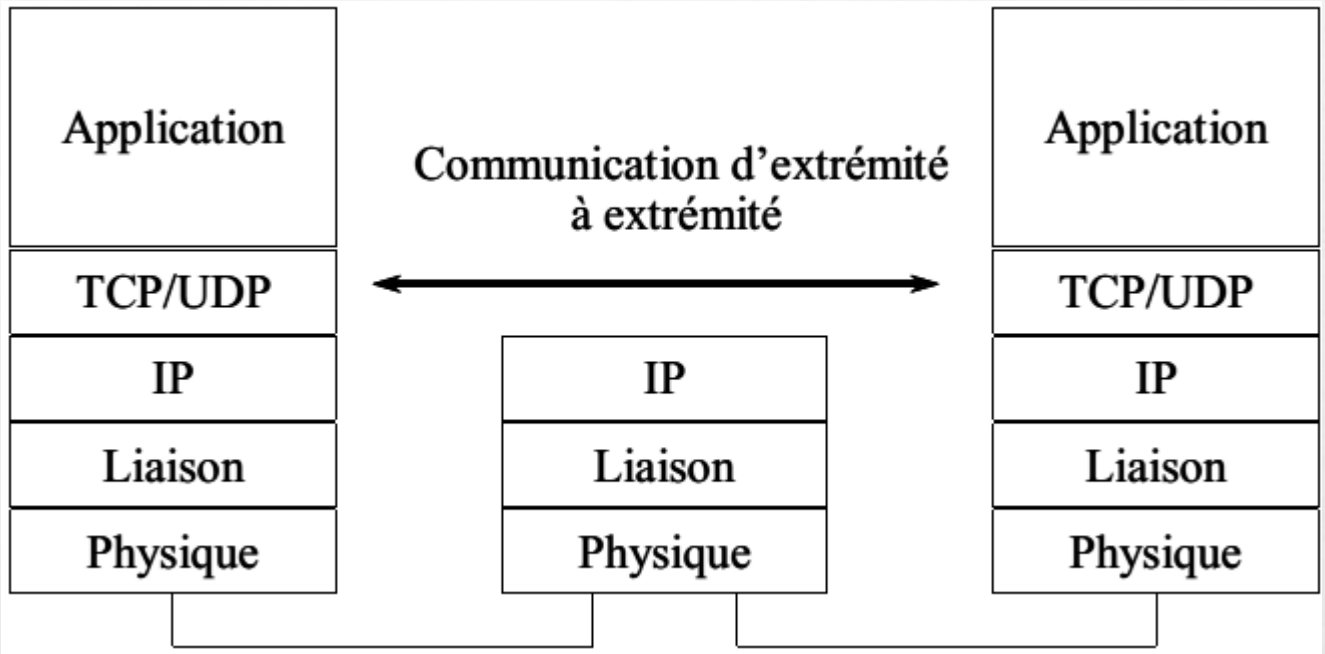
Benkhayat Yassine
yasben11@gmail.com

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 et numéro de port
  - 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*
    - 192.129.1 2.34:80 : accès au serveur Web tournant sur la machine d'adresse IP 192.129.1 2.34

Sockets

- Socket : prise
  - Associée, liée à un port
    - C'est donc 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

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

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 arrivé (perdu par le réseau)
    - Un paquet P2 envoyé après un paquet P1 peut arriver avant ce paquet P1 (selon la gestion des routes dans le réseau)
- 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

Sockets UDP : principe (suite)

- 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
  - Destinataire = 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 pb réseau)
- Si le client envoie un paquet avant que le serveur ne soit prêt à recevoir : le paquet est perdu

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)

Sockets UDP en Java

- 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
    - **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

Sockets UDP en Java

- DatagramPacket
- Classe DatagramPacket
  - Constructeurs
    - 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

Sockets UDP en Java

- 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

Sockets UDP en Java

- 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 UDP en Java

- **Classe DatagramSocket**
  - Socket en mode datagramme
  - Constructeurs
    - public DatagramSocket()
      - Crée une nouvelle socket en la liant à un port quelconque libre
    - public DatagramSocket(int port)
      - Crée une nouvelle socket en la liant au port local précisé par le paramètre port
  - Methodes :
    - 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

Sockets UDP en Java

- **Classe DatagramSocket**
  - Méthodes d'émission/réception de paquet
    - public void send(DatagramPacket p)
      - Envoie le paquet passé en paramètre. Le destinataire est identifié par le couple @IP/port précisé dans le paquet
    - public void receive(DatagramPacket p)
      - 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 UDP en Java

- **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)
    - 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.

Sockets UDP Java – exemple coté Client

```
InetAddress adr;
DatagramPacket packet;
DatagramSocket socket;
// adr contient l'@IP de la partie serveur
adr = InetAddress.getByName("yassine-PC");
// 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 1 5 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) ;
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)
packet.setData((new String("bien recu")).getBytes() ) ;
// on envoie le paquet au client
socket.send(packet);
```

Sockets UDP Java – multidiffusion

- On a vu comment faire communiquer des applications 1 à 1 via des sockets UDP
- 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 : envoie 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 reçoit les données sur cette adresse IP

Sockets UDP Java – multidiffusion

- Adresse IP multicast
  - Classe d'adresse IP entre 224.0.0.0 et 239.255.255.255
    - Casse 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

Sockets UDP Java – multidiffusion

- Classe java.net.MulticastSocket
  - Constructeurs : (identiques à ceux de DatagramSocket)
    - public MulticastSocket() :Crée une nouvelle socket en la liant à un port quelconque libre
    - public MulticastSocket(int port) : Crée une nouvelle socket en la liant au port précisé par le paramètre port : c'est le port qui identifie le groupe de multicast
    - public void joinGroup(InetAddress add) :Rejoint le groupe dont l'adresse IP multicast est passée en paramètre
    - public void leaveGroup(InetAddress mcastaddr): Quitte un groupe de multicast

Sockets UDP Java – multidiffusion

- 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
  - Exécution dans l'ordre
    - Connexion à un groupe.
    - Envoi d'un paquet
    - Réception d'un paquet.
    - Quitte le groupe.

Sockets UDP Java – multidiffusion

Coté Serveur

```
//creation des données à envoyer
byte[] buff = new byte[1024];
buff = (new String("youpi")).getBytes();

// Créer un paquet à envoyer
InetAddress groupe_ip = InetAddress.getByName("225.0.0.1");
DatagramPacket paquet = new DatagramPacket(buff, buff.length, groupe_ip, 4000);

// Envoyer le paquet au groupe Multicast
MulticastSocket ms = new MulticastSocket();
ms.joinGroup(groupe_ip);
ms.send(paquet);
```

Sockets UDP Java – multidiffusion

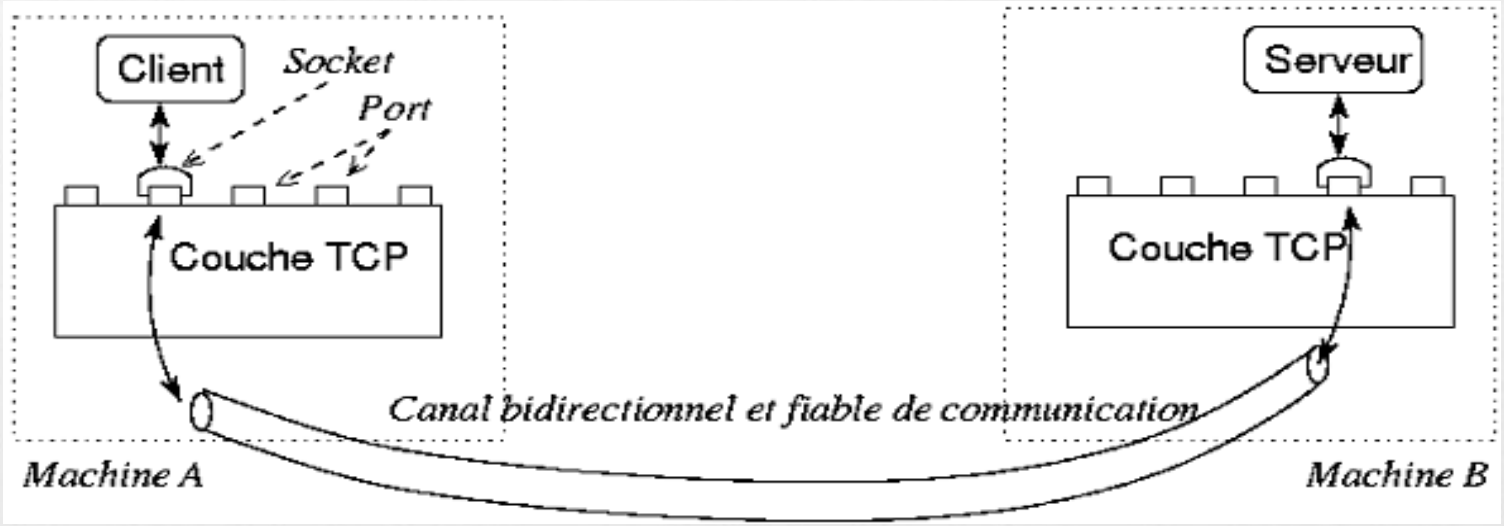
Coté Client

```
// Créer un socket qui est lié à un port
MulticastSocket msocket = new MulticastSocket(4000);
InetAddress ip_groupe= InetAddress.getByName("225.0.0.1");

// Joindre un groupe
msocket.joinGroup(ip_groupe);

// Commencer à recevoir les données diffusées par le groupe
byte[] buff = new byte[1024];
DatagramPacket paquet = new DatagramPacket(buff, buff.length);

//reception du packet multicast
msocket.receive(paquet);
System.out.println ("je suis le client 1 j'ai recu : "+new String(paquet.getData()));
```

Sockets TCP

Sockets TCP: principe

- Fonctionnement en mode connecté
  - Données envoyées dans un « tuyau » et non pas par paquet
    - Flux de données
    - Correspond aux flux Java dans la mise en œuvre Java des sockets TCP
  - 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 d'écoute sur un certain port bien précis et appelle un service d'attente de connexion de la part d'un client
  - Le client appelle un service pour ouvrir une connexion avec le serveur
    - Il récupère une socket (associée à un port quelconque par le système)
  - 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
  - Comme avec sockets UDP : le client et le serveur communiquent en envoyant et recevant des données via leur socket

Sockets TCP en Java

- Classes du package j ava. 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

Sockets TCP en Java

- Classe Socket
  - Socket mode connecté
  - Constructeurs
    - public Socket(InetAddress address, int port)
      Crée une socket locale et la connecte à un port distant d'une machine distante identifié par le couple address/port
    - public Socket(String address, int port)
      - Idem mais avec nom de la machine au lieu de son adresse IP codée
      - Variante de ces 2 constructeurs pour préciser en plus un port local sur lequel sera liée la socket créée

Sockets TCP en Java

- Classe Socket
  - Méthodes d'émission/réception de données
    - Contrairement aux sockets UDP, les sockets TCP n'offre 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

Sockets TCP en Java

- 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 distanteint
    - **getLocalPort**()
      - Renvoie le port local sur lequel est liée la socket
  - public void **setSoTimeout**(int timeout)
    - 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

Sockets TCP en Java

- Classe ServerSocket
  - Socket d'attente de connexion, coté serveur uniquement
  - Constructeurs
    - public ServerSocket(int port)
      - 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()**
      - 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)
      - 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

```
// adresse IP du serveur
InetAddress adr = InetAddress.getByName("pc-yasben");

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

// 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(" la chaine recu du serveur est  : " + chaine);
```

Sockets TCP Java – exemple coté Serveur

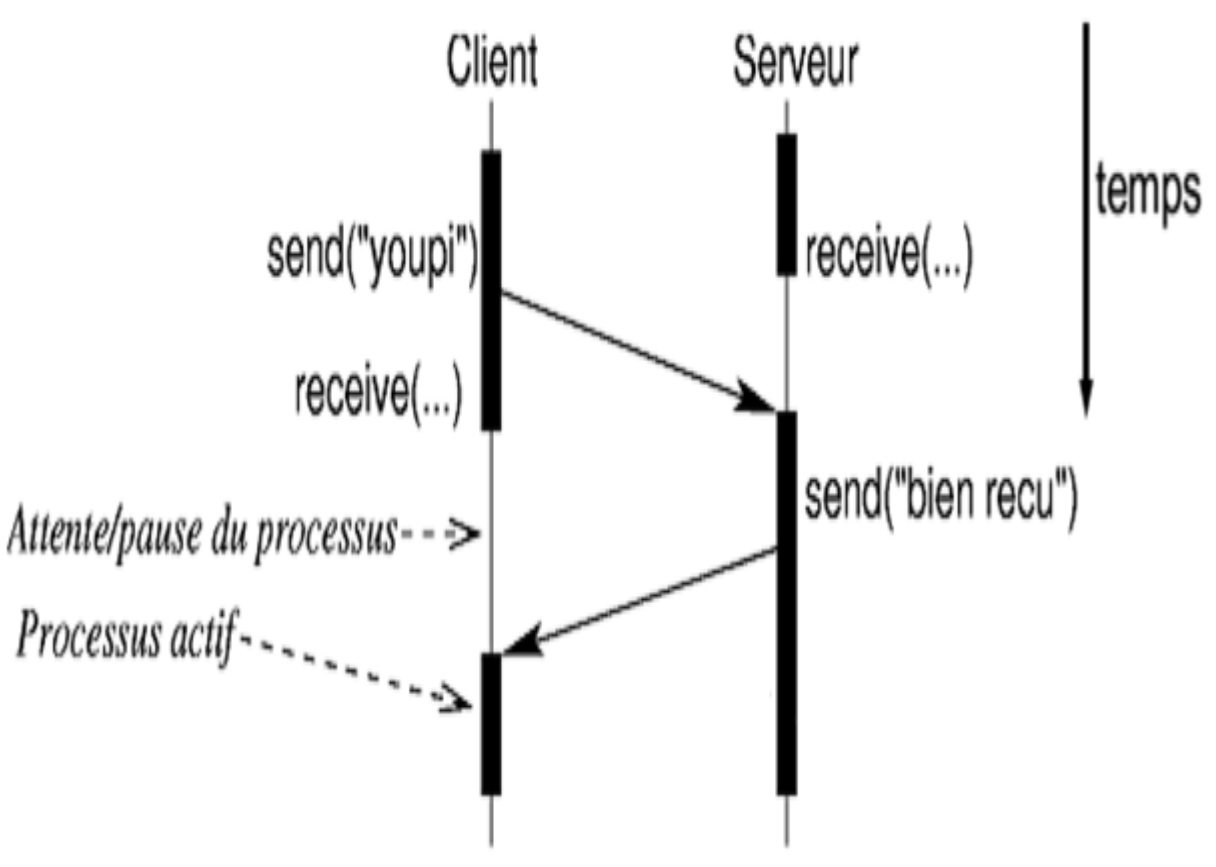
```
// serveur positionne sa socket d'écoute sur le port local 7777
ServerSocket serverSocket = new ServerSocket(7777);

// se met en attente de connexion de la part d'un client distant
Socket socket = serverSocket.accept();
System.out.println("connexion accepté");

// 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);
System.out.println(" ca vient de : " + socket.getInetAddress() + ": " +
socket.getPort());
output.writeObject(new String("Bien Reçu"));
```

Concurrence dans une application

Threads Java

Concurrence

Exemple de flux d'exécution pour notre exemple précédent



Sockets TCP – 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 : 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

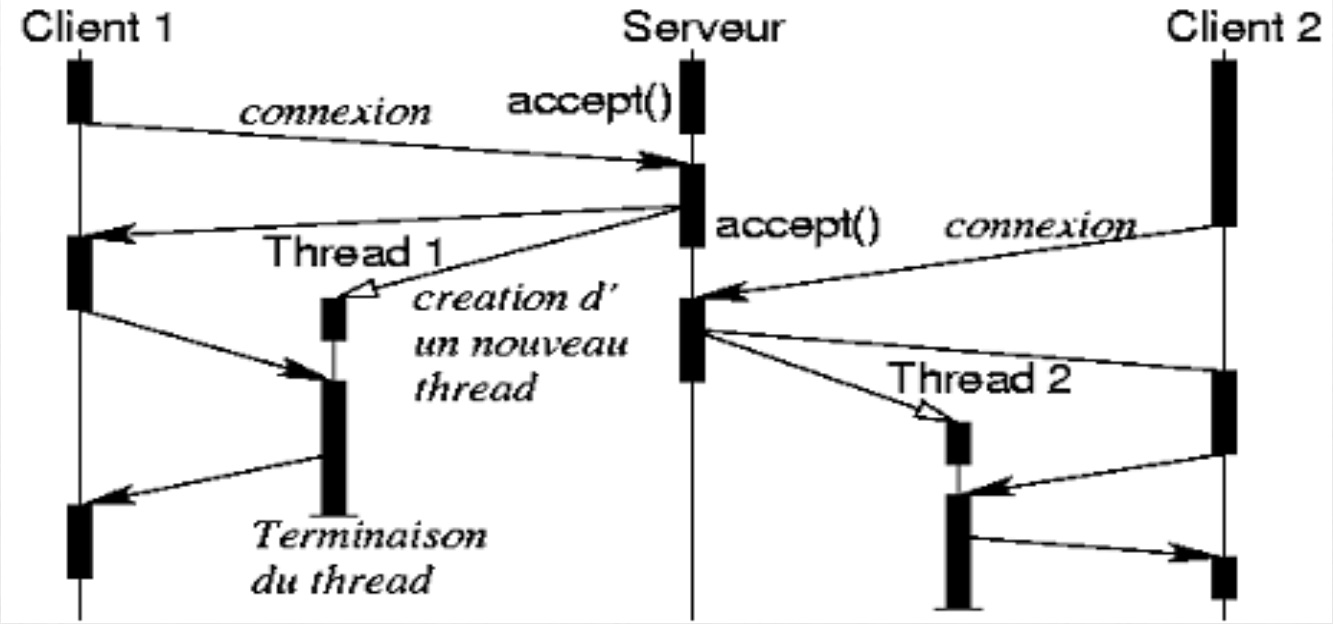
- Particularité coté serveur en TCP(suite)
  - Contrainte
    - Lecture sur une socket : opération bloquante
      - Tant que des données ne sont pas reçues
    - Attente de connexion : opération bloquante
      - Jusqu'à la prochaine connexion d'un client distant
  - Avec un seul processus
    - Si on 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()

    newThread(socketClient)
- Exemple Serveur avec 2 clients



Gestion plusieurs clients

- Java offre un mécanisme permettant de gérer des flux d'exécution parallèle
  - Les threads
- Rappel différence processus/thread
  - Le processus est créé comme une copie d'un processus existant
    - Deux processus distincts avec leur mémoire propre
  - Le thread s'exécute au sein d'un processus existant
    - Nouveau flux d'exécution interne
    - Partage des données du processus

Threads en Java

- Pour créer et lancer un nouveau thread, 2 modes
  - Etendre la classe java.lang.Thread
    - Redéfinir la méthode public void run()
      - Qui contient la séquence de code qu'exécutera le thread
    - Pour lancer le thread
      - Instancier normalement la classe définie
      - Appeler ensuite la méthode start() sur l'objet créé
  - Implémenter l'interface java.lang.Runnable
    - Définir la méthode public void run() de cette interface
      - Qui contient la séquence de code qu'exécutera la thread
    - Pour lancer le thread
      - Instancier normalement la classe définie
      - Créer une instance de la classe Thread en passant cet objet en paramètre
      - Lancer la méthode start() du thread instancié

Thread Java – exemple (avec la classe Thread)

Classe **Exemple_Thread**

```
public class Exemple_Thread extends Thread{
    public void run(){
try {
//le thread va dormir 5 second
Thread.sleep(5000);
      } catch (Exception ex) {
    System.out.println("Exeption");
}
     //affichage du nom du thread
System.out.println("je suis le thread
:"+Thread.currentThread().getName());
    }
}
```

Classe **NewMain : Execution**

```
public class NewMain {
    public static void main(String[] args) {
System.out.println("je suis le programme principale :
Debut ");
//Creation d'un objet t de la classe Thread
Exemple_Thread t = new Exemple_Thread();
//lancer le thread
t.start();
System.out.println("je suis le programme principale:Fin ");
    }
}
```

Le résultat de l'exécution, est :

je suis le programme principale : Debut
je suis le programme principale : Fin
je suis le thread :Thread-0

✓ **Remarque :**

Même si le thread **Thread-0** dort 5 second cela n'a pas empêché le programme de continuer l'exécution

Thread Java – exemple (avec l'interface Runnable)

Classe **Exemple_runnable**

```
public class Exemple_runnable implements Runnable{
    public void run(){
try {
//le thread va dormir 5 second
Thread.sleep(5000);
} catch (Exception ex) {
    System.out.println("Exception");
}//affichage du nom du thread
System.out.println("je suis le thread
:"+Thread.currentThread().getName());
    }
}
```

Classe **NewMain : Execution**

```
public class NewMain {
    public static void main(String[] args) {
System.out.println("je suis le programme principale : Debut
");
//creation d'un objet de la classe Exemple_runnable
Exemple_runnable e =new Exemple_runnable();
//creation d'un objet de la classe thread
/*passer l'objet e de la classe Exemple_runnable en
parametre   */
Thread t = new Thread(e);
//lancer le thread
t.start();
System.out.println("je suis le programme principale : Fin ");
    }
}
```

Le résultat de l'exécution, est :

je suis le programme principale : Debut
je suis le programme principale : Fin
je suis le thread :Thread-0

✓ **Remarque :**

Même si le thread **Thread-0** dort 5 second cela n'a pas empêché le programme de continuer l'exécution

Exemple d'un Serveur multi-clients

Classe NewThread contenant la méthode run()

```
public class NewThread extends Thread {
    Socket s;
    ObjectOutputStream output;
    ObjectInputStream input;
    public NewThread(Socket s, ObjectOutputStream output, ObjectInputStream input) {
this.s = s;
this.output = output;
this.input = input;
    }
    public void run() {
output = new ObjectOutputStream(s.getOutputStream());
input = new ObjectInputStream(s.getInputStream());
// attente les données venant du client
String chaine;
chaine = (String) input.readObject();
System.out.println(" recu : " + chaine);
System.out.println(" ca vient de : " + s.getInetAddress() + ": " + s.getPort());
output.writeObject(new String("Bien Reçu"));
System.out.println("je suis le thread"+Thread.currentThread().getName());
    }}
```

Exemple d'un Serveur multi-clients (suite)

Exécution du Serveur :classe **SocketServerTCP_TH**

```
public class SocketServerTCP_TH {
public static void main(String args[]) throws IOException, ClassNotFoundException {

// 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
while (true) {
Socket socket = serverSocket.accept();
System.out.println("connexion accepté");

//à chaque connexion d'un client on lui assosié un thread a part
NewThread t = new  NewThread(socket, null, null);

//et on démarre ce thread
t.start();
 }}}
```

Sockets TCP Java – exemple coté client

```
// adresse IP du serveur
InetAddress adr = InetAddress.getByName("pc-yasben");

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

// 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(" la chaine recu du serveur est  : " + chaine);
```