SIMPLON.CO

Sockets en Java

Athanasia Katsouraki

13/11/2017

Sockets

❍Qu'est ce qu'un socket?

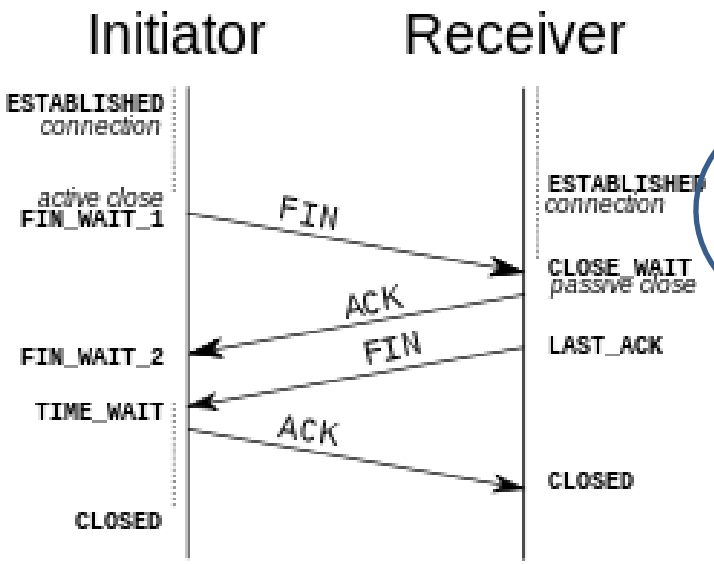
- Point d'entrée entre 2 appli. du réseau
- Permet l'échange de donnée entre elles à l'aide des mécanismes d'E/S (**java.io**)

Chacune de ces applications se voit attribuer une adresse unique sur la machine, codée sur 16 bits: **un port**
→la combinaison *adresse IP + port* est alors une adresse unique au monde: elle est appelée socket).

La notion de **port logiciel permet, sur un ordinateur donné, de distinguer différents interlocuteurs. Ces interlocuteurs sont des programmes informatiques qui, selon les cas, écoutent ou émettent des informations sur ces ports. Un port est distingué par son numéro

Différents types de sockets

Stream Sockets (TCP)



Transmission Control Protocol (TCP)

« protocole de contrôle de transmissions »

- un protocole de transport fiable
- en mode connecté

Dans le modèle Internet:

- appelé modèle TCP/IP

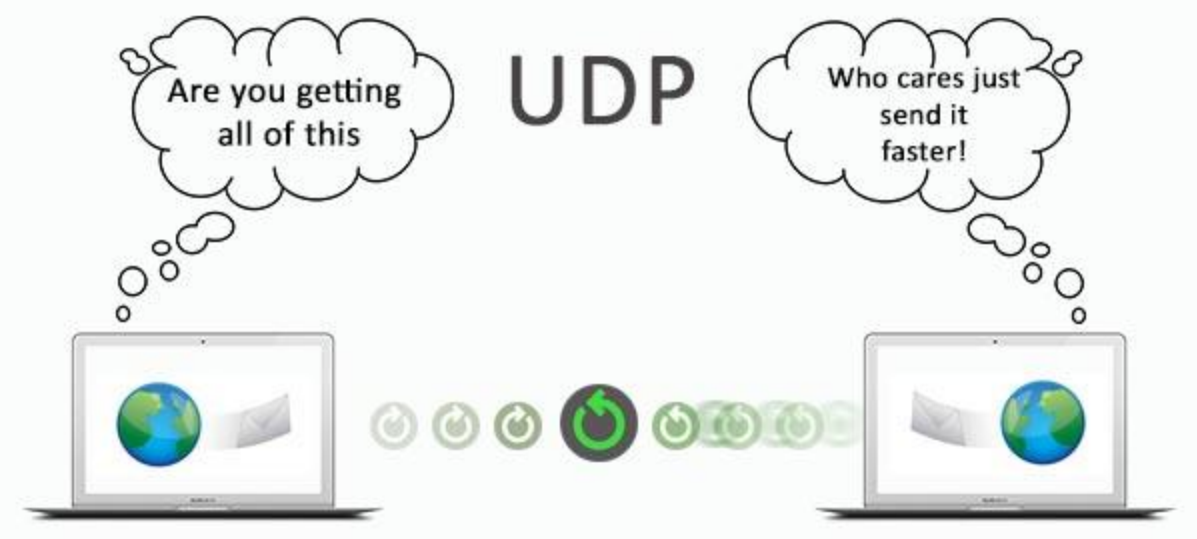
- Les applications transmettent des flux de données sur une connexion réseau.

✓établir une communication en mode connecté

✓si connexion interrompue : applications informées

Différents types de sockets

Datagram Sockets (UDP)



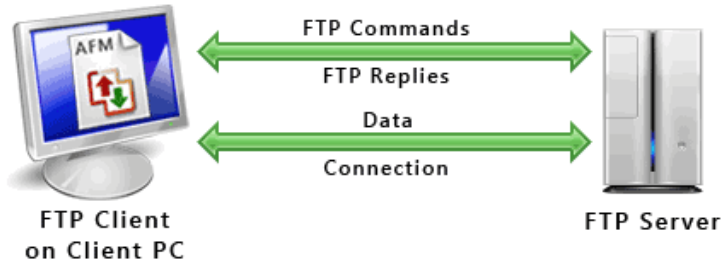
- ✓établir une communication en mode non connecté
- ✓données envoyées sous forme de paquets indépendants de toute connexion.
- ✓Plus rapide MAIS moins fiable que TCP

Exemples d'applications

❍Applications TCP

✓FTP

File Transfer Protocol, ou FTP, est un protocole de communication destiné au partage de fichiers sur un réseau TCP/IP

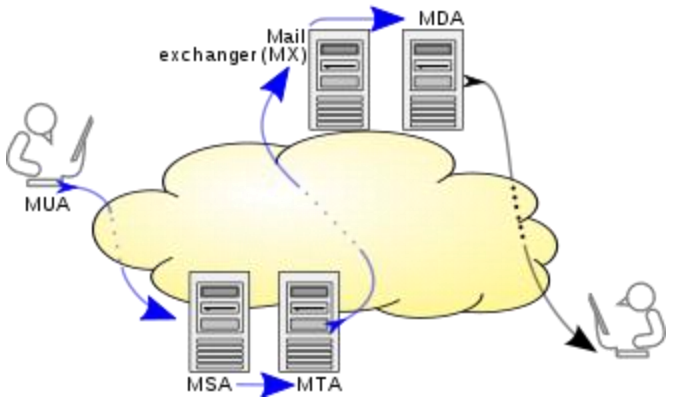


✓ SMTP :

Simple Mail Transfer Protocol

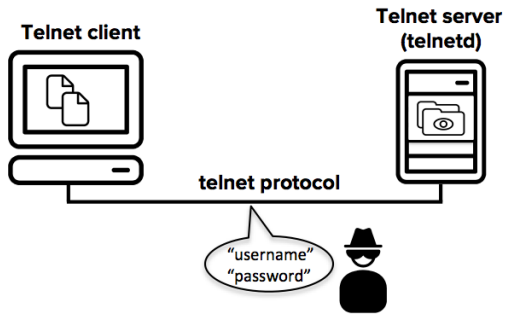
« protocole simple de transfert de courrier »

protocole de communication utilisé pour transférer le courrier électronique (courriel) vers les serveurs de messagerie électronique.



✓TELNET

Un protocole utilisé sur tout réseau TCP/IP, permettant de communiquer avec un serveur distant en échangeant des lignes de texte et en recevant des réponses également sous forme de texte.



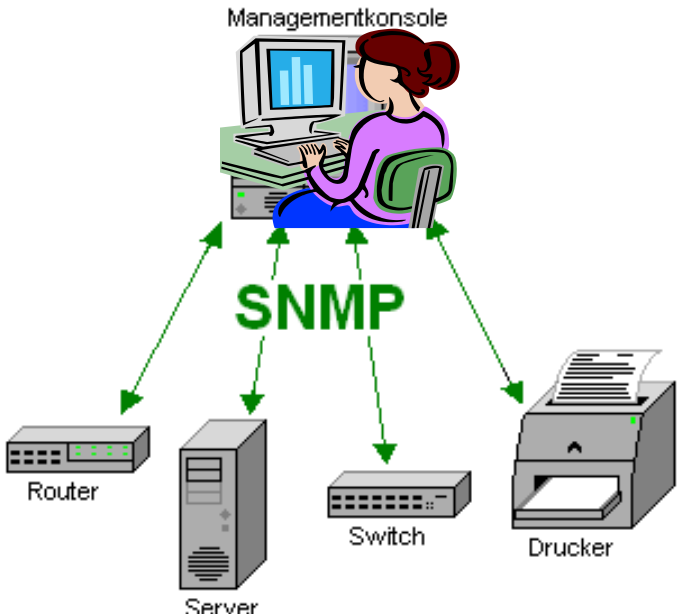
Exemples d'applications

❍Applications UDP

✓Simple Network Management Protocol (SNMP)

« protocole simple de gestion de réseau »: est un protocole de communication qui permet aux administrateurs réseau de:

- gérer les équipements du réseau
- superviser et
- diagnostiquer des problèmes réseaux et matériels à distance.



Concept Client/Server

L'environnement **client-serveur** désigne :

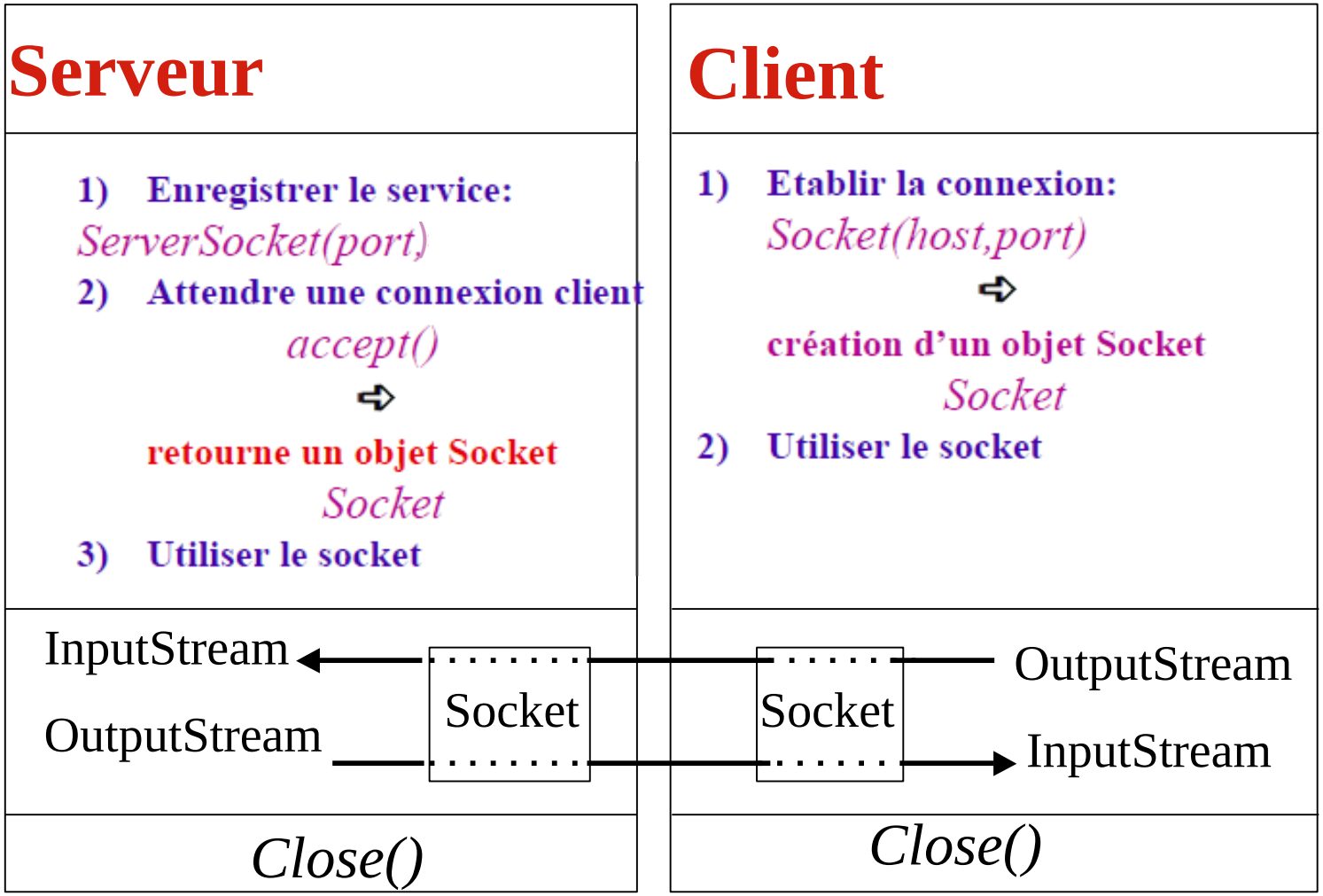
un mode de communication à travers un réseau entre plusieurs programmes :

- l'un, qualifié de client, envoie des requêtes ;
- l'autre/les autres, qualifiés de serveurs, attendent les requêtes des clients et y répondent.

Par extension,

- le client désigne également l'ordinateur ou la machine virtuelle sur lequel est exécuté le logiciel client
- et le serveur, l'ordinateur ou la machine virtuelle sur lequel est exécuté le logiciel serveur.

Le modèle client-serveur Java



Principe de fonctionnement (1)

❍Serveur: enregistrer le service

- le serveur enregistre son service sous un numéro de port, indiquant le nombre de clients qu'il accepte de faire bufférisér à un instant T (new serverSocket(….))

ServerSocket ser= **new ServerSocket(port);**

❍Serveur : attente de connexion

- il se met en attente d'une connexion (méthode accept() de son instance de ServerSocket)

Socket sock = ser.accept();

Principe de fonctionnement (2)

- **Client : établir la connexion**
  - le client peut alors établir une connexion en demandant la création d'un socket (new Socket()) à destination du serveur pour le port sur lequel le service a été enregistré.

  Socket sock= **new Socket("localhost",9999);**
- **Serveur**
  - le serveur sort de son accept() et récupère un Socket de communication avec le client
- **Les deux : utilisation du socket**
  - le client et le serveur peuvent alors utiliser des InputSteam et OutputStream pour échanger les données

PrintStream pr= **new PrintStream(sock.getOutputStream());**

BufferedReader ed = **new BufferedReader(new InputStreamReader(sock.getInputStream()));**

Un serveur TCP/IP (1)

- il utilise la classe **java.net.ServerSocket** pour accepter des connexions de clients
- quand un client se connecte à un port sur lequel un ServerSocket écoute, ServerSocket crée une nouvelle instance de la classe Socket pour supporter les communications côté serveur :

```
int port = …;
ServerSocket server = new ServerSocket(port);
Socket connection = server.accept();
```

Un serveur TCP/IP (2)

- les constructeurs et la plupart des méthodes peuvent générer une IOException

- la méthode accept() est dite bloquante ce qui implique un type de programmation particulier : boucle infinie qui se termine seulement si une erreur grave se produit

java.net.ServerSocket

```
final int PORT = …;   // default : 9999

try {
    ServerSocket serveur = new
    ServerSocket(PORT);

    while (true) {
        Socket socket = serveur.accept();
    }
}
 catch (IOException e){
 ….
 }
```

**https://books.google.fr/books?id=wHgdY-GLMboC&pg=PA280&lpg=PA280&dq=serveur+d%C3%A9marre+sur+le+port+9999&source=bl&ots=mkadCB1Rl7&sig=-OHlAoxnv0NwiXK95yuC7GG_OAs&hl=fr&sa=X&ved=0ahUKEwjC1PmFqLrXAhVH46QKHRaIAMMQ6AEIJjAA#v=onepage&q=serveur%20d%C3%A9marre%20sur%20le%20port%209999&f=false

Un client TCP/IP

- le client se connecte au serveur en créant une instance de la classe
  java.net.Socket : connexion synchrone

```
String host =
…;  int port =
…;
Socket connection = new Socket (host,port);
```

- le socket permet de supporter les communications côté client
- la méthode close() ferme (détruit) le socket
- les constructeurs et la plupart des méthodes peuvent générer une IOException
- le serveur doit être démarré avant le client. Dans le cas contraire, si le client se connecte à un serveur inexistant, une exception sera levée après un *time-out*

java.net.Socket

```
final String HOST = "…
";  final int PORT = …;


try {
      Socket socket = new Socket
      (HOST,PORT);
}
finally {
    try {socket.close();} catch (IOException
    e){}
 }
```

Flux de données (1)

- une fois la connexion réalisée, il faut obtenir les *streams* d'E/S (java.io) auprès de l'instance de la classe Socket en cours
- **Flux entrant**
  - ✓obtention d'un *stream* simple:

    définit les op. de base

    InputSteam in = socket.getInputStream();
  - ✓création d'un stream convertissant les bytes reçus en char

    InputSteamReader reader = new InputStreamReader(in);
  - ✓ création d'un stream de lecture avec tampon: pour lire ligne par ligne dans un stream de caractères

    BufferedReader istream = new BufferedReader(reader);
  - ✓ lecture d'une chaîne de caractères

    String line = istream.readline();

Flux de données (2)

- **Flux sortant**
  - ✓ obtention du flot de données sortantes : bytes

    OutputSteam out = socket.getOutputStream();
  - ✓ création d'un stream convertissant les bytes en chaînes de caractères

PrintWriter ostream = new PrintWriter(out);

✓ envoi d'une ligne de caractères

ostream.println(str);

✓ envoi effectif sur le réseau des bytes

(important)

ostream.flush();

Flux de données (3)

```
try {
 Socket socket = new Socket(HOST,PORT);
//Lecture du flux d'entrée en provenance du serveur
InputStreamReader reader = new
    InputStreamReader(socket.getInputStream());
BufferedReader istream = new BufferedReader(reader);  String line =
istream.readLine();
//Echo la ligne lue vers le serveur
PrintWriter ostream = new PrintWriter(socket.getOutputStream());
ostream.println(line);
ostream.flush();
}catch (IOException e) {…}
finally {try{socket.close();}catch (IOException e){}}
```

**le bloc finally{} exécuté quoiqu'il arrive

- aucune exception
- exception traité dans catch
- exception traité ailleurs

```
try {
uneMéthodeQuiJette();
} catch (ExceptionJettée e) {...}
finally {/* fait le ménage dans
tous les cas */}
```

**le bloc finally{} utilisé principalement pour fermer des ressources

- fichiers ouverts
- sockets
- ….