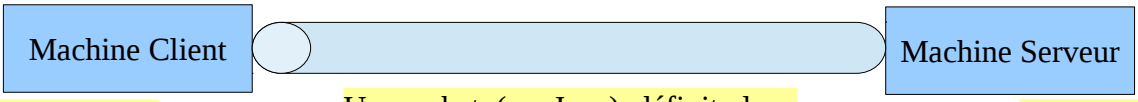
Exemple 02 : Programmation Réseau par Socket

Les Sockets est un mécanisme de communication de base entre des processus s'exécutant sur différents machines éloignées physiquement. Il se base sur la couche 4 (couche transport) de la pile protocolaire TCP/IP.

Une socket définit un canal entre deux programmes : client et serveur. Le premier est l'initiateur de la communication à travers l'envoi d'un message (plus exactement d'une requête) vers le serveur. Le serveur, quant-à-lui, est bloqué en attente d'une requête.



Client crée une socket avec (*adrIP*, *numPort*)

Un socket (en Java) définit deux flux : un flux de sortie (outputStream) et un flux d'entrée (inputStream).
- Clinet utilise OS pour envoyer la requête et le IS pour recevoir la réponse
- Serveur utilise le IS pour recevoir la requête et OS pour la réponse.

Serveur écoute un port : *numPort*. Et s'exécute sur une machine dont l'@IP est *adrIP*.

Dans un vrai programme de client/serveur, le serveur s'exécute continuellement (processus démon).

Le premier programme

Le premier programme est relativement simple, il y a deux classes : *MainServer.java* et *MainClient.java*. (remarquer qu'elles sont séparées dans deux packages différents).

MainServer.java

Le serveur suit les étapes suivante :

- Création d'une instance de la classe SocketServer :

```
ServerSocket serverSocket = new ServerSocket(7877);
Socket socket = serverSocket.accept(); // blocage en attente d'une requête
```

- Une foi la socket est reçue, le serveur continue son exécution avec la création d'un reader et d'un writer (reader de flux et writer de flux).

```
BufferedReader reader = new BufferedReader(new
            InputStreamReader(socket.getInputStream()));
PrintWriter writer = new PrintWriter(new BufferedWriter(new
            OutputStreamWriter(socket.getOutputStream())), true);
```

- Le serveur utilise, respectivement, le reader pour récupérer la requête du

client et le writer pour envoyer la réponse.

- En fin le serveur ferme les reader, writer et le socket

MainClient.java

Le serveur suit les étapes suivante :

- Création d'une instance de la classe SocketServer :

```
ServerSocket serverSocket = new ServerSocket(7877);
Socket socket = serverSocket.accept(); // blocage en attente d'une requête
```

- Une foi la socket est reçue, le serveur continue son exécution avec la création d'un reader et d'un writer (reader de flux et writer de flux).

```
BufferedReader reader = new BufferedReader(new
            InputStreamReader(socket.getInputStream()));
PrintWriter writer = new PrintWriter(new BufferedWriter(new
            OutputStreamWriter(socket.getOutputStream())), true);
```

- Le serveur utilise, respectivement, le reader pour récupérer la requête du

Le deuxième programme

Dans le premier exemple, le serveur est très simple (pas d'exécution continuelle et réponse construite par le serveur lui même), dans le second exemple le serveur réalise une boucle infinie dans la quelle (dans chaque itération) : il attend une socket, il crée et lance un thread de réponse et re-boucle une autre fois (et ainsi de suite). La réponse de la socket reçue est réalisée par le thread. (Voir le T.P. pour plus d'explications)

Récapitulatif

Dans cet exemple, on a vu le modèle de communication à travers les sockets. Ce modèle nous permet de programmer très facilement les application avec une architecture Client/Serveur.

Par contre, dans le T.P. du systèmes distribués, tous les processus peuvent envoyer des requêtes/réponses et/ou recevoir des réponses/requêtes. Ainsi, les deux exemples vues ci-dessus ne sont pas adaptés pour les processus du systèmes distribués (chaque processus peut être serveur et client en même temps : peer-to-peer).

La question qui se pose : *Comment utiliser les sockets pour permettre des processus clients et serveurs en même temps ?*