COURS D'APPLICATIONS DISTRIBUEES

Par Ir. Michée TSHIMANGA

Chapitre 3 :
Les sockets et la programmation réseaux

Principes et applications en Java

Introduction

- Les ***sockets***, ou connecteurs d'interface, constituent un mécanisme universel de bas niveau, utilisable depuis tout langage de programmation. L'objet de ce chapitre sera donc de voir les principes généraux, indépendants du langage, qui régissent les *sockets*.

Principe de sockets appliqué au modèle Client-serveur

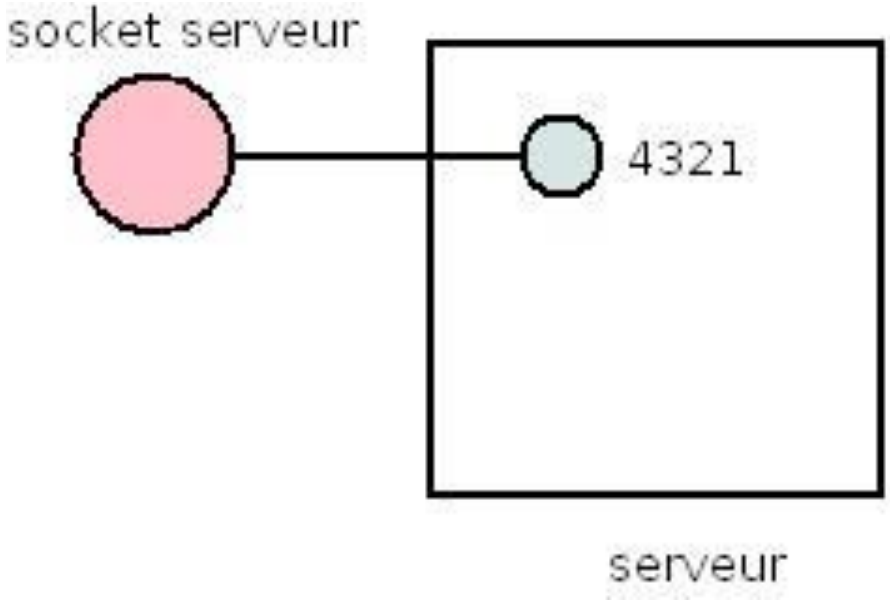
- Tout d'abord, les ***sockets*** ont été créés pour répondre à un besoin : disposer d'un mécanisme de communication exploitant l'interface de transport (TCP-UDP). Introduit dans Unix dans les années **80**, ce mécanisme a ensuite été standardisé.
- La définition d'un ***socket*** peut être la suivante : **un connecteur virtuel défini par un port et créé par un programme pour permettre d'établir une connexion vers un autre *socket***. Dans une connexion entre deux programmes par des *sockets*, il y a un serveur et un client (même si on peut étendre ce cas à d'autres modèles répartis).

- Il existe **trois** types de *sockets* :
  - Côté serveur, le « ***socket* serveur** » qui attend une connexion ;
  - Côté client, le « ***socket client*** » qui tente de se connecter à un *socket* serveur ;
  - Côté serveur, le « ***socket service client*** » qui se substitue au *socket* serveur lorsque la connexion est établie pour le laisser libre.

- La connexion se passe en trois étapes comme dans cet exemple avec TCP :

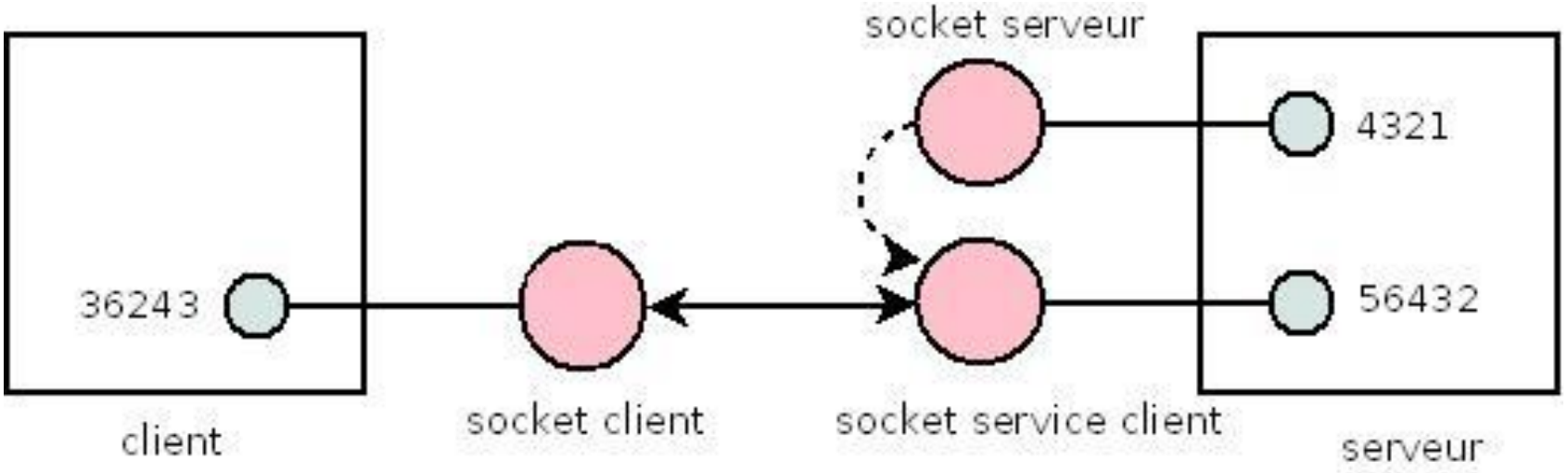
Etape 1 :

- Le serveur crée un *socket* serveur (associé au port 4321) et se met en attente.



Etape 2 :

- Le client se connecte alors au *socket* serveur ; deux *sockets* sont alors créés, un *socket* client (côté client) et un *socket* service client (côté serveur). Ces *sockets* sont interconnectés.



Etape 3 :

- Le **client** et le **serveur** communiquent par les *sockets*. Le *socket* serveur peut accepter de nouvelles connexions.

- Il existe deux modes, dépendant du protocole utilisé : le **mode connecté** avec **TCP** et le **mode non connecté** avec **UDP**. Malgré des différences notables, il y a tout de même des points communs aux deux protocoles : *le client possède l'initiative de la communication (le serveur est à l'écoute), le client doit donc connaître la référence du serveur [adresse IP, n° de port] (il peut la trouver dans un annuaire ou la connaître par convention s'il y a des pré-affectations) et le serveur peut servir plusieurs clients (1 processus unique ou un processus par client).*

Mode connecté avec TCP et Java

- Si on utilise le protocole TCP (*Transmission Control Protocol*), on se trouve en mode connecté (caractéristique de TCP), c'est-à-dire qu'on ***ouvre une liaison***, les **échanges se font**, puis on **ferme la liaison**.

En voici les **caractéristiques** :

- Etablissement préalable d'une connexion (circuit virtuel) : le client demande au serveur s'il accepte la connexion ;
- Fiabilité assurée par le protocole (TCP) ;
- Mode d'échange par flot d'octets : le récepteur n'a pas connaissance du découpage des données effectué par l'émetteur ;

- Possibilité d'émettre et de recevoir des caractères urgents (OOB : Out OF Band) ;
- Après initialisation, le serveur est "**passif**" : il est activé lors de l'arrivée d'une demande de connexion du client ;
- Un serveur peut répondre aux demandes de service de plusieurs clients : les requêtes arrivées et non traitées sont stockées dans une file d'attente.

- Il existe deux modes de gestion des requêtes, le premier, itératif, consiste à traiter les requêtes les unes après les autres ; le second, concurrent, se fait par création de processus fils pour les échanges de chaque requête.
- Nous allons voir maintenant comment programmer des *sockets* avec Java. La première chose à savoir, c'est que la plupart des systèmes d'exploitation possèdent des primitives (appel système) qui permettent la gestion des *sockets*, autrement dit on peut tout à fait utiliser ces primitives depuis un programme pour manipuler les *sockets*. Cependant, la plupart des langages de haut niveau (Java, C, etc…) offrent des outils pour faciliter les usages.

En Java SE (édition standard), ce sont deux classes de base qui permettent l'utilisation de manière simplifiée les *sockets* en mode connecté :

- **ServerSocket** : *socket* côté serveur, attend les connexions et crée le *socket* service client ;
- **Socket** : *sockets* ordinaires, pour les échanges, fournit des classes **InputStream** et **OutputStream** (flots d'octets) pour échanger de l'information.

Exemple (TCP-Java)

- Nous venons donc de voir le code d'un serveur itératif : une seule requête est traitée à la fois, les autres sont conservées dans une file d'attente (associée au *socket* serveur) en attendant d'être traitées. La longueur maximale de la file d'attente peut être spécifiée lors de la création du *socket* serveur, la valeur par défaut est 50.

- On peut aussi utiliser un ***thread*** veilleur qui crée explicitement un nouveau *thread* exécutant qui à chaque nouvelle connexion d'un client (c'est-à-dire à l'exécution de la méthode accept). Un *socket* service client est créé pour chaque client, puis le *thread* veilleur se remet en attente. Côté client, c'est transparent, le code restant inchangé.

Mode non connecté avec UDP - Java

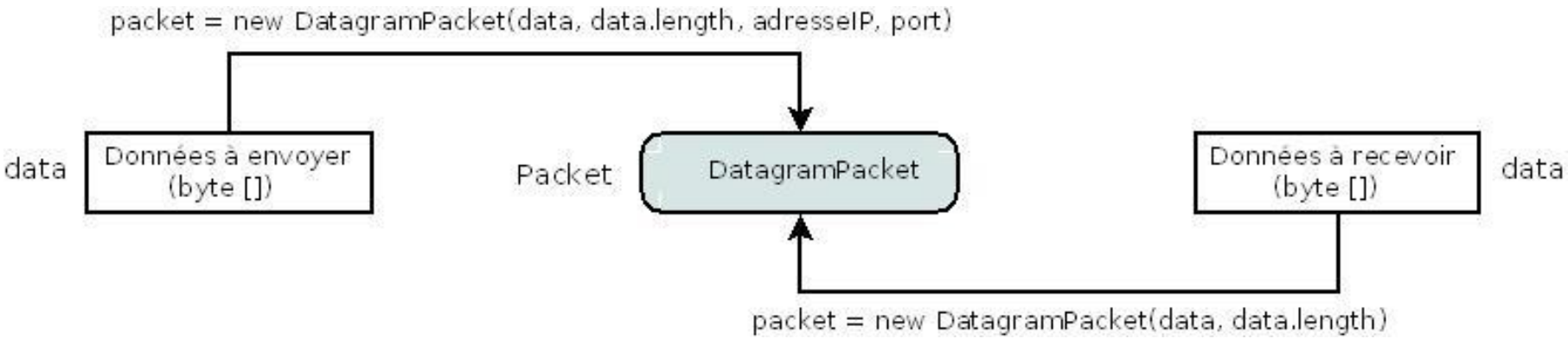
L'autre protocole, UDP (*User Datagram Protocol*), fonctionne en mode non connecté, c'est-à-dire que les requêtes successives sont indépendantes. En voici les caractéristiques :

- Pas d'établissement préalable d'une connexion ;
- Pas de garantie de fiabilité ;
- Adapté aux applications pour lesquelles les réponses aux requêtes des clients sont courtes (1 message) ;

- Le récepteur reçoit les données selon le découpage effectué par l'émetteur ;
- Le serveur doit récupérer l'adresse de chaque client pour lui répondre.

- De même qu'avec TCP, il existe deux modes de gestion des requêtes, le mode **itératif** et le mode **concurrent**.
- Avec le mode non connecté, le client peut envoyer une requête au serveur à tout moment sans demander une connexion, ce qui entraîne un traitement non ordonné des requêtes et l'absence de mémorisation entre deux appels successifs. Un exemple d'appel dans ce mode est celui adressé au DNS (Domaine Name Server) puisque deux appels sont indépendants l'un de l'autre.

- De même que pour les *sockets* en mode connecté avec TCP, il existe des primitives pour la plupart des systèmes d'exploitation. Et, également, il existe des outils fournis par les langages de haut niveau, dont Java.
- En Java SE, ce sont deux classes de base qui permettent l'utilisation de manière simplifiée les *sockets* en mode non connecté :
- **DatagramSocket** : un seul type de *socket* ;
- **DatagramPacket** : format de message pour UDP.


packet = new DatagramPacket(data, data.length, adresseIP, port)
data
Données à envoyer
(byte [])
Packet
DatagramPacket
Données à recevoir
(byte [])
data
packet = new DatagramPacket(data, data.length)

- On peut voir ce mécanisme comme un échange de questions-réponses. Les messages sont brefs, l'envoi se fait en *streaming* (temps réel).
- Voici maintenant un exemple de programme itératif qui utilise les *sockets* en mode non connecté. Il est bien sûr simplifié, pour ne garder que la partie essentielle.

Exemple UDP-JAVA

- À l’instar de ce qui a été vu avec le mode connecté, nous pouvons aussi imaginer un serveur non pas itératif comme dans cet exemple mais concurrent, en utilisant toujours le principe d’un processus veilleur qui crée selon les besoins des processus exécutants.

EXERCICES D’APPLICATION