

Une application Client-Serveur : *Sockets*

1.	UNIX : ENTREES/SORTIES ET SOCKETS	2
1.1	Communication fiable	4
1.2	Communication non fiable de paquets	5
1.3	Internet et les Sockets	6
2.	JAVA, LE MODELE CLIENT-SERVEUR ET LES SOCKETS	7
2.1	Principe d'utilisation des sockets TCP en Java	8
2.2	Principe d'utilisation des sockets UDP en Java	14
3.	ILLUSTRATION : LE PROGRAMME « CHAT » (<i>SOURCES A DISPOSITION !!</i>)	16
3.1	Interface utilisateur	17
3.2	Test de l'application	19
3.3	Protocole Client ⇔ Serveur	21
3.4	Le « Client » : diagramme des états et transitions	23
3.5	Le « Serveur » : diagramme des états et transitions	24
3.6	Risque d'interblocage	25
4.	ANNEXES	26
4.1	Classe « <i>Socket</i> » (<i>non exhaustif</i>)	27
4.2	Classe « <i>ServerSocket</i> » (<i>non exhaustif</i>)	28
4.3	Classe « <i>InetAddress</i> » (<i>non exhaustif</i>)	29

1. UNIX : Entrées/Sorties et sockets

Sous UNIX, les périphériques d'entrées/sorties sont assimilés à des *fichiers spéciaux*.

Par exemple, l'imprimante sera assimilée au fichier spécial «**lp**», affecté à un chemin, situé très souvent dans le catalogue **/dev**.

<code>/dev/lp</code>	→ imprimante
<code>/dev/tty</code>	→ terminal
<code>/dev/net</code>	→ réseau

Les fichiers spéciaux

- Accessibles de la même manière que les autres fichiers
 - ☞ commandes **Open**, **Close**, **Read** et **Write**
- **Premier avantage** : pas besoin de commandes spéciales.
Par exemple, pour imprimer un fichier : `>cp fichier /dev/lp`
- **Deuxième avantage** : les règles de protection des fichiers s'appliquent à la protection des périphériques.
Par exemple, pour interdire l'accès de l'imprimante : interdire l'accès en écriture
- **Troisième avantage** : indépendance des programmes vis à vis du matériel.

Les « sockets »

Historique

- Mécanisme apparu dans le système UNIX de Berkeley,
- A l'origine, les *sockets* sont prévues pour mettre en oeuvre la communication entre terminaux,
- Aujourd'hui, utilisées principalement pour permettre à deux *processus* situés sur des machines distantes de communiquer au travers du réseau.

Manipulation

Au même titre que les périphériques, les sockets sont manipulées comme des *fichiers spéciaux*

- 1) Les sockets peuvent être créées ou détruites dynamiquement
- 2) La création d'une socket retourne un descripteur de fichier.

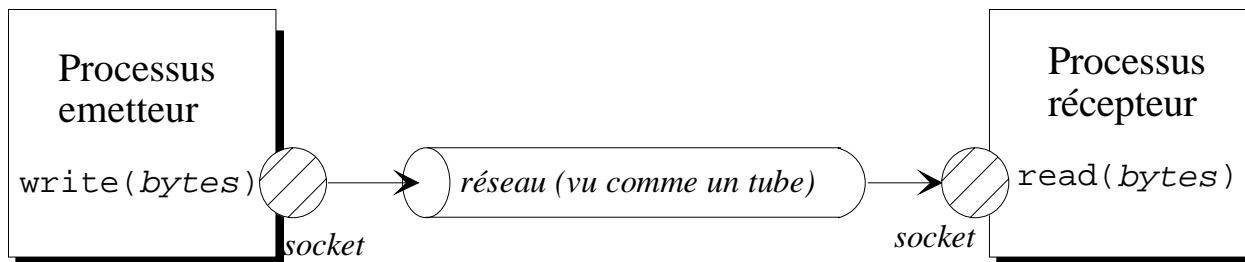
Ce même descripteur sera utilisé ensuite pour créer la connexion («[open](#)»), lire et écrire des données («[write](#)», «[read](#)»), puis enfin pour libérer la connexion («[close](#)»).

Deux types de sockets

- **communication fiable**, orientée « connexion »
- **communication de paquets**, non fiable, mais plus efficace

1.1 Communication fiable

Un « **tube** » (*pipe*) est établi entre deux processus distants.



Communication de type « **Stream** » (« flot ») :

- 1) Le processus émetteur écrit dans le tube comme s'il écrivait dans un fichier
 - 2) Le récepteur lit les données du tube comme s'il lisait dans un fichier.
-
- Les octets qui sont émis à une extrémité sont reçus **dans l'ordre** à l'autre extrémité.
 - Tout byte, émis par le processus émetteur au moyen d'un appel à la primitive «`write`», ne sera envoyé que lorsque le processus récepteur aura fait appel à un «`read`» correspondant.
 - En principe, les deux opérations «`write`» et «`read`» sont **bloquantes**, à moins que le tube ne dispose d'un **tampon**, ce qui permettrait de libérer partiellement le processus émetteur.

1.2 Communication non fiable de paquets

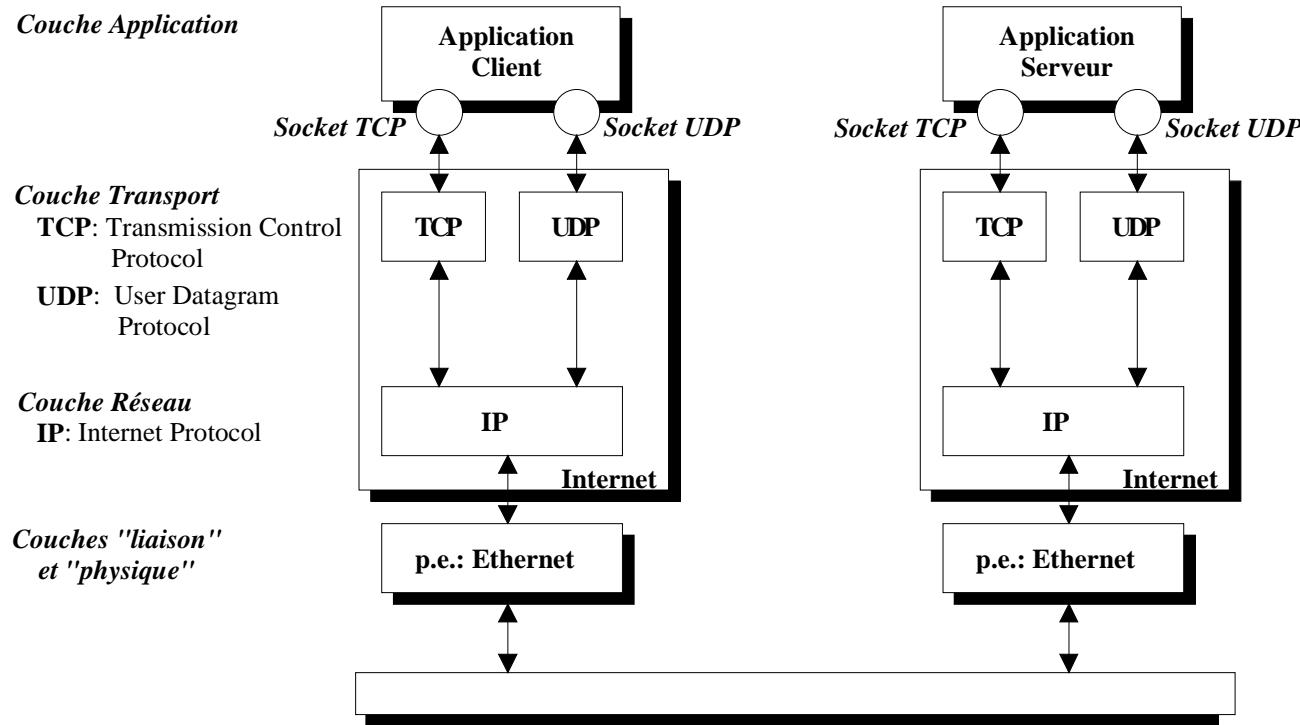
- La connexion n'est pas établie une fois pour toutes,
 - ☞ les paquets peuvent emprunter des chemins différents
 - ☞ les paquets **ne seront pas forcément reçus dans l'ordre d'envoi**
- Problème de transmission ☞ **le paquet n'est pas renvoyé**

Avantage

- Communication plus efficace, utilisée par exemple dans les **applications temps réel**,
- Mais de bas niveau !
C'est à l'application doit mettre en oeuvre un mécanisme de gestion d'erreurs spécifique.

1.3 Internet et les Sockets

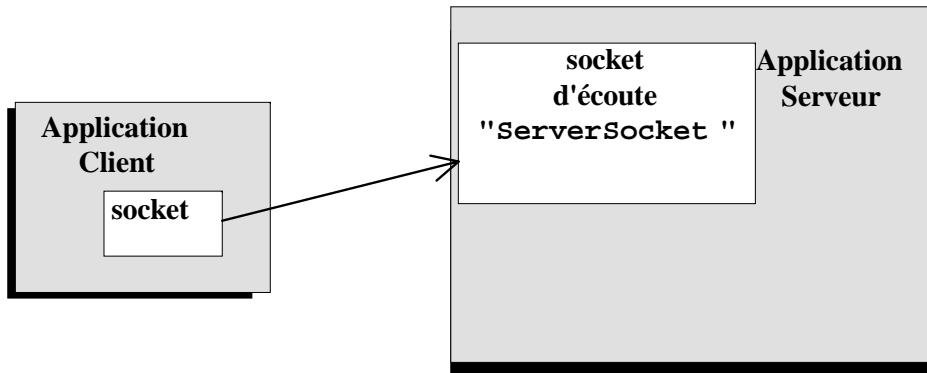
Se basant sur le protocole IP ↗ au choix : connexions TCP ou liaisons UDP



- **Protocole TCP :** communication de type connexion, fiable ...
- **Protocole UDP :** communication par paquets (*datagrammes*), non fiable, pas de connexion
Temps-réel, transmission d'images, vidéo-clips,
SNMP (Simple Network Management protocol)

2. Java, le modèle Client-Serveur et les sockets

- Le **serveur** propose des services
- Le **client** utilise les services offerts par le serveur



Réalisation en Java

- **Sockets** (mécanisme de plus bas niveau) : paquetage **java.net** (**J2SE**)

Ou, en se basant sur les sockets (J2EE)

- RMI (invocation de méthodes à distance), client et serveur écrits tous deux en Java
- CORBA (invocation de méthodes à distance), client et serveur peuvent être hétérogènes

Ou encore, basé sur http, lui-même basé sur les sockets (J2EE)

- JSP/Servlet (html dynamique)
- WEB Services, sur SOAP/http

Ou encore, basé sur RMI/CORBA (J2EE)

- EJB

2.1 Principe d'utilisation des sockets TCP en Java

Le serveur sera caractérisé par deux éléments :

- 1) Une **adresse internet** (adresse IP), désignant la machine sur laquelle s'exécute l'application serveur
- 2) Un **numéro de port**, qui désigne le numéro du service offert par le serveur

□ Du côté « Client »

On peut assimiler la socket à un appareil téléphonique, la création d'une socket revient à installer un appareil téléphonique, et à composer le numéro du serveur.

```
try {  
    socket = new Socket (adresseIP, NO_PORT);  
}  
catch (IOException e) { /* !! PROBLEME DE CONNEXION !! */ }
```

Si le serveur (au bout du fil) décroche son combiné, la communication pourra commencer...

Exception «[IOException](#)» : si le serveur n'est pas libre, ou surchargé, ou autre...

□ Du côté « Serveur »

Ouvrir une connexion en permanence : créer une socket « d'écoute » de type «`ServerSocket` »:

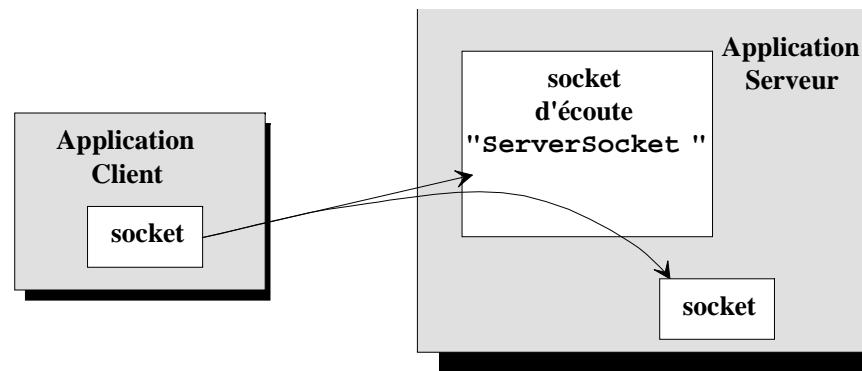
```
try {  
    socketEcouteur = new ServerSocket (NO_PORT);  
}  
catch (IOException e) { /* !! PROBLEME DE CONNEXION ! ! */ }
```

Mise « à l'écoute » : en envoyant le message «`accept()`» :

```
Socket nouvelleSocket = socketEcouteur.accept();
```

- Le serveur « à l'écoute » est un **thread à l'état bloqué** qui attend que le téléphone sonne.
- Si le téléphone sonne et qu'il n'a pas d'autre chose à faire, le serveur décroche pour commencer la communication.
- Le serveur agit plutôt comme un **central téléphonique** : la communication acceptée est déviée sur un nouvel appareil téléphonique ↗ le serveur peut se remettre à l'écoute du premier appareil et attendre une nouvelle sonnerie.

Un « `ServerSocket` » peut admettre jusqu'à 50 demandes de connexions simultanées. Si une connexion est demandée alors que la queue est pleine, cette dernière sera refusée.



■ Construire l'adresse IP

En Java, une adresse IP est un objet de type «**InetAddress**» (paquetage java.net):

```
public static InetAddress getByName(String host) throws UnknownHostException
```

☞ Retourne l'adresse IP de la machine distante

- l'adresse IP peut être spécifiée sous la forme "aaa.bbb.ccc.ddd"
- ou alors par un nom composé : « *machine.sous-domaine.domaine* » ("java.sun.com") en utilisant le **système de noms (DNS, Data Name System)** qui associe les numéros IP à des noms symboliques par le biais de serveurs de noms répartis sur le réseau.

```
public static InetAddress getLocalHost() throws UnknownHostException
```

☞ Retourne l'adresse IP de la machine locale

Utilisation

```
InetAddress adresseIP;  
try {  
    //Adresse du serveur WinCenter de l'Ecole d'Ingénieurs de Genève :  
    adresseIP = InetAddress.getByName("129.194.186.35");  
  
    //ou : travail en mode local (le serveur = machine locale)  
    adresseIP = InetAddress.getLocalHost();  
    //identique à : adresseIP = InetAddress.getByName("127.0.0.1");  
  
}  
catch (UnknownHostException e) { /*adresse inconnue sur le réseau */ }
```

■ Fermeture de la connexion

Connexion fermée dès que l'un **ou** l'autre des deux partenaires ferme la socket:

```
uneSocket.close();
```

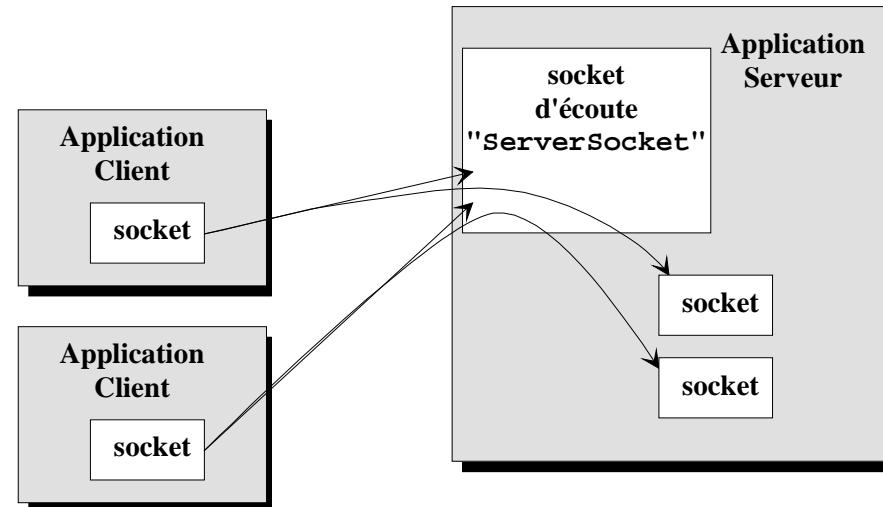
Dès que la connexion est fermée, toute tentative d'accès à la socket se soldera par une levée d'exception de type «**IOException**».

■ Ouverture simultanée de plusieurs connexions

On parle alors de « **serveur concurrent** ».

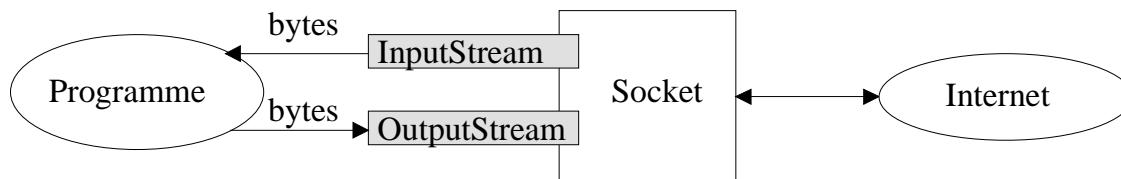
Sous UNIX, ceci est réalisé en créant des processus enfant (en C) avec la fonction `fork()`.

Chaque connexion est ensuite confiée à un processus enfant qui implémente un thread indépendant.



■ Transmettre et recevoir des informations au travers d'une socket

Deux flux de données primaires à disposition: « `InputStream` » & « `OutputStream` » (paquetage `java.io`)



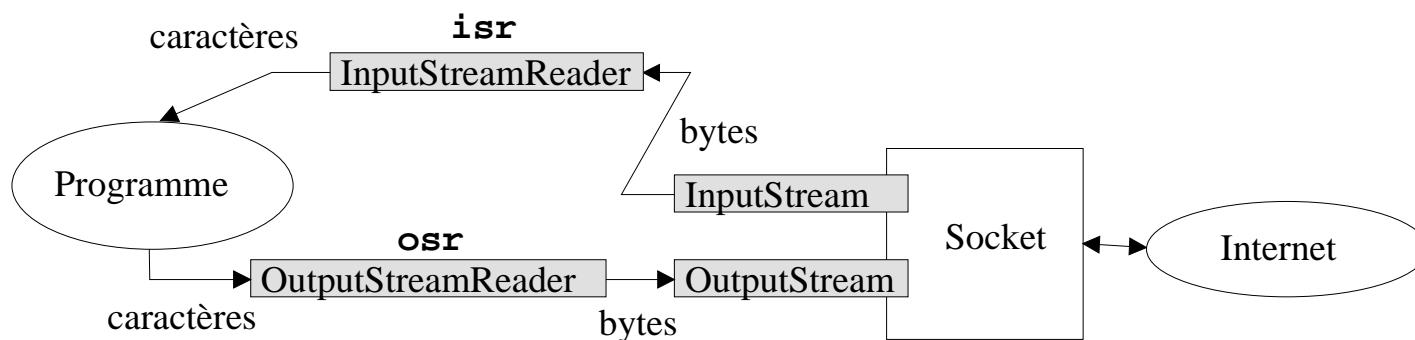
Pour y accéder, la classe «`Socket`» propose les méthodes :

- `InputStream getInputStream()`: pour retourner le flux d'entrée
- `OutputStream getOutputStream()`: pour retourner le flux de sortie

Comme il s'agit de flux de bytes non bufferisés, le programmeur à tout intérêt à travailler de manière plus évoluée en ouvrant des « *flux de traitement* » (flux de deuxième niveau) pour manipuler ces deux flux primaires.

Par exemple, pour convertir les bytes en caractères:

```
InputStreamReader isr = new InputStreamReader(socket.getInputStream())
OutputStreamWriter osr = new OutputStreamWriter(socket.getOutputStream())
```



- Pour **envoyer des informations**, le plus pratique est d'ouvrir un nouveau flux de traitement de type «`PrintWriter`»:

```
PrintWriter pw = new PrintWriter (socket.getOutputStream());
```

☞ textes, types primitifs, et objets sérialisés.

- Pour **lire des informations**, l'équivalent du flux «`PrintWriter`» n'existe pas.

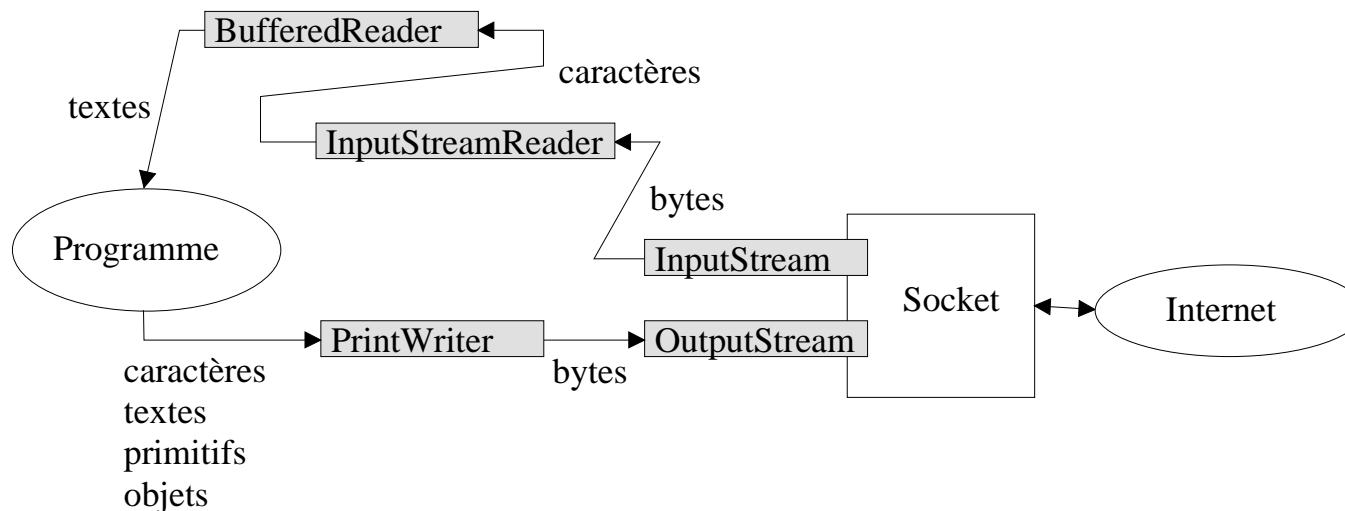
☞ Pour la lecture de types primitifs (entiers, réels, ..),

```
DataInputStream dis = new DataInputStream(socket.getInputStream());
```

☞ Pour la lecture de textes :

```
BufferedReader bfr =  
new BufferedReader (new InputStreamReader(socket.getInputStream()));
```

Exemple : une configuration qui permettrait de communiquer des informations de type « Texte » :



2.2 Principe d'utilisation des sockets UDP en Java

Non traité en détails.. Utiliser principalement les classes **DatagramSocket** et **DatagramPacket** de **java.net**

■ Classe **DatagramSocket** (*non exhaustif*)

This class represents a socket for sending and receiving datagram packets.

A datagram socket is the sending or receiving point for a packet delivery service. Each packet sent or received on a datagram socket is individually addressed and routed. Multiple packets sent from one machine to another may be routed differently, and may arrive in any order.

UDP broadcasts sends are always enabled on a DatagramSocket. In order to receive broadcast packets a DatagramSocket should be bound to the wildcard address. In some implementations, broadcast packets may also be received when a DatagramSocket is bound to a more specific address.

Example: `DatagramSocket s = new DatagramSocket(null); s.bind(new InetSocketAddress(8888));` Which is equivalent to: `DatagramSocket s = new DatagramSocket(8888);` Both cases will create a DatagramSocket able to receive broadcasts on UDP port 8888.

DatagramSocket(int port) throws SocketException

Constructs a datagram socket and binds it to the specified port on the local host machine.

DatagramSocket(int port, InetAddress laddr) throws SocketException

Creates a datagram socket, bound to the specified local address.

public void receive(DatagramPacket p) throws IOException

Receives a datagram packet from this socket. When this method returns, the DatagramPacket's buffer is filled with the data received. The datagram packet also contains the sender's IP address, and the port number on the sender's machine.

public void send(DatagramPacket p) throws IOException

Sends a datagram packet from this socket. The DatagramPacket includes information indicating the data to be sent, its length, the IP address of the remote host, and the port number on the remote host.

■ Classe DatagramPacket (*non exhaustif*)

This class represents a datagram packet.

Datagram packets are used to implement a connectionless packet delivery service. Each message is routed from one machine to another based solely on information contained within that packet. Multiple packets sent from one machine to another might be routed differently, and might arrive in any order. Packet delivery is not guaranteed.

DatagramPacket(byte[] buf, int length)

Constructs a DatagramPacket for receiving packets of length length.

DatagramPacket(byte[] buf, int length, InetAddress address, int port)

Constructs a datagram packet for sending packets of length length to the specified port number on the specified host.

public void setData(byte[] buf, int offset, int length)

Set the data buffer for this packet.

public void setAddress(InetAddress iaddr)

Sets the IP address of the machine to which this datagram is being sent.

public void setPort(int iport)

Sets the port number on the remote host to which this datagram is being sent

public InetAddress getAddress()

Returns the IP address of the machine to which this datagram is being sent or from which the datagram was received.

public int getPort()

Returns the port number on the remote host to which this datagram is being sent or from which the datagram was received.

byte [] getData()

Returns the data buffer.

public int getLength()

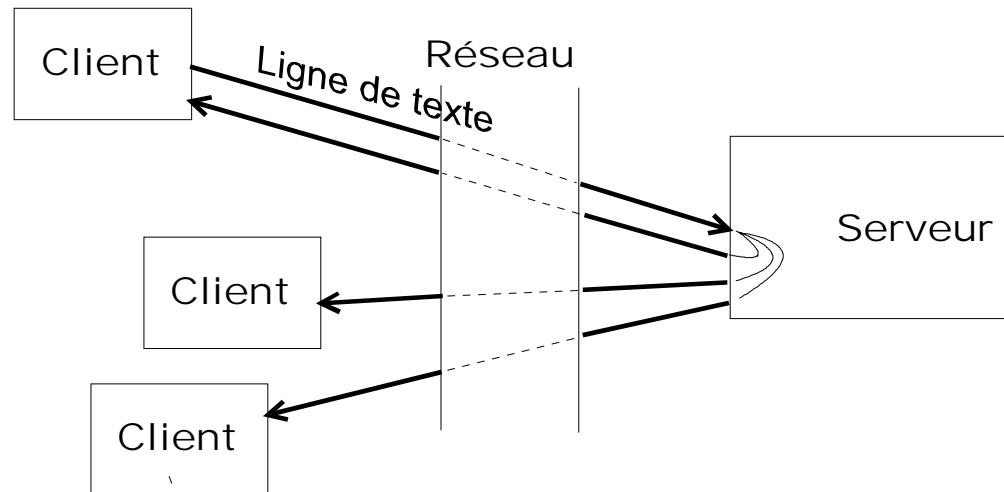
Returns the length of the data to be sent or the length of the data received.

public int getOffset()

Returns the offset of the data to be sent or the offset of the data received.

3. Illustration : le programme « chat » (*Sources à disposition !!*)

Dès qu'un participant (un « client ») envoie un message au « serveur », ce dernier le renvoie à tous les participants à la discussion.

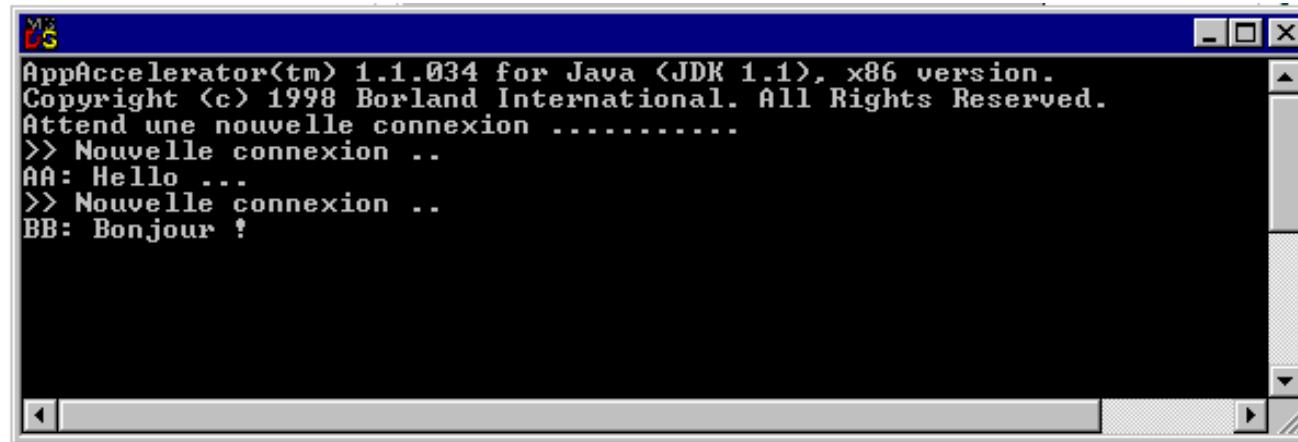


- Le serveur maintient une liste de connexions avec des clients.
Cette liste est mise à jour dès qu'un nouveau client se connecte (ou se déconnecte)
- Les textes renvoyés sont identifiés par le sigle du client

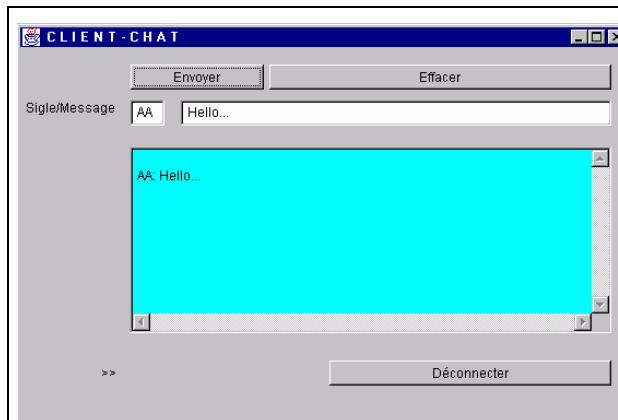
3.1 Interface utilisateur

■ Côté « Serveur »

- Du côté serveur, l'application Java s'exécute dans une fenêtre « DOS » - sur Windows NT -, ou dans une fenêtre de commande de type « Shelltool » - sur une station UNIX de type SUN-Sparc.
- Une fois le serveur lancé, il n'est plus possible d'interagir avec lui, sinon pour l'interrompre au moyen d'un « Control-C »
- Le serveur se contente d'afficher des informations d'état : un nouveau client est connecté, un client s'est déconnecté, ou encore de signaler des problèmes de communication.
- Pour contrôle, le serveur affiche en écho toutes les chaînes de caractères reçues en provenance des clients.



■ Côté « Client »



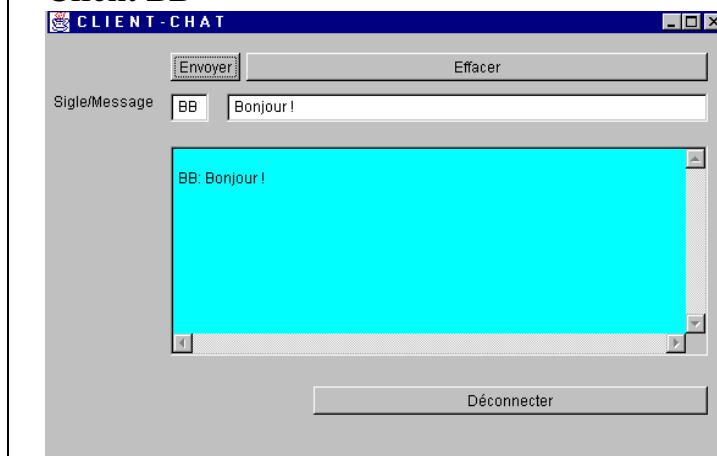
AA: Premier client à s'être connecté avec le serveur.

Il s'est attribué un sigle de reconnaissance : «AA»

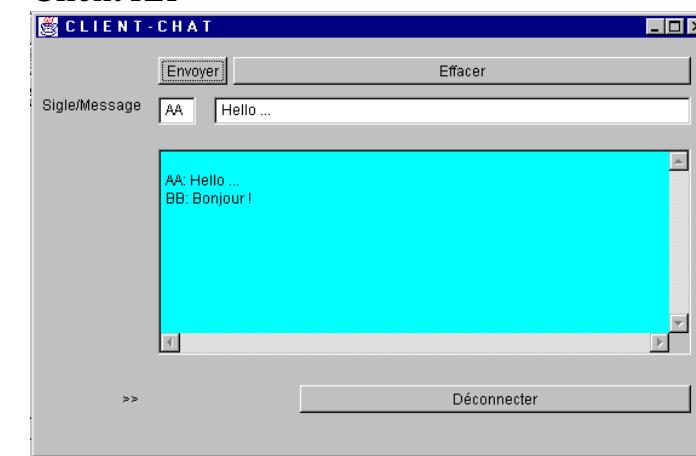
Le serveur lui renvoie en écho tous les textes qu'il envoie.

BB, - un deuxième client -, s'est connecté et discute avec AA

Client BB



Client AA



Le bouton « Connecter-DéConnecter » permet d'ordonner une connexion avec le serveur, ou au contraire d'interrompre la connexion.

3.2 Test de l'application

Le test peut être opéré de deux manières :

1. **Localement** sur une seule et unique station : chaque processus « client » se connecte à une adresse IP correspondant au « Local host » ;

Le serveur est lancé sur la même station.

2. **En mode distant**, en lançant le processus « Serveur » sur une machine dédiée. Les clients utilisent alors l'instruction « **getByName** » en spécifiant l'adresse IP du serveur.

Quelque part dans la classe Client:

```
private boolean connecter () {
    // Retourne "true" si la connexion est établie, "false" sinon
    InetAddress adresseIP;
    try {
        //adresseIP = InetAddress.getByName("129.194.186.35");

        // ou : travail en mode local :
        adresseIP = InetAddress.getLocalHost();    // Serveur NT (local)
    }
    catch (UnknownHostException e) {
        gui.afficherAvertissement
            ("!!! PROBLEME DE CONNEXION !!\n" + e.toString());
        return false;
    }
    //----- Création de la socket
    try {
        socket = new Socket (adresseIP, NO_PORT);
    }
    catch (IOException e) {
        gui.afficherAvertissement ("!!! PROBLEME DE CONNEXION !!\n" + e.toString());
        return false;
    }
    //----- Ouverture des canaux d'entrée et de sortie
    try {
        canalSeSortie = new PrintWriter
            (socket.getOutputStream());

        // Création de l'objet actif en écoute permanente du serveur
        écouteurDuServeur = new ServeurEcouteur(socket.getInputStream(), gui, this);

        return true;      // Connexion OK
    }
    catch (IOException e) {
        try {socket.close(); } catch (IOException ioe) {};
        gui.afficherAvertissement ("!!! PROBLEME DE CONNEXION !!\n" + e.toString());
        return false;
    }
}
```

3.3 Protocole Client ⇔ Serveur

Qui dit « sockets », dit protocole...

L'application Client distingue 4 états:

- 1) Etat “Non connecté”

→ **Etablissement de la connexion**

Le client ouvre une socket qui sera connectée avec une socket correspondant côté serveur

En cas de succès, le client passe à l'état “Connexion en cours”, sinon, reste à l'état “Non connecté”.

- 2) Etat “Connexion en cours”

→ **Etablissement d'une session**

Le client envoie la commande “LOGIN” au serveur, accompagnée de son identificateur personnel.

Le serveur répond par “LOGIN_OK” en cas d'acceptation, par “LOGIN_ERROR” en cas de refus.

En cas de succès, le client passe à l'état “Session en cours”.

En cas de refus, le client démarre une “fermeture de session” (voir étape no 4).

- 3) Etat “Session en cours”

→ **Echange de lignes de texte**

Le client envoie des lignes de texte au serveur. Ce dernier les diffuse à tous les clients connectés, y compris au client qui en est à l'origine.

- 4) Etat “Session en cours”

→ **Fermeture de session**

Le client envoie la commande “LOGOUT” au serveur. Ce dernier ferme alors la socket (côté serveur).

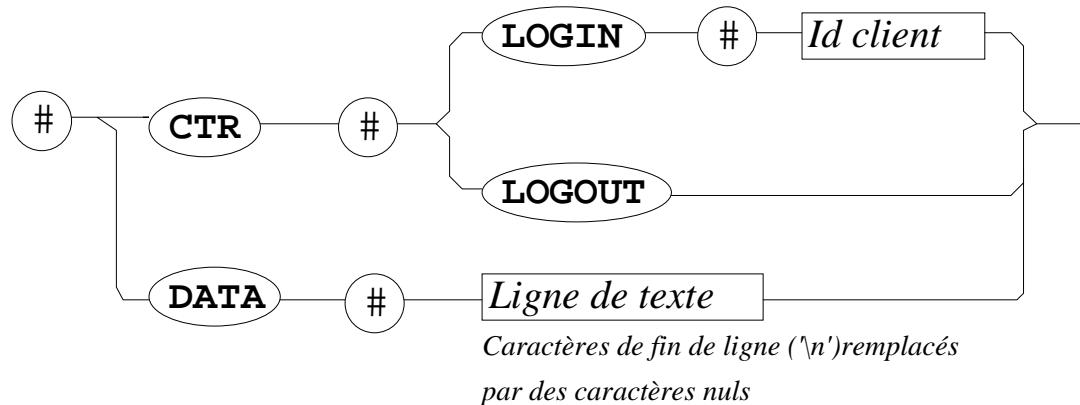
La fermeture de la socket serveur est détectée aussitôt au niveau client: la saisie de la prochaine ligne de texte en provenance du serveur retourne la valeur “null”.

Le client ferme alors sa propre socket, et repasse à l'état “ Non connecté ”

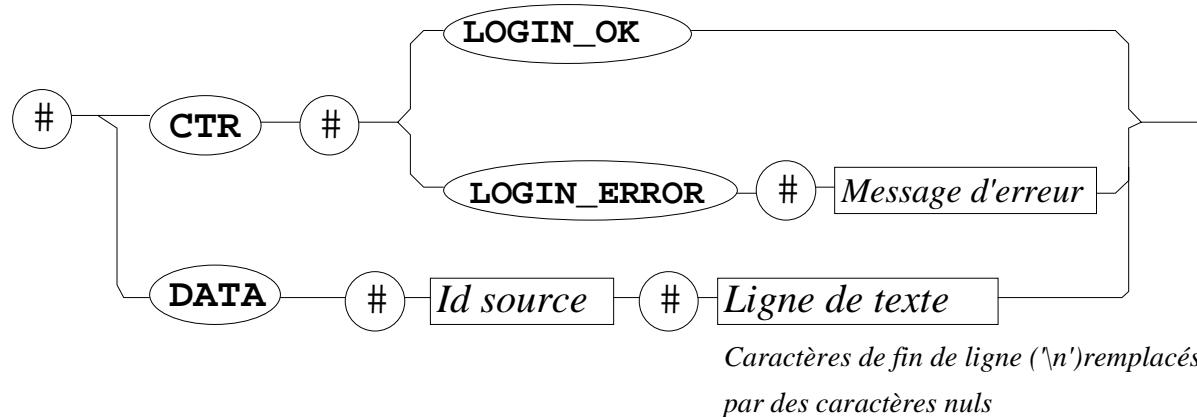
Syntaxe des messages

Deux types de messages : les messages « CTR » pour le contrôle et les messages « DATA » pour l'échange de lignes de textes.

Client > Serveur

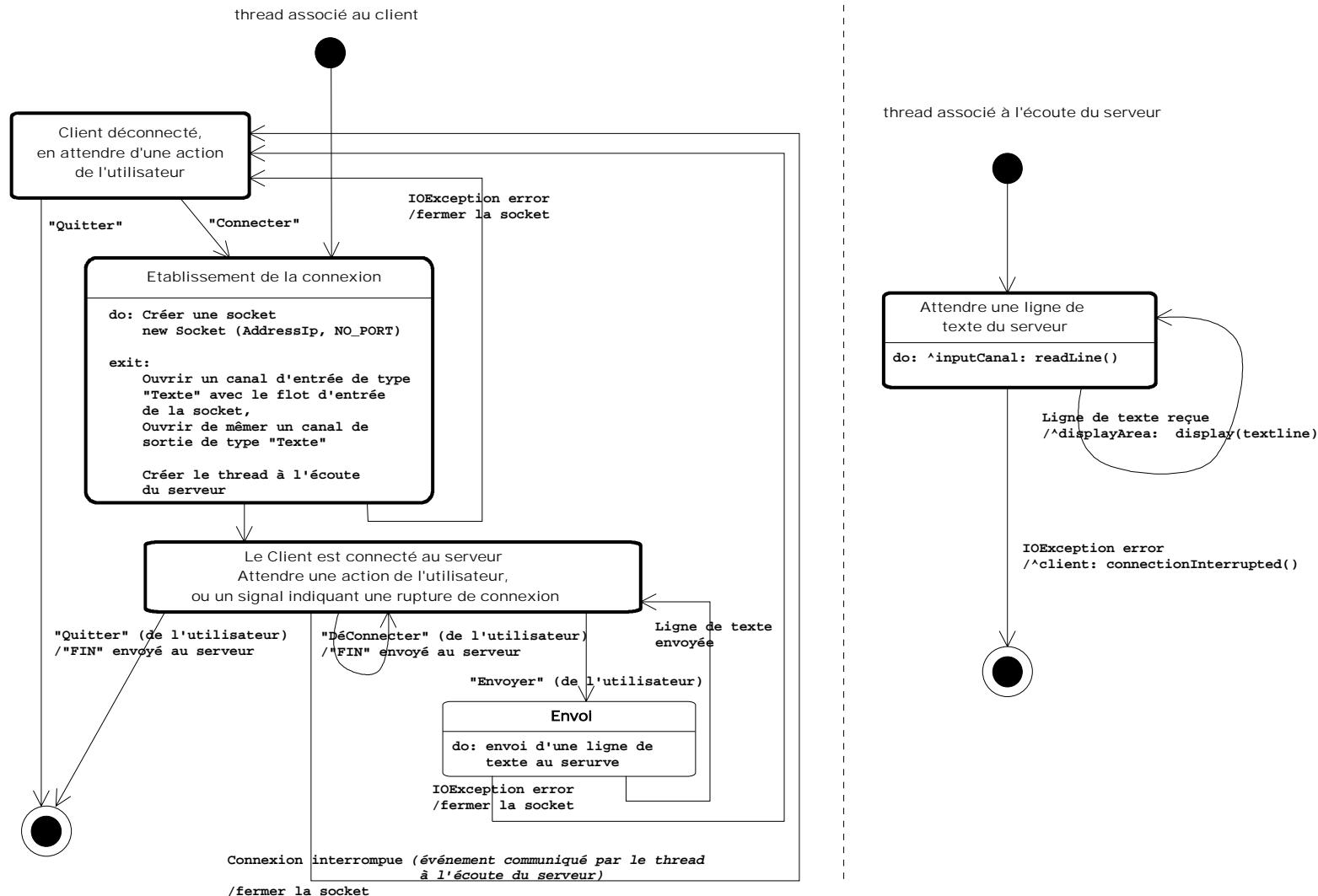


Serveur > Client

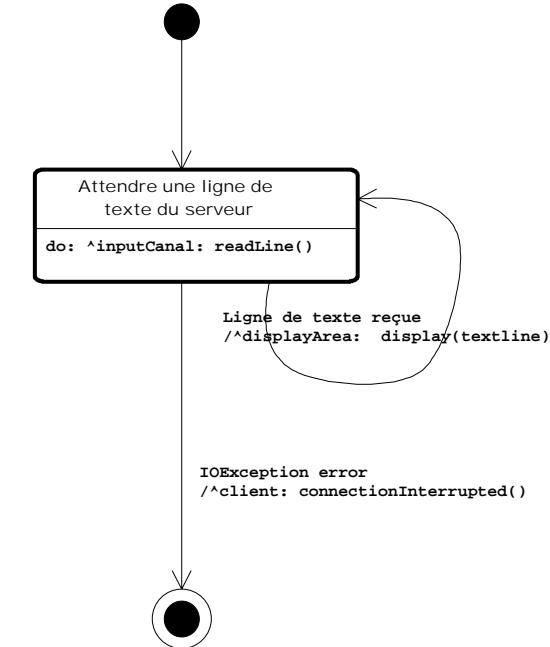


3.4 Le « Client » : diagramme des états et transitions

CLIENT: Diagramme d'états

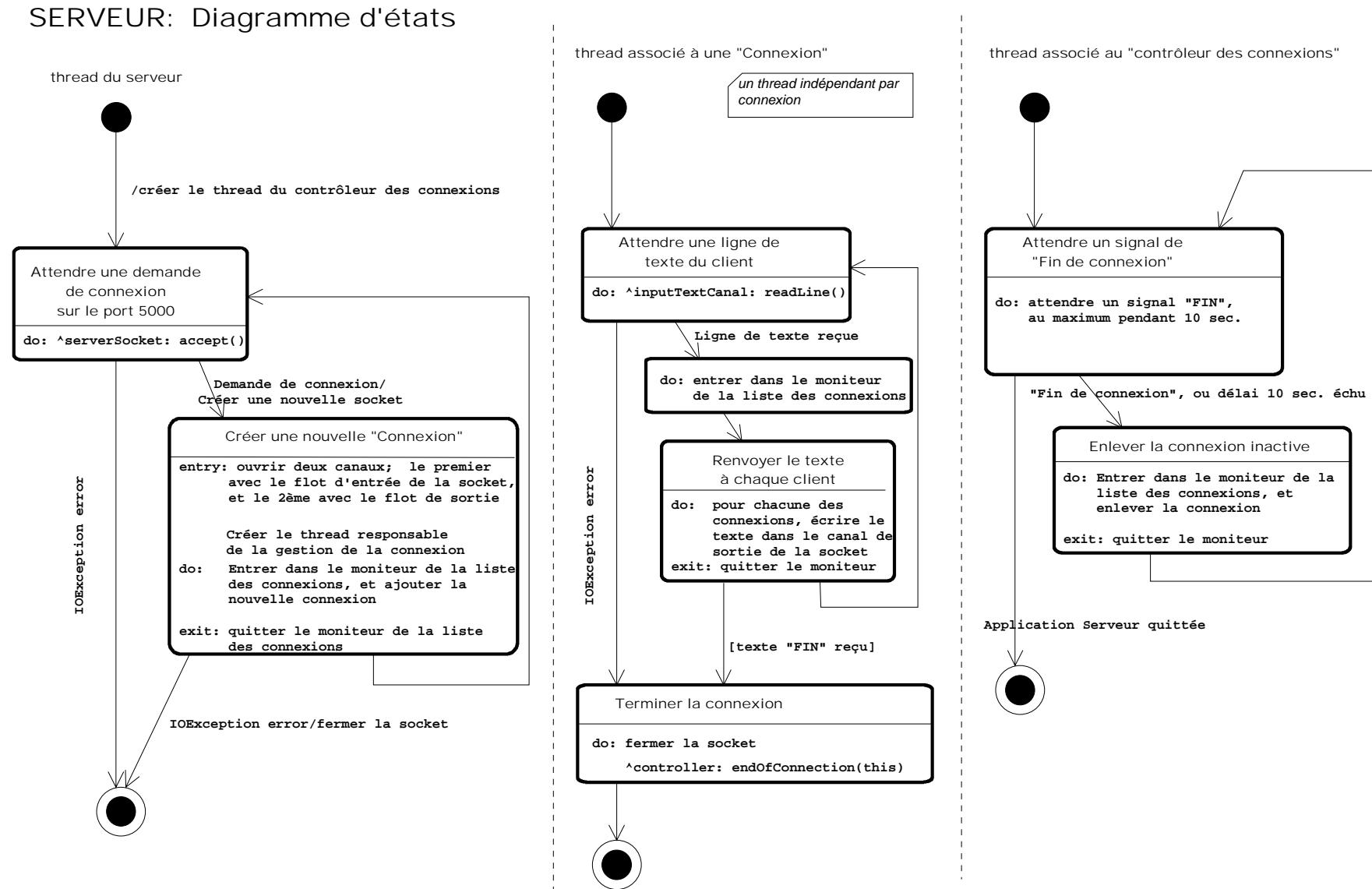


thread associé à l'écoute du serveur



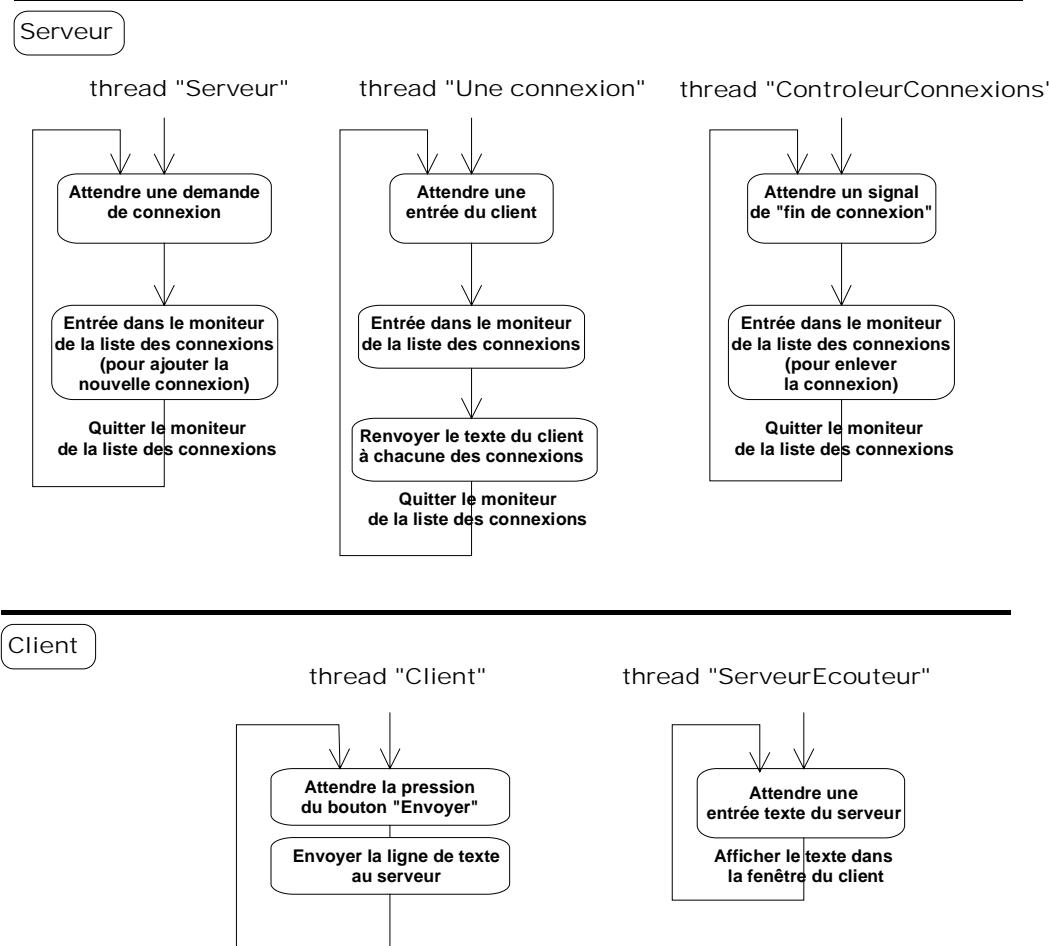
3.5 Le « Serveur » : diagramme des états et transitions

SERVEUR: Diagramme d'états



3.6 Risque d'interblocage

- Le fait que chaque client dispose de deux activités, une pour l'envoi de messages et l'autre pour la réception de messages, facilite l'élimination du risque potentiel d'interblocage.
- Une analyse détaillée de chacun des états bloquants peut montrer qu'aucun d'entre eux n'est bloqué définitivement.



4. Annexes

- Classe “Socket”
- Classe “ServerSocket”
- Classe “InetAddress”

4.1 Classe «Socket» (*non exhaustif*)

- **Socket()** : création d'une *socket* non connectée.
- **Socket(String hote, int port)** : création d'une *socket* connectée à l'hôte en paramètre sur le *port* en paramètre. Peut lever une exception *UnknownHostException* si on ne trouve pas l'hôte, ou une *IOException* s'il n'y a pas de serveur lancé sur le port.
- **Socket(InetAddress hote, int port)** : création d'une *socket* connectée à l'hôte en paramètre sur le port en paramètre. Peut lever une exception *IOException* s'il n'y a pas de serveur lancé sur le port.
- **InetAddress getInetAddress()** : l'adresse de l'hôte auquel la *socket* est connectée.
- **InetAddress getLocalInetAddress()** : l'adresse locale de la *socket*.
- **int getPort()** : le port de l'hôte auquel la *socket* est connectée.
- **int getLocalPort()** : le port local de la *socket*.
- **InputStream getInputStream()** : retourne un *InputStream* pour la *socket*.
- **OutputStream getOutputStream()** : retourne un *OutputStream* pour la *socket*.
- **synchronized void close()** : fermeture de la *socket*.
- **String toString()** : une chaîne représentant la *socket*.

4.2 Classe «**ServerSocket**» (*non exhaustif*)

Cette classe implémente un serveur de *socket*. Un serveur de *socket* attend l'arrivée d'une requête de connexion, et y répond éventuellement en créant une *socket*.

- **ServerSocket(int port)** : crée un serveur de *socket* sur le port en paramètre. Lève éventuellement une *IOException*.
- **ServerSocket(int port, int lq)** : crée un serveur de *socket* sur le port en paramètre, et avec une file d'attente de longueur *lq* (*lq* est à 50 par défaut). Lève éventuellement une *IOException*.
- **ServerSocket(int port, int lq, InetAddress ia)** : crée un serveur de *socket* sur le port en paramètre, et avec une file d'attente de longueur *lq*, et une adresse locale *ia*. Lève éventuellement une *IOException*.
- **Socket accept()** : attend une connexion et l'accepte. Lève éventuellement une *IOException*. La méthode bloque l'exécution du programme jusqu'à ce qu'une connexion soit acceptée.
- **void close()** : ferme la *socket*. Lève éventuellement une *IOException*.
- **InetAddress getInetAddress()** : retourne l'adresse locale du serveur de *socket*.
- **int getLocalPort()** : retourne le port local du serveur de *socket*.

4.3 Classe «**InetAddress**» (*non exhaustif*)

La classe InetAddress représente un nom d'hôte et ses numéros IP :

- **byte[] getAddress()** : retourne le numéro IP sous forme de tableau d'octets.
- **String getHostAddress()** : retourne le numéro IP sous forme de chaîne de caractères.
- **static InetAddress getByName(String hote)** : retourne une *InetAddress* pour un nom d'hôte. Peut lever une exception *UnknownHostException*.
- **static InetAddress getLocalHost()** : retourne une *InetAddress* de la machine locale.