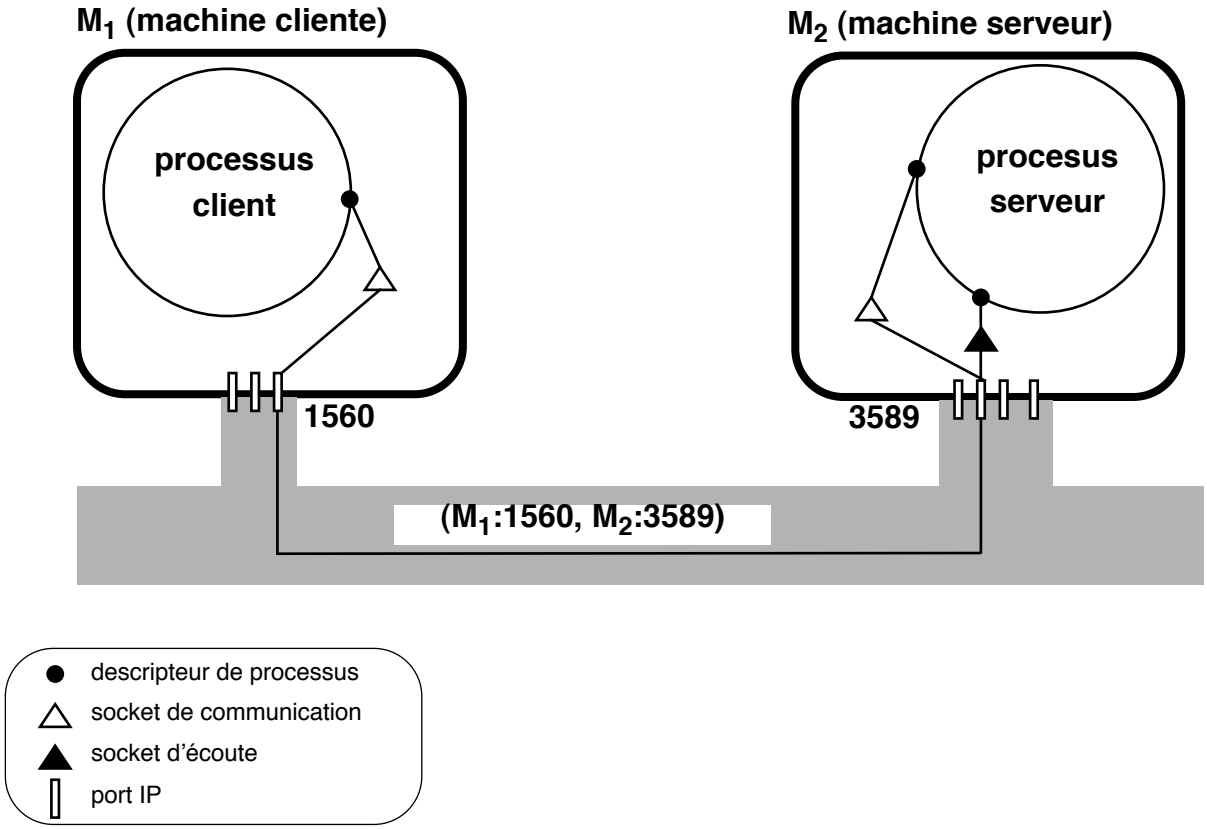
La communication par sockets en Java

L'interface socket BSD est une interface de programmation réseau très répandue. Cette interface est particulièrement adaptée à la réalisation du schéma d'interaction client-serveur. L'interface Java des sockets (package *java.net*) offre un accès simple aux sockets sur IP.



Les classes

Plusieurs classes interviennent lors de la réalisation d'une communication par sockets. La classe *java.net.InetAddress* permet de manipuler des adresses IP. En mode connecté, la classe *java.net.ServerSocket* permet de programmer l'interface côté serveur, tandis que la classe *java.net.Socket* permet de programmer l'interface côté client et la communication effective par flot via les sockets. Les classes *java.net.DatagramSocket* et *java.net.DatagramPacket* permettent de programmer la communication en mode datagramme.

La classe java.net.InetAddress

Cette classe représente les adresses IP et propose un ensemble de méthodes pour les manipuler. Elle encapsule aussi l'accès au serveur de noms DNS.

```
public class InetAddress implements Serializable
```

Les opérations de la classe *InetAddress*

Conversion de nom vers adresse IP (méthodes statiques) :

Un premier ensemble de méthodes permet de créer des objets adresses IP.

- `public static InetAddress getLocalHost() throws UnknownHostException`

  Cette méthode renvoie l'adresse IP du site local d'appel.

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

  Cette méthode construit un nouvel objet *InetAddress* à partir d'un nom textuel de site. Le nom du site est donné sous forme symbolique (`bach.enseeiht.fr`) ou sous forme numérique (`147.127.18.03`).

- `public static InetAddress[] getAllByName(String host) throws UnknownHostException`

  permet d'obtenir les différentes adresses IP d'un site.

Conversions inverses

Des méthodes applicables à un objet de la classe *InetAddress* permettent d'obtenir dans divers formats des adresses IP ou des noms de site. Les principales sont :

- `public String getHostName()`

  obtient le nom complet correspondant à l'adresse IP

- `public String getHostAddress()`

  obtient l'adresse IP sous forme `%d.%d.%d.%d`

- `public byte[] getAddress()`

  obtient l'adresse IP sous forme d'un tableau d'octets.

La classe ServerSocket

Implante un objet ayant le comportement d'un veilleur via une interface par socket.

```
public class java.net.ServerSocket
```

Une implantation standard du service existe mais peut être redéfinie en donnant une implantation explicite sous la forme d'un objet de la classe *java.net.SocketImpl*. Nous nous contenterons d'utiliser la version standard.

Les constructeurs ServerSocket

```
public ServerSocket(int port) throws IOException
public ServerSocket(int port, int backlog) throws IOException
public ServerSocket(int port, int backlog, InetAddress bindAddr)
                                                    throws IOException
```

Ces constructeurs créent un objet veilleur à l'écoute du port spécifié. Le paramètre *backlog* permet de fixer la taille de la file d'attente des demandes de connexion. Si la machine a plusieurs adresses, on peut aussi fixer celle sur laquelle on accepte les connexions.

Appels système correspondants

Ce constructeur correspond à l'utilisation des primitives `socket`, `bind` et `listen`.

Les opérations de la classe ServerSocket

Nous ne retiendrons que les méthodes de base. La méthode essentielle est l'acceptation d'une connexion d'un client :

```
public Socket accept() throws IOException
```

Cette méthode est bloquante, mais l'attente peut être limitée dans le temps par l'appel préalable de la méthode *setSoTimeout*. Cette méthode prend en paramètre le délai de garde exprimé en millisecondes. La valeur par défaut 0 désigne un délai infini. À l'expiration du délai de garde, l'exception *java.io.InterruptedIOException* est levée.

```
public void setSoTimeout(int timeout) throws SocketException
```

La fermeture du socket d'écoute s'exécute par l'appel de la méthode *close*.

Les méthodes suivantes fournissent l'adresse IP ou le port d'un socket d'écoute :

```
public InetAddress getInetAddress()
public int getLocalPort()
```

La classe java.net.Socket

La classe *java.net.Socket* est utilisée pour la programmation des sockets connectés, côté client et côté serveur.

```
public class java.net.Socket
```

Comme pour le serveur, l'implantation standard est redéfinissable (si nécessaire) par le développement d'une nouvelle implantation de la classe *java.net.SocketImpl*.

Constructeurs

Côté serveur, la méthode *accept* de la classe *java.net.ServerSocket* renvoie un socket de service connecté au client. Côté client, on utilise :

- `public Socket(String host, int port)`
  `throws UnknownHostException, IOException`
- `public Socket(InetAddress address, int port) throws IOException`
- `public Socket(String host,int port, InetAddress localAddr, int localPort)`
  `throws UnknownHostException, IOException`
- `public Socket(InetAddress addr,int port,InetAddress localAddr,int localPort)`
  `throws IOException`

Les deux premiers constructeurs construisent un socket connecté à la machine et au port spécifiés. Par défaut, la connexion est de type TCP fiable. Les deux autres interfaces permettent en outre de fixer l'adresse IP et le numéro de port utilisés côté client (plutôt que d'utiliser un port disponible quelconque).

Appels système correspondants

Ces constructeurs correspondent à l'utilisation des primitives *socket*, *bind* (éventuellement) et *connect*.

Opérations de la classe Socket

La communication effective sur une connexion par socket utilise la notion de flots de données (*java.io.OutputStream* et *java.io.InputStream*). Les deux méthodes suivantes sont utilisées pour obtenir les flots en entrée et en sortie.

- `public InputStream  getInputStream()  throws IOException`
- `public OutputStream getOutputStream() throws IOException`

Les flots obtenus servent de base à la construction d'objets de classes plus abstraites telles que *java.io.DataOutputStream* et *java.io.DataInputStream*, ou *java.io.PrintWriter* et *java.io.BufferedReader* (voir exemple ci-après).

Une opération de lecture sur ces flots est bloquante tant que des données ne sont pas disponibles. Cependant, il est possible de fixer un délai de garde à l'attente de données (similaire au délai de garde du socket d'écoute : levée de l'exception *java.io.InterruptedIOException*) :

```
public void setSoTimeout(int timeout) throws SocketException
```

Les caractéristiques de la liaison établie peuvent être obtenues par les méthodes :

- `public InetAddress getInetAddress() # fournit l'adresse IP distante`
- `public InetAddress getLocalAddress() # fournit l'adresse IP locale`
- `public int getPort() # fournit le port distant`
- `public int getLocalPort() # fournit le port local`

L'opération *close* ferme la connexion et libère les ressources du système associées au socket.

Socket en mode datagramme : DatagramSocket

La classe *java.net.DatagramSocket* permet d'envoyer et de recevoir des datagrammes UDP. Il s'agit donc de messages non fiables (possibilité de pertes et de duplication), non ordonnés (les messages peuvent être reçus dans un ordre différent de celui d'émission) et dont la taille (assez faible -- souvent 4 Ko) dépend du réseau sous-jacent.

```
public class java.net.DatagramSocket
```

Constructeurs

- `public DatagramSocket() throws SocketException`
- `public DatagramSocket(int port) throws SocketException`

Construit un socket datagramme en spécifiant éventuellement un port sur la machine locale (par défaut, un port disponible quelconque est choisi).

Émission/réception

- `public void send(DatagramPacket p) throws IOException`
- `public void receive(DatagramPacket p) throws IOException`

Ces opérations permettent d'envoyer et de recevoir un paquet. Un paquet est un objet de la classe *java.net.DatagramPacket* qui possède une zone de données et (éventuellement) une adresse IP et un numéro de port (destinataire dans le cas *send*, émetteur dans le cas *receive*). Les principales méthodes sont :

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

Les constructeurs renvoient un objet pour recevoir ou émettre des paquets.

- `public InetAddress getAddress()`
- `public int getPort()`
- `public byte[] getData()`
- `public int getLength()`

- `public void setAddress(InetAddress iaddr)`
- `public void setPort(int iport)`
- `public void setData(byte[] buf)`
- `public void setLength(int length)`

Les accesseurs *get** permettent, dans le cas d'un *receive*, d'obtenir l'émetteur et le contenu du message. Les méthodes de modification *set** permettent de changer les paramètres ou le contenu d'un message pour l'émission.

Connexion

Il est possible de « connecter » un socket datagramme à un destinataire. Dans ce cas, les paquets émis sur le socket seront toujours pour l'adresse spécifiée. La connexion simplifie l'envoi d'une série de paquets (il n'est plus nécessaire de spécifier l'adresse de destination pour chacun d'entre eux) et accélère les contrôles d'accès (ils ont lieu une fois pour toutes, à la connexion). La « déconnexion » annule l'association (le socket redevient disponible comme dans l'état initial).

- `public void connect(InetAddress address, int port)`
- `public void disconnect()`

Divers

Diverses méthodes renvoient le numéro de port local et l'adresse de la machine locale (*getLocalPort* et *getLocalAddress*), et, dans le cas d'un socket connecté, le numéro de port distant et l'adresse distante (*getPort* et *getInetAddress*). Comme précédemment, on peut spécifier un délai de garde pour l'opération *receive* avec *setSoTimeout*. On peut aussi obtenir ou réduire la taille maximale d'un paquet avec *getSendBufferSize*, *getReceiveBufferSize*, *setSendBufferSize* et *setReceiveBufferSize*.

Enfin, la méthode *close* libère les ressources du système associées au socket.

Exemple

Dans cet exemple, une communication entre un processus client et un processus serveur est établie. Les deux processus échangent des messages sous forme de lignes de texte.

Le processus client commence par émettre un message et le serveur lui répond par un écho de cette ligne. Au bout de 10 échanges, le client envoie une message de terminaison et ferme la connexion.

Le programme du serveur

```
import java.io.*;
import java.net.*;

public class Serveur {
   static final int port = 8080;

   public static void main(String[] args) throws Exception {
        ServerSocket s = new ServerSocket(port);
        Socket soc = s.accept();

        // Un BufferedReader permet de lire par ligne.
        BufferedReader plec = new BufferedReader(
                              new InputStreamReader(soc.getInputStream())
                              );

        // Un PrintWriter possède toutes les opérations print classiques.
        // En mode auto-flush, le tampon est vidé (flush) à l'appel de
println.
        PrintWriter pred = new PrintWriter(
                             new BufferedWriter(
                          new OutputStreamWriter(soc.getOutputStream())),
                             true);    // autoflush

        while (true) {
           String str = plec.readLine();          // lecture du message
           if (str.equals("END")) break;
           System.out.println("ECHO = " + str);   // trace locale
           pred.println(str);                     // renvoi d'un écho
        }
        plec.close();
        pred.close();
        soc.close();
   }
}
```

Le programme du client

```
import java.io.*;
import java.net.*;
/** Le processus client se connecte au site fourni dans la commande
 *   d'appel en premier argument et utilise le port distant 8080. */
public class Client {
   static final int port = 8080;

   public static void main(String[] args) throws Exception {
      Socket sckt = new Socket(args[0], port);
      System.out.println("SOCKET = " + sckt);

      BufferedReader plec =
        new BufferedReader(new InputStreamReader(sckt.getInputStream()));

      PrintWriter pred =
        new PrintWriter(
         new BufferedWriter(new OutputStreamWriter(sckt.getOutputStream()))
        ,true); // autoflush

      String str = "bonjour";
      for (int i = 0; i < 10; i++) {
           pred.println(str);          // envoi d'un message
           str = plec.readLine();      // lecture de l'écho
      }
      System.out.println("END");       // message de terminaison
      pred.println("END") ;
      plec.close();
      pred.close();
      sckt.close();
   }
}
```

Exercices

1) Dans l'exemple précédent, le client et le serveur se synchronisent pour fermer la connexion. Modifier le client pour qu'il s'arrête avant le serveur. Que se passe-t-il côté serveur ? Faites le test inverse : arrêt prématuré du serveur.
2) Sur la base de l'exemple précédent, développer un "chat" à deux participants
   - tout d'abord (éventuellement) en permettant une communication alternée
   - ensuite en permettant l'émission et la réception de messages à tout moment. A cette fin, deux schémas de base peuvent être envisagés :
     ◊ scrutation des flots (Sachant qu'il est possible de savoir si des octets sont disponibles sur un flot d'entrée (méthode *available()*), ou alors, pour les sockets, d'utiliser un délai de garde.)
     ◊ activités (cycliques) concurrentes
3) Etendre la version du "chat" en mode connecté à un nombre quelconque d'utilisateurs . Pour cela, il faudra mettre en place un schéma publier/s'abonner pour gérer l'ensemble des participants. On évaluera l'intérêt (et la pertinence) :
   - d'une organisation répartie, où chaque participant gère et utilise sa propre copie de la liste des participants
   - d'une organisation centralisée, où un serveur spécifique centralise les abonnements et les publications des différents participants.