La programmation réseau

Introduction

Java propose un ensemble de classes pour la programmation réseau. Ces classes sont regroupées dans le paquetage `java.net`.
On peut programmer très facilement des architectures client/serveur en mode TCP ou UDP à l'aide des classes proposées.

Un premier serveur TCP

Des améliorations à ce serveur vont être données par la suite.

```
// Exemple inspire du livre _Java in a Nutshell_
import java.io.*;
import java.net.*;

public class MonServeurSimple {
    public final static int DEFAULT_PORT = 6789;
    protected int port;
    protected ServerSocket listen_socket;
    protected BufferedReader in;
    protected PrintStream out;

     //  traitement des erreurs
    public static void fail(Exception e, String msg)
{
        System.err.println(msg + ": " +  e);
        System.exit(1);
    }
```

```
    // Cree un serveur TCP : c'est un objet de la
    // classe ServerSocket
    // Puis lance l'ecoute du serveur.

    public MonServeurSimple(int port) {
        if (port == 0) port = DEFAULT_PORT;
        this.port = port;
       Socket client = null;
       try { listen_socket = new
ServerSocket(port);
            while(true) {
                    client = listen_socket.accept();
                    in = new BufferedReader(new
InputStreamReader(client.getInputStream()));
              out = new
PrintStream(client.getOutputStream());
              traitement();
              }
 } catch (IOException e) {
     fail(e, "Pb lors de l'ecoute"); }
  finally { try {client.close();} catch (IOException
e2) { } }
   }
    // Le lancement du programme :
    // - initialise le port d'ecoute
    // - lance la construction du serveurTCP
    public static void main(String[] args) {
        int port = 0;
        if (args.length == 1) {
        try { port = Integer.parseInt(args[0]);  }
       catch (NumberFormatException e) { port =
0; }
        }
        new MonServeurSimple(port);  }
```

```
public void traitement() {
        String line;
        StringBuffer revline;
        int len;
        try {
            for(;;) {
                // lit la ligne
                line = in.readLine();
        // Si cette ligne est vide,
       // le serveur se termine
                if (line == null) break;

                // sinon l'ecrit a l'envers
                len = line.length();
                revline = new StringBuffer(len);
                for(int i = len-1; i >= 0; i--)
                    revline.insert(len-1-i, line.charAt(i));
                // et l'envoie dans la socket
                out.println(revline);
            }
        } catch (IOException e) { ; }
    }
}
```

Conclusion : serveur TCP

On utilise donc la structure de code :

```
    ServerSocket listen_socket = new ServerSocket(port);
    Socket client = listen_socket.accept();
     BufferedReader in = new BufferedReader (new
InputStreamReader(client.getInputStream()));
     PrintStream out = new
PrintStream(client.getOutputStream());
```

Client TCP

```
// Exemple inspire du livre _Java in a Nutshell_
import java.io.*;
import java.net.*;

public class MonClient {
   public static final int DEFAULT_PORT = 6789;
   public static void usage() {
      System.out.println("Usage: java MonClient
<machineServeur> [<port>]");
      System.exit(0);
   }
   public static void main(String[] args) {
      int port = DEFAULT_PORT;
      Socket s = null;
      // initialise le port
      if ((args.length != 1) && (args.length != 2))
            usage();
      if (args.length == 1) port =
DEFAULT_PORT;
      else {
         try { port = Integer.parseInt(args[1]); }
         catch (NumberFormatException e) {
usage(); }
      }

      try {
         // Cree une socket pour communiquer
        // avec le service se trouvant sur la
        // machine args[0] au port port
         s = new Socket(args[0], port);
```

```
            // Cree les streams pour lire et ecrire
            // du texte dans cette socket
            BufferedReader sin = new
BufferedReader(new
InputStreamReader(s.getInputStream()));
            PrintStream sout = new
PrintStream(s.getOutputStream());
            // Cree le stream pour lire du texte a
partir du clavier
            BufferedReader in = new
BufferedReader(new
InputStreamReader(System.in));
            // Informe l'utilisateur de la connection
            System.out.println("Connected to " +
s.getInetAddress() + ":"+ s.getPort());

            String line;
            while(true) {
                // le prompt
              System.out.print("> ");
System.out.flush();
                // lit une ligne du clavier
                line = in.readLine();
                if (line == null) break;
                // et l'envoie au serveur
                sout.println(line);
                // lit une ligne provenant de la socket,
                // donc du serveur
                line = sin.readLine();
                // Verifie si la connection est fermee.
        // Si oui on sort de la boucle
          if (line == null) {
              System.out.println("Connection ferme
par le serveur."); break; }
```

```
    // Ecrit la ligne traite par le serveur et envoye
    // par lui.
                System.out.println(line);
            }
        }
       catch (IOException e) { System.err.println(e);
}
        // Refermer dans tous les cas la socket
        finally { try { if (s != null) s.close(); }
                    catch (IOException e2) { ; } }
    }
}
```

Conclusion : client TCP

On utilise donc la structure de code :

```
Socket socket = new Socket(machineDist, port);
BufferedReader in =
  new BufferedReader (new InputStreamReader(
     socket .getInputStream()));
PrintStream out =
    new PrintStream(socket .getOutputStream());
```

Résultat de l'exécution

Sur la machine serveur :

```
java MonServeurSimple 8888
```

Sur la machine client :

```
java MonClient MachineDist 8888
Connected to MachineDist/163.173.XXX.XXX:8888
> bonjour
ruojnob
> ca marche tres bien
neib sert ehcram ac
> au revoir
riover ua
```

La classe `InetAddress`

Les objets de cette classe modélisent les adresses IP. Ils sont utilisés par exemple comme argument des constructeurs de la classe `Socket`.

Cette classe donne aussi des renseignements sur l'adresse IP à l'aide de méthodes statiques. Par exemple pour obtenir l'adresse IP de la machine locale on utilise : `InetAddress.getLocalHost()`

Ceci est très utile pour lancer un client et un serveur sur la même machine locale. On écrit alors :

```
Socket serv = new Socket(InetAddress.getLocalHost(), 4567);
```

Serveur multithreadé

Le serveur ci dessus fait entièrement le traitement avant de revenir en écoute. Si ce traitement est long, il bloque d'autres clients qui le demandent même pour un traitement cours (traitement batch).
Il vaut mieux envisager un algorithme comme :

> le serveur écoute
>
> lorsqu'un client se connecte, une thread est créée pour traiter sa requête
>
> retour de l'écoute.

Voici un tel serveur.

Serveur TCP multithreadé

```
// Exemple inspire du livre _Java in a Nutshell_

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

public class MonServeur extends Thread {
    public final static int DEFAULT_PORT = 6789;
    protected int port;
    protected ServerSocket listen_socket;
    // traitement des erreurs
    public static void fail(Exception e, String msg)
{
        System.err.println(msg + ": " + e);
        System.exit(1);
    }
    // Cree un serveur TCP : c'est un objet de la
classe ServerSocket
    // Puis lance l'ecoute du serveur.
    public MonServeur(int port) {
        this.port = port;
        if (port == 0) port = DEFAULT_PORT;
        try {
            listen_socket = new ServerSocket(port);
        } catch (IOException e) {
            fail(e, "Pb lors de la creation de la socket
server");
        }
        System.out.println("Serveur lance sur le
port " + port);
        this.start();
    }
```

```
    // Le corps de la thread d'ecoute :
    // Le serveur ecoute et accepte les
connections.
    // pour chaque connexion, il cree une thread
    // (objet de la classe Connection,
    // classe derivee de la classe Thread)
    // qui va la traiter.
    public void run() {
        try {
            while(true) {
                Socket client_socket =
listen_socket.accept();
                Connection c = new
Connection(client_socket);
            } }
        catch (IOException e) {
            fail(e, "Pb lors de l'ecoute");
        }
    }
    // Le lancement du programme :
    // - initialise le port d'ecoute
    // - lance la construction du serveurTCP
    public static void main(String[] args) {
        int port = 0;
        if (args.length == 1) {
            try {
                port = Integer.parseInt(args[0]); }
            catch (NumberFormatException e) {
                port = 0;
            }
        }
        new MonServeur(port);
    }
}
```

```
// la classe Connection : c'est une thread
class Connection extends Thread {

    protected Socket client;
    protected BufferedReader in;
    protected PrintWriter out;

    // Initialise les streams and lance la thread
    public Connection(Socket client_socket) {
        client = client_socket;

        try {
            in = new BufferedReader(new
InputStreamReader(client.getInputStream()));
            out = new
PrintWriter(client.getOutputStream());
        }
        catch (IOException e) {
            try { client.close(); }
            catch (IOException e2) { ; }
            System.err.println("Exception lors de
l'ouverture des sockets :
" + e);
            return;
        }
        this.start();
    }
```

```
    // Fournit le service (inverse la ligne recue et
la retourne)
    public void run() {
        String line;
        StringBuffer revline;
        int len;

        try {
            for(;;) {
                // lit la ligne
                line = in.readLine();

                // Si cette ligne est vide, le serveur se
termine
                if (line == null) break;

                // sinon l'ecrit a l'envers
                len = line.length();
                revline = new StringBuffer(len);
                for(int i = len-1; i >= 0; i--)
                    revline.insert(len-1-i, line.charAt(i));

                // et l'envoie dans la socket
                out.println(revline);
                out.flush();
            }
        } catch (IOException e) { ; }
        finally {
            try { client.close(); }
            catch (IOException e2) {;}
        }
    }
}
```

Résultat de l'exécution

Sur la machine serveur :

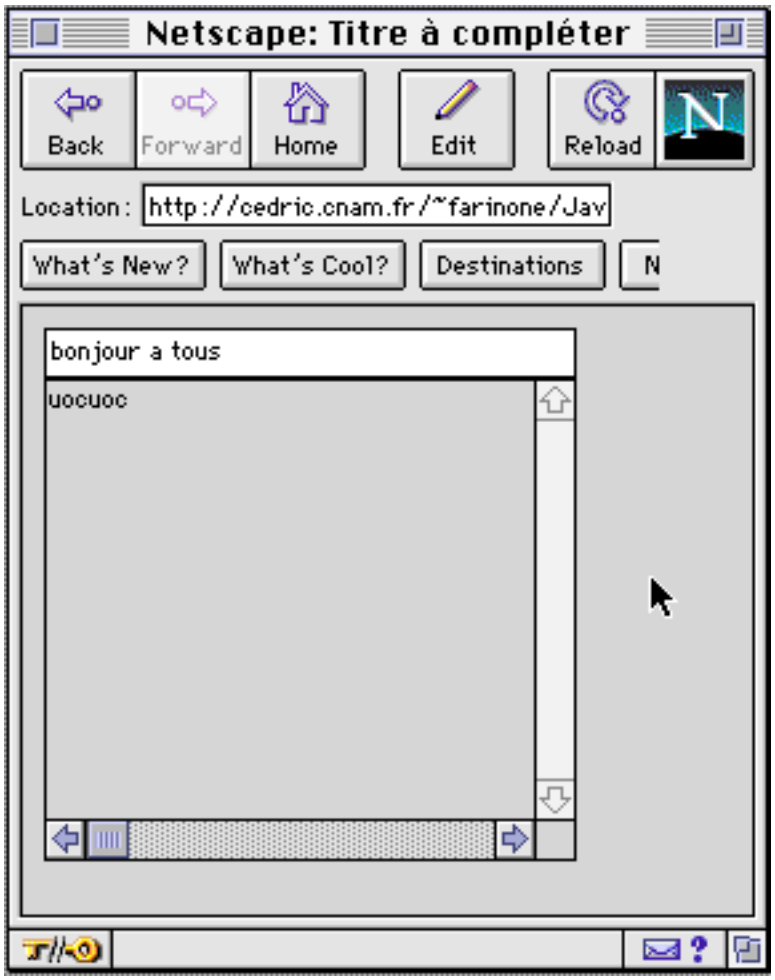
```
java MonServeur &
```

Sur la machine client :

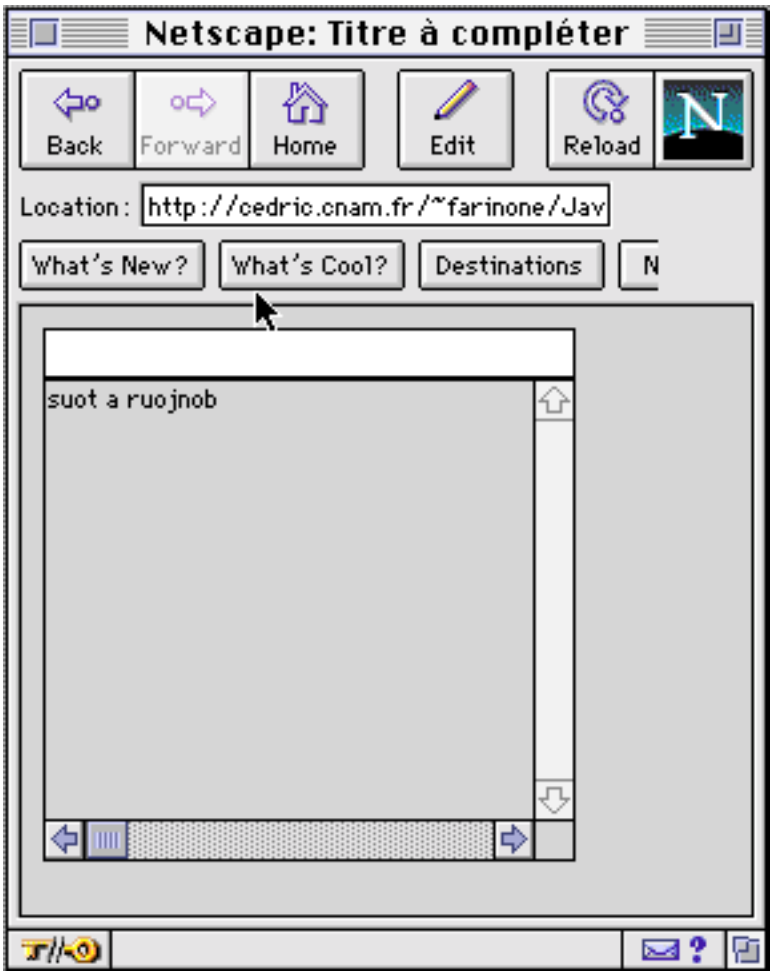
```
java MonClient MachineDist
Connected to MachineDist/163.173.XXX.XXX:6789
> bonjour
ruojnob
> ca marche tres bien
neib sert ehcram ac
> au revoir
riover ua
```

applet cliente

Une applet peut se connecter sur le site d'où elle provient.
En supposant que ce site possède le serveur ci dessus, voici une applet qui affiche une zone de saisie et une fenêtre de texte.
L'utilisateur écrit une chaîne dans la zone de saisie, appuie sur la touche Entrée. Cette chaîne est envoyée au serveur, qui la traite et la retourne (à l'envers).
Le résultat retourné est affiché dans la fenêtre de texte.



applet cliente (suite)



Un exemple

```
// Exemple fortement inspiré du livre "Java in a
Nutshell" de David Flanagan.

import java.applet.*;
import java.awt.*;
import java.io.*;
import java.net.*;

public class AppletClient extends Applet {
    public static final int PORT = 6789;
    Socket s;
    DataInputStream in;
    PrintStream out;
    TextField inputfield;
    TextArea outputarea;
    ThreadD ecoutelistener;

    // Créer une socket qui communique avec
    // la machine distante sur le port 6789 de cette
    // machine distante.
   // Cette machine distante doit être celle
   // d'où provient l'applet.

   // Créer ensuite les deux flots de lecture et
   // d'écriture associé à cette socket.

   // Créer l'interface graphique
   // (un TextField et un TextArea

   // Enfin créer une thread en attente
   // des réponses du traitement du serveur.
```

```
    public void init() {
        try {
            s = new
Socket(this.getCodeBase().getHost(), PORT);
            in = new
DataInputStream(s.getInputStream());
            out = new
PrintStream(s.getOutputStream());
            inputfield = new TextField();
            outputarea = new TextArea();
            outputarea.setEditable(false);
            this.setLayout(new BorderLayout());
            this.add("North", inputfield);
            this.add("Center", outputarea);
            listener = new ThreadDecoute(in,
outputarea);

            this.showStatus("Connected to "
                    +
s.getInetAddress().getHostName()
                    + ":" + s.getPort());
            inputfield.addActionListener(new
ActionListener() {
    // Quand l'utilisateur écrit une ligne et appui
    // RETURN, cette ligne est envoyée au
   // serveur
            public actionPerformed(ActionEvent evt) {
                    out.println(inputfield.getText());
                    inputfield.setText("");
             }
        }
  catch (IOException e) showStatus(e.toString());
    }
}
```

```
// Thread qui attend la réponse du serveur puis
// l'affiche dans la TextArea.

class ThreadDecoute extends Thread {
    DataInputStream in;
    TextArea output;
    public ThreadDecoute(DataInputStream in,
TextArea output) {
        this.in = in;
        this.output = output;
        this.start();
    }
    public void run() {
        String line;
        try {
            for(;;) {
              BufferedReader d = new
BufferedReader(new InputStreamReader(in));
                line = d.readLine();
                if (line == null) break;
                output.setText(line);
            }
        }
        catch (IOException e)
output.setText(e.toString());
        finally output.setText("Connection closed by
server.");
    }
}
```

programmation UDP

UDP est une autre couche transport, plus simple mais moins sécurisée que TCP. Elle est orientée sans connexion et les messages peuvent être perdus ou déséquencés. Cette couche est quand même utilisée pour son efficacité et ce sont alors les couches supérieures qui implantent la sécurité.

Les messages UDP sont appelés des datagrammes. UDP signifiant d'ailleurs "Unreliable Datagram Protocol".

Syntaxiquement, le transport par UDP est fait par des `DatagramPacket` (de taille limité dû au protocole UDP). Ces paquets contiennent la machine et le port destinataire. De plus les données qu'ils transportent sont obligatoirement des tableaux de `byte`. Il faut donc transformer les données à transporter en tableau de `byte` : on utilise souvent les classes `ByteArrayXXXputStream` faites pour cela ou alors une astuce voir exemple suivant.

programmation UDP (suite)

C'est dans le paquet UDP qu'on note la machine et le port destinataire. La socket UDP sert seulement à la connexion IP pour l'envoi et la l'écoute pour la réception. On utilise pour cela des objets de la classe `DatagramSocket` et les méthodes `send()` et `receive()` de cette classe.

En résumé les deux classes essentielles pour programmer en UDP sont les classes `DatagramPacket` et `DatagramSocket`.

Les constructeurs utilisés

Pour l'envoi, on utilise :

```
public DatagramPacket(byte[] buf, int
lg, InetAdress ia, int port)
```

et

```
public DatagramSocket() throws
SocketException
```

Pour la réception, on utilise :

```
public DatagramPacket(byte[] buf, int
lg)
```

et

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

C'est un peu curieux d'avoir des constructeurs d'une même classe avec un sémantique distincte mais c'est le cas !!

Envoyer et recevoir en UDP

Pour l'envoi

Le premier argument du constructeur `DatagramPacket` contient les données à envoyer sous forme d'un tableau de `byte`.
Le second argument est la taille de ces données.
Le troisième et quatrième argument est la machine (sous forme `InetAdress`) et le port destinataire.

Pour envoyer on demande simplement à utilisé une socket UDP à l'aide du second constructeur `DatagramSocket()`.

Pour la réception

Le constructeur `DatagramPacket` possède deux arguments : le premier est un référence sur un tableau. La référence doit être initialisée mais le tableau lui même sera rempli par les données reçues.
Le second argument est la taille de ce tableau "de réception".

Le second constructeur `DatagramSocket` a pour argument le port derrière lequel écoute le programme de réception.

Un émetteur UDP

```
import java.io.*;
import java.net.*;
public class UDPSend {
   static final int port = 6010;
   public static void main(String args[]) throws
Exception {
      if (args.length != 2) {
         System.out.println("Usage: java
UDPSend <hostname> <message>");
         System.exit(0); }
      // On récupère l'adresse IP de l'hôte distant.
      InetAddress address =
InetAddress.getByName(args[0]);
      // Convertion du texte en tableau de byte
      byte[] message;
      message = args[1].getBytes();
      // On construit le datagram.
      // c'est dans le datagram qu'est précisé la
      // destination finale hôte et le numéro de
      // port de la destination finale.
      DatagramPacket packet = new
DatagramPacket(message, msglen,
       address, port);
      // Crée la socket pour l'envoi de datagrams,
      // et effectue cet envoi.
      DatagramSocket socket = new
DatagramSocket();  socket.send(packet);
   }
}
```

Un récepteur UDP

```
// Ce programme écoute le port 6010 et affiche
// la chaine de caractères reçue sur ce port.
import java.io.*;
import java.net.*;
public class UDPReceive {
   static final int port = 6010;
   public static void main(String args[]) throws
Exception
   {  byte[] buffer = new byte[1024];
      String s;
      // Crée une socket d'écoute sur ce port.
      DatagramSocket socket = new
DatagramSocket(port);
      for(;;) {
         // Création du packet de réception.
         DatagramPacket packet = new
DatagramPacket(buffer, buffer.length);
         // écoute et réception d'un datagram.
         socket.receive(packet);
         // Convertion byte -> String à l'aide
         // du constructeur
//String(ta_byte[], octpoidsfort, inddeb, longueur)
         s = new String(buffer, 0, 0,
packet.getLength());
         // affichage du packet reçu en indiquant
         // la machine émettrice et le numéro de
         // port de la machine émettrice.
System.out.println("UDPReceive: received from "
+ packet.getAddress().getHostName() + ":"
+packet.getPort() + ": " + s);
      } } }
```

Une exécution

La machine `MachEmet` est émettrice, la machine `Recept` est réceptrice. On lance le récepteur :

```
Recept: % java UDPReceive
```

on envoie des données UDP :

```
MachEmet: % java UDPSend Recept 'ca marche super'
MachEmet: % java UDPSend Recept '2ieme essai'
```

et on les reçoit au fur et à mesure :

```
Recept: % java UDPReceive
UDPReceive: received from MachEmet:35913:  ca marche super
UDPReceive: received from MachEmet:35914: 2ieme essai
```

La diffusion

Le protocole IP a prévu de pouvoir envoyer des paquets à plusieurs machines sans dupliquer ces paquets dans la partie réseau commune à plusieurs machines. Il faut, pour cela, quelques contraintes comme :
- les routeurs "jouent le jeu" de cette diffusion c'est à dire acceptent de véhiculer de tels paquets.
- les machines possèdent des adresses de diffusion (dite de classe D) c'est à dire possèdent un numéro IP compris entre `224.0.0.1` et `239.255.255.255`.

Lorsqu'on diffuse un paquet celui-ci à la possibilité de parcourir tout l'internet. Pour éviter d'inonder internet les paquets IP possède un champ TTL (Time To Live), entier qui est décrémenté à chaque passage dans les routeurs (souvent de la valeur 1). Lorsque cette valeur passe à 0, le routeur détruit le paquet et ne le diffuse pas. Dans le cas d'un paquet en diffusion, on doit explicitement positionner la valeur de ce champ. Les conventions (approximatives) sont :
valeur TTL de :
1 = le paquet reste dans le réseau local,
16 = le paquet reste dans la région,
127 = le paquet est envoyé au monde entier

Diffusion : syntaxe

Le protocole de diffusion s'appuie sur UDP (donc n'a pas la sécurité de TCP) et la syntaxe de programmation est donc celle de UDP. On utilise donc la classe `DatagramPacket` pour modéliser des paquets à diffuser. La classe qui permet la diffusion est la classe `MulticastSocket` avec :
- le constructeur `public MulticastSocket() throws IOException` pour l'envoi
- le constructeur `public MulticastSocket(int port) throws IOException` pour la réception

On précise qu'on veut bien être une machine ayant un numéro IP qui recevra des paquets de diffusion en lançant dans le programme la méthode
`public void joinGroup(InetAddress mcastaddr) throws IOException`
sur l'objet de classe `MulticastSocke`t.
Pour quitter ce groupe, il suffit de lancer la méthode `public void leaveGroup(InetAddress mcastaddr) throws IOException` sur ce même objet.

Pour diffuser, il n'est pas nécessaire de faire partie du groupe de diffusion.

Diffusion : un exemple

Voici un serveur (qui rend un service !!) d'horloge parlante c'est à dire donne l'heure toutes les secondes. Remarquer que ce serveur diffuse (i.e. envoie) et que ce sont les clients qui seront récepteurs de l'heure (lorsqu'ils la demandent).

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

public class MultigServer {
public static final int PORT = 9999;
public static final String GROUP =
"229.69.69.69";
public static final byte TTL = 1;
public static final int DATA_MAX_SIZE = 512;

public static void main(String[] args) {
byte[] recBuffer = new byte[DATA_MAX_SIZE];
try {
  MulticastSocket veille = new MulticastSocket();
  InetAddress adrGroupe =
InetAddress.getByName(GROUP);
```

```
while (veille != null) {
    ByteArrayOutputStream boz = new
          ByteArrayOutputStream();
    ObjectOutputStream oz = new
ObjectOutputStream(boz);
    oz.writeObject(new Date()));
    DatagramPacket sendPack =
        new DatagramPacket(boz.toByteArray(),
    boz.size(), adrGroupe, PORT);
    veille.send(sendPack, TTL);
    try {
       Thread.sleep(1000);
    } catch (InterruptedException e){}
  } // while(veille != null) {
 } catch (Exception e) {}
} // main
} // class MultigServer
```

Diffusion : un client

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

public class MultigClient {
public static final String GROUP =
"229.69.69.69";
public static final int DATA_MAX_SIZE =
512;
public static void main(String[] args) {
  MulticastSocket socket = null;
  InetAddress adrGroupe = null;
  byte[] recBuffer = new
  byte[DATA_MAX_SIZE];
  try {
    socket = new
      MulticastSocket(MultigServer.PORT);
    adrGroupe =
 InetAddress.getByName(MultigServer.GROUP);
    if (socket != null) {
      socket.joinGroup(adrGroupe);
      DatagramPacket recvPack =
         new DatagramPacket(recBuffer,
           recBuffer.length);
      for (int i = 0; i < 10; i++) {
        socket.receive(recvPack);
        ObjectInputStream inz =
           new ObjectInputStream(new
      ByteArrayInputStream(recvPack.getData()));
        Object obj = inz.readObject();
        System.out.println(obj);
```

```
        inz.close();
      }
    }
  } catch (Exception e) {}
  finally {
    if (socket != null) {
    try { socket.leaveGroup(adrGroupe);
    } catch (Exception e) {}
    } // if
  } // finally (socket != null)
 } // main
} // class MultigClient {
```

Diffusion : lancement des programmes

Lancer sur un machine le serveur par :

```
java MultigServer
```

Le serveur diffuse alors l'heure.

Lancer sur d'autres machines :

```
java MultigClient
```

Les clients recoivent bien "l'horloge parlante"

Connexion sur des URL

Java propose des classes permettant de manipuler des URL.

La classe URL

principales méthodes

La classe URL permet de construire une URL à partir d'une chaîne de caractères.

principales méthodes :

```
public URL(String) throws
MalformedURLException

public URL(String, String hote, String
fic) throws MalformedURLException

public URL(String protocol, String
hote, int port, String fic) throws
MalformedURLException
```

sont des constructeurs retournant un objet URL.

Les méthodes `getFile()`, `getHost()`, `getPort()`, `getProtocol()` retournent des informations sur l'instance.

La classe `URLConnection`

principales méthodes

On permet d'obtenir des informations plus précises d'une ressource repérée par son URL.
On fait correspondre un objet `URL` à un objet `URLConnection` par
`public URLConnection openConnection()`
de la classe `URL` ou le constructeur
`protected URLConnection(URL)`

On peut obtenir des renseignements à l'aide de :
`public String getContentType()`
retourne la valeur du champ `content-type` (i.e. le format MIME (type/soustype))

`public int getContentLength()`
retourne la valeur du champ `content-length` (par exemple la taille du document HTML)

`public long getLastModified()`
retourne la valeur du champ `last-modified` (par exemple la date de dernière modification du document HTML). Le résultat est le nombre de secondes depuis le 1er janvier 1970 GMT.

`public long getExpiration()`
retourne la valeur du champ `expires` (par exemple la date d'expiration du document HTML) ou 0 si ce champ est inconnu.

Exemple

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

public class GetURLInfo {
   public static void printinfo(URLConnection u)
throws IOException {
System.out.println(u.getURL().toExternalForm()
+ ":");
       System.out.println("  Content Type: " +
u.getContentType());
       System.out.println("  Content Length: " +
u.getContentLength());
       System.out.println("  Last Modified: " + new
Date(u.getLastModified()));
       System.out.println("  Expiration: " +
u.getExpiration());
       System.out.println("  Content Encoding: " +
u.getContentEncoding());

  // Lit et ecrit les 5 premieres lignes du
// fichier repere par l'URL
       System.out.println("Les 5 premieres
lignes:");
       DataInputStream in = new
DataInputStream(u.getInputStream());
       for(int i = 0; i < 5; i++) {
           String line = in.readLine();
           if (line == null) break;
           System.out.println("  " + line);  }  }
```

```
    // Cree un objet et ouvre une connexion
    // sur cette URL. Affiche les infos.
    public static void main(String[] args)
        throws MalformedURLException,
IOException
    {
URL url = new URL(args[0]);
URLConnection connection =
url.openConnection();
printinfo(connection);
    }
}
```

Une exécution

```
pouchan: % java GetURLInfo
'http://cedric.cnam.fr/personne/farinone/'
http://cedric.cnam.fr/personne/farinone/:
  Content Type: text/html
  Content Length: 1485
  Last Modified: Tue Sep 10 17:46:23 MET DST
1996
  Expiration: 0
  Content Encoding: null
Les 5 premieres lignes:
  <html>
  <head><title>Page d'accueil de Jean Marc
Farinone</title></head>

  <body>Page d'accueil de Jean Marc Farinone
  <h2>Jean Marc Farinone</h2>
```

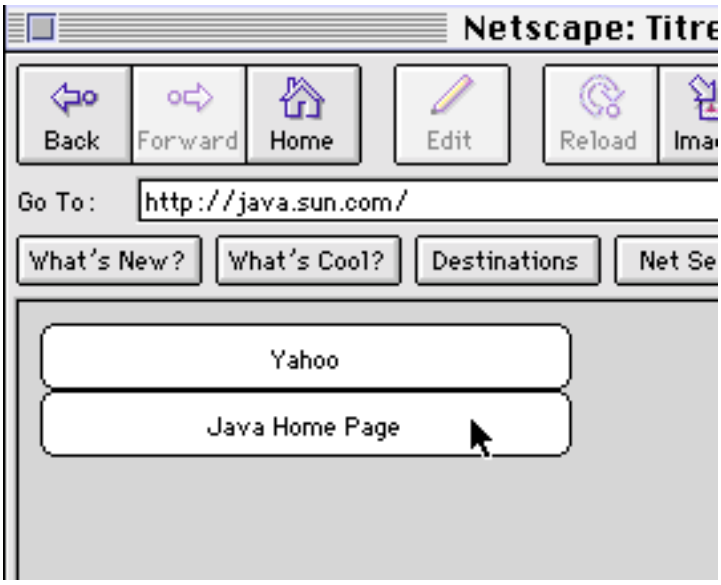
showDocument() de la classe AppletContext

On peut aussi, à l'intérieur d'une applet, lancer une connexion sur une URL et afficher son contenu dans le browser après traitement par l'interpréteur HTML à l'aide de la méthode `showDocument()` de la classe `AppletContext`. On obtient un objet `AppletContext` à partir d'un objet `Applet` grâce à la méthode `getAppletContext()`.

```
/* inspiré de "le programmeur Java" */
import java.awt.*;
import java.net.*;
import java.applet.*;

public class ButtonLink extends Applet {
  static final int nbBoutons = 2;
  Bookmark bmlist[] = new
Bookmark[nbBoutons];
  Button bt[] = new Button[nbBoutons];

  public void init() {
    bmlist[0] = new Bookmark("Yahoo",
"http://www.yahoo.com");
    bmlist[1]= new Bookmark("Java Home Page",
"http://java.sun.com");
    setLayout(new GridLayout(bmlist.length,1));
```

```
  for (int i = 0; i < bmlist.length; i++) {
       bt[i] = new Button(bmlist[i].name);
       add(bt[i]);
       bt[i].addActionListener(new ActionListener(){
       public void actionPerformed(ActionEvent e){
             URL theURL = null;
             for (int i=0; i < bt.length; i++)
                   if (evt.getSource() == bt[i]) {
                         theURL = bmlist[i].url;
                   }
             if (theURL != null) {
         showStatus("chargement de: " + theURL);
      getAppletContext().showDocument(theURL);
             } // if (theURL != null)
       } // actionPerformed(…)
     } // new ActionListener() {
   ); // bt[i].addActionListener(
  } // for (int i = 0; i < bmlist.length;
 } // init()
} // public class ButtonLink

class Bookmark {
  String name;
  URL url;

  Bookmark(String name, String theURL) {
    this.name = name;
    try { this.url = new URL(theURL); }
    catch ( MalformedURLException e) {
      System.out.println("Bad URL: " + theURL);
    }
  }
}
```

Une exécution



Bibliographie

Java Network Programming : Elliotte Rusty Harold ed O'Reilly ISBN 1-56592-227-1
traduit en français par
Programmation réseau avec Java : Elliotte Rusty Harold ; ed O'Reilly ISBN 2-84177-034-6

Java in a nutshell (covers Java 1.0) : David Flanagan ; O'Reilly ISBN 1-56592-183-6 traduit en français