Chapitre 6

Ce chapitre est le premier des chapitres consacrés à la programmation réseau en langage Java. Une étude efficace de ces chapitres nécessite quelques connaissances concernant les réseaux informatiques et les protocoles Internet. Le cours **Réseaux Informatiques** et le cours **Protocoles Internet** sont élaborés pour acquérir ces connaissances.

Le mécanisme de *socket* est essentiel pour le développement des applications sur le réseau Internet. Chaque dispositif connecté à ce réseau possède une adresse internet et chaque application qui communique sur ce réseau est attachée à un port de service. Le couple “adresse Internet - numéro du port” forme un socket.

En général, les données qui circulent sur un réseau peuvent être envoyées en deux modes:

- en mode connecté
- en mode datagramme.

Le langage Java offre un ensemble de mécanismes socket (classes et méthodes) permettant de réaliser les deux modes de communication.

Nous allons étudier la communication basée sur le mécanisme socket dans l’ordre croissant de la complexité des protocoles sous-jacents:

- le protocole UDP et la communication en mode datagramme
- le protocole TCP et la communication en mode connecté
- le protocole HTTP et la communication en mode session

Le protocole UDP est le plus simple protocole de transport sur le réseau Internet. Il offre la possibilité d’envoi des datagrammes sans garantir la fiabilité du transfert. Cette solution est convenable pour les transferts à courte distance sur des supports physiques très fiables. A longue distance, le protocole TCP devient nécessaire.

Le langage Java offre aux utilisateurs un ensemble de classes *socket* dans le package *java.net*. L’implémentation de la communication sur le protocole UDP est basée sur deux classes:

- *DatagramSocket*
- *DatagramPacket*

Un paquet *DatagramPacket* contient les données à envoyer dans un datagramme. Les méthodes de la classe *DatagramSocket* envoient et reçoivent des datagrammes de classe *DatagramPacket*.

La classe *DatagramPacket*

Un datagramme UDP intègre dans son en-tête les numéros de port de source et de destination. Ces deux valeurs sont codées sur des entiers courts de 16 bits. La valeur maximale d’un numéro de port est 65 536. Le champ de données d’un datagramme UDP peut porter jusqu’au 65 507 octets; sa taille effective dépend de la taille des tampons et des trames physiques (e.g. 512 octets).

Pour construire un datagramme en Java nous utilisons le constructeur *DatagramPacket()* avec des paramètres à initialiser.

Un datagramme en **réception** a la forme suivante:

```
public DatagramPacket(byte[] tampon, int taille);
public DatagramPacket(byte[] tampon, int deplacement, int taille);
```

Quand un datagramme arrive, il est stocké dans le tampon à partir de sa position *tampon[0]*.

Attention: Le *tampon* doit être au moins de la taille fournie par l'argument *taille*; sinon le constructeur lève l'exception *IllegalArgumentException*.

Un datagramme en **émission** a la forme suivante:

```
public DatagramPacket(byte[] tampon, int taille, InetAddress destination, int port);
public DatagramPacket(byte[] tampon, int deplacement, int taille, InetAddress destination, int port);
```

L'objet de la classe *InetAddress* doit être fourni par l'utilisateur. Il peut être obtenu dans l'application à partir d'une chaîne de caractères *"www.ireste.fr"* ou *"193.52.81.10"* par l'instanciation type:

```
InetAddress destination = new InetAddress.getByName("www.ireste.fr"");
```

Le numéro du port est une valeur entière. Pour les services connus, cette valeur se situe entre 1 et 1023.

La séquence de code suivante prépare un datagramme pour l'envoyer sur le port 7 (*echo*):

```
String s = "Ceci est un test";
byte[] donnee = s.getBytes("ASCII");  // conversion d'une chaine ASCII vers une chaine d'octets
int port = 7;
try {
InetAdress destination = InetAddress.getByName("www.ireste.fr"");
DatagramPacket dp = new DatagramPacket(donnee,donnee.length,destination,port);  // création d'un paquet
}
catch(IOException e) {
}
```

Attention: Les tampons dans les datagrammes sont du type *byte*; les données de type *String* doivent être converties en *byte* avant leur insertion dans un tampon (*donnee*).

```
byte[] donnee = s. getBytes("ASCII");
```

L'opération inverse est nécessaire à la réception d'un datagramme:

```
String s = new String (source.getData(), "ASCII");
```

où: *source* est un objet de classe *DatagramPacket*

L'exemple suivant est une application complète dont l'objectif est d'illustrer différentes méthodes de type *get* utilisées pour la préparation et la lecture des datagrammes:

- *String.getBytes()* - conversion String vers Bytes,
- *InetAddress.getByName()* - recherche d'une adresse IP dans système DNS,
- *DatagramPacket.getAddress()* - extraction de l'adresse de l'expéditeur,
- *DatagramPacket.getPort()* - extraction du numéro de port de l'expéditeur,

- *DatagramPacket.getLength()* - extraction de la taille du paquet,
- *DatagramPacket.getData()* - extraction des données du paquet,
- *DatagramPacket.getOffset()* - extraction du déplacement des données dans le tampon de réception.

```
import java.net.*;
public class TestDatagramme {
public static void main(String[] args) {
String s = "C'est un test";
byte[] donnee= s.getBytes();        // conversion chaine de caracteres vers chaine d'octets
try {
InetAddress aip = InetAddress.getByName("oak.ireste.fr");
int port = 7;
DatagramPacket paquet = new DatagramPacket(donnee,donnee.length,aip,port);
System.out.println("Ce paquet est adresse a " + paquet.getAddress() + " sur le port " + paquet.getPort());
System.out.println("Il y a " + paquet.getLength() + "octets dans le paquet");
System.out.println(new String(paquet.getData(),paquet.getOffset(),paquet.getLength()));
}
catch(UnknownHostException e) { System.err.println(e);}
}
}
```

Les constructeurs de datagrammes préparent les nouveaux datagrammes à envoyer, mais dans certains cas il est intéressant de modifier le datagramme existant.

Les méthodes *setData(byte[] donnee)* et *setData(byte[] donnee, int deplacement, int taille)* permettent de modifier le contenu d'un datagramme.

Par exemple, si on veut envoyer un fichier en plusieurs morceaux, le même datagramme peut être envoyé plusieurs fois en changeant simplement le champ de données.

```
int deplacement = 0;
DatagramPacket paquet = new DatagramPacket(info,deplacement,256);
int octetsEnvoyes = 0;
while(octetsEnvoyes<info.length) {
socket.send(paquet);
octetsEnvoyes += paquet.getLength();
int octetsAEnvoyer = info.length - octetsEnvoyes;
int taille = (octetsAEnvoyer>256) ? 256:octetsAEnvoyer;
paquet.setData(info,octetsEnvoyes,256);            // modification de la donnee du datagrammee existant
}
```

La méthode *setAddress()* permet de changer de l'adresse du datagramme avant son envoi. Ainsi le même datagramme peut être envoyé vers plusieurs destinations.

```
String s = "message";
byte[] info = s.getBytes("ASCII");
DatagramPacket paquet = new DatagramPacket(info,info.length);
paquet.setPort(6969);
int reseau = "193.52.81.";
for(int hote =1; hote <255; hote++) {
try {  InetAddress cible = InetAddress.getByName(reseau + hote);
paquet.setAddress(cible);     socket.send(paquet);  }          // setPort(int) - permet en plus de modifier le port
catch(Exception e) {}
}
```

Classe *DatagramSocket*

Pour envoyer ou recevoir un datagramme, il faut disposer d'un objet *socket local*. Cet objet est créé par l'un des constructuers de la classe *DatagramSocket*.

```
DatagramSocket(int localPort, InetAddress localAdr);
DatagramSocket(int localPort);
DatagramSocket(0, InetAddress localAdr);
DatagramSocket(0, null);      équivalent à          DatagramSocket();
```

Le constructeur *DatagramSocket* permet de préciser un port UDP et une adresse Internet à utiliser sur la machine locale et d'attacher le socket sur ces paramètres. Si l'utilisateur indique la valeur 0 pour *localPort*, le système d'exploitation va fournir une valeur par défaut (entre 1023 et 5000). La valeur *null* de *localAdr* permet de choisir l'adresse IP par défaut.

L'opération d'attachement s'effectue correctement à condition que le numéro proposé soit toujours disponible; dans le cas contraire une exception est levée.

Remarque: Dans le système UNIX, l'utilisateur "root" a le droit d'exploiter tous les numéros possibles

Exemple:

```
import java.net.*;
public class TestPort {
public static void main(String[] args) {
for(int port=1; port<=65535; port++)
 {
 try
  {
  DatagramSocket serveur = new DatagramSocket(port);
  serveur.close();
  }
  catch(SocketException e) {
  System.out.println("le socket sur le port " + port + " est utilise");  }
```

```
 }
}
}
```

Envoi et réception des datagrammes

Lorsqu'on dispose d'un socket et d'un datagramme, il suffit de leur associer la méthode *send()* pour envoyer le datagramme à l'adresse et au port spécifiés.

```
String s = "message";
byte[] donnee = s.getBytes("ASCII");
InetAddress destination = InetAddress.getByName("oak.ireste.fr")
final static  int port = 7;
DatagramPacket paquet = new DatagramPacket(donnee,donnee.length,destination,port);
DatagramSocket monSocket = new DatagramSocket(port);  // création d'un socket
..
monSocket.send(paquet);                          // envoi du datagramme
..
monSocket.close();
```

La méthode *close()* permet de libérer les ressources qui sont associés à un socket.

Pour recevoir un datagramme, il faut disposer d'un objet datagramme et réserver la zone mémoire dans laquelle vont être stockées les données reçues.

```
public final static int MAX_DONNEE = 512;
byte[] donnee = new byte[MAX_DONNEE];
DatagramPacket paquet = new DatagramPacket(donnee,donnee.length);
DatagramSocket monSocket = new DatagramSocket(port);                  // creation d'un socket
..
monSocket.receive(paquet);
```

L'exemple ci-dessous montre une application côté client, permettant d'envoyer et de recevoir une suite d'octets qui "rebondissent" sur le service *echo* de la machine cible (ici serveur *echo*).

```
import java.net.*;
public class ClientEchoUDP {
```

Le constructeur *DatagramSocket* permet de préciser un port UDP et une adresse Internet à utiliser sur la machine locale et d'attacher le socket sur ces paramètres. Si l'utilisateur indique la valeur 0 pour *localPort*, le système d'exploitation va fournir une valeur par défaut (entre 1023 et 5000). La valeur *null* de *localAdr* permet de choisir l'adresse IP par défaut.

L'opération d'attachement s'effectue correctement à condition que le numéro proposé soit toujours disponible; dans le cas contraire une exception est levée.

Remarque: Dans le système UNIX, l'utilisateur "root" a le droit d'exploiter tous les numéros possibles

```
Exemple:  final static int portecho = 7;
  final static int marge = 12;
public static void main(String[] args) throws Exception {
   if(args.length != 2) {
     System.err.println("Usage: java EchoUDPClient <destination> <text>");
     System.exit(1);
   }
   byte buffer[] = args[1].getBytes();
   int length = args[1].length();
   InetAddress destination = InetAddress.getByName(args[0]);
   DatagramPacket packet = new DatagramPacket(buffer,0,length,destination,portecho);
   DatagramSocket socket = new DatagramSocket();
   System.out.println("Socket local: " +  socket.getLocalAddress() + ":" + socket.getLocalPort());
   socket.send(packet);
   System.out.println(length + " octets emis vers " + destination);
   packet.setData(new byte[length+marge],0,length+marge);
   System.out.println("Capacite de la zone de reception: " + packet.getLength());
   socket.receive(packet);                                          // reception du datagramme
   System.out.println(packet.getLength() + " octets recus: " + new String(packet.getData()));
   System.out.println("provenant de " + packet.getAddress() + ":"  + packet.getPort());
   socket.close();
  }
}
```

L'exemple suivant est un simple programme serveur qui reçoit les datagrammes et qui affiche les arguments des datagrammes reçus.

```
import java.net.*;
import java.io.*;
public class ServerSimpleUDP {
final static int portpardefaut = 7;                          // service ouvert par default
final static int TailleMax = 512;
public static void main(String[] args) throws Exception {
int port = portpardefaut;
byte[] tampon =new byte[TailleMax];
try {
        port = Integer.parseInt(args[0]);
        }
        catch(Exception e) {}
try {
        DatagramSocket socket = new DatagramSocket(port);
        DatagramPacket paquet = new DatagramPacket(tampon,tampon.length);
```

```
            while(true) {
try {
                socket.receive(paquet);
                String donnee = new String(paquet.getData(),0,paquet.getLength());
                System.out.println(paquet.getAddress() + " sur le port " + paquet.getPort());
                System.out.println("donnee recue: " + donnee);
                paquet.setLength(tampon.length);    }
              catch(IOException e) { System.err.println(e);}         }
            }
            catch(SocketException se){ System.err.println(se);}
}
}
```

Option de *timeout*

La méthode *receive()* dans le programme ci-dessus est, en principe, bloquante; le programme attend infiniment l'arrivée d'un datagramme. Heureusement, il existe une option de socket - *SO_TIMEOUT* permettant de libérer la méthode *receive()* après le temps de *timeout*.

Cette option doit être positionnée avant le lancement de la méthode *receive()*; mais après la création du socket.

```
            DatagramPacket paquet = new DatagramPacket(tampon,tampon.length);
DatagramSocket socket = new DatagramSocket(port);
...
socket.setSoTimeout(10000);                    // nombre de milisecondes avant le timeout
...
            socket.receive(paquet);
...
```

Datagrammes et *threads*

Les services réseau sont souvent sollicités par plusieurs clients. Pour pouvoir déployer plusieurs services du même type, il est intéressant d'exploiter le mécanisme du *thread*. L'exemple suivant montre une application client-serveur, où la fonction du serveur est exécutée dans un *thread*.

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

public class ServeurDaytimeUDP extends Thread {          // classe  thread du serveur
final static int portdaytime = 13;
DatagramPacket paquet;
byte[] tampon;
DatagramSocket socket;
int port;
public ServeurDaytimeUDP(int port) {
```

```
   this.port=port;
  }

public void run() {
tampon = new byte[1];
paquet = new DatagramPacket(tampon,0,1);
try {        socket = new DatagramSocket(port); }                    // ouverture d'un socket du thread
   catch(SocketException e) {}
while (true) {
try {        socket.receive(paquet);}
   catch(java.io.IOException e) {
            System.err.println("Probleme de reception "+ e);
            continue;
     }
     Date date = new Date();
     tampon = date.toString().getBytes();                    // conversion de la date en octets
     paquet.setData(tampon,0,tampon.length);
try { socket.send(paquet); }
    catch(java.io.IOException e) {
            System.err.println("Probleme d'emission "+ e);
            continue;
     }
   }
  }

public static void main(String[] args) throws Exception {
int port;
if(args.length == 0) port = portdaytime;
else port = Integer.parseInt(args[0]);
Thread serveur = new ServeurDaytimeUDP(port);        // creation du thread serveur
serveur.start();                                     // activation du thread du serveur
System.out.println("serveur actif");
 // ici le thread main peut continuer son execution
  }
}
```

Les applets et les datagrammes

Les mécanismes de sécurité interdisent aux applets de communiquer avec des machines différentes du serveur WEB sur lequel se trouve la page HTML intégrant l'applet. Néanmoins, le navigateur *Netscape* autorise l'utilisation d'une communication avec une station connectée au réseau à condition d'introduire le package *import netscape.security.** et ses classes. La méthode *PrivilegeManager.enablePrivilege("UniversalConnect");* permet d'afficher un avertissement et de valider une communication sur le réseau.

L’exemple suivant montre l’utilisation de la méthode *PrivilegeManager.enablePrivilege("UniversalConnect");* pour une applet qui sollicite le service *date* par l’envoi d’un datagramme vers une station (*oak*). Cette applet récupère le datagramme du retour et affiche son contenu dans une zone de texte.

```
import java.applet.*;
import java.awt.*;
import java.io.*;
import java.net.*;
import java.util.*;
import netscape.security.*;                    // les classes  necessaires pour la gestion d’acces

public class DaytimeUDPApplet extends Applet {
public static final int port = 13;
public final static int SIZE = 100;
DatagramSocket s;
DatagramPacket p;
TextArea outputarea;

public void init() {
outputarea = new TextArea();
outputarea.setEditable(false);
this.setLayout(new BorderLayout(20,20));
this.add("Center",outputarea);
try {
   PrivilegeManager.enablePrivilege("UniversalConnect");              // demande d’acces au reseau
   s = new DatagramSocket();
   InetAddress serverAddress = InetAddress.getByName("oak");
   byte[] b = new byte[SIZE];
   String str = "une date sur 26 caracteres";
   str.getBytes(0,str.length(),b,0);                                   // chargement du tampon b
   p = new DatagramPacket(b,str.length(),serverAddress,port);         // creation du datagramme
   s.send(p);                                                         // sollicitation du serveur
   s.receive(p);                                                      // reception du datagramme
   String date =new String(b,0,0,p.getLength());                      // conversion bytes => String
   outputarea.setText(date);                                          // affichage de la date recue
   s.close();
   }
  catch (IOException e) {this.showStatus(e.toString());}
  }
}
```

Multicast

Parmi les modes de communication en diffusion, il y a le *broadcast* et le *multicast*. Le *broadcast* diffuse le message vers tous les postes connectés au réseau; le *multicast* délivre les messages aux postes inscrits dans les groupes de diffusion.

Un groupe de diffusion est déterminé par une adresse IP de classe *multicast* (classe D). Un poste qui veut recevoir et/ou envoyer des datagrammes vers les membres d'un groupe doit rejoindre le groupe avant l'émission et/ou réception des messages. Au niveau d'un réseau local, la gestion du *multicast* est effectuée par le protocole IGMP. Ce protocole prévient le routeur (*m-router*) que son poste se joint à un groupe (ou le quitte) de *multicast*.

Un groupe de *multicast* est défini par une adresse Internet de *multicast* et un numéro de port UDP donné. Une application doit pouvoir disposer de l'adresse *multicast* du groupe et du numéro du port définissant le groupe. Les adresses *multicast* se situent entre 224.0.0.0 et 239.255.255.255. Certaines adresses sont réservées pour les protocoles de contrôle et de routage. Par exemple, les adresses qui commencent par 224.0.0. sont réservées pour le routage.

En Java, le *multicast* est basé sur la classe *DatagramPacket* et la classe *MulticastSocket*.

Réception

Pour recevoir un message *multicast*, le récepteur doit créer un *MulticastSocket* par le biais de son constructeur. Ensuite, le socket doit se joindre au groupe par la méthode *joinGroup()*. Cette méthode signale aux routeurs sur le chemin entre l'émetteur et le récepteur d'envoyer les messages du groupe.

Une fois le socket enregistré dans le groupe, il reçoit les datagrammes de classe *DatagramPacket* en faisant appel à la méthode *receive()*.

Pour quitter le groupe, il faut évoquer la méthode *leaveGroup()* sur le socket *multicast*.

Emission

Pour émettre un datagramme en mode *multicast*, l'application doit créer un *MulticastSocket* et un *DatagramPacket*. Puis elle doit insérer l'adresse *multicast* dans le datagramme. L'envoi du datagramme nécessite également la spécification du paramètre TTL (*time-to-live*): *multsock.send(paquet,TTL)*. La valeur 1 de ce paramètre signifie que le datagramme doit être distribué seulement sur le réseau local.

Les constructeurs *MulticastSocket*()

Il y a deux constructeurs du socket *multicast*:

- *MulticastSocket()*
- *MulticastSocket(port)*

Exemples:

```
try {
MulticastSocket multsock = new MulticastSocket();
// envoi d'un datagramme
```

```
}
catch(SocketException e) { System.err.println(e); }
```

```
try {
MulticastSocket multsock = new MulticastSocket(5000);
// reception d'un datagramme
}
catch(SocketException e) { System.err.println(e); }
```

Les datagrammes doivent être envoyés vers un port connu; d'où la nécessité de préciser le numéro du port sur lequel les datagrammes sont reçus.

Une fois, le *MulticastSocket* créé, plusieurs opérations peuvent être effectuées:

- rejoindre le groupe,
- envoyer un datagramme,
- recevoir un datagramme,
- quitter le groupe.

Dans une application de type récepteur, le *MulticastSocket* doit être explicitement lié au groupe du *multicast* avant de pouvoir recevoir les datagrammes.

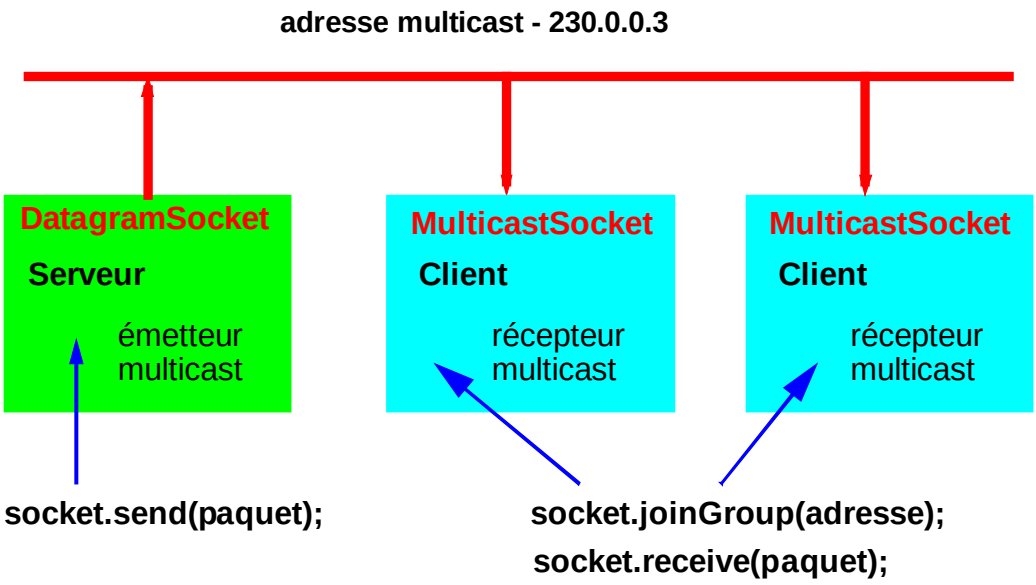
```
try {
MulticastSocket multsock = new MulticastSocket(5000);
InetAddress adresse = InternetAddress.getByName("228.42.22.2");    // une adresse multicast
multsock.joinGroup(adresse);        // attachement dans un groupe pour recevoir des datagrammes multicast
byte[] tampon = new byte[1024];
while(true) {
DatagramPacket paquet = new DatagramPacket(tampon,tampon.length);
multsock.receive(paquet);            // reception d'un datagramme multicast
String s = new String(paquet.getData(),0,paquet.getLength());    // extraction et conversion de la donnee
Dans une application de type émetteur, le datagramme est envoyé vers tous les membres du groupe adressés
par l'adresse du multicast. La méthode joinGroup() du MulticastSocket n'a pas besoin d'être appliquée pour
l'envoi des messages en multicast.System.out.println(s); }
}
catch(IOException e) { System.err.println(e); }
```

Dans une application de type émetteur, le datagramme est envoyé vers tous les membres du groupe adressés par l'adresse du *multicast*. La méthode *joinGroup()* du *MulticastSocket* n'a pas besoin d'être appliquée pour l'envoi des messages en *multicast*.

```
try {
int port = 5005;
int TTL = 1;                        // distance 1 signifie le reseau local
InetAddress adresse = InternetAddress.getByName("multicast.ireste.fr");  // une adresse multicast !!
byte[] tampon = "le message en multicast \n".getBytes();
DatagramPacket paquet = new DatagramPacket(tampon,tampon.length, adresse, port);
```

```
MulticastSocket multsock = new MulticastSocket();
multsock.send(packet,TTL);    // envoi d'un datagramme sur le reseau local par defaut TTL vaut 1
 }
}
catch(IOException e) { System.err.println(e); }
```

Ci-dessous nous présentons une simple application de type client-serveur. Le client reçoit les messages en *multicast*; le serveur génère les messages *multicast*.



Le programme complet du client-récepteur:

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

public class MulticastClient {
public static void main(String[] args) throws IOException {
MulticastSocket socket = new MulticastSocket(5005);
InetAddress adresse = InetAddress.getByName("230.0.0.3");
socket.joinGroup(adresse);
DatagramPacket paquet;
for (int i = 0; i < 100; i++)
        {               // attente d'un message en multicast
        byte[] tampon = new byte[512];
        paquet = new DatagramPacket(tampon, tampon.length);
        socket.receive(paquet);
        String received = new String(paquet.getData(),0,paquet.getLength());
            System.out.println("Received multicast message: " + received);
```

```
                    if(received.equals("end"))
                {           // fin de session multicast
                        System.out.println("End of multicast session");break;
                 }
               }
            socket.leaveGroup(adresse);
            socket.close();
          }
        }
```

Le programme complet du serveur-émetteur:

```
import java.io.*;
public class MulticastServer {
public static void main(String[] args) throws IOException {
new MulticastServerThread().start(); }
}
```

et son *thread*:

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

public class MulticastServerThread extends Thread {
   protected DatagramSocket socket = null;
   protected boolean encore = true;
   public MulticastServerThread() throws IOException {
      super("MulticastServerThread");
      socket = new DatagramSocket();
   }

public void run() {
while (encore) {
try {
    byte[] tampon = new byte[512];
    BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in));
                 String dString = null;
                System.out.println("Type new multicast message: ");
                dString = stdIn.readLine();
    if (dString.equals(""))   dString = new Date().toString();
            if (dString.equals("end")) {
                 System.out.println("End of multicast service!");et son thread:
```

```
                tampon = dString.getBytes();
                        InetAddress group = InetAddress.getByName("230.0.0.3");
                DatagramPacket packet = new DatagramPacket(tampon, dString.length(), group, 5005);
          socket.send(packet);
                try {
               sleep(2000);
                } catch (InterruptedException e) { }
et son thread:                  System.exit(0);
                }
tampon = dString.getBytes();
InetAddress group = InetAddress.getByName("230.0.0.3");
DatagramPacket packet = new DatagramPacket(tampon, dString.length(), group, 5005);
socket.send(packet);
                System.out.println("Wait for prompt !");
   try {
       sleep((long)(Math.random() * 5000));
       } catch (InterruptedException e) { }
       } catch (IOException e) { e.printStackTrace();   encore = false; }
      }
  socket.close();
  }
}
```

Le résultat d'une exécution:

Serveur:

```
Type new multicast message:
tata
Wait for prompt !
Type new multicast message:          // validation d'entree
Wait for prompt !
Type new multicast message:
tete
Wait for prompt !
Type new multicast message:
end
End of multicast service!
```

Client:

```
Received multicast message: tata
Received multicast message: Tue Nov 28 17:02:36 CET 2000
Received multicast message: tete
```

```
Received multicast message: end
End of multicast session
```

Résumé

Dans ce chapitre, nous avons étudié plusieurs exemples d'utilisation des datagrammes pour envoyer et recevoir des données. Toutes les communications nécessitaient l'instantiation d'un socket de type *DatagramSocket()* et d'un paquet datagramme de type *DatagramPacket()*. Le socket serveur doit être connu avant l'envoi des datagrammes; l'adresse et le numéro du port distant doivent être insérés dans chaque datagramme.

Nous avons également étudié le mécanisme de "*multicasting*" offert par les routeurs Internet récents. Ce mécanisme permet d'exploiter les adresses de classe D pour créer les groupes d'utilisateurs et envoyer les messages *multicast* vers tous les membres du groupe connectés au réseau à ce moment là.

Dans le chapitre suivant nous nous intéressons à la communication en mode connecté. Cette communication est basée sur le protocole TCP, donc elle est fiable. Le mode de fonctionnement connecté est asymétrique; les applications coté client utilisent les *sockets* clients (Socket) et les applications coté serveur utilisent les *sockets* de type serveur (*ServerSocket*).