Chapter 6

This chapter is the first of the chapters on network programming with Java language. An effective study of these chapters require some knowledge of computer networks and Internet protocols.
The course Computer Networks and the Internet Protocols are being developed to acquire this knowledge.

The socket mechanism is essential for the development of applications on the Internet. Each device connected to the network has an Internet address and each application that communicates over the network is attached to a service port. The couple "Internet address - port number" forms a socket.

In general, data transfer over a network can be delivered in two modes:

- connected mode
- datagram mode.

The Java language offers a set of socket mechanisms (classes and methods) to provide the two modes of communication.

We will study the communication mechanism based on the socket in order of increasing complexity of the underlying protocols:

- UDP protocol and datagram based communication mode
- TCP and communication in connected mode
- HTTP protocol and session based communication mode

UDP is the most simple transport protocol on the Internet. It offers the possibility of transmission of datagrams without ensuring the reliability of transfer. This solution is suitable for the on short distances where the physical media are very reliable. For long-distance transfers TCP becomes necessary.

Java language provides users with a set of socket classes in *java.net* package. The implementation of the communication on the UDP protocol is based on two classes:

- *DatagramSocket* and *DatagramPacket*

A *DatagramPacket* packet contains data to be sent in a datagram. The methods of the *DatagramSocket* class to send and receive datagrams *DatagramPacket* class.

The class *DatagramPacket*

A UDP datagram includes in its header the port numbers of source and destination. These two values are coded on short integers of 16 bits. The maximum value of a port number is 65 536. The data field of a UDP datagram can carry up to 65 507 bytes, actual size depends on the size of buffers and physical frames (e.g 512 bytes).

To build a datagram in Java we use the constructor *DatagramPacket()* with the initialization.

A *DatagramPacket* packet contains data to be sent/received in a datagram. The methods *send* and *receive* of the DatagramSocket class are used to send and receive the datagrams of DatagramPacket class.

The datagram prepared for the reception has two constructors:

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

When a datagram arrives, it is stored in the buffer from its position buffer [0].

Caution: The buffer must be at least of the size provided by the argument size, otherwise the constructor throws exception IllegalArgumentException.

A datagramme prepared for the transmission also has two constructors: :

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

The object of the InetAddress class must be provided by the user. It can be obtained in the application from a string "www.ireste.fr" or "193.52.81.10" by instantiating type:

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

The port number is an integer value. For known services, this value is between 1 and 1023.

Below a sequence of code that prepares a datagram to be sent to port 7 (echo):

```
String s = "Ceci est un test";
byte[] donnee = s.getBytes("ASCII");  // conversion ASCII to byte
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: The buffers in the datagrams are of type byte; the data type String must be converted to byte before their inclusion into the buffer (data).

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

The reverse operation is required to extract the string data from a datagram:

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

where: source is an object of class *DatagramPacket*

The following is a complete application that aims to illustrate different types of methods used to provide the preparation and reading of datagrams:

- *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 ASCII to byte
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);}}
}
```

Constructors prepare new datagrams to send, but in some cases it is interesting to modify the existing datagram.
The methods *setData(byte [] data)* and *setData(byte [] data, int offset, int size*) can change the contents of a datagram.

For example, if you want to send a file into several pieces, the same datagram can be sent several times by simply changing the data field.

```
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 of the data
}
```

The method *setAddress()* can change the address of the datagram before sending it. Thus the same datagram can be sent to multiple 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) – modifies the port
catch(Exception e) {}   }
```

Class *DatagramSocket*

In order to send or receive a datagram, it must prepare a local socket. This object is created by one of constructor class *DatagramSocket*.

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

The *DatagramSocket* constructor allows to specify a UDP port and an Internet address used on the local host and attach the socket with these parameters. If the user specifies the value 0 for *LocalPort*, the operating system will provide a default value (between 1023 and 5000). The value of null *localAdr* can be used to choose the default IP address.

The binding operation is carried out correctly provided that the proposed number is always available, otherwise an exception is thrown.

Note: In the UNIX system, the user "root" has the right to exploit all possible numbers

Example:

```
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 + " is used");  }
 }
} }
```

Sending and receiving of datagrams

When we have prepared a socket and a datagram, we simply combine them with *send()* method to send the datagram to the specified address and port.

```
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);  //  socket creation
monSocket.send(paquet);                              // sending datagram
..monSocket.close();
```

The method *close()* frees resources associated with a socket.

To receive a datagram, it must prepare a datagram and reserve the memory area where will be stored the received data.

```
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 of a socket
..
monSocket.receive(paquet);
```

The example below shows a client-side application, to send and receive a byte that "bounce" on the echo service on the target host (server echo).

```
import java.net.*;
public class ClientEchoUDP {
  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();
  }
}
```

The following example is a simple server program that receives the datagrams and displays the arguments of the received datagrams.

```
import java.net.*;
import java.io.*;
public class ServerSimpleUDP {
final static int portpardefaut = 7;                          // service by 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);}
}
}
```

timeout option

The receive () method in the above program is, in principle, blocking, the program lloks for the arrival of a datagram. Fortunately, there is a socket option - *SO_TIMEOUT* used to release the method *receive()* after the given timeout period.

This option must be set before launching the method receiv (), but after the creation of the socket.

```
        DatagramPacket paquet = new DatagramPacket(tampon,tampon.length);
DatagramSocket socket = new DatagramSocket(port);
...
socket.setSoTimeout(10000);                    // number of milisecondes before timeout
...
        socket.receive(paquet);
    ...
```

Datagrams and *threads*

Network services are often requested by many customers. To deploy multiple services of the same type, it is interesting to exploit the mechanism of the thread. The following example shows a client-server application where the server function is executed in a thread.

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

```
public class ServeurDaytimeUDP extends Thread {                // class of server  thread
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); }                    // socket of 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 date into bytes
     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
  }
}
```

The applets and the datagrams

The security mechanisms prohibit the applets to communicate with host of Web server different on which is the HTML page incorporating the applet. However, the Netscape browser allows the use of a communication with a terminal connected to the network on the condition to break the netscape.security with the specific classes.

The method *PrivilegeManager.enablePrivilege ("UniversalConnect")* displays a warning and validate a communication network.
The following example shows the use of this method for an applet requesting the service date by sending a datagram to a station (oak).

The applet retrieves the datagram back and displays its content in a text box.

```
import java.applet.*;
import java.awt.*;
import java.io.*;
import java.net.*;
import java.util.*;
import netscape.security.*;                    // access control

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");          // request of access
   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);
   p = new DatagramPacket(b,str.length(),serverAddress,port);
   s.send(p);
   s.receive(p);
   String date =new String(b,0,0,p.getLength());
   outputarea.setText(date);
   s.close();
```

```
    }
   catch (IOException e) {this.showStatus(e.toString());}
   }
  }
```

Multicast

Among the modes of datagram based communication, there are the unicast the broadcast and multicast. The broadcast message broadcast to all stations connected to the network, the multicast communication delivers multicast messages to posts in the mailing groups.

A multicast group is identified by an IP multicast class (class D). A station that wants to receive and/or send datagrams to group members must join the group before the show and/or receiving messages. At a LAN, multicast management is performed by IGMP. This protocol informs the router (m-router) that his post joins (or leave).a multicast group.

A multicast group is defined by an Internet multicast address and a given UDP port number. An application must have access to the multicast group address and port number identifying the group. Multicast addresses are between 224.0.0.0 and 239.255.255.255. Some addresses are reserved for control protocols and routing. For example, addresses starting with 224.0.0. are reserved for routing.

In Java, the multicast is based on class *DatagramPacket* and class *MulticastSocket*.

Reception

To receive a multicast message, the receiver must create a *MulticastSocket* through its constructor. Then, the socket must join the group by the method *joinGroup()*. This method notifies the routers to send group messages on the path between the transmitter and receiver(s).

Once the socket is registered in the group, it receives datagrams of class *DatagramPacket* using the method *receive()*. To leave the group, we must evoke the method *leaveGroup()* on the multicast socket

Transmission

To generate a datagram in multicast mode, the application must create a *MulticastSocket* and a *DatagramPacket*. Then she must insert the address in the multicast datagram. Sending datagram also requires the specification of parameter TTL (time-to-live): *multsock.send(packet, TTL)*. The value 1 of this parameter indicates that the datagram must be distributed only on the LAN ..

Constructors *MulticastSocket*()

There are two constructors of multicast socket:

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

Examples:

```
try {
MulticastSocket multsock = new MulticastSocket();
// sending a datagram
}
catch(SocketException e) { System.err.println(e); }
```

```
try {
MulticastSocket multsock = new MulticastSocket(5000);
// receiving a datagram
}
catch(SocketException e) { System.err.println(e); }
```

Datagrams must be sent to a known port, hence the need to specify the port number on which datagrams are received.

Once the MulticastSocket created several operations can be performed:

• join, • send a datagram, • receive a datagram, • leave the group.

In an application-type receiver the MulticastSocket should be explicitly linked to the multicast group before the reception of datagrams.

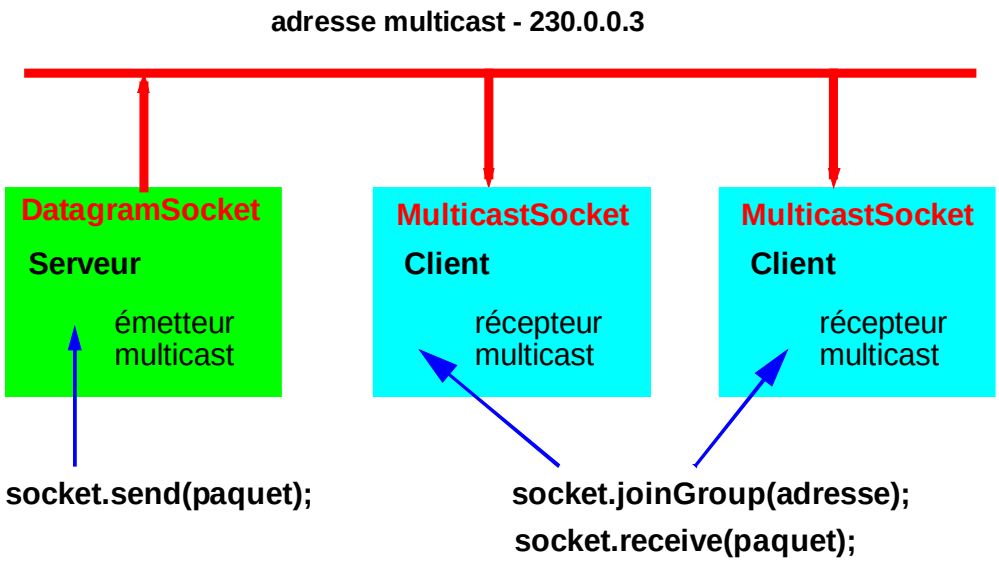
```
try {
MulticastSocket multsock = new MulticastSocket(5000);
InetAddress adresse = InternetAddress.getByName(“228.42.22.2”);    // une adresse multicast
multsock.joinGroup(adresse);         // attachement dans un groupe pour recevoir des datagrammes multicast
byte[] tampon = new byte[1024];
while(true) {
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
System.out.println(s); }
}
catch(IOException e) { System.err.println(e); }
```

In an application type transmitter , the datagram is sent to all members of the group identified by the multicast address. The method *joinGroup()* of *MulticastSocket* need not be first used for sending multicast messages.

```
try {
int port = 5005;
int TTL = 1;                         // distance 1 means local network
InetAddress adresse = InternetAddress.getByName(“multicast.ireste.fr”);  // multicast address !!
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);
 }
}
catch(IOException e) { System.err.println(e); }
```

Below we present a simple client-server application. The client receives the multicast message, the server generates the multicast messages



The full program of the client-receiver:

```
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++)
        {               // waiting for a multicast message
        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();
           }
        }
```

The full program of server-sender:

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

and its *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!");
            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) { }
                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();   }
}
```

The result of an execution:

Server:

```
Type new multicast message:
tata
Wait for prompt !
Type new multicast message:          // validation of input
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
```

Summary

In this chapter, we discussed several examples of the use of datagrams to send and receive data. All communications required the objects of type *DatagramSocket* socket and a packet of type *DatagramPacket*. The server socket must be known before sending datagrams; the address and the remote port number must be included in each datagram.

We also studied the mechanism of "multicasting" offered recently by the Internet routers. This mechanism allows exploiting the Class D addresses to create user groups and send multicast messages to all group members connected to the network at that time.

In the next chapter we focus on the communication in connected mode. This communication is based on TCP protocol, so it is reliable. The connected mode is asymmetrical; client side applications using client sockets (*Socket*) and server side applications using server-socket (*ServerSocket*).