Chapter 7

The mode “connected” of communication is a priori, supported by TCP. This protocol provides reliable communication; the data are transmitted as strings of bytes. Before making a transfer, a connection must be established between communicating stations. In principle, the client requests a connection to the server, the server accepts or rejects the request. If the request is accepted, a communication channel is established. For more details on TCP confer you to the presentation of Internet protocols at www.polytech2go.fr/mmm.

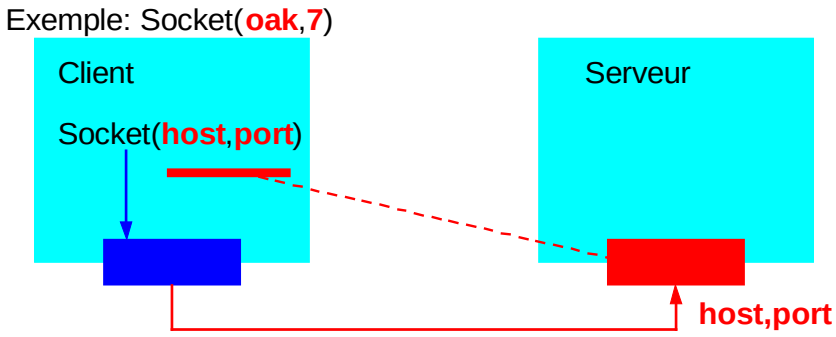
The Java package java.net .* contains all classes necessary for the development of applications running in connected mode.

Client socket

The client socket in connected mode is created by the Socket constructor. The constructor attaches the socket defaults to a local address and the available local port, then initializes the connection request to a remote station.

```
String host = “www.google.fr”;
int port=7;
Socket s = new Socket(host,port);
```

```
String host = “www.google.fr”;
InetAddress ia=InetAddress.getByName(host);
int port=7;
Socket s = new Socket(ia,port);
```



The following example shows an application with a socket client attempting to connect to various services on the host requested (*args [0]*).

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

public class PortScanner {
public static void main(String[] args) {
String host = "localhost";                    // host by default
if(args.length>1) host= args[0];
```

```
try {
   InetAddress adresse = InetAddress.getByName(host);
   for(int i=1; i<5000; i++)
   {
   try
    {
     Socket s = new Socket(adresse,i);
     System.out.println("Host: " + host + " a service on the port: " + i);
     s.close();
     }
     catch(IOException e){ System.err.println(e);}
   }
 }
  catch(UnknownHostException e) {System.err.println(e);}
}
}
```

Warning: The methods InetAddress.getByName (host) and new Socket (address) must catch the exception UnknownHostException and IOException respectively.

The constructor can specify the address and local port used by the socket:

public Socket(String remoteHost, remotePort int, InetAddress localInterface, LocalPort int);

The first two arguments are the same as in the example above, the arguments and *localInterface LocalPort* determine the local socket.

Methods *get*

Type methods applied to get a socket to extract the four arguments used by the constructor above, that is to say *remoteHost*, *remotePort*, *LocalInterface*, *LocalPort*.

```
Socket s = new Socket(“ireste.fr”,80);
InetAddress remoteHost = s.getInetAddress();
int remotePort = s.getPort();
InetAddress localInterface= s.getLocalAddress();
int localPort= g.getLocalPort();
```

Reading and writing to client socket

The methods *getInputStream()* and *getOutputStream()* can read and write streams of data associated with a socket. Usually, a simple stream class *InputStream* is filtered by *BufferedReader* and *InputStreamReader*. The stream output, *OutputStream*, *PrintWriter* is filtered.

The following example shows a simple client application that sends to server "echo" a string of characters entered on the keyboard. The server returns the same string.

```
import java.io.*;
import java.net.*;
public class ClientEchoTCP{
public static void main(String[] args) throws Exception{
 Socket echoSocket;
 PrintWriter out;
 BufferedReader in;
 echoSocket = new Socket("oak", 7);
 out = new PrintWriter(echoSocket.getOutputStream(),true);
 in = new BufferedReader(new InputStreamReader(echoSocket.getInputStream()));
 BufferedReader stdIn = new BufferedReader( new InputStreamReader(System.in));
 while(true)
  {
  String userInput = stdIn.readLine();    // reading the data
  if(userInput.equals("end")) break;
  out.println(userInput);
  out.flush();
  System.out.println(in.readLine());      // displaying the data
  }
 out.close();
 in.close();
 echoSocket.close();
 }
}
```

Options of *socket*

The socket options specify how to read and receive data. The four basic options are:

- *TCP_NODELAY*
- *SO_BINDADDR*
- *SO_TIMEOUT*
- *SO_LINGER*

The data sent by TCP pass through the output and input buffer, their delivery time can be quite long. The first option *TCP_NODELAY* is used for sending urgent data, for example, a byte indicating an interruption. An urgent byte is sent in a TCP segment without going through the queue in the buffer.

The example below type "test and set" cancels buffering (buffering) for bytes to follow:

```
if(!s.getTcpNoDelay()) s.setTcpNoDelay(true)
```

The reading of data by the *read()* method expects the number of bytes specified in its arguments. If the bytes requested fail, the program is blocked. *SO_TIMEOUT* option is used **to unlock** the *read()* method after a specified time period.

For example, the code below, tests the current value of the timeout, if it is zero, a value of 10 seconds is proposed.

```
if(s.getSoTimeout() == 0) s.setSoTimeout(10000);
```

In case of problems with execution, the two methods and *TCP_NODELAY SO_TIMEOUT* raise the exception *SocketException.*

Server Socket

The server socket is used to create an application server. A server running in connected mode waits for a connection request sent by the client socket. When this request is recorded and accepted by the server a socket connection is initialized. By accepting a connection on the *accept()* method , the server side creates a new communication socket (an "ordinary" socket) for performing the data transfer on the connection. At the end of the service, the server closes the communication socket and waits for the next connection request.

There are three constructors of class ServerSocket:

- *public ServerSocket(int port) throws IOException, (BindException)*
- *public ServerSocket(int port, int queueLength) throws IOException, (BindException)*
- *public ServerSocket(int port,int queueLength, InetAddress bindAddress) throws IOException*

The argument *QueueLength* requires the length of the queue for the connection requests. This value is normally smaller than 50.

Examples of constructors:

```
try {  ServerSocket echo = new ServerSocket(7);
}
catch (IOException e) { System.err.println(e); }
try {  ServerSocket echo = new ServerSocket(6969,40);
}
catch (IOException e) { System.err.println(e); }
```

The server runs in a cycle and accepts connections on a “round robin” basis. After accepting a connection request, the method *accept()* returns a Socket object to use in communication. At the end of the cycle the server must close the communication socket by the method close ().

Example of a cycle:

```
ServerSocket serveur = new ServerSocket(6969);
while(true) {
```

```
Socket connexion = serveur.accept();
OutputStreamWriter out = new OutputStreamWriter(connexion.getOutputStream());
out.write("connection is established");
connexion.close(); }
}
```

Warning: The accept()method is blocking. To unblock an accept() we must activate setSoTimeout(time) method . An integer passed as argument determines the maximum wait time in milliseconds.

```
ServerSocket serveur = new ServerSocket(6969,10);
serveur.setSoTimeout(3000);
Socket s = serveur.accept();     // blocked for 3 seconds maximum
```

The following example is more complete and robust due to the addition of test functions defined by the *try* and *catch*.

```
try {
  ServerSocket serveur = new ServerSocket (6969);
  while(true) {
  Socket connexion = serveur.accept();
  try {
    OutputStreamWriter out = new OutputStreamWriter(connexion.getOutputStream());
    out.write("la connexion est établie");  // ici ecrire la fonction du serveur
    connexion.close();  }
    catch(IOException e)  { } // error on connection
  finally {
    try { if(connexion != null) connexion.close();  }
    catch(IOException e) {}
    }
  }
  catch(IOException e) {System.err.println(e);}  // error on socket creation
  }
```

The example below is a simple echo server and it works in connected mode. It is enabled on the port number 7. It can read and write String data.

```
import java.io.*;
import java.net.*;
public class ServerEchoTCP{
public final static int echoport = 7;


public static void main(String[] args){
try {
  ServerSocket serveur= new ServerSocket(echoport);
  while(true) {
    try {
      Socket connexion = serveur.accept();
```

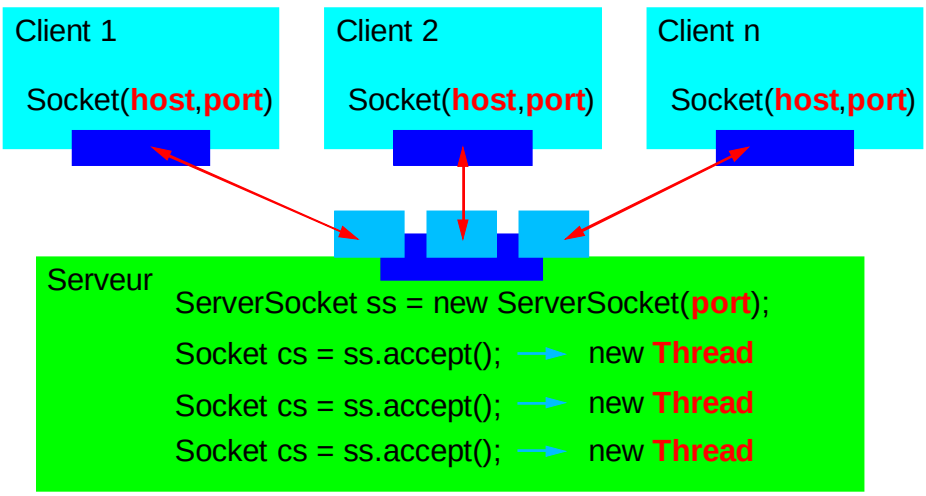
```
    BufferedReader in = new BufferedReader(new InputStreamReader(connexion.getInputStream()));
    PrintWriter out = new PrintWriter(connexion.getOutputStream(),true);
    String line;
    while ((line = in.readLine()) != null) {   out.println(line);  }  // fonction echo
    connexion.close();
    }
    catch(IOException e)  { } // erreur sur cette connexion
  }
 }
 catch(IOException e) {System.err.println(e);}
 }
}
```

Iterative and concurrent services

In our example of the echo server, the service is performed iteratively, that is to say that customer requests are made one after another. In the following example we present a concurrent server: reverse-echo.

In a concurrent server every connection request is performed separately by a thread initiated in parallel with others. In this solution several clients can be served "in parallel" without waiting for the services performed for other clients.

The main class *ServerThread* prepares the socket server and passes control to the method *run()*. The method *run()* loops to listen for connection requests and launches for each accepted connection a new thread - *ConnexionThread*. These threads are running concurrently on the same reverse-echo service. The strings of characters sent by the client are returned in reverse order.




```
import java.net.*;
import java.io.*;
public class ServerThread extends Thread{
```

```
public final static int defport=6969;
protected int port;
protected ServerSocket serveur;

public static void probleme(Exception e, String msg) {
   System.err.println("probleme: " + msg + "" + e);
   System.exit(1);
 }
public ServerThread(int port) {
if(port==0) port=defport;
try {  serveur = new ServerSocket(port);  }
catch(IOException e) { System.out.println(e,"creation of socket server");}
System.out.println("Server listens to the port: " + port);
this.start();
}

public void run() {
try {
  while(true) {                                                    // creation of threads for Connexion
    Socket connexion = serveur.accept();
    ConnexionThread ct = new ConnexionThread(connexion);   // services in parallel  }
  }
  catch(IOException e) {probleme(e,"sur connexion");}
}

public static void main(String[] args) {
int port = 0;
if(args.length ==1) port = Integer.parseInt(args[0]);
else port = 0;
new ServerThread(port);                                          // instantiation of principal thread
}
}

class ConnexionThread extends Thread {                    // thread of service
protected Socket client;
protected BufferedReader in;
protected PrintWriter out;
public ConnexionThread(Socket connexion) {
Socket client = connexion;
try {
     in = new BufferedReader(new InputStreamReader(client.getInputStream()));
     out = new PrintWriter(client.getOutputStream(),true);
  }
```

```
  catch(IOException e) {
  try { client.close();}  catch (IOException f) {};
  System.out.println("Problem on socket stream");
  return;  }
  this.start();
}

public void run() {
String ligne;
StringBuffer revligne;
int taille;
try
  {
  while(true) {
    ligne = in.readLine();
    if(ligne == null) break;
    taille = ligne.length();
    revligne = new StringBuffer(taille);
    for(int i=taille-1; i>=0;i--) revligne.insert(taille-1-i,ligne.charAt(i));
    out.println(revligne); }
  }
  catch(IOException e) {};
  try { client.close();}
  catch(IOException f){};
  }
}
```

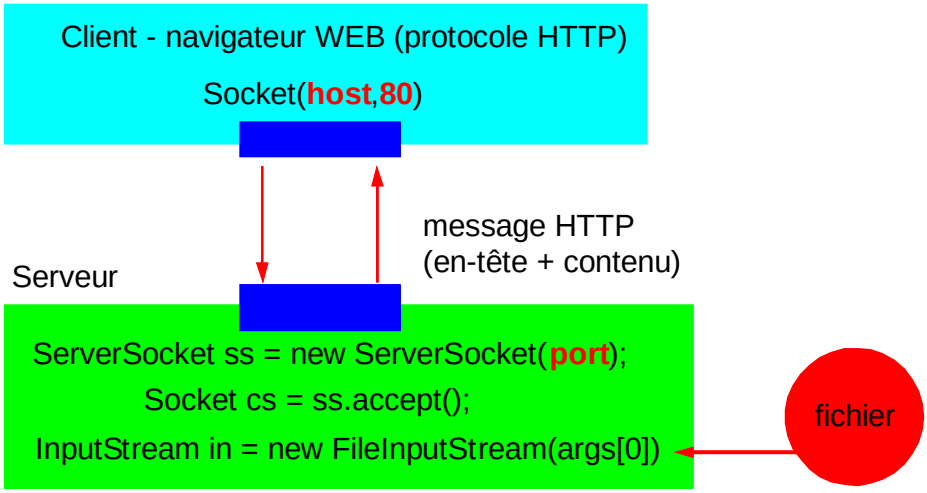
Simple http server (iterative)

The applications running in connected mode on the TCP protocol offer robust services that can carry data throughout the Internet system. The connected mode is used for transferring HTML pages which is by far the most popular service in the Internet world.

Below, before addressing the Java classes specially prepared for the direct use of HTTP, we will develop a very simple HTTP server. This server always sends the same file. The file name must be proposed as an argument to launch the server.

Simple http server (iterative)

The applications running in connected mode on the TCP protocol offer robust services that can carry data throughout the Internet system. The connected mode is used for transferring HTML pages which is by far the most popular service in the Internet world.



Below, before addressing the Java classes specially prepared for the direct use of HTTP, we will develop a very simple HTTP server. This server always sends the same file. The file name must be proposed as an argument to launch the server.

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

public class ServerSimpleHTTP extends Thread{
private byte[] contenu;
private byte[] entete;
private int port = 80;
public static void probleme(Exception e, String msg) {
   System.err.println("probleme: " + msg + "" + e);  System.exit(1);  }

public ServerSimpleHTTP(String data, String encoding, String MIMEType,int port)
throws UnsupportedEncodingException {
this(data.getBytes(encoding), encoding, MIMEType,port);
}

public ServerSimpleHTTP(byte[] data, String encoding, String MIMEType,int port)
 throws UnsupportedEncodingException {
 this.contenu = data;
 this.port = port;
 String entete = "HTTP 1.0 200 OK\r\n"                    // preparation of the http header
 + "Server: UnFichier 1.0\r\n"
 + "Content-length: " + this.contenu.length + "\r\n"
 + "Content-type: " + MIMEType + "\r\n\r\n";
```

```
 this.entete = entete.getBytes("ASCII");  }                    // conversion of the header into bytes
public void run() {
try {
   ServerSocket serveur = new ServerSocket(this.port);
   System.out.println("connection accepted on port: " + serveur.getLocalPort());
   System.out.println("data to send: ");
   System.out.write(this.contenu);
   while(true) {  Socket connexion=null;
    try {
      connexion = serveur.accept();
      OutputStream out = new BufferedOutputStream(connexion.getOutputStream());
      InputStream in = new BufferedInputStream(connexion.getInputStream());
      StringBuffer demande = new StringBuffer(80);
      while(true) {
        int c = in.read();                                          // reading of request
                  if(c=='\r' || c=='\n' || c== -1) break;
        demande.append((char) c);   }
      if(demande.toString().indexOf("HTTP/") != -1)
                                { out.write(this.entete); }          // sending the header
      out.write(this.contenu);                                      // sending the content
      out.flush();
      }
      catch (IOException e) {}
      finally { if(connexion != null) connexion.close(); }
      }
    }
     catch(IOException e) { probleme(e,"port occupe: ");}
 }

public static void main(String[] args) {                    // starting the server
try {
  String contenuType = "text/plain";
  if(args[0].endsWith(".html") || args[0].endsWith(".htm")) {
    contenuType = "text/html";  }
  InputStream in = new FileInputStream(args[0]);        // opening the file to send
  ByteArrayOutputStream out = new ByteArrayOutputStream();
  int b;
  while ((b = in.read()) != -1) out.write(b);
  byte[] data = out.toByteArray();
  int port;
  try {
    port = Integer.parseInt(args[1]);
    if(port<1 || port>65535) port=80; }                     // port by default - 80 : service http
```

```
  catch(Exception e) { port = 80; }
 String encoding = "ASCII";
 if(args.length>=2) encoding = args[2];
 Thread t = new ServerSimpleHTTP(data,encoding,contenuType,port);   // creation of the server thread
 t.start();                                                          // activation of the server  thread
 }
 catch(ArrayIndexOutOfBoundsException e) {
 System.out.println("utilisation: java ServerSimpleHTTP nomfichier port encoding");
 }
 catch(Exception e) { probleme(e,"");}
}
}
```

The server program above can be tested directly by the web browser. Just run the server with a file name, *. htm preference, and insert the IP address of the computer (localhost) in the browser (e.g. http://193.52.81.188).

Summary

The applications running in connected mode over the basic TCP protocol offer robust services that can carry the data throughout the Internet system. In reality, they represent the bulk of programs developed for the Internet.

In the next chapter we will find the classes for the development of Web-oriented applications. Specifically, we examine the Java classes of type URL that provide the ability to write applications based directly on the HTTP protocol.