Chapitre 7

Le mode de communication en connexion est, a priori, supporté par le protocole TCP. Ce protocole fournit une communication fiable; les données sont transmises comme chaînes d'octets. Avant de pouvoir effectuer un transfert, une connexion doit être établie entre les postes communicants. En principe, le poste client demande une connexion au serveur; le serveur accepte ou rejette cette demande. Si la demande est acceptée, un canal de communication est établi.

Pour plus de détails concernant le protocole TCP conférez vous à la présentation des protocoles Internet enseignée pendant .

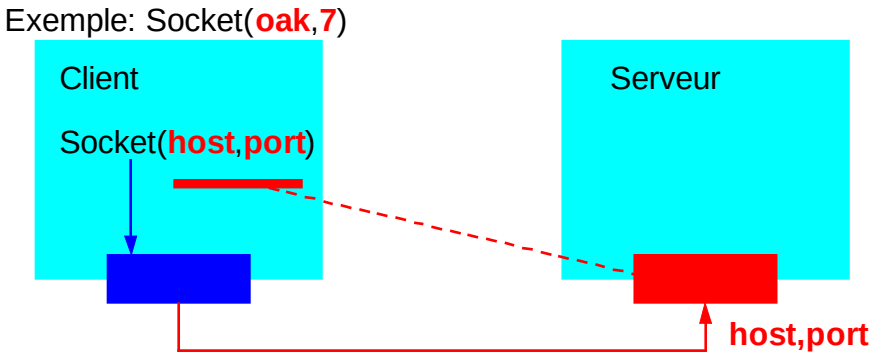
En Java le package*java.net.** contient toutes les classes nécessaires pour le développement des applications fonctionnant en mode connecté.

Socket client

Le socket client en mode connecté est créé par le constructeur *Socket*. Ce constructeur attache le socket par défaut à une adresse locale et à un port local disponible, et initialise la demande de connexion à un poste distant.

```
String host = "oak";
int port=7;
Socket s = new Socket(host,port);
```

```
String host = "oak";
InetAddress ia=InetAddress.getByName(host);
int port=7;
Socket s = new Socket(ia,port);
```



L'exemple suivant montre une application avec un socket client essayant de se connecter sur différents services sur le hôte demandé (*args[0]*).

L'exemple suivant montre une application avec un socket client essayant de se connecter sur différents services sur le hôte demandé (*args[0]*).

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

```
public class PortScanner {
public static void main(String[] args) {
String host = "localhost";                    // hote par defaut
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 + " un service sur le port: " + i);
     s.close();
     }
     catch(IOException e){ System.err.println(e);}
   }
 }
  catch(UnknownHostException e) {System.err.println(e);}
}
}
```

Attention: Les méthodes *InetAddress.getByName(host)* et *new Socket(adresse)* doivent capter respectivement l'exception *UnknownHostException* et l'exception *IOException*.

Le constructeur permet de préciser l'adresse et le port local à utiliser par le socket :

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

Les deux premiers arguments sont les mêmes que dans l'exemple ci-dessus; les arguments *localInterface* et *localPort* déterminent le socket local.

Méthodes *get*

Les méthodes de type *get* appliquées à un socket permettent d'extraire les quatre arguments utilisés par le constructeur ci-dessus, c'est-à-dire: *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();
```

Lecture et écriture dans le socket client

Les méthodes *getInputStream()* et *getOutputStream()* permettent de lire et d'écrire dans les *streams* de données associés à un socket. Habituellement, un *stream* simple de classe *InputStream* est filtré par *InputStreamReader* et *BufferedReader*. Le *stream* de sortie, *OutputStream*, est filtré par *PrintWriter*.

L'exemple suivant montre une simple application client qui envoie au serveur "*echo*" une chaîne de caractères saisie sur le clavier. Le serveur renvoie la même chaîne.

```
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);     // stream de sortie sur socket
 in = new BufferedReader(new InputStreamReader(echoSocket.getInputStream()));   // stream d'entree
 BufferedReader stdIn = new BufferedReader( new InputStreamReader(System.in));  // stream System.in
 while(true)
  {
  String userInput = stdIn.readLine();    // saisie des donnees
  if(userInput.equals("end")) break;
  out.println(userInput);
  out.flush();
  System.out.println(in.readLine());      // affichage des donnees
  }
 out.close();
 in.close();
 echoSocket.close();
 }
}
```

Options de *socket*

Les options *socket* spécifient comment lire et recevoir les données. Les quatre options de base sont:

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

Les données envoyées par le protocole TCP passent par les tampons de sortie et d'entrée; leur temps d'acheminement peut être assez long. La première option *TCP_NODELAY* est utilisée pour l'envoi de données urgentes; par exemple, un octet signalant une interruption. Un octet urgent est envoyé dans un segment TCP sans passer par la file d'attente dans les tampons.

L’exemple ci-dessous de type “test and set” annule la *bufferisation* (mise en tampon) pour les octets à suivre:

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

La lecture des données par la méthode *read()* attend le nombre d’octets spécifié dans ses arguments. Si les octets demandés n’arrivent pas, le programme est bloqué. L’option *SO_TIMEOUT* est utilisée pour débloquer la méthode *read()* après un laps de temps spécifié.

Par exemple, le code ci-dessous, teste la valeur actuelle du *timeout*; si elle est égale à zéro, une valeur de 10 secondes est proposée.

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

En cas de problème à l’exécution, les deux méthodes *TCP_NODELAY* et *SO_TIMEOUT* lèvent l’exception *SocketException*.

Socket serveur

Le socket serveur est utilisé pour créer une application serveur. Un serveur fonctionnant en mode connecté attend une demande de connexion envoyée par le socket client. Quand cette demande est enregistrée et acceptée par le socket serveur une connexion est initialisée. En acceptant une connexion sur la méthode *accept()* le serveur génère un socket de communication (un socket “ordinaire”) permettant d’effectuer le transfert des données sur la connexion. A la fin du service le serveur ferme le socket de communication et attend la demande de connexion suivante.

Il y a trois constructeurs de classe *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*

L’argument *queueLength* impose la longueur de la file d’attente de demandes de connexion. Cette valeur est normalement inférieure à 50.

Exemples de constructeurs:

```
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); }
```

Le serveur fonctionne dans un cycle et accepte des connexions à tour de rôle. Après l’acceptation d’une demande de connexion, la méthode *accept()* renvoie un objet *Socket* à utiliser dans la communication. A la fin du cycle le serveur doit fermer le socket de communication par la méthode *close()*.

Exemple d’un cycle:

```
ServerSocket serveur = new ServerSocket(6969);
while(true) {
Socket connexion = serveur.accept();
OutputStreamWriter out = new OutputStreamWriter(connexion.getOutputStream());
out.write("la connexion est établie");  // ici ecrire la fonction du serveur
connexion.close(); }
}
```

Attention: La méthode *accept()* est bloquante. Afin de pouvoir débloquer une attente sur *accept()* il faut lancer la méthode *setSoTimeout()*. Un entier passé en argument détermine le délai maximal d'attente exprimé en millisecondes.

```
ServerSocket serveur = new ServerSocket(6969,10);
serveur.setSoTimeout(3000);
Socket s = serveur.accept();     // attente bloquee sur 3 secondes au maximum
```

L'exemple suivant devient plus complet et robuste grâce à l'ajout de fonctions de test déterminées par les instructions *try* et *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)  { } // erreur sur cette connexion
  finally {
    try { if(connexion != null) connexion.close();  } // forcer la connexion si une erreur est survenue
    catch(IOException e) {}
    }
  }
  catch(IOException e) {System.err.println(e);}  // erreur sur la creation du socket serveur
  }
```

L'exemple ci-dessous est un simple serveur *echo*; il fonctionne en mode connecté et est activé sur le port numéro 7. Il permet de lire et d'écrire les chaînes de caractères - *String*.

```
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);}  // erreur sur la creation du socket serveur
}
}
```

Services itératifs et services concurrents

Dans notre exemple du serveur *echo*, le service est réalisé de façon itérative, c'est-à-dire que les demandes des clients sont effectuées l'une après l'autre. Dans l'exemple suivant nous présentons un serveur concurrent - *inverse-echo*.

Dans un serveur concurrent chaque connexion est traitée séparément par un *thread* lancé et en parallèle avec les autres. Dans cette solution plusieurs clients peuvent être servis "en parallèle" sans attendre la fin des services réalisés pour les autres clients.

La classe principale *ServerThread* prépare le socket du serveur et passe le contrôle à la méthode *run()*. La méthode *run()* boucle à l'écoute des demandes de connexion et lance pour chaque connexion acceptée un nouveau *thread* - *ConnexionThread*. Ces *threads* exécutent de façon concurrente le même service *inverse-echo*. Les chaînes des caractères envoyées par client sont renvoyées dans l'ordre inverse.

Services itératifs et services concurrents

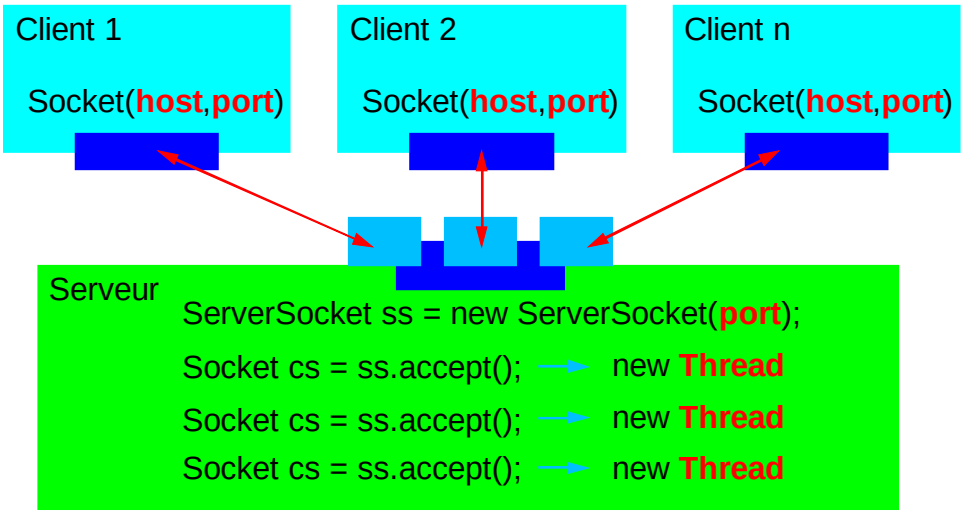
Dans notre exemple du serveur *echo*, le service est réalisé de façon itérative, c'est-à-dire que les demandes des clients sont effectuées l'une après l'autre. Dans l'exemple suivant nous présentons un serveur concurrent - *inverse-echo*.

Dans un serveur concurrent chaque connexion est traitée séparément par un *thread* lancé et en parallèle avec les autres. Dans cette solution plusieurs clients peuvent être servis "en parallèle" sans attendre la fin des services réalisés pour les autres clients.

Services itératifs et services concurrents

Dans notre exemple du serveur *echo*, le service est réalisé de façon itérative, c'est-à-dire que les demandes des clients sont effectuées l'une après l'autre. Dans l'exemple suivant nous présentons un serveur concurrent - *inverse-echo*.

Dans un serveur concurrent chaque connexion est traitée séparément par un *thread* lancé et en parallèle avec les autres. Dans cette solution plusieurs clients peuvent être servis "en parallèle" sans attendre la fin des services réalisés pour les autres clients.



La classe principale *ServerThread* prépare le socket du serveur et passe le contrôle à la méthode *run()*. La méthode *run()* boucle à l'écoute des demandes de connexion et lance pour chaque connexion acceptée un nouveau *thread* - *ConnexionThread*. Ces *threads* exécutent de façon concurrente le même service *inverse-echo*. Les chaînes des caractères envoyées par client sont renvoyées dans l'ordre inverse.La classe principale *ServerThread* prépare le socket du serveur et passe le contrôle à la méthode *run()*. La méthode *run()* boucle à l'écoute des demandes de connexion et lance pour chaque connexion acceptée un nouveau *thread* - *ConnexionThread*. Ces *threads* exécutent de façon concurrente le même service *inverse-echo*. Les chaînes des caractères envoyées par client sont renvoyées dans l'ordre inverse.

```
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) { probleme(e,"creation du socket serveur");}
System.out.println("Serveur en ecoute sur le port: " + port);
```

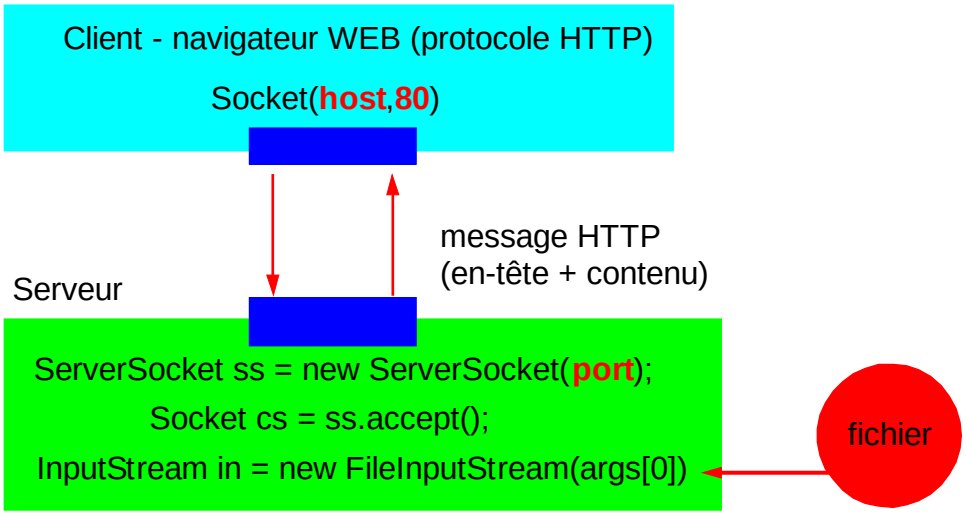
```
this.start();
}
        public void run() {
        try {
          while(true) {                                                    // creation des threads de Connexion
            Socket connexion = serveur.accept();
            ConnexionThread ct = new ConnexionThread(connexion);   // services en parallele  }
          }
          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 du thread principal
        }
        }
        class ConnexionThread extends Thread {                           // le thread de 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("Probleme sur socket streams");
          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 serveur HTTP (itératif)

Les applications fonctionnant en mode connecté à la base du protocole TCP offrent des services robustes qui peuvent s'étaler sur l'ensemble du système Internet. Le mode connecté est exploité pour le transfert des pages *html* qui est de loin le service le plus répandu dans le monde Internet.

Ci-dessous, avant aborder les classes Java préparées spécialement pour l'utilisation directe du protocole HTTP, nous allons développer un serveur HTTP très simple. Ce serveur envoie toujours le même fichier. Le nom du fichier doit être proposé comme argument au lancement du serveur.




```
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 de l'en-tete http
 + "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 de l'en-tete en octets
      public void run() {
      try {
         ServerSocket serveur = new ServerSocket(this.port);                  // creation du socket serveur
         System.out.println("une connexion acceptee sur port: " + serveur.getLocalPort());
         System.out.println("donnees a envoyer: ");
         System.out.write(this.contenu);
         while(true) {  Socket connexion=null;
          try {
            connexion = serveur.accept();                                     // acceptation d'une connexion
            OutputStream out = new BufferedOutputStream(connexion.getOutputStream());
            InputStream in = new BufferedInputStream(connexion.getInputStream());
            StringBuffer demande = new StringBuffer(80);
            while(true) {
              int c = in.read();                                              // lecture d'une requete de page
                        if(c=='\r' || c=='\n' || c== -1) break;
              demande.append((char) c);   }
            if(demande.toString().indexOf("HTTP/") != -1)
                                  { out.write(this.entete); }                 // envoi de l'en-tete
            out.write(this.contenu);                                          // envoi du contenu
            out.flush();
            }
            catch (IOException e) {}
            finally { if(connexion != null) connexion.close(); }
            }
          }
          catch(IOException e) { probleme(e,"port occupe: ");}
       }
```

```
public static void main(String[] args) {                    // demarrage de l'application serveur
try {
  String contenuType = "text/plain";
  if(args[0].endsWith(".html") || args[0].endsWith(".htm")) {
    contenuType = "text/html";  }
  InputStream in = new FileInputStream(args[0]);         // ouverture du fichier a envoyer
  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 par defaut - 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 du thread serveur
  t.start();                                                          // activation du thread serveur
  }
  catch(ArrayIndexOutOfBoundsException e) {
  System.out.println("utilisation: java ServerSimpleHTTP nomfichier port encoding");
  }
  catch(Exception e) { probleme(e,"");}
}
}
```

Le programme serveur ci-dessus peut être testé directement par le navigateur WEB. Il suffit de lancer le serveur avec un nom fichier,*.*htm* de préférence, et d'insérer l'adresse IP de l'ordinateur (*localhost*) dans le navigateur (e.g. http://193.52.81.188).

Résumé

Les applications fonctionnant en mode connecté à la base du protocole TCP offrent des services robustes qui peuvent s'étaler sur l'ensemble du système Internet. En réalité, elles représentent la plus grosse partie de programmes développés pour l'Internet.

Dans le chapitre suivant nous découvrirons les classes permettant le développement des applications orientées WEB. Plus précisément, nous étudierons les classes URL de Java qui donnent la possibilité d'écrire les applications en s'appuyant directement sur le protocole HTTP.