Applications Clients-serveurs: sockets TCP-IP
Implémentation en Java

1 Les sockets

Les sockets sont une interface de programmation de bas niveau de communication en réseau. Elles permettent d'envoyer des flots de données entre des applications ne se trouvant pas forcément sur la même machine.
Le paquetage java.net fournit une interface orientée objet simplifiée (plus simplifiée que les sockets en C) de manipulation de sockets.

Java utilise les sockets pour supporter trois classes de protocoles :

La classe Socket : protocole orienté connexion

La classe DatagramSocket : Protocole sans connexion

La classe MulticastSocket : Protocole multicast.

2 Le protocole TCP

Le protocole Socket fonctionne comme une conversation téléphonique. Après avoir établi une connexion, deux applications peuvent s'envoyer des paquets ou stream. La connexion est maintenue même en l'absence de paquets. Ce protocole garantit l'arrivée des paquets dans l'ordre de leur émission.

Les sockets sont habituellement utilisées pour écrire des applications clients/serveurs.
Le client est celui qui initie la communication.
Le serveur est celui qui accepte une requête émanant du client.
Concrètement :
Un client peut créer une socket pour initier la communication avec le serveur à tout moment.
Un serveur doit être en permanence à l'écoute et prêt à accepter une communication du client.
La classe java.net.Socket() représente une extrémité de connexion socket aussi bien sur le client que sur le serveur. Le serveur utilise en plus la classe java.net.ServeurSocket pour attendre d'éventuelles connexions.

Une application serveur crée un objet ServeurSocket et reste bloquée par un appel de la méthode accept() qu'une connexion se produise. La méthode accept() crée un objet Socket() pour communiquer avec le client. .

Un serveur peut communiquer avec plusieurs clients (séquentiellement) ou en paralllèle (plusieurs threads). Dans les deux cas il existe un seul objet ServeurSocket.

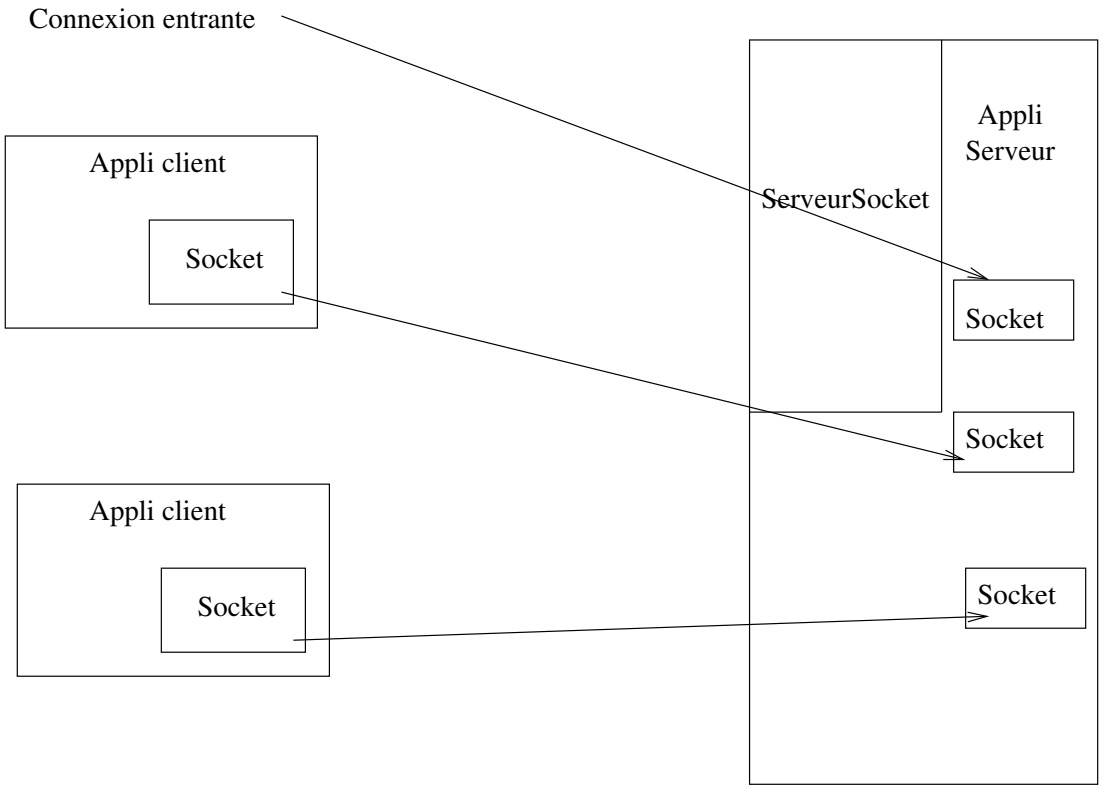

FIGURE 1 – Schéma général d'un Client Serveur.

2.1 La classe Socket

Un client a besoin de deux informations pour localiser le serveur :

1. Le nom de l'hôte (hôte) ;
2. le numéro de port.

Le client choisit le numéro de port affecté au service qu'il veut atteindre.
Pour avoir une connexion au niveau client il faut construire un objet Socket de la manière suivante :

Constructeur :

Socket (String host, int port) :
creation du socket sur le port et la machine hôte spécifiés.

Méthodes :

close()
ferme le socket.

OutputStream getOutputStream() :
renvoie un flux de sortie pour cet socket.

IutputStream getIutputStream() :
renvoie un flux de d'entrée pour cet socket.

Exemple 1 *Socket emission = new Socket (host, port) ;*
Socket rephttp = new Socket ("www.iut.univ-metz.fr", 80) ;

On peut aussi capturer les exceptions :

```
try{
Socket emission = new  Socket ("ariane.iut.univ-metz.fr", 25);
   }
catch(UnknownHostEXception e)
{System.out.println("Impossible de trouver la machine");
}
catch(IOException e)
{ System.out.println(" Impossible de se connecter ");
}
```

Remarque 1 *Ici le client gère le cas où la machine n'existe pas (problème de résolution de nom : UnknownHostEXception) et celui où la connection est refusée (IOException).*
On peut aussi créer une connexion en utilisant l'adresse IP : Socket emission = new Socket (192.134.56.200, 25) ;

Les sockets ainsi crées ont un flux de sortie et un flux d'entrée équivalents à System.out et System.in que l'on récupère avec get.Output.Stream et get.Input.Stream respectivement.

Exemple 2
```
Socket emission = new  Socket ("toto.titi.fr", 1234);
OuputStream out = emission,getOuputStream();
InputStream out = emission,getInputStream();
// Emission d'un octet
out.write(25);

ces flux  peuvent être utilisés directement pour lire et
écrire des bytes.
Mais pour gérer les chaînes de caractères il faut
 envelopper ces flux
dans des  PrintWriter pour  l'écriture
et des BufferedReader pour la lecture
 PrintWriter  out   new PrintWriter
(new OutputStreamWriter(socket.getOutputStream()));

   out.println("HTTP/1.0 200 "); // Version & status code
   out.println();  // End of response headers
   out.flush();

// Envoi de messages ligne par ligne
 PrintWriter out =  new PrintWriter
(new OutputStreamWriter(socket.getOutputStream()));

   out.println("Voici une ligne ");

   out.flush(
```

```
// Réception ligne par ligne
  BufferedReader in = new BufferedReader
(new InputStreamReader( socket.getInputStream()));
   String line;
   while((line = in.readLine()) != null) {
          if (line.length() == 0) break;
          out.println(line);
    }
```

2.2 La classe ServerSocket

L'emploi des sockets par un serveur en mode TCP nécessite la création d'une socket d'écoute. Pour celà on crée d'abord une socket d'écoute particulière grâce à la clase Serveur-Socket, socket qui écoute sur un port désigné. Cette socket ne servira pas à l'échange des données.
La methode "accept" attend la reception d'une connexion et crée une socket d'échange pour la communication avec ce client.

```
ServerSocket ecoute = new ServerSocket(port);
System.out.println('Servant recevant sur: " + ecoute.getLocalPort());

Socket client = ecoute.accept();
System.out.println("  Connexion de: " + client.getInetAddress());
BufferedReader in =
new BufferedReader(new InputStreamReader( client.getInputStream()));
...etc...
```

Constructeur :

ServerSocket (int port) :
creation du socket Serveur sur le port spécifié.

Méthodes :

close()
ferme le socket.

OutputStream getOutputStream() :
renvoie un flux de sortie pour cet socket.

IutputStream getIutputStream() :
renvoie un flux de d'entrée pour cet socket.

Socket accept() :
Ecoute si une connexion est demandée pour cet socket et l'accepte.

Exemple 3

```
ServerSocket ecoute = new ServerSocket(port);
System.out.println("Le serveur écoute sur le port:
 " + ecoute.getLocalPort());
Socket client = ecoute.accept();
```

```
System.out.println(" La connexion provient
 du client dont l'adresse
 est: " + client.getInetAddress());
BufferedReader in = new
 BufferedReader(new InputStreamReader
( client.getInputStream()));
 ...
```

3 classe MulticastSocket

Cette classe permet d'utiliser le multicasting IP pour envoyer des datagrammes UDP à un ensemble de machines repéré grâce à une addresse multicast (classe D dans IP version 4 : de 224.0.0.1 à 239.255.255.255).

3.1 Constructeurs

Les constructeurs de cette classe sont identiques à ceux de la classe DatagramSocket.

3.2 Abonnement/résiliation

Pour pouvoir recevoir des datagrammes UDP envoyés grâce au multicasting IP il faut s'abonner à une adresse multicast (de classe D pour IP version 4). De même lorsqu'on ne souhaite plus recevoir des datagrammes UDP envoyés à une adresse multicast on doit indiquer la résiliation de l'abonnement.

```
public void joinGroup(InetAddress adresseMulticast) throws IOException
```

Cette méthode permet de s'abonner à l'adresse multicast donnée.

A noter que un même socket peut s'abonner à plusieurs adresses multicast simultanément. D'autre part il n'est pas nécessaire de s'abonner à une adresse multicast si on veut juste envoyer des datagrammes à cette adresse.

Une IOException est générée si l'adresse n'est pas une adresse multicast ou si il y a un problème de configuration réseau.

```
public void leaveGroup(InetAddress adresseMulticast)
 throws IOException
```

Cette méthode permet de résilier son abonnement à l'adresse multicast donnée.

Une IOException est générée si l'adresse n'est pas une adresse multicast, si la socket n'était pas abonnée à cette adresse ou si il y a un problème de configuration réseau.

3.3 Choix du TTL

Dans les datagrammes IP se trouve un champ spécifique appelé TTL (Time To Live d'une durée de vie) qui est normalement initialisé à 255 et décrémentée par chaque routeur que le datagramme traverse. Lorsque ce champ atteint la valeur 0 le datagramme est détruit et une erreur ICMP est envoyé à l'émetteur du datagramme. Cela permet d'éviter que des datagrammes tournent infiniment à l'intérieur d'un réseau IP.

Le multicasting IP utilise ce champ pour limiter la portée de la diffusion. Par exemple avec un TTL de 1 la diffusion d' un datagramme est limitée au réseau local.

4 La classe InetAddress : La résolution de nom et d'adresse

Elle est mise en oeuvre à travers un service DNS ou un fichier local (/etc/hosts) ou les pages jaunes.

Les applications doivent utiliser les méthodes getLocalHost, getByName, ou getAllByName pour construire une nouvelle instance de InetAddress.

```
public static InetAddress getLocalHost() throws UnknownHostException
Cette méthode renvoie l'adresse IP du site local d'appel.

public static InetAddress getByName(String host)
 throws UnknownHostException
Cette méthode construit un nouvel objet
InetAddress à partir d'un
 nom textuel de  site. Le nom du  site est donné
 sous forme symbolique
 (iut.univ-metz.fr.fr) ou  sous forme numérique (195.83.142.200).

public static InetAddress[] getAllByName(String host)
   throws UnknownHostException
permet d'obtenir les différentes adresses IP d'un site.
```

4.1 Communication avec un serveur WWW en mode TCP

```
On crée une socket client pour se connecter avec
 un serveur sur le port 80.
Socket répertoire_HHTP = new Socket("www.iut.univ-metz.fr", 80);

On envoie des resquêtes de la forme:
"GET /~zineb/zineb.html"
ou    "GET /~zineb/zineb
```

4.2 Schéma général de conception d'une application serveur en mode TCP

Protocole serveur
créer un socket : new ServeurSocket(...)
attendre les demandes de connexion
entrantes : accept()
définir les I/O
getOutputStream() et getInputStream()
recevoir des paquets : in.readLine()
envoyer des paquets : out.println()
Traiter l'application en question
terminer la communication : close()

Protocole client
créer un socket : new socket(...)
définir les I/O
getOutputStream() et
getInputStream()
envoyer des paquets : out.println()
recevoir des paquets : in.readLine()
Traiter l'application en question
Terminer la communication : close()

4.3 Quelques applications

TP 1 : Un premier exemple : L'application Hello_World *Tester l'application Hello_ World en mode TCP dont le code est donné ci-après.*

TP 2 *Modifier le serveur Hello_ World pour qu'il accepte plusieurs clients en mode itératif (simple !)*

TP 3 *Ecrire un serveur parallèle (une classe Serveur.java) qu'on réutilisera dans la suite de ce TP qui consiste à éxécuter une application donnée en acceptant plusieurs clients en parallèle. Pour l'instant on ne se préoccupe pas de la classe Application. On suppose tout simplement l'exsitence de celle ci que l'on nommera Application.java.*

TP 4 *Ecrire un serveur Hello_ World Parallèle.*

TP 5 Communication avec un serveur Web
Écrire un client java qui communique avec un serveur WWW.
La communication sur le WWW se fait avec des Sockets STREAM. Un serveur WWW crée un ServerSocket sur le port 80 et interagit avec des browsers selon le protocole HTTP dont les commandes de base sont trés simples : une ligne de texte débutant avec un mot clé. On peut communiquer avec un serveur WWW en créant une Socket sur le port 80 et en envoyant des lignes de commande selon le protocole Http .

TP 6 Réalisation d'un serveur http
On veut réaliser ici un serveur http qui écoute sur un port désigné et qui répond aux requêtes HTTP GET.

Principe du serveur http

Chaque serveur de la toile (serveur Web) a un processus serveur qui écoute sur le port 80 TCP dans l'attente de connexions clients qui sont en général des navigateurs (Mozilla, Netscape, ...). Quand une connexion est établie, le client envoie une demande et le serveur envoie une réponse que le client affiche. Le protocole qui définit les demandes et les réponses est le protocole HTTP. Les différentes étapes de fonctionnement sont :

a) L'utilisateur clique sur une morceau de texte appelée URL (Uniform Ressource Locator) de la forme hppp ://adresse_serveur : numéro_port/Chemin. où l'on dégage trois parties :

1. *le nom du protocole (http)*
2. *le nom de la machine sur laquelle se trouve la page ainsi que le numéro de port (généralement port 80)*
3. *Le nom du fichier contennant la page HTML.*

b) Le navigateur détermine l'URL à partir de la sélection.

c) Il demande au serveur DNS l'adresse IP du serveur

d) Le serveur DNS répond

e) Le navigateur établit une connexion TCP avec le serveur sur le port 80.

f) Il envoie la requête HTTP GET qui est de la forme GET chemin HTTP/1.1

g) Le serveur analyse la requête syntaxiquement et extrait à partir de celle ci le chemin du fichier html.

h) Le serveur envoie alors le fichier html indiqué par le chemin bloc par bloc.

i) La connexion TCP est libérée

j) Le navigateur affiche tout le texte

Pour écrire le serveur http on vous demande d'utiliser :

1. *La classe Serveur.java écrit précédemment.*
2. *de développer la classe Application.java qui réalise les fonctions du protocole http côté serveur (c'est à dire les étapes g,h et j).*
   Pour réaliser l'étape g on utiliser les lignes de code suivantes.

```
// On utilise dans ce qui suit l'API java.util.regex
qui permet d'utiliser les
 expressions régulières ou regex
// comme on les connaît déjà dans les langages de script
 de type shell.
//  Cette API permet de traiter des morceaux de texte
 particuliers ou motifs
comme un analyseur lexical.

// La methode Pattern.compile permet de compiler
 un motif et la méthode
 matcher permet de créer un comparateur. matchs
 permet de vérifier si
la requête donnée sous forme de texte correspond
à l'expression régulière, c'est à dire si elle est
 correcte syntaxiquement.
Dans le cas où la requête
est correcte alors on extrait de celle ci ce qui nous intéresse,
 c'est à dire le chemin du fichier html.
Matcher get = Pattern.compile("GET (/?\\S*).*").matcher(requête);
if(get.matches()){
requête = get.group(1);


// Les deux lignes de code qui suivent peuvent être
aussi utilisés pour
 afficher le fichier index.html dans le cas où le
 chemin indiqué est un répertoire.
if(requête.endsWith("/") || requête.equals(""))
requête = requête + "index.html";
```

TP 7 Calcul du carré d'un nombre simple

Ecrire une application client serveur réalisant le carré d'un nombre.

TP 8 Réalisation de serveurs reposant sur des objets.
Pour aborder cette parite un peu délicate, on vous demande dans un premier temps d'analyser et de tester l'application client serveur dont le code est donné en annexe2(code tiré du livre Introduction à Java, P. Niemeyer et J. Knudsen, Editions O'Reilly). Prenez soin de découper la serveur en dégageant bien la partie commune à tous les serveurs parallèles (Serveur.java vu plus haut) et la partie Application.java.

TP 9 Résolution d'une équation du second degré
Réaliser suivant le modèle précédent (une fois bien compris) une application client serveur réalisant la résolution d'une équation du second degré.

TP 10 Application des sockets Multicast

- *Ecrire un serveur Multicast qui lit des chaînes à partir du clavier et qui les renvoie tous les clients membres d'un même groupe.*
- *Ecrire un chat en multicast*

5 Le Protocole UDP ou mode datagramme

La communication en mode UDP ou datagramme correspond à l'envoi de messages par la poste. Il n y a pas de garantie d'arrivée des messages expédiés, à moins que le destinataire n'envoie de façon explicite un accusé de réception.

5.1 La classe DatagramSocket()

Pour envoyer des datagrammes au niveau il faut créer un objet de type DatagramSocket de la manière suivante : **Constructeur**
DatagramSocket dsock_in = new DatagramSocket(port)
Création d'une socket UDP sur le port spécifié pour recevoir

ou

DatagramSocket dsock_out = new DatagramSocket()
Création d'une socket UDP sur libre pour émettre

Il faut aussi créer un objet de type DatagramPacket (tableau d'octets) comme suit :
Pour recevoir
DatagramPacket packet_in = new DatagramPacket(buffer, buffer.length) ;

Pour recevoir
DatagramPacket packet_out = new DatagramPacket(buffer, buffer.length, address, port) ;

Pour émettre des paquets
dsock_out.send(packet_out) ;

Pour recevoir des paquets des paquets
dsock_in.receive(packet_in) ;

```
Exemple
Message reçu
String message_recu  = new String(buffer,0,packet.getLength());
Host Nom_hote: packet_in.getAddress().getHostName() );
Adresse Internet: packet_in.getAddress().getHostAddress() );
Port émetteur Port: packet_.getPort());

Réponse:
dsock_out.send( new DatagramPacket
( "HELLO".getbytes(), "HELLO".length
( packet_out.getAddress(),packet_out.getPort());
```

5.2 Schéma général de conception d'une application client serveur en mode UDP

Protocole serveur :
créer un socket : dsocket = new DatagramSocket(...)
définir les buffers d'E/S
recevoir des paquets : dsocket.receive
envoyer des paquets : dsocket.send
Traiter l'application en question
terminer la communication : close()

Côté client :
créer un socket : dsocket = new DatagramSocket(...)
définir les I/O
envoyer des paquets : dsocket.send
recevoir des paquets : dsocket.receive
Traiter l'application en question
terminer la communication : close()

5.3 Quelques applications

TP 11 Un simple client serveur en mode UDP

```
// Programme Serveur.java
import java.io.*;
import java.net.*;

public class Serveur
{    public static void main(String args[])
    {
   DatagramSocket dsocket ;
// Création d'un buffer d'octets pour recevoir des données.
     byte[] buffer = new byte[2048];


  try {
       // Créer une socket pour écouter sur le port spécifié
  int port = Integer.parseInt(args[0]);
 dsocket = new DatagramSocket(port);
  System.out.println("\nLocal Host is:
"    + InetAddress.getLocalHost());
   System.out.println("Serveur écoutant sur le port:
 " + dsocket.getLocalPort());


   //  Boucle pour attendre de recevoir de nouveaux paquets  et

  for(;;) {
    // Créer un paquet avec un buffer vide pour recevoir les données
DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
```

```
  System.out.println("\n>>> Serveur en attente de datagrammes .....");
   dsocket.receive(packet);

  // Extraire le message reçu


     String msg = new String(buffer,0,packet.getLength());

     System.out.println(packet.getLength() +
                     " octets recu de:
" + packet.getAddress().getHostName() );
        System.out.println("Message: " + msg);
        System.out.println(" Adresse Internet:
 " +packet.getAddress().getHostAddress() );
        System.out.println(" Port Emetteur:
" + packet.getPort());

        if (msg.startsWith("QUIT"))
         break;
      }
      dsocket.close();
    }
    catch (Exception e) {
      System.err.println(e);

    }
  }
}


// Programme Client.java en mode UDP
import java.io.*;
import java.net.*;


public class Client{
  public static void main(String args[]) {
    try {
      // Contrôle le nombre d'arguments
      if (args.length < 3)
        throw new IllegalArgumentException
("Nombre d'arguments insuffisant ! ");

      // Analyse des arguments

      String host = args[0];
      int    port = Integer.parseInt(args[1]);
```

```
      //  Construire le message à envoyer à l'aide des arguments
//   3, 4 et plus.

      byte[] message;
      String msg = args[2];

      for (int i = 3; i < args.length; i++) msg += " " + args[i];
      message = msg.getBytes();

      // Obtenir l'adresse internet du hôte spécifié
      InetAddress address = InetAddress.getByName(host);

      // Initialiser un paquet datagramme avec
 les données et l'adresse.
      DatagramPacket packet = new DatagramPacket
(message, message.length, address, port);

      // Créer un datagramme socket et le paquet à
 travers cette socket.
      DatagramSocket dsocket = new DatagramSocket();
      dsocket.send(packet);
      dsocket.close();
    }
    catch (Exception e) {
      System.err.println(e);

    }
  }
}
```

Annexe1

```
// Le client Client.java
import java.io.*;
import java.net.*;
public class Client {
public static void main(String argv[]) throws Exception {

// Le client se connecte au site donné en premier argument
// et sur un numéro de port donné en deuxième argument
Socket emission = new Socket(argv[0], Integer.parseInt(argv[1]));

System.out.println("SOCKET = " + emission );
System.out.println("Socket de connexion:
 " + emission.getLocalPort());

// Création des flux entrants et sortants
 BufferedReader in = new BufferedReader(
 new InputStreamReader(emission.getInputStream()) );

PrintWriter out = new PrintWriter(
  new BufferedWriter(
  new OutputStreamWriter(emission.getOutputStream())),
                            true);

// Envoi de la chaîne Hello_World
 String str = "Hello_World";
out.println(str);

// Lecture du message echo (Hello_World)
str = in.readLine();
System.out.println(in.readline);

out.close();
in.close();
emission.close();
}
}


// Le serveur Serveur.java
import java.io.*;
import java.net.*;

public class Serveur {
public static void main(String argv[]) throws Exception {
```

```
// Le serveur crée une socket d'écoute sur le port
 indiqué comme premier argument
ServerSocket ecoute =
 new ServerSocket(Integer.parseInt(argv[0]));
System.out.println("\n Le serveur reçoit sur le port : "
 + ecoute.getLocalPort());

System.out.println(">>> Serveur prêt!! ");

// Le serveur crée une socket d'échange sock_com
Socket soc_com = ecoute.accept();
System.out.println(" " + soc_com.getInetAddress());


// Le client qui s'est connecté est connu par
 getInetAddress et getHostName
InetAddress origin = soc_com.getInetAddress();
System.out.println("Connection venant de:
 " + origin.getHostName());


// Un BufferedReader permet de lire par ligne.
 BufferedReader in = new BufferedReader(
   new InputStreamReader(soc_com.getInputStream())
                              );

 // Un PrintWriter possède toutes les opérations print classiques.
 // En mode auto-flush, le tampon est vidé (flush) à l'appel de println.
  PrintWriter out = new PrintWriter( new BufferedWriter(
  new OutputStreamWriter(soc_com.getOutputStream())),
                             true);


  String str = in.readLine(); // lecture du message

  System.out.println("Message reçu  = " + str);
  out.println(str);     // renvoi du message reçu écho

    in.close();
        out.close();
        soc_com.close();
   }
}
```

Annexe2

```
// 1.  FICHIER  Requête.java
public class Request implements java.io.Serializable{}

// 2. FICHIER  DateRequest.java


// 3. FICHIER  DateRequest.java
public  abstract class WorkRequest extends Request {
public abstract Object execute();
}

// 4. FICHIER  carre.java

public class carre extends WorkRequest {
int n;

public carre (int n) {
this.n =n;
}

public Object execute() {
return new Integer (n*n);
}
}



// 5.  FICHIER  serveur.java
import java.io.*;
import java.net.*;
import java.util.*;
public class serveur {
public static void main(String argv[])throws IOException  {
ServerSocket ecoute = new ServerSocket(Integer.parseInt(argv[0]));
System.out.println("Le serveur reçoit sur le port: " + ecoute.getLocalPort());

while(true)
new ServeurConnexion(ecoute.accept()).start();
}
}

// Fin de la classe serveur


class ServeurConnexion extends Thread {
```

```
Socket client;
ServeurConnexion(Socket client ) throws SocketException {
this.client = client;
}

public void run() {
try  {
ObjectInputStream in  =
new ObjectInputStream(client.getInputStream());
ObjectOutputStream  out =
new ObjectOutputStream(client.getOutputStream());
while(true) {
out.writeObject(traiteRequete(in.readObject()));
out.flush();}
}  catch(EOFException e3) {//  Fin de fichier normale
try{
client.close();
}  catch( IOException  e) { }
}  catch(IOException e) {
System.out.println("Erreur d'entree sortie " +e );
}  catch(ClassNotFoundException e2) {

System.out.println(e2); // Le type de l'objet demandé est inconuu
}
}
private Object traiteRequete(Object requete) {

if (requete instanceof DateRequest )
return new java.util.Date();
else if (requete instanceof WorkRequest )
return ((WorkRequest)requete ).execute();
else return null;
}
}

// 6. Fichier CLIENT.java
import java.io.*;
import java.net.*;
import java.util.*;

public class client {
public static void main(String argv[]) {
try{
Socket emission = new Socket(argv[0], Integer.parseInt(argv[1]));

// System.out.flush();
```

```
ObjectOutputStream  out =
new ObjectOutputStream(emission.getOutputStream());
ObjectInputStream in  =
new ObjectInputStream(emission.getInputStream());



System.out.println("Connexion établie ");

//  Lancer le calcul du carré
System.out.println(" Calcul du carré ");
System.out.flush();

out.writeObject(new carre(2));
out.flush();

// Résultat du carré
System.out.println(" Résultat du  carré ");
System.out.flush();

System.out.println(in.readObject());

emission.close();
out.flush();
} catch( IOException  e) {
System.out.println("Erreur d'entree sortie " +e );
}   catch(ClassNotFoundException e2) {
System.out.println(e2);
}
}
}
```