

PROBLÈME I (sockets)

Vous avez à réaliser avec des sockets en Java un répartiteur de charge (une classe LoadBalancer) qui reçoit des requêtes HTTP et les distribue de façon aléatoire vers un ensemble de serveurs Web. Le début de la classe LoadBalancer est le suivant :

```
public class LoadBalancer {
    static String hosts[] = {"host1", "host2"};
    static int ports[] = {8081,8082};
    static int nbHosts = 2;
    static Random rand = new Random();
    ...
}
```

A chaque réception d'une requête HTTP, LoadBalancer transfère la requête à un des serveurs web (les adresses de ces serveurs sont données par les tables hosts et ports) et LoadBalancer transfère le résultat de la requête à l'émetteur. Le choix du serveur Web est aléatoire (rand.nextInt(nbHosts) retourne un entier entre 0 et nbHosts-1). Pour être efficace, LoadBalancer est évidemment multi-threadé.

Pour la programmation des entrées/sorties, on utilisera :
- InputStream : public int read(byte[] b); // bloquante, retourne le nombre de bytes lus
- OutputStream : public void write(byte[] b, int off, int len); // écrit les len bytes à la position off et on suppose que les requêtes et réponses sont lues ou écrites en un seul appel de méthode avec un buffer de 1024 bytes.

```
Java
public class LB {

    static String hosts[] = {"host1", "host2"};
    static int ports[] = {8081,8082};
    static int nbHosts = 2;
    static Random rand = new Random();

    public static void main(String args[]) {

        ServerSocket ss = new ServerSocket(8080);

        while(true) {

            Socket s = ss.accept();
            Slave slave = new Slave(s);
            slave.start();
        }
    }

    class Slave extends Thread {

        Socket s1;

        public Slave (Socket s) {

            this.s1= s;
        }

        public void run() {
```

```
int n = LB.rand.nextInt(LB.nbHosts);
Socket s2 = new Socket(LB.hosts[n], LB.ports[n]);

InputStream s1_in = s1.getInputStream();
OutputStream s1_out = s1.getOutputStream();

InputStream s2_in = s2.getInputStream();
OutputStream s2_out = s2.getOutputStream();

byte[] buff = new byte[1024];
int nblu;

// Transfert client -> serveur
nblu = s1_in.read(buff);
s2_out.write(buff, 0, nblu);

// Transfert serveur -> clien
nblu = s2_in.read(buff);
s1_out.write(buff, 0, nblu);

s2.close();
s1.close();
}
}
```

PROBLÈME II (sockets)

Losrque des abonnés à Internet sont connectés par la technologie ADSL (Asymétric Digital Subscriber Line), le débit montant (upload) est nettement inférieur au débit descendant (download). Ceci implique que si UsagerClient télécharge un gros document donné par UsagerServer, le transfert est limité par le débit montant (upload) de UsagerServer. Pour diminuer le temps de transfert, il est possible pour UsagerClient de lancer le téléchargement de différents fragments du même document à partir de plusieurs usagers possédant une copie du document. Cette technique est utilisée par de nombreuses plate-formes de téléchargement dites P2P.

Nous voulons implanter le schéma de principe de cette technique. Vous devez programmer avec des sockets en Java deux programmes exécutés respectivement sur le client et sur les machines serveurs :
- UsagerClient télécharge des fragments d'un document depuis plusieurs serveurs
- UsagerServer_i fournit les fragments du document

On suppose que les programmes UsagerClient et UsagerServer_i ont les variables suivantes définies :

```
final static String hosts[] = {"host1", "host2", "host3"};
final static int ports[] = {8081,8082,8083};
final static int nb = 3;
static String document[] = new String[nb];
```

Les 3 premières variables donnent les adresses des serveurs. Un serveur démarre en prenant en paramètre son numéro de serveur (0, 1 ou 2, l'indice dans les tables hosts et ports). On ne gère qu'un seul document qui est ici un tableau de String

(document) et on considère qu'un fragment est une String de ce tableau. On suppose que le tableau document est initialisé pour les serveurs (vous n'avez pas à le faire) et il doit être rempli par le téléchargement pour le client.

On simplifie donc ce schéma avec 1 document composé de 3 String, téléchargées depuis 3 serveurs.

Pour les requêtes, il est recommandé d'utiliser la sérialisation d'objets Java. Une requête peut simplement envoyer un numéro de fragment sur une connexion TCP et recevoir la String correspondant au fragment.

Donnez une implantation des programmes UsagerClient et UsagerServer_i en suivant ce schéma.

```
public class Client extends Thread {

    final static String hosts[] = {"host1", "host2", "host3"};
    final static int ports[] = {8081,8082,8083};
    final static int nb = 3;

    static String document[] = new String[nb]; // vide au début

    int myno;

    public Client (int n) {

        this.myno = n;
    }

    public static void main (String args[]) {

        for (int i=0; i<nb; i++) {

            Client cli = new Client(i);
            cli.start;
        }

    }

    public void run () {

        Socket s = new Socket(hosts[myno], ports[myno]);

        InputStream is = s.getInputStream();
        OutputStream os = s.getOutputStream();

        // Permet de lire et envoyer des objets (int, String) (sérialization)
        ObjectInputStream ois = new ObjectInputStream(is);
        ObjectOutputStream oos = new ObjectOutputStream(os);

        oos.writeObject(myno); // Envoi de la requête
        document[myno] = (String) ois.readObject(); // Récupération de la réponse
    }
}
```

```

public class Server {

    static String document[] = new String[nb]; // plein au début

    public static void main (String args[]) {

        int myport = Integer.parseInt(args[0]);

        ServerSocket ss = new ServerSocket(port);

        while (true) {

            Socket s = ss.accept();

            InputStream is = s.getInputStream();
            OutputStream os = s.getOutputStream();

            // Permet de lire et envoyer des objets (int, String) (s rialization)
            ObjectInputStream ois = new ObjectInputStream(is);
            ObjectOutputStream oos = new ObjectOutputStream(os);

            int nb = (int) ois.readObject();
            oos.writeObject(document[nb]);

        }
    }
}

```

PROBL ME III (RMI)

Vous avez   r aliser un service d'ex cution de commande   distance (comme rsh) en utilisant Java RMI.

Ce service est compos  de 2 programmes :

- un programme Daemon.java qui doit  tre lanc  sur toutes les machines vers lesquels on veut pouvoir lancer des commandes   distance. Ce programme enregistre un objet RMI sur son RMIRegistry local.
- Un programme RE.java qui permet l'ex cution d'une commande   distance. Ce programme interagit avec le programme Daemon du site sur lequel il faut ex cuter la commande.

Le programme RE s'utilise de la fa on suivante : `java RE <host> <command>`

Le programme RE re oit dans `args[0]` et `args[1]` les 2 chaines de caract res `host` et `command`.

Pour ex cuter une commande localement sur les machines distantes, on utilise : `localExec(String cmd);`

Question 1 : Donnez une implantation en Java des programmes Daemon et RE

```

import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

// C t  serveur
public interface Daemon extends Remote {

    // M thode appell e pour ex cuter la commande
    public void doIt (String command) throws RemoteException;

}

public class DeamonImpl extends UnicastRemoteObjects implements Daemon {

    public DeamonImpl() throws RemoteException {

    }

    public void doIt (String command) throws RemoteException {

        // Si on suit le sujet, on ex cute : localExec(String command)
        // L  pour que  a fonctionne, on fait juste un print
        System.out.println("received command");

    }

    public static void main(String args[]) {

        try {

            Daemon d = new DeamonImpl();
            Naming.rebind("//localhost/toto", d); // Export dans le registry

        } catch (Exception ex) {
            ex.printStackTrace();
        }

    }

}

// C t  client
public class RE {

    public static void main(String args[]) {

        String host = args[0];
        String command = args[1];

        try {

            Daemon d = // Import depuis le registry
                (Daemon) Naming.lookup("//" + host + "toto");
            d.doIt(command);

        } catch (Exception ex) {
            ex.printStackTrace();
        }

    }

}

```

Question 2 : On veut maintenant g rer l'affichage console de la commande ex cut e   distance. On veut que les affichages de la commande soient effectu s sur la console du client qui a lanc  la commande.

On ex cute maintenant une commande localement avec : `localExec(String command, Console console);`

Console est une interface Java qui fournit notamment la m thode `println(String s);`

La commande ex cut e r alise ses affichages en appelant cette m thode sur l'objet Console pass  en param tre de `localExec()`.

Modifiez votre implantation pr c dente pour permettre l'affichage sur le poste client de la commande ex cut e   distance

```

import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

// C t  serveur
public interface Daemon extends Remote {

    // M thode appell e pour ex cuter la commande
    public void doIt (String command, Console c) throws RemoteException;

}

public class DeamonImpl extends UnicastRemoteObjects implements Daemon {

    public DeamonImpl() throws RemoteException {

    }

    public void doIt (String command, Console c) throws RemoteException {

        // Si on suit le sujet, on ex cute : localExec(command, c)
        // L  pour que  a fonctionne, on fait juste un print
        c.out.println("received command");

    }

    public static void main(String args[]) {

        try {

            Daemon d = new DeamonImpl();
            Naming.rebind("//localhost/toto", d); // Export dans le registry

        } catch (Exception ex) {
            ex.printStackTrace();
        }

    }

}

// C t  client
public class RE {

    public static void main(String args[]) {

        String host = args[0];
        String command = args[1];

        try {

            Daemon d = // Import depuis le registry
                (Daemon) Naming.lookup("//" + host + "toto");
            Console c = new ConsoleImpl();
            d.doIt(command, c);

        } catch (Exception ex) {
            ex.printStackTrace();
        }

    }

}

```

```
// ---- CONSOLE ----

public interface Console extends Remote {

    public void println (String s) throws RemoteException;
}

public class ConsoleImpl extends UnicastRemoteObjects implements Console {

    public ConsoleImpl() throws RemoteException {

    }

    public void doIt (String command) throws RemoteException {

        // Si on suit le sujet, on exécute : localExec(String command)
        // Là pour que ça fonctionne, on fait juste un print
        System.out.println("received "+s);
    }
}
```

TP RMI

L'objectif de ce TP est d'utiliser Java RMI pour implanter une petite application dans laquelle une applet permet de saisir les coordonnées d'une personne (nom, adresse email) et de les enregistrer dans un serveur. L'interface permet de sélectionner le serveur dans lequel la personne doit être enregistrée. On gèrera 2 serveurs. L'application permet également de rechercher l'email d'une personne qui a été enregistrée en indiquant le serveur dans lequel faire cette recherche.

Nom :	Xxx
Email :	Yyy
Serveur :	Serveur1
<input type="button" value="Enregistrer"/> <input type="button" value="Consulter"/>	

Si on ne trouve pas la personne dans le serveur indiqué, la recherche est alors propagée au second serveur.

L'interface d'un serveur est la suivante :

```
public interface Carnet extends Remote {
    public void Ajouter(SFiche sf) throws
    RemoteException;
    public RFiche Consulter(String n, boolean forward)
    throws RemoteException;
}

public interface SFiche extends Serializable {
    public String getNom ();
    public String getEmail ();
}

public interface RFiche extends Remote {
    public String getNom () throws RemoteException;
    public String getEmail () throws RemoteException;
}
```

L'enregistrement d'une personne utilise la sérialisation pour envoyer au serveur une copie d'un objet d'interface SFiche. La consultation retourne une référence à un objet RMI d'interface RFiche ; l'applet qui consulte peut alors récupérer l'adresse email par un appel à distance sur cette référence. La méthode de consultation inclut un paramètre booléen forward indiquant si la requête doit être propagée à l'autre serveur (pour éviter de boucler).

On vous donne :

- Carnet.java : l'interface d'un serveur
- SFiche.java : l'interface de l'objet sérialisé pour l'enregistrement auprès d'un serveur
- RFiche.java : l'interface de l'objet RMI retourné lors de la consultation auprès d'un serveur
- Saisie.java et page.html : l'applet implantant l'interface graphique de l'application

Vous devez :

- implanter CarnetImpl.java : la classe d'un serveur
- implanter SFicheImpl.java : la classe de l'objet sérialisé
- implanter RFicheImpl.java : la classe de l'objet RMI retourné par le serveur
- compléter Saisie.java : pour faire les appels aux serveurs

Carnet.java

```
import java.rmi.*;

public interface Carnet extends Remote {
    public void Ajouter(SFiche sf) throws RemoteException;
    public RFiche Consulter(String n, boolean forward) throws RemoteException;
}
```

SFiche.java

```
import java.io.*;

public interface SFiche extends Serializable {

    public String getNom ();
    public String getEmail ();
}
```

RFiche.java

```
import java.rmi.*;

public interface RFiche extends Remote {

    public String getNom () throws RemoteException;
    public String getEmail () throws RemoteException;
}
```

Saisie.java

```
import java.lang.*;
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import java.rmi.*;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

import javax.swing.*;

public class Saisie extends JApplet {
    //... ((du code java swing))
}

// La reaction au bouton Consulter
class CButtonAction implements ActionListener {
    public void actionPerformed(ActionEvent ae) {
        String n, c;
        n = nom.getText();
        c = carnets.getSelectedItem();
        message.setText("Consulter("+n+", "+c+"      ");

        try {

            Carnet carnet = (Carnet) Naming.lookup("///localhost:1234/carnet");
            String emailReturned = carnet.Consulter(n, false).getEmail();

            if (emailReturned != null) {

                email.setText(emailReturned);
            }
            else {
                System.out.println("no email found");
            }
        }
        catch (Exception e) {

            e.printStackTrace();
        }
    }

// La reaction au bouton Ajouter
class AButtonAction implements ActionListener {
    public void actionPerformed(ActionEvent ae) {
        String n, e, c;
        n = nom.getText();
        e = email.getText();
        c = carnets.getSelectedItem();
        message.setText("Ajouter("+n+", "+e+", "+c+"      ");

        try {

            Carnet carnet = (Carnet) Naming.lookup("///localhost:1234/carnet");
            carnet.Ajouter(new SFicheImpl(n, e));
        }
        catch (Exception ex) {

            ex.printStackTrace();
        }
    }
}

public static void main(String args[]) {
    // du java swing (je laisse au cas où):
    Saisie a = new Saisie();
    a.init();
    a.start();
    JFrame frame = new JFrame("Applet");
    frame.setSize(400,200);
    frame.getContentPane().add(a);
    frame.setVisible(true);
}
}
```

CarnetImpl.java

```

import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.server.UnicastRemoteObject;
import java.util.HashMap;
import java.rmi.Naming;

public class CarnetImpl extends UnicastRemoteObject implements Carnet {

    HashMap<String, RFiche> carnet = new HashMap<>();

    public CarnetImpl() throws RemoteException {}

    @Override
    public void Ajouter(SFiche sf) throws RemoteException {

        carnet.put(sf.getNom(), new RFicheImpl(sf.getNom(), sf.getEmail()));
    }

    @Override
    public RFiche Consulter(String n, boolean forward) throws RemoteException {

        return carnet.get(n);
    }

    public static void main(String[] args) {

        try {

            LocateRegistry.createRegistry(1234);
            Carnet c = new CarnetImpl();
            Naming.bind("//localhost:1234/carnet", c);
        }
        catch (Exception e) {

            e.printStackTrace();
        }
    }
}

```

RFicheImpl.java

```

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class RFicheImpl extends UnicastRemoteObject implements RFiche{

    private String nom;
    private String email;

    public RFicheImpl (String nom, String email) throws RemoteException {

        this.nom = nom;
        this.email = email;
    }

    @Override
    public String getNom() throws RemoteException {

        return this.nom;
    }

    @Override
    public String getEmail() throws RemoteException {

        return this.email;
    }
}

```

SFicheImpl.java

```

import java.rmi.RemoteException;

public class SFicheImpl implements SFiche {

    private String nom;
    private String email;

    public SFicheImpl (String nom, String email) {

        this.nom = nom;
        this.email = email;
    }

    @Override
    public String getNom() {

        return this.nom;
    }

    @Override
    public String getEmail() {

        return this.email;
    }
}

```