UNIVERSITE D'EVRY VAL D'ESSONNE

Examen de Conception et Programmation d'Applications Réparties

(Durée: 3 heures)

NOTE:
- Cet examen comporte 4 pages.
- Tout document papier est autorisé. Téléphones et ordinateurs interdits.
- Le barème est donné à titre indicatif.

EXERCICE 1 : Parallélisme

Soit les trois processus suivants :

- P1=while(true) {A;B;C;}
- P2=while(true) {D;E;F;}
- P3=while(true) {G;H;}

Contrairement aux processus vu en TD, les processus considérés ici sont itératifs. Nous voulons, comme en TD, réaliser le Workflow représenté par la figure 1.

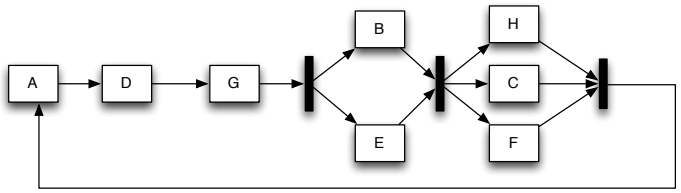

Figure 1: Le workflow des trois processus

Pour cela nous utilisons la variante de JobController suivante :

```
public class JobController {
    boolean done=false;

    public JobController(boolean b){done=b;}

    synchronized public void jobDone(){done=true; notify();}

    synchronized public void isJobDone(){
        if(!done)
            try { wait(); done=false;}
            catch (InterruptedException e) {e.printStackTrace();}
            }
}
```

suite

Il y a deux différences entre la classe *JobController* vue en TD et celle présentée ici :

- Le *JobController* présenté ici dispose d'un constructeur qui permet de fixer la valeur de l'attribut *done*.
- La méthode *isJobdone()* remet la valeur de *done* à *false* après le blocage *wait()*.

1. À votre avis à quoi servent les deux modifications citées plus haut. (2 pts)

2. Étant données trois instances de processus, respectivement de type P1, P2 et P3, de combien d'instances de *JobController* avez-vous besoin pour réaliser le Workflow de la figure 1 ? Expliquez. (3 pts)

3. Donnez le code des trois classes P1, P2 et P3 permettant de réaliser le Workflow. Pour les actions, par exemple pour `A`, utilisez `System.out.println("A")`. (3 pts)

4. Écrivez une méthode `main` permettant de lancer les instances des trois processus. (2 pts)

EXERCICE 2 : socket et RMI

Le but de cet exercice est de mettre en place un objet distant en n'utilisant que les *sockets*, sans passer par RMI. En d'autres termes, nous allons programmer une version simplifiée des coulisses de RMI. Ce qui veut dire que nous allons implémenter nous-mêmes les *stub*s et les *skeleton*s.

1. Rappelez brièvement les rôles du *stub* et du *skeleton* dans l'invocation d'une méthode à distance. (2 pts)

Soit la classe *ObjetDistantImpl* (figure 2) qui implémente les deux méthodes *M1* et *M2* de l'interface *ObjetDistant*. Les deux méthodes prennent respectivement en paramètre un objet de type *ObjetParam1* et *ObjetParam2* et renvoient respectivement *ObjetParam2* et *ObjetParam1*. La figure 2 contient la définition des quatre types.

2. Quelle est la différence entre le passage par valeur et le passage par référence ? Dans le cadre des invocations à distance, seules les valeurs peuvent transiter sur le réseau, comment simule-t-on le passage par référence ? (2 pts)

suite

```
public class ObjetParam1{ }
-------------------------------------------------------------------------
public class ObjetParam2{ }
-------------------------------------------------------------------------
public interface ObjetDistant {
        public ObjetParam2  M1 (ObjetParam1 arg) ;
        public ObjetParam1  M2 (ObjetParam2 arg) ;
}
-------------------------------------------------------------------------
public class ObjetDistantImpl implements
ObjetDistant{
        public ObjetParam2  M1 (ObjetParam1 arg)
        {System.out.println("M1 vient d'être invokée");
        return new ObjetParam2();}
        public ObjetParam1  M2 (ObjetParam2 arg)
        {System.out.println("M2 vient d'être invokée");
        return new ObjetParam1();}
}
```

Figure 2: Implémentation de ObjetDistantImpl, son interface ObjetDistant et les deux classes *ObjetParam1* et *ObjetParam2*

```
import java.io.*; import java.net.*;
public class Stub_ObjetDistant implements ObjetDistant {
        int port;
        String adresse;
        public Stub_ObjetDistant(String ad, int p)
        {port =p; adresse= ad;}
        public ObjetParam2 M1(ObjetParam1 arg) {
                /*  à compléter */}
        public ObjetParam1 M2(ObjetParam2 arg) {
                /* à compléter */ }
}
-----------------------------------------------------------
import java.io.*;
import java.net.*;
public class Skeleton_ObjetDistant extends Thread{
        int port;
        ObjetDistant od;
        public Skeleton_ObjetDistant(ObjetDistant o, int p)
        {od=o; port=p;}
        public void run()
        {/* à compléter */}
        public void traiter_Requete (Socket s)
        { /*à compléter */ }
}
```

Figure 3: Code à remplir du "stub" et du "skeleton" de l'objet distant

3. Dans le cas de notre classe *ObjetDistantImpl*, on suppose que les passages des paramètres pour les deux méthodes de notre objet distant se fait par valeur. Quelles modifications faut-il apporter aux classes *ObjetParam1* et *ObjetParam2* pour que cela soit possible ? (2 pts)

Développement du "*stub*" et du "*skeleton*"

suite

On désigne par *Stub_ObjetDistant* la classe *stub* de notre objet distant. Le *stub* est une classe qui a la même signature que l'objet distant (implémente la même interface, voir figure 3) et qui est située coté client. Au lieu du code des méthodes, les méthodes du *stub* contiennent le code nécessaire pour transformer les appels locaux à des requêtes vers le *skeleton* responsable de l'objet : à chaque invocation de l'une de ses méthodes, le *stub* ouvre une connexion avec le *skeleton* et envoi une sérialisation de l'appel. Chaque appel est sérialisée comme suit :

(a) L'envoi du nom de la méthode qu'on veut invoquer,

(b) L'envoi de l'ensemble des paramètres (dans l'ordre de la signature),

(c) L'attente de la réponse.

La réponse ainsi obtenue sera le retour de la méthode invoquée.

Côté serveur, chaque objet distant (ici instance de *ObjetDistantImpl*) est confié à un objet "*skeleton*", *Skeleton_ObjetDistant*, sur une adresse particulière et un port particulier. Le "*skeleton*" est un processus qui a pour charge de répondre aux demandes de connexion des clients. Pour chaque demande de connexion, la socket résultante est confiée à la méthode *traiter_Requete* (voir figure 3) qui écoute de manière symétrique ce que le "*stub*" envoi, ensuite elle invoque la méthode correspondante de l'instance de l'objet géré par le "*skeleton*" et renvoie la réponse à travers la *socket* au client (en vérité au stub).

4. Complétez le code des deux classes *Skeleton_ObjetDistant* et *Stub_ObjetDistant* de la figure 3 afin de simuler l'invocation de méthode à distance des méthodes M1 et *M2*. (4 pts)

Rappel:
Pour envoyer sur une socket *s* un objet "*Serializable*" *obj*, on connecte la sortie de la socket (*s.getOutputStream*()) avec celle d'un objet de type *ObjectOutputStream o* comme suit :
ObjectOutputStream o=new ObjectOutputStream(ss.getOutputStream());
oo.writeObject(obj);
pour lire :
ObjectInputStream oo=new ObjectInputStream(ss.getInputStream());
obj=o.readObject();
