Université Evry Val d'Essonne

Examen de Conception et Programmation d'Applications Réparties

(Durée: 3 heures)

NOTE:
- Tout document papier est autorisé. Téléphones et ordinateurs interdits.
- Le barème est donné à titre indicatif.

Concurrence : Réalisation de workflow

Trois partenaires P1, P2 et P3, chacun offrant un certain nombre d'activités, doivent se coordonner pour offrir le workflow de la figure 1. A côté de chaque activité vous trouverez le numéro du partenaire à qui appartient l'activité. L'objectif final de cet exercice, comme vu en cours et en TD, est de mettre en place un mécanisme

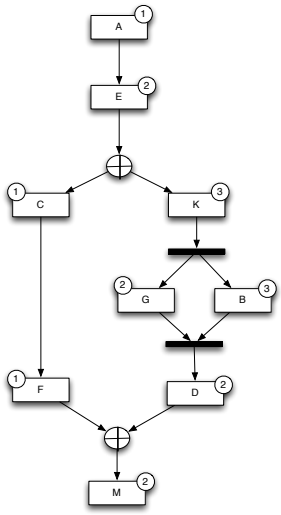

Figure 1: Workflow entre les trois partenaires

de coordination qui permet de déduire trois types de processus (Thread) qui représentent le comportement de chaque partenaire et qui garantissent que seulement les exécutions de figure 1 auront lieu. Durant le cours et le TD on s'est limité à deux opérateurs de workflow à savoir la succession, représenté par des flèches, et l'opérateur de parallélisme et de jointure représentés par une barre horizontale.

Comme vous pouvez le constater, en plus des flèches et de la barre, nous avons un nouvel opérateur, $\oplus$. Cet opérateur délimite n sous-workflow indépendants. La sémantique de cet opérateur exprime une course de déclenchement entre les n sous workflow. Avec l'opérateur $\oplus$, quand une activité d'un processus est activé elle ne sera pas forcément exécutée. Dans notre cas, l'opérateur $\oplus$ délimite deux sous-workflows le premier est une séquence de $C; F$ et le deuxième est également une séquence de l'activité K, un opérateur parallélisme-jointure (entre G et B) suivit de l'activité D. Une fois E terminée, les deux premières activités, à savoir K et C, sont déclenchées. La première qui aura accès à la notification de la fin de E va être exécutée par son processus alors que l'autre que le processus de l'autre activité va ignorer son execution mais il va quand même propager aux activités suivante sa terminaison. Si on suppose que le processus 1 reçoit la notification de la fin de E en

premier, alors la séquence $C; F$ va s'exécuter. Par contre, le processus 3 bien qu'il a reçu la notification de la fin de E, ne vas pas executer K mais il va quand marqué que K est terminé pour déclencher les activités suivante qui eux même ne vont pas s'exécuter et ainsi de suite. La fin de l'opérateur $\oplus$ coincide avec la fin de tout les sous workflow même si un seul a été exécuté. Dans notre cas, l'activité M, qui se trouve derrière l'opérateur $\oplus$, se déclenchera après la fin de D et la fin de F même si y'a que F qui a été exécutée.

1. Donnez les différentes exécutions possibles du workflow de figure 1. (1 pt)

Dans la figure 2, vous trouverez un rappel du code $JAVA$ de $JobController$ vu en cours et en TD. $Jobcontroller$ a été utilisé comme mécanisme de coordination de workflow entre plusieurs processus. On a également vu en cours et en TD, comment appliquer des règles qui nous permettent d'obtenir le code de chaque partenaire en utilisant le mécanisme de $JobController$.

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

    public synchronized void isJobDone(){
        if(!done)
        {
            try {wait();}

            catch (InterruptedException e)
            {e.printStackTrace();}

        }
    }

    public synchronized void jobDone(){
        done=true;
        this.notifyAll();
    }

}
```

Figure 2: code $JAVA$ de JobController

2. Est-ce que $JobController$, tel qu'il est donné dans la figure 2, ainsi que les règles vues en cours nous permettent de traiter le cas de l'opérateur $\oplus$. Argumentez votre réponse. (2 pts)

Pour mieux comprendre l'opérateur $\oplus$ il faut faire la distinction entre le déclenchement d'une activité et son execution. En effet dans l'exemple vu en TD si une activité déclenchée est forcément exécutée, ce qui explique que le $JobController$ a un seul attribue ($done$). Avec $\oplus$, quand une activité transmet à une autre activité l'information de sa terminaison, elle lui transmet également une autre information qui est le droit d'éxécution. Ce droit d'exécution se propage comme l'activation d'une activité à une autre selon les règles des opérateurs :

- Si une activité se trouve à la suite d'une activité (relié par un simple arc) alors son droit d'execution est celui transmis par la l'activité qui la précède.
- Si une activité se trouve à la suite d'un opérateur de jointure son droit d 'execution est la conjonction (et) des droits transmis par les activités qui précède la jointure.
- Si une activité qui se trouve à la suite de la fin d'un $\oplus$ son droit d'execution est la disjonction (ou) des droits transmis par les activités qui précède la fin du $\oplus$.

3. Donnez une nouvelle version de $JobController$ qui prendra en considération la particularité de l'opérateur $\oplus$. (3 pts)

4. Donnez les nouvelles règles qui permettent de déduire le comportement de chaque partenaire pour le nouvel opérateur $\oplus$. (3 pts)

5. Donnez le code de chaque processus des partenaires P1 et P3. (2 pts)

6. Ecrivez un *main* qui lance trois instances une pour chaque partenaire. (1 pt)

Objects Réparties et *Socket*

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 3) 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 3 contient la définition des quatre types.

```
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 3: Implémentation de ObjetDistantImpl, son interface ObjetDistant et les deux classes *ObjetParam1* et *ObjetParam2*

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)

```
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 4: 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*"

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 4) 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 4) 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 4 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();