Examen S1 – Développement des applications réparties (Corrigé)

Exercice 1 : QCM (10 pts)

Question1. L'avantage de l'utilisation des Threads par rapport aux processus est que **(2pts)** :

A. Le dysfonctionnement d'un thread peut perturber les autres threads
B. Le dysfonctionnement d'un thread n'a pas d'incidence sur les autres processus
C. Il est souvent plus efficace de coder une application nécessitant des traitements parallèles avec des threads

Question2. Les threads sont vus "extérieurement" comme des instances de la classe **(2pts)**:

A. java.lang.Thread
B. lang.java.Thread
C. java.run.Thread

Question3. Les outils de synchronisation dans la programmation des applications réparties sont **(2pts)**:

A. Mutex
B. Sémaphores
C. Moniteurs

Question4. Les avantages des architectures des applications distribuées par rapport aux architectures des applications distribuées sont **(2pts)**:

A. Tous les pairs fournissent des ressources (bande passante, stockage, puissance de calcul,...). On obtient ainsi des architectures qui supportent beaucoup mieux la montée en charge ("scalability") que les architectures centralisées.
B. La distribution augmente la robustesse du réseau dans le cas d'une panne par la réplication des données sur plusieurs pairs.
C. Elles ne permettent pas un contrôle avancé des échanges d'information entre les pairs.

Question5. Pourquoi utiliser des threads dans la programmation réseau? **(2pts)**

A. Nous ne voulons pas qu'un seul client se connecte au serveur à un moment donné, mais plusieurs clients simultanément.
B. Nous voulons utiliser le protocole TCP et UDP.
C. Nous voulons appliquer les sockets dans les deux cotés (clients et serveurs).

Exercice 2 :

On commence par écrire le code pour une voiture, pour en simuler plusieurs on lancera simplement plusieurs threads.

Examen S1 – Développement des applications réparties (Corrigé)

Chaque voiture sera donc simulée par un thread. Une voiture dispose d'un nom et pointe sur le *Parking* qui ici est partagé. Le comportement de chaque voiture est cyclique et peut se décrire comme suit:

Répéter:

- Se déplacer à l'extérieur un *certain temps*.
- Demander et obtenir l'accès au parking.
- Rentrer dans le parking se garer et y rester un *certain temps*
- Sortir du parking.

Exemple pour la classe voiture¶

```
1  public class Voiture implements Runnable {
2          String nom;
3          Parking park;
4    public Voiture(String name, Parking park){
5          this.nom=name;
6          this.park=park;
7
8          public void run(){
9          System.out.format("[%s]: Je débute !  \n", this.nom);
10         try {
11           while(true){
12                 Thread.sleep((long)  (50000* Math.random()));
13                 System.out.format("[%s]: Je demande à rentrer  \n", this.nom);
14                 this.rentrer();
15                 System.out.format("[%s]: Je viens d'entrer \n", this.nom);
16                 Thread.sleep((long)  (50000* Math.random()));
17                 System.out.format("[%s]: Je demande à sortir  \n", this.nom);
18                 this.park.leave(this);
19         }}
```

Le Parking est simpliste, on le crée en passant sa taille au constructeur. Il propose deux méthodes atomiques (*synchronized*), *leave()* et *accept()*.

- *accept()* retourne vrai si il y a de la place et reserve la place (incrémente le nombre de places occupées), et faux sinon.
- *leave* laisse sortir la voiture et décrémente le nombre de places occupées.

Examen S1 – Développement des applications réparties (Corrigé)

J ai ajouté une structure qui maintient l'ensemble des voitures situées dans le parking mais ce n'etait pas demandé. Il serait facile de comptabiliser de manière analogue le temps passé par chaque voiture dans le parking.

Exemple pour la classe Parking¶

```
 1 import java.util.*;
 2 public class Parking {
 3    int PlacesOccupees;
 4    int Capacite ;
 5    public HashSet<Voiture> infoVoitures = new HashSet<Voiture>();
 6    Parking(int size){ this.Capacite = size;}
 7    int places(){ return (this.Capacite - this.PlacesOccupees); }
 8
 9  synchronized boolean  accept(Voiture myVoit) {
10             if  (this.places() >0 )
11               {
12                        this.PlacesOccupees ++ ;
13                        infoVoitures.add(myVoit);
14                        System.out.format("[Parking] :%s acceptée, il reste %d places \n",
15 myVoit.nom, this.places());
16                        System.out.format("Voiture Garees\n");
17                        System.out.println(infoVoitures);
18                        return (true) ;
19               }
20             else {
21               System.out.format("Parking : %s refusée, il reste  %d places \n",
22 myVoit.nom,this.places());
23               return(false) ;
24             }
25    }
26 synchronized void leave(Voiture myVoit) {
27             PlacesOccupees --;
28             infoVoitures.remove(myVoit);
               System.out.format("Parking :[%s] est sortie, reste  %d places\n", myVoit.nom,
   places());
     }}
```

La méthode rentrer qui bloque la voiture tant qu'il n y a pas de places innonde le parking de demandes, entre chaque demande la voiture attend un peu. On verra plus tard comment éviter cela.

Méthode rentrer¶

Examen S1 – Développement des applications réparties (Corrigé)

```
public class Voiture implements Runnable {
        public void rentrer() throws InterruptedException{
                while (!(this.park.accept(this)))
                {
                  Thread.sleep((long)  (1000* Math.random()));
                  System.out.format("[%s]  : Je redemande à rentrer  \n", this.nom);
                }
        }
```

Le programme principal se contente de créer le parking ainsi que les thread associés aux voitures tout en démarramt ces threads.

Main du simulateur de parking¶

```
public class Voiture implements Runnable {
  public static void main(String[] args) {
        int TailleParking=8;
        int nbVoitures=15;
        Parking leParking = new Parking(TailleParking);
        Thread MesVoitures[] = new Thread[nbVoitures];
        for (int i =0; i< nbVoitures; i++){
          MesVoitures[i]= new Thread(new Voiture(String.format("Voit %d ", i) ,
leParking));
          MesVoitures[i].start();}}}
```