Master d'informatique | **module PC2R** | **Université Pierre et Marie Curie**
Documents autorisés | *Durée 2 heures* | **Année 2013-2014**

Examen réparti du 07 janvier 2014

Exercice 1 : Robot suivant un trait (Esterel)

On souhaite reprendre l'exercice concernant le robot roulant sur un trait continu. Pour le besoin de cet exercice et pour simplifier, on garde certains signaux en les renomant et on ajoute d'autres :

```
output G_Avance, D_Avance;
output G_Stop, D_Stop;
input  TOUCH;

% Et les nouveaux signaux

input  ARRET;
INPUT  G, D;
output SOS;
```

Un petit rappel :

. Lorsque l'on émet `G_Avance` (resp. `D_Avance`), le moteur gauche (resp. droit) est actionné et ne s'arrêtera qu'à la réception de `G_Stop` (resp `D_Stop`).

On suppose que le trait est assez long et surtout assez large pour faire fonctionner correctement 2 détecteurs, un placé à gauche et l'autre à droite. Le détecteur gauche (resp. droit) émet à chaque `tick` le signal `G` (resp. `D`) lorsqu'il est sur le trait et s'arrête de l'émettre lorsqu'il est en dehors. L'absence du signal `G` (resp. `D`) indique que le robot est sorti par la gauche (resp. droite).

Le robot doit se comporter de la manière suivante :

. Le robot est à l'arrêt. Il attend le signal `TOUCH` pour démarrer.
. Un opérateur place correctement le robot sur le trait et actionne (émet) le signal `TOUCH`.
. Au démarage, le robot avance en tentant de rester sur le trait.
. On suppose que les 2 moteurs ont toujours la même vitesse.
. S'il sort par la gauche (resp. droite), il arrête son moteur droit (resp. gauche) pendant un délai de `MAX` (une constante) `tick`, ce qui lui fait tourner à droite (resp. gauche). Après ce délai, il réactive son moteur droit (resp. gauche) pour retrouver son chemin.
. Lorsqu'il est perdu, c'est-à-dire qu'il ne reçoit aucun des signaux `G` et `D`. Il s'arrête en émettant en continu le signal `SOS` à l'attention de l'opérateur.
. L'opérateur viendra alors le remettre sur le trait et doit actionner (émettre) le signal `TOUCH`.
. A tout moment, l'opérateur peut arrêter le robot en actionnant (émettant) le signal `ARRET`. Le robot s'arrête alors et termine son programme (fin de son module).

Question 1 – implantation du module rouler

Ecrire le module `rouler` pour le robot en respectant son comportement décrit ci-dessus. Vous pouvez ajouter les signaux, modules ou autres que vous jugez utiles pour l'écriture de votre programme Esterel.

Exercice 2 : Emissions synchrones (C, OCaml ou Java)

Dans les réseaux asynchrones, l'envoi d'un message est indépendant de sa réception et l'ordre des messages n'est pas préservé par le réseau. Après avoir effectué un envoi, le programme émetteur continue son exécution sans savoir quand (si) le message a

atteint le programme récepteur. Si le programme émetteur effectue successivement deux envois, le deuxième peut être reçu par le programme récepteur avant le premier.

Question 2 – préliminaires – ordre des messages

On considère deux clients, reliés à un même serveur, qui envoient des messages selon les protocoles ci-dessous. La première valeur reçue par le serveur (peu importe son origine) est mise dans x_1, la deuxième dans x_2, ...

Client 1 :	**Client 2 :**	**Serveur :**
`Envoyer` 1;	`Envoyer` 3;	`Recevoir dans` x_1;
`Envoyer` 2;		`Recevoir dans` x_2;
		`Recevoir dans` x_3;

1. On suppose que le réseau est asynchrone et les trois programmes sont lancés simultanément. A la fin du protocole, quelles sont les valeurs possibles du triplet (x_1, x_2, x_3) ?
2. La plupart des *couches message* du monde réel s'assurent que deux messages ayant le même origine et la même destination arrivent dans le même ordre que celui de leur émission. Dans ce cas quelles sont les valeurs possibles du triplet (x_1, x_2, x_3) ?

Question 3 – envois synchrones séquentiels

Il est parfois nécessaire de limiter le caractère asynchrone d'un réseau en implémentant des envois bloquants (comme le `Event.sync` (`Event.send c`) de OCaml). Cela permet d'obtenir une notion de causalité.

On se propose d'écrire un protocole client-serveur (en C, Java ou OCaml [1]) qui réalise l'envoi synchrone dans le sens Client→Serveur. L'idée est de créer du côté serveur et du côté client des outils qui assurent l'envoi synchrone de manière transparente : le client reste bloqué dans son exécution jusqu'à la bonne réception de son message par le serveur.

On suppose dans un premier temps que le client est totalement séquentiel (pas de threads, les émissions sont réalisées les unes après les autres).

Dans ce cas, une manière simple d'obtenir des émissions synchrones et d'envoyer une confirmation du serveur vers le client : à chaque fois qu'il reçoit un message, le serveur envoie un message "ack" au client. Ce dernier attend de recevoir cette confirmation avant de continuer son exécution.

Réaliser une implantation comportant :

- Un mécanisme client-serveur classique, à travers une socket dont le port est donné en argument au serveur et au client.
- Un outil côté client pour envoyer un message de manière synchrone. L'appel à la fonction/méthode `syncSend` envoie un message au serveur et termine seulement après avoir reçu une confirmation.
- Un outil côté serveur. L'appel à la fonction/méthode `syncReceive` bloque le serveur dans l'attente d'un message. A la réception, elle envoie une confirmation puis retourne comme résutat le contenu du message.
- un code simple pour le client et le serveur qui permet de tester les fonctions `syncSend` et `syncReceive` (par exemple, l'envoi des différents mots d'une phrase dans l'ordre).

Question 4 – test

Proposer un test permettant de vérifier que l'envoi se fait bien de manière synchrone.

Question 5 – envois synchrones concurrents - serveur

On se place maintenant dans un cadre plus général. On suppose que le client lance plusieurs threads "ouvriers" qui effectuent tous des `syncSend`.

Notre précédente méthode n'assure plus la synchronie : une confirmation correspondant à un message envoyé par le thread $\texttt{th}_1$ peut être reçue par le thread $\texttt{th}_2$ et confondue avec une confirmation pour un message envoyé par lui-même (et qui n'a pas encore été reçu par le serveur).

On peut résoudre ce problème en personnalisant les confirmations (à chaque envoi est associé un jeton qui lui est propre, et qui sera recopié dans la confirmation, permettant au client de faire le lien entre les deux).

Une méthode est l'utilisation d'un compteur global sur le client (protégé par un mécanisme de verrou) : à chaque envoi synchrone, le message est annoté avec un jeton entier correspondant à la valeur du compteur, qui est ensuite incrémenté.

Le serveur parse les messages reçus en séparant le jeton du véritable message. Puis envoie une confirmation au client en l'annotant avec le même jeton.

Décrire une manière d'annoter les messages et de les parser. Ecrire la fonction `syncReceive` du serveur qui réalise la réception d'un message et l'envoi d'une confirmation annotée et qui retourne comme résultat le message sans le jeton.

1. dans ce cas, bien entendu, on n'utilisera pas la bibliothèque `Event`

Question 6 – envois synchrones concurrents - handler

Il faut construire du côté client une structure qui gère la réception des différentes confirmations et signale les threads émetteurs associés. L'idée est de créer un thread sur le client qui s'occupe de recevoir toutes les confirmations et les place dans une structure de données (tableau, table, . . .).

Ecrire une fonction exécutée par un thread "handler" lancé par le client, qui écoute sur la socket, recoit des confirmations, parse le jeton (l'entier associé) des confirmations et la place dans une structure de données. Faire attention à ce que l'accès à la structure de données soit protégé.

Question 7 – envois synchrones concurrents - envoi

Il faut maintenant écrire la fonction `syncSend`, qui sera présente dans chaque thread "ouvrier" lancé par le client. Cette fonction devra annoter le message à partir d'un compteur global (du coté client), envoyer le message, puis attendre que la confirmation correspondante (avec le bon jeton) arrive dans la structure de données.

Ecrire la fonction `syncSend`, en faisant attention à ne pas créer d'attente active.

Question 8 – envois synchrones concurrents - fin

Pour finir, écrire le code des threads "ouvriers" en entier. On fera en sorte qu'ils envoient successivement, dans des délais aléatoires, les différents mots d'une même phrase.

Ecrire enfin le code de client, qui lance le thread "handler" et plusieurs threads "ouvriers".

Exercice 3 : des « futures » répartis (Java RMI)

Dans cet exercice on cherche à définir des `Future` répartis utilisant le mécanisme RMI d'appels distants. L'idée est de pouvoir effectuer des appels distants sans attendre la fin du calcul pour passer localement à d'autres actions sachant que l'accès au résultat de ce calcul est bloquant si le calcul n'est pas fini. Pour cela on définit les interfaces suivantes :

- `Callable` pour la définition d'un calcul retournant une valeur d'un certain type,
- `CallableSer` pour pouvoir sérialiser un tel calcul et l'envoyer comme argument d'une méthode distanteenvoyer un tel calcul
- `ExecutorRMI` pour accepter de recevoir un calcul à exécuter et retourner un `FutureRMI`
- et `FutureRMI` pour le contrôle d'un "future" distant.

```
1  public interface Callable<V> { V call() throws Exception;}
2  public interface CallableSer<V extends Serializable> extends Callable<V>{}
3
4  public interface ExecutorRMI extends Remote {
5      <V extends Serializable> FutureRMI<V> submit(CallableSer<V> job) throws RemoteException,InterruptedException;
6   }
7
8   public interface FutureRMI<V extends Serializable> extends Remote {
9       public V get() throws RemoteException,InterruptedException,ExecutionException;
10    public boolean isDone() throws RemoteException;
11 }
```

L'envoi d'un calcul de type `CallableSer<V>` via la méthode `submit` (sur un objet distant exposé d'interface `ExecutorRMI`) retournera un `FutureRMI` qui pourra être consulté par le client pour vérifier que le calcul est fini (`isDone`) sur le serveur et récupérer le calcul le cas échéant avec la méthode `get`). Si le calcul n'est pas fini, l'appel de `get` est alors bloquant. Le `FutureRMI` est l'interface d'un objet distant qui n'a pas besoin d'être exposé, il ne sera connu que par l'appelant.

Question 9 – classe FutureD

On cherche à écrire la classe `FutureD<V>` qui implante l'interface `FutureRMI`. Cette classe aura un constructeur prenant un `Future` comme argument. On rappelle l'interface `Future` :

```
1  public interface Future<V> {
2     public V get();
3     public V get(long timeout, TimeUnit unit);
4     public boolean isDone();
5     public boolean cancel(boolean mayInterruptIfRunning);
6     public boolean isCancelled();
7  }
```

Compléter le squelette suivant.

```
1  public class FutureD<V extends Serializable> extends UnicastRemoteObject implements FutureRMI<V> {  // A COMPLETER
2     public V get() throws RemoteException,InterruptedException,ExecutionException { /* A COMPLETER */}
3     public boolean isDone() throws RemoteException { /* A COMPLETER */}
4  }
```

Question 10 – classe ExecutorD

Ecrire une implantation pour ce mécanisme général en implantant une classe `ExecutorD` implantant l'interface `ExecutorRMI` et qui implante un pool de threads dont le nombre maximal peut être paramétré à la création (`Executor.nexFixedThreadPoolint capacity)`). L'appel du `submit` créera une instance de `FutureD` qui sera retournée comme résultat pour le client appelant, lui permettant ainsi d'avoir accès au calcul en cours. Compléter le squelette suivant.

```
1  public class ExecutorD extends UnicastRemoteObject implements ExecutorRMI {
2    private ExecutorService pool;
3    ExecutorD(int capacity) throws RemoteException { pool = Executors.newFixedThreadPool(capacity);}
4
5    public <V extends Serializable> FutureRMI<V> submit(CallableSer<V> job) throws RemoteException,InterruptedException {
6       // A COMPLETER
7    }
8  }
```

Question 11 – classe MyJob

On cherche maintenant à définir une classe `MyJob` pour des calculs qui retournent des entiers en implantant `CallableSer` et `Serializable` de la manière suivante :

```
1  public class MyJob implements CallableSer<Integer>, Serializable {  /* ... */ }
```

Compléter cette classe pour construire des "jobs" permettant de calculer la somme des n premiers nombres entiers (le n est fourni au constructeur) que l'on pourra soumettre au serveur.

Question 12 – création d'un client

On supposera existant un serveur sur la machine `exam.jussieu.fr` qui expose une instance de `ExecutorD` sous le nom `ex0` de la manière suivante :

```
1     try {
2         ExecutorD ex0 = new ExecutorD(10);
3         Naming.rebind("//localhost/ex0",ex0);
4         System.out.println("Objet distribue 'ex0' est enregistre");
5     }
6     catch (Exception e) { e.printStackTrace(); }
```

On demande d'écrire une classe `Client` en plus sieurs étapes :

1. récupération d'une référence distante sur `ex0`,
2. envoi de p jobs sur `ex0` en stockant dans une `ArrayList<FutureRMI<Integer>>` les références sur les "futures" distants créés.
3. récupération des résultats dans une `ArrayList<Integer>>`
4. calcul de la somme de ceux-ci.

Question 13 – interruption de calculs

Que faut-il ajouter à ces "futures" distants pour pouvoir interrompre un calcul distant ? On pourra s'inspirer des méthodes `boolean cancel(boolean mayInterruptIfRunning)` et `boolean isCancelled()` des "futures" classiques.

Question 14 – mécanisme de rappel

On cherche maintenant à utiliser le mécanisme de rappel du RMI pour ne pas créer d'objets supplémentaires sur le serveur. La soumission d'un job d'un client à un serveur prendra un paramètre suppélmentaire, de type `RappelRMI`, pour que le serveur puisse appeler la méthode distante `put` quand le calcul sera terminé.

```
1  public interface RappelRMI<V extends Serializable> extends Remote {
2     void put(V r) throws RemoteException;
3  }
4  public interface MRExecutorRMI extends ExecutorRMI {
5      <V extends Serializable> void submit(RappelRMI<V> r, CallableSer<V> job) throws RemoteException,↩
           InterruptedException;
6   }
```

1. Ecrire une classe `FutureRappel` qui implante `RappelRMI` et qui fournit les méthodes `get` et `isDone` des "futures". Préciser bien la synchronisation utilisée avant d'écrire le code demandé.
2. Ecrire une sous-classe de `ExecutorD`, appelée `MRExecutorD`, qui implante l'interface `MRExecutorRMI`, c'est-à-dire qui surcharge la méthode `submit` en précisant comment est lancé le calcul demandé et comment le résultat est transmis au client.
3. En supposant qu'il existe une instance de `MRExecutorD` exposée sou sle nom `mrex1` sur la machine `exam.jussieu.fr`, écrire un client qui effectue le même calcul qu'à la question 4.