Master d'informatique | **module PC2R (4I507)** | **Université Pierre et Marie Curie**
Documents autorisés | *Durée 2 heures* | **Année 2016-2017**

Examen réparti du 18 mai 2017

Exercice 1 : Système d'enchères par canaux synchrones

Dans cet exercice, on s'intéresse à la rédaction d'un système d'enchères utilisant les canaux synchrones d'OCaml [1]. Le système permet à un nombre indéterminé d'acheteurs de proposer des offres pour l'achat d'un objet. Il est composé ainsi :

- les *acheteurs* envoient régulièrement des offres sur le canal du commissaire. Une offre est composée d'un entier (la valeur de l'offre), d'une chaine de caractères (le nom de l'acheteur) et d'un canal (celui de l'acheteur). Apres l'envoi d'une offre, l'acheteur attends sur son canal personnel (celui envoyé dans l'offre) une confirmation du commissaire, puis il peut envoyer une nouvelle offre. Le comportement exact de l'acheteur reste abstrait (son code n'est pas demandé).
- le *commissaire* mémorise la meilleure offre et écoute à la fois sur son propre canal et sur le canal *temps*. S'il recoit une offre d'un acheteur sur son propre canal, il met à jour (si nécessaire) la meilleure offre, envoie une confirmation à l'acheteur et se relance. S'il recoit la valeur () sur le canal temps, il envoie sur le canal du *maître* l'offre mémorisée et s'arrête.
- à l'initialisation du système, le *maître* attends un temps déterminé (10 secondes, par exemple). Quand ce temps est écoulé il envoie sur le canal temps la valeur () puis attends sur son propre canal la réception de la meilleure offre, qu'il écrit sur la sortie standard.

Question 1 – Système

Decrire le système par un schéma, représentant des acheteurs, le commissaire et le maître, les canaux utilisés et les valeurs transitant par ces canaux.
Donner le type de tous les canaux synchrones du système.

Question 2 – Maître

Donner le code du maître.

Question 3 – Commissaire

Donner le code du commissaire. On fera attention à faire en sorte que le commissaire puisse recevoir des messages sur les deux canaux (son canal propre et le canal temps).

Question 4 – Fin d'enchère

Que se passe t-il à la fin d'une enchère pour les acheteurs ? Décrire une amélioration.

Question 5 – Equité

Définir la notion d'équité pour le système considéré. Proposer une amélioration du système pour que cette propriété soit garantie.

1. Une syntaxe fonctionnelle approximative est acceptée. Un autre langage manipulant des envois/réceptions synchrones (Promela, Go, ...) est accepté.

Exercice 2 : Traction d'un train en programmation synchrone

Nous voulons simuler un fonctionnement (très simplifié) de la partie "traction" d'un train en *esterel*. Nous proposons de composer le système ici en 3 parties :

- le *conducteur* de train,
- le bloc de contrôle *moteur*,
- le dispositif de *veille automatique*.

Le *conducteur* et ses actions :

- C'est l'utilisateur qui joue le rôle du conducteur.
- A tout moment, il peut :
  - . changer de vitesse *v* en émettant un signal valué *C_Vitesse(v)* ,
  - . freiner normalement en émettant sur un ou plusieurs *tick* le signal *FreinageNormal* ,
  - . ou freiner en urgence en émettant sur un ou plusieurs *tick* le signal *FreinageUrgent* .
- Il doit régulièrement signaler sa présence en émettant un signal *Presence* . Entre deux signaux *Presence*, il ne faut pas attendre plus de 10 *tick*. (voir le dispositif de *veille automatique* ci-dessous).
- Dans un train, ces signaux pourraient être provoqués par des leviers, boutons, volants ou autres.
- Pour éviter que le conducteur simule sa présence en bloquant le dispositif concerné, il ne peut pas émettre en continu le signal *Presence* (Voir le dispositif de *veille automatique* ci-dessous).

Le bloc de contrôle *moteur* :

- Il gère la vitesse.
- Il émet à chaque *tick* sa vitesse courante *v* sous forme d'un signal valué *Vitesse(v)* (affiché dans le terminal).
- Lorsqu'il reçoit un signal *C_Vitesse(v)* émis par le conducteur, il augmente ou diminue sa vitesse d'une unité par *tick* pour positionner sa vitesse à la valeur demandée.
- A chaque signal *FreinageNormal* reçu, il diminue sa vitesse d'une unité par *tick* .
- A chaque signal *FreinageUrgent* reçu, il diminue sa vitesse de 5 unités par *tick* .
- Il maintient sa vitesse courante si aucun de ces signaux ci-dessus n'est reçu.

Le dispositif de *veille automatique* :

- Ce dispositif permet de vérifier en continu la présence du contrôleur.
- Il doit recevoir régulièrement du contrôleur le signal *Presence* .
- Entre deux signaux *Presence*, il ne doit pas s'écouler plus de 10 *tick*.
- Au bout de 10 *tick* sans signal *Presence*, il émet le signal *Alerte* pendant 5 *tick*. Et au bout de ces 5 *tick* toujours sans signal *Presence*, il émet en continu le signal *FreinageUrgent*.
- Par contre, si le signal *Presence* est émis en continu sur 5 *tick*, il émettra le signal *Alerte* pendant 5 *tick*. Et pendant ces 5 *tick* d'alerte, si le signal *Presence* est toujours présent, il émettra en continu le signal *FreinageUrgent*.
  Le dispositif du signalement de présence pourrait être bloqué intentionnellement ou accidentellement.
- Un changement de présence ou absence (selon la situation) du signal *Presence* arrête l'envoi du signal *FreinageUrgent*.

Question 1
Mettre en place la déclaration des signaux énnonés ci-dessus. (Vous êtes libre d'ajouter vos signaux ou autres outils que vous jugez nécessaires pour le programme)

Question 2
Ecrire le module *moteur* pour répondre aux comportements énoncés dans le bloc de contrôle *moteur*.

Question 3
Ecrire le module *veille* pour répondre aux comportements énoncés dans le dispositif de *veille automatique*.

Question 4
Ecrire le module principal *train* pour faire fonctionner le système.

Exercice 3 : Migration de processus par agents RMI

Récemment, la recherche en systèmes répartis a vu l'émergence du modèle de *programmation par agents mobiles.* Dans ce modèle, un *agent* est un processus possédant un contexte d'exécution, incluant du code et des données, pouvant se déplacer de machine en machine (appelées serveurs) afin de réaliser la tâche qui lui est assignée. Les agents mobiles visent principalement des applications réparties sur des réseaux à grande distance, car ils permettent de déplacer l'exécution vers les serveurs et de diminuer ainsi le coût d'accès à ces serveurs. Nous allons ici modéliser un tel paradigme en s'appuyant sur le modèle RMI de Java.

Nous allons ici implémenter une version simplifiée de ce modèle en se basant sur le principe de l'existence de *plusieurs serveurs* de calcul. Chaque serveur contient la possibilité d'effectuer une partie d'un calcul réparti de grande envergure. Ainsi, au lieu d'interroger chaque serveur indépendemment, le client va créer un *agent.* Cet agent va passer de site en site, et sur chacun de ces sites effectuer une partie du calcul correspondant et mettre à jour son ensemble de solutions. Une fois tous les sites consultés, l'agent donne au client le résultat complet du calcul. Pour ce faire, le client va lancer sa demande depuis un serveur RMI principal qui sera appelé le *site initiateur*. Ce client va initialiser un tableau de sites à parcourir, puis exécuter sur le premier site de la liste (appelé *host*) la méthode `migrate`, à laquelle il a passé l'agent en argument. Ceci fait migrer l'agent d'un hôte à l'autre jusqu'à arriver de nouveau sur la hôte initiateur. Pour commencer nous considèrerons que l'agent ne sert qu'à contenir des résultats (donc uniquement des données sans contexte d'exécution ni code)

Question 1. D'après vos connaissances en applications réparties, discuter les différentes manières d'implémenter un tel agent et quels sont leurs avantages et inconvénients. Justifier le choix d'un modèle pour le reste de votre implémentation.

Question 2. Quelles seront les structures et spécificités à prendre en compte leur de l'implémentation de l'agent ?

Question 3. Quel est le rôle de la méthode migrate ? Cette méthode sera-t'elle implémentée de la même manière à travers tout le réseau ?

Nous allons maintenant passer à l'implémentation à proprement parler du système. Nous commencons par remarquer que les mécanismes peuvent être implémenter de manière à être complètement indépendants du calcul à traiter. Nous devons ainsi réaliser l'implémentation de trois grands mécanismes.

1. Les services (*host* et *agent*) étant partagés avec les clients mais dont l'implémentation est propre au serveur
2. Les serveurs, avec notamment la distinction entre serveur initiateur et sous-serveur de calcul
3. Le client et les éventuels threads lui correspondant dans les serveurs

Question 4. Donner les types, interfaces et implémentations des services (*host* et *agent*).

Question 5. Donner les types interfaces et implémentations des serveurs

Question 6. Donner le code résumé du client ainsi qu'un exemple d'execution

Nous voulons maintenant complexifier ce mécanisme pour permettre une réelle *migration* de processus. Dans ce cadre, nous considérons maintenant que tout les serveurs effectuent le *même calcul* mais que la migration nous permettra de déplacer l'exécution d'un calcul d'un site à l'autre en fonction de son taux d'utilisation actuel. Ainsi un calcul en cours pourra être déplacé d'un serveur à l'autre mais également être surveillé de manière asynchrone.

Question 7. Sans vous inquiéter de la manière dont les serveurs (host) sont choisis pour la migration, décrivez les modifications à opérer avec le système précédent pour obtenir un tel modèle. Quel mécanisme Java utiliser pour réussir à migrer un calcul en cours d'exécution ?

Question 8. Effectuer l'implémentation d'un tel système

Question bonus Décrire une manière de partager des codes binaires en temps réel pendant l'exécution (sans utiliser de partage de NFS), permettant ainsi à l'agent de modifier dynamiquement son comportement