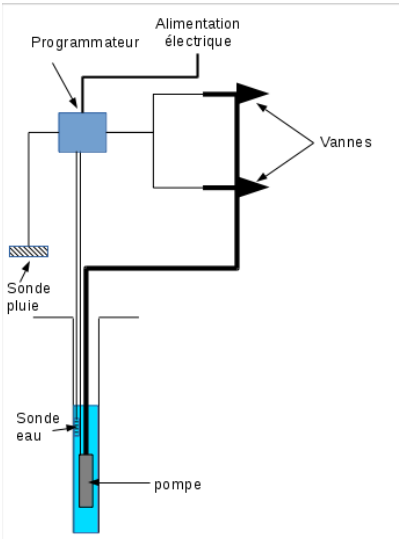
Master d'informatique | **module PC2R (4I507)** | **Université Pierre et Marie Curie**
Documents autorisés | *Durée 2 heures* | **Année 2014-2015**

Examen du 18 juin 2015 (2ème session)

Exercice 1 : Arrosage synchrone en Esterel

Nous voulons simuler en Esterel un programmateur d'arrosage. Ce dernier fait partie d'un système composé de : (voir dessin ci-dessous)

- une pompe immergée dans un puits
- une sonde permettant de détecter le manque d'eau dans le puits
- une sonde permettant de détecter la pluie
- un programmateur
- 2 électrovannes.



Le programmateur fonctionne en cycle (voir la constante `NB_TICKS_PAR_CYCLE` et le signal `CYCLE`). Dans chaque cycle, il doit

- attendre le signal CYCLE pour démarrer ou redémarrer.
- lorsque, à tout instant, le signal `MANQUE_EAU` apparaît, il doit impérativement arrêter la pompe en émettant le signal `POMPE_ARRET` et attendre le cycle suivant pour recommencer (ce qui permet d'éviter de faire fonctionner dangereusement la pompe à vide),
- lorsque, à tout instant, le signal `PLUIE` apparaît, il doit arrêter d'arroser en émettant le signal `POMPE_ARRET` pour arrêter la pompe et attendre le cycle suivant pour recommencer,
- lorsque le signal PLUIE apparaît dans plusieurs ticks très rapprochés, la pompe s'arrête de fonctionner pour un bon moment et la reprise risque d'être difficile.
  Pour éviter qu'elle ne s'arrête trop longtemps, on décide pour chaque cycle de la faire tourner pendant un tick à la fin du cycle (au `(NB_TICKS_PAR_CYCLE - 2)`ème tick), sans ouvrir les vannes.
  Cette opération est bien sûr inutile si la pompe a déjà fonctionné dans ce cycle.
- commencer à émettre `POMPE_MARCHE` et `VANNE_1_OUVERTE` ;
  puis au bout de 5 ticks, émettre `VANNE_1_FERMEE` et `VANNE_2_OUVERTE` ;
  et enfin au bout de 5 ticks, émettre `POMPE_ARRET` et `VANNE_2_FERMEE` ;

Question 1

Compléter le module `principale` qui simule le fonctionnement du système.

```
1 module principale :
2 constant NB_TICKS_PAR_CYCLE = 25 : integer;
3 inputoutput DEBUT_CYCLE, PLUIE, MANQUE_EAU,
4             POMPE_ARRET, POMPE_MARCHE,
5             VANNE_1_OUVERTE, VANNE_1_FERMEE, VANNE_2_OUVERTE, VANNE_2_FERMEE;
6
7 ...
8
9 end module
```

Question 2
Ecrire le module `emetteur_cycle` qui émet continuellement le signal `DEBUT_CYCLE` tous les `NB_TICKS_PAR_CYCLE` ticks.

Question 3
Ecrire le module `programmateur` dont les comportements sont indiqués ci-dessus.

Exercice 2 : Topologie maillée en Client/Serveur et Threads

On se place dans un réseau de N participants identiques, identifiés par un numéro unique entre 1 et N, déployés chacun à une adresse différente, interagissant entre eux. Chaque participant se comporte à la fois comme un client (il peut contacter les autres participants) et comme un serveur (il écoute sur le port 2015).

Chaque participant connaît **partiellement** l'ensemble des adresses des autres participants du système. Il dispose d'une structure (de votre choix) $\mathrm{AdressesConnues}$ qui associe à un numéro une adresse IP. Par exemple, chez le participant 3, la structure $\mathrm{AdressesConnues}$ peut associer $17.85.1.128$ au numéro 1 et $14.41.5.47$ au numéro 4, mais ne rien associer au numéro 2. Dans ce cas, le participant 3 connait l'adresse de 1 et 4 (et de 3, puisque c'est sa propre adresse) mais pas de 2.

Question 4
Donner une interface et une implémentation de $\mathrm{AdresseConnues}$. Un participant doit être capable de décider si l'adresse d'un autre participant est connue ou non, de la récupérer le cas échéant et d'ajouter une nouvelle association (numéro, adresse) à la structure.

On suppose qu'à l'initialisation, la structure $\mathrm{AdresseConnues}$ de chaque participant est non-vide (chaque participant connait l'adresse d'au moins un autre participant).

Les messages qui transitent dans le système contiennent un destinataire (un numéro), un corps (une chaîne de caractère) et un auteur (un numéro) qui a envoyé le message initialement. Le protocole associé à un envoi de message est le suivant :

1. on ouvre une connexion sur le port 2015 d'un participant n,
2. on envoie la commande $\mathrm{DESTINATAIRE}$ suivi d'un entier d,
3. on envoie la commande CORPS suivi d'une chaîne de caractère,
4. on envoie la commande AUTEUR suivi d'un entier a,
5. le participant n peut ensuite exécuter trois comportements différents :
   (a) si le participant n est le destinataire du message ($n = d$), il répond avec la commande OK, ferme la connexion et affiche le corps du message sur sa sortie standard,
   (b) si le destinataire du message d est connu du participant n (d est dans $\mathrm{AdressesConnues}$ de n), le participant n renvoie la commande CONNU, ferme la connexion et envoie le message à d (il ouvre une connexion avec d et applique ce protocole),
   (c) si le destinataire du message d est inconnu du participant n (d n'est pas dans $\mathrm{AdressesConnues}$ de n), le participant n renvoie la commande $\mathrm{INCONNU}$, ferme la connexion et envoie le message à un participant p dont il connaît l'adresse **au hasard** (il ouvre une connexion avec un participant aléatoire p de $\mathrm{AdressesConnues}$ et applique ce protocole),

Question 5
Donner un exemple d'un système (les participants et leurs $\mathrm{AdressesConnues}$) et d'un message tel que le message ne pourra jamais atteindre son destinataire.
Donner un critère sur le système pour qu'un message puisse toujours atteindre sa destination.

Les participants sont composés de deux parties, une partie serveur qui accepte des connexions sur le port 2015 et agit selon le protocole décrit plus haut, côté serveur, et une partie client qui s'occupe de transférer les messages en contactant d'autres participants et agit selon le même protocole, côté client.

Question 6
Ecrire la partie serveur d'un participant. Elle doit être capable d'accepter simultanément plusieurs connexions.

Dans les cas $5.b$ et $5.c$, elle fait appel a la fonction/méthode $\mathrm{Transfere(m, k)}$ (qui fait l'objet de la question suivante) qui opère le transfert du message m (encodage au choix) au participant k (symbolisé par son numéro, et dont l'adresse est connue).

Question 7

Ecrire la partie client d'un participant, c'est-à-dire la fonction/méthode $\mathrm{Transfere(m, k)}$.

Attention, un participant doit pouvoir envoyer deux messages simultanément si nécessaire. Par exemple, si la partie serveur du participant n reçoit deux messages pour $p1$ et $p2$ et que les adresses de $p1$ et $p2$ sont toutes les deux connues de n, la partie client de n doit effectuer le transfert des messages vers $p1$ et $p2$ simultanément (en ouvrant deux connexions) et non séquentiellement.

Question 8

Modifier le protocole et le code des parties clients et serveur pour que dans le cas $5.b$, le participant n renvoie l'adresse du destinataire. Le participant qui a contacté n modifie la structure $\mathrm{AdressesConnues}$ pour ajouter l'adresse du destinataire.

Attention, la structure $\mathrm{AdressesConnues}$ peut être utilisée de manière concurrente.

Exercice 3 : Redondance de calculs

On cherche dans cet exercice à proposer un mécanisme simple de redondance de calcul. Un calcul aura n instances, potentiellement différentes, qui se dérouleront en parallèle et qui passeront par des points de synchronisation permettant de vérifier que les calculs ne divergent pas. Pour cela il sera nécessaire de pouvoir tester les résultats intermédiaires des calculs, et de signaler dès que possible une divergence entre deux calculs. S'il y a divergence, tous les calculs s'arrêteront et une exception sera déclenchée pour indiquer cette situation anormale dans le déroulement du programme.

Chaque instance du calcul correspondra à un thread et un état. Une nouvelle primitive `sync_step()` indiquera un point de synchronisation et permettra de passer d'une instance à l'autre. Quand chaque instance aura effectué son étape de calcul, les différents états associés aux instances seront comparés. Ces états sont connus au lancement de ces instances. S'ils sont tous égaux, toutes les instances repartent pour une étape de calcul jusqu'à la prochaine synchronisation. S'il y a des différences, le calcul peut être considéré comme divergent et une exception doit être lancée contenant l'ensemble des états des instances.

Pour le début de l'exercice, on simplifie le problème en ne s'intéressant qu'aux calculs à 2 instances.

Question 9

Proposer les principes pour un tel mécanisme de redondance de calcul utilisant seulement les deux commandes suivantes : `sync_execute2(f1,s1,f2,s2,e)` et `sync_step()`. L'état $s1$ est passé à la fonction de calcul $f1$ (resp. $s2$ pour $f2$), l'exception e est déclenchée en embarquant les états $s1$ et $s2$ en cas de divergence. Si les codes de $f1$ ou $f2$ ne possèdent pas de commande `sync_step` il n'y aura pas de vérification possible. Il en est de même si un calcul termine avant les autres au niveau d'une étape complète. Préciser les limitations de votre solution.

Question 10

Que fait le programme suivant (écrit en pseudo-code) en fonction de votre proposition précédente ?

```
function void f1(int* s) {
  s[1] = 1 ;
  s[2] = 1 ;
  while (s[1] <= s[0]) {
    s[2]  = s[2] * s[1] ;
    s[1] = s[1] + 1 ;
    sync_step();
  }
}
```

```
function void f2(int* s) {
  if (s[1] <= s[0]) {
    s[2] = s[2] * s[1] ;
    s[1] = s[1] + 1 ;
    sync_step() ;
    f2(s);
  }
}
```

```
main () {
  int[3] s1 = {5, 1, 1} ;
  int[3] s2 = {5,1, 1} ;
  sync_execute2(f1, &s1, f2, &↩
       s2, e) ;
}
```

Question 11

Donner une implantation de ces commandes sous la forme que vous préférez (macros, fonctions, méthodes, ...) dans un langage de votre choix dans la liste suivante : OCaml, Java ou C. Ne pas hésiter à introduire les bibliothèques dont vous avez besoin en précisant tout de même les signatures des fonctions que vous utiliserez.

Question 12

Traduire dans le langage choisi le programme en pseudo-code précédent, en particulier implanter la gestion de l'exception.

Question 13

Proposer puis implanter les modifications pour pouvoir passer à n instances de calcul.