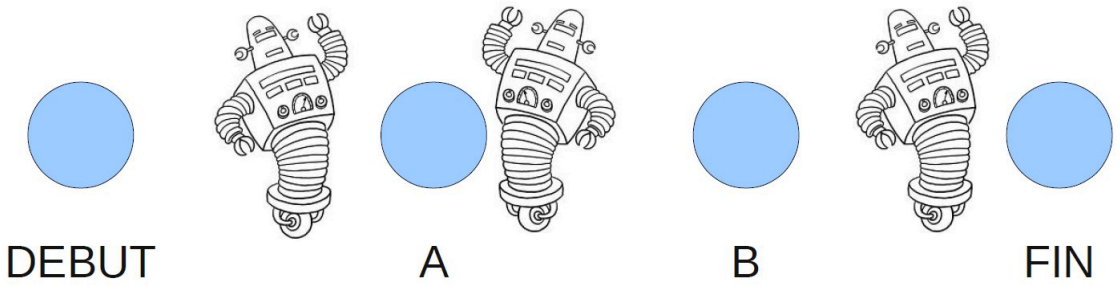
Master d'informatique | **module PC2R** | **Université Pierre et Marie Curie**
Documents autorisés | *Durée 2 heures* | **Année 2011-2012**

Examen réparti du 10 janvier 2012

Exercice 1 : Chaîne de montage en Esterel



Une chaîne de montage est composée de 4 paniers (DEBUT, A, B et FIN) et de 3 robots (voir image ci-dessus) et fonctionne de la façon suivante :

- Au départ, tous les paniers sont vides et les robots en attente.
- Chaque panier est soit vide, soit plein. Son contenu est un entier.
- On propose de stocker cet entier dans une variable `n` locale. La valeur -1 (resp. positive) indiquera que le panier est vide (resp. plein).
- Lorsqu'un panier `x` (avec x = DEBUT, A ou B) est vide, il n'émet rien et attend le signal valué `REMPLIR_x` (valeur toujours positive) pour lui premettre de remplir (affecter) sa variable locale `n` avec la valeur du signal.
- Lorsqu'un panier `x` est plein, il émet en continu son signal valué `PANIER_PLEIN_x` associé à la valeur de `n` jusqu'à l'apparition du signal `VIDER_x`. Ce dernier lui permet de se vider en affectant sa variable locale `n` à -1.
- Le premier robot travaille avec deux paniers `DEBUT` (son panier d'entrée) et `A` (son panier de sortie), le deuxième avec deux paniers `A` (entrée) et `B` (sortie) et le dernier avec paniers `B` (entrée) et `FIN` (sortie).
- Un robot cherche à prendre un entier dans son panier d'entrée si ce dernier est plein (présence du signal `PANIER_PLEIN_x`). Il prendra alors l'entier en émettant le signal `VIDER_x`, le garde pendant un nombre aléatoire de `tick` et le dépose ensuite dans son panier de sortie si ce dernier est vide, sinon il attend que ce dernier soit vide. Si son panier d'entrée est vide (absence du signal `PANIER_PLEIN_x`), le robot attendra le remplissage du panier.
- Le dernier panier `PANIER_FIN` fonctionne comme les autres sauf qu'il se vide automatiquement au prochain `tick` sans attendre l'apparition d'un signal.
- Pour tester la chaîne, l'utilisateur émet manuellement lorsque le panier DEBUT est vide (absence du signal `PANIER_PLEIN_DEBUT`) plusieurs signaux valués (un à la fois) `REMPLIR_DEBUT(n)`. Voir l'exemple d'exécution ci-dessous.

La fonction écrite en C `int rand (int x)` est supposée connue, elle renvoie un nombre aléatoire compris entre 0 et x.

Question 1 Compléter le module `PANIER` permettant de répondre au comportement énoncé ci-dessus. On évite de faire des attentes actives.

Question 2 Compléter le module `ROBOT` permettant de répondre au comportement énoncé ci-dessus. On évite de faire des attentes actives.

Question 3 Compléter le module `PANIER_FIN` permettant de répondre au comportement énoncé ci-dessus.

Question 4 Compléter le module principal `Examen_Esterel` pour faire fonctionner en parallèle les différents acteurs de la chaîne.

Modules

```
module Panier :

inputoutput REMPLIR : integer;
inputoutput PANIER_PLEIN : integer;
inputoutput VIDER;

     % ????????????????????????????????????? %
end module

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

module Panier_FIN :

inputoutput  REMPLIR_FIN : integer,
             FIN : integer;
inputoutput PANIER_PLEIN_SORTIE : integer;

     % ????????????????????????????????????? %
end module

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

module Robot :

inputoutput  PANIER_PLEIN_ENTREE : integer;
inputoutput VIDER_ENTREE;
inputoutput  PANIER_PLEIN_SORTIE : integer;
inputoutput REMPLIR_SORTIE : integer;

     % ????????????????????????????????????? %
end module

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

module Examen_Esterel :

inputoutput REMPLIR_DEBUT : integer,
      REMPLIR_A : integer,
      REMPLIR_B : integer,
      REMPLIR_FIN : integer;

inputoutput PANIER_PLEIN_DEBUT : integer,
       PANIER_PLEIN_A : integer,
       PANIER_PLEIN_B : integer,
       PANIER_PLEIN_SORTIE : integer;

inputoutput VIDER_DEBUT, VIDER_A, VIDER_B;

inputoutput FIN : integer;

    % ????????????????????????????????????? %
end module
```

Exécution

```
%{ Et un exemple d'execution


$ Examen_Esterel
Examen_Esterel> REMPLIR_DEBUT(1);
--- Output: REMPLIR_DEBUT(1)
Examen_Esterel> ;
--- Output: PANIER_PLEIN_DEBUT(1)
Examen_Esterel> ;
--- Output: VIDER_DEBUT
Examen_Esterel> REMPLIR_DEBUT(2);;;;;;;;;;;;;;;;;;
--- Output: REMPLIR_DEBUT(2)
--- Output: PANIER_PLEIN_DEBUT(2)
--- Output: REMPLIR_A(1) PANIER_PLEIN_DEBUT(2)
--- Output: PANIER_PLEIN_DEBUT(2) PANIER_PLEIN_A(1)
--- Output: VIDER_DEBUT VIDER_A
--- Output:
--- Output: REMPLIR_A(2)
--- Output: REMPLIR_B(1) PANIER_PLEIN_A(2)
--- Output: PANIER_PLEIN_A(2) PANIER_PLEIN_B(1)
--- Output: VIDER_A VIDER_B
--- Output:
--- Output:
--- Output: REMPLIR_B(2) REMPLIR_FIN(1) \
PANIER_PLEIN_SORTIE(1)
--- Output: PANIER_PLEIN_B(2) FIN(1)
--- Output: VIDER_B
--- Output:
--- Output: REMPLIR_FIN(2) PANIER_PLEIN_SORTIE(2)
--- Output: FIN(2)
Examen_Esterel>

}%
```

Exercice 2 : Serveur pour le jeu Pacman

La mission dans le jeu bien connu Pacman est d'aider le petit Pacman à manger tous les points qui se trouvent dans le labyrinthe pour avoir un bon score en évitant qu'il soit manger par ses ennemis qui risquent de croiser son chemin comme dans la figure 1.

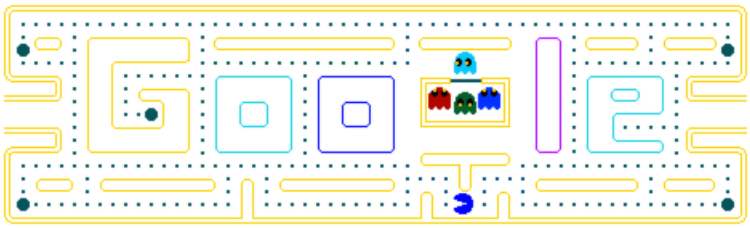


FIG. 1 – Configuration d'un tableau de partie

Pour cet exercice on cherchera à implanter une version réseau simplifiée de ce jeu tout d'abord avec une version mono-joueurs puis une version multi-joueurs où plusieurs petits Pacman seront des programmes indépendants qui interagiront en communiquant sur le même serveur. Dans les deux versions chaque ennemi est implanté par un thread différent géré par le serveur. On ne s'intéressera qu'à la partie serveur et au protocole de communications. Le langage d'implantation est à choisir entre C, Java et OCaml.

Le serveur attend une connexion d'un joueur et lance un thread particulier pour communiquer avec ce joueur. Ce thread gère le tableau de la partie, les différents éléments qui en font partie : le Pacman que manipule le client, les différents ennemis qui interviennent dans le jeu, les cases spéciales, le score du joueur et la fin de la partie. Le tableau d'une partie est un labyrhinte rectangulaire possédant des murs et des points à ramasser pour les cases sans mur. La partie s'arrête quand tous les points sont ramassés ou quand le Pacman se fait manger par un ennemi. On utilisera que deux types d'ennemis : le fixe qui occupe une case et mange ce qui y passe, et l'aléatoire qui se déplace de manière aléatoire et peut manger le Pacman s'il le rencontre.

A la connexion du joueur, le serveur envoie la description du tableau de la partie. Le joueur indique le début de partie et peut ensuite changer la direction de déplacement du Pacman dans les 4 directions possibles. Le personnage avance selon la direction donnée si cela est possible mais peut être bloqué par un obstacle. Quand un ennemi apparait, le serveur indique sa nature et sa position au client joueur. L'ennemi fixe ne bouge pas, par contre l'ennemi aléatoire oui ; les deux indiquent à chaque instant du jeu leur position au joueur. Il peut avoir plusieurs ennemis (plus que deux) en même temps. Les ennemis peuvent se croiser. A chaque instant (tour de jeu), le serveur regarde si le client change de direction, le fait bouger si cela est possible et compte s'il y a des points pris, fait bouger les différents ennemis, et vérifie que la partie n'est pas finie, à ce moment là il envoie le score au client.

1. Donner la représentation de ces différentes informations (tableau de la partie, score, joueur, ennemis) et préciser le protocole textuel que vous utiliserez.
2. En réutilisant une structure de serveur comme celle présentée dans le polycopié, écrire la gestion d'une partie où il n'y aurait pas d'ennemi, mais seulement un joueur qui se déplace dans le labyrinthe. Le serveur doit tenir compte de l'interaction.
3. Ajouter ensuite l'apparition d'un ennemi tous les n instants pour une durée de vie de p instants (tours de jeu). La détection de la fin de partie doit alors prendre en compte le fait qu'un ennemi puisse manger le joueur. Indiquez aussi comment vous détectez le fait que le Pacman puisse être mangé. Chaque ennemi est implanté par un thread. Le thread s'arrête quand l'ennemi disparait.

On passe maintenant à une version multi-joueurs. La partie démarre lorsque les n joueurs sont connectés. Chacun sera initialement à une position différente. Seul un joueur peut être sur une case, et donc gagner le point qui y est. Les joueurs ne peuvent donc pas se croiser. La partie est finie quand tous les points sont gagnés ou quand tous les joueurs sont mangés.

4. Indiquer les modifications à apporter au protocle pour tenir compte de l'information reçue par les joueurs et de celle envoyées à tous les joueurs par le serveur.
5. Indiquer les changements à apporter à votre serveur pour tenir compte des modifications apportées au protocole de communication et pour gérer le déplacement de chaque joueur. Si un joueur est mangé, il reçoit toujours les informations de déplacement des autres joueurs. A la fin de la partie les scores de tous les joueurs sont envoyés.

Exercice 3 : Messagerie en RMI

Le but de cet exercice est d'implanter un logiciel de messagerie instantanée (un chat) en RMI.

On souhaite utiliser objet centralisé qui se chargera de communiquer les différents messages aux logiciels le référençant. On souhaite aussi que l'objet partagé soit nommé `CHATRMI` et qu'il mette à disposition des utilisateurs les méthodes :

- `void add_string(String text);` qui rajoute une chaîne à un tableau local de chaînes
- `String [] get_strings()` qui retourne une copie du tableau local de chaînes.

1. Décrire le schéma général de l'application en identifiant quelles données résident coté serveur et quelles données coté client. On nommera en outre les différent éléments logiciels (serveur etc) intervenant dans la transaction RMI.
2. Ecrire une interface pour un composant RMI mettant à disposition les fonctions précédentes.
3. Ecrire la définition d'une classe correspondante.
4. Ecrire un client utilisant un référence distante sur un tel objet et implantant le chat comme suit :
   - Le client se connecte à l'objet distant
   - le client initialise $compt$, un entier local, à 0
   - le client lance un thread qui, toutes les secondes :
     - récupère les chaînes présentes dans le chat
     - affiche (sur la sortie std) les chaînes dont l'indice dans tableau retourné par `get_strings` est compris entre $compt$ et la taille du tableau. Le client positionne son entier $compt$ à la taille du tableau de chaînes qu'il vient de récupérer.
   - Le client lance un thread qui attend la rentrée d'une chaîne au clavier et appelle la méthode `add_string` pour la rajouter sur la table du serveur.
5. Identifier un problème de concurence lors de l'appel à `add_string` par plusieur personnes discutant sur le chat et proposer une solution *simple*.
6. On souhaite maintenant que le rafraîchissement coté client se fasse de manière assynchrone : c'est-à-dire que le client se voit notifier par le serveur de l'arrivée d'une nouvelle chaîne. Pour ce faire nous allons définir une classe `rappel` coté client qui dérivera de `UnicastRemoteObject` et dont une référence sera envoyée au serveur par le biais d'une méthode `void connect(rappel r)` qu'on ajoutera à l'interface du serveur. La classe `rappel` sera dotée d'une méthode publique `remotesignal()` permettant une invocation distante de la part du serveur pour signaler au client l'arrivée d'un nouveau message.

   Implanter ce mécanisme coté client et coté serveur (on pourra se contenter de mentionner sans les recopier les parties de code communes avec les questions précédentes). Préciser tout d'abord les interfaces puis les classes qui les implantent.