Examen - 22 mars 2015

Exercice 1 -

1. Dans une vague produite par l'algorithme d'Écho rappelé dans la figure 1, il n'existe qu'un seul processus possédant l'instruction qui puisse décider : le processus Q, soit le processus initiateur de l'algorithme.

2. Etant donné que nous ne possédons pas une liste exhaustive des différents voisins des n processus, il n'est pas possible de donner avec certitude le nombre de messages échangés lors d'une vague produite par l'algorithme. Toutefois, sachant que chaque processus envoie un seul et unique message à chacun de ses voisins, le nombre de messages échangés est fini. Ainsi, le nombre de messages échangé sera de $\sum_{p=1}^{n} \square(Voisins_p)$.

3. En estimant que la phase préliminaire de l'algorithme pourrait être une cartographie du réseau - tout à fait faisable, sachant que le graphe est connexe, il existe forcément un chemin entre chaque couple (u,v) - on sera en mesure d'extraire une liste de couples [(Q, P_i), (P_i, P_j) …] représentant le chemin le plus court possible pour contacter tous les processus. Le code de l'initiateur Q effectuera un **send(List \ { (Q, P_i), msg)** - List privé du couple (Q, P_i) - au processus P_i - processus auquel il sera couplé dans la liste List. Les processus effectueront ensuite des **receive(List \ { (P_i, P_j), msg)** jusqu'à atteindre le processus P_{n-1} où la liste List sera vide.

Exercice 2 -

On sait que chaque processus connaît ses propres voisins : le moyen le plus efficace de faire une cartographie du réseau est donc de créer une liste vide que Q initialisera et enverra à sa liste de voisins. Chaque voisin va ensuite ajouter une paire constituée de son père et de lui-même et envoyer la liste mise à jour à ses propres voisins. Il se mettra ensuite en attente de recevoir les listes mises à jour de ses voisins, et mettra sa propre liste à jour à chaque réception de liste. Une fois que tous les voisins ont répondu, le processus en cours renverra la liste à son père. De fait, le processus Q finira par obtenir une liste complète de toutes les paires disponibles dans le réseau de communication G.

Exercice 3 -

Données : Les processus sont organisés en anneau bidirectionnel. La taille de l'anneau est connue de tous les processus. Les processus ne connaissent pas leur identité. Sachant qu'il s'agit d'un réseau anonyme, tous les processus ont le même code.

Il s'agit donc d'un anneau bidirectionnel anonyme. Hors, il n'existe aucun **algorithme déterministe d'élection** sur un anneau anonyme quelle que soit la taille des processus, et qu'elle soit connue ou non.

Hypothèse H_i: Les processus sont tous dans le même état au début de la ronde *r*.

L'hypothèse à la première ronde est vraie car les processus sont **anonymes**.
Supposons que H_i soit vraie : dans ce cas, tous les processus vont effectuer les mêmes actions (le code étant identique d'un processus à l'autre) : ils vont donc envoyer les mêmes messages (demander à récupérer les variables en l'occurrence) à leur prédécesseur et successeur. Par conséquent, ils recevront tous les mêmes messages et passeront au même état : H_{i+1} est vraie.
Par récurrence, à chaque ronde *r*, les processus seront dans le même état. Supposons alors qu'un algorithme d'élection existe : si un des processus est élu, alors tous le seront également. Le principe d'un algorithme d'élection étant qu'il n'existe qu'**un seul** élu, alors cet algorithme n'existe pas.

Exercice 4 -

1) Par définition, il est impossible de faire un algorithme de consensus si la communication n'est pas fiable. Ainsi, dans ce cas, il est **impossible** de faire un algorithme de consensus tolérant une panne par arrêt.

2) On sait qu'on peut réaliser un algorithme de consensus à condition que le nombre de pannes tolérées soit inférieur au nombre de processus. Or, ici, $n/2 < n$, donc il est **possible** faire un algorithme de consensus tolérant n/2 pannes

3) En suivant le même raisonnement que pour la question précédente, n-1 étant inférieur a n, alors il est **possible** de faire un algorithme de consensus tolérant n-1 pannes.

4) Si on veut faire un algorithme de consensus qui tolère n pannes byzantines, il faut un minimum de 3n processus. Ainsi, un algorithme ayant n processus pourra tolérer jusqu'à n/3 pannes. Or, étant donné que $n/2 > n/3$, il sera **impossible** de faire un algorithme de consensus tolérant n/2 pannes.

Exercice 5 -

1) Le consensus est impossible à réaliser dans un système asynchrone où un seul processus peut subir une panne par arrêt.

2) Le consensus est impossible à réaliser dans un système asynchrone dès qu'au moins un processus peut tomber en panne crash

Examen - 24 mars 2016

Déterministe asynchrone = caca
Exercice 1 -
Propriété de vivacité : garantit qu'une propriété sera vraie dans un système à partir d'une certaine étape d'exécution.
Propriété de sûreté : ne pas sortir d'un certain ensemble d'état.
Propriété d'équité : garantit que tous les threads concernés par ordonnancement (ou synchronisation) sont traités de manière identique

⇒ NON

Propriété d'absence de famine : tous les processus demandant accès à une section critique y accède ⇒ NON

Exercice 2 -

1) **Terminaison** : Au bout d'un moment, tous les processus se finissent.
   Décision : Pour toute exécution, il y a un évènement qui décide.
   Dépendance : Quand l'événement décide se produit, un événement se produit sur tous les autres processus.

2) **Algorithme de vague sur un anneau :** On a l'initiateur qui envoie un message à son successeur qui va le transmettre à son successeur et ainsi de suite...
   Algorithme de vague sur un arbre : les feuilles (initiateurs) envoient un message, les pères des feuilles reçoivent un message de leurs fils (voisins), sauf qu'un (celui auquel ils enverront le message) : quand il aura reçu un message de tous ses voisins, le processus décidera.

3) Non ?

Exercice 3 -

Pour effectuer une table de routage du graphe des processus, le processus initiateur peut envoyer à ses voisins une liste préalablement remplie des couples (p_0, $voisin_q$) pour tout processus q voisin de p_0, puis se mettre en attente de réponse de ses voisins. Les processus non-initiateurs, à réception de cette liste, vont y ajouter leurs propres couples (p_i, $voisin_q$) pour tout processus q voisin de p_i et l'envoyer à leurs propres voisins (sont exceptés les couples (p_i, $voisin_q$) déjà présents dans la liste et le processus père (celui ayant envoyé la liste)), puis se mettre en attente d'une réponse. A chaque réponse d'un voisin, le processus va fusionner la liste reçue avec sa propre liste ; une fois toutes les réponses obtenues, le processus enverra la liste obtenue à son père (soit le processus qui l'a activé). Le processus p_0, une fois toutes ses réponses obtenues, aura une liste contenant tous les couples (p_i, p_j) présents dans le graphe de communication : on pourra en obtenir une table de routage.

Exercice 4 -

1) On prend une liste de taille n rempli d'entiers uniques entre 1 et n, le processus initiateur prend une valeur de la liste, la retire et se l'assigne comme nom, puis l'envoie à son successeur. Une fois que la liste est vide, alors on aura fait un tour complet de tous les processus et chacun aura un nouveau nom.

2) On aura envoyé n messages.

Exercice 5 -

1) Il est impossible de faire une élection déterministe sur un anneau anonyme quelle que soit la taille des processus, et qu'elle soit connue ou non.

Hypothèse H_i: Les processus sont tous dans le même état au début de la ronde *r*. L'hypothèse à la première ronde est vraie car les processus sont **anonymes**.
Supposons que H_i soit vraie : dans ce cas, tous les processus vont effectuer les mêmes actions (le code étant identique d'un processus à l'autre) : ils vont donc envoyer les mêmes messages (demander à récupérer les variables en l'occurrence) à leur prédécesseur et successeur. Par conséquent, ils recevront tous les mêmes messages et passeront au même état : H_{i+1} est vraie.
Par récurrence, à chaque ronde *r*, les processus seront dans le même état. Supposons alors qu'un algorithme d'élection existe : si un des processus est élu, alors tous le seront également. Le principe d'un algorithme d'élection étant qu'il n'existe qu'**un seul** élu, alors cet algorithme n'existe pas.

2) Il est possible de faire une élection probabiliste (retrouver dans le cours)

3) Le mécanisme permet qu'un seul processus à la fois prenne la valeur de ses voisins.
   a) Oui c'est possible avec 3.
   b) C'est pas possible avec 6 : (on peut modéliser ça avec deux ensembles de 3 reliés) donc il se peut que deux processus prennent la même valeur en même temps.

Exercice 6 -

1) Par définition, il est impossible de faire un algorithme de consensus si la communication n'est pas fiable. Ainsi, dans ce cas, il est **impossible** de faire un algorithme de consensus tolérant une panne par arrêt.

2) On sait qu'on peut réaliser un algorithme de consensus à condition que le nombre de pannes tolérées soit inférieur au nombre de processus. Or, ici, $n/2 < n$, donc il est **possible** faire un algorithme de consensus tolérant n/2 pannes

3) En suivant le même raisonnement que pour la question précédente, n-1 étant inférieur a n, alors il est **possible** de faire un algorithme de consensus tolérant n-1 pannes.

4) Si on veut faire un algorithme de consensus qui tolère n pannes byzantines, il faut un minimum de 3n processus. Ainsi, un algorithme ayant n processus pourra tolérer jusqu'à n/3 pannes. Or, étant donné que $n/2 > n/3$, il sera **impossible** de faire un algorithme de consensus tolérant n/2 pannes.

Exercice 7 -

1) Il est **impossible** de faire un algorithme déterministe de consensus tolérant une panne en asynchrone (**FLP 85**)

2) Un algorithme déterministe de consensus est **impossible** à réaliser dans un système asynchrone dès qu'au moins un processus peut tomber.

3) Un système asynchrone est f-tolérant si f < n/2

4) Un système asynchrone est f-tolérant si f < n/2

Séance révision

Examen 2009/2010 :
Exercice 1 -
Données :
Communication **point à point**, **synchrone**, **sans perte de messages** (fiable).
Panne crash ⇒ Quand un processus tombe en panne, **certains processus vont recevoir son message et d'autres non**.

A chaque ronde r :
- chaque processus p envoie Val_p vers tous les autres processus (dans n'importe quel ordre !)
- chaque processus reçoit les messages de la ronde r
- chaque processus change d'état suivant les messages reçus et détermine le message à envoyer lors de la ronde r + 1.

s^r_p ⇒ message envoyé par p à la ronde r.
R^r_p ⇒ messages reçus par p lors de la ronde r.
Viv(r) ⇒ les processus vivants lors de la ronde r.
Crash(r) ⇒ les processus qui ne sont pas vivants lors de la ronde r.
Val_p ⇒ liste contenant les val de tous les processus qui n'ont pas crashé.
Correct ⇒ ensemble des processus vivants pour toutes les rondes pour une exécution donnée.

Chaque processus p a une valeur initiale v_p.
Chaque processus a une variable D_p qui ne peut être écrite qu'une seule fois.

Un processus p décide v s'il écrit v dans D_p.

1) Si on a 4 processus où les valeurs initiales sont : $v_1 = 1$, $v_2 = 2$, $v_3 = 3$, $v_4 = 4$.

durant la ronde 1 :

p_1 envoie $s^1_1 = 1$ à tous les processus
p_2 envoie $s^1_2 = 2$ à tous les processus
p_3 n'envoie $s^2_3 = 3$ à p_4 et tombe en panne (***Propriété du crash***)
p_4 envoie $s^1_4 = 4$ à tous les processus
$\Rightarrow val^1_1 = \{1, 2, 4\}$, $val^1_2 = \{1, 2, 4\}$, $val^1_3 = \{ 3 \}$, $val^1_4 = \{1, 3, 4, 2\}$

durant la ronde 2 :

p_1 envoie $s^2_1 = \{1, 2, 4\}$ à tous les processus
p_2 envoie $s^2_2 = \{1, 2, 4\}$ à tous les processus
p_4 envoie $s^2_4 = \{1, 3, 4, 2\}$ à p_1 et tombe en panne (***Propriété du crash***)
$\Rightarrow val^2_1 = \{1, 2, 3, 4\}$, $val^2_2 = \{1, 2, 4\}$, $val^2_3 = \{ 3 \}$, $val^2_4 = \{1, 3, 4, 2\}$

Donc, Correct = $\{p_1, p_2\}$ et $val_1 \neq val_2$.

2) Soit Viv(r) = Viv(r+1), c'est à dire que les processus vivants à la ronde r sont les même à la ronde r+1. On peut donc en déduire qu'aucun processus n'est mort durant la ronde r + 1, et que tous les processus ont pu envoyer des messages qui ont tous pu joindre les processus destinataires restés vivants. Ainsi, pour tout (p, q) $\in$ Viv(r+1), on a bien $val^{r+1}_p = val^{r+1}_q$

3) L'algorithme ne permet pas de supprimer des valeurs de Val_p : en effet, dans le cas où le processus qui envoie et le processus qui reçoit ne souffrent pas de crash, on recevra un message qu'on ajoutera à Val_p. Dans le cas où le processus p ou le processus émetteur q souffrent d'une panne et que le message n'est pas reçu, alors Val_p ne sera pas modifié. De fait, si $r \leq r'$, alors $Val^r_p \subseteq Val^{r'}_p$.

4) Si Viv(r) = Viv(r+1), alors aucun processus n'est mort lors de la ronde r+1 ; tous les messages échangés auront trouvés leur destinataire et les val subiront une complétion : les valeurs absentes de val_p mais présente dans un val_q seront ajoutées et inversement. On aura donc bien pour tout $r' > r$, pour tout (p, q) $\in$ Viv(r), $Val^{r'}_p = Val^{r'}_q$.

5) A la ronde 1 :

$Val_p = v_p$ le message envoyé par le processus p

R^1_p est l'ensemble des messages reçus par le processus p. Sachant que les messages reçus par le processus p sont les Val_q de tous les processus q différents de p, alors

$\Rightarrow R^1_p = \{ Val_1, \ldots Val_q\} \Rightarrow R^1_p = \{ v_1, \ldots v_q\}$ pour tout $q \neq p$

A la ronde r+1 :

$Val^r_p = Val^{r-1}_p \cup R^r_p$

R^{r+1}_p est l'ensemble des messages reçus par le processus p. Sachant que les messages reçus par p sont les Val_q de tous les processus q différents de p, alors

$\Rightarrow R^{r+1}_p = \{ Val_1, \dots Val_q \}$ pour tout $q \neq p$

Les seuls messages envoyés et qui peuvent être ajoutés à l'ensemble de valeur de Val_p sont les valeurs v_p de chaque processus $p \in \mathbf{\Pi}$. De fait, pour toute ronde r, on a bien $U_{p \in \mathbf{\Pi}} Val^r_p \subseteq \{ v_1, \dots v_n\}$

6) Si on a au plus t processus pouvant tomber en panne, alors en admettant qu'un processus par ronde tombe en panne, à la ronde t+1 aucun processus ne tombera en panne : les valeurs manquantes dans le Val_p du processus p (s'il y en a) seront donc complétées à cette ronde là. Pour tout $(p, q) \in Viv(t+1)$, on aura bien $Val^{t+1}_p = Val^{t+1}_q$.

7)
```
pour i de 0 à t+1 :
        faire algorithme
décide
```

Terminaison : Un processus ***correct*** est un processus qui est vivant tout le long de l'exécution. Ici, les seuls processus qui ne peuvent pas décider sont les processus qui sont crashés (n'envoient et ne réceptionnent plus de messages), donc non correct.

Intégrité : Prouvé lors de la question 5.

Accord : Considérant qu'ils décident selon les mêmes conditions, alors ils auront tous les éléments pour décider de la même valeur, même en cas de pannes (Prouvé lors de la question 6).

8) Si on reprend le cas de la question 1, alors n = 4 et t = 2 (donc $t < n + 1$). Si un processus envoie à un processus puis crash, puis celui qui a reçu envoie à un processus et crash et ainsi de suite… ainsi on aura jamais $Val_p = Val_q$. [CF Question 1, j'ai envie de dire].
9) Pour qu'on arrive à un consensus, il faut pouvoir s'assurer que les processus ne connaîtront plus aucune pannes : sachant qu'on ne connaît ni le nombre de processus ni le nombre de processus pouvant tomber en panne, l'une des manières de procéder serait de garder un compteur du nombre de messages reçus au tour précédent : si un processus reçoit le même nombre de messages au tour r+1 qu'au tour r, alors tous les processus auront reçu le même nombre de messages durant les deux tours : il sera donc possible de parvenir à une décision, sachant que tous les processus décident sur la base des mêmes éléments. [Fonctionne également avec le contenu du message]