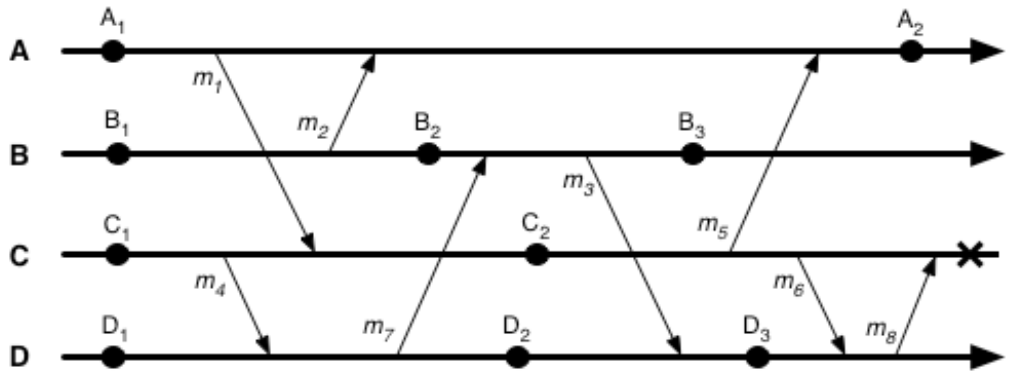
ARA – Novembre 2010
Durée : 2 heures
Documentation autorisée : polycopie du cours
L. Arantes and O. Marin

<u>Répondre aux questions du premier exercice dans une feuille séparée.</u>

Exercice 1: Points de reprise

La figure suivante représente une exécution d'application. Au cours de celle-ci, des sauvegardes locales ont été effectuées sur disque de manière indépendante et spontanée par chacun des processus A, B, C, et D. Les points représentent des sauvegardes locales et les flèches des messages échangés au cours de chaque exécution.



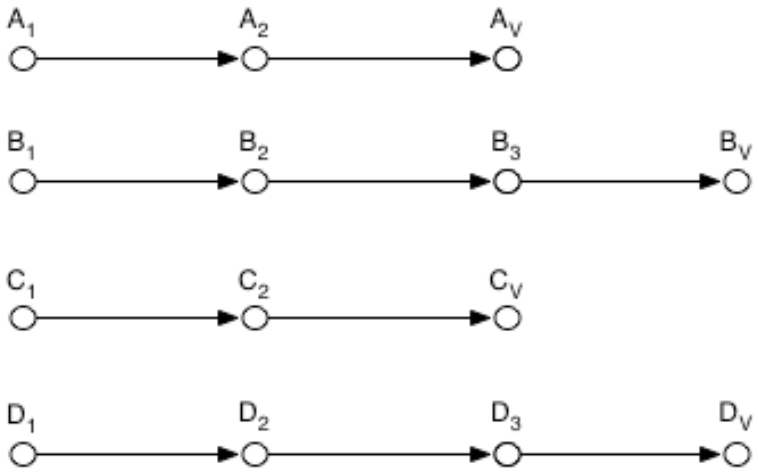
1.1

Existe-t-il dans cette exécution des points de sauvegarde qui ne pourront faire partie d'aucune ligne de recouvrement ? Si oui, combien et comment peut-on prédire ce résultat ?

Une panne de courant interrompt soudainement l'exécution sur le nœud C, à l'endroit où la ligne temporelle associée à l'exécution du processus est marquée d'une croix. Au moment de redémarrer, on cherche à établir la coupure cohérente la plus avancée dans le temps.

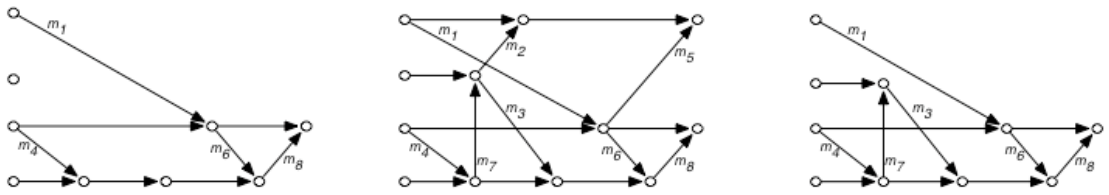
1.2

Complétez le graphe de dépendances de reprises ci-dessous, marquez les points de reprise qui doivent être abandonnés, et déduisez-en l'ensemble de points de sauvegarde {Aw, Bx, Cy, Dz} qui correspond à la coupure cohérente la plus avancée dans le temps pour l'exécution.



1.3

On utilise l'approche du système Manetho pour faire de la journalisation causale. Indiquez lequel des graphes de précédences suivants est obtenu ainsi sur le nœud C après la réception du message m_8. Justifiez votre réponse.



Graphe A Graphe B Graphe C

1.4

Juste avant d'être interrompu, le nœud C parvient à créer un point de reprise, et suit l'algorithme de Koo & Toueg 87 vu en cours pour coordonner les points de reprises. Indiquez le(s) message(s) de coordination envoyé(s) ? Cela déclenche-t-il la création de nouveaux points de reprise ? Si oui, sur quel(s) nœud(s) ?

Exercice 2: Protocole de Diffusion Fiable Uniforme

Considérez la spécification de la diffusion fiable uniforme :

1. **Validité :** si un processus correct diffuse le message *m*, alors tous les processus corrects délivrent *m*
2. **Accord uniforme** : si un processus (**correct** ou **fautif**) délivre le message *m*, alors tous les membres corrects délivrent *m.*
3. **Intégrité uniforme**: Un message *m* est délivré au plus une fois à tout processus (**correct** ou **fautif**), et seulement s'il a été diffusé par un processus.

Le système est **asynchrone** et les canaux sont fiables. Les processus sont susceptibles de subir des pannes franches. Le système possède au début N processus corrects : $\Pi = \{p_1, p_2, .. p_n\}$.

2.1

Complétez le corps de la primitive **Unif_real_broadcast (m)** afin d'offrir une diffusion fiable uniforme et le code lorsqu'un processus reçoit un message (**upon event** recv(m, p_j)) afin d'offrir une délivrance du message *m* selon la spécification décrite ci-dessus. Vous pouvez ajouter des variables locales si nécessaire. Cependant vous **ne pouvez pas** ajouter des détecteurs de fautes à votre solution.

Observation pour vous aider : Il faut penser à diffuser un message avant le délivrer.

Processus p_i :

Unif_real_broadcast (m)
. . .

upon event recv(m, p_j) **do**
. . .
Unif_real_deliver(m) /* délivrer le message */

Exercice 3: Détecteur de fautes

On considère un ensemble de processus $\Pi = \{p_1, p_2, .., p_n\}$ communiquant par messages. Ils sont organisés logiquement dans un anneau selon l'ordre croissant de leurs identifiants. Cependant, chaque processus peut aussi envoyer un message directement à chacun des autres processus. Les liens de communication sont fiables.

On considère des fautes de type « crash ». Le réseau est considéré partiellement synchrone : après le temps GST (inconnu), il existe des bornes sur les délais de transmission et de traitement des messages et tous les messages émis avant GST ont été reçus.

L'algorithme de détecteur de fautes décrit ci-dessous utilise un mécanisme de « ping » pour surveiller les autres sites : pour surveiller *q, p* lui envoie le message « ARE-YOU-ALIVE ? » et *q* lui répond avec le message « I-AM-ALIVE ». On note Δ_{GST} le délai maximum après GST entre l'envoie d'un message « ARE-YOU-ALIVE ? » et la réception du message « I-AM-ALIVE » correspondant.

On note *succ(p)* le successeur de *p* et *pred(p)* son prédécesseur dans l'anneau. L_p correspond à la liste de nœuds que *p* suspecte d'être en panne.

Considérez le détecteur de fautes associé à *p* dont le code est le suivant :

1. **Initialisation**
2. $target_p = succ(p)$

3. $L_p = \emptyset$
4. Pour tout q : $\Delta_{p,q}$ = Timeout par défaut

5. Tâche 1 :
6. Boucle infinie
7. Si ($target_p$!= p) {
8. send ARE-YOU-ALIVE ? to $target_p$
9. $t_{out} = \Delta_{p,targetp}$
10. received=false
11. Attendre t_{out}
12. Si (! received) {
13. $\Delta_{p,targetp} = \Delta_{p,targetp} + 1$
14. $L_p = L_p \cup \{ target_p \}$
15. $target_p$ = succ ($target_p$)
16. }
17. }
18. Fin boucle

19. Tâche 2 :
20. Boucle infinie
21. réception du message *m* de *q*
22. Si (q = $target_p$)
23. received=true
24. Sinon
25. Si (m = I AM ALIVE) {
26. Si (q in L_p) {
27. $L_p = L_p - \{ q,\ldots, pred(target_p) \}$
28. $target_p$ = q,
29. received=true
30. }
31. }
32. Sinon
33. send I-AM-ALIVE to q
34. Fin boucle

3.1

Quels sont les processus que *p* surveille à l'instant t?

3.2

Est-il possible que *p* ne surveille jamais un processus *q*? Justifiez votre réponse.

3.3

Supposons que *p* surveille *r* et qu'il reçoit un message du processus *q*, qu'il suspectait être en panne. Quels sont les processus que *p* continue à suspecter après la réception du message de *q*? Justifiez votre réponse.

3.4

Supposons que l'exécution est arrivée à GST et que $t_{out} > \Delta_{GST}$ pour tous les sites corrects. Supposons aussi que $\Pi = \{p_1, p_2, p_3, p_4, p_5\}$ et que p_2 et p_3 se trouvent en panne à GST. Donnez le contenu de L_p pour tous les processus corrects.

3.5

Quelle est la classe de justesse implémentée par l'algorithme ?

3.6

Quelles seraient les modifications nécessaires pour que l'algorithme implémente un Ω? Il n'est pas nécessaire de donner le code

3.7

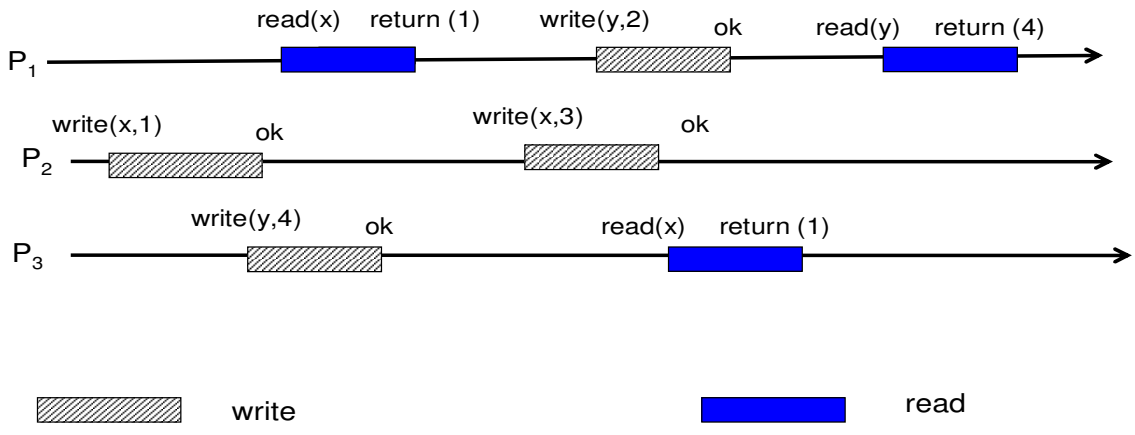
On note *corr_succ(p)* le processus successeur de *p* qui est le processus correct le plus proche de *p*.
Montrez informellement que pour toute exécution, tous processus correct *p* va suspecter *corr_succ(p)* un nombre fini de fois.

3.8

Modifiez l'algorithme afin d'offrir un détecteur de défaillance de classe <> P. Indiquez seulement les lignes à ajouter et/ou modifier.

Exercice 4: Modèle de Cohérence

Considérez le schéma ci-dessous :



4.1

A quel type de cohérence ce schéma correspond-t-il ? Justifiez votre réponse.