ARA – Mars 2013-2014
Durée : 2 heures
Documentation autorisée : polycopie du cours
Luciana Arantes, Pierre Sens

Exercice 1: Détecteur de fautes

On considère un ensemble de processus $\Pi = \{p_1, p_2, .., p_n\}$ communiquant par messages. Les liens de communication sont bidirectionnels et fiables. On ne considère que des fautes du type « crash ». Le réseau forme un graphe complet. Il y au plus *f* processus fautifs.

On définit le « *scope* » de la propriété de justesse (accuracy) comme le nombre *k* de processus qui ne suspectent pas de façon erronée un processus correct. Soit CORRECT l'ensemble de processus corrects et $f<k$. La propriété de *k-justesse* est satisfaite s'il existe un ensemble Q de processus tel quel :

1. $|Q| = k$ (scope) ;
2. $Q \cap CORRECT \neq \emptyset$;
3. Il existe $p \in Q \cap CORRECT$ tel que : pour tout $q \in Q$: q ne suspecte pas p.

La *k-justesse faible* (weak k-accuracy) est satisfaite si la k-justesse est satisfaite à terme (eventually).

On définit $\Diamond S_k$ comme la classe de détecteur de défaillance qui satisfait la complétude forte (strong completeness) et la k-justesse faible.

L'algorithme ci-dessous transforme un détecteur $\Diamond S_k$ en un détecteur de classe $\Diamond S$. Le code correspond au processus p_i.
Processus p_i possède deux variables locales :

- *k_suspect [1:n, 1:n]* (matrice de booleéns) correspond à la vision global de p. La ligne *k_suspect [i,*]* est mise à jour en appelant le détecteur $\Diamond S_k$ (ligne 2). Si *k_suspect[i,m]=true*, p_i k-suspecte p_m. Pour les lignes telles que i≠j, si *k_suspect[j,m]=true*, p_i considère que p_j n'est pas fautif et que p_j k-suspecte p_m.
- *suspected [n]* (vecteur de booléens): si suspected[j] = true, p_i suspects p_j.

```
(1)  init: ∀(x, ℓ) : k_suspect_i[x, ℓ] ← false;  ∀ℓ : suspected_i[ℓ] ← false;

(2)  repeat forever: ∀p_j: do send K_SUSPICION(k_suspect_i[i, ∗], i) to p_j enddo

(3)  when K_SUSPICION(k_susp, j) is received:
(4)    ∀ℓ : k_suspect_i[j, ℓ] ← k_susp[ℓ];
(5)    ∀ℓ: if ¬ (k_suspect_i[j, ℓ])
(6)                        then  suspected_i[ℓ] ← false
(7)                        else  if | {x | k_suspect_i[x, ℓ]} | > (n − k)
(8)                                then  suspected_i[ℓ] ← true
(9)                                      ∀y : k_suspect_i[ℓ, y] ← false
(10)       endif                      endif
```

1.1

Expliquez informellement (2 ou 3 lignes) la k-justesse. Est-ce que les k processus ne comprennent que des processus corrects ?

1.2

Pourquoi est-ce que le nombre de processus qui suspectent p_l doit être supérieur à n-k (ligne 7) pour que p_i considère p_l comme fautif ?

1.3

Montrez que l'algorithme assure la complétude forte.

1.4

Soit p_u un processus correct qui à terme n'est plus jamais suspecté par un ensemble Q de k processus ($p_u \in$ Q). Montrez qu'à terme | {x | k_suspect[x,u]} | ≤ (n-k).

1.5

Quels sont les avantages et/ou inconvénients de $\Diamond S_k$ par rapport à $\Diamond S$?

Exercice 2: Δ-diffusion causale

Nous considérons N processus qui communiquent par passage de message. Leur horloge locale est synchronisée par rapport à une horloge globale physique.

Nous définissons *la durée de vie* Δ d'un message comme la durée de temps physique, après l'envoi du message, pendant laquelle le contenu du message est valable pour l'application. Un message qui arrive à sa destination après sa durée de vie Δ est inutile et sera donc rejeté (« discarded »). Nous considérons que la valeur de Δ est la même pour tous les messages.

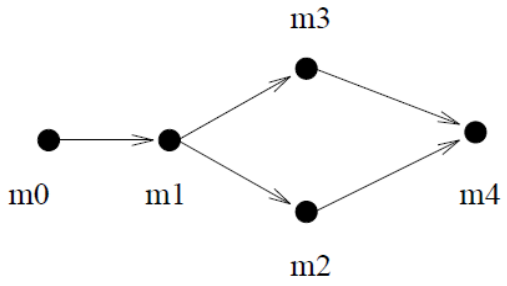
Nous voulons offrir un algorithme qui assure la Δ-diffusion causale.

Une exécution respecte la Δ-diffusion causale (Δ-causal ordering) si :

- tous les messages qui arrivent pendant leur intervalle de vie Δ sont délivrés dans l'intervalle Δ ; Les autres ne sont jamais délivrés.
- La délivrance des messages respecte l'ordre causale :
  Tous messages m_1 et m_2 qui arrivent à leur destination pendant leur intervalle Δ respectent la précédence causale : Si *send*(m_1) -> *send* (m_2) et m_1 et m_2 ont le même processus destinataire, alors *delivery*(m_1) -> *delivery* (m_2) .

Par conséquent, lorsqu'un message arrive sur un site, soit il est rejeté parce que sa date butoir de délivrance (« deadline ») a été dépassée soit sa délivrance est retardée jusqu'à ce que la précédence causale décrite ci-dessus soit satisfaite.

On dit que m_1 est un *prédécesseur immédiat* de m_2, si *send*(m_1) -> *send* (m_2) et il n'existe pas un message *m* tel que : *send*(m_1) -> *send* (*m*) et *send*(*m*) -> *send* (m_2). Par exemple, la figure ci-dessous montre un graphe de causalité des messages où m_2 et m_3 sont des prédécesseurs immédiats de m_4. Donc, dans l'algorithme, la délivrance de *m* sera retardée jusqu'à ce que ses messages *prédécesseurs immédiats* aient été soit délivrés soit rejetés dû au non respect de leur « deadline ».



Tout message est estampillé avec (*sender_id, send_time*) où *sender_id* est l'identifiant de l'émetteur et *send_time* est l'instant où le message a été envoyé. De plus, chaque message contient aussi l'ensemble d'estampilles de ses messages *prédécesseurs immédiats*. Pour un

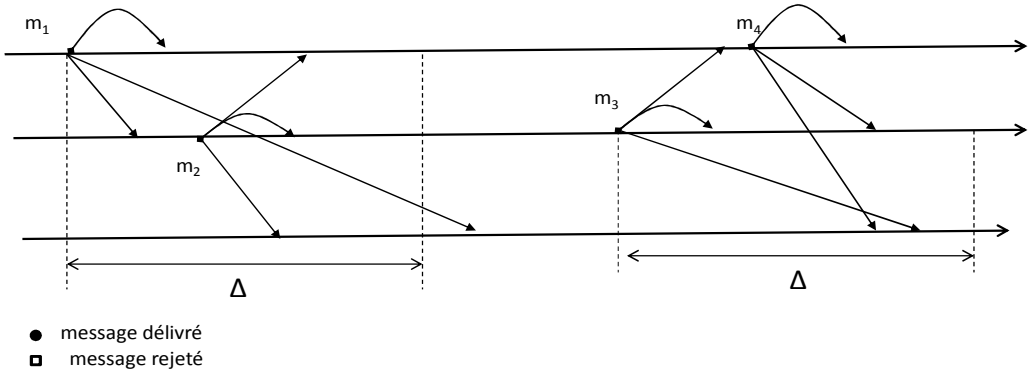
message *m*, cet ensemble est noté CB_m (Causal Barrier). Par exemple, le message m_4 contiendra une estampille correspondant à l'envoie de m_4 plus un ensemble CB_{m4} avec les estampilles d'envoi de m_3 et m_2.

Chaque processus p_i possède les variables locales suivantes:

- $current\_time_i$: heure physique courante;
- $sent\_time_i$: instant de la dernière diffusion de p_i ;
- $DEL_i[n]$: DEL*i*[j]=d indique que le dernier message diffusé par p_j et délivré par p_i a été envoyé au temps physique *d* ;
- CB_i : ensemble de messages *prédécesseurs immédiats*.

Le code de la fonction *Causal_broadcast (m)* est le suivant :

```
Causal_broadcast (m) {
   send_time_i = current_time_i ;
   for (j<1; j<=N; j++)
      send (m, i, send_time_i, CB_i)  to p_j;
  CB_i=(i,send_time_i);
 }
```



2.1

La figure ci-dessus montre la diffusion et réception de 4 messages ainsi que la durée de vie Δ. Complétez la figure en indiquant quand les messages sont délivrés (rond noir) ou rejetés (carré blanc).

2.2

Par rapport au contenu du message *m* et les valeurs des variables locales de p_i, quelle est la condition pour que p_i rejette *m* lors de la réception de ce message?

2.3

Par rapport au contenu du message *m* et les valeurs des variables locales de p_i, quelle est la condition pour que p_i délivre *m* ?

2.4

Donnez le code de la fonction *Upon reception (m, j, send_time$_m$, CB$_m$)* lors de la réception par p_i du message *m* envoyé par p_j. Le message *m* devra être soit rejeté soit délivré (après une attente, si nécessaire).

2.5

Est-ce que l'algorithme marche si les canaux ne sont pas faibles ? Justifiez votre réponse.

2.6

Sous quelles conditions la Δ-diffusion causale est égale à la diffusion causale originale vue en cours ?