Polytech Marseille

TD 3
Algorithmes répartis : exclusion mutuelle et élection

Informatique - Formation par alternance HUGo

1 Horloges logiques

Le problème : Dans un système réparti, chaque processus i possède sa propre horloge H_i fonctionnant à sa propre vitesse. Nous allons voir ici comment définir un temps global cohérent (à défaut d'être identique) pour tous les processus. Par cohérent, nous voulons dire que le temps global doit respecter les deux règles suivantes :

— si un événement e à lieu avant un événement f au regard du temps local au processus i alors e doit aussi avoir lieu avant f au regard du temps global ;
— un envoi de message doit toujours avoir lieu avant sa réception au regard du temps global.

Solution : Pour cela nous allons considérer un temps logique, qui n'est pas lié au temps physique, mais qui permet tout de même de préciser l'ordonnancement des événements (internes ou de communication) qui ont lieu sur les processus. Ce temps logique est mesuré à l'aide d'une horloge logique discrète. Il existe plusieurs types d'horloges logiques : les scalaires, vectorielles ou matricielles. Nous présentons ici les horloges logiques scalaires de Lamport.

Exercice 1. Estampillage scalaire de Lamport

On considère le schéma de datation des horloges de Lamport. Dans ce schéma, une date ou estampille est représentée par un couple $(site, compteur)$. L'estampillage de Lamport consiste à affecter à chaque événements (internes, d'envoi ou de réception) une estampille. Il s'effectue ainsi : Pour un site s donné, le premier événement est estampillé $(s, 1)$.

— tout message envoyé porte l'estampille de l'événement émetteur ;
— Soit e un événement interne ou un événement d'émission du site s. Soit f l'événement précédent immédiatement e sur le site s. Si f est estampillé (s, cpt) alors e est estampillé $(s, cpt + 1)$;
— Soit e un événement de réception d'un message estampillé (s', cpt'). Soit f l'événement précédent immédiatement e sur le site s. Si f est estampillé (s, cpt), alors e porte l'estampille $(s, 1 + max(cpt, cpt'))$.

Questions.

On considère le diagramme de séquence de la figure 1.

1. Trouvez deux événements d'émission qui ne sont pas causalement dépendants.
2. Trouvez tous les événements de réception qui sont postérieurs à r dans le temps absolu, mais qui ne sont pas causalement liés à r.
3. Décorer le diagramme de séquence en précisant la valeur de chaque horloge locale à chaque événement, ainsi que la date ajoutée aux messages envoyés.
4. L'estampille a été ici définie par un couple $(site, compteur)$. En vous appuyant sur le diagramme de séquence complété, déterminer si on pouvait se passer de l'information $site$ dans l'estampille pour ordonner les messages seulement.

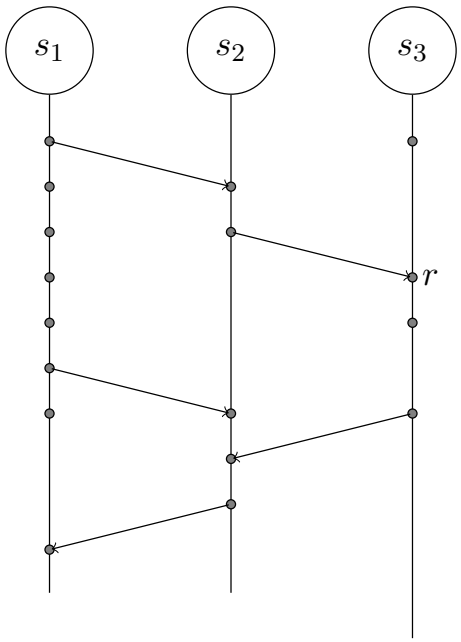


FIGURE 1 – Un diagramme de séquence

2 Exclusion mutuelle dans les systèmes répartis

Le problème : Plusieurs processus partagent une ressource commune qui ne peut être utilisée que par un processus à la fois (ex : une imprimante). Comment garantir le bon fonctionnement du système, les processus étant répartis sur plusieurs sites distants ?

Différence avec le cas centralisé : Une section critique est une zone de code concernant la ressource partagée : un seul processus au plus doit être en section critique afin de garantir une utilisation correcte de la ressource. Les processus voulant accéder à la ressource sont répartis. Par conséquent, il n'est plus possible d'utiliser les algorithmes d'exclusion mutuelle basés sur l'hypothèse que les processus utilisent le même processeur (par exemple, les sémaphores).

Les approches réparties : On distingue généralement deux approches pour gérer l'exclusion mutuelle dans un système réparti :

— approche par jeton (Ex : algorithmes de Le Lann et Suzuki & Kasami)
— approche par permission (Ex : algorithmes de Lamport, Ricart & Agrawal et Maekawa)

Dans les algorithmes suivants, les réceptions sont bloquantes.

Exercice 2. Exclusion mutuelle : algorithme de Le Lann

Nous allons étudier ici l'algorithme de Le Lann (1977). L'idée : On va ordonner les processus sur un anneau logique (sur lequel circule un jeton) afin de gérer l'exclusion mutuelle.

Questions.

1. Imaginez la suite de l'algorithme. Donnez des arguments qui montre que votre algorithme fonctionne (au plus un processus en section critique, pas de famine...)
2. Avantages et inconvénients ?

Il existe de nombreuses variantes d'algorithmes utilisant un jeton pour simuler la permission d'entrer en section critique plus ou moins efficace suivant la structure logique utilisée pour transmettre le jeton (anneau, arbre...).

Exercice 3. Exclusion mutuelle : algorithme de Ricart et Agrawala

Nous considérons l'algorithme de Ricart & Agrawala (1981). On suppose que les communications sont sûres (pas de perte de message) et FIFO, et que les processus sont identifiés par un numéro et sont fiables (pas de panne). Le principe de l'algorithme de Ricart & Agrawala est le suivant. Un processus demande la permission d'accéder à une ressource à tous les processus (sauf lui). Un processus peut donner sa permission à plusieurs processus à la fois. Chaque processus P_i tient à jour une liste locale L_i des processus en attente d'une permission. Un processus peut accéder à sa section critique si tous les autres processus lui ont donné leur permission. Pour l'estampillage, nous utilisons l'algorithme d'estampillage scalaire de Lamport.

L'algorithme de Ricart & Agrawala est le suivant :

$DemandeurSC(i)$
Lorsqu'un processus P_i veut entrer dans sa section critique :
1. il détermine son horloge locale H_i
2. il envoie à tous les autres processus un message ($Demande_i$; H_i), où H_i est l'estampille
3. il attend que tous les processus contactés lui envoient leur permission
4. il entre dans sa section critique (i.e. toutes les permissions ont été reçues)
5. puis, lorsqu'il sort de sa section critique, il envoie une permission (OK_i ; H_i) à tous les processus en attente dans la liste L_i et vide la liste L_i

$NonDemandeurSC(i)$
Lorsqu'un processus P_i reçoit une demande de permission ($Demande_j$; H_j) d'un processus P_j :
1. P_i met à jour son horloge locale H_i
2. Si P_i n'est pas en train d'attendre pour entrer en section critique, il envoie son autorisation (OK_i ; H_i) à P_j
3. Sinon,
3a. si P_i a demandé la ressource partagée avant P_j alors P_j est rangé dans la liste locale L_i suivant son estampille H_j avec les autres processus en attente de permission
3b. si P_j a demandé la ressource avant P_i alors P_i envoie la permission (OK_i ; H_i) à P_j

Questions.

1. Appliquez cet algorithme sur le diagramme de la figure 2 (les flèches représentent les messages du type $(Demande_i, H_i)$) et déterminez l'ordre d'entrée des processus en section critique.
2. Est-ce que l'algorithme est toujours correct si les communications ne sont pas FIFO ?
3. Que doit-on faire si un processus tombe en panne après qu'un autre lui ait demandé la permission ?

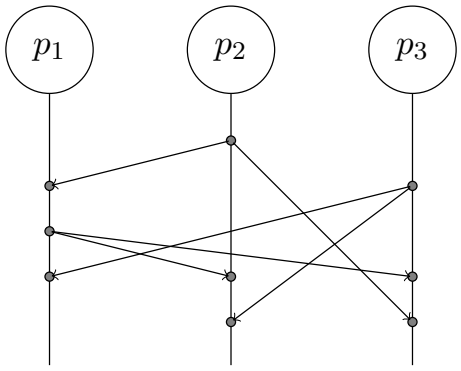


FIGURE 2 – Encore un diagramme de séquence

3 Élection dans les systèmes répartis

Exercice 4. Élection d'un coordinateur : algorithme de Chang et Roberts

Pré-requis.

— chaque processus possède un identifiant unique
— chaque processus connaît l'identifiant des autres processus
— un processus ne connaît pas les processus défaillants
— les communications doivent être fiables et synchrones (on connaît une borne sur le temps de transmission d'un message)
— les processus sont physiquement ou logiquement ordonnés sur un anneau par un rang
— les processus ont la même politique d'élection (par ex. le processus de rang le plus grand est élu).

Résultat.

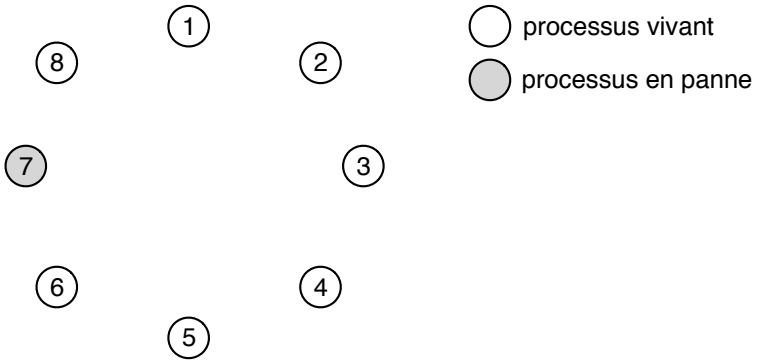
un seul processus est élu coordinateur au bout d'un temps fini

L'algorithme.

1. lorsqu'un processus de rang i détecte la mort du coordinateur, il envoie un message $(election, \{i\})$ à son successeur sur l'anneau (i.e le premier processus non défaillant situé après i sur l'anneau)
2. lorsqu'un processus de rang j reçoit un message $(election, L)$, où L est un ensemble de rangs, il ajoute son rang à la liste puis transmet le message $(election, L \cup \{j\})$ à son successeur
3. lorsque le message revient à l'initiateur (le processus de rang i), l'initiateur choisit le coordinateur dans la liste : par ex, le processus de rang le plus grand. Supposons que ce rang soit k.
4. un message $(coordinateur, k, L)$ est alors émis où k est le rang du nouveau processus coordinateur, et L est la liste des processus vivants.
5. le message circule sur l'anneau jusqu'à être reçu par tous.

Questions.

1. Faire fonctionner l'algorithme sur l'anneau de la figure suivante lorsque 6 détecte la panne.
2. Quelle est la complexité de cet algorithme (en nombre de messages) ?



Exercice 5. Élection d'un coordinateur : algorithme de Bully

Pré-requis.

— chaque processus possède un identifiant unique
— chaque processus connaît l'identifiant des autres processus
— un processus ne connaît pas les processus défaillants
— les communications doivent être fiables et synchrones (on connaît une borne sur le temps de transmission d'un message)

Résultat

Le processus non-défaillant avec l'identifiant le plus grand est élu coordinateur au bout d'un temps fini.

L'algorithme.

Chaque processus i peut exécuter les deux algorithmes $Initiateur(i)$ et $NonInitiateur(i)$ en parallèle. Lorsqu'un processus i détecte une panne du coordinateur, il initie l'algorithme d'élection en exécutant $Initiateur(i)$. Cet algorithme va provoquer l'exécution de l'algorithme $NonInitiateur(j)$ par tous les autres processus j.

$Initiateur(i)$

1. i envoie un message $(election, i)$ à tous les processus $j > i$
2. i attend un acquittement ack pendant T ms.
   Si un processus répond avant T ms, **GOTO 6**.
3. i s'autoproclame coordinateur
4. i envoie $(coordinateur, i)$ à tous les processus non défaillants
5. **FIN**
6. i attend un message du type $(coordinateur, j)$ pendant T' ms.
   Si aucun message de ce type n'est reçu avant T' ms, **GOTO 1**.
7. i détermine que j est le coordinateur
8. **FIN**.

$NonInitiateur(j)$

1. j reçoit un message $(election, i)$ (alors nous avons forcément $j > i$)
2. j envoie l'acquittement ack à i
3. j exécute $Initiateur(j)$
4. **FIN**

Questions.

1. Faire fonctionner l'algorithme avec le scénario suivant (il y a 8 processus au total, le processus 5 est le coordinateur) :
   — initialement seul le processus 5 est en panne ;
   — le processus 3 détecte que le processus *coordinateur* 5 est en panne (i.e. le processus 5 ne répond pas au message *request* envoyé par 3) ;
   — le processus 8 tombe en panne durant l'étape 3 de $Initiateur(8)$ avant l'envoi de $(coordinateur, 8)$.
2. Quelle est la complexité de cette méthode en nombre de messages ?
3. Comment un processus qui aurait redémarré après une panne peut-il réintégrer correctement le groupe de processus participant à l'élection ?