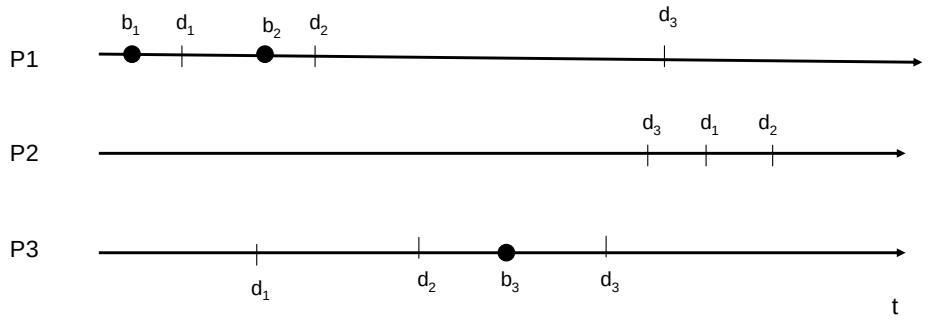
TD Protocole de Diffusion
ARA
Octobre 2020

Exercice 1

Considérez le scénario ci-dessous :



● b_i = diffusion (broadcast) of message i

| d_i = délivrance correspondante à la diffusion du message i

1.1

Est-ce que le scénario représente une diffusion FIFO, causal ou totalement ordonnée ? Justifiez votre réponse par rapport à chaqu'un de ces type de diffusion.

Exercice 2 : Protocole de Diffusion FIFO et Causal

Soit un groupe *fermé* composés de trois processus P_1, et P_2. Considérez que :
- A t= 0, P_1 diffuse dans le groupe le message m_1 et P_2 diffuse m_2.
- Après la délivrance de m_1, P_2 diffuse le message m_3 dans le groupe.

2.1

Quels sont les ordres de délivrance de messages possibles si on utilise :
- **diffusion FIFO**
- **CBCAST (diffusion causal) ;**

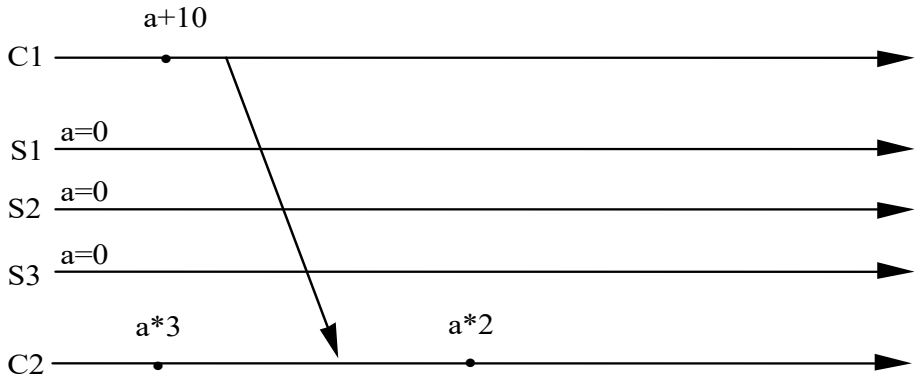
Exercice 3 Protocole de Diffusion Atomique et Causal

Des clients envoient des requêtes à 3 serveurs répliqués afin de mettre à jour des données stockées sur les serveurs.

3.1

Quels sont les avantages et les inconvénients de répliquer les serveurs ?

Les serveurs gèrent des mises à jour sur une variable ‘a’. Deux clients C1 et C2 diffusent à à chaque fois leurs mises à jour aux trois serveurs. Soit le scénario suivant:



Les points représentent les diffusions des requêtes aux serveurs.

3.2

- Quelles sont les valeurs possibles de ‘a’ sur les différents serveurs si les clients utilisent un ABCAST ou un CBCAST ?

- Dans quels cas peut-on avoir des valeurs différentes sur les serveurs ?

Exercice 4 : Choix du protocole de diffusion

On considère un groupe de serveurs de log répliqués permettant de stocker des événements d’une application répartie composée de processus communiquant. Chaque serveur maintient un log qui contient une liste d’événements. Une insertion est toujours faite en fin de log (append). Un processus p_i peut envoyer au groupe de serveurs trois types d’événements :

- (1) Emission de message vers p_j
- (2) Réception de message en provenance de p_j
- (3) Mise à jour de l’état local

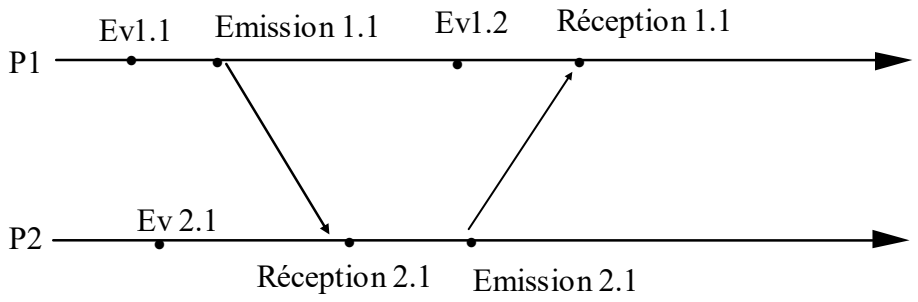
De plus, les processus peuvent envoyer un ordre aux serveurs pour compacter leur log. Pour assurer la cohérence des logs tous les serveurs doivent gérer les ordres de compaction dans le même ordre.

On dispose de trois primitives de diffusions permettant de faire de la diffusion fiable, un ABCAST et un CBCAST.

4.1

Quel type de diffusion est associé à l’envoi de chacune des opérations sur le serveur (justifier brièvement) ?

Soit la configuration suivante. Les points représentent les événements envoyés aux serveurs répliqués. `Ev i.k`, `Emission i,k` et `Réception i,k` correspondent respectivement au kième événement local, émission et réception sur Pi.



4.2

Indiquez le contenu possible des logs à la fin de cette séquence en supposant qu'initialement les logs sont vides.

Exercice 5 : Diffusion fiable avec détecteur de fautes parfait

L'algorithme de diffusion fiable (reliable broadcast) vu en cours envoie N*(N-1) messages par diffusion étant *N* le nombre de nœuds. Nous voulons offrir un algorithme de diffusion fiable que n'envoi que *N* messages en absence de fautes. Pour cela, nous utiliserons un détecteur de fautes parfait *P*. Si un processus *q* tombe en panne, *p* est a terme informé et il n'y a pas de fausses suspicions:

Processus p :
upon <crash, q>
$correct_p = correct_p \setminus \{q\}$

5.1

Donnez le pseudo code du nouvel algorithme de diffusion fiable en utilisant un détecteur de fautes parfait.

5.2

Quel est le nombre de messages en plus envoyés si un processus *p* tombe en panne ?

5.3

Quels sont les avantages et les inconvénients d'un tel algorithme ?

5.4

Quelle modification devrait-on faire dans l'algorithme si on utilise un détecteur de fautes <>*P* au lieu de P ? <> *P* offre la complétude forte mais la justesse faible (il peut y avoir des fausses suspicions). Justifiez votre réponse.

Exercice 6 : Protocole de Diffusion Fiable Uniforme

Considérez la spécification de la diffusion fiable uniforme :

- **Validité :** si un processus correct diffuse le message *m*, alors tous les processus corrects délivrent *m*
- **Accord uniforme** : si un processus (**correct** ou **fautif**) délivre le message *m*, alors tous les membres corrects délivrent *m.*
- **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.

Supposez aussi l'existence d'un détecteur de défaillance **parfait** sur chaque processus *p*.

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

6.1

Nous voulons offrir le pseudo code de l'algorithme de diffusion fiable uniforme. Est-ce qu'un processus peut délivrer un message *m* dès qu'il le reçoit ? Justifiez votre réponse.

6.2

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 ci-dessus décrite. Vous pouvez ajouter d'autres variables locales si vous trouvez nécessaire ainsi que d'autres fonctions et/ou événements ou même changer le code du événement <*crash,q*>.

Processus p_i :

Variables locales :

$correct_{pi} = \Pi$; /* Π = ensemble de tous les processus du système */

. . .

Unif_real_broadcast (m)

. . .

upon event recv(m, p_j) **do**

. . .

Real_deliver(m) /* délivrer le message */

upon <crash, p_j>

$correct_{pi} = correct_{pi} \setminus \{ p_j \}$

6.3

Considérez que *f* est le nombre maximal de pannes pouvant se produire (f < N). Est-ce que votre algorithme pourrait être optimisé ? Justifiez votre réponse sans donner un nouveau pseudo code.

Exercice 7: Diffusion Totalement ordonnée

Nous considérons un système avec *n* processus (p_1, p_2,.., p_n) identifiés de façon unique : 1,2,3,.. n. Les canaux sont fiables et **FIFO.**

Le but est d'offrir un algorithme de diffusion totalement ordonnée en utilisant les horloges logiques scalaires de Lamport enrichies avec l'identifiants des processus pour la définition d'un ordre total sur les messages délivrés. Cela dit, tous les processus délivrent les messages en respectant cet ordre total.

RAPPEL : Horloges scalaire et ordre total:

Chaque site p_i gère un compteur d'entier h_i, initialisé à 0 et modifié de la façon suivante :

- si *e* est un événement interne à p_i : h_i++ ; tm (e)=h_i
- si *e* est l'envoi d'un message *m* par p_i : h_i++ ; tm (e)=h_i ; envoyer (<m, hi>)
- si *e* est la réception du message <m,h> : h_i= max (h_i,h)+1; tm (e)=h_i.

où *tm(e)* correspond à l'estampillage affecté à l'événement *e*.

La relation de l'ordre total << sur deux événement e_i de p_i et e_j de p_j , estampillés avec h_i et h_j respectivement, est définie de la façon suivante :

$e_i << e_j \Leftrightarrow h_i < h_j$ or ($h_i=h_j$ and i<j)

Tous les processus appellent la fonction *TO_broadcast (m)* **en permanence** à des intervalles fixes. Les processus ne se terminent pas. Chaque processus p_i gère une variable vecteur locale $H_i[n]$ initialisée à {0,…,0} où $H_i[j]$ indique la valeur d'horloge de p_j connue par p_i. Le code de la fonction *TO_broadcast (m)* est le suivant :

```
TO_broadcast (m) {
   Hi[i]++;
   for j=1,..,N send (<m,Hi[i]>) to j
}
```

Vous ne pouvez pas modifier la fonction *TO_broadcst(m)*.

7.1

Quand est-ce qu'un processus est sûr qu'il peut délivrer un message reçu?

7.2

Donnez le code correspondant à la réception par p_i du message *m* envoyé par p_j :

Upon recv (<m,h>) from j

Le message doit être délivré selon l'ordre total défini ci-dessus. Indiquez les nouvelles variables locales que vous ajouterez à l'algorithme ainsi que leur initialisation.