Systèmes et algorithmique répartis

ENSEEIHT/DIMA & Master Informatique/PSMSC
1h45, documents autorisés

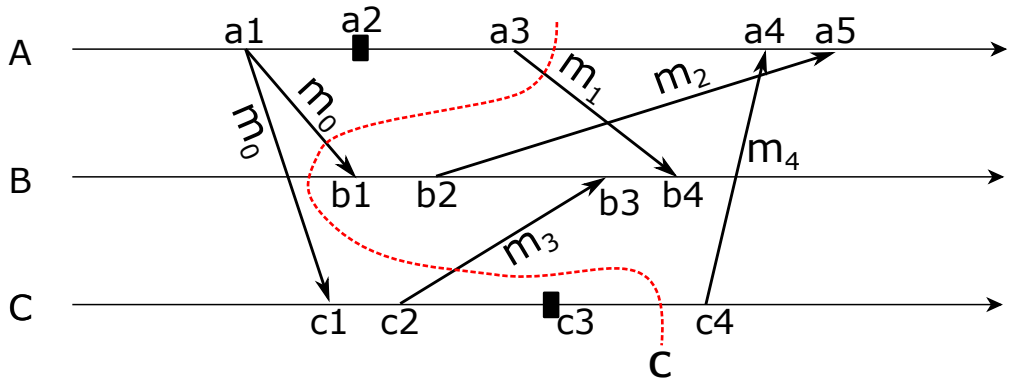
28 novembre 2017

Toutes les réponses doivent être justifiées. Un simple "oui", "non" ou "42" est considéré comme une absence de réponse.

Les éléments de correction sont extrêmement sommaires. Il ne s'agit pas d'une correction mais d'indications pour construire une réponse.

1 Calcul réparti et causalité (6 points)

On considère les échanges de messages entre 3 sites A, B, C représentés par le chronogramme suivant :



Questions (0.75 pt par question)

1. Quel est le passé causal de b_3 ?
   $\{c_2, b_2, b_1, c_1, a_1\}$
2. Quel est le futur causal de c_2 ?
   $\{b_3, b_4, c_3, c_4, a_4, a_5\}$
3. La coupure C est-elle cohérente ? Justifier la réponse.
   Oui, pas de "trou" sur un site, et pas de réception sans son émission.
4. Si la coupure C est cohérente, y inclure un unique événement du calcul la rendant incohérente ; si elle est incohérente, en exclure un unique événement du calcul la rendant cohérente.
   Par exemple, ajouter a_4.

5. Déterminer la valeur de l'horloge de Lamport de b_4. Justifier cette valeur soit par raisonnement, soit en calculant les horloges des événements le précédant.
   $(B, 5)$
6. Déterminer la valeur de l'horloge vectorielle de b_4. Justifier cette valeur soit par raisonnement, soit en calculant les horloges des événements le précédant.
   $(3, 4, 2)$
7. Déterminer l'histoire causale des messages m_2 et m_4 et en déduire si les délivrances a_4, a_5 respectent la causalité.
   $HC(m_2) = \{m_0\}, HC(m_4) = \{m_3, m_0\}$. $m_2 \notin HC(m_4) \wedge m_4 \notin HC(m_2)$ *donc indépendants causalement et n'importe quel ordre de délivrance est valide.*
8. Si les délivrances a_4, a_5 respectent la causalité, ajouter un unique message pour qu'elles ne respectent plus la causalité ; en cas inverse, enlever un unique message pour qu'elles deviennent bien ordonnées.
   Ajouter un message de B vers C, émis après b_2 *et reçu avant* c_4.

2 Consensus synchrone (14 points)

2.1 Objectif

L'objectif est d'étudier plusieurs algorithmes de consensus pour le modèle synchrone, avec différents types de défaillances. Dans la suite, N est le nombre total de processus et f est le nombre de processus défaillants.

Rappel : dans le modèle synchrone, un calcul est décomposé est une suite de tours (*round*). À chaque tour, le comportement des processus est synchrone : chacun diffuse (ou peut diffuser) une valeur envoyée à tous les autres ; chacun récupère les messages qui le concernent ; chacun fait un traitement local. Puis l'ensemble du système passe au tour suivant.

2.2 Définition du consensus

Le système est composé de N processus $\mathcal{P} = \{P_1, \ldots, P_N\}$. Chaque processus P_i propose une valeur v_i. À la terminaison de l'algorithme, chaque processus décide d'une valeur d_i, ou $\perp$ pour indiquer une absence de décision. On appelle *bad* l'ensemble des processus défaillants, et *good* l'ensemble des processus corrects. Les propriétés que doit vérifier un algorithme de consensus sont :

Accord : la valeur décidée est la même pour tous les processus corrects :

$$\forall i, j \in 1..N : \Box(P_i \in good \wedge P_j \in good \wedge d_i \neq \perp \wedge d_j \neq \perp \Rightarrow d_i = d_j)$$

Accord uniforme : la décision est la même pour tous les processus qui décident (corrects ou pas) :

$$\forall i, j \in 1..N : \Box(d_i \neq \perp \wedge d_j \neq \perp \Rightarrow d_i = d_j)$$

Intégrité : tout processus décide au plus une fois (sa décision est définitive).

$$\forall i \in 1..N, \forall k \neq \perp : \Box(d_i = k \Rightarrow \Box(d_i = k))$$

Validité : la valeur décidée est l'une des valeurs proposées :

$$\forall i \in 1..N : \Box(d_i \neq \perp \Rightarrow \exists j \in 1..N : d_i = v_j)$$

Terminaison : tout processus correct décide d'une valeur proposée au bout d'un temps fini.

$$\forall i \in 1..N : (\Diamond P_i \in bad) \vee (\Diamond d_i \neq \perp)$$

Dans toute la suite, l'opération `diffuser(m)` est l'envoi *non atomique* et non déterministe d'un message à tous les processus, y compris l'émetteur :

$$\texttt{diffuser(m)} \triangleq \forall i \in 1..N : \text{envoyer m à } p_i$$

2.3 Consensus sans défaillance

En absence de défaillance, on propose l'algorithme en 1 tour suivant :

```
Processus P_i
  var locales : est_i, V_i
  début tour synchrone
     diffuser(v_i)
     V_i ← { valeurs reçues pendant ce tour }
     est_i ← min(V_i)
  fin tour
  d_i ← est_i
```

Questions (0.5 pt par question)

1. Cet algorithme a-t-il la propriété d'accord ?
   Oui, tous les processus reçoivent les mêmes valeurs et tous calculent la même fonction déterministe → tous choisissent la même valeur.
2. Cet algorithme a-t-il la propriété d'intégrité ?
   Oui, un unique point de décision puis le processus a fini.
3. Cet algorithme a-t-il la propriété de validité ?
   Oui, les valeurs échangées sont toutes des valeurs proposées et le min d'un ensemble appartient à l'ensemble.
4. Cet algorithme a-t-il la propriété de terminaison ?
   Oui : un seul tour synchrone.

2.4 Consensus avec arrêt de processus

On considère la défaillance d'arrêt : un processus défaillant peut s'arrêter définitivement (*crash*) en tout point de son exécution. Rappel : `diffuser` n'est pas atomique et un processus défaillant peut s'arrêter après avoir envoyé un message à seulement un sous-ensemble des processus.

Pour tolérer f défaillances, on propose l'algorithme suivant en $f+1$ tours :

```
Processus P_i
  var locales : est_i, prevest_i, V_i
  est_i ← v_i
  prevest_i ← ⊥
  pour tour = 1,...,f+1 faire
     début tour synchrone
        si est_i ≠ prevest_i alors diffuser(est_i); fin si
        V_i ← { valeurs reçues pendant ce tour }
        prevest_i ← est_i
        est_i ← min(V_i ∪ {est_i})
     fin tour
  fin pour
  d_i ← est_i
```

Questions (0.75 pt par question)

5. Donner un contre-exemple où l'algorithme ne vérifie pas la propriété d'accord s'il ne faisait que f tours avec f processus défaillants par arrêt. (indication : utiliser 3 processus ayant 3 valeurs proposées distinctes)
   3 processus dont 1 défaillant, 1 tour.
   Processus 1 propose 1, l'envoie à 2 et crashe. 2 propose 2 et l'envoie à tous. 3 propose 3 et l'envoie à tous.
   2 reçoit $\{1,2,3\}$ et décide 1.
   3 reçoit $\{2,3\}$ et décide 2.
   Accord invalidé.

6. Montrer que cet algorithme vérifie la propriété d'accord avec $f+1$ tours et au plus f processus défaillants par arrêt.

   Il existe au moins un tour où il n'y a pas de défaillance. Dans ce tour, tous les processus corrects jusque là prennent la même valeur (min des valeurs diffusées jusque là), comme dans l'algorithme précédent. Les tours ultérieurs ne changent plus leur valeur.
   Le pire cas est le schéma du poly, où chaque processus possédant le min défaille après l'avoir envoyée un à unique autre. Mais en $f+1$ tours, tous les corrects obtiennent cette valeur (dans le pire cas, au dernier tour).

7. Cet algorithme a-t-il la propriété de validité ?

   oui, uniquement des valeurs proposées sont échangées, et le min d'un ensemble appartient à l'ensemble.

8. Cet algorithme a-t-il la propriété de terminaison ?

   oui, $f+1$ tours.

2.5 Un détour en asynchrone

Pour cette section uniquement, on considère un système *asynchrone*. Il n'y a donc plus de tour synchrone avec tous les processus. On se donne un détecteur de fautes parfait (P). Chaque processus avance par tour, indépendamment des autres, en attendant à chacun de ses tours un message de chacun des processus non suspectés.

```
Processus P_i
  var locales : est_i, V_i, tour_i
  est_i ← v_i
  pour tour_i = 1,...,f+1 faire
        diffuser(est_i);
        attendre un message de chaque processus non suspecté par le détecteur P
        V_i ← { valeurs reçues pendant ce tour }
        est_i ← min(V_i ∪ {est_i})
  fin pour
  d_i ← est_i
```

Questions (1 pt par question)

9. Montrer que deux processus non défaillants ne peuvent être qu'à au plus un tour d'écart, i.e. $\forall i, j : \Box(|tour_i - tour_j| \leq 1)$.

   Un processus ne passe au tour suivant que quand il a reçu un message de tous les autres non défaillants. Comme le détecteur est parfait, il ne passera pas au tour suivant trop tôt (ce qui serait le cas avec un détecteur sans propriété d'exactitude). Il ne pourra passer au tour d'après que quand les autres l'auront rattrapé. Plus précisément, un processus P_i ne peut passer au tour $k+1$ que s'il a reçu k messages de P_j non défaillant (ça se démontre trivialement par récurrence). Comme un processus ne diffuse qu'un message par tour, P_j est au moins au tour k. cqfd.

10. Montrer qu'en conséquence, le comportement asynchrone est assimilable au comportement synchrone de la section 2.4 et que l'algorithme a les mêmes propriétés.

Vu que le détecteur de défaillance est parfait, un processus ne reste pas bloqué dans un tour : il finira par recevoir un message de tous les corrects, et il cessera d'attendre un message des processus crashés. Tous les processus corrects font donc $f+1$ tours. À chaque tour, chacun a le même comportement que dans un tour synchrone, et aucun n'avance "trop vite" par rapport aux autres (question précédente). En conséquence, la coupure C_k constituée par les états des processus au k-ième tour est cohérente. La suite des C_k est identique aux états de l'algo synchrone.

2.6 Consensus avec omission

On considère les défaillances d'omission. Un processus qui défaille par omission peut ignorer un nombre arbitraire de messages (omission en réception) ou ne diffuser un message qu'à un sous-ensemble des processus (omission en émission). Un processus est considéré comme défaillant (*bad*) s'il effectue au moins une défaillance d'omission ou d'arrêt au cours de sa vie.

On ajoute à l'algorithme 2.4 un tour supplémentaire pour déterminer si une valeur est retenue à une majorité stricte. Note : ne pas oublier qu'un processus peut aussi effectuer des omissions ou un crash dans ce dernier tour !

```
Processus P_i
  var locales : est_i, prevest_i, V_i
  est_i ← v_i ; prevest_i ← ⊥
  pour tour = 1,...,f+1 faire
     début tour synchrone
        si est_i ≠ prevest_i alors diffuser(est_i); fin si
        V_i ← { valeurs reçues pendant ce tour }
        prevest_i ← est_i
        est_i ← min(V_i ∪ {est_i})
     fin tour
  fin pour
  début tour synchrone
     diffuser(est_i)
     V_i = { valeurs reçues pendant ce tour }
     si ∃ v ∈ V_i qui a été reçue plus de N/2 fois alors
         d_i ← v
     sinon
         d_i ← ⊥
     fin si
  fin tour
```

Questions (1 pt par question)

11. Montrer que sur l'algorithme tolérant aux arrêts (section 2.4), un unique processus faisant des omissions suffit à invalider l'algorithme, et ce quelque soit le nombre de tours.

    Processus ayant la valeur minimale, défaillant pendant tous les tours, sauf au dernier où il envoie sa valeur à certains seulement ⇒ ceux-ci obtiennent le min, pas les autres.

12. Montrer que le nouvel algorithme ne vérifie pas la propriété de terminaison si $2 * f = N$.

    Trivial : $N/2$ font omission au dernier tour ⇒ pas de majorité.

13. Montrer que tout processus *good* (non défaillant) termine nécessairement avec une décision différente de $\perp$ si $2 * f < N$.
    À $f+1$, tous les corrects (qui sont au moins $N/2$) ont la même valeur, à $f+2$ un processus correct obtient une majorité sur la valeur des corrects.
14. Montrer que cet algorithme vérifie la propriété d'accord.
    TODO cf ci-dessus
15. Montrer que, contrairement au cas du 2.4, un processus défaillant (*bad*) peut parfois décider d'une valeur. Cet algorithme a-t-il la propriété d'accord uniforme ?
    Pour qu'un processus décide (qu'il soit défaillant ou pas), il faut qu'il obtienne une valeur majoritaire (reçue plus de $N/2$ au dernier tour). Cette valeur est majoritaire, donc aussi celle que les autres processus décideront. $\Rightarrow$ tous les processus qui décident décident identiquement $\Rightarrow$ accord uniforme.

2.7 Consensus avec défaillances byzantines

Outre les défaillances précédentes, un processus peut maintenant mentir quand il diffuse une valeur : dans le même tour, il peut envoyer une valeur différente à différents processus, voire une valeur proposée par aucun processus. On suppose qu'on dispose d'un détecteur de fautes parfait, qui permet à un processus P_i de savoir si P_j est défaillant (i.e. s'il ment).

Question (1 pt par question)

16. Proposer une adaptation simple de l'algorithme précédent pour traiter ces défaillances byzantines. Combien de défaillances peut-il tolérer par rapport au nombre de sites et/ou de tours ?
    Si un site ment $\Rightarrow$ ignorer la valeur envoyée.
    Toute défaillance de mensonge devient une défaillance d'omission.
    $2 * f < N$.
17. Ce nouvel algorithme a-t-il la propriété de validité (y compris pour les processus menteurs qui décideraient d'une valeur) ?
    Oui, uniquement des valeurs proposées sont échangées par les processus corrects (vu qu'ils rejettent les valeurs des processus menteurs). Une coalition de menteurs ne peut atteindre la majorité de $N/2$.