Algorithmique Distribuée

Cours 3

Shantanu Das

http://pageperso.lif.univ-mrs.fr/~shantanu.das/M1algodist/

Aix-Marseille Université

13-14 mars 2018

Rappel : Dernier Cours

- Calcul dans les arbres : Convergecast (*Saturation*)
- Construction d'arbres couvrants
- Bourne inférieur pour la complexité d'Election : $O(m + n \log n)$
- Algorithme d'GHS pour Election (optimal complexité)
- Algorithme YO-YO, Algorithme de KKM (Algorithme moduler pour Election)
- Election sur les familles de graphes spécifiques
- Synchronizers

Aujourd'hui

Les Réseaux Synchrones

1. Calculs Synchrones et Synchronisation, horloge logique
2. Detection de Propriétés Globales : Deadlock et Terminaison
3. Tolerance aux Pannes (Fautes permanents)

Réseaux Synchrones

Suppositions :

- Chaque processus a une horloge.
- Les horloges sont synchronisées.
- Chaque message prend une unité de temps pour traverser un arête.

- C'est possible de trouver plus efficace algorithmes, quand le réseau est synchrone.
- C'est plus facile de détecter le terminaison, détecter les fautes, etc.

Temps contre Coût de communication

- Avec les horloges, un processus peut attendre pour un temps précis.
- On peut réduire le quantite de communication.
- Compris entre le temps et nombre de messages transmis

Temps contre Coût de communication

- Avec les horloges, un processus peut attendre pour un temps précis.
- On peut réduire le quantite de communication.
- Compris entre le temps et nombre de messages transmis

Exemple : Election dans l'anneau avec O(n) messages est possible dans les réseaux synchrones.

(Rappel : Election dans les anneaux asynchrones nécessite $O(n \log n)$ messages.)

Election en $O(n)$ messages

Election dans un anneau synchrone avec UID

(Elire le plus petit UID)

- Ralentir les messages qui ne sont pas utiles.
- Un message contenant i attend 2^i temps, avant d'être transmis.
- Le message avec la plus petit UID sera le plus rapide à traverser l'anneau.
- Le deuxième plus petit ne peut traverser que la moitié

Complexité : $O(n)$ messages, $O(n \cdot 2^i)$ temps

Complexité en Bits : $O(n \log n)$ bits.

Communication en utilisant l'attente

Supposition : Tout processus ont la même horloge.

- Pour envoyer le valeur x
- Transmettre un bit pour commencer.
- Attendre x unité de temps
- Transmettre un bit pour finir.

Complexité : 2 Bits transmis,
Complexité en temps $= x + 2$

Election avec Attendre

Suppositions :

- Tout processus commence en même temps.
- Tout processus connait n (Topologie arbitraire).

Algorithm :

- Si $UID = i$, attendre $n * i$ unités de temps.
- Si aucun message reçu, diffuser “Je suis Elu "

Complexité : $O(m)$ Bits transmis,
temps $= n * Min(UID)$

Synchronization des Horloges

Synchronisation des horloges

- Si les horloges des processus ne sont pas d'accord.
- Le but : Toutes les horloges doivent être synchronisées.

Synchronisation des horloges

- Si les horloges des processus ne sont pas d'accord.
- Le but : Toutes les horloges doivent être synchronisées.

Algorithme d'Inondations

- (1) Envoyer le valeur de horloge H au voisins.
- (2-...) $H = 1+$ MAX(H, valeurs reçus) ; Renvoyer H à tous.
- Après D rondes, toutes les horloges ont la même valeur.

Probleme de Firing Squad

- Tout les processus se commence pas a la même temps.
- Les horloges des processus ne sont pas d'accord.
- Le But : Tout les processus doivent arriver à l'état FIRE à la même temps.

Probleme de Firing Squad

- Tout les processus se commence pas a la même temps.
- Les horloges des processus ne sont pas d'accord.
- Le But : Tout les processus doivent arriver à l'état FIRE à la même temps.

- Commencer le même algorithme avec $H = 0$.
- Quand $H = D$, changer à l'état FIRE.

Complexité : 2mD messages, D rondes.

Horloge Logique [Lamport]

Dans le réseau asynchrone, il n'y a pas des horloges, mais il existe un ordre causal sur les événements.

Définition

Pour les événements X et Y, X précède Y ssi (i) ils sont sur le meme processus et X qui s'est passé avant Y, ou (ii) X est le cause de Y, ou (iii) il ya une série d'événements $X = e_1, e_2, e_3, \ldots e_k = Y$ dans cet ordre.

Horloge Logique [Lamport]

Algorithme pour Horloge logique

- Commencez avec l'horloge $H = 0$;
- Incrémenter H pour chaque événement (envoyer, recevoir, interne) ;
- Ajouter la valeur H à chaque message ;
- Sur réception d'un message m, $H = MAX(H, H_m) + 1$;

2. Detection de Propriétés Globales

Detection de Propriétés Globales

- Détection de Deadlock (Interblocage)
- Détection de Terminaison

Detection de Propriétés Globales

- Détection de Deadlock (Interblocage)
- Détection de Terminaison

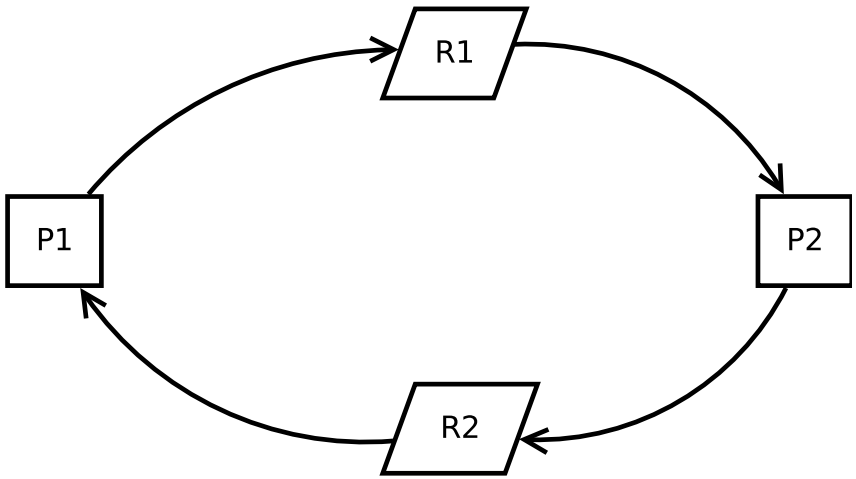
Suppositions :

- Les processus sont en train de réaliser un calcul.
- Le réseau est synchrone ou asynchrone.

Detection de Deadlock

Définition

Deadlock *Si il existe un ensemble de processus* $P_1, P_2, \ldots P_k$ *telle que chaque* P_i *est en attente pour le prochain processus* $P_{(i+1) \mod k}$



Un processus peut attendre pour plusieurs autres processus à le même temps.

Detection de Deadlock

Graphe d'attente (W) : Il y a un arête dirige de P_x à P_y ssi processus P_x est en attendre pour processus P_y

- Le graphe W est une DAG (Graphe Dirige Acyclique) => Pas de Deadlock
- Il existe un cycle dirige dans le graphe W => Deadlock existe !

Un processus envoie un "Verifier" message vers les arêtes orientées. Si le message revient finalement à lui-même, il existe un cycle dirige => Deadlock !

Detection de Terminaison (DT)

T1 Terminaison Implicite
- Tout processus sont passives.
- Pas de messages en transit.

T2 Terminaison Explicite
- Tout les processus sait que [T1] est vrai.

Définition (Algorithme pour D.T.)

Un algorithme qui transforme le condition [T1] à [T2].

Algorithme D.T.

Propriétés :

- **Non-Interference :** n'influence pas le calcul originale.
- **Liveness :** Une fois [T1] est vrai, tout le monde dit "Terminé" dans un temps fini
- **Safety :** Si un processus dit "Terminé" $=>$ [T1] est vrai

Théorème

Un algorithme D.T. nécessite la transmission de $\Omega(M_c)$ messages, si le calcul originale a transmis M_c messages.

Algorithme de Shavit-Francez

- Une forêt F où les sommets sont les processus et les messages.
- Chaque initiateur est racine de un arbre.
- Si q deviens actif a cause de une message de p, $(q, p) \in F$.
- Si p envoie un message m, $(m, p) \in F$.
- Quand un message m est reçu, m est supprimé de F.
- Si une processus est passive est n'a pas d'enfants, il est supprimé de F.

Si $F = \phi => [T1]$ est vrai. Comment informer ?

Algorithme de Shavit-Francez

- Une forêt F où les sommets sont les processus et les messages.
- Chaque initiateur est racine de un arbre.
- Si q deviens actif a cause de une message de p, $(q, p) \in F$.
- Si p envoie un message m, $(m, p) \in F$.
- Quand un message m est reçu, m est supprimé de F.
- Si une processus est passive est n'a pas d'enfants, il est supprimé de F.

Si $F = \phi \Rightarrow$ [T1] est vrai. Comment informer ?

- Utiliser Saturation avec délai sur T.
- Complexité : $O(n + M_c)$ messages

Algorithme Récupération du Crédit

- Chaque actif processus possède une crédit $\in [0, 1]$.
- Chaque message transmis possède une crédit $\in [0, 1]$.
- Pour envoyer un message, partager le crédit.
- Supposition : Seule Initiateur P_0 (Crédit $= 1$).

Algorithme Récupération du Crédit

- Chaque actif processus possède une crédit $\in [0, 1]$.
- Chaque message transmis possède une crédit $\in [0, 1]$.
- Pour envoyer un message, partager le crédit.
- Supposition : Seule Initiateur P_0 (Crédit $= 1$).
- Si un processus deviens passive, il donne son crédit à P_0
- Si P_0 est passive et crédit(P_0)$= 1$, le calcul a terminé.

3. Tolérance au Pannes

Tolérance au Pannes

- Dans les systèmes centralisés une seule faute peut arrêter le système.
- Les systèmes distribués ont l'avantage de tolérer les fautes.
  (e.g. les processus corrects peut prendre le travail des ceux qui sont en panné)
- Il faut trouver les algorithmes qui tolère les fautes.

Types de fautes

- Fautes Permanents
  - Défaillances de processus
  - Défaillances de liens
- Fautes Temporaires
  - omissions de messages
  - additions de messages
  - corruptions de messages
- Fautes Byzantins

Algorithme tolérant aux pannes

Définition

Un algorithme est f-tolérant si elle produit un résultat correct même si f ou moins processus (liens) sont cassées.

Algorithme tolérant aux pannes

Définition

Un algorithme est f-tolérant si elle produit un résultat correct même si f ou moins processus (liens) sont cassées.

- Si $f = n$, on peut faire rien !
- Dans une arbre, il n'existe pas un algorithme 1-tolérante.
- Pour l'existence d'algorithme f-tolérante, c'est nécessaire que le réseau est $(f + 1)$-connexe.

Exemple : Problème de Consensus

Consensus

Etant donnés des processus avec une valeur initiale $\in \{0, 1\}$, chaque processus doit décider une valeur t.q.

- **Accord :** Tous les processus qui décident une valeur décident la même valeur.
- **Validité :** Si tous les processus ont w comme valeur initiale, alors c'est la valeur décidée.
- **Terminaison :** Tous les processus (non défaillants) décident.

Exemple : Problème de Consensus

Théorème

Pour $f < n$ défaillances, il existe un algorithme f-tolérant pour consensus dans un réseau synchrone qui est $(f+1)$-connexe.

- Si le réseau est un graphe complet et $f < n$,
  - Envoyer le valeur initiale à chaque voisin ;
  - Attendre ; Recevoir les valeurs de voisins ;
  - Décider sur l'ET logique(&) de tout les valeurs vu.
- Complexité : 2 rondes, $2m$ messages
- Correction :
  - Tous les processus recoivent le même ensemble de valeurs, donc ils décident la même valeur.

Exemple : Problème de Consensus

Théorème

Pour $f < n$ défaillances, il existe un algorithme f-tolérant pour consensus dans un réseau synchrone qui est $(f+1)$-connexe.

- Pour le graphes quelconques, répéter n fois
  - Envoyer tout les valeurs vu à chaque voisin ;
  - Recevoir les valeurs de voisins ;

  et décider sur l'ET logique de tout les valeurs vu.
- Complexité = $O(n^3)$ messages, $O(n)$ temps.

Exemple : Problème de Consensus

Théorème

Pour $f < n$ défaillances, il existe un algorithme f-tolérant pour consensus dans un réseau synchrone qui est $(f+1)$-connexe.

- Pour le graphes quelconques, répéter n fois
  - Envoyer tout les valeurs vu à chaque voisin ;
  - Recevoir les valeurs de voisins ;

  et décider sur l'ET logique de tout les valeurs vu.
- Complexité = $O(n^3)$ messages, $O(n)$ temps.
- C'est possible de réduire le complexité à $O(n^2)$.
- Envoyer 0 la première fois un 0 est vu, sinon attendre n rondes ;

Consensus avec Fautes Byzantins

- Faute Byzantin : Un processus qui ne suit pas les règles de l'algorithme.
- Il peut envoyer des informations incorrectes au autres processus.

Théorème

Pour $f < n/3$ fautes byzantins, il existe un algorithme f-tolérant pour consensus dans un réseau complet synchrone

Byzantin Consensus Algorithme

Suppositions

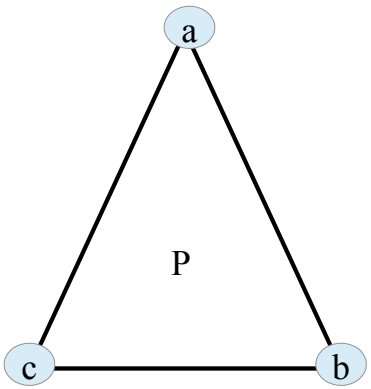
$f < n/3$ Fautes Byzantins, Graphe complet, Synchrone

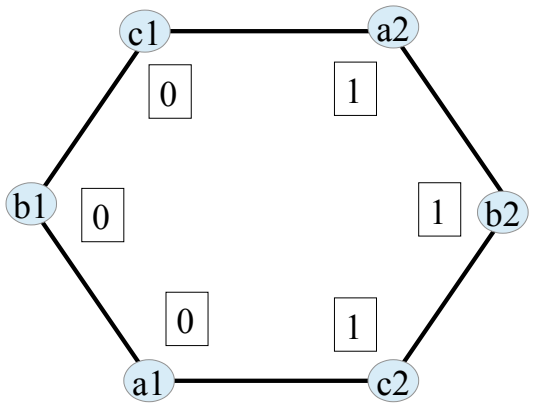
- Si $v = 0$, envoyer $(0, UID, t)$ à temps t
- Si reçu $(0, x, t)$ de x à temps $t + 1$, envoyer 'Echo$(0, x, t)$'
- Si reçu 'Echo$(0, x, t)$' de $f + 1$ voisins, à temps $t + 2$, accepter 0 comme le valeur de processus x.
- Si accepté 0 de $2f + 1$ voisins, décider 0.
- Si pas encore décidé 0 après $2f + 2$ rondes, => décider 1.

Impossibilité

Théorème

Avec $f \geq n/3$ fautes byzantins, consensus est impossible dans un réseau synchrone.

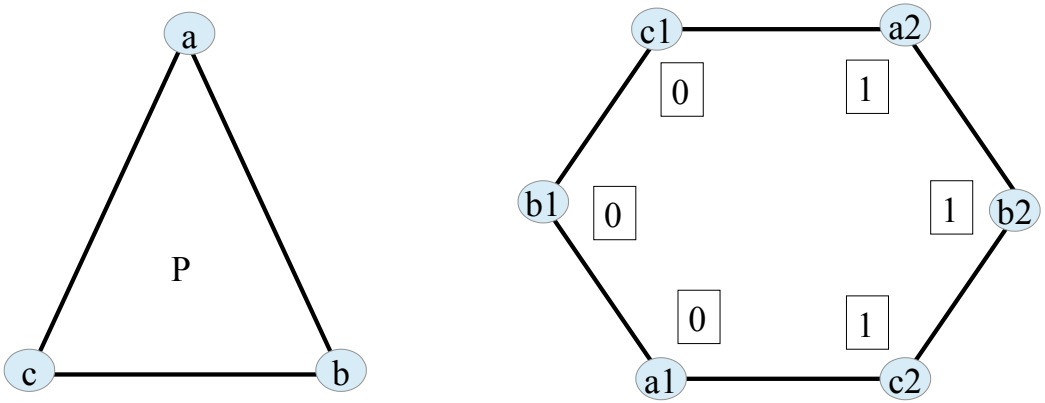




Impossibilité

Théorème

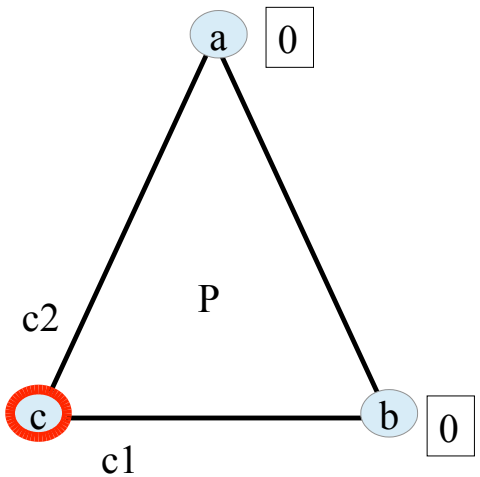
Avec $f \geq n/3$ fautes byzantins, consensus est impossible dans un réseau synchrone.

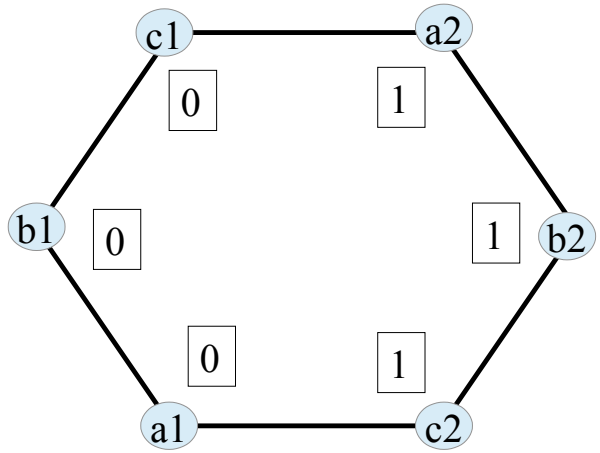


Supposons qu'il existe un algorithme P pour consensus $(n = 3\,;\, f = 1)$

Impossibilité

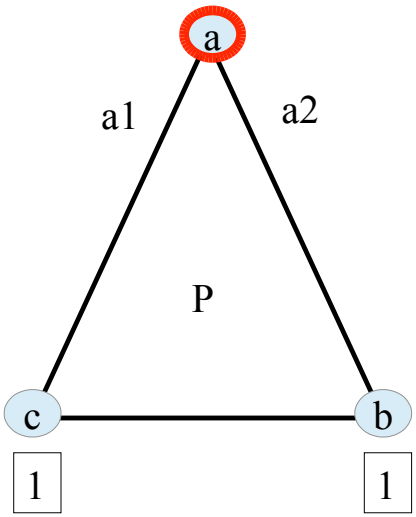
Processus c est byzantin ;
a et b doit décider 0 $=>$ $a1$ et $b1$ decide 0.

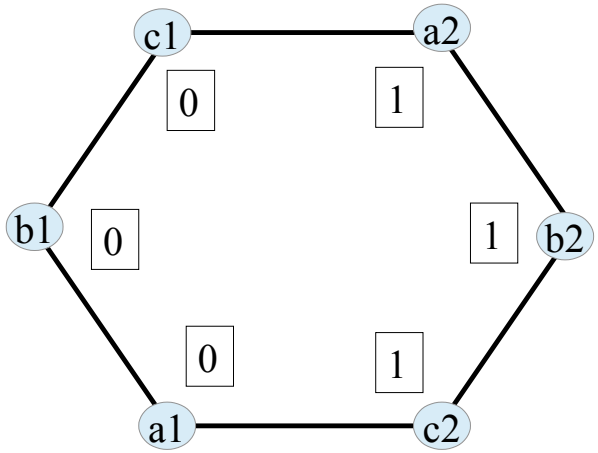




Impossibilité

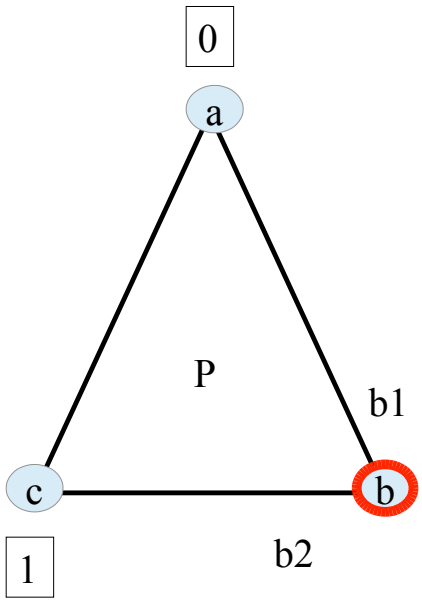
Processus a est byzantin ;
b et c doit décider 1 $\Rightarrow$ $b2$ et $c2$ decide 1.

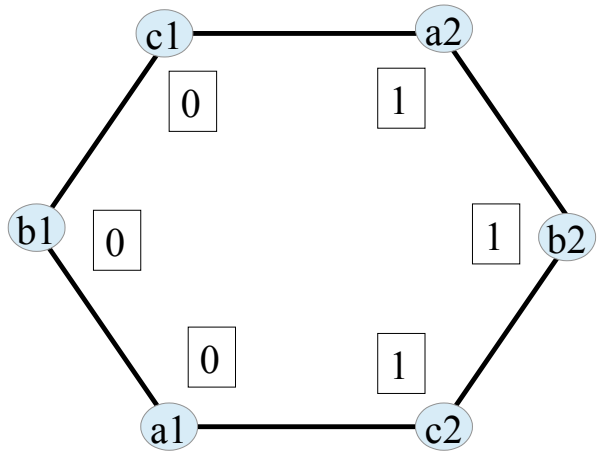




Impossibilité

Processus b est byzantin ; Decision(a)=Decision($a1$)=0 et Decision(c)=Decision($c2$)=1 (Pas de consensus !)





Alors l'algorithme P est faux =>

Il existe aucun algorithme pour consensus ($n = 3$; $f = 1$)

Consensus dans Réseaux Asynchrone

Dans les systèmes synchrones :

- Consensus avec $f < n$ défaillances, est possible.
- Consensus avec $f < n/3$ fautes byzantins, est possible.

Théorème (Fischer, Lynch, Paterson '85)

Le consensus est impossible pour des processus asynchrones s'il peut se produire au moins une défaillance de type crash.

Pas d'algorithme asynchrone pour consensus même si le réseau est complètement connexe.