Algorithmique Distribuée

Cours 4

Shantanu Das

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

Aix-Marseille Université

20-21 mars 2018

Rappel : Dernier Cours

Réseaux Synchrone

- Calculs Synchrones
  (Complexité en temps contre complexité en bits)
- Synchronisation d' horloges, Horloge logique
- Détection de Propriétés Globales
  - Deadlock Detection
  - Terminaison Detection
- Tolérance aux Pannes

  (Fautes permanentes : Défaillances ou Fautes Byzantines)

Tolérance au Pannes

Réseaux Synchrones :

- Fautes Permanentes (Crash)
  - Défaillances de processus
  - Défaillances de liens
- Fautes Temporaires
  - omissions de messages
  - additions de messages
  - corruptions de messages
- Fautes Byzantines

Est-ce qu'on peut tolérer des pannes dans les réseaux asynchrones ?

Consensus dans Réseaux Asynchrone

Dans les systèmes synchrones :

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

Dans les systèmes asynchrones :

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

Consensus avec Défaillances

Rappel : Problème de Consensus

- Le but : Etre d'accord sur un valeur $\{0, 1\}$.
  (Seulement les processus corrects doit être d'accord)
- Validité : Doit choisir une valeur proposée.
- Terminaison : Doit décider dans un temps fini.

Théorème (FLP : Fischer, Lynch, Paterson '85)

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

Le faute peut survenir à tout moment pendant l'exécution.

Preuve de la theorme FLP

Supposons qu'il existe un algorithme correct $\mathcal{A}$ pour consensus.

- Pendant l'execution de l'algorithme, un événement e au P est soit
  - Processus P a envoyé un message.
  - Processus P a reçu un message.
  - Processus P a fait un calcul interne.
- On considère un adversaire qui va jouer contre l'algorithme.
- L'adversaire peut ralentir un événement dans un processus P.

Preuve de la theorme FLP

- **Une configuration** : l'ensemble des états de tous les processus.

Preuve de la theorme FLP

- **Une configuration** : l'ensemble des états de tous les processus.
- 0-valent configuration : Si on commence avec cette configuration, seulement le résultat "0" est possible.
- 1-valent configuration : Si on commence avec cette configuration, seulement le résultat "1" est possible.
- Bivalent configuration : Si on commence avec cette configuration, les deux résultats "0" ou "1" sont possible.

Preuve de la theorème FLP

[L1] : Il existe une configuration initiale qui est bivalent.

[L2] : Si C est une configuration bivalent et e est un événement possible à C (selon l'algorithme $\mathcal{A}$), il existe une exécution de l'algorithme tel que la configuration après événement e, est encore bivalent.

[L1] & [L2] $=>$ l'algorithme $\mathcal{A}$ peut jamais décider une valeur.
Donc, il n'existe pas un algorithme correct pour Consensus

Preuve de la theorème FLP

Preuve du lemme [L1]

- Supposons que toutes les configurations initiales sont soit 0-valent ou 1-valent !
- La configuration initiale $\{0, 0, \ldots 0\}$ est 0-valent.
- La configuration initiale $\{1, 1, \ldots 1\}$ est 1-valent.
- Si on commence avec le configuration $\{0, 0, \ldots 0\}$ et on change les valeurs d'un processus après l'autre, on va trouver 2 configurations : C_x(0-valent) et C_y(1-valent) qui diffère dans la valeur de un seul processus P_i.
- Si P_i est en panne, C_x et C_y produisent le même résultat $=>$ une de ces deux configurations est bivalent !

Preuve du lemme [L2]

- C est une configuration **bivalent** et e est une événement possible à C. On suppose le contraire du [L2].

Preuve du lemme [L2]

- C est une configuration **bivalent** et e est une événement possible à C. On suppose le contraire du [L2].
- L'adversaire va ralentir l'événement e. Si $\mathcal{A}$ est correct, les autres processus ne peuvent pas attendre pour événement e.

Preuve du lemme [L2]

- C est une configuration **bivalent** et e est une événement possible à C. On suppose le contraire du [L2].
- L'adversaire va ralentir l'événement e. Si $\mathcal{A}$ est correct, les autres processus ne peuvent pas attendre pour événement e.
- Dans cette exécution, il existe une configuration C' qui n'est pas bivalent (et une configuration C'' juste avant qui est bivalent). Considerons l'événement e' qui transforme C'' à C'.

Preuve du lemme [L2]

- C est une configuration **bivalent** et e est une événement possible à C. On suppose le contraire du [L2].
- L'adversaire va ralentir l'événement e. Si $\mathcal{A}$ est correct, les autres processus ne peuvent pas attendre pour événement e.
- Dans cette exécution, il existe une configuration C' qui n'est pas bivalent (et une configuration C'' juste avant qui est bivalent). Considerons l'événement e' qui transforme C'' à C'.
- On peut montrer que C'' suivi par e est 0-valent, et, C' suivi par e',e est 1-valent (ou vice versa).

Preuve du lemme [L2]

- C est une configuration **bivalent** et e est une événement possible à C. On suppose le contraire du [L2].
- L'adversaire va ralentir l'événement e. Si $\mathcal{A}$ est correct, les autres processus ne peuvent pas attendre pour événement e.
- Dans cette exécution, il existe une configuration C' qui n'est pas bivalent (et une configuration C'' juste avant qui est bivalent). Considerons l'événement e' qui transforme C'' à C'.
- On peut montrer que C'' suivi par e est 0-valent, et, C' suivi par e',e est 1-valent (ou vice versa).
- Les événements e et e' forcement se produise dans le même processus P.

Preuve du lemme [L2]

- C est une configuration **bivalent** et e est une événement possible à C. On suppose le contraire du [L2].
- L'adversaire va ralentir l'événement e. Si $\mathcal{A}$ est correct, les autres processus ne peuvent pas attendre pour événement e.
- Dans cette exécution, il existe une configuration C' qui n'est pas bivalent (et une configuration C'' juste avant qui est bivalent). Considerons l'événement e' qui transforme C'' à C'.
- On peut montrer que C'' suivi par e est 0-valent, et, C' suivi par e',e est 1-valent (ou vice versa).
- Les événements e et e' forcement se produise dans le même processus P.
- Si P est en panne, les autres processus ne peuvent jamais décider => une contradiction ! Donc [L2] est vrai.

Consensus avec Défaillances

Théorème (FLP : Fischer, Lynch, Paterson '85)

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

Le faute peut survenir à tout moment pendant l'exécution.

Consensus avec Défaillances

Théorème (FLP : Fischer, Lynch, Paterson '85)

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

Le faute peut survenir à tout moment pendant l'exécution.

Si les fautes se produisent avant mais pas pendant l'exécution de l'algorithme ?

Consensus avec fautes passées

Si tous les fautes se produisent avant le commencement

- Il existe une algorithme f-tolérant pour $f < n/2$

Consensus avec fautes passées

Si tous les fautes se produisent avant le commencement

- Il existe une algorithme f-tolérant pour $f < n/2$

Algo (au moins $L = (n+1)/2$ processus corrects)

- Envoyer UID à tous ; (Le réseau est K_n)
- Attendre pour au moins L messages de voisins ;
- Si reçu message de w, ajouter un arête dirigée vers w ;
- Trouver la plus grande composante fortement connexe H dans l'ensemble d'arêtes dirigés. ($|H| \geq L$)
- Effectuer Election sur H et décider la valeur du Elu.

Consensus avec fautes passées

- L'algorithme est f-tolérant pour $f < n/2$ (quand tout les fautes se produisent au début)

Correction de l'algorithme

- Le réseau original est un graphe complet.
- Chaque processus correct est inclus dans H.
  $=>$ taille de $H \geq L = (n+1)/2$
- Il y a seulement une composante fortement connexe.
- Il y a seulement une processus qui est Elu et tous les processus corrects décident le valeur de Elu.

Détecteur de Fautes

Failure Detector (FD)

Détecteur de Fautes

Supposons qu'il y un mécanisme pour détecter des fautes.

Définition (Failure Detector (FD))

Un module qui donne une liste de processus qui sont suspectés d'être en panne.

- Trouver un algorithme qui utilise le FD pour éviter des fautes.
- Attendre pour une message de p_i seulement si $p_i \notin$ Liste de suspects.

Types de FD

1. Complet : Si p est en panne $=>$ p est suspecté par chaque q.
2. Précise : Si p est correcte, $p \notin \text{Suspect}(q)\ \forall q$
3. Faiblement précise : $\exists p$ tells que $p \notin \text{Suspect}(q)\ \forall q$.
4. Eventuellement Précis : Si p est correcte, $\exists$ temps p tells que $p \notin \text{Suspect}(q, t)\ \forall q$.
5. Parfaite : Complet + Précise
6. Eventuellement Parfaite : Complet + eventuellement Précis

Consensus Asynchrone avec FD

Hypothèse : Réseau K_n avec faiblement précis FD

Algorithme :

- Processus P_i avec valeur v_i : $X = v_i$;
- Pour ronde $r = 1$ à n :
  - Si $r == i$, Envoyer v_i à tous ;
  - Si $P_r \notin$ Suspect, attendre pour message v_r de P_r ;
  - Si reçu v_r, $X = v_r$;
- Décider la valeur X ;

Correction : Il y un moins un processus qui n'est pas suspecté. Tous les processus va recevoir sa valeur.

Problèmes avec FD

- Le liste de Suspecté n'est pas la même pour tous les processus.
- Si un processus p est suspecté, p n'est pas forcement en panne. Le résultat d'un algorithme dépend du nombre de processus suspectés.
- Difficile de construire un détecteur de fautes.
  - Il faut regarder les états des machines, les statistiques, les probabilités de défaillances, les temps de réponse, etc.

Types de Pannes

- Fautes Permanentes (Crash)
  - Défaillances de processus
  - Défaillances de liens
- Fautes Temporaires
  - omissions de messages
  - additions de messages
  - corruptions de messages
- Fautes Byzantines

Fautes Dynamiques

- Défaillances de liens (temporaires)
- Pas localisés : peuvent apparaître n'importe où dans le réseau.
- Il y a une borne sur le nombre de fautes par ronde.

Réseau Synchrone : Message α envoyé à temps t, message β reçu à temps $t+1$;

1. Omissions : $\alpha \neq \emptyset$, $\beta = \emptyset$
2. Additions : $\alpha = \emptyset$, $\beta \neq \emptyset$
3. Corruptions : $\alpha, \beta \neq \emptyset$, $\alpha \neq \beta$

Problème d'Accord

Problème de k-Accord : Au moins k processus doivent être d'accord

1. Unanimité : $k = n$ (Consensus)
2. Majorité : $k = (n+1)/2$

- S'il y a $Deg(G)$ fautes d'omissions par ronde, k-Accord est impossible pour $k > n/2$.
- S'il y a $Deg(G)$ fautes d'additions ou corruptions par ronde, k-Accord est impossible pour $k > n/2$.
- S'il y a $Deg(G)/2$ fautes d'omissions, additions ou corruptions par ronde, k-Accord est impossible pour $k > n/2$.

Algorithme pour Unanimité

1. Fautes d'Omissions :
   - Possible de tolérer $f = c - 1$ fautes, si G est c-arête-connexe.
   - Algorithme : Pour $c(n-1)$ rondes, envoyer v_i à tous les voisins ;

Algorithme pour Unanimité

1. Fautes d'Omissions :
   - Possible de tolérer $f = c - 1$ fautes, si G est c-arête-connexe.
   - Algorithme : Pour $c(n-1)$ rondes, envoyer v_i à tous les voisins ;
2. Additions : Possible de tolérer f fautes, $\forall f$
   - Envoyer un message dans chaque ronde.
3. Corruptions : Possible de tolérer f fautes, $\forall f$
   - Si $v_i = 0$ envoyer 0, sinon envoyer rien.

Additions et Corruptions

Possible de tolérer $f = c - 1$ fautes, si G est c-arête-connexe.

Additions et Corruptions

Possible de tolérer $f = c - 1$ fautes, si G est c-arête-connexe.

- Si $v_i = 0$ envoyer 0 dans rondes $2j$ (pair).
- Si $v_i = 1$ envoyer 1 dans rondes $2j + 1$ (impair).

Additions et Corruptions

Possible de tolérer $f = c - 1$ fautes, si G est c-arête-connexe.

- Si $v_i = 0$ envoyer 0 dans rondes $2j$ (pair).
- Si $v_i = 1$ envoyer 1 dans rondes $2j + 1$ (impair).

- Pour envoyer un message de x à y, on transmet sur chaque chemin entre x et y.
- On répète pour $(c - 1)$ rondes ; chaque ronde a l étapes.
  (l est taille le plus long chemin entre x et y)

Auto-stabilisation

- Comment tolérer des corruptions transitoires de mémoires ?
  (Quand l'état de processus peut être modifié par un adversaire, au cours d'une exécution)

Self Stabilizing Algorithms
(Algorithmes Auto-stabilisant)

Technique de Stabilisation

- Approche optimiste : Supposons qu'il existe $t > 0$, telle que après temps t, aucune fautes ne se produisent.
- Trouver un algorithme qui se stabilise quand il n' y a plus de fautes.

Définition (Self-Stabilizing Algorithm (SS))

Pour un ensemble bien défini de configurations légitimes, L, un algorithme stabilisant satisfait :

- *Convergence : l'exécution atteint une configuration $\in L$*
- *Si une configuration $\in L$ => le prochaine configuration $\in L$*
- *Correction : Chaque configuration légitime est correct.*

Stabilisation : Exemple

La Problème : Orienter un anneau

- L'anneau est bidirectionnel.
- Chaque P_i doit marquer ses 2 arêtes comme 'Pré' & 'Succ' (pour prédécesseur & successeur) telle que l'anneau devient orienté.

Configuration Légitime

- Si l'arête (p, q) est marqué **Pré** cote p => l'arête est marqué **Succ** cote q

Stabilisation : Exemple

Algorithme : Orienter un anneau

- Si l'arête (p, q) est marqué **Succ** à deux côtés, envoyer un jeton à q.
- Si p est d'accord avec q sur (p, q), il devient passif.
- Si reçu jeton de Pré, l'envoyer à Succ.
- Si reçu jeton de Succ, échanger Pre/Succ et envoyer le jeton.
- Si reçu jeton de deux côtés, détruire les jetons.

Propriétés : Finalement tous les jetons circulent dans le même sens et on arrive à un configuration légitime.

NB. Il n'y a pas de terminaison pour un algorithme stabilisant.

Algorithme Stabilisant : ELECTION

- Construire un arbre couvrant raciné au sommet qui a l'UID maximale.
- Chaque processus p a les variables suivantes :
  - $Root_p$: Racine de l'arbre couvrant.
  - $Dist_p$: Distance de la racine.
  - Par_p : Parent de p.

Algorithme Stabilisant : ELECTION

- Construire un arbre couvrant raciné au sommet qui a l'UID maximale.
- Chaque processus p a les variables suivantes :
  - $Root_p$: Racine de l'arbre couvrant.
  - $Dist_p$: Distance de la racine.
  - Par_p : Parent de p.

Configuration Légitime :

- $(Root_p = Par_p = p\,;\, Dist_p = 0)$ ou $(\exists q$ t.q. $Par_p = q)$
- Si $(Par_p = q) => Dist_p = Dist_q + 1$
- Pour chaque voisin q, $Root_p \geq Root_q$

Algorithme Stabilisant : ELECTION

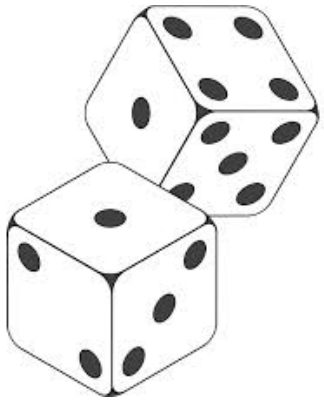
Configuration Légitime :

1. $(Root_p = Par_p = p\,;\, Dist_p = 0)$ ou $(\exists q$ t.q. $Par_p = q)$
2. Si $(Par_p = q) => Dist_p = Dist_q + 1$
3. Pour chaque voisin q, $Root_p \geq Root_q$

Algorithme :

- Si pas légitime, devenir racine (t.q. (1) est vrai) et envoyer requête au voisin q avec le plus grand $Root_q$.
- La requête est transmis vers la racine de q.
- Quand la requête est acceptée, p va devenir un enfant de q.

Randomisation



Algorithmes Randomisés

pour le calcul distribué

Algorithmes Randomisé

- A chaque étape i, le processus peut obtenir un nombre aléatoire x_i. La prochaine action dépend de la valeur x_i.
- Technique puissante : peut être utilisée pour surmonter l'impossibilité (e.g. Election anonyme)
- Attention : Un algorithme randomisé ne réussit pas toujours.
- Il y a une probabilité de réussite $p \in (0, 1)$. Il faut que $p > 1/2$.

Algorithmes Randomisé : Exemple-1

Théorème

Election (déterministe) est impossible dans les réseaux anonymes. [Mais il existe un algorithme randomisé]

Algorithme : (Pour Graphe Complet) Dans chaque ronde,

- Choisir $v = 0$ avec une probabilité $1/n$;
- Choisir $v = 1$ avec probabilité $1 - 1/n$;
- Si $v = 0$, envoyer v à tous ;
- Si un seul processus a $v = 0$, il est Elu.

Algorithmes Randomisé : Exemple-1

Théorème

Election (déterministe) est impossible dans les réseaux anonymes. [Mais il existe un algorithme randomisé]

Algorithme : (Pour Graphe Complet) Dans chaque ronde,

- Choisir $v = 0$ avec une probabilité $1/n$;
- Choisir $v = 1$ avec probabilité $1 - 1/n$;
- Si $v = 0$, envoyer v à tous ;
- Si un seul processus a $v = 0$, il est Elu.

Probabilité du succès $= n \cdot 1/n \cdot (1 - 1/n)^{n-1} \approx 1/e$
Donc, l'algorithme va terminer après ≈ 2.72 rondes.

Algorithmes Randomisé : Exemple-2

Théorème (FLP'85)

Le consensus (asynchrone) est impossible s'il y a une défaillance de type crash.

- Il existe un algorithme randomisé pour consensus avec f fautes de type *Crash*, $f < n/2$

Algorithmes Randomisé : Consensus

Algorithme f-tolérant ($f < n/2$) :

- $Pref = v_i$
- Dans chaque ronde r,
  - Envoyer (VOTE, r, $Pref$) à tous ;
  - Attendre pour $(n - f - 1)$ messages ;
  - Si reçu $n - f$ (VOTE, r, x), $Pref = x$;
  - Sinon $Pref$ =?
  - Envoyer (CONFIRM, r, $Pref$) ;
  - Attendre pour $(n - f - 1)$ messages ;
  - Si tous dit (CONFIRM, r, x), décider x ;
  - Sinon $Pref$ = aléatoire$(0, 1)$;

Données Distribués

Données Distribués

Base de données distribués

Données distribués :

- Un ensemble D de valeurs partagées entre les processus d'un réseau
- Chaque processus p_i a une sous-ensemble $D_i \subset D$
- Comment trier cette ensemble ?
- Minimiser la communication entre les processus.
- Algorithme Distribué de Tri (*Distributed Sorting*)

Algorithme Distribué de Tri

Si le réseau est une ligne de processus.

- Dans ronde $(2j+1)$, redistribuer ensemble D_{2i-1} avec D_{2i}, $\forall i$
- Dans ronde $(2j)$, redistribuer ensemble D_{2i} avec D_{2i+1}, $\forall i$
- Répéter jusqu'à qu'aucun échange ne soient possibles.

Complexité : $O(N \cdot n)$ messages.
$N = |D|$

Algorithme Distribué de Tri

- Si le réseau est un graphe complet, est-il possible d'avoir un algorithme plus efficace ?

Algorithme Distribué de Tri

- Si le réseau est un graphe complet, est-il possible d'avoir un algorithme plus efficace ?
- Algorithme de Tri par Selection Distribué
  - Trouver les éléments $d_1, d_2, \ldots d_{n-1} \in D$, t.q. $\text{Rang}(d_i) = i \cdot N/n$;
  - Envoyer chaque $v \leq d_1$ à P_1 ;
  - Envoyer chaque v, $d_{i-1} < v \leq d_i$ à P_i, $\forall i > 1$;
  - Envoyer chaque $v > d_{n-1}$ à P_n ;
- Complexité : $O(N)$ + coût de trouver les d_i.

Donnée Distribué

- Etant donné un ensemble D de valeurs partagées entre les processus d'un réseau, comment trouver le rang d'un élément $x \in D$?

Donnée Distribué

- Etant donné un ensemble D de valeurs partagées entre les processus d'un réseau, comment trouver le rang d'un élément $x \in D$?
- Supposons qu'il existe un arbre couvrant ; La racine va diffuser la valeur x.
- Chaque feuille va calculer $n_{i,x} = |\{y \in D_i : y < x\}|$;
- Effectuer une saturation pour compter $\sum_i n_{i,x} = Rang(x) - 1$;

Donnée Distribué

- Etant donné un ensemble D de valeurs partagées entre les processus d'un réseau, comment trouver le rang d'un élément $x \in D$?
- Supposons qu'il existe un arbre couvrant ; La racine va diffuser la valeur x.
- Chaque feuille va calculer $n_{i,x} = |\{y \in D_i : y < x\}|$;
- Effectuer une saturation pour compter $\sum_i n_{i,x} = Rang(x) - 1$;
- Si on peut calculer le rang de tous les éléments, on peut trouver l'élément qui a le rang k. Existe-il un algorithme plus efficace ?