Algorithmique Distribuée
Snapshots

Laurent PHILIPPE

Master 2 Informatique
Université de Franche-Comté
UFR des Sciences et Techniques

2016/2017

Sommaire

Introduction

Généralités

- Les algorithmes et propriétés des snapshot servent à analyser les propriétés d'une exécution, comme la terminaison,
- Travail difficile : datation, fautes, etc.
- Servent à calculer et mémoriser une unique vue traduisant une configuration du système,
- Permettent donc de faire une analyse *off-line* de l'exécution plutôt que d'observer des processus dont l'état varie,
- Utilisés en tolérance, debug, etc.

Introduction

Rappels

- Un calcul distribué peut être modélisé comme un système de transitions défini par un triplet $S = (\mathcal{C}, \rightarrow, \mathcal{I})$, où :
  - $\mathcal{C}$ est un ensemble de configurations,
  - $\rightarrow$ est une relation binaire sur C,
  - $\mathcal{I}$ est un sous-ensemble de configurations intiales, issues de C
- On note $\gamma \rightarrow \delta$ la transition de la configuration γ à la configuration δ,
- Une exécution du système S est une séquence maximale $\mathcal{E} = (\gamma_0, \gamma_1, \gamma_2, ...)$ où $\gamma_0 \in \mathcal{I}$ et $\forall i \geq 0, \gamma_i \rightarrow \gamma_{i+1}$
- Une configuration terminale γ est telle $\nexists \delta$ tel que $\gamma \rightarrow \delta$,
- Une configuration δ est accessible à partir d'une configuration γ s'il existe une séquence $\mathcal{E} = (\gamma_0, \gamma_1, \gamma_2, ...\gamma_k)$ où $\gamma = \gamma_0$, $\delta = \gamma_k$ et $\gamma_i \rightarrow \gamma_{i+1} \forall 0 \leq i \leq k$, cette relation est notée : $\gamma \rightsquigarrow \delta$.

Introduction

Définition d'une propriété stable

- Une propriété de configuration P est stable si, elle est conservée à partir d'une configuration γ pour toutes les configurations δ accessibles à partir de γ :

$$P(\gamma) \wedge \gamma \rightsquigarrow \delta \Rightarrow P(\delta)$$

- C'est le cas de la terminaison, des deadlocks, de la perte de jeton, etc.

Introduction

Contexte

- Soit ,$\mathcal{C}$ un calcul distribué utilisant un ensemble $\mathbb{P}$ de processus,
- L'ensemble des évènements est noté E_v,
- Les messages sont reçus dans un temps fini,
- Le réseau est fortement connecté,
- Le calcul local d'un processus p est composé d'une séquence $c_p^0, c_p^1, c_p^2, ...$ d'états du processus, où c_p^0 est l'état initial.
- La transition du l'état c_p^{i-1} à l'état c_p^i du processus p est marquée par l'occurence de l'évènement e_p^i
- On définit $E_v = \bigcup_{p \in \mathbb{P}} \{e_p^0, e_p^1, e_p^2, ...\}$

Introduction

Contexte

- Un ordre causal local est définit sur les évènements internes du processus p : $e_p^i \preceq_p e_p^j \Leftrightarrow i \leq j$
- Chacun des évènements peut-être un envoi, une réception ou un évènement interne,
- L'historique complète des communications d'un processus est mémorisée dans son état. Donc, si il existe un lien de p à q alors :
  - l'état c_p^i du processus p contient la liste $sent_{pq}^i$ des messages envoyés de p à q parmis les évènements e_p^0 à e_p^i,
  - l'état c_q^i du processus q contient la liste $rcvd_{pq}^i$ des messages que q à reçu de p parmis les évènements e_p^0 à e_p^i,

Sommaire

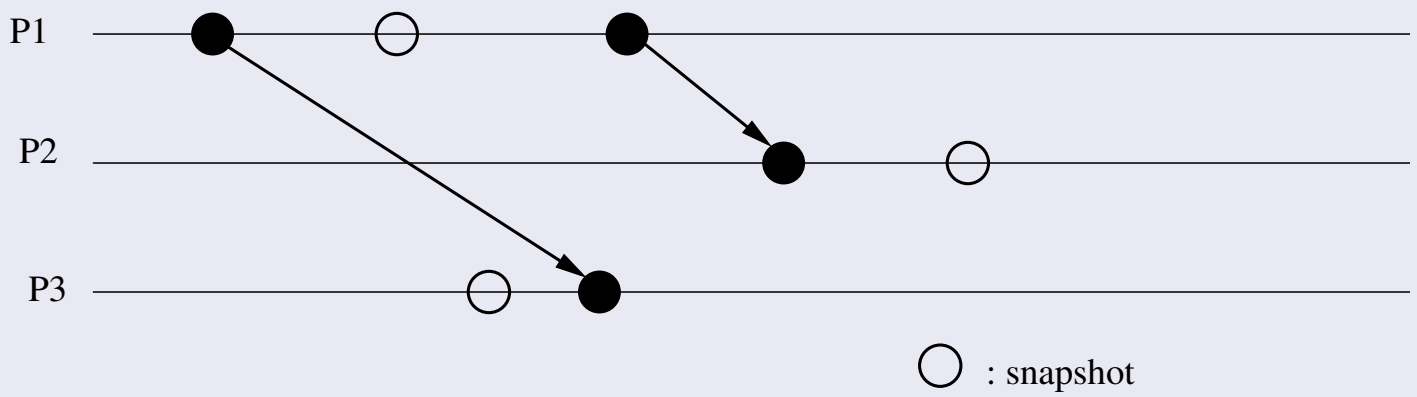
Algorithmes de snapshot

Définition d'un snapshot

- Le but d'un algorithme de snapshot est de construire explicitement une configuration globale S^* (snapshot courant) des processus $p \in \mathbb{P}$,
- Chaque processus réalise donc un snapshot local c_p^* entre e_p^i et e_p^{i+1},
- Comme les configurations locales incluent l'historique des communications, l'état d'un lien pq est l'ensemble des messages envoyés par p et non reçus pas q : $sent_{pq}^* \setminus rcvd_{pq}^*$,
- Des anomalies peuvent se produire si $rcvd_{pq}^*$ n'est pas un sous-ensemble de $sent_{pq}^*$,

Algorithme de snapshot

Exemple d'anomalie



Définitions

- Le snapshot S^* est faisable si, pour chaque paire de processus p et q, on a $rcvd^*_{pq} \subseteq sent^*_{pq}$.
- Un évènement est *preshot* si il a lieu avant le snapshot et *postshot* si il a lieu après.

Les coupures

Définitions

- Une coupure L (*cut*) de E_v est un ensemble $L \subseteq E_v$ tel que : $e \in L \wedge e' \preceq_p e \Rightarrow e' \in L$
- La coupure L_2 est dite postérieure à L_1 si $L_1 \subseteq L_2$,
- L'ensemble des évènements preshot S^* est donc une coupure,
- Attention, dans une coupure, la précédence est définie sur p. On définit alors une coupure consistente par : $e \in L \wedge e' \preceq e \Rightarrow e' \in L$

Snapshots et coupures

Théoreme

Soit S^* un snapshot et L la coupure induite pas S^*, alors les trois affirmations suivantes sont équivalentes :

- S^* est faisable,
- L est une coupure consistente,
- S^* a du sens (ne possède pas de contradiction).

Algorithmes de snapshot

Généralités

- Idéalement, on souhaite que l'algorithme de snapshot enregistre une configuration qui arrive au cours de l'exécution,
- Malheureusement, comme nous n'avons pas d'horloge globale cela n'est pas possible et il faut donc se contenter des configurations possibles,
- La faisabilité d'un snapshot impose des relations entre les snapshots locaux alors que le fait d'avoir du sens est une propriété globale sur le snapshot,
- Le théorème précédent permet d'établir qu'il suffit de coordonner les snapshots locaux pour garantir que le snapshot obtenu est faisable.

Algorithmes de snapshot

Propriétés

Les propriétés suivantes sont donc suffisantes pour définir un algorithme de snapshot :

- Chaque processus doit réaliser des snapshots locaux,
- Aucun message émis après le snapshot n'est reçu dans un évènement qui précède le snapshot.

Il faut donc assurer que les processus prennent leurs snaphosts avant de recevoir des messages émis après le snapshot.

Rappel

On appelle *messages de contrôle* les messages qui sont émis dans le cadre de l'algorithme et *messages de base* les messages qui sont émis dans le cadre du calcul.

Sommaire

Algorithme de Chandy-Lamport

Généralités

- Hypothèses : les liens de communication fifo et messages reçus au bout d'un temps fini,
- Les processus s'informent mutuellement de la construction du snapshot en envoyant des messages $\langle mkr \rangle$, appelé marqueur, sur tous les liens,
- Chaque processus envoie une seule fois le message marqueur, lorsqu'il réalise son snapshot local,
- A la réception d'un message $\langle mkr \rangle$, un processus qui n'a pas fait son snapshot le fait et propage le message $\langle mkr \rangle$,
- L'algorithme est exécuté concurrement avec l'exécution.

Algorithme de Chandy-Lamport

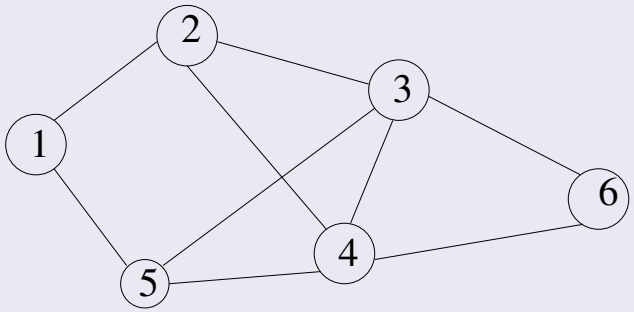
var $taken_p$ boolean **init** false

Pour commencer l'algorithme :
début
    Record local state
    $taken_p = \text{true}$
    pour $q \in Neigh_p$ **faire**
        send $\langle mkr \rangle$ to q

If a marker has arrived
début
    receive $\langle mkr \rangle$
    si $\neg taken_p$ **alors**
        Record local state
        $taken_p = \text{true}$
        pour $q \in Neigh_p$ **faire**
            send $\langle mkr \rangle$ to q

Algorithme de Chandy-Lamport

Exercice



Le processus 1 initie le snapshot. Juste avant de recevoir le message $\langle mkr \rangle$, le processus 2 envoie un message au processus 3, puis le processus 3 répond au processus 2 de manière à ce qu'il croise le marqueur. Après avoir reçu le marqueur, le processus 4 envoie un message au processus 6.

Algorithme de Chandy-Lamport

Lemme

Si au moins un processus initie l'algorithme alors tous les processus réalisent un snapshot au bout d'un temps fini.

Théorème

L'algorithme produit un snapshot réalisable au bout d'un temps fini après son initialisation par un processus.

Exercice

- Démontrer le lemme puis le théorème précédents
- Combien faut-il de messages pour réaliser le snapshot ?
- Quelle est sa complexité en temps (retransmissions de messages) ?

Sommaire

Algorithme de Lay et Yang

Généralités

- Ne fait la supposition que les liens sont fifo,
- Les messages $\langle mkr \rangle$ ne peuvent garantir la séparation entre les messages préshot et postshot,
- Une information $taken_p$ est ajoutée aux messages de l'exécution pour les définir comme préshot ou postshot. Ces messages sont notés $\langle mes, c \rangle$
- L'algorithme contrôle les messages reçus et enregistre l'état local avant la réception du premier message postshot,
- Pas de message de contrôle, interfère avec les échanges de l'exécution de base.

Algorithme de Lay-Yang

var $taken_p$ boolean **init** false

Pour commencer l'algorithme :
début
    Record local state
    $taken_p = \text{true}$

Pour envoyer un message :
début
    send $\langle mess, taken_p \rangle$

A la réceptoin d'un message $\langle mess, c \rangle$
début
    si $c \wedge \neg taken_p$ **alors**
        Record local state
        $taken_p = \text{true}$
        Réaliser la réception du calcul

Algorithme de Lay et Yang

Propriétés

- L'algorithme ne garantit pas que chaque processus enregistre son état : si aucun processus n'envoie de message à p et si celui-ci n'est pas l'initiateur alors celui-ci ne réalise jamais de snapshot local,
- L'algorithme peut donc terminer avec un snapshot incomplet,
- Dépendant de l'exécution de base : si tous les processus sont atteints au bout d'un temps fini alors l'algorithme réalise un snapshot complet.

Exercice

- Proposer une solution pour compléter l'algorithme.
- Quel problème pose la prise régulière de snapshots ?

Algorithme de Lai et Yang

Théorème

L'algorithme de Lai et Yang ne génère que des snapshots réalisables.

La preuve est basée sur le fait qu'à la réception du premier message où $taken_p$ est vrai le snapshot est pris avant de délivrer le message donc bien dans un état préshot.

Propriété

Si on utilise ni la propriété de fifo ni le numérotage des messages alors il faut interrompre l'exécution de base pour générer un snapshot.

Sommaire

Utilisation des algorithmes de snapshot

Calcul de l'état des liens de communication

- Nous avons supposé que l'état de chaque processus contient l'historique locale et de communication
- Ceci permet de calculer l'état des liens des processus connectés
- Pour la plupart des algorithmes cela s'avère trop coûteux, on peut se limiter à un sous-ensemble d'informations : savoir si les liens sont vides ou non
- Dans un snapshot réalisable, $sent^*_{pq} \setminus rcvd^*_{pq}$ est égal à l'ensemble des messages envoyé par p dans un évènement preshot et reçu par q dans un évènement postshot

Utilisation des algorithmes de snapshot

Déctection de propriétés stables

- Terminaison : si γ est une configuration terminale et $\gamma \rightsquigarrow \delta$ alors $\gamma = \delta$. Donc le problème de terminaison peut-être résolu en calculant un snapshot et en y cherchant des processus actifs et les messages de base. Les algorithmes vu dans le chapitre terminaison sont cependant plus efficaces
- Deadlocks, perte de jeton, rammase-miettes, ...
- Evaluation de fonctions monotones f :
  - calcul de convergence, temps distribué, ...
  - f est monotone si $\gamma \rightsquigarrow \delta \Rightarrow$
  - Si γ^* est un snapshot et $f^* = f(\gamma^*)$ est évaluée, alors $f(\delta) \geq f^*$ pour toute configuration δ postérieure

Utilisation des algorithmes de snapshot

Déctection de deadlock (algorithme général)

- Deadlock : attente de message, concurrence d'accès, transactions, ...
- $\mathbb{P}$: ensemble de processus qui exécutent un algorithme de base
- Etat bloqué (*blocked*) traduit l'attente (message, accès, données, ...)
- Lorsque p devient bloqué (action B_p) il envoie des messages de requête à un ensemble de processus noté $Reqs_p$
- Pour redevenir actif (action F_p), p doit recevoir des accords (*grants*) de processus dans $Reqs_p$, mais pas forcément de tous. Le sous-ensemble de $Reqs_p$ suffisant pour rendre p actif, est constitué des processus qui satisfont le prédicat $Free_p$
- Les requêtes reçues par p (action R_p) sont mémorisées dans $Pend_p$ et le processus peut y répondre (action G_p) lorsque p est ou redevient actif.
- p attache un numéro à chaque requête envoyée et ce numéro est attaché au message d'accord, ce qui permet à p d'ignorer les messages d'accord obsolètes

Algorithme basique de détection de deadlock (1/2)

var $state_p$: (active, blocked) **init** active
v_p : entier **init** 0 /* Numéro de requête */

$Reqs_p$: ens. de processus ; /* Ensemble de requêtes */

$Grant_p$: ens. de processus ; /* Grants reçus */

$Pend_p$: ens. de requêtes **init** $\emptyset$; /* Requêtes en attente */

$Free_p$: prédicat ; /* Contrainte de libération */

B$_p$: $state_p = active$ **début**
determine $Reqs_p$ et $Free_p$; $v_p = v_p + 1$
pour $q \in Reqs_p$ **faire**
send $\langle req, v_p \rangle$ à p

$Grant_p = \emptyset$; $state_p = blocked$

Algorithme basique de détection de deadlock (2/2)

G$_p$: $state_p = active \wedge (q, v) \in Pend_p$ **début**
  envoie $\langle grant, v \rangle$ à q ; $Pend_p \leftarrow Pend_p \backslash \{(q, v)\}$

F$_p$: Un message $\langle grant, v \rangle$ arrive de q
début
  receive $\langle grant, v \rangle$ de q
  si $state_p = blocked \wedge v = v_p$ **alors**
    $Grant_p = Grant_p \cup q$
    si $Free_p(Grant_p)$ **alors**
      $state_p = active$

Algorithme basique de détection de deadlock

Exemples

- Si p a besoin d'un accord à chaque requête, $Free_p$ est vrai si l'ensemble $Grant_p = Reqs_p$
- Si p a besoin d'au moins une réponse, $Free_p$ est vrai si l'ensemble $Grant_p$ est non vide
- Si p a besoin d'un espace de stockage de 10 et il envoie une requête aux serveurs A, B, C et D. A dispose de 9, B de 7, C de 3 et D de 1. Il nous faut donc A et un autre ou B, C et D.
  - Prédicat $Free_p(S) : (\#S \geq 2 \wedge A \in S) \vee (\#S \geq 3)$