Cours 1

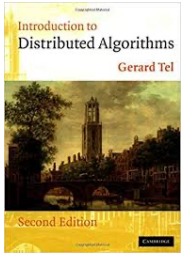
Introduction et préliminaires

Algorithmique répartie

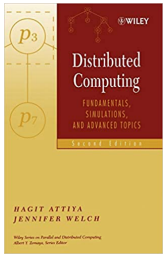
Plan du cours (sous réserve)

- Introduction et présentation générale
- Partie I: algorithmes classiques
  - Construire un état global à partir d'états locaux
    - Exemple de la terminaison distribuée
      - Algorithmes de vagues et de traversée, probe-echo, gossip etc…
    - Snapshot
  - Election de leader
    - Réseaux anonymes, élection probabiliste
  - Introduction à l'auto-stabilisation: protocole du bit alterné
- Partie II: algorithmique tolérante aux pannes
  - Défaillances des liens de communications: attaque coordonnée
  - Modèle synchrone avec défaillances de processus
    - Consensus
    - Accord byzantin
    - « commit » 2-phase et 3-phase commit
  - Modèle asynchrone avec défaillances
    - Diffusion fiable, diffusion atomique
    - Impossibilité du consensus et ses conséquences

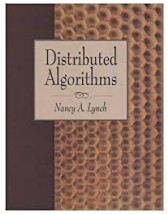
Bibliographie



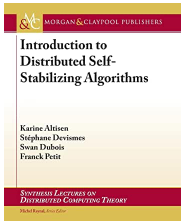
Gérard Tel
Introduction to Distributed Algorithms
Cambridge University Press



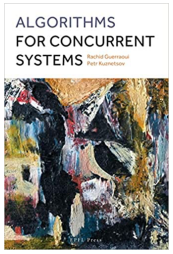
Hagit Attila, Jennifer Welch
Distributed computing: Fundamentals, Simulations, and Advanced Topics
Wiley–Blackwell



Nancy Lynch
Distributed Algorithms
Morgan Kaufmann Publishers

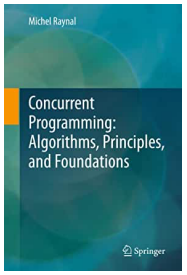


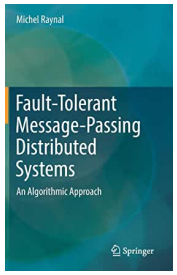
de Karine Altisen, Stephane Devismes, et al.
Introduction to Distributed Self-stabilizing Algorithms
Morgan & Claypool Publishers

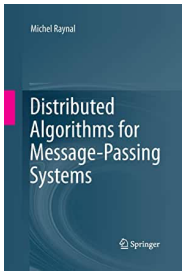


R. Guerraoui, P. Kuznetsov
Algorithms for concurrent systems
EPFL Press

Et…







Michel Raynal
Springer

Introduction

Algorithmique répartie?

- Répartition du calcul
  - Multicore
  - Grilles de calcul
  - LAN,
  - Réseaux de capteur (ressources en calcul limitées)
  - Internet WAN, Internet, Internet des objets (quelques millions, milliards)
- Répartition spatiale
  - D'une même « puce » -avionique spatial- jusqu'au monde entier (univers ?)
  - Mobilité
- Répartition des données (et taille des données)
  - Shared memory
  - Serveurs
  - P2P, cloud…

Modèles…

grandes classes de modèles

- Synchrone / asynchrone
- Mémoire partagée /échange de message

(Dans ce cours)

Modèles

Processus (processeurs) (par exemple avec la puissance d'une machine de Turing avec sa propre mémoire locale)

- Synchrone
- Asynchrone

+

Interaction entre les processus

- Messages (envoi-réception)
- Partage de mémoire - partage d'objets
- (Graphe de communication)

Modèle: processus

Processus (a priori séquentiels)

- Les processus ont des identités (connues):
  - $\Pi = \{p_1, \ldots, p_n\}$
  - (Variante: anonymat)
- Des pas de calculs locaux indépendants de chaque processus
  - En général, puissance des machines de Turing
  - Mais aussi, Réseaux de capteurs: limite sur la puissance de calcul
- Asynchrone : un pas de calcul prend un temps arbitraire (imprévisible)
- Synchrone: un pas de calcul prend (au plus?) un temps δ
- Interaction explicite avec les autres processus envoi-réception de messages
  - (Variante partage d'objets: accès à un objet partagé.
    - Exemple R.read(), R.write(v), O.test&set(x) etc…
  -)

Processus…

- En général, on suppose que les processus exécutent (séquentiellement) des actions « atomiques » (événements).
  - E_p est l'ensemble des événements de p
  - (En général: $E_p \cap E_q$ si $p \neq q$)
- Le comportement d'un processus (dans une exécution) est une séquence d'événements (mot fini ou infini)
- On ne suppose pas (en général) de « simultanéité » et une exécution sera une séquence d'événements (de divers processus)
- Si $e = a_1, a_2, \ldots, a_m, \ldots$ est une exécution et soit π_p est la projection sur E_p (=effacement de tous les événements sauf ceux de p) alors pour cette exécution: le comportement de p pour e est $\pi_p(e)$
- (Une exécution est le résultat d'un « interleaving » des comportements des processus)

Processus: défaillances (Pour la partie II)

Défaillances des processus:

- Crash (panne franche) : le processus arrête d'exécuter son code.
- Omission-émission: un processus « oublie » d'envoyer un message
- Omission-réception: un processus « oublie » de recevoir un message
- Omission émission-réception; un processus « oublie » d'émettre ou de recevoir un message
- Byzantin: un processus fait n'importe quoi. Il peut envoyer (ou ne pas envoyer) n'importe quel message et agir comme un adversaire du système.
  - (En cryptographie et concernant la sécurité on peut avoir des restrictions sur la puissance de calcul d'un adversaire)
- (On considère en général que tous ces modèles incluent les crashs: omission réception = omission + crash)

Processus: défaillances

- Processus <u>correct</u> un processus qui exécute (pour toujours) correctement son code!
  - (En général on considère des comportements infinis des processus et à l'instant t on ne peut pas dire qu'un processus est correct!)
- Processus défaillant = un processus non correct
  - n processus parmi lesquels au plus t peuvent avoir des défaillances (on a toujours au moins $n - t$ processus qui exécutent correctement leur code) : algorithme t-résilient

(Attention les pannes par omission sont de pannes des processus et ne sont pas des pannes des liens de communications)

Qu'est ce que cela signifie?

Défaillance: un exemple

Généraux byzantins:

- Un général G est à la tête d'un armée de capitaines. Tout le monde peut communiquer avec tout le monde. (n est le nombre de processus -général + capitaine)
- Il donne un ordre: attaque ou non attaque (un booléen)
- Les capitaines doivent se mettre d'accord (décider) sur cet ordre.
  - Les capitaines et/ou le général peuvent être des traîtres (t est le nombre maximal de traîtres)
  - On veut un algorithme qui assure:
    - Tous les capitaines honnêtes doivent décider et se mettre d'accord sur une seule valeur (attaque ou non)
    - Si le général est honnête, les capitaines honnêtes doivent se mettre d'accord sur l'ordre du général
    - Tous les capitaines honnêtes doivent décider

on montrera qu'il faut que $n > 3t$ (t étant le nombre maximal de processus défaillants) pour pouvoir un algorithme.

Exercice: essayer pour n=3, n=4

Modèles: Communication…

Communication par messages:

- Point à point:
  - Processus p_i : send m to p_j et dans p_j receive m
- (Variante broadcast(m) envoyer m à tous les processus)
- Réseau de communication complet (tout processus peut envoyer un message à n'importe quel processus)
- (Variante graphe de communication particulier: anneau, arbre…)
- Sans perte: m sera reçu
- Asynchrone: délai inconnu entre émission et réception de m
- Synchrone: m sera reçu au plus tard dans Δ unités de temps

Liens de communication

Ordre des réceptions (lien de p à q)

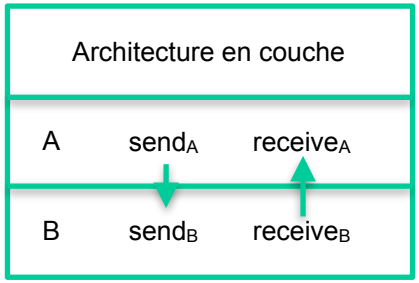
- Sans ordre:
  - Lien: $L_{p,q}$ est un ensemble de messages
    - Send (m) : put m in $L_{p,q}$
    - m:=receive(): m=get() in $L_{p,q}$
    - ($L_{p,q}$ est-il fiable? Équitable? Juste?)
- FIFO: les messages sont reçus dans l'ordre d'émission
  - Lien: $L_{p,q}$ est une File (FIFO)
    - Send (m) : enfiler m in $L_{p,q}$
    - m:=receive(): m=défiler ($L_{p,q}$)
    - ($L_{p,q}$ est-il fiable? Équitable? Juste?)
- Exercice: comment émuler FIFO à partir d'un canal sans ordre?

Remarques:

Émuler canal avec propriété A à partir d'un canal avec propriété B?

B: $send_B(m)$ et $m = receive_B()$

A: $send_A(m)$ et $m = receive_A()$



Implémenter $send_A(m)$:

- avec un code pouvant contenir des $send_B$ et des $receive_B$

Implémenter $receive_A()$:

- avec un code pouvant contenir des $send_B$ et des $receive_B$

De façon à ce que $send_A(m)$ et $receive_A()$ assurent les propriétés de A si les $send_B(m)$ et $receive_B()$ ont les propriétés de B

Causalité

E: ensemble des événements: événements locaux, et événements de communication (émission de m, réception de m). E_p est l'ensemble des événements de p et $E = \cup_{p \in \Pi} E_p$

Relation de causalité (ordre de Lamport)

$a \preceq b$ si et seulement si:

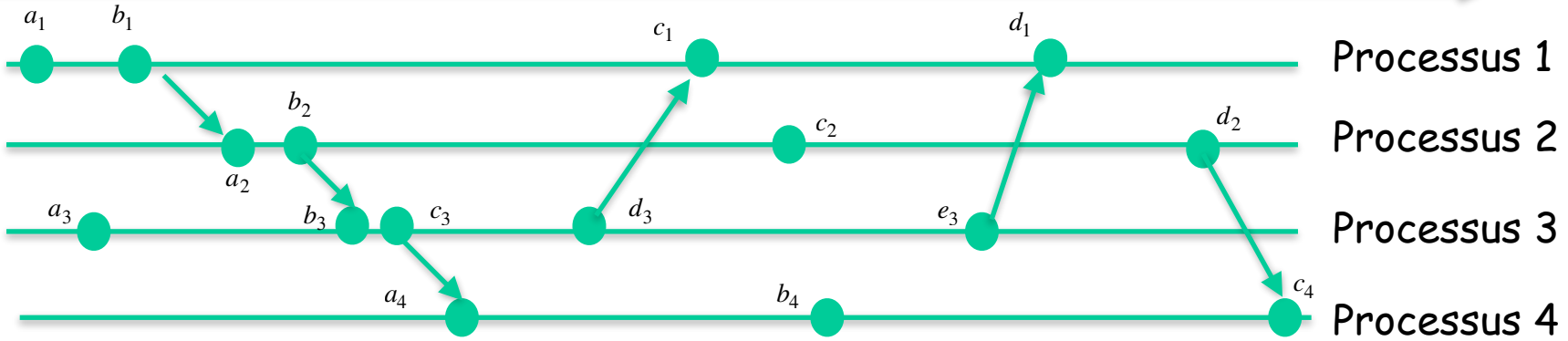
- $a = b$ (réflexivité)
- a et b sont sur le même processus a est séquentiellement avant b (séquentialité)
- a est l'émission du message m et b la réception de ce message (communications)
- Il existe c tel que $a \preceq c \wedge c \preceq b$ (fermeture transitive)

$\preceq$ est une relation d'ordre (partielle) (prouvez-le!)

Si a et b sont incomparables ($\neg(a \preceq b) \wedge \neg(b \preceq a)$ a et b sont concurrents

Si $a \prec b$ alors quelque soit l'exécution l'événement a se produit (pour le temps) avant l'évènement b: a peut être une cause de b mais b ne peut pas être une cause de a (on ne remonte pas le temps!)

Temps



$a_1 b_1 a_3 a_2 a_3 b_2 c_2 d_2 b_3 c_3 a_4 b_4 d_3 e_3 c_1 c_4 d_1$

$a_1 b_1 a_2 b_2 c_2 d_2 a_3 b_1 c_3 d_3 e_3 c_1 d_1 a_4 b_4 c_4$

...

$a_2 \preceq d_1$

a_4 et d_1 concurrents

Remarque: Si $a_1, \cdots, a_k$ une exécution et σ une permutation soit $a_{\sigma(1)}, \ldots a_{\sigma(k)}$ telle que $a_{\sigma(i)} \preceq a_{\sigma(j)} \Rightarrow i \leq j$ alors $a_{\sigma(1)}, \ldots a_{\sigma(k)}$ est aussi une exécution