Algorithmique Distribuée
Détection de la terminaison

Laurent PHILIPPE

Master 2 Informatique
Ingénierie des Systèmes et Logiciels

2019 / 2020

UNIVERSITÉ DE FRANCHE-COMTÉ

femto-st SCIENCES & TECHNOLOGIES

Sommaire

1. Introduction
   - Conditions de terminaison
   - Terminaison des processus
2. Terminaison pour le calcul sur arbres
3. Solutions utilisant un anneau

Introduction

Terminaison

- Systèmes distribués composés de processus indépendants
- Implémentent des algorithmes, non infinis, à échanges de messages
- **Problème** : savoir quand l'ensemble des processus a terminé (tous dans un état terminal ou ne progressent plus)
- Peut être généralisé à une étape d'un calcul
- Exemples :
  - fin d'exécution d'un algorithme parallèle (convergence)
  - fin d'un pas de temps SMA

Hypothèses de travail

Communication et processus

- Temps de communication finis mais non prévisibles
- Exécution d'un processus est une suite finie d'évènements
- Évènements internes ou externes
- Pas de panne des processus (machines)
- Messages arrivent intacts
- Messages ne sont pas dupliqués

Définitions

Configuration terminale

Fin de l'exécution d'un algorithme distribué : plus aucune progression n'est possible dans l'algorithme. Les processus peuvent être en attente de réception d'un message.

État terminal

Fin de l'exécution global du système/application. Il n'y a plus d'évènement activable

Détection de la terminaison

Algorithme de terminaison

Méthodes permettant de passer d'un état *configuration terminale* à un état *état terminal*. Basées sur le schéma suivant :

1. détection de la configuration terminale
2. diffusion de l'information à tous : algorithme de terminaison
3. collecte des terminaisons
4. passage dans un état terminal.
5. algorithme d'**annonce**.

Phases

- Algorithme de **base** : ajouté à l'application pour gérer l'état des processus,
- Algorithme de **contrôle** : ajouté pour gérer la terminaison,

Algorithme de base

État des processus

- Deux états possibles : $etat_p = (actif, passif)$
- Un processus est actif (resp. passif) si un événement interne ou une réception de message est (resp. n'est pas) accessible à son état c_p.
- Un processus p passif n'admet que des réceptions et peut à tout instant devenir actif à la réception d'un message.

Règles de changement d'état des processus

- Un processus actif devient passif uniquement suite à un événement interne.
- Un processus devient actif à la suite de la réception d'un message.

Algorithme de base

Définition générique

$\mathbf{S}_p$: $\{etat_p = actif\}$
/* p envoie un message */

début
  send $\langle$msg$\rangle$

$\mathbf{I}_p$: $\{etat_p = actif\}$
/* événement interne */

début
  $etat_p \leftarrow passif$

$\mathbf{R}_p$: $\{\langle$msg$\rangle$ arrive en $p\}$
/* p reçoit un message */

début
  receive $\langle$msg$\rangle$
  $etat_p \leftarrow actif$

Théorème de terminaison

Théorème

Théorème de terminaison

$$\textbf{\textit{term}} \Leftrightarrow \left\{ \begin{array}{ll} & (\forall p \in \mathbb{P} : etat_p = passif) \\ \wedge & (\forall p, q \in E : M_{pq} \text{ ne contient pas de messages } \langle \mathtt{msg} \rangle) \end{array} \right.$$

*Le prédicat **term** est vrai quelque soit la configuration où aucun évènement de l'exécution ne peut être déclenché.*

Théorème de terminaison

Preuve

1. Si tous les processus p sont passifs, alors aucun événement interne n'est possible. De plus, si aucun canal de réception ne contient de message, alors aucune instruction **receive** n'est possible. Par conséquent, aucun événement de base n'est applicable, l'état terminal est atteint.
2. Si certains processus sont actifs, une instruction **send** ou un événement interne est possible. De plus, si certains canaux d'entrée contiennent des messages, alors une instruction **receive** est possible. Par conséquent, le prédicat **term** ne peut être vrai. □

Conditions de détection d'un état terminal

Propriétés

L'algorithme de détection de la terminaison doit satisfaire aux conditions suivantes :

1. **Non interférence** : pas d'interférence avec l'exécution de l'algorithme de base,
2. **Vivacité** (Liveness) : si le prédicat *term* est **vrai** alors l'algorithme de terminaison est appelé au bout d'un nombre fini d'étapes,
3. **Sûreté** (Safety) : si l'algorithme de détection de terminaison est appelé, la configuration satisfait au prédicat *term*.

L'annonce de la terminaison

Initialisation de l'algorithme

- On peut montrer qu'au moins un message de contrôle doit être envoyé sur chaque lien pour garantir la terminaison.
- Lorsqu'une configuration terminale est détectée, on passe dans l'état terminal via « l'annonce ».
- chaque processus passe dans l'état terminal en exécutant proprement l'instruction « stop ».
- un message $\langle \text{stop} \rangle$ est diffusé à tous les processus :
  - Chaque processus relaie le message à ses voisins au plus une fois.
  - Quand un processus reçoit un message $\langle \text{stop} \rangle$ de tous ses voisins, il exécute la procédure **annonce stop** pour entrer dans un état terminal et arrêter son activité.

L'annonce de la terminaison

$\texttt{envoiStop}_p$: booléen initialisé à faux /* vrai si p a déjà envoyé stop */
$\texttt{recStop}_p$: entier initialisé à 0 /* nb de messages stop reçus par p */
V_p : ensemble des voisins du processus p
Règle **annonce()**
début
  si $\neg\texttt{envoiStop}_p$ **alors**
    $\texttt{envoiStop}_p \leftarrow$ vrai
    pour tous les $q \in V_p$ **faire** send $\langle$stop$\rangle$ à q

Règle stop : le message $\langle$stop$\rangle$ est reçu par p
début
  receive $\langle$stop$\rangle$
  $\texttt{recStop}_p \leftarrow \texttt{recStop}_p + 1$
  annonce()
  si $\texttt{recStop}_p = |V_p|$ **alors** stop()

Détection centralisée et détection décentralisée

Classes d'algorithmes

Les algorithmes peuvent être classés suivant leurs propriétés :

- les algorithmes centralisés : un seul processus initie l'algorithme de terminaison,
- algorithmes décentralisés : plusieurs processus initient l'algorithme de terminaison,
- Variations sur la centralisation : initiateur concentre tout ou simplement initie puis valide.

Sommaire

1. Introduction
2. Terminaison pour le calcul sur arbres
   - Algorithme de Dijkstra et Scholten
   - Algorithme de Shavit et Francez
3. Solutions utilisant un anneau

Terminaison pour le calcul d'arbres

Principes généraux

- Le graphe de calcul est en forme d'arbre et l'initiateur est la racine,
- L'algorithme de contrôle s'exécute en parallèle avec l'algorithme de calcul (base) mais sans interférence (messages de contrôle),
- On considère que les liens réseaux sont bi-directionnels (ils sont indépendants du graphe de l'application),
- Exemples : algorithme d'exploration ou "divide-and-conquer"

Algorithme de Dijkstra et Scholten

Principes de fonctionnement

- Solution centralisée : il y a un seul initiateur, pour l'algorithme de *base* et pour l'algorithme de *contrôle*,
- L'algorithme gère un arbre dynamique $T(V_T, E_T)$
  - Les sommets sont des processus actifs ou des messages en transit
  - Les arrêtes relient un sommet et celui qui lui a donné naissance (activation d'un processus suite à la réception d'un message ou le message lui-même)
  - l'arbre possède les propriétés suivantes :
    1. $T = \emptyset$ ou bien T est un arbre orienté de racine p_0.
    2. V_T contient tous les processus actifs et les messages en transit.
- p_0 appelle l'algorithme de l'`annonce()` de la terminaison dès que $p_0 \notin V_T$ ou $T = \emptyset$, ce qui est équivalent à **term** = `vrai`.

Algorithme de Dijkstra et Scholten

Ajout des éléments dans l'arbre

- Les évènements qui conduisent à créer un sommet sont :
  - p envoie le message $\langle \text{msg} \rangle$

  $\rightarrow$ le sommet $\langle \text{msg} \rangle$ est créé dans l'arbre, le père de $\langle \text{msg} \rangle$ est p.
  - si $p \notin T$, p devient actif à la réception d'un message de q

  $\rightarrow$ le sommet p est créé avec q le père de p.
- On représente un message avec l'émetteur par le couple $\langle \text{msg}, q \rangle$ si q est l'émetteur du message $\langle \text{msg} \rangle$.
- Les messages sont des feuilles alors que les processus sont soit des feuilles soit des sommets internes de T.
- Les sommets de l'arbre maintiennent le compte de leurs fils, soit des messages, soit des autres processus

Algorithme de Dijkstra et Scholten

Suppression des éléments dans l'arbre

- Les évènements qui conduisent à supprimer des éléments sont :
  - Acquittement d'un message, le message en transit était dans l'arbre. Il est supprimé dès lors qu'il est consommé par son destinataire : envoi d'un message de contrôle,
  - Passage d'un processus de l'état **actif** à l'état **passif** : envoi d'un message de contrôle,
- Chaque processus compte le nombre de messages et le nombre d'instances dans le sous-arbre dont il est la racine.
- L'arbre se résorbe en un nombre fini d'étapes après la terminaison.

Algorithme de Dijkstra et Scholten

Effacement d'un fils de p

- Par défaut on considère que le premier message crée le processus fils et les suivants sont nécessaires au calcul. Tous les messages ne conduisent donc pas à l'activation d'un processus, d'où les 2 cas suivants :
  1. un message pour prévenir l'initiateur de l'arrivée de son message lorsque le processus destinataire est déjà actif
  2. un message pour prévenir l'initiateur de l'arrêt d'un processus fils actif.
- ces deux cas conduisent à la même conclusion, la suppression d'un fils pour le sommet courant dans T.

$\rightarrow$ le message de contrôle $\langle \text{sig}, p \rangle$ permet d'informer le processus père p. Ce message est effacé dès sa réception dans p et le compteur des fils de p est décrémenté.

Algorithme de Dijkstra-Scholten

$etat_p$: $(actif, passif)$ init à $actif$ si $p = p_0$ sinon $passif$
sc_p : entier (nombre de fils dans T)
$pere_p \in \mathbb{P}$ init à p si $p = p_0$ sinon $nDef$

S$_p$: $\{etat_p = actif\}$
début
  `send` $\langle$msg,$p\rangle$
  $sc_p + +$

R$_p$: {le message $\langle$msg,$q\rangle$ arrive en p}
début
  `receive`$\langle$msg,$q\rangle$
  $etat_p \leftarrow actif$
  si $pere_p = nDef$ **alors** $pere_p \leftarrow q$
  sinon `send`$\langle$sig,$q\rangle$ à q

I$_p$: $\{etat_p = actif\}$
début
  $etat_p \leftarrow passif$
  si $(sc_p = 0)$ **alors**
    si $pere_p = p$ **alors** `annonce()`
    sinon `send` $\langle$sig,$pere_p\rangle$ à $pere_p$
    $pere_p \leftarrow nDef$

A$_p$: {le signal $\langle$sig,$p\rangle$ arrive en p}
début
  `receive`$\langle$sig,$p\rangle$
  $sc_p - -$
  si $(sc_p = 0) \wedge (etat_p = passif)$ **alors**
    si $pere_p = p$ **alors** `annonce()`
    sinon `send` $\langle$sig,$pere_p\rangle$ à $pere_p$
    $pere_p \leftarrow nDef$

Algorithme de Dijkstra-Scholten

Exercice

- Pour une exécution, un processus initiateur créé deux fils pour réaliser deux calculs,
- Pour exécuter le premier calcul, le premier fils crée deux petits fils auxquels il soumet successivement les deux messages de calcul sans que ceux-ci deviennent inactifs entre temps,
- Avant de recevoir tous les messages de ses fils, le premier processus fils devient inactif
- Donner la trace d'exécution de l'algorithme de Dijkstra-Scholten dans ce cas.

Algorithme de Dijkstra-Scholten

Arbre de l'application

Le graphe $T(V_T, E_T)$ de l'application distribuée de l'algorithme de Dijkstra-Scholten est plus formellement défini par :

$$\begin{cases} V_T & = & \{p : pere_p \neq \text{undef}\} \\ & & \cup \{\langle \text{msg}, p\rangle \text{ en transit}\} \\ & & \cup \{\langle \text{sig}, p\rangle \text{ en transit}\} \\ E_T & = & \{(p, pere_p) : pere_p \neq \text{undef} \wedge pere_p \neq p\} \\ & & \cup \{(\langle \text{msg}, p\rangle, p) : \langle \text{msg}, p\rangle \text{ en transit}\} \\ & & \cup \{(\langle \text{sig}, p\rangle, p) : \langle \text{sig}, p\rangle \text{ en transit}\} \end{cases}$$

Où p et $pere_p$ sont des processus

Algorithme de Dijkstra-Scholten

Correction de l'algorithme

Soit M le nombre de messages de l'algorithme de base.
Dijkstra et Scholten ont montré pour cet algorithme l'existence d'un invariant qui conduit au théorème suivant :

Théorème

L'algorithme de Dijkstra-Scholten est un algorithme correct pour la détection de la terminaison et a recours à M messages de contrôle.

Algorithme de Dijkstra-Scholten

Correction de l'algorithme

- La preuve de ce théorème passe d'abord par la démonstration de la validité de l'invariant.
- L'idée de la preuve de ce théorème est assez simple :

  Le compteur sc_p, toujours positif, est incrémenté par S_p et décrémenté par A_p. Or chaque nouvelle activité naît d'un message de l'algorithme de base et termine par la réception d'un message contrôle. Ainsi, après la réception du même nombre de messages de contrôle que de messages de base, tous les processus sont redevenus passifs et les messages sont tous reçus. Nous sommes bien dans un état terminal, le prédicat ***term*** *est vrai et T est vide.*

Algorithme de Dijkstra-Scholten

Invariant de sûreté

La sûreté de l'algorithme est garantie par l'invariant P suivant :

$$\begin{cases} P \equiv & etat_p = actif \Rightarrow p \in V_T & (1) \\ \wedge & (u,v) \in E_T \Rightarrow u \in V_T \wedge v \in V_T \cap \mathbb{P} & (2) \\ \wedge & sc_p = \#\{v : (v,p) \in E_T\} & (3) \\ \wedge & V_T \neq \emptyset \Rightarrow T \text{ est un arbre de racine } p_0 & (4) \\ \wedge & (etat_p = passif \wedge sc_p = 0) \Rightarrow p \notin V_T & (5) \end{cases}$$

Algorithme de Dijkstra-Scholten

Traduction de l'invariant P

Par définition, T contient à la fois les messages de l'algorithme de base et les messages de l'algorithme de contrôle. D'après l'invariant P :

(1) T contient aussi les processus actifs,

(2) T est un graphe et ses arêtes sont orientées vers les processus,

(3) exprime le nombre de fils pour chaque processus, sommet de T,

(4) T est un arbre de racine p_0,

(5) T se vide lorsque l'algorithme de base termine.

Lemme

P est un invariant pour l'algorithme de Dijkstra-Scholten

Preuve du lemme (1/5)

Initialement

- $etat_p = passif \ \forall p \neq p_0 \wedge pere_p \neq undef \Rightarrow$ **(1)**
- $E_T = \emptyset \Rightarrow$ **(2)**
- $sc_p = 0 \ \forall p \in \mathbb{P} \Rightarrow$ **(3)**
- $V_T = \{p_0\} \wedge E_T = \emptyset$ donc T est un arbre de racine $p_0 \Rightarrow$ **(4)**
- $\forall p \notin V_T \Rightarrow sc_p = 0 \wedge etat_p = passif \Rightarrow$ **(5)**

Preuve du lemme (2/5)

Exécution de l'événement S_p

(1) est conservé car $\mathbf{S}_p$ ne permet à aucun processus de devenir actif

(2) est conservé car p envoie le message $\langle \text{msg}, p \rangle$ avec $p \in V_T$, ajout de $(\langle \text{msg}, p \rangle, \text{p})$ dans E_T

(3) est conservé car $\langle \text{msg}, p \rangle \Rightarrow (\langle \text{msg}, p \rangle, p)$ est ajouté à E_T, T reste un arbre et sc_p est bien le nombre de fils car il est incrémenté

(4) est conservé, idem **(3)**

(5) est conservé car aucun processus ne devient passif

Preuve du lemme (3/5)

Exécution de l'événement R_p

(1) est conservé car soit p est inséré dans V_T soit il y est déjà

(2) idem car si $pere_p$ n'est pas initialisé, il prend q avec $q \in V_T$ sinon il ajoute $\langle \text{sig}, p \rangle$ à E_T avec $q \in V_T$

(3) idem car on ne change pas le nombre de fils de q : $\langle \text{msg}, q \rangle$ est remplacé par p ou par $\langle \text{sig}, q \rangle$

(4) idem car pas de changement dans la structure du graphe

(5) idem car $p \in V_T$ dans tous les cas

Preuve du lemme (4/5)

Exécution de l'événement I_p

- si p devient passif $\Rightarrow$ **(1)**, **(2)**, **(3)** et **(4)**, p était dans V_T
- si $sc_p = 0$ alors p est effacé de la structure $V_T \Rightarrow$ **(5)**
- si p est effacé de T alors $\langle \text{sig}, pere_p \rangle$ est envoyé (et remplace p dans V_T) $\Rightarrow$ **(2)**
- le signal remplace p dans l'arbre comme fils de $pere_p$, sc_p reste donc correct $\Rightarrow$ **(3)**
- la structure du graphe n'est pas changée $\Rightarrow$ **(4)**
- si $pere_p = p$ alors $p = p_0$ et p est une feuille $\Rightarrow$ p est le seul élément de T et lorsque p est effacé T devient vide $\Rightarrow$ **(5)**

Preuve du lemme (5/5)

Exécution de l'événement A_p

- sp_p est décrémenté et le message $\langle \text{sig}, p \rangle$ est supprimé de V_T $\Rightarrow$ **(3)**
- p est le père du message $\langle \text{sig}, p \rangle \Rightarrow p \in V_T$
  si $etat_p = passif \wedge sc_p = 0$ alors p est supprimé dès la réception $\Rightarrow$ **(5)**
- le signal remplace p dans l'arbre comme fils de $pere_p$, sc_p reste donc correct $\Rightarrow$ **(3)**
- si p est effacé de V_T alors l'invariant est pour les mêmes raisons que pour $\mathbf{I}_p$.

Preuve du théorème

Preuve du nombre de messages

- Soit $S = \sum sc_{p \in \mathbb{P}}$, somme de tous les comptes sc_p.
- Initialement, $S = 0$,
- S est incrémenté à chaque message de base, par $\mathbf{S}_p$, et décrémenté à chaque message de contrôle par $\mathbf{A}_p$ et $S \geq 0$, à cause de **(3)**.

$\Rightarrow$ le nombre de messages de contrôle n'excède jamais le nombre de messages de l'algorithme de base, donc M. $\square$

Preuve du théorème

Preuve de la vivacité

Lorsque l'algorithme de base a terminé :

- Seul A_p peut encore être exécuté,
- S est décrémenté seulement d'une unité à chaque appel de A_p,

$\Rightarrow$ l'algorithme atteint une configuration terminale

- Dans cette configuration, V_T n'a plus de message.
- Or d'après **(5)**, V_T ne contient pas de processus passif, donc T n'a plus de feuille ce qui implique qu'il est vide.
- T devient vide seulement lorsque p_0 se supprime.

Preuve du théorème

Preuve de la sûreté

- T devient vide seulement lorsque p_0 se supprime,
- D'après le programme, l'annonce de la terminaison est donc appelée par p_0 seulement.

$\Rightarrow$ **term** est vérifié par **(4)** avec $T = \emptyset$ $\square$

Algorithme de Shavit et Francez

Généralités

Généralisation décentralisée de l'algorithme de Dijkstra-Scholten :

- l'algorithme de base est un algorithme décentralisé, plusieurs initiateurs,
- chaque initiateur p gère un arbre indépendant de ses calculs, noté T_p,
- l'algorithme maintient un graphe $F = (V_F, E_F)$ tel que :
  - soit F est vide ou F est une forêt dans laquelle les initiateurs sont les racines des arbres,
  - chaque arbre contient les messages en transit et les sommets utilisés par le calcul,
- les processus p peuvent être utilisés dans différents calculs.

Algorithme de Shavit et Francez

Principe

- La terminaison est détectée quand la forêt F devient vide,
- Détection non triviale car chaque racine ne peut observer que son arbre : p_0 ne peut détecter que le vide de l'arbre T_{p_0},
- On suppose que si l'arbre T_p s'est vidé, il reste vide ensuite (note : p peut redevenir actif mais dans un autre arbre),
- Terminaison d'un arbre ne signifie pas terminaison de la forêt,
- Vérification de la terminaison faite par un algorithme de vague appelé lorsque que l'arbre est vide,
- Algorithme de vague prend la décision mais peut-être groupé avec *annonce()*.

Algorithme de Shavit-Francez

$etat_p$: $(actif, passif)$ initialisé à $actif$ si p est un initiateur, $passif$ sinon
sc_p : entier (nombre de fils dans T)
$pere_p \in \mathbb{P}$ initialisé à p si $p = p_0$ sinon $ndef$
$vide_p$: booléen initialisé à `faux` si p est un initiateur et `vrai` sinon

S$_p$: $\{etat_p = actif\}$
début
  `send` $\langle$msg,$p\rangle$
  sc_p++

R$_p$: {le message $\langle$msg,$q\rangle$ arrive en p}

début
  `receive`$\langle$msg,$q\rangle$
  $etat_p \leftarrow actif$
  si $pere_p = ndef$ **alors** $pere_p \leftarrow q$
  sinon `send`$\langle$sig,$q\rangle$ à q

Algorithme de Shavit-Francez

I_p : $\{etat_p = actif\}$
début
  $etat_p \leftarrow passif$
  si $(sc_p = 0)$ **alors**
    si $pere_p = p$ **alors**
      $vide_p \leftarrow$ `vrai`
    sinon `send` $\langle sig, pere_p \rangle$ à $pere_p$
    $pere_p \leftarrow ndef$

A_p : {le signal $\langle sig, p \rangle$ arrive en p}
début
  `receive`$\langle sig, p \rangle$
  $sc_p - -$
  si $(sc_p = 0) \wedge (etat_p = passif)$
  alors
    si $pere_p = p$ **alors**
      $vide_p \leftarrow$ `vrai`
    sinon
      `send` $\langle sig, pere_p \rangle$ à $pere_p$
    $pere_p \leftarrow ndef$

Une vague est lancée concurremment par tous les processus p dès lors que $vide_p$ est `vrai`. p *décide* ou envoie $vide_p =$ `vrai`. C'est lorsqu'un processus *décide* qu'est lancée **`annonce()`**

Correction de l'algorithme de Shavit-Francez

Les résultats prouvés au paragraphe précédent pour un arbre sont généralisables à une forêt, comme le théorème précédent :

Théorème

L'algorithme de Shavit-Francez est un algorithme correct pour la détection de la terminaison et utilise $M + W$ messages de contrôle.

La preuve reprend encore une fois la même idée que précédemment. Ce qui change ici, c'est la nécessite qu'une vague de W messages soit lancée entre tous les initiateurs, une fois qu'au moins un arbre est vide. L'algorithme termine lorsque tous les arbres sont vides.

Sommaire

1. Introduction
2. Terminaison pour le calcul sur arbres
3. Solutions utilisant un anneau
   - Algorithme de Dijkstra, Feijen et Van Gasteren
   - Algorithme de Safra

Détection sur un anneau

Propriétés

- Les algorithmes de terminaison sur un anneau utilisent un **jeton**,
- Les processus sont numérotés de p_0 à p_{N-1} avec N le nombre de processus,
- Le sens d'envoi est tel que le processus p_{i+1} envoie au processus p_i,
- p_0 est l'initiateur qui lance la vague sur l'anneau en direction du processus p_{N-1},
- Un tour complet est fait lorsque le jeton retourne en p_0 qui conclue la terminaison ou non.
- La structure de l'anneau est utilisée pour l'algorithme de terminaison mais les processus communiquent en dehors de l'anneau pour l'algorithme de base.

Algorithme de Dijkstra, Feijen et Van Gasteren

Principe

- Les communications sont **synchrones** ou sur rendez vous,
- Lorsque l'instruction *send*() termine, le message est déjà reçu par le destinataire.

$\rightarrow$ il n'y a pas de message en transit, les canaux de communication sont vides.

- Le prédicat **term** s'écrit :

$$\forall p \in \mathbb{P} : etat_p = passif$$

Algorithme de Dijkstra, Feijen et Van Gasteren

Algorithme distribué de base

$etat_p = (actif, passif)$

$\mathbf{C}_{pq} : \{etat_p = actif\}$
début
  $\texttt{send}\langle \text{msg}, q \rangle$
  $etat_q \leftarrow actif$

$\mathbf{I}_p : \{etat_p = actif\}$
début
  $etat_p \leftarrow passif$

Invariant de l'algorithme de contrôle

Mise en évidence

Pour valider son fonctionnement et justifier la construction de l'algorithme

- Soit t le numéro du processus détenteur du jeton, ou le numéro du processus destinataire si il est en transit,
- A l'initialisation de la vague $t = N - 1$, si N est le nombre de processus, et la vague termine lorsque $t = 0$
- Lorsque le processus p_i envoie le jeton au processus p_{i-1} $(0 < i < N)$, t est décrémenté de 1 (le jeton circule dans le sens inverse des numéros de processus),
- Soit P l'invariant de cet algorithme, il doit être tel que nous puissions conclure la terminaison si $t = 0$.

Construction de l'invariant P (1/5)

Première proposition

- Tous les processus qui ont été traversés par le jeton sont passifs.
- Soit $P = P_0$ avec :
  $P_0 \quad \equiv \quad \forall i \ (t < i < N) \ : \ etat_{p_i} = passif$,
- P_0 est vrai lorsque $t = N - 1$,
- Si $t = 0$ et $etat_p = passif$ alors la terminaison est détectée.
- P_0 n'est préservé que dans le cas où l'émetteur du jeton est passif

Règle 1 : *Seul les processus passifs peuvent envoyer le jeton.*

Construction de l'invariant P (2/5)

- L'invariant P_0 est préservé par le transfert du jeton et les évènements internes,
- Par contre, il n'est pas préservé par les communications,
- Si p_i envoie un message au processus p_j, avec $i \leq t$ et $t < j$, alors p_j peut redevenir actif : cas où un processus non encore visité par le jeton envoie un message à un processus déjà visité.
- Il faut donc modifier cet invariant avec une assertion plus faible P_1 telle que $P \equiv P_0 \vee P_1$ et, après chaque violation de P_0, P_1 est vrai.
- L'idée repose sur la mise en place d'une couleur, verte ou rouge, associée à chaque processus avec :

$$P_1 \quad \equiv \quad \exists j \ (0 \leq j \leq t) \ : \ couleur_j = \texttt{rouge}$$

Et les couleurs des nœuds sont initialisées à vert.

Construction de l'invariant P (3/5)

- Chaque fois que P_0 devient faux en raison des messages conduisant au réveil des processus déjà visités, il faut que P_1 deviennent vrai.
- La règle 2 permet cela, tout en renforçant la contrainte puisqu'il n'est pas possible de savoir dans ce contexte si le message est envoyé en direction d'un processus déjà visité ou non :

Règle 2 : *Un processus émetteur d'un message devient rouge.*

- Comme $(P \wedge (couleur_{p_0} = \texttt{vert}) \wedge (t = 0)) \quad \Rightarrow \quad \neg P_1$, la détection de la terminaison est possible avec le nouvel invariant.

Construction de l'invariant P (4/5)

- Par contre,le processus t peut toujours transférer le jeton si il est rouge,
- Il faut donc encore affaiblir l'assertion précédente avec une nouvelle assertion P_2 qui associe une couleur, rouge ou verte, au jeton,
- $P \equiv P_0 \vee P_1 \vee P_2$ avec : $P_2 \quad \equiv \quad$ le jeton est rouge
- La règle 3 définit le changement de couleur du jeton :

Règle 3 : *Un processus rouge autre que P_0 envoie un jeton rouge.*

Ainsi, (jeton `vert`) $\Rightarrow \neg P_2$ et la terminaison peut-être détectée par la réception d'un jeton vert.

Construction de l'invariant P (5/5)

- Maintenant, toute action préserve l'invariant.
- On peut alors établir les conditions pour lesquelles une vague ne permet pas de détecter la terminaison : *le jeton est rouge.* D'où la règle 4 pour la détection de la terminaison :

Règle 4 : *Si une vague est infructueuse pour la détection de la terminaison, on relance une nouvelle vague.*

- Or, si une vague est infructueuse, il faut réinitialiser la couleur de tous les processus rouges, afin qu'ils ne transmettent pas un jeton rouge alors qu'ils n'ont pas envoyé de message depuis la dernière vague :

Règle 5 : *Chaque processus reprend la couleur verte après le passage du jeton*

Correction de l'algorithme de Dijkstra, Feijen et Van Gasteren

Théorème

L'algorithme de Dijkstra, Feijen et Van Gasteren est un algorithme correct pour la détection de la terminaison pour les algorithme distribués utilisant le passage de messages synchrone.

Algorithme de Dijkstra, Feijen et Van Gasteren

Règles

1. Seul les processus passifs peuvent envoyer le jeton.
2. Un processus émetteur d'un message devient rouge.
3. Un processus rouge autre que P_0 envoie un jeton rouge.
4. Si une vague est infructueuse pour la détection de la terminaison, on relance une nouvelle vague.
5. Chaque processus reprend la couleur verte après le passage du jeton

Exercice

Écrire l'algorithme de Dijkstra, Feijen et Van Gasteren

Algorithme de Dijkstra, Feijen et Van Gasteren (1/3)

Données de chaque processus p :

$state_p$: $(active, passive)$ initialisé à $active$
$procColor_p$: $(green, red)$ couleur du processus, initialisée à $green$
$ownToken_p$: le processus possède le jeton, initialisée à $false$
$tokenColor_p$: $(green, red)$ couleur du jeton, initialisée à $green$

Les messages :

$\langle msg, p \rangle$: message de l'exécution de base
$\langle token, color \rangle$: message de contrôle

Règle D_p : $\{state_p = active\}$ `// Début de la détection`

début
  si $p = p_0$ **alors** `send` $\langle token, green \rangle$ à p_{n-1}

Algorithme de Dijkstra, Feijen et Van Gasteren (2/3)

Règle S_p : $\{state_p = active\}$ // Le processus envoie un message à q
début
    send $\langle msg, q \rangle$
    $color_p \leftarrow red$

Règle I_p : $\{state_p = active\}$ // Le processus passe à l'état inactif
début
    $state_p \leftarrow passive$
    si $ownToken_p$ **alors**
        si $color_p = red$ **alors** send $\langle token, red \rangle$ à p_{p-1}
        sinon send $\langle token, tokenColor_p \rangle$ à p_{p-1}
        $color_p \leftarrow green$

Algorithme de Dijkstra, Feijen et Van Gasteren (3/3)

Règle R_p : // Le processus reçoit $\langle token, col \rangle$

début

- **si** $etat_p \neq passif$ **alors**
  - $ownToken_p \leftarrow true$; $tokenColor_p \leftarrow col$
- **sinon**
  - **si** $(p = p_0)$ **alors**
    - **si** $(col = green) \wedge (color_p = blanc)$ **alors** annonce()
    - **sinon** send $\langle token, green \rangle$ à p_{p-1}
  - **sinon**
    - **si** $couleur_p = red$ **alors** send $\langle token, red \rangle$ à p_{p-1}
    - **sinon** send $\langle token, col \rangle$ à p_{p-1}
  - $color_p \leftarrow green$

Généralisation avec l'algorithme de Safra

Généralités

- Communications synchrones garantissent que le message est arrivé. Il est donc inutile de compter les messages en transit,
- Généralisation aux communications asynchrones, basée sur une comptabilité des messages de l'algorithme de base
- Cet algorithme a la même complexité en moyenne que l'algorithme de Dijkstra, Feijen et Van Gasteren

Comme précédemment, le raisonnement s'appuie sur la construction d'un invariant, et sur la construction des règles permettant de placer l'algorithme de contrôle dans les conditions de pouvoir détecter la terminaison, que si l'application vérifie le prédicat **term**.

Invariant pour l'algorithme de Safra (1/6)

- On définit B le nombre de messages en transit,
- Le prédicat **term** devient :

$$\textbf{term} \Leftrightarrow (\forall p\ etat_p = passif) \wedge (B = 0)$$

- On définit mc_p, un compteur de messages par processus. Lorsque p envoie un message mc_p est incrémenté. Lorsque p reçoit un message, mc_p est décrémenté.
- Soit $P = P_m$ l'invariant :

$$P_m \equiv \left(B = \sum_{p \in \mathbb{P}} mc_p\right) \text{ avec au départ } mc_p = 0$$

Invariant pour l'algorithme de Safra (2/6)

- Pour que p_0 décide la terminaison, il faut qu'il ait accès à la somme des compteurs des messages, qui doit donc être véhiculée par le jeton
- Pour l'invariant, on introduit l'assertion P_0 telle que $P = P_m \wedge P_0$:

$$P_0 \quad \equiv \quad \left(\forall i, t < i < N \ : \ etat_{p_i} = passif\right) \wedge \left(q = \sum_{t<i<N} mc_{p_i}\right)$$

- P_0 est vrai si $t = N - 1$ et $q = 0$,
- Si $t = 0$ alors P implique que tous les processus avant p_0 sont passifs et que le nombre de messages de l'algorithme de base est $B = q + mc_{p_0}$
- Donc p_0 peut valider la terminaison si p_0 est passif et si $(q + mc_{p_0}) = 0$
- Pour que P_0 joue son rôle, on définit la règle 1 suivante :

Règle 1 : *Un processus ne prend le jeton que lorsqu'il est passif est lorsqu'il passe le jeton, il ajoute la valeur de son compteur de messages à q.*

Invariant pour l'algorithme de Safra (3/6)

- L'invariant P_0 est protégé des transferts de jeton et des événements internes
- Mais il n'est pas préservé lorsqu'un message arrive sur le processus p_i avec $i > t$ puisque, dans ce cas, la seconde partie du prédicat $(q = \sum_{t<i<N} mc_{p_i})$ est violée
- Cela ne peut arriver que pour une réception et nous avons forcément $B > 0$
- Il est alors possible d'affaiblir P_0 par une nouvelle assertion afin de préserver l'invariant
- $P = P_m \wedge (P_0 \vee P_1)$ et :

$$P_1 \quad \equiv \quad \left(\sum_{i \leq t} mc_{p_i} + q \right) > 0$$

Invariant pour l'algorithme de Safra (4/6)

- Comme pour l'algorithme précédent, lorsqu'un message arrive sur un processus p_i avec $i \leq t$, P_1 n'est pas préservé.
- On ajoute alors P_2 et P devient $P_m \wedge (P_0 \vee P_1 \vee P_2)$
- P_2 est définit par :

$$P_2 \quad \equiv \quad \exists j \ (0 \leq j \leq t) : couleur_{p_j} = \texttt{rouge}$$

- Pour respecter l'invariant, on établit la règle 2 :

Règle 2 : *Un processus qui reçoit un message devient* **rouge**.

- Si P_1 est faux, alors P_2 est vrai. La détection de la terminaison est toujours possible dès que :

$$(P \wedge (couleur_{p_0} = \texttt{vert}) \wedge (t = 0)) \quad \Rightarrow \quad \neg P_2$$

- On conserve donc l'invariant pour tout événement interne et toute communication de l'algorithme de base.

Invariant pour l'algorithme de Safra (5/6)

- Il faut également gérer le passage du jeton lorsque le processus qui le possède est rouge,
- On ajoute l'assertion P_3 à P : $P = P_m \wedge (P_0 \vee P_1 \vee P_2 \vee P_3)$
- Avec : $P_3 \quad \equiv \quad$ le jeton est rouge
- Cette assertion est en accord avec la règle 3 suivante :

Règle 3 : *Un processus rouge fait passer le jeton à rouge*

- On a (jeton `vert`) $\Rightarrow \neg P_3$. Ce qui permet au processus p_0 de conclure quant à la détection de la terminaison pour la vague courante.

Invariant pour l'algorithme de Safra (6/6)

- Si la terminaison n'a pu être détectée, une nouvelle vague doit être relancée :

Règle 4 : *Si une vague est infructueuse pour la détection de la terminaison, alors une nouvelle vague est relancée.*

- Pour que deux vagues successives ne donnent pas le même résultat, il faut réinitialiser les couleurs des processus. D'où la règle 5 :

Règle 5 : *À chaque passage du jeton au processus suivant, le processus reprend la couleur verte.*

Correction de l'algorithme de Safra

Théorème

L'algorithme de Safra est un algorithme correct pour la détection de la terminaison d'une application distribuée avec l'utilisation de passage de messages asynchrones.

Algorithme de Safra (1/2)

Variables de chaque processus P_i
$etat_p = (actif, passif)$ **init** $actif$
$couleur_p = (vert, rouge)$ **init** $vert$
mc_p : entier **init** 0

Messages utilisés :
$\langle msg \rangle$: message de l'algorithme de base
$\langle$**jeton**$, c, m\rangle$: message jeton, couleur c, entier m

Début de détection exécutée une seule fois par le processus p_0
début
  send $\langle$**jeton**, $vert$, 0$\rangle$ à p_{n-1}

Règle $S_p : \{etat_p = actif\}$
début
  send $\langle$msg$\rangle$
  $mc_p \leftarrow mc_p + 1$

Algorithme de Safra (2/2)

Règle R_p : {le message $\langle msg \rangle$ arrive en p}
début
  receive $\langle$msg$\rangle$
  $etat_p \leftarrow actif$
  $mc_p \leftarrow mc_p - 1$
  $couleur_p \leftarrow rouge$

Règle I_p : $\{etat_p = actif\}$
début
  $etat_p \leftarrow passif$

Règle T_p : $\{etat_p = passif\}$ `/* p reçoit le message` $\langle$`jeton`$, c, m\rangle$ `*/`
début
  si $p = p_0$ **alors**
    si $(c = vert) \wedge (couleur_p = vert) \wedge (mc_p + m = 0)$ **alors** `annonce()`
    sinon `send` $\langle$**jeton**$, vert, 0\rangle$ à P_{n-1}
  sinon
    si $(couleur_p = vert)$ **alors** `send` $\langle$**jeton**$, c, m + mc_p\rangle$ à P_{p-1}
    sinon `send` $\langle$**jeton**$, rouge, m + mc_p\rangle$ à P_{p-1}
  $couleur_p \leftarrow vert$