Algorithmique Répartie Avancée : Introduction à l'Auto-Stabilisation

Swan Dubois*

Résumé

Ce cours est consacré à une introduction à l'auto-stabilisation. Il s'agit d'une méthode de tolérance aux fautes transitoires dans les systèmes répartis. Ce cours a pour objectif de vous familiariser avec cette notion à travers quelques exemples classiques. Nous insisterons en particulier sur l'impact des critères d'asynchronisme du système sur les algorithmes auto-stabilisants.

Vous pouvez remarquer que ce polycopié de cours n'est pas complet. Ce résumé comporte uniquement les énoncés formels et les algorithmes que nous verrons dans ce cours. Il ne s'agit que d'un support afin de vous aider dans votre prise de notes et en aucun cas d'un cours exhaustif.

1 Introduction

1.1 Fautes

1.2 Auto-Stabilisation

1.3 Modèle à États

Nous considérons le système réparti comme un graphe connexe non orienté $G = (V, E)$ où V est un ensemble de processeurs et E est une relation binaire qui modélise la capacité de deux processeurs à communiquer. Nous dirons que p et q sont voisins si $(p, q) \in E$. Tout processeur p peut distinguer localement ses voisins que le système soit identifié ou anonyme. Dans la suite, nous désignerons par n le nombre de processeurs dans V, par N_p l'ensemble des voisins d'un processeur p, par $d(p, q)$ la distance entre deux processeurs p et q, par Δ le degré maximal de G et par D le diamètre de G.

Nous considérons le modèle à états qui est un modèle à mémoire partagée dans lequel les communications entre processeurs voisins sont réalisées par des lectures directes de variables plutôt que par des échanges de messages. Dans ce modèle, écrire un algorithme réparti $\mathcal{A}$ consiste à associer à chaque processeur un ensemble de variables et de règles. Un processeur peut lire et écrire ses propres variables tandis qu'il peut exclusivement lire celles de ses voisins. Chaque règle du processeur p est de la forme $<\textit{Étiquette}> :: <\textit{Garde}> \longrightarrow <\textit{Action}>$. L'étiquette de la règle est simplement un nom pour faire référence à la règle. La garde de la règle est un prédicat impliquant uniquement des variables de p et de ses voisins. L'action de la règle est un ensemble d'instructions modifiant uniquement les variables de p. Une action ne peut être exécutée que si la garde de la règle correspondante est vraie (la règle est alors dite activable).

L'état d'un processeur est défini par la valeur de toutes ses variables. La configuration du système est le produit de l'état de tous les processeurs. Nous notons Γ l'ensemble des configurations du système. Un processeur est activable dans une configuration si et seulement si au

*swan.dubois@lip6.fr

moins une de ses règles est activable dans cette configuration. Une exécution de $\mathcal{A}$ issue d'une configuration γ_0 est une séquence maximale de configurations $\epsilon = \gamma_0\gamma_1 \ldots \gamma_i\gamma_{i+1} \ldots$ telle que, pour tout i, l'étape (γ_i, γ_{i+1}) corresponde à l'exécution d'une règle de $\mathcal{A}$ par un sous-ensemble non-vide des processeurs activables dans γ_i (s'il en existe). Le fait que l'exécution soit maximale signifie qu'elle est infinie ou bien qu'aucun processeur n'est activable dans la dernière configuration (qui est alors dite terminale). Un processeur est neutralisé durant une étape s'il est activable au début de l'étape, non activable à la fin mais qu'il n'a exécuté aucune règle durant cette étape.

Nous pouvons à présent donner une définition formelle de l'auto-stabilisation. Nous associons à chaque problème une spécification. Il s'agit d'un prédicat qui indique si une exécution d'un algorithme satisfait les contraintes du problème ou pas.

Définition 1 (Auto-stabilisation) *Un algorithme réparti $\mathcal{A}$ est auto-stabilisant pour une spécification $\mathcal{S}$ s'il existe un ensemble non-vide de configurations (dites légitimes) tel que :*

(clôture) toute exécution de $\mathcal{A}$ issue d'une configuration légitime satisfait $\mathcal{S}$; et

(convergence) toute exécution de $\mathcal{A}$ issue de toute configuration de Γ contient une configuration légitime.

2 Premier Algorithme : Exclusion Mutuelle

2.1 Spécification du Problème

Spécification 1 (Exclusion mutuelle)

Vivacité : Tout processeur peut entrer infiniment souvent en section critique.

Sûreté : Dans toute configuration, au plus un processeur est en section critique.

Spécification 2 (Circulation de jeton)

Vivacité : Tout processeur possède infiniment souvent le jeton.

Sûreté : Dans toute configuration, au plus un processeur possède un jeton.

2.2 Algorithme

Nous posons les hypothèses suivantes. Le système est un anneau orienté de n processeurs d'identités $p_0, \ldots, p_{n-1}$ (dans le sens de l'orientation). Chaque processeur p_i a une unique variable $c_i \in \{0, 1, \ldots, k-1\}$ (où k est une constante telle que $k > n$).

Nous définissons la présence du jeton de la manière suivante. Le processeur p_0 possède le jeton si $c_0 = c_{n-1}$ alors qu'un processeur p_i ($i \neq 0$) possède le jeton si $c_i \neq c_{i-1}$.

Algorithme 1 Algorithme de Dijkstra pour le processeur p_i

Variable pour p_i :

$c_i \in \{0, 1, \ldots, k-1\}$

Règle pour p_0 :

$(R_1) :: c_0 = c_{n-1} \longrightarrow c_0 := c_0 + 1 \text{ modulo } k$

Règle pour p_i, $i \neq 0$:

$(R_2) :: c_i \neq c_{i-1} \longrightarrow c_i := c_{i-1}$

2.3 Exemples

2.4 Preuve

Lemme 1 *Au moins un processeur possède le jeton dans toute configuration de* Γ.

Lemme 2 *Tout processeur possède le jeton infiniment souvent dans toute exécution issue de toute configuration de* Γ.

Lemme 3 *Toute exécution issue d'une configuration dans laquelle existe un unique jeton satisfait la spécification de la circulation de jeton.*

Lemme 4 *Toute exécution issue de toute configuration de* Γ *atteint en un temps fini une configuration dans laquelle existe un unique jeton.*

Théorème 1 *L'algorithme de Dijsktra est un algorithme auto-stabilisant d'exclusion mutuelle.*

3 Modéliser l'Asynchronisme : les Démons

3.1 Caractéristiques des Démons

Un démon est un adversaire extérieur à l'algorithme qui sélectionne à chaque étape le sous-ensemble des processeurs activables autorisés à exécuter une de leur règles. Si on ne met aucune restriction sur le démon, nous pouvons simuler tout entrelacement d'actions des processeurs donc les exécutions considérées sont totalement asynchrones.

L'objectif du démon étant d'empêcher la stabilisation de l'algorithme si cela lui est possible, on peut être amené à vouloir restreindre le pouvoir du démon en lui imposant certaines contraintes. Nous nous intéresserons ici à sa distribution et à son équité.

Distribution

Démon distribué : Pas de contrainte sur la distribution.

Démon localement central : Deux processeurs voisins ne peuvent pas être choisis simultanément par le démon.

Démon central : Deux processeurs ne peuvent pas être choisis simultanément par le démon.

Équité

Démon inéquitable : Pas de contraintes sur l'équité

Démon faiblement équitable : Un processeur ne peut pas être infiniment longtemps activable sans être choisi par le démon.

Démon fortement équitable : Un processeur ne peut pas être infiniment souvent activable sans être choisi par le démon.

3.2 Exemple du Problème du Coloriage

Spécification 3 (Coloriage)

Vivacité : Toute exécution est finie.

Sûreté : Dans toute configuration terminale, deux processeurs ne portent pas la même couleur.

Théorème 2 *Il n'existe pas d'algorithme auto-stabilisant pour le coloriage sous le démon distribué et inéquitable.*

Algorithme 2 Algorithme du $(\Delta+1)$-coloriage pour le processeur p

Variable pour p :

$c_p \in \{0, \ldots, \Delta+1\}$: couleur du processeur p

Règle pour p :

$(C) :: (\exists q \in N_p, c_q = c_p) \longrightarrow c_p := c$ avec $c \in \{0, \ldots, \Delta+1\} \setminus \{c_q | q \in N_p\}$

Théorème 3 *L'algorithme du $(\Delta+1)$-coloriage est un algorithme auto-stabilisant de coloriage sous le démon central et inéquitable.*

4 Deuxième Algorithme : Arbre en Largeur

4.1 Spécification du Problème

Spécification 4 (Construction de l'arbre en largeur)

Vivacité : Toute exécution est finie.

Sûreté : Dans toute configuration terminale, il existe un arbre couvrant en largeur enraciné sur le processeur racine.

4.2 Algorithme

Nous faisons les hypothèses suivantes. Le graphe de communication est quelconque. Seul un processeur r est distingué comme étant la racine de l'arbre à construire (les autres processeurs peuvent être anonymes).

Algorithme 3 Algorithme du $min+1$ pour le processeur p

Variables pour p :

$P_p \in N_p \cup \{\perp\}$: pointeur sur le père de p dans l'arbre

$d_p \in \mathbb{N}$: distance à la racine de p dans l'arbre

Règle pour $p = r$:

$(R_1) :: (P_r \neq \perp) \vee (d_r \neq 0) \longrightarrow P_r := \perp;\ d_r := 0$

Règle pour $p \neq r$:

$(R_2) :: (P_p = \perp) \vee (d_p \neq min\{d_q | q \in N_p\} + 1) \vee (d_p \neq d_{P_p} + 1)$
$\longrightarrow P_p := q;\ d_p := d_q + 1$ avec q tel que $d_q = min\{d_k | p_k \in N_p\}$

Nous supposons que cet algorithme s'exécute sous un démon distribué faiblement équitable. Notez que cette hypothèse n'est pas nécessaire à la stabilisation de l'algorithme mais il rend la preuve bien plus simple et élégante.

4.3 Exemples

4.4 Preuve

Nous introduisons pour le besoin de la preuve le prédicat suivant :

$$\forall \gamma \in \Gamma, \forall i \in \{0,\dots,D{+}1\}, A(\gamma,i) \equiv \begin{cases} P_r = \perp \wedge d_r = 0 \\ et \\ \forall p \in V \setminus \{r\}, d(p,r) < i \Rightarrow P_p \neq \perp \wedge d_p = d(p,r) \wedge d_{P_p} = d_p - 1 \\ et \\ \forall p \in V, d(p,r) \geq i \Rightarrow P_p \neq \perp \wedge d_p \geq i \end{cases}$$

Définissons une famille d'ensemble de configurations de la manière suivante :

$$\forall i \in \{0,\dots,D+1\}, \Gamma_i = \{\gamma \in \Gamma | A(\gamma,i) = vrai\}$$

Nous allons utiliser la notion d'attracteur dans la preuve de cet algorithme. Étant donné deux sous-ensembles de configurations $\Gamma_2 \subseteq \Gamma_1 \subseteq \Gamma$, nous disons que Γ_2 est un attracteur de Γ_1 (noté $\Gamma_1 \rhd \Gamma_2$) si toute exécution de l'algorithme partant de toute configuration de Γ_1 atteint en un temps fini une configuration de Γ_2 et si Γ_2 est clos pour l'algorithme.

Nous introduisons également la notion de ronde. La première ronde d'une exécution ϵ, notée ϵ', est le préfixe minimal de ϵ qui contient l'exécution d'au moins une règle ou la neutralisation de chaque processeur qui était activable au début de la ronde. Soit ϵ'' le suffixe de ϵ tel que $\epsilon = \epsilon'\epsilon''$. La seconde ronde de ϵ est la première ronde de ϵ'', etc. Le fait de supposer un démon faiblement équitable nous garantit que toute ronde est finie.

Lemme 5 *Toute configuration $\gamma \in \Gamma_{D+1}$ est terminale et les pointeurs P de l'ensemble des processeurs forment un arbre couvrant en largeur enraciné en r dans γ.*

Lemme 6 $\Gamma \rhd \Gamma_0$.

Lemme 7 $\forall i \in \{0,\dots,D\}, \Gamma_i \rhd \Gamma_{i+1}$.

Théorème 4 *L'algorithme du min+1 est un algorithme auto-stabilisant de construction d'arbre en largeur.*

5 Troisième Algorithme : Mariage Maximal

5.1 Spécification du Problème

Définition 2 (Couplage) *Pour tout graphe $G = (V,E)$, un couplage C sur G est un sous-ensemble de E tel que tout sommet de V appartient à au plus une arête de C. Un couplage est maximal (resp. maximum) s'il n'existe aucun couplage C' tel que $C \subsetneq C'$ (resp. $|C| < |C'|$).*

Spécification 5 (Mariage Maximal)

Vivacité : Toute exécution est finie.

Sûreté : Dans toute configuration terminale, il existe un couplage maximal sur le graphe de communication.

5.2 Algorithme

Nous faisons les hypothèses suivantes dans la suite. Le graphe de communication est arbitraire et le système est anonyme. Le démon est supposé central et inéquitable. Chaque processeur p a une unique variable $pref_p \in N_p \cup \{\perp\}$ qui indique le voisin préféré de p pour un mariage. Par exemple, si $pref_p = q$, cela signifie que p souhaite ajouter l'arête $\{p, q\}$ au mariage en construction. Pour la suite, nous définissons pour chaque processeur p l'ensemble de prédicats suivants (il est facile de vérifier que pour toute configuration γ et pour tout processeur p, exactement un de ces prédicats est vrai pour p dans γ) :

$$\begin{array}{rcc} \text{demandeur}_p & \equiv & (pref_p = q) \wedge (pref_q = \perp) \\ \text{marié}_p & \equiv & (pref_p = q) \wedge (pref_q = p) \\ \text{condamné}_p & \equiv & (pref_p = q) \wedge (pref_q = r) \vee (r \neq p) \\ \text{mort}_p & \equiv & (pref_p = \perp) \wedge (\forall q \in N_p, \text{marié}_p = vrai) \\ \text{libre}_p & \equiv & (pref_p = \perp) \wedge (\exists q \in N_p, \text{marié}_p = faux) \end{array}$$

Algorithme 4 Algorithme du mariage glouton pour le processeur p

Variables pour p :

$pref_p \in N_p \cup \{\perp\}$

Règle pour p :

/* Règle de mariage */
$(M) :: (pref_p = \perp) \wedge (\exists q \in N_p, pref_q = p) \longrightarrow pref_p := q$
/* Règle de séduction */
$(S) :: (pref_p = \perp) \wedge (\exists q \in N_p, pref_q = \perp) \longrightarrow pref_p := q$
/* Règle d'abandon */
$(A) :: (pref_p = q) \wedge (pref_q \neq p) \wedge (pref_q \neq \perp) \longrightarrow pref_p := \perp$

5.3 Exemples

5.4 Preuve

Lemme 8 *Si une configuration γ satisfait $\forall p \in V, marié_p \vee mort_p$, alors l'ensemble d'arêtes $\{\{p, pref_p\} | pref_p \neq \perp\}$ est un mariage maximal dans γ.*

Lemme 9 *Toute configuration terminale satisfait $\forall p \in V, marié_p \vee mort_p$.*

Lemme 10 *Toute configuration qui satisfait $\forall p \in V, marié_p \vee mort_p$ est terminale.*

Lemme 11 *Toute exécution issue de toute configuration de Γ atteint en un temps fini une configuration terminale.*

Nous allons utiliser ici une fonction de potentiel pour prouver la convergence de cet algorithme. Une fonction de potentiel est une fonction bornée qui associe à toute configuration du système une valeur qui décroît strictement à chaque exécution d'une règle par l'algorithme. Dans ce cas, si la valeur minimale de la fonction est associée à une configuration terminale, l'existence d'une fonction de potentiel suffit à prouver la convergence de l'algorithme. La difficulté étant de trouver la bonne fonction de potentiel, nous donnons ici celle qui convient pour notre preuve.

Pour toute configuration $\gamma \in \Gamma$, définissons les fonctions suivantes : $d(\gamma)$ renvoie le nombre de processeurs demandeurs dans γ, $c(\gamma)$ renvoie le nombre de processeurs condamnés dans γ et $l(\gamma)$ renvoie le nombre de processeurs libres dans γ. Nous définissons alors la fonction de

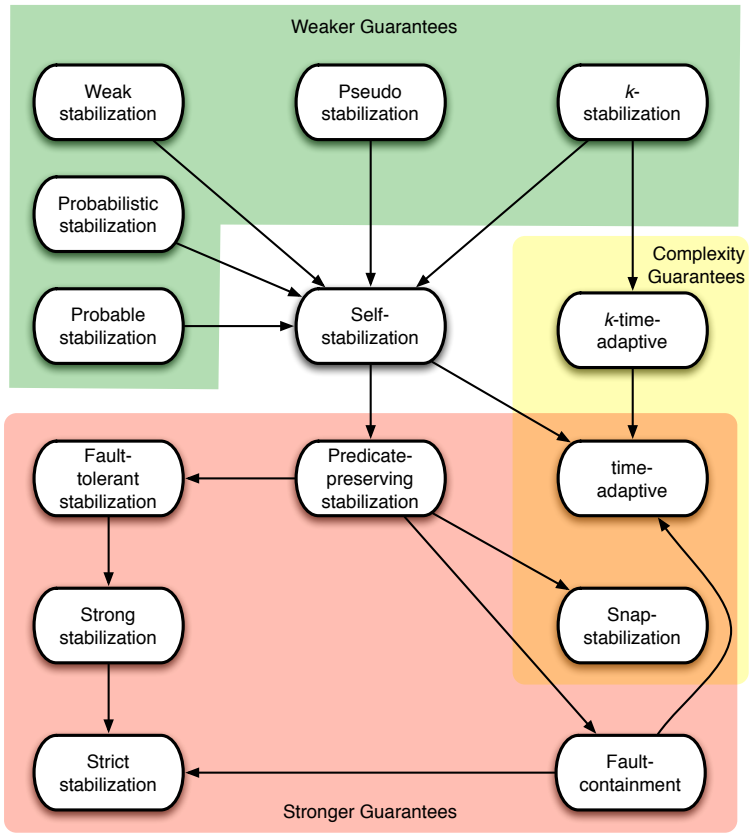

FIGURE 1 – Variantes de l'auto-stabilsiation d'après [7].

potentiel suivante $P(\gamma) = (d(\gamma) + c(\gamma) + l(\gamma), 2c(\gamma) + l(\gamma))$. Nous comparons les valeurs de P par ordre lexicographique.

Théorème 5 *L'algorithme du mariage glouton est un algorithme auto-stabilisant de mariage maximal sous un démon central et inéquitable.*

6 Variantes de l'Auto-Stabilisation

Pour aller plus loin...

[1] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11) :643–644, 1974.

[2] Shlomi. Dolev. *Self-stabilization*. MIT Press, March 2000.

[3] Swan Dubois and Sébastien Tixeuil. A taxonomy of daemons in self-stabilization. Technical Report 1110.0334, ArXiv eprint, October 2011.

[4] Ted Herman. A comprehensive bibliography on self-stabilization. http ://www.cs.uiowa.edu/ftp/selfstab/bibliography/, 2002.

[5] Su-Chu Hsu and Shing-Tsaan Huang. A self-stabilizing algorithm for maximal matching. *Inf. Process. Lett.*, 43(2) :77–81, 1992.

[6] Shing-Tsaan Huang and Nian-Shing Chen. A self-stabilizing algorithm for constructing breadth-first trees. *Inf. Process. Lett.*, 41(2) :109–117, 1992.

[7] Sébastien Tixeuil. *Algorithms and Theory of Computation Handbook, Second Edition*, chapter Self-stabilizing Algorithms, pages 26.1–26.45. Chapman & Hall/CRC Applied Algorithms and Data Structures. CRC Press, Taylor & Francis Group, November 2009.