Algorithmes distribués avec mémoire partagée

November 28, 2013

Contents

Introduction

On peut considérer l'algorithmique distribuée selon plusieurs aspects :

la mémoire : mémoire partagée (shared memory) vs. message passing. Dans premier cadre, les écritures sont plus fiable que dans le second dans lequel il faut prendre en compte les défaillance.

le temps : caractère synchrone (délais connus bornés mais nécessite une horloge globale) ou asynchrone (délais non bornés, un message peut prendre un temps arbitraire pour être reçu)

Quelles genre de défaillances ?

- ▷ les pannes des liens de communication
- ▷ les pannes des processus : crash (arrêt du processus), omissions (oubli d'envoi de messages), byzantines (éxécution de code arbitraire menant le processus à se comporter comme un adversaire)

À quels paramètres va-t-on s'intéresser ?

▷ **n** le nombre de processus

▷ **f** le nombre de processus défaillants (dans l'éxécution donnée)

▷ **t** le nombre de processus défaillants tolérés par le réseau

On va s'intéresser à des problèmes d'accord : les processus sont initialisés avec des valeurs éventuellement distinctes et doivent se mettre en accord pour obtenir un unique résultat final qui corresponde au cadre de notre problème.

1 Attaque coordonnée

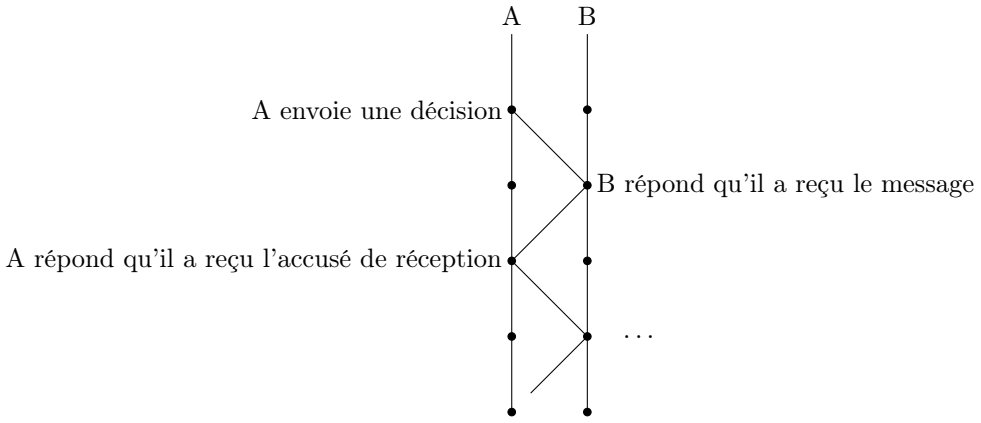
Il s'agit du problème de défaillance des liens de communication. On va se placer dans le cadre du message passing avec un système synchrone (le résultat d'impossibilité de cette partie s'applique donc aussi pour les systèmes asynchrones).

Spécification du problème : Le problème `attaque_coordonnée` correspond à n processus qui communiquent en envoyant des messages petits. Chaque processus `p` a une valeur initiale $v_{\mathtt{p}} (\in \{0,1\})$. Ces processus doivent prendre une décision (irrévocable) : chaque processus `p` posséde une variable $d_{\mathtt{p}}$ qui ne peut être écrite qu'une seule fois. Les propriétés suivantes doivent être vérifiées :

1. *l'accord :* si `p` et `q` décident alors $d_{\mathtt{p}} = d_{\mathtt{q}}$
2. *la terminaison :* tout processus `p` décide
3. *la validité :* si $\forall \mathtt{p}, v_{\mathtt{p}} = 0$ alors la seule décision possible est 0 ($d_{\mathtt{p}} \neq \perp \Longrightarrow d_{\mathtt{p}} = 0$) ; et si $\forall \mathtt{p}, v_{\mathtt{p}} = 1$ et qu'il n'y a aucune perte de message alors la seule décision possible est 1.

On peut interpréter la valeur 0 par un message "abort" tandis que 1 correspond à "commit". Le nom provient de l'image suivante : si plusieurs armées sont de part et d'autre de l'ennemi et veulent se mettre d'accord pour lancer une attaque coordonnée (les messagers qui doivent traverser les rangs ennemis peuvent être ammenés à connaître un destin funeste). On cherche donc un protocole pour résoudre ce problème.

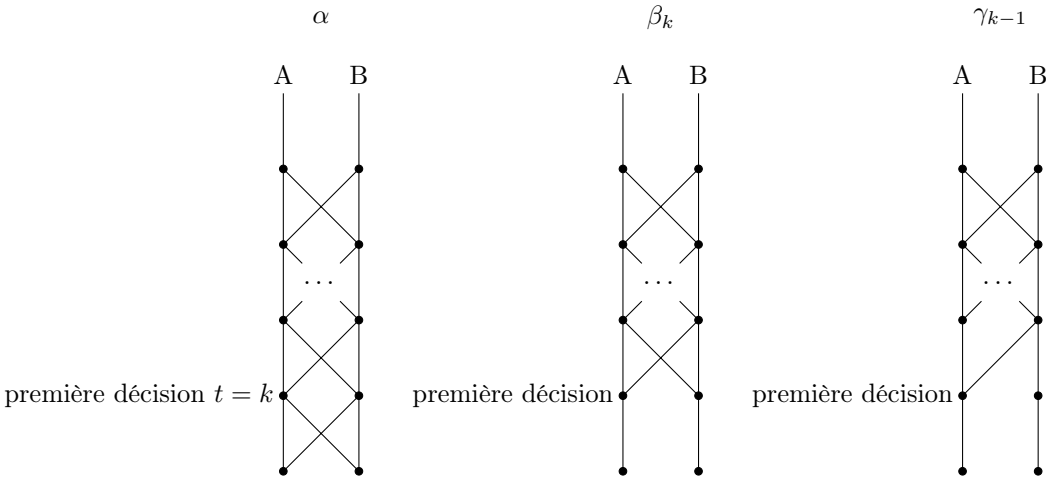
1.1 Impossibilité de la résolution



Theorem 1.1. *Pour tout graphe de communication avec au moins* 2 *processus alors il n'y a pas de solution au problème* `attaque_coordonnée`

Proof : On se ramène à 2 processus et un graphe complet : si on a plus de processus, on considére le premier processus comme le processus A et tous les autres comme un seul processus B ; si le graphe n'est pas complet alors il est impossible à un processus d'envoyer un message et de savoir que son message a bien été réceptionné. De plus, on a une "ronde synchrone" : au temps i, chaque processus envoie un message qui est reçu avant le temps $i+1$. Une solution au cadre générale s'adapte toujours à ce cadre particulier. Les processus sont déterministes : l'état en $i+1$ de `p` ne dépend que de son état en i et des messages reçus.

On considère le schéma de communication S composé de l'ensemble des triplets (i,j,k) correspondant au fait que le message de i vers j de la ronde k est reçu. Une éxécution α est une paire (I,S) où S est un schéma de décision et I l'ensemble des valeurs initiales. Étant donnés deux éxécutions α et α', on dit qu'elles sont *indistingables* pour le processus i ce que l'on note $\alpha \sim_i \alpha'$ si les deux éxécutions ont le même état initial et les mêmes messages pour i. On note $\alpha \sim \alpha'$ s'il existe un i tel que $\alpha \sim_i \alpha'$. Pour α une éxécution, si on note $d(\alpha)$ *la* décision correspondant à α alors la terminaison dit qu'il existe au moins une décision et l'accord qu'il y en a au plus une. Si $\alpha \sim_i \alpha'$ alors par déterminisme $d(\alpha) = d(\alpha')$.



On considére d'abord l'éxécution $\alpha = (\{1,1\}, C_\varnothing)$ où $C_\varnothing$ est le schéma de communication sans perte. Il existe un certain k tel que un des processus (disons le processus A) décide 1 (c'est la seule possibilité dans ce cas). On crée l'éxécution β_k qui est la même que α jusqu'au temps k puis dans lequel plus aucun

message n'est reçu pour $t > k$. L'éxécution β_k décide toujours 1. On fabrique ensuite γ_{k-1} à partir de β_k en supprimant uniquement le dernier message de A vers B. On a $\beta_k \sim_A \gamma_{k-1}$ et $\gamma_{k-1} \sim_B \beta_{k-1}$. En répétant l'opération, on obtient que β_0 décide 1 mais $\beta_0 \sim_A \delta_{(1,0)}$ et $\delta_{(0,1)} \sim_B \delta_{(0,0)}$ où $\delta_{(a,b)}$ est l'éxécution avec les valeurs initiales $\{a, b\}$ et sans aucun message. $\delta_{(0,0)}$ doit décider 0 par validité et 1 par indistingabilité, ce qui est donc absurde.

1.2 Solution non-déterministe

On considère un cadre dans lequel un adversaire $\mathfrak{B}$ *choisit* le schéma S de communication et les valeurs initiales. Les processus sont randomizés : $pr^{\mathfrak{B}}(X)$ est la probabilité de X pour un adversaire $\mathfrak{B}$ donné. Le problème ϵ-`attaque_coordonnée` est essentiellement le même que `attaque_coordonnée` mais la régle de validité est remplacée par l'ϵ-validité :

$$\text{pour tout adversaire } \mathfrak{B},\ pr^{\mathfrak{B}}(\text{"un processus décide 0 et un processus décide 1"}) \leq \epsilon$$

1.2.1 Dans le cas de 2 processus

On introduit la notion de "niveau de connaissance" $l(i, k)$ où i est le processus et k la ronde :

- ▷ niveau 0 : pour chaque processus, à chaque ronde, on ne sait rien de l'autre
- ▷ niveau $k + 1$: je "sais" que l'autre processus est au niveau k

Certaines propriétés intéressantes sont vérifiées :

1. Chaque processus peut calculer son niveau à chaque ronde
2. À la ronde k, le niveau de connaissance entre A et $\mathfrak{B}$ diffère au plus d'un $|l(A, k) - l(B, k)| \leq 1$
3. Si $\mathfrak{B}$ n'élimine aucun message, $l(A, k) = l(B, k) = k$

On peut alors écrire l'algorithme suivant :

```
Choisir aléatoirement k entre 1 et r
    faire r rondes
    au bout des r rondes
        regarder si le niveau k est atteint
            -> s'il est atteint et si les deux valeurs initiales sont 1
               alors décider 1
            -> sinon décider 0
```

Notre algorithme est-il valide ? Dans le cas d'un schéma sans perte, la propriété (3) ci-dessus donne que $l(A, r) = l(B, r) = r \geq k$ donc on décide bien 1. Si toutes les valeurs initiales sont nulles, on décide aussi 0. S'il n'y a pas d'accord, considérons que les valeurs initiales sont 1. Le niveau de connaissances pour l'un est k et l'autre est $k - 1$. Si l est le max des niveaux de connaissances, il y a un désaccord possible si $l = k$. L'adversaire choisit l, il y a désaccord possible si $k = l$, c'est à dire avec probabilité $\frac{1}{r}$.

1.2.2 Avec n processus

Pour une éxécution γ avec un schéma de communication S, on definit $\leq_\gamma$ par :

- ▷ $(i, k) \leq_\gamma (i, k')$ si $k < k'$
- ▷ $(i, j, k) \in S \Longrightarrow (i, k - 1) \leq_\gamma (j, k)$
- ▷ en prenant la fermeture transitive.

Le niveau de connaissance $l(i,k)$ est :

- ▷ 0 s'il existe un j tel que $\neg((j,0) \leq (i,k))$
- ▷ $\min_j(l_j)+1$ si $\forall j,(j,0) \leq (i,k)$ en posant $l_j = \max\{l(j,k') \mid (j,k') \leq (i,k)\}$

Les propriétés vérifiées dans ce cas sont :

1. chaque processus peut calculer $l(j,k)$ à la ronde k
2. Pour tout k,j,i $|l(j,k)-l(i,k)| \leq 1$
3. Si S est un schéma sans perte $l(j,k)=k$ pour tous j et k

En effet, pour (0), chaque processus j ajoute son $l(j,k)$ L. $L_i[j]$ dénote le niveau de connaissance de j connu par i. Quand i reçoit M de j,

$$\forall k, L_i[k] = \max(L_i[k], M[k])$$

Si $\forall k, L_i[k] \neq 0$, alors $L_i[i] = \min(L[j])+1$, on calcule ainsi $l(i,k)$.

L'algorithme : Le processus 1 choisit aléatoirement k compris entre 1 et r. À chaque processus, on ajoute (L,V,k) avec V les valeurs initiales connues. Pour les rondes de 1 à r, on envoie (L,v,k) à tous, puis on met à jour L,V,k. À la fin des rondes, si $k \neq \perp$, $L(i) \geq k$ et $V[j]=1 \forall j$ alors on décide 1, sinon on décide 0.

Validité de l'algorithme : Si une des valeurs initiales est nulle alors on décide 0. Si toutes les valeurs initiales sont à 1 alors il y a désaccord si $k = \max_i l(i,k)$ c'est à dire avec probabilité $\frac{1}{r}$.

Proposition 1.1. *Tout algorithme en r rondes qui résout* `attaque_coordonnée` *a une probabilité de désaccord d'au moins* $\frac{1}{r+1}$.

On montre :

Lemma 1.1. *Soit ϵ la probabilité de désaccord alors pour tout $\mathfrak{B}$*

$$p^{\mathfrak{B}}(d(i)=1) \leq \epsilon(l_{\mathfrak{B}}(i,r)+1)$$

Pour le schéma sans perte avec 1 comme valeurs initiales $p(d(i)=1)=1$, $\epsilon \geq \frac{1}{r+1}$.

On se fixe une éxécution γ, et on introduit le schéma de communication $past(\gamma,i)$ qui ne contient que le passé de i :

$$(i,j,k) \in past(\gamma,i) \iff (j,k) \leq_\gamma (i,r)$$

Pour $\mathfrak{B}$ un adversaire (γ, I), on fabrique $\mathfrak{B}'$ avec $(past(\gamma,i), I')$ où $I' =$ _____ complete me !

Par induction sur $l_{\mathfrak{B}}(i,r)$:

- ▷ $l_{\mathfrak{B}}(i,r)=0$, il existe j tel que $\neg((j,0) \leq_\gamma (i,r))$. On considère $\mathfrak{B}''$ qui consiste à $\mathfrak{B}'$ dans lequel il n'y a aucun message de j et vers j avec la valeur initiale 0.

$$p^{\mathfrak{B}'}(d(j)=1) = p^{\mathfrak{B}''}(d(j)=1) = 0$$

$$\left| p^{\mathfrak{B}'}(d(i)=1) - \underbrace{p^{\mathfrak{B}'}(d(j)=0)}_{=0} \right| \leq p^{\mathfrak{B}'}(j \text{ décide 0 et } i \text{ décide 1}) \leq \epsilon$$

$$p^{\mathfrak{B}}(d(i)=1) = p^{\mathfrak{B}'}(d(i)=1) \leq \epsilon$$

▷ $l_{\mathfrak{B}}(i,r) > 0$, il existe j tel que $l_{\mathfrak{B}'}(j,r) \leq l-1$.

$$p^{\mathfrak{B}'}(d(j)=1) \leq \epsilon(l_{\mathfrak{B}'}(j,r)+1) \tag{HI}$$

$$\left| p^{\mathfrak{B}'}(d(i)=1) - \underbrace{p^{\mathfrak{B}'}(d(j)=1)}_{\epsilon l} \right| \leq \epsilon$$

$$p^{\mathfrak{B}}(d(i)=1) = p^{\mathfrak{B}'}(d(i)=1) \leq \epsilon(l+1)$$

2 Consensus synchrone

2.1 Cadre

Soit $\Pi = \{p_1, \ldots, p_n\}$ un ensemble fini de processus. On suppose que les processus peuvent communiquer deux à deux (graphe de communication complet) et on se place dans le modèle synchrone :

- La vitesse des processus est connue
- Les délais de communication sont connus

L'éxécution d'un programme consiste en une succession de rondes dont les étapes sont :

- Construire un message m
- Envoyer le message
- Recevoir le message M émis par les autres processus

On dit qu'un processus est *byzantin* s'il peut dévier de la spécification de l'algorithme en cours d'éxécution. Une *panne* de processus peut consister soit en un arrêt, soit en une omission d'envoi / de réception de message. On note $F(r)$ l'ensemble des processus en panne à partir de la ronde r, avec pour conséquence que $F(r) \subset F(r+1)$. Attention, les processus corrects ne correspondent pas aux processus vivants (les bizantins étant encore vivant).

Problème du consensus : Il s'agit d'un problème d'accord, chaque processus a une valeur initiale et doit décider une *valeur de décision*. Ce choix irrévocable doit respecter les trois contraintes suivantes :

Accord : Si deux processus corrects décident, ils décident la même valeur

Terminaison : Tous les processus corrects décident

Validité : Si un processus décide, la valeur décidée a été proposée par un processus et si tous le processus corrects proposent la même valeur, ils doivent décider cette valeur.

2.2 Premier exemple

On modifie légérement la spécification de la validité :

Validité : Si un processus décide, la valeur lui a été proposée.

Algorithme en $t+1$ rondes : Il existe un algorithme déterministe qui, pour n processus, tolère t pannes, satisfait la spécification du consensus en synchrone et s'éxécute en $t+1$ rondes :

- À chaque ronde, on envoie les valeurs reçues précédemment
- Au bout de $t+1$ rondes, on décide la valeur minimale

Il y a une ronde sans panne, tout le monde a alors les même informations qui sont maximales (c'est à dire jamais plus d'information). Il n'y a plus de nouvelle informations ni de perte dans les rondes suivantes.

2.3 Peut-on détecter les pannes ?

On note t le nombre de pannes tolérées et f le nombre de pannes effectives. Si $f = 0$, on peut finir en 2 rondes : si on reçoit tous les messages, puis un message de tous les processus avec tous les messages, on décide. On a donc un algorithme en $\min(f + 2, t + 1, n - 1)$ rondes.

Theorem 2.1 (IPL, A simple bivalence proof that t-robust consensus requires $t + 1$ rounds)**.** *Dans un système synchrone de n processus, tout algorithme qui résout le consensus en tolérant $t < n - 1$ pannes a une éxécution en $t + 1$ rondes.*

La fin de cette sous-section est dédiée à la preuve de ce théorème.

Soit S un système avec au plus une panne par ronde. On se ramène au cas du consensus binaire : les seules valeurs de décision sont 0 ou 1. On considère les configurations obtenues à la fin de chaque ronde composée de l'état des processus vivants et de l'ensemble des processus crashés. Une *éxécution partielle* de k rondes n_k est une éxécution de l'algorithme jusqu'à la fin de la ronde k et on note C_k la configuration obtenue à la fin de cette ronde.

Une configuration C_k est 0-valente (resp. 1-valente) si pour toute éxécution (et donc toutes pannes), les processus décident 0 (resp. 1). On dit alors qu'elle est monovalente et bivalente dans le cas contraire.

Par l'absurde, on suppose qu'il existe un algorithme qui résout le consensus et termine en au plus t rondes dans S.

Lemma 2.1. *Toute éxécution partielle de $t - 1$ rondes n_{t-1} est monovalente.*

Supposons r_{t-1} bivalente. On étend r_{t-1} par une ronde sans panne qui décide puisque l'algorithme s'éxécute en t rondes. On peut supposer que la valeur décidée est 0 et on appelle alors r^0 cette éxécution. Comme r^{t-1} est bivalente, il existe une éxécution r^1 étendant r^{t-1} décidant 1 et qui a exactement un processus p qui est tombé en panne en n'envoyant pas de message à un autre processus c avec c correct. Il existe un autre processus c' correct. Mais Pourquoi ? Parce que $t < n - 1$. Soit $r^{1,0}$ (l'éxécution de shrodinger) identique à r^1 mais dans laquelle p envoie un message à c'. $r^{1,0}$ et r^0 sont indistinguables pour c' qui décide donc r^0. $r^{1,0}$ et r^1 sont indistinguables pour c donc il décide 1. Il n'y a donc pas accord sur l'éxécution $r^{1,0}$.

Lemma 2.2. *Il existe une configuration initiale bivalente.*

Considérons les configurations C^i dans laquelle tous les processus commencent avec la valeur i (et décide donc i). C^1 est 1-monovalente, C^0 est 0-monovalente et on a une suite C_i de configurations où les i premiers processus commencent avec 1 et les autres avec 0. Si le $i^{\text{ème}}$ processus tombe en panne à la première ronde, C_{i-1} et C_i sont indistinguables. En particulier, si toutes les configurations initiales sont monovalentes, il existe i tel que C_{i-1} soit 0-valente et C_i 1-valente ce qui est absurde.

Lemma 2.3. *Il y a une éxécution partielle de $t - 1$ rondes bivalente.*

Par induction sur k, le cas de base étant donné par le lemme précédent. Supposons donc $0 \leq k \leq t - 1$ et r_k bivalente. Si toutes les extensions de r_k sont monovalentes alors r_k suivi d'une ronde sans panne r^0_{k+1} est monovalente, on peut la supposer 0-monovalente, et il existe r^n_{k+1} 1-valente. Il y a donc un processus qui tombe en panne dans r^0_{k+1} et omet d'envoyer son message à $q_1, \ldots, q_n$. En partant de r^0_{k+1}, on construit r^i_{k+1} pour $i \geq 1$ qui est r^{i-1}_{k+1} dans laquelle p n'envoie pas de mmessage à q_i. Il existe donc un l tel que r^l_k et r^{l-1}_k soient de valence distincte et en mettant en panne q_l, ces deux éxécutions doivent nécessairement être bivalentes.

2.4 Consensus byzantin

On modifie légérement la spécification de la validité :

Validité : Si tous les processus corrects proposent la même valeurn aucun processus correct ne décide une valeur différente.

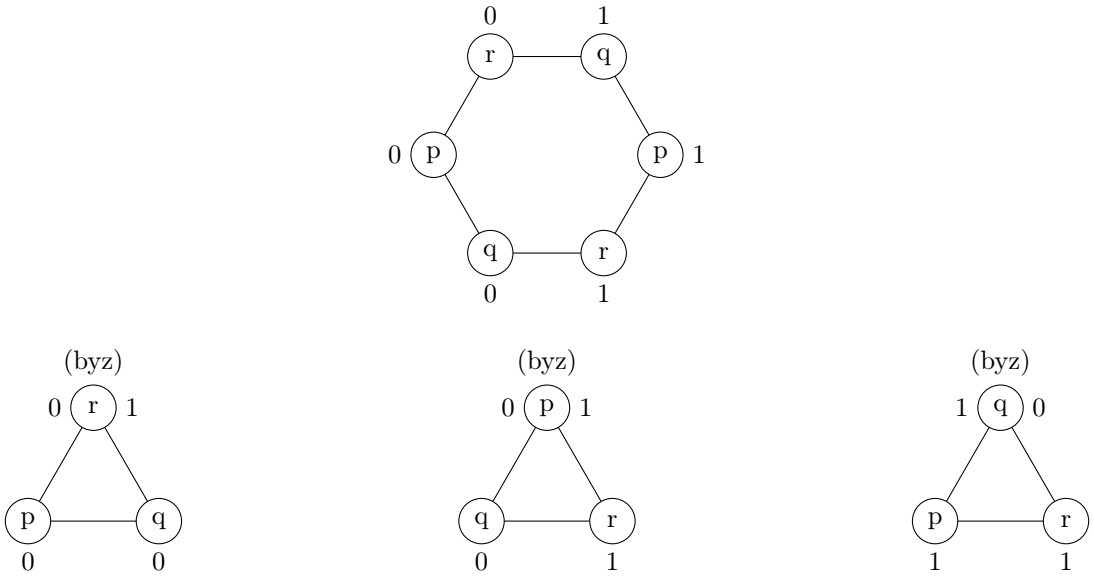
Pour valider cette spécification on a nécessairement besoin de $n > 2t$ (sinon considérer une configuration avec $\frac{n}{2}$ 0 et $\frac{n}{2}$ 1 et alterner le coté byzantin).

Il y a un lien du processus p au processus q : si q reçoit un message il connaît son expéditeur. L'authentification est un paramètre important : on peut résoudre le consensus dans le cas $n > 2t$ avec authentification, mais seulement dans le cas $n > 3t$ si on n'a pas d'authentification.

2.5 Consensus sans authentification

Theorem 2.2. *Tout algorithme pour n processus ne peut tolérer que t pannes avec $n > 3t$.*

Soit 3 processus p, q, r. On suppose que l'on dispose d'un algorithme qui résout le consensus en tolérant une panne. On considère le système suivant :



dans lequel il ne peut pas y avoir d'accord.

Puis pour $3 \leq n \leq 3t$ et un algo A qui résout le consensus en tolérant t byzantins, on réunit les n processus en 3 paquets disjoints que l'on simule avec 3 processus X, Y et Z ce qui nous donne un algorithme résolvant le consensus byzantin avec 1 panne ce qui est impossible par la construction précédente.

2.6 Terminating Reliable Broadcast (TRB)

On a un émetteur s qui veut diffuser un message (les processus ont connaissance de l'émetteur). Tous les processus corrects doivent délivrer un message en respectant les conditions suivantes :

Vaidité : Si S est correct et diffuse m, alors s délivre m.

Accord : Si un programme correct délivre un message m alors tous les processus corrects délivrent m.

Terminaison : Tous les processus corrects délivrent un message.

Intégrité : Tous les processus corrects délivrent un message au plus 1 fois.

Theorem 2.3. *Pour $n > 2t$, le problème du consensus et du TRB sont équivalents.*

Consensus $\Longrightarrow$ TRB : s envoie son message dans la $1^{\text{ère}}$ ronde. Soit p un processus, s'il a reçu un message m, il propose m. On éxécute un consensus sur les messages proposés et quand p décide x il le délivre.

TRB $\Longrightarrow$ Consensus : Chaque processus i fait un TRB en émettant son message initial v_i. Les processus ont donc une séquence $(v_1, \ldots, v_n)$ de messages disponibles et ils choisissent la valeur majoritaire (qui est correcte car $n > 2t$).

2.7 Mécanisme d'authentification

On considère que $n > 2t$. On cherche à construire un mécanisme d'authentification ayant les propriétés suivantes. Si x est une chaine et p un processus alors p peut signer x (ce que l'on note $p : x$) tel que

- Si q reçoit $p : x$, il peut extraire x
- Si p est correct, aucun processus ne peut envoyer un message contenant $p : x$ à moins que p ne l'ait envoyé dans le passé

Si $p_k : \ldots : p_1 : s$ est reçu par p, il peut donc à la fois extraire s et connaître les différents réémetteurs du message.

Definition 2.1. Un message reçu dans la ronde i est *valide* si et seulement si il est de la forme $m : p_1 : \ldots : p_i$ où p_i est l'émetteur et tous les p_k sont distincts. m est la valeur du processus correct.

message?

Code de p : Si p est l'émetteur (s) et veut diffuser m alors `extracted = relay =` $\{m\}$, sinon `extracted = relay =` $\varnothing$. À la ronde i, un processus envoie $\{m : p \mid m \in \texttt{relay}\}$ et reçoit les messages valides (jette les autres) de la ronde que l'on note $\mathcal{V}$. Puis il pose `relay` $= \varnothing$ et $\forall m : p_1 : \ldots : p_n \in \mathcal{V}$,

- si $m \notin$ `extracted`, `extracted` $= \{m\} \cup$ `extracted` et `relay` $=$ `relay` $\cup \{m : p_1 : \ldots : p_n\}$
- Si $i = t + 1$, si $|$`extracted`$| > 1$ alors `Faulty Sender` sinon délivrer m tel que $extracted = \{m\}$

2.8 Authentificated Broadcast ($n > 3t$)

Le problème de l'authentificated broadcast peut être présenter de la manière suivante. Un processus p émet un message m à la ronde k (`Broadcast`(p, m, k)). Les processus acceptent un message (p, m, k) à une ronde r. De plus certaines propriétés doivent être satisfaites :

Correction : Si p est correct et émet (p, m, k) à une ronde k, alors tous les corrects acceptent (p, m, k).

Non forgeabilité : Si un processus p correct n'a pas fait $Broadcast(p, m, k)$ alors aucun processus correct ne peut accepter (p, m, k).

Relai : Si un processus cirrect accepte (p, m, k)dans la ronde $r \geq k$ allors tous les processus corrects acceptent (p, m, k) au plus tard à la ronde $r + 1$.

On décompose chaque ronde en deux phases : la ronde k correspond donc aux phases $2k - 1$ et $2k$. Pour éxécuter `Broadcast`(p, m, k), p envoie (`init`$, p, m, k$) à tous à la phase $2k - 1$. Puis, à la phase $2k$, si un processus p' a reçu (`init`$, q, m, k$) de q à la phase $2k - 1$, alors p' envoie (`echo`$, q, m, k$). Si p a reçu (`echo`$, q, m, k$) d'au moins $n-t$ processus dans la phase $2k$ alors p accepte (q, m, k) ce qui assure la correction.

Pour une ronde, $r > k$, si au moins $n - 2t$ messages (`echo`$, q, m, k$) ont été reçus de la part de processus distintcts dans les phases précédentes et qu'aucun message (`echo`$, q, m, k$) n'a été envoyé auparavant, alors on envoie (`echo`$, q, m, k$). Si au moins $n - t$ messages (`echo`$, q, m, k$) on été reçus de la part de processus distincts dans k phases distinctes dans les phases précédentes, alors on accepte (q, m, k) si ça n'a pas encore été fait.

L'algorithme précédent vérifie éffectivement le **relai**. Si un processus correct p accepte (q, m, k) alors p a reçu $n - t$ `echo` de processus distincts à la phase x. Par conséquent, $n - 2t$ processus ont envoyé `echo` à la phase x.

À la fin de la phase x, tous les processus corrects ont reçus $n - 2t$ messages. Si un processus ne l'a pas encore fait, alors il doit envoyer `echo` à la phase $x + 1$.

$\rightsquigarrow$ Tous les processus corrects vont envoyer `echo` à une phase $\leq x+1$

$\rightsquigarrow$ Si $n-t$ message `echo` sont reçus, `accept` est émis à la phase $x+1$

Lemma 2.4. *Si un processus correct envoie* $(\texttt{echo}, p, m, k)$ *alors* p *doit avoir envoyé* $(\texttt{init}, p, m, k)$ *à au moins un processus correct en phase* $2k-1$.

Proof : Soit l la première phase à laquelle un processus correct q envoie $(\texttt{echo}, p, m, k)$:

Si $l > 2k$**,** alors q a reçu $n-2t$ messages `echo` de processus distincts, en particulier il a reçu un message `echo` de la part d'un processus correct et ce processus correct l'a envoyé à la phase précédente.

Si l $= 2k$**,** q a reçu un message $(\texttt{init}, p, m, k)$ et p a envoyé un message `init` à la phase $2k-1$

La **non-forgeabilité** est aussi vérifiée. Soit p un processus correct qui n'a pas envoyé $(\texttt{init}, p, m, k)$ à la phase $2k-1$. Si on suppose qu'un processus correct accepte (p, m, k) alors il a reçu $n-t$ messages $(\texttt{echo}, p, m, k)$ et au moins un processus correct a envoyé $(\texttt{echo}, p, m, k)$ ce qui implique que p a envoyé le message $(\texttt{init}, p, m, k)$ à au moins un processus correct, contradiction.

2.9 TRB pour un message $\in \{0, 1\}$

Code de p **:** Si p est l'émetteur alors `valeur` $= m$ sinon `valeur` $= 0$. Puis, pour $r = 1$ à $t+1$:

- Si `valeur` $= 0$ et p n'a pas émis $(p, 1, *)$ avant alors faire `Broadcast`$(\texttt{p}, 1, \texttt{r})$
- Si dans la ronde $r' < r$ finir code

L'algorithme précédent délivre effectivement un TRB correct :

Terminaison : il y a exactement $t+1$ rondes.

Intégrité : (chaque processus délivre au plus un message) évident.

Validité : Si l'émetteur s est correct et posséde initialement le message 1, alors `valeur` $= 1$, s éxécute un `Broadcast`$(\texttt{s}, 1, 1)$ et par correction de Authentificated Broadcast tous les corrects acceptent $(s, 1, 1)$ et délivrent donc la valeur 1. Si l'émetteur s correct posséde initialement le message 0, alors s n'émet jamais $(s, 1, 1)$. Par non-forgeabilité de l'Authetificated Broadcast, aucun processus correct n'accepte $(s, 1, 1)$, donc `valeur` $= 0$ pour chaque processus correct qui délivre alors 0.

Accord : Soit r la première ronde où `valeur` $= 1$ pour un processus correct. Si $r < t+1$, p a accepté $(p_k, 1, r_k)$ de r processus distincts et $p_1 = s$. Par la propriété de relai de l'Authentificated broadcast, tous les processus corrects ont accepté au plus tard à la ronde $r+1$ ces messages.

- Si $r+1 \leq t+1$ et ils reçoivent en plus le message de p (par correction de l'Authentificated broadcast) compléter
- Si $r = t+1$, p a accepter $(p_k, 1, r_k)$ de $t+1$ processus distincts à la ronde r. Parmi ceux-ci, il y a au moins un correct et ce processus a changé sa valeur de 1 à 0 à la ronde $r-1$ ce qui contredit la définition de r.

Theorem 2.4. *L'algorithme présenté ci-dessus permet de réaliser un TRB en* $t+1$ *rondes avec un Authentificated Broadcast.*

3 Consensus asynchrone

On se place maintenant dans le cadre des processus asynchrone où les communications sont asynchrone. Un résultat de Fischer, Linch et Patterson de 85 montre l'impossibilité du consensus (même binaire) en présence de pannes (crash), même en se limitant à au plus une panne dans un système asynchrone.

Remark 3.0.1. Le reliable broadcast est possible en présence de crash, il n'est donc pas équivalent au consensus dans le cadres asynchrone.

Atomic broadcast : On introduit la notion d'*atomic broadcast* : chaque processus diffuse un message et les processus doivent tous délivrer la même liste de message en préservant l'ordre. Les propriétés suivantes doivent être vérifiées : correct ?

Accord uniforme : deux processus ne peuvent pas décider différemment

Terminaison faible : au moins un processus termine.

Non trivialité : 0 et 1 sont des décisions possibles.

On se place dans le modèle de calcul suivant : Les messages émis et non consommés sont conservés dans un buffer. Un *pas de calcul* (atomique) d'un processus peut être :

- de prendre un message (qui lui est destiné) dans le buffer s'il y en a un
- d'envoyer des messages aux autres
- de changer d'état

Chaque message de la forme (m, o) où m est le contenu et o le destinataire est unique. Un processus est correct s'il fait une infinité de pas. Les processus sont déterministes (état + message).

Plus formellement, soit Π l'ensemble des processus, $\mathcal{M}$ l'ensemble des messages. On suppose que l'on a un message distingué $\lambda \in \mathcal{M}$ qui représente l'absence de message. Un processus $p = \mathcal{A}_p = (\Sigma_p, I_p, \mu_p, \delta_p)$ est un automate où Σ_p est l'ensemble infini des états, I_p l'ensemble des états initiaux, $\mu_p : \Sigma_p \times \mathcal{M} \to \mathcal{M} \times \Pi$ la fonction d'envoi de messages et $\delta_p : \Sigma_p \times \mathcal{M} \to \Sigma_p$ la fonction de transition. Une *configuration* c du système est une paire (S, M) où $S = (S_1, \ldots, S_n)$ est l'état courant de chaque processus et M est le contenu du buffer, c'est à dire une liste de message (m, o). On note $state(p, c)$ l'état S_p du processus p dans la configuration $c = (S, M)$ et $buf(p, c)$ l'ensemble des messages de M destinés à p. Une configuration c est initiale si $\forall p$ processus $state(p, c) \in I_p$. Étant donné un état initial, l'évolution du processus ne dépend que des messages e qu'il reçoit. Un message e est dit *applicable* à une configuration $c = (S, M)$ si $e \in M$ ou si $e = \lambda$. On note $e(c)$la configuration obtenue aprés éxécution du message applicable $e = (m, p)$ sur la configuration c :

$\rightsquigarrow$ $state(p, e(c)) = \delta_p(state(p, c), m)$, $state(q, e(c)) = state(q, c)$ si $q \neq p$

$\rightsquigarrow$ Les messages envoyés sont $\mu_p(state(p, c), m)$

$\rightsquigarrow$ Le nouveau contenu du buffer est donné par $M' = M \setminus \{e\} \cup \mu_p(state(p, c), m)$

Un schedule est une séquence fini ou infini de pas de calculs $e_1, \ldots, e_n, \ldots$ tel que chaque chaque message e_{i+1} soit applicable à la configuration obtenue par l'application des messages précédents : $(e_1, \ldots, e_n)(c) = (e_2, \ldots, e_n)(e_1(c))$. Une configuration c' est accessible depuis la configuration c s'il existe un schedule S applicable à c tel que $S(c) = c'$. Une configuration est accessible si elle est accessible depuis une configuration initiale I.

Soit S un schedule infini, p est *défaillant* pour S si p ne fait qu'un nombre fini de pas de calcul. Un schedule S est t-*admissible* pour c si :

1. S est applicable en c
2. Il y a au plus t processus défaillants dans S

3. Si le message (m, p) est dans le buffer et que le processus p n'est pas défaillant, alors il est consommé lors d'un pas de calcul de S (c'est à dire qu'il est présent dans S)

On note D_p^0 l'ensemble des états de décision 0 pour p et D_p^1 l'ensemble des états de décision 1 pour p. Ces ensembles sont disjoints et si $\sigma \in D_p^i$ alors tout état accessible depuis σ est aussi dans D_p^i

On peut maintenant reformuler les propriétés de l'atomique broadcast dans ce cadre plus formel :

Accord uniforme : $\nexists c$ accessible tel que $state(p, c) \in D_p^x$ et $state(q, c) \in D_q^{1-x}$

Non-trivialité : $\exists c \neq c'$ accessible tel qu'il existe p décidant 0 en c et q décidant 1 en c'

Terminaison faible : $\forall$

On appelle valence d'une configuration c l'ensemble $val(c)$ définit par

$$val(c) = \left\{ v \in \{0, 1\} \;\middle|\; \exists p, c' \text{ accessible depuis } c \text{ tel que } state(p, c') \in D_p^v \right\}$$

c est univalent si $|val(c)| = 1$ (0-valente ou 1-valente) et bivalente si $|val(c)| = 2$.

Remark 3.0.2. Si c est accessible et $\exists p$ tel que $state(p, c) \in D_p^v$ alors c est v-valent.

Theorem 3.1. *Le consensus est équivalent à l'atomic broadcast.*

Proof : Compléter

4 Détection de défaillance

Articles de référence : Failure detectors, Chandra et Toueg, JALM96.

Soit Π un ensemble fini de processus, $\mathcal{T}$ l'ensemble des temps discrets. Un schéma de panne est une fonction $F : \mathcal{T} \to \mathcal{P}(\Pi)$ où $F(\tau)$ représente l'ensemble des processus en panne au temps $\tau \in \mathcal{T}$. On suppose $\mathcal{F}(\tau) \subseteq \mathcal{F}(\tau + 1)$, c'est à dire que les pannes correspondent à des arrêt. On note $panne(\mathcal{F}) = \bigcup_{\tau \in \mathcal{T}} F(\tau)$ et $correct(\mathcal{F}) = \Pi \setminus panne(\mathcal{F})$. Un *environnement* est un ensemble de schémas de panne. Une histoire d'un détecteur de panne sur un certain domaine D est une fonction $H : \Pi \times \mathcal{T} \to D$ (en général $D = \mathcal{P}(\mathcal{T})$). Un *détecteur de panne/défaillance* $\mathcal{D}$ sur un domaine D pour un environnement $\mathcal{E}$ est une fonction qui à chaque schéma de panne $\mathcal{F}$ de $\mathcal{E}$ associe un ensemble d'histoires sur D, $\mathcal{D}(\mathcal{F})$.

On peut demander des propriétés :

- $\rightsquigarrow$ de complétude : est-ce qu'on suspecte bien les processus défaillants ?
- $\rightsquigarrow$ d'exactitude : est-ce qu'on a confiance en les processus corrects ?

Un détecteur de panne "parfait" et instantanné est $\mathcal{D}(\mathcal{F})(p, t) = \mathcal{F}(p, t)$.

Example : Prenons $D = \mathcal{P}(\Pi)$ comme ensemble de processus suspectés.

Complétude forte : $\forall \mathcal{F} \in \mathcal{E}, \forall H \in \mathcal{D}(\mathcal{F}), \exists \tau \in \mathcal{T}, \forall p \in panne(\mathcal{F}), \forall q \in correct(\mathcal{F}), \forall \tau' \geq \tau, p \in H(q, \tau')$
Tous les processus correct reçoit ultimement un ensemble dans lequel sont inclus les processus en panne.

Complétude faible : $\forall \mathcal{F} \in \mathcal{E}, \forall H \in \mathcal{D}(\mathcal{F}), \exists \tau \in \mathcal{T}, \forall p \in panne(\mathcal{F}), \exists q \in correct(\mathcal{F}), \forall \tau' \geq \tau, p \in H(q, \tau')$
Au moins un processus correct reçoit ultimement un ensemble dans lequel sont inclus les processus en panne.

Exactitude forte : $\forall \mathcal{F} \in \mathcal{E}, \forall H \in \mathcal{D}(\mathcal{F}), \forall \tau \in \mathcal{T}, \forall p, q \in \Pi \setminus \mathcal{F}(\tau), p \notin H(q, \tau)$ Les processus corrects ne sont jamais suspectés par un autre processus correct.

Exactitude faible : $\forall \mathcal{F} \in \mathcal{E}, \forall H \in \mathcal{D}(\mathcal{F}) \forall \tau \in \mathcal{T} \exists p \in \Pi \setminus \mathcal{F}(\tau) \forall q \in \Pi \setminus \mathcal{F}(\tau), p \notin H(q, \tau)$ Il existe un processus correct qui n'est jamais suspecté.

On peut aussi parler d'exactitude ultime, il suffit de remplacer le $\forall \tau \in \mathcal{T}$ par $\exists \tau' \in \mathcal{T}, \forall \tau \geq \tau'$.

On considère deux détecteurs :

P **parfait :** détecteur avec complétude et exactitude forte

$\Diamond S$ **:** détecteur avec complétude forte et exactitude ultime faible

Le pas d'un processus correspond à la succession d'opérations suivante :

- recevoir les messages
- faire un appel au Failure Detector
- changer d'état
- envoyer un message (optionnel)

Une éxécution d'un algorithme A qui utilise un Failure Detector$\mathcal{D}$ est décrit par le $5-upplet$ $\langle \mathcal{F}, H, I, S, T \rangle$ où $\mathcal{F}$ est le schéma de panne, H l'histoire de $\mathcal{D}$, I l'entrée, S la suite des pas de calculs, T les instants auxquels s'éxécutent les pas. Le $i^{\text{ème}}$ pas de calcul qui a lieu au temps $T[i]$ est représenté par $S[i] = (p, M)$ tel que $p \notin \mathcal{F}(T[I])$ et M l'ensemble des messages qui doivent avoir été envoyé par un processus auparavant. Les processus de $correct(\mathcal{F})$ doivent faire une infinité de pas et tous les messages émis vers un processus correct sont reçus.

5 Détecteur de défaillances

Partiellement synchrone : implémente un Failure Detector. On dl les solutions en asynchrone + Failure Detector.
Système synchrone : On a (éventuellement) des bornes connues sur le délai δ de communication et la vitesse des processus v (correspond grossièrement au calcul d'une instruction).

On se place dans un cadre où on suppose l'*existence* de telles bornes :

$$\exists T \exists b_1, b_2, B_1, B_2, \forall \tau > T, b_1 < \delta < b_2 \wedge B_1 < v < B_2$$

On suppose une liste de processus suspects telle que : Failure DetectorP $\equiv$ système synchrone

- Un processus correct et vivant n'est jamais suspecté par aucun autre processus (exactitude)
- Tout processus incorrect sera suspecté (complétude)

Ça marche pas. On fait un Failure Detector$\Diamond P$ tel que $\exists T, \forall \tau > T$ les conditions de P sont vérifiées.

Code de p

- Envoyer tous les η un message $(p,$ "I am alive")
- Initialisation : $Timeout[q] = 1$ $\forall q \in \Pi$ [[schema 1]]
- Mettre en route les timers avec pour échéance TimeOut.
  - Si $\exists q$ tel que le timer de q expire, output = output $\cup \{q\}$
  - Sur réception de $(q,$ "I am alive"), si $q \notin$ output, alors remettre en route le timer avec pour échéance TimeOut$[q]$; sinon, TimeOut$[q]++$ et remettre en route le timer avec pour échéance TimeOut$[q]$

Si on considère un processus x en panne, il existe un temps T tel que tous les messages de x ont été reçus par p. [[schema 2]]

x sera à partir de $T + \text{TimeOut}_p^T[x]$ dans output_p : on a la complétude.

Exactitude Soit x un processus correct. À partir d'un instant T, dans tous les intervalles $[\tau, \tau + \eta + b_2]$, p reçoit un message de x.
Le $\text{TimeOut}_p[x]$ augmente donc jusqu'à ce que x ne soit plus suspecté.

Moralité : Si on a un tel système, en implémentant $\Diamond P$, si on a une majorité de corrects, on a un consensus.

Cependant, $\Diamond S$ suffit : il existe un processus correct pour lequel il existe T et les bornes b_1, b_2, B_1, B_2. Il suffit de prendre le processus correct pour lequel on a des bornes pour les preuves : $\Diamond S$ et une majorité de correct permet de faire le consensus.

Theorem 5.1. *FLP Dans un système asynchrone, il n'existe pas d'algo de consensus* ***qui tolère une panne****. Un système partiellement synchrone tel qu'il existe un lien pour lequel des bornes de communication existe suffit pour réaliser le consensus en tolérant une panne.*

[[Alors je vous ai dit que. Donc un avantage du Failure Detectorenfin plutôt un intéret du Failure Detectorc'est que d'une part on peut les comparer parceque certains sont on peut voir la force de l'un par rapport à l'autre et on peut trouver le plus faible Failure Detectorpour résoudre un problème. Alors plus faible Failure Detectorpour résoudre un problème ça veut dire que ce Failure Detectoril résoud le problème et que en plus tout Failure Detectorqui le résout permet alors ce Failure Detectoron va l'appeler D et alors plus fort ben comparaison c'est à partir de ceclui-ci on peut implémenter D. Donc le Failure Detectoril caractérise le problème et donc on va pouvoir aussi classifier les problèmes suivant le Failure Detectorqui permet de les résoudre. D'un autre coté le Failure Detectoron peut l'implémenter dans un système partiellement synchrone et donc ça permet aussi d'obtenir des d'obtenir alors pas une équivalence parceque on peut pas [...]]]

Failure Detector- comparer - le plus faible Failure Detectorpour résoudre P - ce FD D résoud le pb P - tout FD qui résoud P est plus fort que D (on peut implémenter D à partir de lui)

6 Atomic Snapshot

Un tableau où le processus i peut modifier la case i et lire le tableau. On a une primitive `update(T v)` et on veut implémenter une opération `T[] scan()`. La spécification séquentielle vérifie que le `scan` retourne le tableau tel que la i^e valeur du tableau est celle du dernier `update` du processus p_i où la valeur initiale si p_i n'a pas fait de `update`.

Première tentative : faire une simple boucle (avec `update(T v)` par p_i donne `tab[i] = v`)

```
T[] Collect() =
    for (i = 0 ; i < n ; i++)
      T[i] = Tab[i]
    return T ;
```

Cette solution ne marche pas, elle peut retourner des valeurs de tableau qui n'ont jamais existée : Si `scan` lit $\perp$ depuis la case 1, que p_1 écrit 1, p_2 écrit 2 puis `scan` lit 2 depuis la case 2 ce qui résulte en le tableau $[\perp; 2]$.

Seconde tentative : on réitère l'opération précédente jusqu'à ce qu'elle retourne consécutivement les memes valeurs. En effet, si deux `collect` successifs retournent les memes valeurs alors n'importe quel point de linéarisation entre le premier et le second `collect` est valide à condition que l'on interdise à un processus de réécrire la meme valeur, on rajoute donc un compteur dans chaque processus et on modifie l'update afin d'écrire des paires (compteur, valeur) avant d'incrémenter le compteur. Cet algorithme n'est pas wait-free

puisqu'un processus qui fait des `update` répétés peut empécher un autre processus de finir son scan mais il est non-blocking puisqu'à tout instant soit il existe un processus qui fait un `update` soit tous les processus font un scan et il y en a au moins un qui termine.

Troisième tentative : on veut un algorithme wait-free. On commence par traficoter l'opération d'`update` :

```
update(T v) =
    compteur++ ;
    snapshot = scan() ;
    Tab[i] = (compteur, snapshot, v)
```

Puis le collect devient :

```
T[] scan() =
   boolean[] moved;  // initialisé à false
   bool unchanged = true ;
   A = collect() ;
   while true
       B = collect() ;
       for (int j = 0 ; j < n ; j++)
           if (A[j].compteur != B[j].compteur)
               unchanged = false
               if  (moved[j]) return B[j].snapshot
               else moved[j] = true
       if (unchanged)
           T[] result ;
           for (int i = 0 ; i < n ; i++)
               result[i] = A[i].valeur ;
           return result ;
       A = B ;
```

Si 2 `collect` successif sont identiques, on appelle cela un clean double collect.

Lemma 6.1. *Si un* **`scan`** *fait un clean double collect alors il termine et les valeurs retournées sont les valeurs du tableau à un point de l'éxécution.*

En effet, aucun registre n'a été mis à jour entre les deux collect et n'importe quel point entre ces deux opérations est valide.

Lemma 6.2. *Si un processus p_i observe* 3 *valeurs distinctes à une case j lors d'un* **`scan`***, alors la valeur du snapshot à la case j est un* **`scan`** *ayant commencé aprés le premier* `collect` *et avant le dernier.*

En effet, si l'on appelle c_1, c_2 et c_3 les trois collectes, il y a un `write` w_1 entre c_1 et c_2 puis un autre `write` w_2 entre c_2 et c_3, donc il y a un `scan` entre w_1 et w_2, donc entre c_1 et c_3.

Lemma 6.3. **`scan`** *termine et les valeurs retournées par un* **`scan`** *sont un état des registres entre le début et la fin de son éxécution.*

Soit `scan` termine par un clean double collect (qui est valide par le lemme 1), soit au bout de $3(n-1)$ `collect` on est dans le cadre du lemme 2 (un processus a fait au moins deux écritures) dont le scan est valide par hypothése d'induction.

Propriété de l'atomic snapshot : quand le processus ne font qu'une mise à jour au plus (`update(v) ; sw = scan() ;`), on a

1. $v_i \in sw_i$
2. $\forall i, j, sw_i \subseteq sw_j$ ou $sw_i \supseteq sw_j$

7 Renaming

On considére $\Pi = \{p_1, \ldots, p_n\}$, où chaque processus p_i porte un nom $old_i \in \mathcal{M}$. Le but est d'attribuer un nom new_i à chaque p_i dans un sous-ensemble de $\mathcal{M}$ tel que l'on ait les propriétés suivantes :

Terminaison : tout processus correct décide un nouveau nom

Unicité : si $i \neq j$ alors $new_i \neq new_j$

Validité : $\forall i, new_i \in \mathcal{M}$

Anonymat : le nouveau nom est indépendant de l'index du processus

On cherche à obtenir un ensemble $\mathcal{M}$ petit. On suppose que $\mathcal{M}$ est totalement ordonné.

Algorithme wait-free avec $|\mathcal{M}| = 2n - 1$ Initialement, $\forall i, v[i] = \perp$. Pour p_i :

```
s = 1
while true
    update(< oldi, s >)
    v = scan()
    if (exists j != i / snd v[j] = s)
       X = { t / exists j, snd v[j] = t }
       F = M \ X
       Y = { o / exists j, fst v[j] = o}
       r = rang(oldi, v)
       s = X[r]
    else newi = s
```

1. La validité est évidente quitte à identifier $\mathcal{M}$ avec des entiers
2. Pour l'unicité si $i \neq j$ ont tous les deux obtenus $new_i = new_j = s$, soit v_i et v_j leurs derniers snapshots respectifs. Pour le processus i, $v_i[i]$ contient $\langle old_i, s\rangle$ et il n'y a aucun $k \neq i$ tel que $v_i[k] = \langle -, s\rangle$, on peut linéariser et supposer i avant j et donc $v_j[i] = \langle old_i, s\rangle$ d'où la contradiction.
3. Par l'absurde, soit α une éxécution où des processus corrects ne terminent pas. Soit A l'ensemble des processus corrects qui ne terminent pas dans α. Soit α' un segment initial de α tel que tous les processus qui terminent dans α ont terminé, tous les processus faulty sont morts et tous les processus de A ont fait un update aprés les deux événements précédents. Dans α', les rangs de chaque processus de A dans Y ne change plus et les rangs sont distincts. Soit NF les nouveaux noms utilisés par les processus qui ont un nouveau nom dans α. Soit $L = M \setminus NF = \{z_1, \ldots, z_r, \ldots\}$ les noms libres (aprés α'). Soit p_i de rang r minimal dans A. Aprés α', aucun processus ne choisira un élément de $\{z_1, \ldots, z_r\}$ car le rang de $p_j \in A$ pour $j \neq i$ est $> r$ et X lu par p_j contient NF, un nouveau nom $p_j \notin \{z_1, \ldots, z_r\}$. Dans X, plus aucun $\{z_1, \ldots, z_r\}$ ne sera présent et p_i choisira z_r comme nouveau nom.

Combien de noms utilise-t-on ? On peut prendre comme nouveaux noms $\mathcal{M} = \{1, \ldots, 2n - 1$ car le rang est au plus n et dans F il y a au moins n noms $\rightsquigarrow$ la nouvelle identité est dans $\mathcal{M}$.

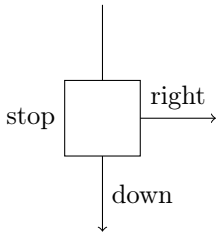
Extension : t résilient, renomage avec $n + t$ noms Pour p_i :

```
s = Bot
while true
   update(< oldi, s, false >)
   v = scan()
   if (exists j != i / v[j] = < _, s, _> ou s = Bot)
```

```
       r = rang(oldi, {t / exists j, v[j] = < t, s, false>, s != Bot})
      if (r <= t+1)
          s = {u / exists j, v[j] = < _, u, _> }[r]
   else
      newi = s
      update (< oldi, s, true>)
```

Cet algorithme est wait-free et fait un renommage en $2n-1$. Un résultat de Herlily montre qu'il est impossible de faire un renommage avec moins de $2n-2$ noms (et presque toujours pour moins de $2n-1$). Cet algorithme est adaptatif : avec k participants, on a un renommage en $2k-1$ noms.

Splitter :



Le splitter retourne soit stop, right ou down en vérifiant les deux propriétés suivantes :

solo : Si un seul processus invoque le splitter alors stop.

concurrence : si x processus invoquent le splitter, au plus $x-1$ processus obtiennent right, au plus $x-1$ processus obtiennent down et au plus un processus obtient stop.

On peut maintenant implémenter l'algorithme suivant qui nécéssite $\theta(n^2)$ splitter (et donc noms) mais un temps seulement linéaire :

```
new_name()
    d = 1
    r = 1
    move = down
    while (move != stop) do
        move = splitter()
        if move = down
            d = d + 1
        else if movei  = right
            r = r + 1
    return (d + r - 2)(d + r - 1)/2 + r
```

Implémentation du splitter :

```
closed = false

splitter()
    last = mon_nom
    if (closed)
        return right
    else
        closed = true
        if last = mon_nom
```

```
        return stop
    else
        return down
```

Vérifions que les propriétés requises sont présentes :

Solo : clair

Concurrence : Si x processus invoquent splitter :

- Au plus $x-1$ processus vont à droite car il faut au moins un processus qui éxécute `closed = true`
- Au plus $x-1$ vont en bas car si aucun ne va dans right, le dernier qui passe a son nom dans `last`
- Au plus un va dans stop car les processus passant aprés par `last = mon_nom` ont `closed = true` et les autres n'ont as `last = mon_nom`