Rapport de stage

Boubacar Kane - M2 AMIS

4 septembre 2019

Table des matières

1 Remerciements

A l'occasion de ce rapport je tiens d'abord à remercier l'équipe du Département Informatique de Télécom-SudParis, notamment mon tuteur de stage, M. Pierre Sutra, Maître de conférence à Télécom-SudParis, pour son accompagnement tout au long de cette expérience, ses conseils, sa patience ainsi que sa pédagogie.

2 Introduction

Le but de l'algorithmique répartie est de partager la charge de travail entre plusieurs unitées de calcul afin de pouvoir améliorer les performances d'une application. Cette technique peut présenter un inconvénient car il faut pouvoir assurer une certaine coordination entre les processeurs. Puisque les données qui sont partagées ne mutent pas de la même façon dans les différentes unitées de calcul, il est nécessaire de maintenir une certaine consistance entre les données. Une forte consistance permet d'avoir une application qui se déroule à peu près de manière séquentielle, presque de la même manière que si l'algorithme avait été effectué par une seule unitée de calcul [5]. Mais dans ce cas on perd en efficacité. Avoir une consistance faible améliore considérablement les performances et la mise à l'échelle cependant, cela peut être moins évident à programmer. Trouver le juste milieu semble être la meilleure solution.

Dans une même application, on peut vouloir que des données soientt plus ou moins consistante au cours du temps, c'est pour cela que, au cours de ce stage, nous avons introduit la notion *d'objets dégradables*. Un objet dégradable est un objet que l'on va altérer au cours d'un programme de façon à pouvoir modifier le critère de consistance entre les données.

Dans un premier temps, nous présentons le modèle que nous avons utilisé pour introduire les objets dégradables au cours de notre étude. Ensuite, nous mettons en avant le concept de *pouvoir de consensus* qui permet de hiérarchiser les objets. Et enfin, nous présentons un exemple d'objet dégradable.

3 Modèle de système

Pour représenter les objets que l'on a manipulé pendant tout le stage, nous avons décidé de nous baser sur le modèle des automates d'entrés et sorties (*I/O Automata*) [1].

Le modèle I/O Automata est un modèle qui permet de représenter un système asynchrone dans lequel les différentes entitées communiquent via des *actions* qui permettent de changer l'état des machines. Il y a trois types d'actions : *entrées*, *sorties* et *internes*. Les actions entrées et sorties sont celles qui permettent la communication et les actions internes ne sont vues que par l'automate qui l'invoque. Les actions entrées ne sont pas sous le contrôle de l'automate qui les reçoit, tandis qu'il spécifie lui-même les actions sorties et les actions internes. Dans un système à passage de message, un automate P_i qui veut communiquer avec un automate P_j va effectuer une action sortie $send(m)_{(i,j)}$. De son côté, l'automateP_j va effectuer une action entrée $receive(m)_{(j,i)}$. Afin d'assurer la transition entre une action *send()* et une action *receive()*, on utilise un système de message via un automate intermédiaire $C_{i,j}$ qui sert donc de canal de communication. Un exemple d'une communication est illustré dans la Figure 1.

Dans la suite du rapport, nous considérerons que les processeurs, registres et autres objets seront implémentés suivant un modèle I/O Automata.

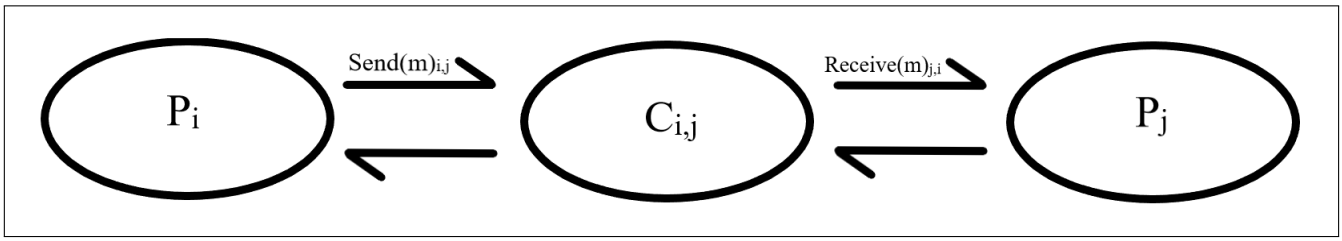

FIGURE 1 – Exemple d'une communication entre deux automates

4 Pouvoir de consensus

Dans cette partie, je vais présenter et expliquer la notion de pouvoir de consensus que l'on a utilisé durant tout le stage pour définir le niveau de dégradation des objets que l'on a étudié.

4.1 Implémentation sans-attente

Dans un système à mémoire partagée, on retrouve souvent des sections dites critiques dans lesquelles un seul processeur peut effectuer des calculs à la fois, les autres devant attendre leur tour. Cependant il est possible que le processeur qui est dans la section critique tombe en panne, ou bien ralentisse pour une quelconque raison, auquel cas les autres processeurs ne pourront terminer leurs exécutions. Dans ce cas, on dit que l'implémentation du système n'est pas bloquante.
Une implémentation sans-attente d'un système garantit que chaque processeur peut terminer son exécution en un nombre fini d'étapes et ce, peu importe l'attitude des autres processeurs (faute ou ralentissement). Cette implémentation garantit que le système est tolérant aux fautes car un processus est assuré de finir une tâche même s'il existe des fautes non détectées. Dans un système contenant deux objets X et Y partagés, on veut savoir s'il existe une implémentation sans-attente de X par Y.

Comme il n'est pas évident de montrer qu'il n'existe pas d'implémentation sans-attente de X par Y, les objets ont été classés par niveau de tel sorte que l'on puisse affirmer qu'un objet de niveau L ne puisse pas implémenter un objet de niveau supérieur. Ainsi, à chaque objet est associé un *numéro de consensus* qui indique le nombre maximum de processeurs pour lequel l'objet peut résoudre le problème de consensus. Dans la prochaine sous-section, nous expliquons ce qu'est le problème du consensus et nous donnons un exemple d'algorithme qui résout ce problème.

4.2 Problème de consensus

Le problème de consensus est un problème classique d'algorithmie distribuée. Dans les systèmes concurrents on a des *objets partagés*, qui sont des structures de données qui sont

partagées par des processus concurrents. Pour gérer la consistance de ces données, on veut que les processus puissent se mettre d'accord sur une valeur qui a été proposée par l'un d'entre eux. Plus formellement, on utilise un **protocole de consensus**. Un protocole de consensus doit être :

— **Consistant** : Des processus différents ne peuvent pas se mettre d'accord sur une valeur différente.

— **Sans-attente** : Les processus doivent se mettre d'accord après un nombre fini d'étapes.

— **Valide** : La valeur sur laquelle les processus se mettent d'accord est une valeur d'entrée.

Il existe deux types de consensus : le consensus général et le consensus binaire. Pour résoudre le problème de consensus général, les processus doivent se mettre d'accord sur une valeur quelconque et pour le problème de consensus binaire ils doivent se mettre d'accord sur une valeur binaire (0 ou 1). Il est possible de résoudre le problème de consensus général en faisant une succession de consensus binaires si la valeur quelconque est représentée en binaire. Un objet de consensus admet une opération unique qui est *propose(v)*. A titre d'exemple, un protocole de consensus simple serait que chaque processeur propose sa valeur à l'objet, et qu'ils décident tous la première valeur qui a été proposée (Algorithme 1). Dans cet algorithme, on doit considérer que le bloc *if* est effectué de manière atomique sinon deux processeurs peuvent avoir un problème de concurrence et décider de deux valeurs différentes.

Algorithm 1 $Propose(obj : object, v : value)$**returns**$(value)$

Require: $v \in \mathbb{N}$
  if $obj.val = \perp$ **then**
    $obj.val \leftarrow v$
  end if
  return $obj.val$

Cet algorithme est bien consistant : le premier processus à inscrire sa valeur dans la variable *obj.val* décide cette même valeur et chaque processeur qui va essayer d'inscrire une valeur dans l'objet après que la variable *obj.val* ait été modifiée ne pourra pas inscrire sa propre valeur. De ce fait il décidera la valeur qui était déjà inscrite. Il est aussi sans-attente car les processeurs ne sont pas dépendants et il n'y a pas de boucle dans l'algorithme, donc il se termine. L'algorithme est aussi valide : le premier processus va décider sa propre valeur, ce qui est bien une valeur d'entrée et tous les autres processus décideront la valeur du premier processus. Donc tout le monde décide bien une valeur qui est une valeur d'entrée. En fonction de l'objet que l'on utilise, on peut résoudre ou non le problème de consensus pour un nombre de processeurs donnés. On dit qu'un objet a un nombre de consensus de n s'il peut résoudre le problème de consensus dans un système à n processeurs.

Il est donc intéressant de voir comment a été calculé le nombre de consensus pour les objets courants.

Avant de voir toutes ces preuves, il est nécessaire de définir le modèle du système que l'on va utiliser. L'**état** d'un système correspond à la valeur des variables dans tous les objets du système ainsi que tout les messages qui sont en transit, dans notre modèle cela correspond aux actions *entrées*, *sorties* et *internes*. On dit qu'un système est dans un état **x-valent** (respectivement y-valent) si à partir de celui-ci, n'importe quelle exécution amène le système à décider la valeur x (respectivement y). On dit qu'un état est **bivalent** si depuis cet état, on peut encore décider la valeur x ou y. On dit qu'un état est **décisionnel** si à partir de cet état, la prochaine action amène le système dans un état univalent.

4.3 Registre atomique (lecture/ecriture)

Il a été montré que le registre atomique ne peut pas résoudre le problème de consensus pour deux processeurs ou plus [4]. Cette preuve a été construite par l'absurde et ressemble à celle utilisée par Fischer et al. [3]. On admet donc qu'il existe un protocole qui résout le problème de consensus pour deux processeurs ou plus. Comme les processeurs peuvent proposer des valeurs de départ différentes, on peut dire que l'état initial du système est bivalent. Soit P, un processeur qui exécute une série d'opérations jusqu'à ce que le système atteigne un état décisionnel où la prochaine opération amène le système dans un état x-valent. On sait que P arrive forcément à cet état car le protocole est sans-attente donc il se termine. Soit Q, un processeur qui exécute une série d'opérations qui amènent le système dans un état similaire. Il existe un état s à partir duquel les prochaines opérations de P et de Q amènent le système dans un état univalent. Supposons qu'une action du processeurs P amène le système dans un état x-valent et qu'une action du processeur Q dans un état y-valent. Considérons maintenant les différents scénarios :

— P lit un registre partagé en premier, Q effectue un calcul : Une fois que le processeur P a lu dans le registre, il arrive dans un état s' qui est un état univalent (x-valent étant donné que c'est le processeur P qui effectue l'opération de lecture). Cependant, les états s et s' ne diffèrent que dans l'état interne de P. Ainsi, si une opération de Q à partir de l'état s amène le système dans un état y-valent, c'est aussi le cas à partir de l'état s'. C'est une contradiction car l'état s' est censé être x-valent.

— P et Q écrivent dans deux registres différents : Si P écrit en premier, le système se retrouve dans un état x-valent. Si Q écrit en premier, le système se retrouve dans un état y-valent. Pourtant, le système doit se retrouver dans le même état que P ait effectué l'opération d'écriture en premier et Q en second ou que ce soit l'inverse étant donné qu'ils écrivent dans deux registres différents. Or, un état ne peut pas être x-valent et y-valent à la fois, il y a donc une contradiction.

— P et Q écrivent dans le même registre : On suppose que P écrit en premier, amenant ainsi le système dans un état s' étant x-valent. Soit s" l'état y-valent du système après

que l'écriture du processeur Q soit écrasée par l'écriture du processeur P. Les états s' et s" ne diffèrent que dans l'état interne du processeur Q par conséquent, le processeur P voit le système dans un état x-valent et le processeur Q voit le système dans un état y-valent, ce qui est aussi une contradiction.

On peut donc affirmer que le registre atomique a un nombre de consensus de 1.

4.4 L'objet *snapshot*

Dans un système distribué et asynchrone, il peut être intéressant de voir l'état d'un système entier à un instant donné. Cependant, comme chaque unité de calcul avance à son rythme, entre le début du scan du système et la fin du scan, l'état du système peut changer. Avec l'objet *snapshot* il est possible de prendre une "photo" des registres à un temps donné. Cette opération est possible car l'objet est atomique : on peut écrire l'exécution de l'opération de l'objet de manière séquentielle tel qu'il n'y a pas d'opération effectuée entre l'invocation et la fin de l'exécution.

Soit W un domaine composé de variables qui ont pour valeurs initiales w_0. Soit V un domaine contenant des vecteurs d'éléments de W de taille m. On définit v_0 comme étant un vecteur composé uniquement d'éléments w_0. Un objet *snapshot* T est définit par un ensemble de vecteurs de V [6]. L'objet snapshot comporte deux opérations : l'invocation $update(i, w)$ et l'invocation $snap()$. L'invocation $update(i, w)$ permet de modifier le composant i du vecteur $(1 \leq i \leq m)$ qui est mis à jour avec la valeur w et renvoie une réponse *ack*. On peut aussi écrire cette invocation comme suit : $update(w)_i$. L'invocation $snap()$ quant à elle renvoie la valeur du vecteur en entier.

On définit une face externe de l'objet qui contient n ports avec $n = m+p$, où p est un entier positif. Les processus y accèdent à des entrées différentes. Les ports de 1 à m sont les ports réservés aux invocations $update()$ et les ports de $m + 1$ à n (les p derniers ports) sont ceux réservés aux invocations $snap()$. Une représentation de cette interface externe est présentée dans la Figure 2.

L'algorithme utilise m variables *swmr* (*single writer multiple reader*) $x(i)$ ($1 \leq i \leq m$). Un processeur i est connecté au port i et peut uniquement écrire dans la variable $x(i)$ mais il peut lire dans toutes les autres variables *swmr*. Les variables $x(i)$ contiennent un élement de W ainsi que plusieurs variables nécessaires pour l'algorithme. Ainsi, un processeur qui exécute l'opération $update(w)_i$ sur le port i va inscrire la valeur w dans la variable $x(i)$. Un processeur qui exécute l'opération $snap()$ va renvoyer l'ensemble des variables $x(i)$ du système avec pour condition qu'elles aient co-existées à un moment donné. Pour réaliser cela, l'exécution est la suivante :

Un processeur i qui effectue une invocation $update(w)_i$ va inscrire la valeur qu'il veut dans une variable $x(i)$ et lui associer un *tag* unique. Un autre processeur qui veut effectuer une

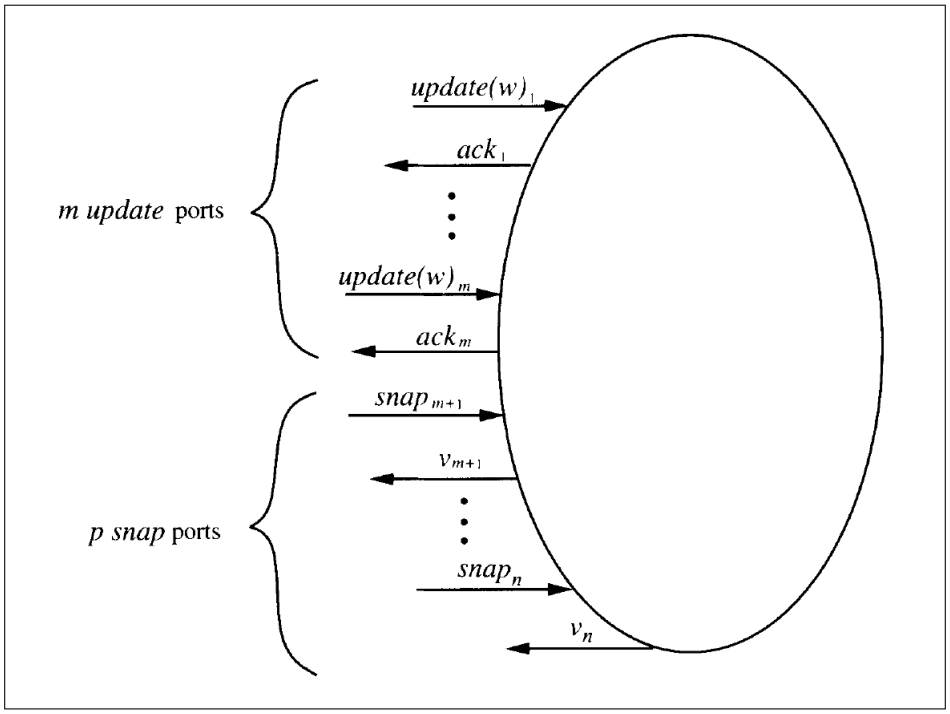

FIGURE 2 – Représentation de l'interface externe de l'objet *snapshot*

opération *snap*() va lire deux fois l'ensemble des variables. Si dans les deux ensembles de lectures la valeur des *tags* est identique pour chaque variable $x(i)$, alors il sait que les deux ensembles de lectures représentent un ensemble de variable qui a co-existé pendant l'opération *snap*(). Il a notamment existé après la fin du premier ensemble de lectures, et avant la fin du second ensemble de lectures. L'algorithme qui permet de réaliser cette opération est simple. Chaque processeur qui va effectuer l'invocation $update(w)_i$ va associer un *tag* unique en partant de 1 et en incrémentant ce *tag* pour chaque invocation *update*() exécuter sur la variable $x(i)$. Un processeur qui exécute l'opération *snap*() va lire l'ensemble des variables jusqu'à ce qu'il y ait deux ensembles de lectures avec un *tag* identique pour chaque variable. Ensuite, le processeur renverra le deuxième ensemble de lectures. Cependant, cet algorithme peut ne pas terminer dans le cas où entre chaque ensemble de lectures, un processeur fait une opération *update*, auquel cas l'algorithme n'est pas sans-attente.

Chaque processeur qui veut effectuer une opération *update* va d'abord effectuer un *snap embarqué* qu'il associera à cet update. Un *snap embarqué* est exécuté de la même manière qu'un snap et est par ailleurs stocké dans la variable $x(i)$, dans un champs *view*.

Pour avoir un algorithme sans-attente, on utilise l'observation suivante : quand un processus va voir quatre tags différents (tag_1, tag_2, tag_3 et tag_4) pour une même variable $x(i)$ alors il sait qu'une opération $update_i$ a commencé après le début l'opération *snap*() et avant sa fin. Il sait que cette opération $update_i$ est celle associée au tag_3 car, elle a forcément eu lieu après le début de l'opération *snap*(). En effet, les tags associés aux deux updates précédents ont été

vus par le processeur faisant le snap. Et l'opération $update_i$ a forcément eu lieu avant la fin de l'opération $snap()$ car le processeur voit aussi l'opération update associé au tag_4.

L'algorithme 2 présente l'implémentation de l'invocation $snap()$. La variable $reads$ est un tableau à deux dimensions qui conserve les variables x. Le vecteur $reads[i]$ contient l'ensemble des variables x lues lors de la $i^{ème}$ lecture. La fonction $equals$ prend deux vecteurs en argument et renvoie : vrai si les deux vecteurs sont identiques (si leurs champs tag sont les mêmes pour chaque variable), ou faux si ce n'est pas le cas.

Algorithm 2 *snap*()**returns** *(vector of values)*

```
for i ← 0; i < 4; i++ do
  for j ← 0; j < n; j++ do
    reads[i][j] ← x(j)
  end for
  if (i > 0) ∧ equals(reads[i − 1], reads[i]) then
    return reads[i]
  end if
  if i = 3 then
    for j ← 0; j < n; j++ do
      if reads[0][j] = reads[1][j] = reads[2][j] = reads[3][j] then
        return reads[2][j].view
      end if
    end for
  end if
end for
```

Un processeur qui termine l'exécution de l'algorithme après avoir vu quatre tags différents pour une même variable $x(i)$ va renvoyer le *snap embarqué* associé au troisième *tag* de cette variable $x(i)$. Cet algorithme se termine car au bout de $3m + 1$ ensembles de lectures, soit un processeur aura vu deux ensembles de lectures identiques, soit il aura vu quatre tags différents pour une même variable $x(i)$. D'après Herlihy [4], il n'existe pas d'objet possédant un pouvoir de consensus de n pouvant implémenter un autre objet ayant un pouvoir de consensus strictement supérieur à n. L'objet *snapshot* étant composé uniquement de registres atomiques avec un pouvoir de consensus de 1, on peut affirmer qu'il a aussi un pouvoir de consensus égale à 1.

4.5 Les objets *Read-Modify-Write*

Un objet *read-modify-write* ou RMW est défini de cette manière : l'objet contient un registre r et une fonction f qui prend en entrée une valeur et qui renvoie une valeur. On a accès à cet objet *read-modify-write* via l'opération $RMW(r, f)$ (Algorithme 3). Si la fonction f est une fonction identité, alors l'opération $RMW(r, f)$ est simplément une opération de lecture. Une opération $RMW(r, f)$ est dite non-trivial si la fonction f n'est pas une fonction identité. Cette opération est exécutée de manière atomique.

Algorithm 3 $RMW(r : register, f : function)$**returns**$(value)$

```
previous ← r
r ← f(r)
return previous
```

Plusieurs objets bien connus sont des objets *read-modify-write*. Voici une liste non-exhaustive des objets étant dans cette classe :

- **test & set** : cette instruction met la valeur 1 dans le registre et renvoie l'ancienne valeur, si elle était de 0, l'instruction est un succès, si elle était de 1, c'est un échec.
- **compare & swap** : cette instruction compare la valeur dans le registre à une valeur passé en argument. Si les deux valeurs sont identiques, l'objet renvoie un 1 et change la valeur qu'il y avait dans le registre, sinon il renvoie O.
- **fetch & increment** : cette instruction incrémente de 1 la valeur déjà inscrite dans le compteur et renvoie la valeur qui été inscrite dans le registre.
- **fetch & add** : cette instruction est quasiment identique à la précédente, à la différence près que l'on incrémente le registre d'une valeur quelconque.

Il a été démontré que les objets *read-modify-write* avait au moins un pouvoir de consensus de 2. La preuve est la suivante :

La fonction f n'est pas une fonction identité, ainsi, il existe une valeur v telle que $v \neq f(v)$. Soit deux processeurs P et Q voulant résoudre le problème de consensus en proposant une valeur via une fonction *decide*. Admettons que le processeur P propose la valeur 1 et le processeur Q la valeur 0. Leurs propositions sont stockées dans un tableau partagé $prefer$ initialisé à $\perp$. Ils partagent également un registre r initialisé à v. Chacun des deux processeurs va invoquer l'opération $RMW(r, f)$ et suite à cette invocation, si un processeur reçoit la valeur v, il décide la valeur qu'il a proposé. Sinon, il décide la valeur de l'autre processeur. L'exécution du processeur P est décrite dans l'algorithme 4. L'exécution du processeur Q est quand à elle symétrique. On peut donc affirmer que le pouvoir de consensus des objets *read-modify-write* est supérieur ou égale à 2. Ainsi, on sait qu'il est impossible d'implémenter un objet *read-modify-write* uniquement en associant plusieurs registres atomiques étant donné que ceux ci ont un pouvoir de consensus égale à 1.

Algorithm 4 *decide*(*input* : *values*)**returns**(*value*)

```
prefer[P] ← input
if RMW(r, f) = v then
  return prefer[P]
else
  return prefer[Q]
end if
```

Un ensemble **interférent** est décrit comme suit : Soit F un ensemble de fonctions indexées par un ensemble S. Pour une valeur quelconque v et i, j dans S, les fonctions f_i et f_j sont soit commutatives : $f_i(f_j(v)) = f_j(f_i(v))$, soit f_i "écrase" (en anglais *overwrite*) f_j : $f_i(v) = f_i(f_j(v))$.

Les objets *read-modify-write* appliquant des fonctions à partir d'un ensemble F interférent ne peuvent pas non plus résoudre le problème de consensus dans un système comportant trois processeurs ou plus. La preuve est la suivante :

Soit trois processeurs P, Q et R. On construit une exécution amenant le système dans un état s qui est décisionnel. A partir de cet état, une opération $RMW(r, f_i)$ du processeur P amène le système dans un état x-valent et une opération $RMW(r, f_j)$ du processeur Q amène le système dans un état y-valent. P et Q exécutent leurs opérations sur le même registre ainsi, nous avons deux cas à considérer : soit les deux opérations sont commutatives, soit l'opération de P écrase l'opération de Q (ou inversement) :

- **Commutativité** : Soit l'état s' si le processeur P exécute son opération en premier et le processeur Q en second. Cet état s' est donc x-valent car c'est le processeur P qui exécute l'opération RMW en premier. Soit l'état s" si le processeur Q exécute son opération en premier et le processeur P en second. Cet état s" est donc y-valent car c'est le processeur Q qui exécute l'opération RMW en premier. La valeur qui est inscrite au sein du registre dans les états s' et s" est identique étant donné que les opérations des processeurs P et Q sont commutatives ce qui est contradictoire car les états s' et s" ne peuvent pas être à la fois x-valent et y-valent (Figure 3a).
- **L'opération P écrase l'opération Q** : Soit s' l'état du système quand le processeur Q exécute son opération. L'état s' est donc un état y-valent. Soit s" l'état du système quand le processeur P a exécuté son opération après le processeur Q, l'état s" est x-valent. Un processeur R qui exécute une série d'opérations à partir de l'état s' décide y, et s'il exécute une série d'opérations à partir de l'état s" il décide x. Ceci est une contradiction car le premier processeur à avoir exécuté une opération dans l'état s" est le processeur Q, donc le processeur R ne peut pas décider x car le système était passé dans un état y-valent après que le processeur Q ait effectué son opération (Figure 3b).

On peut donc affirmer que les objets *read-modify-write* ont un pouvoir de consensus de 2.

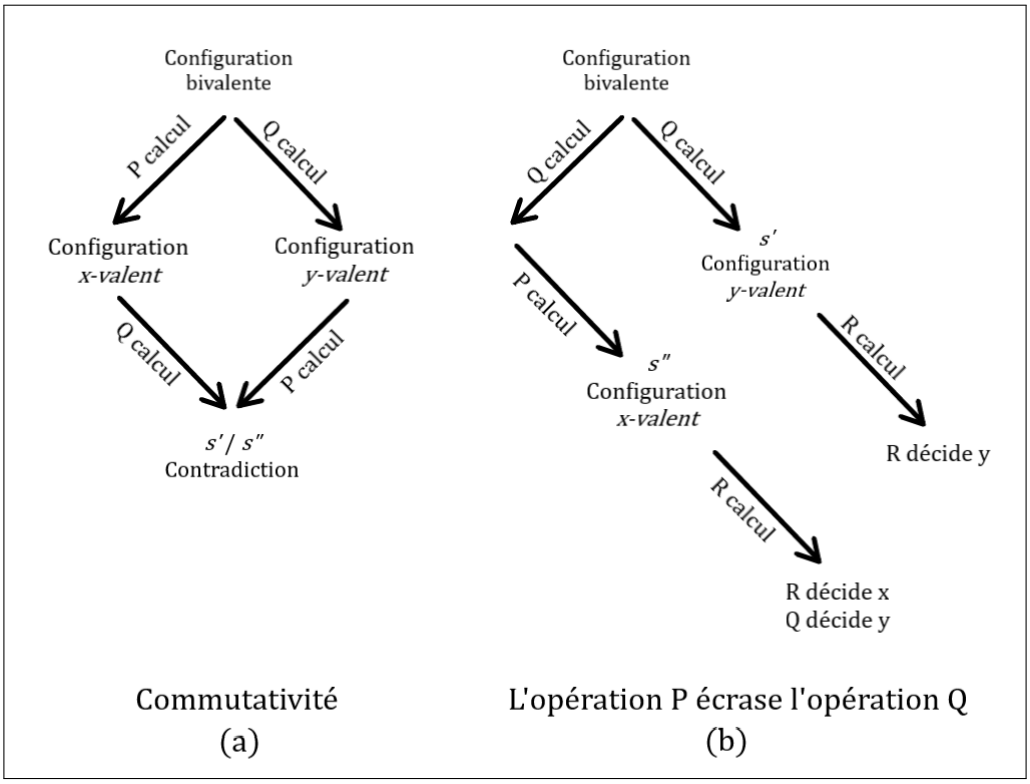

FIGURE 3 – Illustration de deux exécutions avec deux et trois processeurs

4.6 Les objets universels

On dit qu'un objet est *universel* s'il peut implémenter n'importe quel autre objet. Par conséquent, dans un système avec n processeurs ou moins, un objet de pouvoir de consensus n est universel. Un tel objet est définit par Herlihy [4].

4.7 Un objet à pouvoir de consensus k

Il est possible de créer un objet ayant un pouvoir de consensus k qui varie en fonction d'un paramètre k, Mostéfaoui et al. [7] en présente un. Cet objet est le registre atomique lecture/écriture k-glissant, ou RW_k. Pour une valeur de k égale à 1, cet objet se comporte comme un registre atomique. Notons $KREG$ un objet RW_k qui contient une séquence de valeurs accessibles via deux opérations : $KREG$.read() et $KREG$.write().

L'opération $KREG$.write(v) inscrit la valeur v à la fin de la séquence contenue dans l'objet et l'opération $KREG$.read() lit les k dernières valeurs de cette séquence (s'il y a moins de k valeur inscrite dans la séquence, l'opération comble avec la valeur $\perp$). L'algorithme qui résoud le problème de consensus dans un système à k processus est présenté dans l'algorithme 5.

La preuve du pouvoir de consensus de cet objet est la suivante :

On considère un système avec k processeurs. Comme les opérations *read*() et *write* sont

Algorithm 5 $propose(v_i : values)$**returns**$(value)$

$KREG.write(v_i)$
$seq_i \leftarrow KREG.read()$
Soit d la première valeur $\neq \perp$ dans seq_i
return (d)

atomiques, elles se terminent et l'algorithme *propose* n'ayant pas de boucle, lui aussi il se termine. Ainsi cet algorithme est bien $sans - attente$. On a au plus k processeurs qui vont effectuer l'opération $propose()$ et donc au plus k valeurs dans notre objet $KREG$. Tous renverront la première valeur v inscrite par le premier processeur. La preuve est donc similaire à celle décrite pour l'algorithme 1. On peut donc affirmer que les objets RW_k ont un pouvoir de consensus supérieur ou égale à k.

Un objet RW_k a aussi un pouvoir de consensus inférieur ou égale à $k+1$. La construction de la preuve est similaire à celle utilisée pour les objets *read-modify-write* : on amène le système dans un état bivalent puis on trouve une incohérence, cette fois ci non pas avec deux processeurs mais avec $k+1$. De cette manière on a bien un objet de pouvoir de consensus k.

Grâce à cette notion de pouvoir de consensus, nous avons pu mettre en évidence le concept d'objet dégradable.

5 Objets dégradables

Maintenant que nous avons donné le modèle sur lequel nous avons travaillé ainsi que les définitions qui permettront de comprendre l'axe de recherche que nous avons choisi, je vais introduire le concept d'objet dégradable.

5.1 Notion de dégradabilité

Un objet dégradable est une structure de données variable qui va s'adapter aux besoins de l'application. C'est plus précisément un ensemble d'objets qui ont la même fonction mais qui ont des *pré-conditions* et des *post-conditions* différentes, ce qui va modifier le critère de consistance. Une classe d'objets dégradables a plusieurs niveaux de dégradation. Un objet ayant un niveau de dégradation N, aura un critère de cohérence plus fort qu'un objet ayant un niveau de dégradation égale à N+1. On peut par exemple attendre d'un objet qu'il ait une seule action *entrée* quand il est à un niveau N et deux actions *entrée* quand il est au niveau $N+1$, ou on peut attendre que dans un cas les actions aient une valeur de retour et dans l'autre non (valeur vide/*null*).

Afin de mesurer le niveau de dégradation des objets, nous avons décidé de nous baser sur le pouvoir de consensus. Ainsi, pour "dégrader" un objet, nous allons chercher à modifier sa

spécification pour le faire changer de niveau de dégradation. L'objet que nous avons choisi d'utiliser pour mettre en évidence ce concept est le **compteur**.

5.2 Le compteur

L'objet que nous avons choisi de dégrader est un objet simple : **le compteur**. Un objet compteur est un objet auquel n processeurs ont accès d'au moins deux façons différentes.

Soit un processeur va incrémenter le compteur de 1, soit il va regarder la valeur actuellement stockée dans le compteur. La spécification de l'objet changera tout de même en fonction de son niveau de dégradabilité.

Comme indiqué dans la section précédente, nous avons choisi de nous baser sur le pouvoir de consensus de l'objet afin de définir son niveau de dégradabilité. De ce fait, notre objet compteur a trois niveaux de dégradabilité. Le premier est quand il a un pouvoir de consensus égale à 1, le second est quand il a un pouvoir de consensus égale à 2 et le dernier est quand son pouvoir de consensus est égale à n. Différents objets avec un pouvoir de consensus correspondant ont été utilisés afin d'implémenter l'objet compteur. Dans les parties suivantes, je donnerai l'implémentation de l'objet compteur pour chaque niveau de dégradabilité, sa spécification ainsi que le preuve de son pouvoir de consensus.

5.2.1 Spécification 1-consensus

Pour implémenter l'objet compteur avec un pouvoir de consensus de 1, nous avons choisi d'utiliser l'objet *snapshot* présenté en partie 4.4. Chaque processeur va opérer une invocation *increment*() dans le port qui lui est réservé puis sera capable de regarder la valeur totale stockée dans le compteur quand une opération *get*() est appelée. Cette opération renverra la somme des valeurs comprises dans le vecteur renvoyé par l'opération *snap*(). Ainsi, pour notre objet compteur nous avons m ports avec $m = n$ pour un système à n processeurs. Pour notre objet, les opérations *increment*() et *get*() peuvent avoir lieu sur le même port car chaque processeur doit être capable d'incrémenter le compteur et de lire son contenu.

A chaque port i associé au processeur P_i, il y a une variable x_i qui est une variable *swmr*. Cette variable contient les champs suivants :

— $val \in \mathbb{N}$, initialisé à 0

— *view*, vecteur de n valeurs $v \in \mathbb{N}$ initialisé à O

Un exemple de représentation du système est montré dans la Figure 4.

Le champs $x_i.val$ contient la valeur incrémentée par le compteur P_i. Le champs $x_i.view$ contient le snapshot embarqué effectué lors de l'opération *increment*(). Contrairement à l'objet *snapshot*, il n'est pas nécessaire d'associer un *tag* pour chaque opération d'*increment* (ou d'*update*). En effet, étant donné que chaque processeur incrémente sa variable x de 1, il ne peut pas y avoir deux opérations *increment*() pour un même processeur qui donne la même

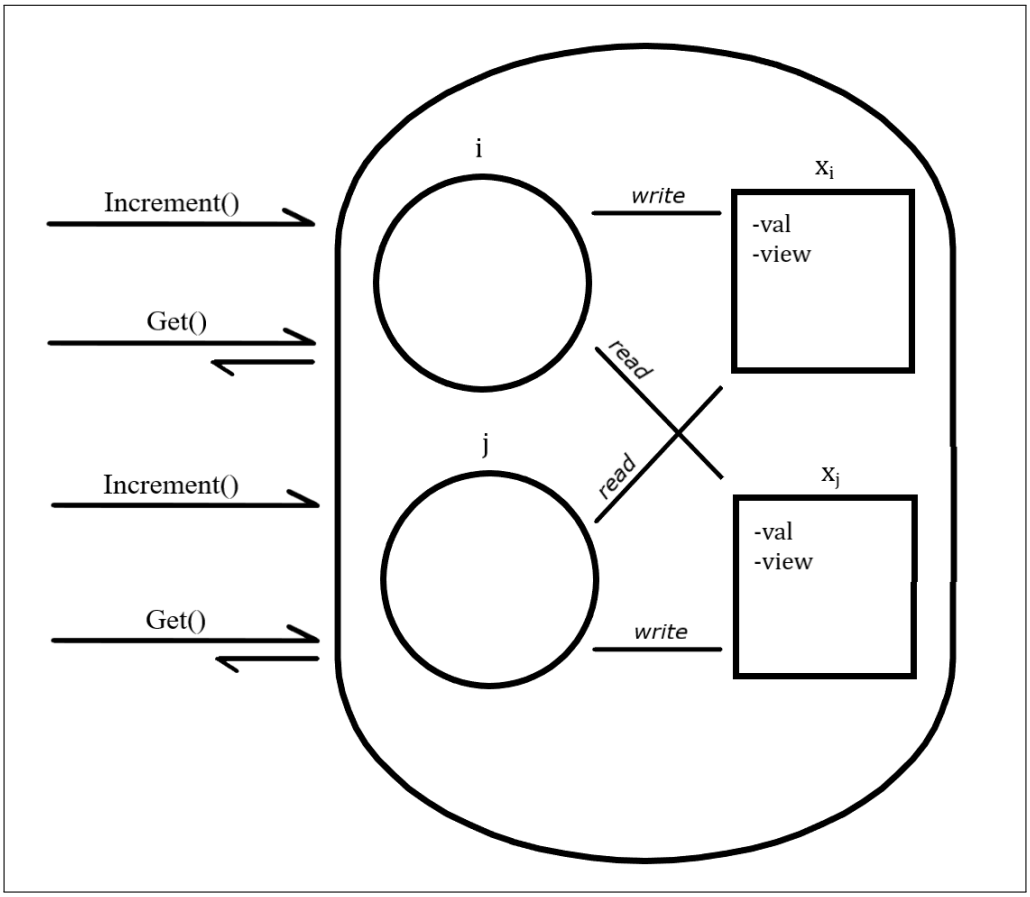

FIGURE 4 – Représentation d'un objet compteur pour deux processeurs

variable x. Ainsi, chaque valeur $x.val$ est son propre *tag*. Quand un processeur va effectuer une opération *increment*(), il va donc effectuer le même algorithme qu'un processeur effectuant une opération *update*() pour l'objet *snapshot* à l'exception près qu'il n'aura pas de *tag* à mettre à jour. L'algorithme effectué par le processeur P_i quand il exécute l'opération *increment*() est présenté dans l'algorithme 6. Celui exécuté quand il fait l'opération *get*() est présenté dans l'algorithme 7. La fonction $sum(x)$ renvoie la somme des entiers contenue dans le vecteur x.

Algorithm 6 *increment*()

$vec \leftarrow snap()$
$x_i.val \leftarrow x_i.val + 1$
$x_i.view \leftarrow vec$

Algorithm 7 *get*()**returns**(*value*)

$vec \leftarrow snap()$
return $sum(vec)$

Avec cet algorithme, on peut aussi incrémenter le compteur d'une valeur v tant que celle-ci est supérieur ou égale à 0. On peut ainsi se passer d'un *tag* pour les mêmes raisons que lorsque l'on incrémente le compteur de 1.

On a donc notre objet compteur qui a la spécification suivante :

— $increment() : \emptyset$

— $get() : \mathbb{N}$

L'implémentation de l'objet utilise uniquement des registres. On peut donc affirmer que l'objet compteur qui est caractérisé par la spécification ($increment() : \emptyset$; $get() : \mathbb{N}$) a un pouvoir de consensus de 1.

5.2.2 Spécification 2-consensus

Pour avoir un objet avec un pouvoir de consensus égale à deux, il faut changer la spécification et opter pour celle-ci :

— $increment() : \mathbb{N}$

— $get() : \mathbb{N}$

Nous avons implémenter cet objet à l'aide d'un objet *read-modify-write* : l'objet *fetch-and-add* [2]. Cet objet contient un registre partagé que l'on incrémente d'une valeur v et il renvoie le contenu du registre de manière atomique. Il a donc le même comportement que le compteur que l'on veut définir. Pour implémenter cet objet, nous avons besoin d'utiliser deux autres objets : l'objet *test-and-set* et l'objet *wigwag* [2].

L'objet *test and set* est un objet de pouvoir de consensus 2 et on y a accès via l'opération $T\&S()$. Si un processeur P exécute l'opération $T\&S()$, il essaye d'écrire dans le registre partagé : si personne n'a écrit dans le registre, le processeur P écrit 1 et la valeur de retour de

l'opération est 1, si un autre processeur a déjà écrit dans le registre partagé, le processeur P n'écrit pas dans le registre et la valeur de retour de l'opération est 0.

L'objet *wigwag* quand à lui est un objet ayant un pouvoir de consensus égale à 1. Il fonctionne comme une porte : le premier processeur qui passe la porte la ferme avec une valeur v. Les autres processeurs peuvent passer la porte uniquement s'ils essayent de passer avec cette même valeur v. L'objet *wigwag* est accessible via deux opérations : *wig-Pass* et *wig-Block*.

L'opération *wig-Pass* prend en entrée trois arguments : w, i et val. Ainsi, un processeur i tentera de passer par l'objet *wigwag* w avec la valeur val. Cette opération renvoie un booléen qui vaut vrai si aucun autre processeur n'a bloqué l'objet *wigwag* avec une valeur différente de val avant le passage du processeur i, et renvoie faux sinon.

L'opération *wig-Block* prend aussi trois arguments en entrée : w, i et $vec[1..n]$. De la même manière un processeur i bloquera l'objet *wigwag* w pour chaque processeur j avec la valeur $vec[j]$. Cette opération renvoie un vecteur de booléen $bvec[1..n]$ qui, pour chaque processeur j vaudra vrai si le processeur j n'a pas encore passé l'objet *wigwag* ou bien s'il est passé avec la valeur $vec[j]$ et renvoie faux sinon. Dans le second cas, le processeur i ne peut pas être sûr que le processeur j soit bloqué car les opération *wig-Pass* et *wig-Block* ne sont pas atomiques. En effet, si un processeur i voit une valeur différente de $vec[j]$ dans $w.WWPass[i][j]$, il ne peut pas savoir s'il a bloqué l'objet *wigwag* avant la fin de l'opération *wig-Pass*.

Un objet *wigwag* est mis en œuvre à l'aide de deux tableaux à deux dimensions $WWPass[i][j]$ et $WWBlock[i][j]$ avec $1 \leq i, j \leq n$. Un processeur qui tente de passer un objet *wigwag* l'indiquera à tous les autres processeurs via le tableau $WWPass$, puis il regardera dans le tableau $WWBlock$ pour savoir si l'objet a déjà été bloqué ou non. L'implémentation de l'algorithme effectuée quand un processeur i exécute l'action *wig-Pass* est présentée dans l'algorithme 8. Un processeur qui bloque l'objet *wigwag* va indiquer via le tableau $WWBlock$ pour chaque autre processeur, la valeur avec laquelle il bloque l'objet. Ensuite il regarde dans le tableau $WWPass$ quel processeur est susceptible d'être bloqué. L'implémentation de l'algorithme effectuée quand un processeur i exécute l'action *wig-Pass* est présentée dans l'algorithme 9.

Algorithm 8 $wig - Pass(w : wigwag, i : value, val : value)$**returns**$(boolean)$

```
for k ← 0; k < n; k++ do
  w.WWPass[k][i] ← val
end for
for k ← 0; k < n; k++ do
  if w.WWBlock[k][i] ≠ null ∧ w.WWBlock[k][i] ≠ val then
    return false
  end if
end for
return true
```

Algorithm 9 $wig - Block(w : wigwag, i : value, vec : vector)$ **returns***(boolean vector)*

```
for k ← 0; k < n; k++ do
  w.WWBlock[i][k] ← vec[k]
end for
for k ← 0; k < n; k++ do
  bvec[k] ← (w.WWPass[i][k] = null ∨ w.WWPass[i][k] = vec[k])
end for
return bvec
```

Cet objet est utilisé afin de pouvoir rendre linéarisable l'objet *fetch-and-add*.

L'objet *fetch-and-add* fonctionne de la manière suivante :

On a une liste infinie de cellules dans lesquelles tous les processeurs sont enregistrés avec une valeur. A la fin de chaque exécution, il y a un unique gagnant dans chaque cellule. Quand un processeur devient le vainqueur d'une cellule, il la bloque. Ensuite, le processeur qui sera le vainqueur dans cette cellule renverra en résultat de l'opération $fetch - and - add()$ la somme des valeurs avec lesquelles les processeurs ayant gagné dans une cellule inférieure à la sienne se sont inscrits.

L'idée c'est qu'un processeur va parcourir les cellules jusqu'à ce qu'il arrive à un index x tel qu'il puisse être le gagnant de la cellule $T[x]$. Le problème qui se pose, c'est qu'un processeur i peut perdre indéfiniment si un autre processeur j gagne constamment. Dans ce cas l'implémentation n'est pas *sans-attente*. Pour pallier à ce problème, les processeurs les plus rapides laissent des "trous" pour les processeurs les plus lents. Ainsi, si on admet le tableau de cellule $T[\infty]$, un processeur va approximer une position p tel que entre $T[p]$ et $T[p-K]$ (K est une valeur bornée par le nombre total de processeurs) il sera sûr de trouver une cellule dans laquelle il pourra gagner. Pour calculer cette valeur p, chaque processeur va stocker le nombre d'opérations $fetch - and - add()$ qu'il aura entamé dans une variable $my - inc$. Ensuite il fera un *snapshot* de toutes les valeurs $my - inc$ calculées par les processeurs du système. Comme les processeurs vont incrémenter la variable $my - inc$ au début de l'opération $fetch - and - add()$, un processeur trouvera forcément une cellule dans laquelle il peut gagner après un nombre fini mais non borné d'étapes.

En effet, ce nombre d'étapes sera non borné car entre le moment où le processeur incrémente sa variable $my - inc$ et le moment ou il fait son *snapshot*, il peut y avoir un grand nombre d'opérations $fetch - and - add()$ effectuées par les autres processeurs. Dans ce cas, la valeur de p sera excessivement grande et le processeur trouvera une cellule libre après un nombre non borné d'étapes. Pour borner la recherche d'une cellule dans laquelle il sera assuré de gagner, un processeur va calculer un *snapshot pertinent*. Le calcul de ce snapshot est décrit dans [2].

L'implémentation de l'objet *fetch-and-add* est la suivante : la liste infinie de cellules est organisée sous forme d'*étages* où chaque étage contient n cellules (n étant le nombre de processeurs dans le système). Chaque processeur i comptera le nombre d'opérations $fetch - and -$

$add()$ qu'il a entamé dans la variable $my-inc[i]$ et la valeur des k premières incrémentations dans la variable $total[k][i]$. La variable $entry$ contient l'étage à partir duquel le processeur cherchera une cellule libre. De cette manière, le premier étage contient les n premières cellules, le deuxième étage contient les cellules de $n+1$ à $2n$ etc. Un *snapshot* $SS[\ ]$ des valeurs $my-inc$ est également associé à chaque étage. Dans chaque cellule il y a un objet *wigwag* et un objet *test-and-set*. Les processeurs stockeront leur emplacement dans la variable $level[\infty][n]$. Le groupe de processeurs situé dans une cellule de niveau inférieur sera stocké dans la variable $LEFT$. Un schéma de cette implémentation est présenté dans la Figure 5.

L'algorithme appliqué par un processeur i qui effectue l'opération $fetch-and-add()$ est le suivant : il commence par mettre à jour la valeur totale des incrémentations via la variable $total[my-inc+1,i]$ (ligne 1) puis le nombre d'incrémentations qu'il a entamé via la variable $my-inc$ (ligne 2). Ensuite il calcule son point d'entrée $entry$ dans la liste de cellules via le *snapshot pertinent* M (lignes 3-4-5). Ce point d'entrée correspond à l'étage dans lequel va commencer la recherche d'une cellule pour le processeur. Le processeur i va par la suite stocker le *snapshot pertinent* qu'il a effectué dans le *snapshot* $SS[entry]$ dédié à l'étage $entry$ (ligne 6). Le processeur i va ensuite descendre d'étage en étage jusqu'à trouver quel est le *snapshot* $SS[]$ le plus récent qui a vu la dernière opération $fetch-and-add()$ effectuée par le processeur i (lignes 7-8-9). Cette étape permet de borner la recherche d'une cellule dans laquelle il peut gagner.

Ensuite, le processeur i va remonter les cellules une à une (de la gauche vers la droite dans la Figure 5) en passant par les objets *wigwag* des différentes cellules et ne s'arrêtera que dans deux cas : soit il sera bloqué par un objet *wigwag*, soit il aura atteint la dernière cellule de son *étage* d'entrée (lignes 10 à 18). Une fois que le processeur i est stoppé (d'une des deux manières), il entame une phase de recherche où il va chercher une cellule dans laquelle il peut remporter le *test-and-set* (avoir une valeur de retour de 1) et dans laquelle il peut être sûr qu'aucun autre processeur ne pourra passer cette cellule avec une valeur qui pourrait altérer la valeur de retour de l'opération $fetch-and-add()$ du processeur i. Il fait sa phase de recherche en redescendant les celulles (de la droite vers la gauche dans la Figure 5). Il y parvient en bloquant l'objet *wigwag* avec le *snapshot* dédié à l'étage courant. Ainsi, si un processeur réussit à passer l'objet *wigwag*, c'est que l'opération $fetch-and-add$ qu'il a **déjà réalisé** à été vu par le processeur i.

Après, le processeur i va regarder à quel niveau il se situe puis va compter le nombre de processeurs qui sont, soit bloqué dans le *wigwag* courant, soit dans une cellule inférieur à la sienne (ligne 22). Ce nombre de processeurs sera stocké dans la variable $LEFT$. Si le nombre de processeurs dans $LEFT$ est égale au niveau auquel se situe le processeur i, alors le processeur i est sûr qu'aucun autre processeur ne pourra rejoindre le groupe $LEFT$ (ligne 23). Dans ce cas, le processeur tente de remporter le *test-and-set* pour éviter les problèmes de

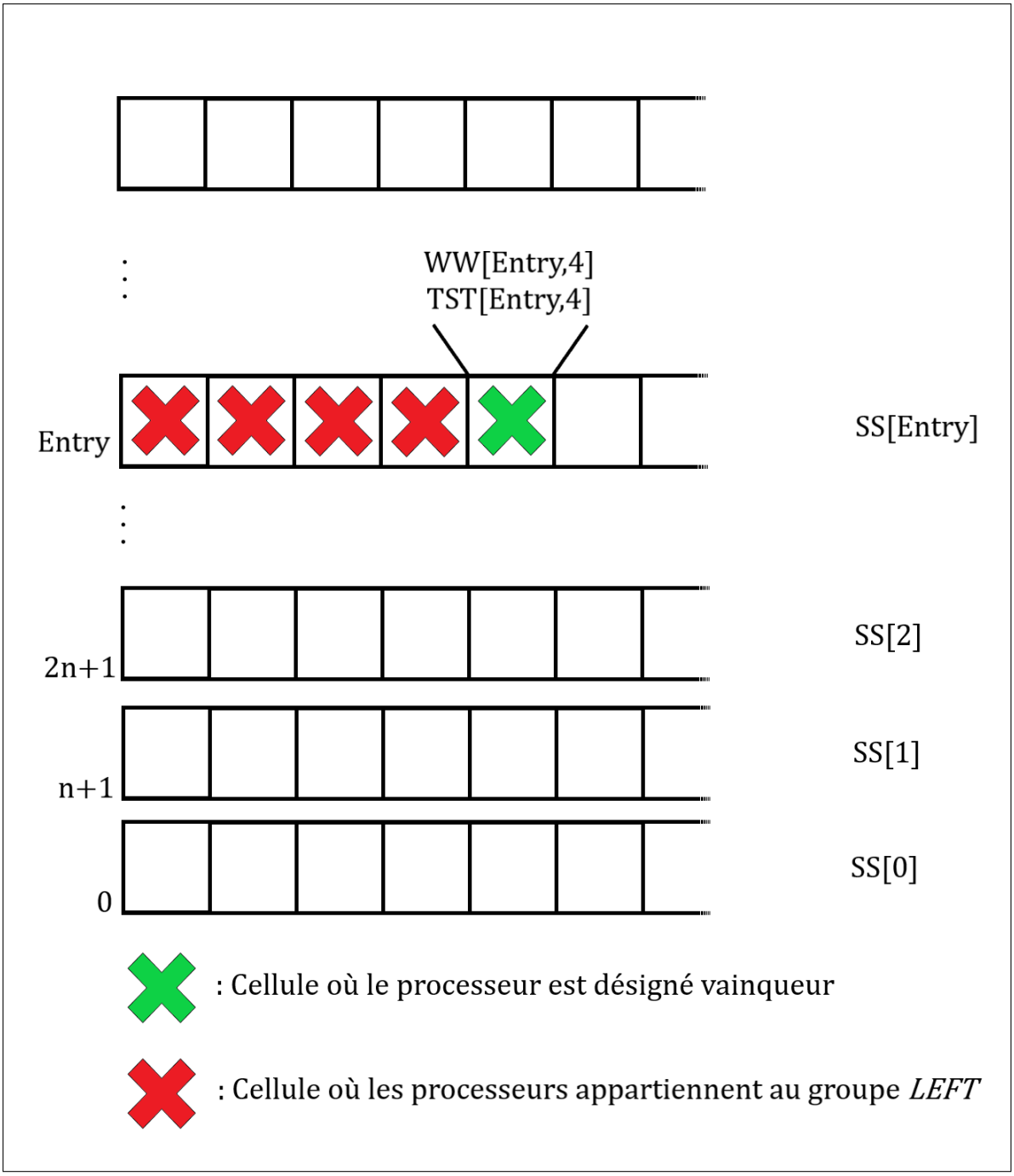

FIGURE 5 – Schéma de l'objet *fetch-and-add*

concurrence (ligne 24). S'il réussit, il renvoie la somme des valeurs comprises dans les variables *total* des processeurs qui sont dans $LEFT$ (lignes 25 à 31). Sinon il réessaye dans la cellule de niveau inférieur (ligne 34).

L'implémentation de l'algorithme effectué quand un processeur i exécute l'action $fetch - and - add()$ est présenté dans l'algorithme 10.

Algorithm 10 $fetch - and - add(input : value) return(value)$

```
1:  total[my − inc + 1][i] ← total[my − inc][i] + input
2:  my − inc[i] ← my − inc[i] + 1
3:  S ← snap() (snapshot of my-inc registers)
4:  M ← relevant_snapshot()
5:  entry ← r ← sum(M)
6:  SS[entry] ← M
7:  while SS[r][i] ≠ my − inc[i] − 1 do
8:     r ← r − 1
9:  end while
10: blocked ← false
11: while ¬blocked ∧ (r < entry) do
12:    r ← r + 1
13:    t ← 0
14:    while ¬blocked ∧ (t ≤ n) do
15:       t ← t + 1
16:       blocked ← ¬wig − Pass(WW[r][t], i, my − inc[i] − 1)
17:    end while
18: end while
19: while true do
20:    bvec ← wig − Block(WW[r][t], i, SS[r])
21:    level[r][i] ← t
22:    LEFT ← {∀j|bvec[j] ∨ level[r][j] ≤ level[r][i]}
23:    if |LEFT| ≥ t then
24:       if T&S(TST[r][t]) then
25:          for k ← 0; k < n; k + + do
26:             if (k ∈ LEFT) ∧ (k ≠ i) then
27:                faa ← faa + total[SS[r][k]][k]
28:             else
29:                faa ← faa + total[SS[r][k] − 1][k]
30:             end if
31:          end for
32:       end if
33:    end if
34:    t ← t − 1
35: end while
```

L'objet $fetch - and - add$ étant un objet *read-modify-write*, son pouvoir de consensus est égale à 2. Ainsi, nous avons bien un objet compteur avec un pouvoir de consensus de 2.

5.2.3 Spécification n-consensus

L'objet compteur avec un pouvoir de consensus de n est un objet avec la spécification suivante :

— $increment() : \emptyset$

— $seal(); \emptyset$

— $get() : \mathbb{N}$

On peut sceller ce compteur c'est-à-dire faire en sorte que lorsque l'invocation $seal()$ est effectuée, aucune invocation $increment()$ ne soit prise en compte. Avec cette spécification, il est possible de résoudre le problème de consensus pour un système avec n processeurs grâce à l'algorithme 11.

Algorithm 11 $propose(v : value)$**returns**$(value)$

```
if v = 0 then
  compt.seal()
else
  compt.increment()
end if
tmp ← compt.get
if tmp > 0 then
  return 1
end if
return 0
```

Dans cet algorithme, chaque processeur va proposer 0 ou 1. Un processeur qui propose 0 va sceller le compteur et un processeur qui propose 1 va l'incrémenter. Ensuite, les processeurs vont regarder l'état du compteur et baser leur choix sur la première action qui a été effectuée. Si la première action effectuée est $seal()$ alors le compteur est resté à 0 et tous les processeurs décideront 0, sinon ils décideront tous 1.

Pour implémenter cet objet il existe plusieurs objets avec un pouvoir de consensus de n décrit par Herlihy [4] (*file augmenté, registre avec mouvement, registre swap...*). Toujours dans [4], Herlihy montre que n'importe quel objet avec un pouvoir de consensus de n est *universel* dans un système avec n processeurs : c'est-à-dire qu'il peut implémenter n'importe quel autre objet. C'est pour cette raison que nous avons choisi d'utiliser l'implémentation de l'*objet universel* décrit par Herlihy [4] pour implémenter notre objet compteur scellé.

6 Conclusion

Maintenant qu'un objet dégradable a été mis en évidence, il serait intéréssant de voir si son utilisation permettrait d'améliorer les performances d'un programme. On pourrait regarder comment évolue la complexité des algorithmes en fonction du pouvoir de consensus pour avoir une idée de son efficacité. On pourrait aussi chercher une spécification qui utiliserait l'objet k-glissant afin d'avoir une infinité de niveau de dégradabilité.

Cette étude à été effectuée dans le modèle à mémoire partagée donc il serait également intéressant de voir comment se comporte les objets dégradables dans un modèle à passage de messages.

7 Références

Références

[1] N. A. Lynch and M. Tuttle. An introduction to input/output automata. revision. *CWI Quarterly*, 2 :27, 11 1988.

[2] Y. Afek, E. Weisberger, and H. Weisman. A completeness theorem for a class of synchronization objects. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing*, PODC '93, pages 159–170, New York, NY, USA, 1993. ACM. ISBN 0-89791-613-1. doi : 10.1145/164051.164071. URL `http://doi.acm.org/10.1145/164051.164071`.

[3] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2) :374–382, Apr. 1985. ISSN 0004-5411. doi : 10.1145/3149.214121. URL `http://doi.acm.org/10.1145/3149.214121`.

[4] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13 (1) :124–149, Jan. 1991. ISSN 0164-0925. doi : 10.1145/114005.102808. URL `http://doi.acm.org/10.1145/114005.102808`.

[5] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9) :690–691, Sept. 1979. ISSN 0018-9340. doi : 10.1109/TC.1979.1675439. URL `https://doi.org/10.1109/TC.1979.1675439`.

[6] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996. ISBN 9780080504704.

[7] A. Mostéfaoui, M. Perrin, and M. Raynal. A simple object that spans the whole consensus hierarchy. *CoRR*, abs/1802.00678, 2018. URL `http://arxiv.org/abs/1802.00678`.