Accord et cohérence sans connaître le nombre de participants

Mohssen Abboud, Carole Delporte-Gallet, Hugues Fauconnier

Accord et cohérence sans connaître le nombre de participants

Mohssen Abboud maboud@liafa.jussieu.fr
Carole Delporte cd@liafa.jussieu.fr
Hugues Fauconnier hf@liafa.jussieu.fr

Résumé

On étudie ici trois problèmes classiques de la tolérance aux défaillances dans le cas où l'ensemble des processus est inconnu. Ces trois problèmes sont : le problème du consensus, l'implémentation de registres atomiques et l'élection ultime de leader.

Pour cela on considère différents modèles. Dans le premier, la communication et les processus sont asynchrones, ces trois problèmes ne peuvent alors être résolus, mais on définit quels sont les détecteurs de défaillances minimaux pour ces problèmes.

On considère ensuite un modèle où les processus et la communication sont synchrones, ce qui permet de réaliser des rondes synchronisées. Ici, les processus sont créés dynamiquement et peuvent avoir de défaillances de type "crash". On montre que si que pour toute ronde r au moins un processus est vivant dans la ronde r et la suivante, le problème du consensus et de l'implémentation de registres peuvent être résolus. Le problème de l'élection de leader, qui est ici moins intéressant peut aussi être résolu.

Entre ces deux extrêmes on s'intéresse au cas où les communications sont asynchrones, mais où un processus une fois créé est vivant pour toujours. Dans ce cas, si l'élection de leader est aisée, le consensus et l'implémentation de registres est impossible sauf à supposer l'existence d'un détecteur de défaillances (Σ) qui permet de réaliser un quorum.

On considère enfin des modèles partiellement synchrones et on montre que le problème du consensus et de l'implémentation de registres peuvent être résolus s'il existe un processus tel que toutes les communications issues de ce processus sont synchrones.

1 Introduction

Les problèmes classiques de tolérance aux défaillances comme la diffusion atomique, le consensus, le maintien de la cohérence dans des serveurs répliqués ou l'élection de leader sont au coeur des réseaux dynamiques ou à grande échelle. Les solutions de ces problèmes supposent en général que l'on connaît, d'une part, le nombre de processus présents dans le système, et, d'autre part, que l'on peut borner le nombre de processus défaillants.

Ces hypothèses qui sont naturelles pour des réseaux locaux ne sont pas réalistes dès que l'on considère des réseaux à plus large échelle ou des réseaux dynamiques. Par exemple dans un système pair à pair, le nombre de participants peut être arbitrairement grand (potentiellement tous les sites Internet peuvent y participer) et dans un système dynamique le nombre de participants change avec le temps. La notion de processus défaillant est elle-même difficile à définir, par exemple, un processus qui peut, potentiellement participer et qui ne participe pas au système est-il défaillant ?

Dans cet article, nous allons nous intéresser à trois problèmes fondamentaux : le consensus, l'élection de leader, et le réalisation de registres atomiques. Nous étudions dans quelle mesure et comment on peut les résoudre dans des systèmes où le nombre ou l'ensemble des processus est inconnu mais où chaque processus a une identité unique.

La modélisation des réseaux dynamiques est encore actuellement un sujet d'études (par exemple [4, 1]). On fera ici plusieurs simplifications : on ne considère ici que des pannes "crash" (le processus s'arrête définitivement) et on suppose que le graphe de communication est complet. On s'intéresse à plusieurs modèles de calculs suivant le caractère plus ou moins synchrone de la communication.

Le premier modèle est asynchrone aussi bien pour les processus que pour la communication. Les résultats classiques d'impossibilité s'appliquent ici et on ajoutera des détecteurs de défaillances permettant de résoudre ces trois problèmes. Sans surprise, on peut dans ce cas vérifier que, pour l'essentiel, les détecteurs de défaillances minimaux pour le cas classique où l'ensemble et le nombre de processus sont connus sont aussi minimaux ici.

Sans détecteurs de défaillances, si la communication est asynchrone mais que les processus une fois créés sont définitivement corrects, on peut alors résoudre dans tous les cas le problème de l'élection, par contre pour les autres problèmes il faut des hypothèses supplémentaires (soit l'existence du détecteur de défaillances Σ soit la connaissance d'information sur le nombre des processus corrects et incorrects).

On s'intéresse ensuite au cas où la communication est synchrone et les processus, qui sont ici synchrones, sont créés dynamiquement et peuvent tomber en panne crash, les trois problèmes considérés peuvent y être résolus dès qu'à tout instant au moins un processus est vivant pendant suffisamment longtemps.

Enfin on considère des modèles partiellement synchrones où uniquement la communication issue de certains processus (appelés sources) est synchrone et les processus peuvent tomber en panne crash. On montre alors que l'existence d'au moins une source initiale permet de résoudre les trois problèmes considérés. On montre aussi que sans cette hypothèse les modèles partiellement synchrones classiques ne peuvent permettre de réaliser des registres atomiques.

En comparant avec le cas classique où le nombre processus est connu, on voit qu'ici et en particulier en ce qui concerne les modèles partiellement synchrones, l'implémentation de registres atomiques devient difficile alors que le l'élection reste généralement possible sous les mêmes hypothèses.

Les résultats présentés ici ont surtout un impact théorique en essayant de définir la frontière entre ce qui est possible et ce qui ne l'est pas. Par manque de place on ne donnera que des indications sur les preuves des algorithmes et des propositions.

2 Modèle

2.1 Processus

Chaque processus a une identité unique prise dans un ensemble Π, qui représente l'ensemble des identités possibles des processus. On suppose que cet ensemble Π est totalement ordonné. Dans la suite on identifiera un processus avec son identité. L'ensemble Π peut être infini, mais on suppose que dans toute exécution au plus un nombre fini de processus est crée.

Le temps $\mathcal{T}$ est discret, et sera identifié à l'ensemble des entiers naturels. Les processus exécutent des pas de calculs atomiques aux divers instants t. Plus précisément, $a(p,t)$ représente l'activité du processus p à l'instant t : $a(p,t)$ est soit une action atomique soit ϵ si p ne fait pas de pas de calcul à l'instant t.

Pour tout $p \in \Pi$, le *début* de p, $d(p)$, est l'instant, s'il existe, du premier pas de p : $d(p) = \inf\{t | a(p,t) \neq \epsilon\}$, par convention, si pour t on a $a(p,t) = \epsilon$ (p ne fait aucun pas de calcul) alors $d(p) = \infty$.

Pour tout $p \in \Pi$, la *fin* de p, $f(p)$ est l'instant, s'il existe, du dernier pas de p : $f(p) = \sup\{t | a(p,t) \neq \epsilon\}$, par convention, si pour tout t il existe $t' > t$ tel que $a(p,t') \neq \epsilon$ (p fait une infinité de pas de calcul) alors $f(p) = \infty$.

Introduisons quelques définitions. Considérons une exécution, un processus p est *vivant à l'instant* t si $d(p) \neq \infty$ et $d(p) \leq t \leq f(p)$; un processus p est *mort à l'instant* t si $f(p) < t$. p a été *créé* si $d(p,t) \neq \infty$; p est *correct* si p a été créé et $f(p) = \infty$. On notera que p est correct si et seulement si il fait une infinité de pas de calcul. On dira que p est *défaillant* si p fait un nombre fini non nul de pas de calcul ($d(p) \neq \infty$ et $f(p) \neq \infty$). On dira qu'un algorithme tolère k pannes s'il vérifie sa spécification pour au plus k processus défaillants.

En général, les processus peuvent être *asynchrones* : à chaque instant t ils peuvent ne pas faire de pas de calcul. Un processus p est *synchrone* si une fois créé il exécute des pas de calcul chaque instant t entre sa création et sa fin : pour t tel que $d(p) \leq t \leq f(p)$, $a(p,t) \neq \epsilon$.

Comme un processus asynchrone n'est pas obligé de faire des pas de calcul à tout instant t, la date exacte de création ou de mort n'est pas significative, les seules notions significatives sont les faits que $d(p)$ est ou non égal à ∞ (p est ou n'est pas crée) et que p fait ou non une infinité de pas de calculs (p est correct). Ainsi pour des processus asynchrones on s'intéressera aux propriétés suivantes de p : p est créé (il fait au moins un pas de calcul), p est correct (il fait une infinité de pas de calculs).

2.2 Communication

Les processus communiquent par envoi et réception de messages. On suppose que le graphe de communication est complet : il existe un canal de communication entre tout p et tout q. On suppose de plus que les messages émis ne sont ni altérés ni dupliqués (propriété d'intégrité). D'une manière générale un message émis par p à l'instant t vers q sera reçu par un processus q correct à un instant $t' > t$ (on suppose donc qu'il n'y a pas de pertes de messages).

Comme dans les algorithmes que l'on va considérer ici, les processus envoient des messages à *tous* on considère une primitive de diffusion $diff(m)$ qui permet à un processus p d'envoyer un message m à tous les processus vivants. Plus précisément, si p fait un $diff(m)$ à l'instant t et p est vivant à l'instant t, et si q est correct alors q recevra le message m. Cette propriété n'est pas vérifiée si p effectue $diff(m)$ à l'instant t de son dernier pas de calcul ($t = f(p)$), il est alors possible que certains processus reçoivent m alors que d'autres ne le reçoivent pas.

Comme on suppose ici qu'un processus peut envoyer un message à tous les processus, cette primitive de diffusion peut s'implémenter simplement : pour faire un $diff(m)$, p envoie m successivement à tous les processus. [1]

Quand la communication ne satisfait que les propriétés précédentes, on dira que la communication est *asynchrone*.

Dans la suite, *Async* représentera un système où les processus sont asynchrones et la communication est asynchrone.

Nous allons maintenant nous intéresser aux cas où les délais de communication sont bornés et connus pour certains canaux. Pour cela nous supposons dans les deux sous sections suivantes que les processus sont synchrones et peuvent mesurer le temps.

2.2.1 Systèmes synchrones et partiellement synchrones

Considérons des processus synchrones, le canal de p vers q est *ponctuel pour* δ si, (1) pour toute exécution, aucun message émis par p vers le processus q n'est reçu par q plus de δ unités de temps après son émission et (2) tout message émis par p au temps t sera reçu au plus tard au temps $t + \delta$ si q est vivant du temps t au temps $t + \delta$. Notons que si q ou p ne sont pas créés, le canal de p à q est trivialement ponctuel pour n'importe quel δ.

[1] Une implémentation plus efficace dans laquelle p n'enverrait m qu'aux seuls processus vivants est possible mais sort du cadre de cet article.

Si tous les canaux sont ponctuels pour un certain δ connu, on dira que la *communication est synchrone*. Dans ce cas, tout message émis par un processus vivant à l'instant t sera toujours reçu avant le temps $t + \delta$ si le destinataire est vivant dans l'intervalle de temps t à $t + \delta$.

Dans la suite un système où les processus sont synchrones et la communication est synchrone sera noté *Sync*.

On peut restreindre le caractère synchrone de la communication en supposant que tous les canaux ne sont pas ponctuels : Si, pour un certain δ connu des processus, les canaux d'un processus p *correct* vers n'importe quel autre processus sont ponctuels pour δ, on dira que p est une *source*. [2] Si cette propriété n'est réalisée pour le processus p que de façon ultime, c'est-à-dire s'il existe un instant à partir duquel, les canaux de communication de p vers n'importe quel autre processus sont ponctuels, on dira que p est une *source ultime*.

En considérant la primitive $diff$, si s est une source pour un certain δ, alors tout message m diffusé à l'instant t par s sera reçu avant l'instant $t + \delta$ par tous les processus vivants de l'instant t à l'instant $t + \delta$. Cette propriété s'étend bien entendu à tous les processus dans le cas où le système est synchrone.

Dans la suite un système où les processus sont synchrones avec une source sera noté $PSync+Source$, de même un système où les processus sont synchrones avec au moins une source ultime sera noté $PSync + ESource$.

Dans le cas de la communication synchrone, on considérera un modèle de *rondes synchronisées* [19]. Les processus exécutent des rondes synchronisées successives. A chaque ronde chaque processus vivant (1) diffuse un message vers tous (2) reçoit les messages de la ronde des autres processus (3) change d'état en conséquence et passe à la ronde suivante. Si un processus tombe en panne durant la phase (1) de la ronde, certains processus peuvent recevoir son message alors que d'autres ne le reçoivent pas.

On peut de façon relativement simple implémenter des rondes synchronisées dans le cas où la communication est synchrone.

2.3 Consensus, registres atomiques et élection de leader

Nous considérons ici trois problèmes particuliers.

Le problème du consensus est un problème classique fondamental de l'algorithmique distribuée tolérante aux pannes. Un algorithme de consensus est un algorithme de décision pour lequel on suppose que chaque processus p dispose d'une valeur initiale v_p, les décisions des processus sont irrévocables et doivent assurer les propriétés suivantes :

- *Terminaison :* tout processus correct doit décider d'une valeur,
- *Accord :* si p et q décident ils doivent décider la même valeur,
- *Validité :* si p décide la valeur v alors v est la valeur initiale d'un processus.

Rappelons tout d'abord qu'il n'existe pas de solution au problème du consensus tolérant même une seule panne dans le cas où la communication est asynchrone (même si le nombre de processus est connu) [13].

Le deuxième problème considéré ici est le problème de l'implémentation de registres atomiques. Un registre partagé est défini par les primitives $read()$ et $write(v)$. Essentiellement, le $read$ retourne la dernière valeur écrite dans le registre par un $write()$. Les registres que l'on considère ici sont des registres atomiques qui vérifient la propriété de linéarisabilité [16]. Cette propriété signifie que les $read$ et les $write$ exécutés par des processus corrects terminent toujours et que, pour toutes les exécutions, pour toute opération $read$ ou $write$, on peut choisir un instant t compris entre le début et la fin de cette opération de façon à ce que la séquence des

[2] on peut noter que si une communication synchrone implique que tous les corrects sont des sources, une communication synchrone implique en plus que les communications issues de processus qui ne sont pas corrects sont elles aussi ponctuelles.

opérations corresponde à une séquence des opérations sur un registre qui ne serait pas partagé et où les opérations *read* et *write* seraient instantanées et exécutés à ces instants t.

Rappelons qu'on peut toujours réaliser un registre partagé entre plusieurs lecteurs et plusieurs écrivains à partir de registres partagés entre un seul lecteur et un seul écrivain [21, 17]. Aussi dans la suite, on ne s'intéresse qu'à de tels registres qu'on appelle $SRSW$-registres.

Le troisième problème considéré ici est celui de l'élection ultime de leader, dans ce problème chaque processus p dispose d'une variable $leader_p$ qui contient l'identité d'un processus. Dans une élection ultime de leader, il existe un instant t après lequel pour tout processus p, pour toujours $leader_p$ est l'identité d'un même processus correct, s'il en existe. Notons que pour un temps arbitrairement long les processus peuvent ne pas avoir choisi le même leader. Notons aussi que le processus choisi doit être correct.

3 Systèmes asynchrones et détecteurs de défaillances

Rappelons que dans les systèmes asynchrones avec un nombre connu de processus, (1) il n'y a pas d'algorithmes de consensus tolérant même qu'une seule panne crash [13], (2) un registre ne peut être implémenté que s'il y a une majorité de processus corrects [3] et (3) il n'existe pas d'algorithmes d'élection de leader [13] même en présence d'une seule panne crash. De ces résultats, on déduit aisément :

Proposition 1 *Dans Async, il n'existe pas d'algorithmes tolérant même une seule panne ni pour le problème consensus, ni pour l'implémentation d'un registre, ni pour l'élection de leader ultime.*

Pour contourner ces résultats d'impossibilité, on peut ajouter au système des *détecteurs de défaillances* [8]. Un détecteur de défaillance est un oracle distribué qui donne à chaque processus des informations qui peuvent être inexactes sur les pannes. On trouvera des définitions formelles dans [8]. Si un détecteur de défaillances $\mathcal{X}$ est invoqué par le processus p à l'instant t, on notera $\mathcal{X}(p,t)$ la sortie de ce détecteur ; on suppose que cette sortie est à chaque invocation une liste de processus. On considérera essentiellement les détecteurs suivants :

- Le détecteur parfait, noté $\Diamond\mathcal{P}$, propose des listes de processus considérés comme vivants : qui vérifient les propriétés suivantes : (a) il existe un instant t à partir duquel pour tout $t' > t$ et pour tout p, $\Diamond\mathcal{P}(p,t')$ contient tous les processus vivants (exactitude ultime) et (b) il existe un instant t tel que pour tout processus p, $\Diamond\mathcal{P}(p,t)$ ne contiendra que des processus corrects (complétude).
- Le détecteur Ω : la sortie de ce détecteur ne contient l'identité que d'un seul processus, considéré comme un leader possible, Ω est tel qu'à partir d'un certain instant t il propose pour toujours et à tous l'identité d'un même processus correct.
- Le détecteur Σ est un détecteur de défaillances réalisant un quorum : toutes les réponses ont toujours un élément commun. Plus précisément il assure (a) pour tout t, t' pour tout p,p' $\Sigma(p,t) \cap \Sigma(p',t') \neq \emptyset$ (propriété d'intersection) et (b) la propriété de complétude définie ci-dessus.

On peut remarquer que Ω correspond exactement au problème de l'élection ultime d'un leader tel qu'il a été défini ci-dessus : réaliser une élection ultime de leader ou implémenter un détecteur de défaillances Ω revient exactement à la même chose.

Une propriété importante des détecteurs de défaillances est le fait qu'on peut les comparer par réduction : un détecteur $\mathcal{Y}$ est plus faible qu'un détecteur $\mathcal{X}$ s'il existe un algorithme qui peut transformer toute sortie de $\mathcal{X}$ pour en faire une sortie de $\mathcal{Y}$. Par exemple, on peut montrer aisément que Ω est plus faible que $\Diamond\mathcal{P}$ (il suffit de choisir comme leader le processus ayant l'identité la plus petite dans les sorties du détecteur $\Diamond\mathcal{P}$).

Pour certains problèmes on peut déterminer le plus faible détecteur de défaillances permettant de le résoudre. Si $\mathcal{X}$ est un plus faible détecteur de défaillances pour un problème P alors (1)

$\mathcal{X}$ permet de résoudre le problème P et (2) tout détecteur de défaillances permettant de résoudre P peut être réduit en $\mathcal{X}$. Si $\mathcal{X}$ est le détecteur de défaillance minimal permettant de résoudre P, $\mathcal{X}$ intuitivement encapsule les informations sur les défaillances nécessaires et suffisantes pour résoudre P.

En adaptant les résultats obtenus pour le cas où le nombre de processus est connu, on obtient :

Proposition 2 *Pour ASync, Σ est le plus faible détecteur de défaillances permettant d'implémenter un registre atomique, $\Omega \times \Sigma$ [3] est le plus faible détecteur de défaillances permettant de résoudre le consensus.*

Pour prouver cette proposition, le cas où le nombre de processus est inconnu étant une généralisation du cas où le nombre de processus est connu, il suffit de vérifier que les réductions définies dans [7] et [9, 11] s'appliquent ici. Il faut aussi montrer qu'il est possible avec $\Omega \times \Sigma$, respectivement Σ, de réaliser le consensus, respectivement d'implémenter un registre atomique. L'algorithme de la Figure 2 permet de réaliser le consensus. Notons que les algorithmes classiques comme [8, 20, 10] utilisent la connaissance du nombre de processus pour choisir un "coordinateur tournant" ou nécessitent de lire tous les registres [18].

Cet algorithme utilise certaines techniques de l'algorithme de [5]. Les processus font d'abord deux rondes successives asynchrones (correspondant à la variable r) qu'on appellera phase. Dans la première, ils échangent leur valeur avec un message $PROP$. La propriété d'intersection de Σ assure que tous les processus reçoivent parmi les messages $PROP$ le même message $PROP$ du même processus. Si un processus ne reçoit qu'une seule valeur v dans les messages $PROP$ il envoie ensuite cette valeur v dans le message DEC, sinon il envoie $\perp$. On peut remarquer que, à ce moment, il est impossible qu'il y ait deux valeurs v et v' différentes envoyées dans les messages DEC. Si le processus ne reçoit que des messages DEC avec la même valeur v, il décide cette valeur, s'il a reçu au moins une valeur v il optera pour cette valeur, s'il n'a reçu que des $\perp$ il garde sa valeur. Il faut noter qu'ici si un processus décide v, la propriété d'intersection de Σ et la remarque précédente entraîne que tous les processus opteront pour v. Les processus envoient ensuite la valeur qu'ils ont adoptée et attendent ensuite la valeur retournée par le processus qu'ils pensent être le leader qu'ils proposeront à la phase suivante. Cette valeur ne peut pas contredire une valeur déjà décidée. On remarque que si tous les processus proposent à une phase la même valeur, cette valeur sera décidée par tous à la fin de la phase. Le leader permet d'assurer qu'un jour tous les processus proposeront sa valeur au début de la phase.

L'algorithme de la Figure 1 [9] implémente un $SRSW$-registre atomique.

Comme toutes les exécutions des divers systèmes considérés ici ($Sync$, $PSync + Source$, $PSync + ESource$) sont des sous-ensembles des exécutions du système asynchrone, on a les résultats suivant :

Proposition 3 *Pour tout système dont les exécutions peuvent être considérés comme des exécutions de ASync :*

- *Si on peut implémenter Σ, alors il est possible d'implémenter un registre atomique.*
- *Si on peut implémenter Ω et Σ, alors il est possible de résoudre le problème du consensus.*

D'autre part, on peut remarquer que le problème du consensus est plus fort que le problème de l'implémentation d'un registre atomique. En effet, à partir du consensus on peut réaliser la diffusion atomique [15] et il est facile de réaliser un registre atomique à partir d'une primitive de diffusion atomique.

4 Systèmes synchrones

Pour la description des algorithmes on suppose que les processus exécutent des rondes synchronisées comme définies ci-dessus. Il est clair que si tous les processus participant à une ronde

[3] $\mathcal{X} \times \mathcal{Y}$ est un détecteur de défaillances qui retourne à la fois la sortie de $\mathcal{X}$ et de $\mathcal{Y}$.

```
1  Tous les processus p_i (y compris p_w et p_r) exécutent le code suivant :
2    Initialisations :
3      current := ⊥
4      last_write := −1
5    upon receive (WRITE, y, s) from p_w
6    if s > last_write then
7      current := y
8      last_write := s
9      send(ACK_WRITE, s) to p_w
10   upon receive (READ, s) from p_r
11     send(ACK_READ, last_write, current, s) to p_r
12 Code pour p_w l'(unique) écrivain :
13   Initialisation :
14     seq := 0 /* numéro de séquence */
15   procedure write(x)
16     send(WRITE, x, seq) to all
17     wait until received (ACK_WRITE, seq) from tous les processus donnés par Σ
18     seq := seq + 1
19   end write
20 Code pour p_r l'(unique) lecteur :
21   Initialisation :
22     rc := 0 /* compteur */
23   function read()
24     rc := rc + 1
25     send(READ, rc) to all
26     wait until received (ACK_READ, ∗, ∗, rc) from tous les processus donnés par Σ
27     a := max{v | (ACK_READ, v, ∗, rc) est un message reçu }
28     if a > last_write then
29       current := v such that (ACK_READ, a, v, rc) est un message reçu
30       last_write := a
31     return(current)
32   end read
```

FIG. 1 – Implémentation d'un $SRSW$ registre atomique avec Σ.

sont morts avant la suivante il est impossible de maintenir une quelconque information. On fera donc l'hypothèse suivante :

(H1) pour toute ronde r et $r+1$ il existe au moins un processus vivant dans les deux rondes.

Proposition 4 *Si (H1) n'est pas assurée, il n'est pas possible ni de réaliser un registre ni de réaliser un consensus*

En effet, considérons un registre partagé entre un écrivain p_w et un lecteur p_r, si p_w termine l'écriture d'une valeur v en ronde i, et que tous les processus qui étaient vivants jusqu'en ronde i meurent dans la ronde, supposons que le lecteur n'est créé qu'après la ronde i, alors aucun processus ayant pu voir la valeur écrite n'existe après la ronde i, et la valeur retournée pas un *read* ne peut donc être v. Un argument similaire montre la même chose pour le consensus.

Dans la suite, on supposera donc que (H1) est toujours réalisée.

On peut alors implémenter facilement (mais de façon peu efficace) un registre. Pour cela, les valeurs écrites contiennent un numéro d'ordre s, quand l'écrivain écrit v dans le registre, et qu'il s'agit de la i-ème écriture, il envoie à tous le message $(REG, (v, i))$. Chaque processus maintient une copie, C, du registre contenant la valeur et le numéro d'ordre associé. A chaque

```
Code pour p :
exécuter en parallèle les 3 taches suivantes :
Tâche 0 :
1       upon receive(AVIS, *, k) pour la première fois
2           let (AVIS, w, k) be ce message
3           send(LEADER, w, k) to all

Tâche 1 :
4     r := 0

5     loop forever

6       send(PROP, v, r) to all
7       wait until receive (PROP, *, r) from  tous les processus de Σ
8       let L = {w | (PROP, w, r) a été reçu }
9       if L={rec} then
10        est := rec
11      else
12        est := ⊥

13      send(DEC, est, r) to all
14      wait until receive (DEC, *, r) from  tous les processus de Σ
15       let L = {w | (DEC, w, r) a été reçu }
16       if L = {rec} for some rec ≠ ⊥ then
17        send (DECIDE, rec) to all
18        decider(rec)
19        stop
20      else
21        if L = {rec, ⊥} such that rec ≠ ⊥ then
22          w := rec
23        else
24          w := v
25        send(AVIS, w, r)  to all
26        wait until received(LEADER, *, r) from Ω
27        let (LEADER, val, r)be le message reçu
28          v := val

Tâche 2 :
29      upon received(DECIDE, k) from q
30            send(DECIDE, k) to all
31            decider(k)
32            stop
```

FIG. 2 – Consensus avec $\Sigma \times \Omega$

ronde, chaque processus envoie à tous $(REG, (x, j))$ x et j étant respectivement la valeur de la copie et le numéro d'ordre associé. A la fin de la ronde, les processus mettent à jour la valeur de la copie du registre en choisissant le couple (v, i) tel que $(REG, (v, i))$ a été reçu dans la ronde et i est maximal pour le j parmi $(REG, (x, j))$ messages reçus dans la ronde. Pour lire la valeur du registre le lecteur retourne simplement la valeur de sa copie du registre à la fin de la ronde. On notera, que si l'écrivain meurt dans la ronde les valeurs des copies ne sont pas nécessairement toutes identiques. La propriété (H1) assure que la valeur du registre est conservée d'une ronde à l'autre.

Il est intéressant de noter que bien qu'il soit possible d'implémenter un registre dans *Sync* vérifiant $(H1)$, il n'est pas possible dans ce système d'implémenter Σ. Cela provient du caractère synchrone du système qui permet de se passer des informations sur les pannes qui seraient

nécessaires avec une communication asynchrone.

Il est facile d'élire un leader ultime. Chaque processus maintient une variable *leader* initialisée à sa propre identité $MonId$. A chaque ronde, si un processus pense être leader ($leader{=}MonId$), il diffuse le message $(LEADER, MonId)$, sinon il ne diffuse aucun message. S'il reçoit des messages de $(LEADER, *)$, il choisit comme nouveau leader celui qui a la plus petite identité, s'il n'en reçoit aucun dans une ronde il positionne *leader* à $MonId$.

Le consensus est un peu plus difficile à réaliser. Nous allons décrire un algorithme (Figure 3) qui permet de réaliser le consensus sans la connaissance de l'ensemble Π.

Rappelons tout d'abord que si p envoie à tous un message m, il se l'envoie aussi à lui-même. Chaque processus p maintient un ensemble S_p des valeurs proposées qu'il a vu. Initialement, cet ensemble est vide. Il maintient aussi un ensemble $Participants_p$ qui contient la liste des identités de processus desquels il a reçu un message dans la ronde. A chaque ronde, exceptée la première à laquelle il participe, p enverra un message (SET, S_p) contenant son ensemble S_p. A chaque ronde aussi chaque processus met à jour l'ensemble $Participants_p$ comme étant l'ensemble des processus desquels p a reçu un message dans la ronde.

La première ronde à laquelle p participe est particulière. Dans cette ronde,, p envoie à tous un message $(PROP, v_p)$ contenant sa valeur initiale v_p. S'il ne reçoit à la fin de sa première ronde que des messages $(PROP, x)$ (c'est donc qu'il n'a reçu que des messages provenant de processus qui exécutent aussi leur première ronde), il initialise l'ensemble S_p à la réunion des valeurs x des messages $(PROP, X)$ reçus dans la ronde. Sinon, s'il reçoit au moins un message $(SET, -)$, il met à jour S_p en faisant l'union des ensembles X contenus dans les messages (SET, X) reçus dans la ronde. Intuitivement cela signifie que, s'il voit que d'autres ont commencé avant le consensus et ont donc envoyé leur message SET, il ne propose pas sa valeur mais participe à l'algorithme avec les valeurs des autres. Notons que $(H1)$ permet d'assurer que si un consensus a déjà commencé alors p le voit.

Après cette première ronde, p met à jour S_p à la fin de chaque ronde en faisant l'union de tous les X reçus dans un message (SET, X).

Il pourra décider si les deux conditions suivantes sont réunies : (1) p a reçu d'exactement des mêmes processus dans la ronde et la ronde précédente et (2) tous les ensembles X des (SET, X) reçus dans la ronde sont identiques.

En résumé :

Proposition 5 *Si Sync assure la propriété $(H1)$ l'implémentation de registres, le consensus et l'élection ultime de leader peuvent être résolus.*

5 Systèmes asynchrones sans défaillances

Dans cette section nous considérons des systèmes asynchrones dans lesquels, une fois créés, un processus est sans défaillances. On ne connaît pas ici le nombre de processus qui vont être créés.

L'élection ultime de leader est simple : chaque processus envoie à tous son identité et chaque processus choisit comme leader le processus d'identité minimale parmi les messages reçus. En effet, soit p_m le processus d'identité minimale parmi ceux qui sont créés pour l'exécution minimale, comme, une fois créé, p_m reste vivant, il enverra à tous son identité ; tous les processus qui seront créés reçoivent cette identité minimale et finiront par choisir p_m comme leader.

Par contre, la connaissance du nombre de processus est nécessaire pour pouvoir implémenter un registre atomique et donc résoudre le problème du consensus :

Proposition 6 *Si les processus sont synchrones, la communication asynchrone et les processus une fois créés n'ont pas de défaillances, il n'existe pas d'algorithme ni pour résoudre le problème du consensus ni pour implémenter des registres.*

```
Code pour p :
initialisations
1    S_p := ∅
2    Participants_p := ∅
3    FirstR_p := True;
4    Decision := ⊥;

5  pour chaque ronde r :
6  SEND :
7    if Decision ≠ ⊥ then
8      send (DECIDE, Decision) to all
9    else
10     if FirstR_p then
11       FisrtR_p := False;
12       send (PROP, v_p) to all
13     else
14       send (SET, S_p) to all
15 RECEIVE :
16   if Decision = ⊥ then
17     let MSET be l'ensemble des messages (SET, S) reçus dans la ronde
18     let MPROP be l'ensemble des messages (PROP, v) reçus dans la ronde
19     let P_p be l'ensemble des q tel qu'un message (SET, S) a été reçu de q dans la ronde
20     if reçu (DECIDE, v) then
21       Decision := v
22       decider(v)
23     else
24       if MSET ≠ ∅ then
25         S_p = {X|(SET, X) ∈ MSET}
26       else
27         S_p = {v|(PROP, v) ∈ MPROP}
28       if(P_p = Participants_p) ∧ (∀(SET, X) ∈ MSET : X = S_p) then
29         Decision := min{v ∈ S_p}
30         decider(Decision)
31       Participants_p := P_p
```

FIG. 3 – Consensus dans un système synchrone.

Par contre le problème de l'élection de leader peut être résolu.

En considérant les processus qui ne sont pas créés comme des processus qui seraient en panne dès le début, [4] on peut déduire de [3] qu'il faut une majorité de processus corrects pour pouvoir implémenter un registre. On peut alors prouver que, sans la connaissance de Π, il est impossible d'implémenter un registre et donc aussi de réaliser un consensus.

Par contre, en supposant que l'on dispose du détecteur de défaillances Σ, en appliquant la Proposition 3 on obtient :

Proposition 7 *Dans un système asynchrone avec le détecteur de défaillances Σ dans lequel les processus une fois créés n'ont pas de défaillances il est possible de résoudre le problème du consensus*

[4] Dans la terminologie de [13], il s'agit de processus "initially dead".

6 Systèmes partiellement synchrones

Dans [12], les auteurs proposent un algorithme adapté de [2] qui permet de réaliser une élection de leader ultime dans un système partiellement synchrone avec au moins une source ultime sans la connaissance du nombre de processus.

Par contre, un argument simple de partition, montre que l'existence d'une source ultime n'est pas suffisante pour permettre d'implémenter un registre atomique. En résumant :

Proposition 8 *Dans $PSync + ESource$:*

- *on peut réaliser une élection de leader ultime même si le nombre de processus est inconnu,*
- *il n'existe pas d'algorithme pour résoudre le consensus ou pour implémenter un registre atomique si le nombre de processus n'est pas connu.*

Si maintenant on suppose que le système partiellement synchrone contient une source, alors l'algorithme de la Figure 4 implémente Σ. On déduit alors de la Proposition 3 :

Proposition 9 *Dans $PSync + Source$ il est possible de résoudre le problème du consensus et d'implémenter un registre atomique.*

/ output est la sortie du détecteur de défaillances */*
Code pour p :
Tâche 0 :
1 **tous les** δ
2 **send** $(ALIVE, p)$
Tâche 1 :
3 **upon receive** $(ALIVE, q)$
4 **then** $S := S \cup \{q\}$
Tâche 2 :
5 **tous les** 2δ
6 $output := S$
7 $S := \emptyset$

FIG. 4 – Implémentation de Σ dans un système avec une source.

7 Conclusion et perspectives.

On a étudié dans cet article dans quelle mesure les trois problèmes fondamentaux que sont le consensus, l'implémentation des registres et l'élection de leader ultime peuvent être résolus si on ne connaît pas le nombre de processus. Certains des résultats d'impossibilité peuvent au premier abord paraître étonnants et montrent que la connaissance du nombre de processus est une information nécessaire dans de nombreux cas en particulier en ce qui concerne le problème de l'implémentation de registres atomiques.

Cependant, la portée de ces résultats est surtout théorique. Il reste un travail important à faire pour donner à ces résultats une portée pratique. Il faudrait d'abord améliorer le modèle de façon à donner, par exemple, un sens plus pratique à la possibilité de communiquer avec *tous* les processus vivants quand on ne connaît pas le nombre de processus. Pour cela les approches décrites dans [6, 14] peuvent être un premier pas dans ce sens. Il faudrait ensuite développer des algorithmes efficaces dans ces modèles.

Références

[1] M. K. Aguilera. A pleasant stroll through the land of infinitely many creatures. Technical report, ACM SIGACT News Distributed Computing Column, 2004.

[2] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. On implementing omega with weak reliability and synchrony assumptions. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing*, pages 306–314, Boston, Massachusetts, USA, July 2003.

[3] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1) :124–142, 1995.

[4] R. Baldoni, M. Bertier, M. Raynal, and S. T. Piergiovanni. Looking for a definition of dynamic distributed systems. In V. E. Malyshkin, editor, *PaCT*, volume 4671 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 2007.

[5] M. Ben-Or. Another advantage of free choice : Completely asynchronous agreement protocols. In *Proceedings of the 2nd ACM Symposium on Principles of Distributed Computing*, pages 27–30, Aug. 1983.

[6] D. Cavin, Y. Sasson, and A. Schiper. Consensus with unknown participants or fundamental self-organization. In I. Nikolaidis, M. Barbeau, and E. Kranakis, editors, *ADHOC-NOW*, volume 3158 of *Lecture Notes in Computer Science*, pages 135–148. Springer, 2004.

[7] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4) :685–722, July 1996.

[8] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2) :225–267, Mar. 1996.

[9] C. Delporte-Gallet, H. Fauconnier, and R. Guerraoui. Shared memory vs. message passing. Technical Report IC/2003/77, EPFL, Dec. 2003. Availabe at http ://icwww.epfl.ch/publications/.

[10] C. Delporte-Gallet, H. Fauconnier, and R. Guerraoui. (almost) all objects are universal in message passing systems. In P. Fraigniaud, editor, *DISC*, volume 3724 of *Lecture Notes in Computer Science*, pages 184–198. Springer, 2005.

[11] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, V. Hadzilacos, P. Kouznetsov, and S. Toueg. The weakest failure detectors to solve certain fundamental problems in distributed computing. In S. Chaudhuri and S. Kutten, editors, *PODC*, pages 338–346. ACM, 2004.

[12] A. Fernández, E. Jiménez, and M. Raynal. Eventual leader election with weak assumptions on initial knowledge, communication reliability, and synchrony. In *DSN*, pages 166–178. IEEE Computer Society, 2006.

[13] 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.

[14] F. Greve and S. Tixeuil. Knowledge connectivity vs. synchrony requirements for fault-tolerant agreement in unknown networks. In *DSN*, pages 82–91. IEEE Computer Society, 2007.

[15] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. J. Mullender, editor, *Distributed Systems*, chapter 5, pages 97–145. Addison-Wesley, 1993.

[16] M. Herlihy and J. M. Wing. Linearizability : a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3) :463–492, June 1990.

[17] A. Israeli and M. Li. Bounded time-stamps. *Distributed Computing*, 6(4) :205–209, July 1993.

[18] W.-K. Lo and V. Hadzilacos. Using failure detectors to solve consensus in asynchronous shared memory systems. In G. Tel and P. Vitányi, editors, *Proceedings of the eighth International Workshop on Distributed Algorithms*, volume 857 of *LNCS*, pages 280–295. Springer-Verlag, Sept. 1994.

[19] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[20] A. Mostéfaoui and M. Raynal. Solving consensus using Chandra-Toueg's unreliable failure detectors : A general quorum based approach. In *Proceedings of the 13th International Symposium on Distributed Computing*, LNCS 1693, pages 49–63. Springer-Verlag, Sept. 1999.

[21] P. Vitanyi and B. Awerbuch. Atomic shared register access by asynchronous hardware. In *Proceedings of the 27th Symposium on Foundations of Computer Science*, pages 233–246, 1986.