Consensus byzantin et hypothèses de synchronie

Olivier Baldellon

Consensus byzantin et hypothèses de synchronie

Olivier Baldellon
École Normale Supérieure de Cachan, antenne de Bretagne
Master informatique spécialité recherche en informatique (IRISA)

Équipe ASAP – IRISA
Encadrants : Michel Raynal et Achour Mostefaoui

Le 4 juin 2010

Table des matières

Introduction

Parmi les nombreux intérêts de l'informatique répartie, un des principaux est de permettre l'existence de services tolérants aux fautes. En effet, supposons qu'un service (le site d'une banque par exemple) ne s'exécute que sur une seule machine, la probabilité que cette dernière soit défaillante (à cause de problèmes matériels par exemple) est relativement grande. Et donc la probabilité que le site ne soit plus accessible est, en pratique, beaucoup trop importante.

Il est donc courant que les services informatiques s'exécutent sur de nombreuses machines. Ainsi, si l'une d'entre elles tombe en panne, le service continuera de manière transparente pour l'utilisateur.

Cependant, cette approche apporte un nouveau problème non trivial : les différentes machines doivent proposer des services cohérents. Dans l'exemple de la banque en ligne, si je consulte mon compte depuis Rennes, je dois avoir le même résultat que si je le consulte depuis Lyon. Une telle cohérence n'est possible que si les différentes machines qui proposent le service effectuent les mêmes opérations dans le même ordre.

Il a été démontré que pour pouvoir implémenter de tels services tolérants aux fautes, il est nécessaire de savoir résoudre le problème du consensus. De manière informelle, un protocole de consensus est un protocole dans lequel initialement, chaque processus propose une valeur et qui vérifie trois propriétés :

1. Chaque processus décide au bout d'un temps fini une valeur (Terminaison).
2. Si deux processus décident, alors ils décident la même valeur (Accord).
3. Si un processus décide une valeur, alors cette valeur a été proposée initialement par un processus (Validité).

On sait depuis longtemps que l'on peut résoudre le problème du consensus lorsque tous les liens entre les processus (c'est-à-dire les machines) sont synchrones [1], mais qu'il n'existe pas de protocole de consensus tolérant au moins un processus fautif dans un système dans lequel tous les liens sont asynchrones [6]. La question qui se pose naturellement est alors de connaître le nombre de liens synchrones à partir duquel on est capable de résoudre le problème du consensus.

Nous nous concentrerons dans ce rapport tout particulièrement sur le problème du consensus en présence de processus byzantins avec l'utilisation de signatures (à base de cryptographie asymétrique). Les processus byzantins sont souvent utilisés pour modéliser des processus malicieux qui tentent de faire échouer le consensus. Ces derniers ne suivent donc pas forcément le même protocole que les processus corrects.

Notre rapport se découpera en cinq parties. La première fera un bref rappel de l'état de l'art, puis nous enchaînerons sur certaines définitions afin de pouvoir poser rigoureusement le problème.

1. Un lien entre deux processus p et q est dit synchrone si le délai entre l'envoi d'un message par p et sa réception par q est borné par une constante δ connue *a priori*.

Ensuite, nous détaillerons puis nous améliorerons un protocole nécessitant peu d'hypothèses de synchronie pour résoudre le problème du consensus byzantin avec signatures. En effet, ce dernier nécessite seulement d'avoir un processus correct possédant t liens synchrones avec d'autres processus correct, où t représente le nombre maximal de processus fautifs (on parle de $[t+1]$bi-source). Nos deux nouveaux protocoles termineront plus rapidement dans le meilleur des cas (2 étapes au lieu de 6 étapes). La première amélioration nécessite moins d'un tiers de processus fautifs (ce qui est optimal dans le cas du consensus byzantin) mais termine plus rapidement uniquement si tous les processus sont corrects ; la seconde modification peut terminer rapidement même en présence de processus byzantins, mais nécessite moins d'un cinquième de processus fautifs.

Dans une quatrième partie nous donnerons une condition nécessaire et suffisante sur le nombre de liens synchrones et leur répartition pour résoudre le consensus byzantin avec signatures et nous prouverons ce résultat. Nous discuterons ensuite sur les conséquences de ce théorème pour montrer que la présence d'une $[t+1]$bi-source est nécessaire (et suffisante) pour un certaine classe d'hypothèses sur les liens synchrones ; plus précisément, ces hypothèses sont de la forme « Il y a un nombre b de $[x]$bi-sources dans le système. ».

Enfin, dans une cinquième partie nous donnerons le début d'une démonstration pour un résultat équivalent dans le cas du consensus byzantin sans signatures et nous commencerons une discussion similaire sur la même classe d'hypothèses.

1 État de l'art

Le problème du consensus et les comportements byzantins ont été introduits par Lamport, Shostacket et Pease en 1982 [9, 10]. Ce sont dans ces mêmes articles qu'il a été prouvé qu'il est nécessaire et suffisant d'avoir $n > 3t$ (où n est le nombre total de processus et t le nombre maximal de processus fautifs) pour résoudre le problème du consensus byzantin dans un environnement ne contenant que des liens synchrones. Ce résultat implique trivialement l'impossibilité du consensus byzantin pour $n \leq 3t$ avec des hypothèses de synchronie quelconques (système asynchrone, présence ou non de bi-sources, etc.).

Il a cependant été démontré que dans un système dans lequel tous les liens sont asynchrones, le problème du consensus ne peut être résolu. Ce résultat, qui est sûrement l'un des plus importants de l'algorithmique réparti, a été démontré par Fisher, Linsh et Paterson. Il est donc connu sous le nom de FLP.

La première approche pour affaiblir les hypothèses de synchronie est de considérer des liens partiellement synchrones, c'est-à-dire des liens qui ne seront synchrones non pas depuis le début, mais seulement à partir d'un certain temps non connu *a priori*. La notion de lien partiellement synchrone a été introduite par Dwork, Lynch et Stockmeyer dans [5].

Une autre approche a été de limiter le nombre de liens partiellement synchrones. Aguilera, Delporte-Gallet, Fauconnier et Toueg ont tout d'abords montré qu'il était suffisant, dans le cas du consensus avec crashs, d'avoir un processus

correct p dont tous les liens sont partiellement synchrones en sortie [1] (c'est-à-dire que pour tout processus q, le lien $p \to q$ est synchrone). On appelle un tel processus une $\diamond$source. Ces mêmes auteurs ont ensuite démontré que, dans le cas du consensus byzantin, [3], il était suffisant d'avoir un unique processus p (que l'on qualifiera de $\diamond$bi-source) possédant tous ses liens partiellement synchrones (cette fois ci pour tout processus q, les liens $p \to q$ et $q \to p$ sont partiellement synchrones). Les autres liens entre les autres processus pouvant être asynchrones.

Enfin les derniers résultats ont démontré qu'il n'était pas nécessaire que l'unique bi-source ne possède que des liens synchrones. En particulier, dans le cas du consensus avec crashs, la $\diamond$source doit seulement posséder t liens partiellement synchrones en sortie [2]. Dans la cas du consensus byzantin avec signatures, Hamouma, Mostefaoui et Trédan ont montré qu'une $\diamond[t+1]$bi-source (un processus correct possédant $t+1$ liens partiellement synchrones avec d'autres processus corrects) était suffisante [7]. De plus, ces mêmes auteurs ont montré (dans un papier inédit) que dans le cas du consensus byzantin sans signatures, une $\diamond[2t+1]$bi-source suffisait.

Si la présence de liens synchrones permet de résoudre le problème du consensus, cela est surtout dû au fait qu'ils permettent aux processus corrects de détecter les crashs. En effet si un lien entre deux processus p et q est synchrone, alors si p envoie un message à q à l'instant T, q recevra le message à l'instant $T+\delta$ et p recevra la réponse à l'instant $T+2\delta$; si p n'a pas reçu la réponse, alors p sait que q a crashé.

C'est pourquoi une nouvelle approche a été introduite pour définir la limite permettant de résoudre le problème du consensus. Elle consiste à modéliser l'information que chaque processus possède sur le système par un oracle. Cette notion a été introduite par Chandra, Hadzilacos et Toueg en 1996 [4] dans les cas des systèmes avec crashs. Elle a été ensuite généralisée pour les systèmes byzantins par Kihlstrom, Moser et Melliar-Smith en 2003 [8].

2 Modèles et définitions

Dans cette partie, nous présenterons différentes définitions sur les environnements répartis.

2.1 Modèle de système

Le système que nous utiliserons dans ce rapport est un ensemble de n processus distincts $\Pi = \{p_1, \ldots, p_n\}$. Nous faisons de plus l'hypothèse d'un réseau point-à-point, c'est-à-dire que pour tout couple de processus $(p, q) \in \Pi^2$, il existe un lien direct entre p et q que nous noterons dorénavant $p \to q$.

Liens fiables Les liens connectant deux processus sont supposés fiables. Dit autrement, si un processus envoie un message à un autre processus, alors le message sera reçu au bout d'un temps fini.

Connections point-à-point Pour éviter qu'un processus ne tente de se faire passer pour un autre processus, nous considérons uniquement des connections point-à-point. Ainsi, chaque processus peut connaître l'expéditeur de chaque message qu'il reçoit (si p reçoit un message provenant du lien $q \to p$, alors p sait que l'expéditeur de ce message est q).

En pratique, ces connections point-à-point peuvent être facilement implémentées avec de la cryptographie symétrique. Plus précisément chaque processus p_i possède n clés secrètes $c_{i,1}, \ldots, c_{i,n}$ telle que pour tout couple de processus p_i, p_j, on ait $c_{i,j} = c_{j,i}$ (ce qui signifie que les deux processus possèdent une clé en commun), et aucun processus distinct de p_i et p_j ne connaît cette clé.

2.2 Modèle de fautes

Les protocoles répartis étant souvent utilisés pour implémenter des services tolérants aux fautes, il est nécessaire de définir exactement ce que nous entendons par fautes. Nous allons considérer deux types de systèmes, ceux autorisant les fautes de crash et ceux autorisant les fautes byzantines.

Crashs On dit qu'un processus p crashe ou commet une faute de crash si le processus p s'arrête (alors qu'il ne le devrait pas) pendant l'exécution d'un protocole.

Byzantins On dit qu'un processus p commet une erreur byzantine s'il se ne comporte pas selon le protocole. Un tel processus est appelé *processus byzantin*. En particulier les fautes byzantines sont utilisées pour modéliser des comportements malicieux à partir desquels un processus va tenter de faire échouer un calcul.

Nous nous concentrerons dans ce rapport sur les fautes byzantines. On notera de plus t le nombre maximal de processus fautifs.

2.3 Synchronie

Processus synchrones On supposera dans la suite de ce papier que les opérations internes effectuées par les processus sont instantanées. En particulier, si un processus p reçoit un message à un instant τ, il enverra la réponse à ce message immédiatement (à l'instant τ donc).

Liens synchrones On dit qu'un lien $p \to q$ est synchrone s'il existe une constante $\delta_{p\to q}$ telle que tout message envoyé par p à l'instant t sera reçu par q avant l'instant $t + \delta_{p\to q}$. Sans mention contraire, on fait l'hypothèse que si le lien $p \to q$ est synchrone, alors le lien $q \to p$ l'est aussi. Dorénavant on notera δ le maximum de l'ensemble :

$$\{\delta_{p\to q} \backslash\ p \to q \text{ est synchrone}\}$$

Ainsi, si p est q sont synchrones, tout message envoyé par p à l'instant t sera reçu par q avant l'instant $t + \delta$.

Liens partiellement synchrones On dit qu'un lien $p \to q$ est partiellement synchrone si le lien $p \to q$ est synchrone à partir d'un certain temps non connu *a priori*. À la différence des liens synchrones, la constante δ n'est pas connue.

Liens asynchrones On dit qu'un lien $p \to q$ est asynchrone s'il n'est ni synchrone ni partiellement synchrone.

$[x]$**bi-source** On dit qu'un processus p est une $[x]$bi-source si ce processus est correct et s'il existe x processus $p_1, \ldots, p_x$ corrects (dont p) tel que pour tout i inférieur à x, le lien $p \to p_i$ soit synchrone.

Cette définition peut paraître surprenante, car dans la littérature, la définition classique d'une $[x]$bi-source n'impose pas que les x voisins soient corrects. Cette nouvelle définition peut se justifier par le fait que nous travaillons ici essentiellement avec des fautes byzantines alors que la définition originale été créée pour les crashs.

La différence principale est que dans un système avec crashs, si p est correct et q est fautif, un lien synchrone entre p et q permet à p de détecter les fautes de q (voir partie suivante), par contre si le lien est asynchrone p n'a aucune information sur l'état de q. Or, dans le cas d'un système byzantin, si q est byzantin et si le lien $p \to q$ est synchrone, alors q peut volontairement attendre avant d'envoyer sa réponse et faire ainsi croire à p que le lien $p \to q$ est asynchrone. Pour résumer, un lien synchrone entre un processus correct et un processus fautif n'apporte aucune information supplémentaire (par rapport à un lien asynchrone) dans le cas d'un système byzantin.

$[x]$**IO-processus** On dit qu'un processus p est un $[x]$IO-processus si ce processus est correct et s'il existe x processus $p_1, \ldots, p_x$ corrects (dont p) tel que pour tout i inférieur à x, le lien $p \to p_i$ est synchrone (mais le lien $p_i \to p$ ne l'est pas forcement) et x processus $q_1, \ldots, q_x$ corrects (dont p) tels que les liens $q \to_i p$ sont synchrones (cette fois encore, aucune hypothèses ne sont faite pour les liens $p \to q_i$).

On définit de même les $\diamond[x]$bi-sources et les $\diamond[x]$IO-processus en remplaçant dans les définitions les liens synchrones par des liens partiellement synchrones.

2.4 Oracles et détection de fautes

2.4.1 Synchronie et détection de fautes

L'intérêt théorique d'avoir un système possédant des liens synchrones est que cela permet aux processus corrects de détecter des crashs. En effet, considérons deux processus p et q, et supposons que le lien $p \to q$ soit synchrone. Le processus p peut alors savoir à chaque instant si q est défaillant ou s'il ne l'est pas. Pour cela p envoi un message à q à l'instant t et attend la réponse ; si p reçoit une réponse avant $t + 2\delta$, alors q est correct, au contraire si à l'instant $t + 2\delta$ le processus p n'a toujours pas de réponse, p sait que q est défaillant.

Par contre si le lien $p \to q$ est asynchrone, p n'a absolument aucune possibilité de savoir si q est correct ou si q est en panne ; en effet même si p n'a pas reçu

de nouvelle de q depuis un temps très long, il est toujours possible que q soit correct et que cette absence de nouvelle soit due à l'asynchronie du système. Cette ambiguïté permanente est la base du raisonnement de FLP qui montre que l'on ne peut pas résoudre le problème du consensus dans un système purement asynchrone.

Dans le cas où les seules fautes autorisées sont les crashs, il est intéressant de noter qu'un lien synchrone entre un processus p et un processus fautif q, permet à p d'avoir de l'information sur l'état du système, même si p ne connaît pas le lien synchrone (mais il sait qu'il en existe un). Par exemple prenons trois processus p, q et r. On suppose que p sait que 1) un des deux processus q ou r est fautif et 2) un des deux liens $p \to q$ et $p \to r$ est synchrone. Si p envoie à l'instant τ un message à r et q et attend la réponse, p est censé recevoir la réponse provenant du lien synchrone avant $\tau + 2\delta$. Si ce n'est pas le cas, il sait qu'un des processus r ou q a crashé, et il lui suffit d'attendre la première réponse pour savoir lequel des deux est correct. Si par exemple il reçoit une réponse de r, il sait que r n'a pas crashé (puisqu'il répond) donc r est correct et q est fautif.

Cependant, dans un système avec fautes byzantines, la présence d'un lien synchrone entre un processus correct et un processus fautif n'apporte pas d'information. En effet, si l'on reprend l'exemple précédent, après que p ait reçu la réponse de r, le processus p ne peut pas savoir si le lien $p \to q$ est synchrone et si q est fautif (ce qui expliquerait l'absence de nouvelle), ou si le processus r est byzantin et retarde l'envoi de la réponse à p pour faire croire que le lien $r \to p$ est asynchrone.

2.4.2 Oracles

On vient de voir qu'un des intérêts principaux de la présence de liens synchrones est que cela permet d'avoir des informations sur l'état du système. Cette information peut être abstraite par l'utilisation d'oracles, aussi appelés *détecteurs de fautes*.

Par exemple, en considérant la discussion de la partie précédente, il est aisé de constater que l'on peut implémenter une détecteur de fautes $\diamond\mathcal{P}$ dans un système dans lequel tous les liens sont synchrones, avec $\diamond\mathcal{P}$ définit par :

1. à partir d'un certain temps, chaque processus fautif (crash) sera suspecté (de manière définitive) par tous les processus corrects ;
2. à partir d'un certain temps, aucun processus correct ne sera suspecté.

Il a été démontré que dans un système dans lequel plus de la moitié des processus sont corrects, un oracle de la classe $\diamond\mathcal{P}$ n'était pas nécessaire, contrairement aux oracles de la classe $\diamond\mathcal{W}$ qui sont nécessaires et suffisants pour résoudre le consensus.

Définition 2.1 *On dit qu'un oracle $\mathcal{O}$ appartient à la classe $\diamond\mathcal{W}$ si :*

1. *à partir d'un certain temps, chaque processus fautif (c'est-à-dire ceux qui crashent pendant l'exécution) est suspecté par au moins un processus correct ;*
2. *à partir d'un certain temps, il existe un processus correct qui n'est plus suspecté par aucun processus correct.*

Cependant, ce travail sur les oracles a surtout été fait dans le cadre des fautes de crash. En effet, s'il est « relativement » facile de détecter un crash (il suffit de détecter que le processus ne répond plus), il est par contre beaucoup plus difficile de détecter un comportement byzantin, qui par définition peut correspondre à tout et à n'importe quoi. En particulier certaines de ces fautes peuvent être non détectables, ou peuvent ne pas permettre de savoir quel processus est à l'origine de la faute. Par exemple si un processus byzantin p envoie un message m à tous sauf à un processus q avec lequel il est asynchrone, alors q n'a aucun moyen de détecter que ce message n'a pas été envoyé (en effet le lien $p \rightarrow q$ étant asynchrone, il se peut que q n'ait pas reçu le message car ce dernier est toujours en transit.), une telle faute est dite indétectable. De plus si un processus p affirme avoir reçu un message m de la part de q à un troisième processus r, et si q affirme ne pas avoir envoyé de message, r n'a aucun moyen de savoir qui entre p et q est fautif. Une telle faute est dite indistinguable.

Ainsi, les détecteurs de fautes pour les systèmes byzantins ne considèrent qu'un sous-ensemble particulier de fautes appelées fautes détectables.

Définition 2.2 *On dit qu'une faute commise par un processus byzantin est une faute détectable si l'une des trois propriétés suivantes est vérifiée :*

- ***omission*** *: un processus p commet une faute d'omission s'il n'envoie à aucun processus un message qu'il est censé envoyer.*
- ***commission (1)*** *: un processus p commet une faute de commission si p envoie un message m à un premier processus et un message m' à un second processus avec $m \neq m'$ alors que selon le protocole on aurait dû avoir $m = m'$.*
- ***commission (2)*** *: un processus p commet une faute de commission si p envoie un message m qui n'aurait pas pu être envoyé par un processus qui exécute le protocole correctement, compte tenu des messages que p a déjà reçus.*

On définit alors une classe de détecteur de fautes byzantines : $\diamond\mathcal{W}(Byz)$.

Définition 2.3 *On définit la classe $\diamond\mathcal{W}(Byz)$ des détecteurs de fautes qui vérifient les deux propriétés suivantes (on rappelle que t est le nombre maximal de processus fautifs) :*

1. *à partir d'un certain temps, chaque processus qui commet une faute détectable est suspecté par au moins $t + 1$ processus corrects ;*
2. *à partir d'un certain temps, il existe un processus correct qui n'est plus suspecté par aucun processus correct.*

Il a été démontré qu'un détecteur de fautes de la classe $\diamond\mathcal{W}(Byz)$ est suffisant pour résoudre le problème du consensus byzantin à partir du moment où il y a strictement moins d'un tiers de processus fautifs [8].

2.5 Le problème du consensus

2.5.1 Définition classique

On considère un ensemble de processus $\Pi = \{p_1, \ldots, p_n\}$, on associe à chaque processus une valeur v_i. On dit que v_i est la valeur proposée par p_i. On appelle protocole de consensus tout protocole qui vérifie les trois propriétés suivantes :

– **Terminaison** : tout processus correct décide au bout d'un temps fini ;
– **Accord** : si deux processus corrects décident, alors ils décident la même valeur ;
– **Validité** : si un processus correct décide, alors il décide une valeur parmi celles proposées.

Si les processus ne peuvent initialement proposer que 0 ou 1, on parle de *consensus binaire*. Si par contre l'ensemble des valeurs proposables contient plus de deux valeurs, alors on parle de *consensus multivalué*.

2.5.2 Variantes byzantines et remarques

Avec l'introduction des comportements byzantins, la propriété de validité précédente n'est plus suffisante. En effet, on veut éviter autant que possible que la valeur finale décidée provienne d'une proposition byzantine. Une autre définition pour la validité est alors utilisée :

– **Validité** : si tous les processus corrects proposent une valeur v, alors seul la valeur v peut être décidée par un processus correct.

On appelle alors *consensus byzantin* le problème du consensus (avec la nouvelle définition de la validité) en présence de processus byzantins.

Nous nous concentrerons dans ce rapport sur deux problèmes particuliers : le problème du consensus byzantin multivalué avec signatures et le problème du consensus byzantin multivalué sans signatures.

Les signatures permettent de limiter le pouvoir des processus byzantins, en effet si un processus p envoie un message m, alors un processus byzantin ne pourra pas prétendre que p a envoyé m', le message m' n'étant pas signé par p.

3 Étude d'un protocole de consensus byzantin avec signatures

Dans le cas du consensus byzantin avec signatures (les processus signent leur messages grâce à de la cryptographie asymétrique) il a été démontré, comme nous l'avions indiqué précédemment, dans [7] qu'une $\diamond[t+1]$bi-source suffisait pour résoudre le problème du consensus. Ce papier propose en effet un algorithme qui, s'il est intéressant en théorie, n'est pas vraiment conçu pour un usage pratique. En particulier 6 étapes au moins sont nécessaires avant que les processus décident. Dans la suite de cette partie, nous détaillerons ce protocole et nous proposerons deux améliorations pour terminer plus rapidement dans le meilleur des cas.

3.1 Le protocole

3.1.1 Principes généraux

La structure de ce protocole est relativement classique. Elle consiste en une suite de rondes. Chaque ronde étant dirigée par un coordinateur. Au début de la ronde ρ, le processus coordinateur p_ρ propose une valeur qui sera décidée par tous les processus si p_ρ est une $[t+1]$bi-source. Si par contre p_ρ n'est pas une

bi-source, il se peut que les processus ne décident pas pendant cette ronde ; dans ce cas une nouvelle ronde $\rho + 1$ commence avec un autre coordinateur.

3.1.2 Signatures et authentification des messages

Généralement, un protocole réparti se résume en une série d'étapes, chaque étapes se déroulant en deux temps. Tout d'abord chaque processus envoie un message à tous les autres processus ; ensuite chaque processus attend pour recevoir un maximum de messages. En pratique, dès qu'un processus a reçu $n - t$ messages, il passe à l'étape suivante. En effet, étant donné qu'il y a t processus fautifs, il se peut qu'ils n'aient pas envoyé leur message lors de cette étape. Donc, attendre plus de $n - t$ messages peut bloquer l'exécution d'un processus.

Afin d'empêcher les processus byzantins d'avoir un comportement non conforme au protocole, chaque processus doit prouver que le message qu'il envoie à l'étape e_{i+1} est compatible avec les messages qu'il a reçus lors de l'étape précédente e_i. Pour cela à chaque fois qu'un processus p_i envoie un message m_i à l'ensemble des processus, il envoie le message signé avec sa clé privée $\langle m \rangle_i$ et il y joint les $n - t$ messages reçus à l'étape précédente. Ces $n - t$ messages étant signés par leur expéditeur d'origine, un processus byzantin ne peut pas les modifier.

3.1.3 Déroulement du protocole

Pendant l'exécution du protocole, le processus p_i utilise la variable est_i pour stocker la valeur qu'il pense décider. Ainsi, au commencement du protocole, chaque processus correct aura sa variable est_i initialisée à la valeur qu'il propose ; à la fin de l'exécution du protocole, la variable est_i correspondra à la valeur décidée.

Le protocole consiste en trois *threads* qui tournent en parallèle, le premier *thread*, le principal, correspond à la succession des rondes. Le second, appelé *thread* de coordination, est le *thread* dans lequel les processus coordinateurs vont tenter de forcer le consensus en proposant une valeur. Enfin le dernier *thread* est celui qui va faire terminer un processus lorsqu'un message DEC est reçu.

Décision Pour forcer la terminaison de l'ensemble des processus corrects à partir du moment où l'un d'entre eux décide, chaque processus qui décide envoi une message DEC(v) à tous les processus. Ce message étant authentifié (voir partie 3.1.2 page 9), chaque processus correct le recevant a la preuve qu'il doit décider et donc décide et fait suivre ce message.

Ainsi, la propriété de terminaison est vérifiée à partir du moment où un processus correct termine. Ce qui suit décrit le *thread* principal.

Initialisation Chaque processus propose une valeur initiale. La propriété de validité (si tous les processus corrects proposent la même valeur, alors cette valeur est la seule qui puisse être décidée) impose que si $n - t$ processus proposent v, alors v sera décidée. Soit v une valeur proposée initialement par tous les processus corrects, et p un processus. Le processus p n'attendra que $n - t$ messages

Protocole 1 Protocole de consensus terminant avec une $\diamond[t+1]$bi-source

▷ **Initialisation**

1: $r_i \leftarrow 0$; $\Delta_i[1..n] \leftarrow 1$;
2: *send* INIT(v_i) to all ;
3: **Wait until**(INIT received from $n-t$ processes)
4: **if** $\exists v$ received at least $n-2t$ **then** $est_i \leftarrow v$ **else** $est_i \leftarrow v_i$

▷ **Thread principal**

5: **loop**
6: $coord \leftarrow (r_i \mod n)$; $r_i \leftarrow r_i + 1$ ▷ Ronde r_i

// Étape 1 : Chaque processus donne sa valeur au coordinateur puis attend la réponse.

7: *send* QUERY(r_i, est_i) to p_{coord} ; *set_timer*$(\Delta_i[C])$
8: **Wait until**(received COORD(r_i, est) received or time_out)
9: **if** timer times out **then** $\Delta_i[C] \leftarrow \Delta_i[C] + 1$; $aux_i \leftarrow \perp$
10: **else** disable_timer ; $aux_i \leftarrow est$ **endif**

// Étape 2 : Chaque processus fait suivre si possible la valeur provenant du coordinateur.

11: *send* RELAY(r_i, aux_i) to all
12: **Wait until** RELAY$(r_i, *)$ received from $n-t$ proc. **store values** in V_i
13: **if** $V_i \setminus \{\perp\} = \{v\}$ **then** $aux_i \leftarrow v$ **else** $aux_i \leftarrow \perp$ **endif**

// Étape 3 : Les processus vérifient alors que les valeurs sont cohérentes.

14: *send* FILT1(r_i, aux_i) to all
15: **Wait until** FILT1$(r_i, *)$ received from $n-t$ proc. **store val.** in V_i
16: **if** $V_i = \{v\}$ **then** $aux_i \leftarrow v$ **else** $aux_i \leftarrow \perp$ **endif**

// Étape 4 : Si tout s'est bien passé, les processus décident.

17: *send* FILT2(r_i, aux_i) to all
18: **Wait until** FILT2$(r_i, *)$ received from $n-t$ proc. **store val.** in V_i
19: **if** $V_i = \{v\}$ **then** *send* DEC(v) to all ; **return** v
20: **if** $V_i = \{v, \perp\}$ **then** $est_i \leftarrow v$
21: **end loop**

▷ **Thread de coordination**

22: **upon** receipt of QUERY(r, est) for the first time : send COORD(r, est) to all

▷ **Threads de décision**

23: **upon** receipt of DEC(est) *send* DEC(est) to all ; **return** est

à la ligne 3 du protocole 1, et sur ces $n-t$ messages, il se peut que t d'entres eux proviennent de processus fautifs. Cependant, on est sûr que p recevra au moins $n-2t$ fois la valeur v. Dans ce cas, pour satisfaire la propriété de terminaison, p met à jour la variable $est_i \leftarrow v$.

L'étape d'initialisation n'est exécutée qu'une fois, et le reste du protocole consiste en une suite de rondes, elles mêmes découpées en quatre étapes : Coordination, Relai, Premier filtrage et Second filtrage.

À l'intérieur de chaque ronde, le processus coordinateur va proposer une valeur que nous noterons le temps de l'explication v. Les différentes étapes décrites ci-après ont pour but de s'assurer que le processus coordinateur n'a proposé qu'une valeur et qu'un nombre suffisant de processus l'ont reçue. Si un processus correct se rend compte que ces deux propriétés sont assurées, alors il saura que la propriété d'accord sera valide (puisque tous les processus, corrects ou non, auront mis leur variable est_i à v), et il pourra décider (permettant alors la terminaison). La variable aux_i servira à stocker la valeur présumée de v.

Coordination Lorsqu'un processus p_i atteint la ronde ρ, il se peut que, du fait de l'asynchronie, le processus coordinateur n'ait pas encore atteint la ronde ρ. Ainsi, au lieu d'attendre que $p_{coord(\rho)}$ atteigne la ronde ρ et envoie sa valeur estimée $est_{coord(\rho)}$, chaque processus en commençant la ronde ρ envoie sa variable est_i (ligne 7) au processus coordinateur qui se chargera (grâce au *thread* de coordination) de faire suivre la première valeur qu'il reçoit à tous les autres processus.

Le processus coordinateur sait que la valeur qu'il transmet est correcte grâce au mécanisme de certificat (voir partie 3.1.2 page 9).

Dans le cas où le processus coordinateur est une $[t+1]$bi-source correcte, au moins $t+1$ processus corrects vont recevoir à la ligne 8 le message du coordinateur.

Relai Cette étape permet de diffuser davantage la valeur proposée par le coordinateur. Si un processus a reçu cette valeur, il la fait suivre à tous les autre processus, sinon il envoie la valeur $\perp$ pour préciser qu'il n'a rien reçu de la part du coordinateur.

Dans le cas où le processus coordinateur est une $[t+1]$bi-source (et donc correct), chaque processus recevra (ligne 12) au moins une fois la valeur v (en effet sur les $n-t$ messages reçus, au moins un provient des $t+1$ processus corrects synchrones avec le coordinateurn car $n > 3t$).

De plus, si le coordinateur était correct, les processus byzantins peuvent faire suivre la valeur v ou ne pas la faire suivre, mais en aucun cas il ne peuvent faire suivre une autre valeur (car il ne sont pas capables de générer un message signé avec la signature du coordinateur).

Premier filtrage Si le processus coordinateur est une bi-source correcte, alors tous les processus corrects p_i ont la variable aux_i égale à v. Cependant, si le coordinateur était byzantin, il a pu proposer deux valeurs distinctes v_1 et v_2. Cette étape permet aux processus corrects de s'assurer que si p_i et p_j sont deux processus corrects et si $aux_i \neq \perp$ et $aux_j \neq \perp$, alors $aux_i = aux_j$.

On note alors V_i l'ensemble des valeurs aux_j reçues à cette étape par le processus p_i (ligne 15). S'il existe une valeur v telle que $V_i = \{v\}$, le processus p_i dans ce cas exécute $aux_i \leftarrow v$, sinon, p_i a découvert que le processus coordinateur n'était pas une bi-source (au moins pendant cette ronde) et va tenter d'empêcher la décision lors de cette ronde en mettant aux_i à $\perp$.

Second filtrage Cette ronde, comme les deux précédentes, commence par l'envoi des variables aux_i. Lors de cette étape, nous sommes sûrs que l'ensemble des valeurs aux_i envoyées est de la forme $\{v, \perp\}$ ou $\{v\}$. En effet, pour envoyer une valeur $aux_i = v$ lors de cette étape, un processus doit l'authentifier par $n-t$ signatures, dont $n-2t$ provenant de processus corrects. Or deux ensembles de taille au moins $n-2t$ possèdent au moins un processus correct en commun (car $n > 3t$), processus qui n'a pas pu signer pour deux valeurs distinctes.

Si un processus p_i reçoit un ensemble V_i de la forme $\{v\}$, alors il considère qu'il peut décider. Sinon, l'ensemble des valeurs reçues étant de la forme $\{v, \perp\}$, chaque processus (correct ou non) qui ne décide pas doit mettre est_i à v, et dans ce cas v sera la seule valeur qui pourra être décidée lors de la suite du protocole.

Pour une démonstration plus détaillée des propriétés de terminaison, de validité et de correction, le lecteur pourra se référer à l'article original [7].

3.2 Terminaison rapide dans le meilleur des cas

Le protocole décrit dans la partie précédente termine dans le meilleur des cas relativement lentement. Il est en effet constitué d'une étape d'initialisation et de rondes de 4 étapes. Ainsi, dans le meilleur des cas, le protocole termine en 6 étapes (en comptant l'envoi des messages DEC), ce qui est assez important.

Nous présentons dans cette partie deux améliorations (en terme de nombre d'étapes) de ce protocole . Pour rappel ce protocole est correct avec l'hypothèse $n > 3t$ (où n est le nombre total de processus et t le nombre maximal de processus fautifs) et l'existence d'une $\diamond[t+1]$bi-source.

La première amélioration que nous proposons permet de terminer rapidement si tous les processus sont corrects et si le premier coordinateur est une $[n]$bi-source en seulement deux étapes. Il est cependant nécessaire pour cela que tous les processus proposent initialement la même valeur. De même si lors d'une ronde, tous les processus se comportent de manière correcte et que le processus coordinateur est une $[n]$bi-source, alors la décision ce fera lors de la deuxième (au lieu de la quatrième) étape de cette ronde. Si le meilleur cas n'est pas atteint, ce protocole se comporte comme l'original et tout comme ce dernier, nécessite pour être correct une $\diamond[t+1]$bi-source et $n > 3t$.

Il est cependant possible de proposer une autre amélioration qui termine rapidement même en présence de fautes byzantines. Cette fois ci, la terminaison rapide nécessite que le premier processus coordinateur soit une $[n-t]$bi-source et que tous les processus corrects proposent la même valeur (on remarquera que l'on autorise les comportements byzantins pour les processus fautifs). De même, si lors d'une ronde le processus coordinateur est une $[n-t]$bi-source, la décision se fera lors de la deuxième (au lieu de quatrième) étape de cette ronde. Le prix à

payer pour autoriser les comportements byzantins même dans les cas favorables est que le protocole nécessite dorénavant pour être corrects $n > 5t$.

3.2.1 Terminaison rapide pour n>3t

Cette partie présente un protocole qui termine dans tous les cas avec $n > 3t$ et une $\diamond[t+1]$bi-source. Dans le meilleurs des cas, seules deux étapes sont nécessaires pour atteindre le consensus. Le meilleur des cas étant défini par les trois conditions suivantes :

- tous les processus sont corrects ;
- le coordinateur est une $[n]$bi-source ;
- tous les processus corrects proposent la même valeur.

De même si lors d'une ronde, ces trois conditions sont vérifiées, alors la décision se fera prématurément avant la fin de la ronde (2 étapes au lieu de 4).

Il est aisé de remarquer que ce protocole, dont le pseudo-code se trouve page 14 (protocole 2), est sensiblement le même que l'original et qu'il ne contient que deux changements, le premier, lignes 5 et 26, et le second, lignes 15 et 27. La preuve se fera en deux temps. Nous allons déjà montrer que la première modification conserve les propriétés de terminaison, de validité et d'accord, puis nous montrerons qu'il en est de même après l'ajout de la seconde modification.

Propriété 3.1 *L'ajout de la première modification au protocole initial conserve les propriétés de terminaison, de validité et d'accord. Nous noterons alors $\mathscr{P}$ le protocole ainsi modifié.*

Démonstration : Pour information, les deux lignes de modification sont rappelées page 15 (protocole 3). Intuitivement, ces deux lignes permettent de terminer rapidement lorsque tous les processus proposent la même valeur lors de la phase d'initialisation.

Terminaison Supposons tout d'abord que tous les processus corrects n'envoient pas la même valeur v lors de l'étape d'initialisation, dans ce cas le coordinateur ne sera pas en mesure de générer un certificat contenant n signatures valides de la même valeur et donc le protocole qui s'exécutera sera la même que l'original.

Supposons maintenant que tous les processus corrects envoient la même valeur v. Si un processus correct reçoit le message DEC-FAST(v) alors il terminera et fera suivre le messages à tous les autres processus corrects et dans ce cas tous les processus corrects décideront. Enfin si aucun processus correct ne reçoit le message DEC-FAST, alors dans ce cas, pour les processus corrects, l'exécution sera la même que celle du protocole original.

Validité Il est assez aisé de voir que si tous les processus corrects proposent la même valeur v, alors la seule valeur que le coordinateur pourra proposer sera cette valeur v et donc, seule cette valeur pourra être décidée.

Protocole 2 Terminaison rapide pour n>3t

▷ **Initialisation**

// Étape 0 : Décision rapide si possible

1: $r_i \leftarrow 0$; $\Delta_i[1..n] \leftarrow 1$;
2: *send* INIT(v_i) to all ;
3: **Wait until**(INIT received from $n-t$ processes)
4: **if** $\exists v$ received at least $n-2t$ **then** $est_i \leftarrow v$ **else** $est_i \leftarrow v_i$
5: **if** $i = coord$ and $\exists v$ received n times **then** *send* DEC-FAST(v) to all

▷ **Thread principal**

6: **loop**
7: $coord \leftarrow (r_i \mod n)$; $r_i \leftarrow r_i + 1$ ▷ Ronde r_i

// Étape 1 : Chaque processus donne sa valeur au coordinateur puis attend la réponse.

8: *send* QUERY(r_i, est_i) to p_{coord} ; *set_timer*($\Delta_i[C]$)
9: **Wait until**(received COORD(r_i, est) received or time_out)
10: **if** timer times out **then** $\Delta_i[C] \leftarrow \Delta_i[C] + 1$; $aux_i \leftarrow \perp$
11: **else** disable_timer ; $aux_i \leftarrow est$ **endif**

// Étape 2 : Chaque processus fait suivre si possible la valeur provenant du coordinateur (et terminaison rapide si possible)

12: *send* RELAY(r_i, aux_i) to all
13: **Wait until** RELAY($r_i, *$) received from $n-t$ proc. **store values** in V_i
14: **if** $V_i \setminus \{\perp\} = \{v\}$ **then** $aux_i \leftarrow v$ **else** $aux_i \leftarrow \perp$ **endif**
15: **if** $\exists v$ received n times **then** *send* DEC-FAST2(v) to all

// Étape 3 : Les processus vérifient alors que les valeurs sont cohérentes.

16: *send* FILT1(r_i, aux_i) to all
17: **Wait until** FILT1($r_i, *$) received from $n-t$ proc. **store val.** in V_i
18: **if** $V_i = \{v\}$ **then** $aux_i \leftarrow v$ **else** $aux_i \leftarrow \perp$ **endif**

// Étape 4 : Si tous s'est bien passé, les processus décident.

19: *send* FILT2(r_i, aux_i) to all
20: **Wait until** FILT2($r_i, *$) received from $n-t$ proc. **store val.** in V_i
21: **if** $V_i = \{v\}$ **then** *send* DEC(v) to all ; **return** v
22: **if** $V_i = \{v, \perp\}$ **then** $est_i \leftarrow v$
23: **end loop**

▷ **Thread de coordination**

24: **upon** receipt of QUERY(r, est) for the first time : send COORD(r, est) to all

▷ **Threads de décision**

25: **upon** receipt of DEC(est) *send* DEC(est) to all ; **return** *est*
26: **upon** receipt of DEC-FAST(est) *send* DEC-FAST(est) to all ; **return** *est*
27: **upon** receipt of DEC-FAST2(est) *send* DEC-FAST2(est) to all ; **return** *est*

Protocole 3 Première modification pour n>3t

▷ **Thread principal**

5: **if** $i = coord$ and $\exists v$ received n times **then** *send* DEC-FAST(v) to all

▷ **Threads de décision**

26: **upon** receipt of DEC-FAST(est) *send* DEC-FAST(est) to all ; **return** est

Accord Supposons qu'un processus correct décide après avoir reçu un message DEC-FAST. Dans ce cas, cela signifie que tous les processus corrects ont proposé cette même valeur et la propriété de validité permet de conclure que tous les processus corrects qui décideront, décideront la même valeur. Si par contre aucun processus correct ne reçoit de message DEC-FAST, alors cela signifie que, pour les processus corrects, l'exécution correspondra à celle du protocole original. □

Propriété 3.2 *L'ajout de la seconde modification au protocole $\mathscr{P}$ conserve les propriétés de terminaison, validité et d'accord.*

Démonstration : Pour rappel la seconde modification consiste en l'ajout de deux lignes :

Protocole 4 Seconde modification pour n>3t

▷ **Thread principal**

15: **if** $\exists v$ received n times **then** *send* DEC-FAST2(v) to all

▷ **Threads de décision**

27: **upon** receipt of DEC-FAST2(est) *send* DEC-FAST2(est) to all ; **return** est

Terminaison On suppose tout d'abord qu'un processus correct reçoit un message DEC-FAST2. Dans ce cas, ce même processus correct va faire suivre le message et finalement, les processus qui n'auront pas encore terminé termineront lors de la réception de ce message.

Si par contre aucun processus correct ne reçoit de message DEC-FAST2, cela signifie que le protocole se comportera exactement comme le protocole $\mathscr{P}$ pour lequel la propriété de terminaison a déjà été prouvée.

Validité Si tous les processus corrects proposent la même valeur v, alors l'exécution sera exactement la même que celle de $\mathscr{P}$ et donc cette valeur sera décidée.

Accord On suppose tout d'abord qu'aucun processus correct ne reçoit de message DEC-FAST2. Dans ce cas, l'exécution du protocole étant la même que si le protocole avait été $\mathscr{P}$ la propriété de validité a déjà été démontrée.

Si par contre un processus correct reçoit un tel message, alors cela signifie que n processus ont envoyé le message v lors de l'étape précédente. Dans ce cas tous les processus corrects proposeront v lors de l'étape suivante (FILT1). De plus lors de cette même étape les processus byzantins ne pourront proposer que v ou $\perp$, mais ne pourront pas proposer une autre valeur, car ils sont incapables de générer le certificat correspondant.

Ainsi, lors de l'étape FILT2, chaque processus sera obligé de proposer v et cette valeur sera donc décidée. Ainsi, si DEC-FAST(v) est reçu par un processus correct, seul la valeur v peut être décidée.

□

3.2.2 Terminaison rapide pour n>5t

Il est possible de modifier encore légèrement le protocole pour accepter certaines erreurs byzantines. Ainsi l'exécution pourra quand même se terminer en deux étapes malgré la présence de processus byzantins. Cependant cette modification a un coût car le protocole ne pourra pas supporter plus d'un cinquième de processus byzantin ($n > 5t$). On définit alors le meilleur des cas par les deux propriétés suivantes :

- le coordinateur est une $[n-t]$bi-source ;
- tous les processus corrects proposent la même valeur.

De même, si lors d'une ronde, ces trois conditions sont vérifiées, alors la décision se fera de manière précoce avant la fin de la ronde (2 étapes au lieu de 4). Le pseudo-code de ce protocole est décrit page 17 (protocole 5).

Nous allons montrer que les deux modifications que nous avons rajoutées par rapport à l'algorithme original conservent les propriétés de terminaison, validité et d'accord.

Propriété 3.3 *L'ajout de la première modification au protocole initial conserve les propriétés de terminaison, validité et d'accord. Nous noterons alors $\mathscr{P}$ le protocole ainsi modifié.*

Démonstration : Pour rappel la première modification consiste en l'ajout de deux lignes décrites ci-après (protocole 6). La preuve de cette modification est relativement semblable à celle du protocole précédent. En particulier les preuves de la terminaison et de la validité sont les mêmes.

En ce qui concerne l'accord, supposons qu'un processus correct décide après avoir reçu un message DEC-FAST(v). Dans ce cas, cela signifie que $n-t$ processus ont effectivement proposé cette valeur v et donc en particulier $n-2t$ processus corrects. Montrons alors que cette valeur est la seule que peut proposer le coordinateur.

Pour que le coordinateur puisse proposer une autre valeur que v lors de l'envoi de messages COORD, deux possibilités s'offrent à lui :

- trouver $n-t$ signatures pour le certificat tel que ces $n-t$ signatures ne contiennent pas $n-3t$ signatures de v ;
- trouver $n-t$ signatures qui contiennent plus de $n-3t$ fois une autre valeur que v.

Pour la première possibilité, il y a $n-2t$ processus corrects qui ont proposé v, t processus corrects qui ont pu proposer une autre valeur, et t valeurs proposées par des processus byzantins. Ainsi un processus byzantin ne peut fournir un ensemble de signatures contenant moins de $n-3t$ signatures de v.

De même il ne peut pas trouver une valeur distincte de v qui recueille plus de $n-3t$ signatures, car le coordinateur ne peut trouver que $2t$ signatures ne correspondant pas à v, or $2t < n-3t$.

□

Propriété 3.4 *L'ajout de la seconde modification au protocole $\mathscr{P}$ conserve les propriétés de terminaison, de validité et d'accord.*

Protocole 5 Terminaison en deux étapes dans le meilleur des cas (n>5t)

▷ **Initialisation**

// Étape 0 : Décision rapide si possible

1: $r_i \leftarrow 0$; $\Delta_i[1..n] \leftarrow 1$;
2: *send* INIT(v_i) to all ;
3: **Wait until**(INIT received from $n-t$ processes)
4: **if** $\exists v$ received at least $n-2t$ **then** $est_i \leftarrow v$ **else** $est_i \leftarrow v_i$
5: **if** $i = coord$ and $\exists v$ received $n-t$ times **then** *send* DEC-FAST(v) to all

▷ **Thread principal**

6: **loop**
7: $coord \leftarrow (r_i \mod n)$; $r_i \leftarrow r_i + 1$ ▷ Ronde r_i

// Étape 1 : Chaque processus donne sa valeur au coordinateur puis attend la réponse.

8: *send* QUERY(r_i, est_i) to p_{coord} ; *set_timer*($\Delta_i[C]$)
9: **Wait until**(received COORD(r_i, est) received or time_out)
10: **if** timer times out **then** $\Delta_i[C] \leftarrow \Delta_i[C] + 1$; $aux_i \leftarrow \perp$
11: **else** disable_timer ; $aux_i \leftarrow est$ **endif**

// Étape 2 : Chaque processus fait suivre si possible la valeur provenant du coordinateur (décision rapide si possible).

12: *send* RELAY(r_i, aux_i) to all
13: **Wait until** RELAY($r_i, *$) received from $n-t$ proc. **store values** in V_i
14: **if** $\{x \in V_i \ / \ \#x > n-3f\} = \{v\}$ **then** $aux_i \leftarrow v$ **else** $aux_i \leftarrow \perp$ **endif**
15: **if** $\exists v$ received $n-t$ times **then** *send* DEC-FAST2(v) to all

// Étape 3 : Les processus vérifient alors que les valeurs sont cohérentes.

16: *send* FILT1(r_i, aux_i) to all
17: **Wait until**FILT1($r_i, *$) received from $n-t$ proc. **store val.** in V_i
18: **if** $V_i = \{v\}$ **then** $aux_i \leftarrow v$ **else** $aux_i \leftarrow \perp$ **endif**

// Étape 4 : Si tout c'est bien passé, les processus décident.

19: *send* FILT2(r_i, aux_i) to all
20: **Wait until**FILT2($r_i, *$) received from $n-t$ proc. **store val.** in V_i
21: **if** $V_i = \{v\}$ **then** *send* DEC(v) to all ; **return** v
22: **if** $V_i = \{v, \perp\}$ **then** $est_i \leftarrow v$
23: **end loop**

▷ **Thread de coordination**

24: **upon** receipt of QUERY(r, est) for the first time : send COORD(r, est) to all

▷ **Threads de décision**

25: **upon** receipt of DEC(est) *send* DEC(est) to all ; **return** *est*
26: **upon** receipt of DEC-FAST(est) *send* DEC-FAST(est) to all ; **return** *est*
27: **upon** receipt of DEC-FAST2(est) *send* DEC-FAST(est) to all ; **return** *est*

Protocole 6 Première modification pour n>5t

▷ **Thread principal**

5: **if** $i = coord$ and $\exists v$ received $n - t$ times **then** *send* DEC-FAST(v) to all

▷ **Threads de décision**

26: **upon** receipt of DEC-FAST(est) *send* DEC-FAST(est) to all ; **return** est

Démonstration : Pour rappel la seconde modification consiste en l'ajout de deux lignes :

Protocole 7 Seconde modification pour n>5t

▷ **Thread principal**

15: **if** $\exists v$ received $n - t$ times **then** *send* DEC-FAST2(v) to all

▷ **Threads de décision**

27: **upon** receipt of DEC-FAST2(est) *send* DEC-FAST2(est) to all ; **return** est

Les démonstrations de la validité et de la terminaison sont exactement les mêmes que dans le cas $n > 3t$. Quand à la propriété de validité, le raisonnement est sensiblement le même que pour la première modification pour $n > 5t$ et ne pose donc aucune difficulté.

□

4 Hypothèses minimales de synchronie pour le consensus byzantin avec signatures

Nous allons dans cette partie exhiber les hypothèses minimales de synchronie (Combien de liens synchrones sont nécessaires ? Comment doivent-ils être répartis ?) nécessaires et suffisantes pour résoudre le problème du consensus.

S'il a déjà été montré qu'une $[t+1]$bi-source était suffisante pour résoudre le problème du consensus, est-il possible de résoudre ce problème avec seulement des $[t]$bi-source ?

Nous supposons bien évidemment dans cette partie que $n > 3t$.

4.1 Définitions

Définition 4.1 *On appelle propriété de synchronie toute propriété portant sur les liens synchrones, qui peut s'exprimer sous forme de propriété sur un graphe non orienté dans lequel les nœuds correspondent aux processus et les arêtes aux liens synchrones entre processus corrects.*

Par exemple « Il y a 3 liens synchrones » peut s'exprimer par « la graphe contient trois arrêtes » ou « il y a 5 [2]bi-source » peut se traduire par le graphe contient 5 nœuds qui possèdent deux arêtes.

Par contre une propriété de la forme « à chaque instant il y a un lien synchrone dans le système, ce lien pouvant changer pendant l'exécution » ne peut pas s'exprimer sous forme de propriété sur un graphe. De même la propriété « il y a un [3]IO-processus » ne peut pas s'exprimer dans un graphe non orienté.

Définition 4.2 *Soit un graphe $\mathcal{G}$ et $\mathcal{S}$ une propriété de synchronie. On dit que $\mathcal{G}$ est (t, S)-ambigu si pour toute composante connexe C de $\mathcal{G}$ on a :*

– $|C| \leq t$;
– $\mathcal{G} \setminus C$ *vérifie* $\mathcal{S}$

Définition 4.3 *Soit un système de n processus et un graphe $\mathcal{G}$ de n sommets (on peut donc identifier chaque processus à un sommet du graphe). On note $E_{\mathcal{G}}$ l'ensemble des exécutions vérifiant les quatre conditions suivantes :*

- *si p et q sont dans la même composante connexe, alors p et q sont synchrones ;*
- *si p et q sont dans deux composantes connexes distinctes, alors p et q sont asynchrones ;*
- *si p et q sont dans la même composante connexe, alors si p est en fautif, q est en fautif ;*
- *si p et q sont dans deux composantes connexes distinctes, alors si p est en fautif, q est correct.*

Définition 4.4 *On note $E_{\mathcal{G}}(t, \mathcal{S})$ l'ensemble des exécutions de $E_{\mathcal{G}}$ qui possèdent moins de t processus fautifs et qui vérifient $\mathcal{S}$.*

Propriété 4.1 *Si $\mathcal{G}$ est un graphe $(t, \mathcal{S})$-ambigu, alors $E_{\mathcal{G}} = E_{\mathcal{G}}(t, \mathcal{S})$.*

Démonstration : Soit $\mathcal{G}$ un graphe (t, S)-ambigu. Considérons alors une exécution E de $E_{\mathcal{G}}$.

Supposons qu'il y ait au moins un processus fautif lors de l'exécution. Par définition, l'ensemble des processus fautifs de E correspond exactement à une composante connexe. Or étant donné que $\mathcal{G}$ est $(t, \mathcal{S})$-ambigu, chaque composante connexe contient moins de t processus. Donc l'exécution E comporte moins de t processus fautifs.

Montrons maintenant que E vérifie la propriété $\mathcal{S}$. On note C_f la composante connexe fautive (si aucun processus n'est fautif, C_f désigne une composante connexe quelconque choisie au hasard). Le sous-graphe $\Pi \setminus C_f$ ne contient que des processus corrects et vérifie $\mathcal{S}$ et donc l'exécution E contient vérifie $\mathcal{S}$. □

Intuitivement ce résultat veut dire que si $\mathcal{G}$ est un graphe $(t, \mathcal{S})$-ambigu, toute exécution qui respecte les liens synchrones définit par le graphe $\mathcal{G}$ et dans laquelle une seule composante connexe peut correspondre à des processus fautifs vérifie $\mathcal{S}$.

4.2 Une généralisation de FLP

Propriété 4.2 *S'il existe un graphe $\mathcal{G}$ de n processus qui est (t, S)-ambigu, alors le problème du consensus avec signatures ne peut pas être résolu avec n processus dont t byzantins en présence avec les hypothèses $\mathcal{S}$.*

Démonstration : Supposons qu'il existe un protocole A qui permette de résoudre le problème du consensus avec n processus dont t byzantins et avec les hypothèse $\mathcal{S}$. Supposons de plus qu'il existe un graphe $\mathcal{G}$ de taille n qui est (t, S)-ambigu.

On note alors ν le nombre de composantes connexes de $\mathcal{G}$. Nous allons alors montrer qu'il est possible de résoudre le problème du consensus dans un environnement purement asynchrone de ν processus et en présence d'un processus fautif.

On note $C_1, \ldots, C_\nu$ les composantes connexes de $\mathcal{G}$, $\Pi_1 = p_1, \ldots, p_\nu$ un ensemble de ν processus asynchrones entres eux et Π_0 un ensemble de n processus. On remarque alors que les processus de Π_1 peuvent simuler l'exécution de l'algorithme A par les processus de Π_0 (voir figure 1).

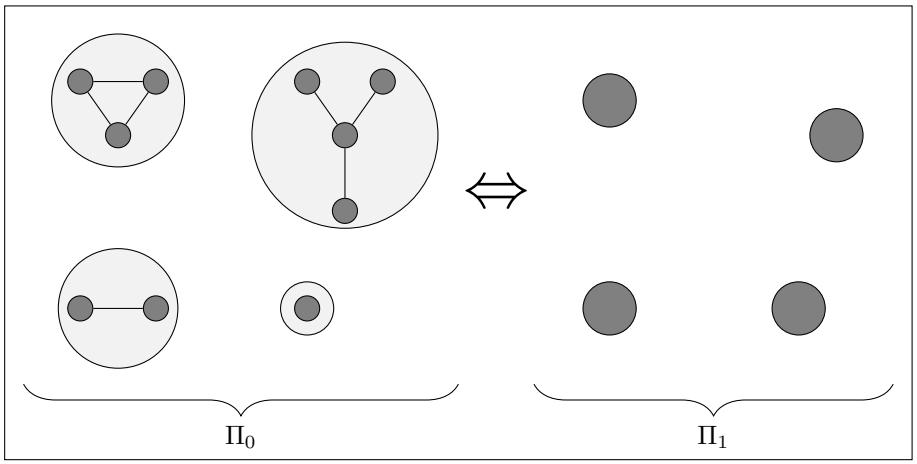

FIGURE 1 – Chaque processus (à droite) simule un cluster de processus (à gauche)

Pour cela on fait simuler à chaque processus $p_i \in \Pi_1$ les processus du cluster C_i. Une telle simulation correspond pour les processus de Π_0 à une exécution de $E_\mathcal{G}$ (cela découle immédiatement de la définition de $E_\mathcal{G}$). Or étant donné que $\mathcal{G}$ est (t, S)-ambigu, la simulation correspond à une exécution de $E_\mathcal{G}(t, S)$ et donc le protocole A termine.

Ainsi puisque l'on est capable de simuler une exécution d'un protocole correct de consensus dans Π_1, on peut donc résoudre le problème du consensus en présence de ν processus asynchrones et d'un processus fautif. Absurde car cela est en contradiction avec le résultat de FLP. □

Propriété 4.3 *S'il n'existe pas de graphe $\mathcal{G}$ de n processus qui est (t, S)-ambigu, alors le problème du consensus avec signatures possède une solution avec n processus dont t byzantins en présence des hypothèses $\mathcal{S}$.*

Démonstration : Pour cela nous allons montrer que l'on peut implémenter l'oracle $\diamond\mathcal{W}(Byz)$ dans un système de t processus fautifs avec les hypothèses $\mathcal{S}$. Pour rappel un tel oracle doit vérifier les deux propriétés suivantes :

1. À partir d'un certain temps, chaque processus qui commet un faute détectable est suspecté par au moins $t + 1$ processus corrects ;
2. À partir d'un certain temps, il existe un processus correct qui n'est plus suspecté par aucun processus correct.

On suppose pour cela que chaque processus possède les variables suivantes :
- SUPECTED$_i$: l'ensemble des processus suspectés par p_i ;
- BYZANTIN$_i$: l'ensemble des processus que p_i sait être byzantin ;
- LEADER$_i$: un processus jugé correct par p_i ;

– ASYNCH$_i$: l'ensemble des liens que p_i sait être asynchrones (un lien synchrone entre un processus correct et un processus byzantin est considéré asynchrone si le processus byzantin fait croire au processus correct que le lien est asynchrone).

Les variables SUPECTED$_i$, BYZANTIN$_i$, ASYNCH$_i$ sont initialisées à $\emptyset$, la variable LEADER$_i$ est initialisée à p_0. Au commencement, chaque processus considère donc que tous les processus sont corrects, choisit p_0 comme leader et considère que le système ne contient que des liens synchrones.

Il suffit alors pour les processus de partager l'information à leur disposition. La présence de signatures permet d'empêcher les processus byzantins de donner des informations non prouvées.

– À chaque fois qu'un processus reçoit un message, il le fait suivre à tous.
– À chaque fois qu'un processus p remarque qu'un lien $p \to q$ est asynchrone, il envoie un message ASYNCHRONE(p, q) à tous les processus.
– À chaque fois qu'un processus p remarque qu'un processus q a commis une faute de commission (par exemple si un processus a envoyé deux messages distincts lors d'une même étape, ou si un processus envoie un message incompatible avec les messages qu'il a précédemment reçus), il envoie BYZANTIN(q) et la preuve (c'est-à-dire les messages signés par q qui prouvent la faute de commission) à tous les processus.

Il suffit alors pour les processus de mettre à jour l'ensemble BYZANTIN$_i$ à chaque fois qu'ils reçoivent un message de la forme BYZANTIN$(*)$ et l'ensemble ASYNCH à chaque fois qu'ils reçoivent un message de la forme ASYNCHRONE$(*)$.

On remarque qu'à partir d'un certain moment, que nous noterons τ, les ensembles BYZANTIN$_i$ et ASYNCH$_i$ ne changeront plus et seront communs à tous les processus corrects. Or l'ensemble ASYNCH$_i$ permet de partitionner l'ensemble Π des processus en composantes connexes. Cette décomposition sera donc la même pour tous les processus corrects à partir de τ. Le graphe ainsi défini ne peut pas être $(t, \mathcal{S})$-ambigu par hypothèse. Donc il existe une composante connexe C tel que l'on ait soit $|C| \geq t + 1$, soit $\Pi \setminus C$ ne vérifie pas les hypothèses $\mathcal{S}$. Dans les deux il existe forcément un processus correct dans ce cluster que nous noterons $p_{correct}$.

Notons p_C le processus du cluster C avec le plus petit indice, il suffit alors pour chaque processus d'exécuter LEADER$_i \leftarrow p_C$. Il est aisé de voir qu'ainsi, à partir du temps τ, tous les processus auront élu le même leader. Il suffit alors de montrer que ce leader est correct.

Supposons que cela ne soit pas le cas, alors cela veut dire que ce processus commet soit une faute d'omission, soit une faute de commission (si ce n'est pas le cas, cela signifie qu'il se comporte comme un processus correct). Si p_C commet une faute d'omission, alors il n'envoie pas un message qu'il est censé envoyer à tous les processus. Mais s'il n'envoie pas ce message, cela veut dire que $p_{correct}$ découvrira que le lien $p_{correct} \to p_C$ est asynchrone, et donc modifiera la variable ASYNCH$_i$. Absurde car par définition de τ, les variables ne sont plus modifiées. Si par contre p_C commet une faute de commission, alors tous les processus s'en rendront compte (car à chaque fois qu'un processus correct reçoit un message,

il le transmet à tous les autres processus corrects), et les variables BYZANTIN$_i$ seront modifiées. Absurde par définition de τ.

Il reste encore à faire en sorte que chaque processus qui commet une faute détectable soit soupçonné par tous les processus corrects (et donc en particulier $t+1$ processus corrects). Les fautes de commissions étant détectée de manière efficace, tous les processus qui commettent de telles fautes se retrouve dans les ensembles BYZANTIN$_i$ de tous les processus correct p_i.

On considère un processus correct p_i et on note f le cardinal de l'ensemble BYZANTIN$_i$.

Il suffit alors pour p_i de mettre dans l'ensemble SUPECTED$_i$ les f processus de l'ensemble BYZANTIN$_i$ et les $t-f$ processus qui ont potentiellement commis des fautes d'omissions.

Pour cela p_i considère les dernières nouvelles qu'il a reçues de la part de chaque processus. Il suffit alors de compléter l'ensemble SUPECTED$_i$ par les $t-f$ processus qui ont donné des nouvelles le moins récemment.

Ainsi si des processus crashent, ils seront forcément au bout d'un certain temps dans cet ensemble SUPECTED$_i$.

Nous venons donc de montrer que nous pouvons implémenter l'oracle $\diamond\mathcal{W}(byz)$ s'il n'existe pas de graphe $(t,\mathcal{S})$-ambigu, et donc que l'on peut résoudre le problème du consensus. □

Théoreme 4.1 *Le problème du consensus avec signatures possède une solution avec n processus dont t byzantins et avec les hypothèses $\mathcal{S}$ si et seulement s'il n'existe pas de graphe (t,S)-ambigu.*

Démonstration : Ce résultat est une conséquence directe des deux propriétés précédentes (4.2 et 4.3). □

Ce résultat est une généralisation de FLP. Plus exactement la propriété S dans FLP est la propriété *vrai* (en effet le fait de considérer un graphe asynchrone revient au même que de ne considérer aucune hypothèse de synchronie). Le théorème 4.1 peut alors s'exprimer sous la forme :

« Le problème du consensus avec signatures possède une solution avec n processus dont 1 *byzantin et aucun lien synchrone si et seulement s'il n'existe pas de graphe* $(1, vrai)$*-ambigu. »*

Or, étant donné qu'il existe un toujours un graphe $(1, vrai)$-ambigu (le graphe dans lequel tous les liens sont asynchrones), alors FLP ne possède pas de solution (le consensus avec crashs étant un cas particulier de consensus byzantin avec signatures).

4.3 Cas particulier des systèmes avec $b[x]$bi-sources

Il a été démontré qu'une $[t+1]$bi-source suffisait à résoudre le problème du consensus. Et en effet cela correspond à un cas particulier du théorème 4.1 démontré dans la partie précédente. La question que l'on peut alors légitimement

se poser est de savoir si le problème du consensus byzantin avec signatures peut être résolu en l'absence de $[t+1]$bi-sources. Par exemple est-ce qu'une $[t]$bi-source suffit ? Est-ce que si tous les processus corrects sont des $[t]$bi-sources, le problème possède une solution ?

Nous étudierons dans cette sous-partie les propriétés de synchronie de la forme « le système possède un nombre b de $[x]$bi-sources », et nous montrerons qu'en considérons uniquement cette classe de propriétés, la présence d'une $[t+1]$bi-source est nécessaire et suffisante pour résoudre le problème du consensus. Nous montrerons en particulier le théorème suivant :

Théoreme 4.2
On pose $n = kt + r$ avec $0 \leq r < t$. On sait résoudre le problème du consensus
- *si $x \geq t+1$;*
- *si $x < t$, $b > (k-1)t$ et $n - t < kx$.*

Par contre le problème du consensus ne possède pas de solution
- *si $x < t$, $b > (k-1)t$ et $n - t \geq kx$;*
- *si $x < t$ et $b \leq (k-1)t$.*

Ce résultat peut être résumé dans un tableau (voir figure 2).

$x \backslash b$	$1 \ldots (k-1)t$	$(k-1)t+1 \ldots n-t$
$n-t$ $\vdots$ $t+1$	✓	
t $\vdots$ 1	×	si $n - t < kx$: ✓ si $n - t \geq kx$: ×

FIGURE 2 – Consensus byzantin avec signatures en présence de $b[x]$bi-sources

Avant de détailler la démonstration de ce résultat, il est intéressant de faire deux remarques. Tout d'abord, une $[t]$bi-source ne suffit pas à résoudre le problème du consensus. On montrera par la suite que le seul cas où l'on peut résoudre le problème du consensus avec uniquement des $[x]$bi-sources avec $x \leq t$, c'est lorsque quelque soit la façon dont on arrange les $b[x]$bi-sources, on crée obligatoirement une $[t+1]$bi-source.

On notera dorénavant $\mathcal{S}(b, x)$ l'hypothèse *« il y a un nombre b de $[x]$bi-sources dans le système »*.

4.3.1 Existence d'une solution pour $x > t$

L'impossibilité d'avoir un graphe $(t, \mathcal{S}(1, t+1))$-ambigu est assez triviale. En effet la présence d'une $[t+1]$bi-source implique l'existence d'une composante

connexe de taille au moins $t+1$, ce qui est en contradiction avec la première hypothèse nécessaire pour avoir un graphe $(t, \mathcal{S}(1, t+1))$-ambigu.

Étant donné qu'il est possible de résoudre le problème du consensus byzantin avec signatures avec une $[t+1]$bi-source, on peut trivialement le résoudre avec $b[x]$bi-sources avec $b \geq 1$ et $x \geq t+1$.

4.3.2 Impossibilité du consensus avec $b \leq (k-1)t$ et $x \leq t$

Dans cette partie, nous allons montrer l'impossibilité de résoudre le problème du consensus avec $(k-1)t$ $[t]$bi-sources. Cette impossibilité implique trivialement celle pour des valeurs de b ou x plus petites.

On rappelle que $n = kt+r$ avec $0 \leq r < t$. Considérons k ensembles de taille t et r singletons (cf. figure 3).

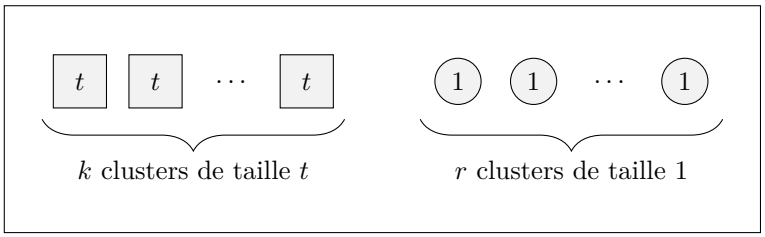

FIGURE 3 – $(k-1)t$ $[t]$bi-sources

Ce découpage correspond trivialement à un graphe $(t, \mathcal{S}((k-1)t, t))$-ambigu. Le théorème 4.1 permet alors d'affirmer que dans cette configuration le consensus ne peut pas être résolu. Ce qui montre notre résultat.

4.3.3 Impossibilité du consensus si $n - t \geq kx$ et $b > (k-1)t$

On suppose dans cette partie que $n - t \geq kx$ et $b > (k-1)t$. Nous allons montrer qu'il est possible de trouver un graphe $\mathcal{G}$ qui soit $(t, \mathcal{S}(n-t, x))$-ambigu. Un tel graphe montre, en vertu du théorème 4.1, que le consensus ne peut être résolu en présence de $(n-t)$ $[x]$bi-sources et donc en présence d'un nombre b de $[x]$bi-sources ($\forall b \leq n-t$). On pose alors $b = n-t$.

Nous aimerions bien sûr faire un partitionnement semblable à celui présenté dans la partie précédente, mais le système ne contient pas suffisamment de processus pour faire $k+1$ clusters de taille t (et k cluster de taille t ne suffisent pas à avoir b $[x]$bi-sources). Nous allons donc faire $k+1$ clusters de taille x (voir figure 4). Un tel partitionnement est cette fois possible, car :

$$(k+1)x = kx + x \leq (n-t) + t = n$$

Nous avons donc obtenu $(k \cdot x)$ $[x]$bi-sources (car on suppose qu'un des clusters peut ne contenir que des processus fautifs). Cependant il est possible que b soit supérieur à kx et donc nous allons rajouter dans les $k+1$ clusters suffisamment de processus pour atteindre un nombre suffisant de $[x]$bi-sources.

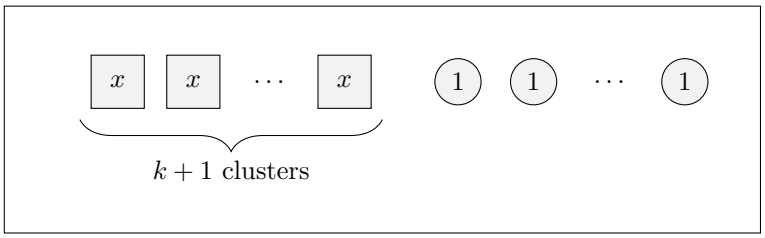

FIGURE 4 – Partitionnement pour obtenir des $[x]$bi-sources

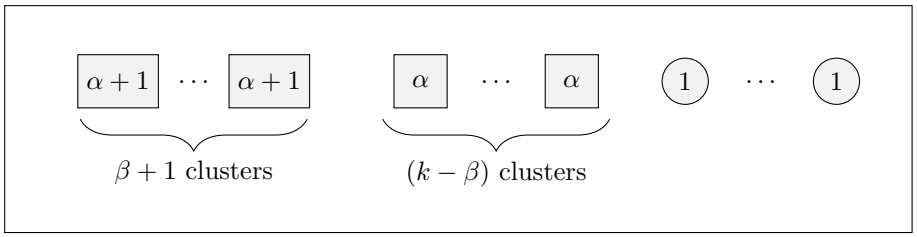

FIGURE 5 – Partitionnement pour $b \geq kx$ et $b > (k-1)t$

On pose $b = \alpha k + \beta$ avec $0 \leq \beta < k$ (α et β sont respectivement le quotient et le reste de la division euclidienne de b par k). On partitionne alors le système de la manière suivante (voir figure 5) :

- $(\beta + 1)$ clusters de taille $(\alpha + 1)$;
- $(k - \beta)$ clusters de taille α ;
- $(n - (\alpha + 1) - b)$ clusters de taille 1.

Montrons alors que ce graphe est bien ambigu. Tout d'abord, il est aisé de remarquer que $\alpha + 1 \leq t$. En effet, sachant que $n < (k+1)t$ (par définition de k), on a :

$$\alpha t \leq b = n - t < kt$$

Soit C un cluster quelconque, si C est byzantin, le système contient quand même b $[x]$bi-sources.

- si C est un des clusters de taille $\alpha + 1$ et est composé exclusivement de processus byzantins alors que le reste du système est correct, il reste alors dans le système $k - \beta$ clusters de taille α et β clusters de taille $\alpha + 1$, ce qui nous donne $\alpha k + \beta = b$ clusters de taille supérieure ou égale à x et donc en particulier b $[x]$bi-sources.
- si C est un des clusters de taille α et est composé exclusivement de processus byzantins alors que le reste du système est correct, il reste alors dans le système $k - \beta - 1$ clusters de taille α et $\beta + 1$ clusters de taille $\alpha + 1$, ce qui nous donne $b + 1 = \alpha k + \beta + 1$ clusters de taille supérieure ou égale à x et donc en particulier $b + 1$ $[x]$bi-sources.
- Enfin si C est un cluster qui ne contient qu'un processus et que ce dernier est byzantin, et si tous les autres processus sont corrects, alors la présence

de b $[x]$bi-sources est trivial.

On déduit alors que $\mathcal{G}$ est $(t, \mathcal{S}(b, x))$-ambigu. Ainsi le théorème 4.1 s'applique et le consensus ne peut être pas résolu pour $b = n - t \leq kx$ et $b > (k-1)t$ et donc plus généralement pour tout b avec $n - t \leq kx$ et $b > (k-1)t$.

4.3.4 Synchronie et relation d'équivalence

Dans cette partie nous allons montrer que l'on peut transformer un système quelconque en un système dans lequel la relation entre processus corrects « être synchrone » est une relation d'équivalence. On rappelle que l'on considère que pour tout couple de processus (p, q) si le lien $p \to q$ est synchrone, alors le lien $q \to p$ l'est aussi.

Définition 4.5 *On dit qu'un système est δ-synchrone si $\forall p, q$, si le lien entre p et q est synchrone, alors tout message envoyé par p à la date t sera reçu par q à la date $t + \delta$*

À chaque fois qu'un processus reçoit un message pour la première fois, il le fait suivre à tout le monde. On démontre aisément que le fait de faire suivre les messages permet de transformer un système δ-synchrone en un système Δ-synchrone avec $\Delta = (n-t)\delta$ dans lequel la relation être synchrone est transitive entre processus correct.

4.3.5 Possibilité du consensus si $n - t < kx$ et $b > (k-1)t$

On suppose que la relation être synchrone est transitive entre processus corrects (si ce n'est pas la cas il suffit d'appliquer la transformation décrite dans la partie 4.3.4). Ainsi si on considère l'ensemble Π_c des processus corrects, alors on peut partitionner Π_c avec la relation d'équivalence « être synchrone ». On appelle clusters les sous-ensembles correspondant à la partition, et on note C_p le cluster qui contient p. Ainsi si p est une $[x]$bi-source mais pas une $[x+1]$bi-source, p appartient forcément à un cluster C_p de taille x qui est asynchrone avec le reste du système.

On considère alors l'ensemble des clusters $\mathcal{C}_b$ contenant les $[x]$bi-sources, c'est-à-dire l'ensemble des clusters de taille supérieur à x. On constate tout d'abord que quelque soit la valeur de b, nous n'avons pas suffisamment de bi-sources pour les répartir dans k clusters de taille supérieure à x (car le nombre total de processus corrects, $n - t$, est strictement inférieur à kx) et donc en notant $|\mathcal{C}_b|$ le cardinal de $\mathcal{C}_b$, on a $|\mathcal{C}_b| \leq (k-1)$. Enfin comme il y a strictement plus de $(k-1)t$ bi-sources et qu'il y a moins de $(k-1)$ clusters distincts, un des clusters est constitué de strictement plus de t processus, et donc possède une $[t+1]$bi-source.

Étant donné que l'on sait résoudre le consensus byzantin en présence de $[t+1]$bi-sources, on sait alors le résoudre pour $n - t < kx$ et $b > (k-1)t$.

5 Hypothèses minimales de synchronie pour le consensus byzantin sans signatures

Nous allons dans cette partie commencer le même travail que celui fait dans la partie précédente. C'est-à-dire que nous définirons une certaine classe de

graphes, les graphes $(t, \mathcal{S})$bi-ambigus, qui permettent de montrer que le consensus ne possède pas de solution avec les hypothèses S et un nombre t de processus fautifs. Contrairement à la partie précédente, nous ne montrerons pas que l'absence de tels graphes est suffisante pour résoudre le consensus.

5.1 Définitions

Les définitions de cette partie sont relativement semblables à celles de la partie précédente. En particulier la définition de propriété de synchronie est exactement la même et donc ne sera pas répétée.

Définition 5.1 *Soit un graphe $\mathcal{G}$ et $\mathcal{S}$ une propriété de synchronie. On dit que $\mathcal{G}$ est (t, S)bi-ambigu si toute composante connexe C de $\mathcal{G}$, peut être partitionnée en deux clusters C_1 et C_2 tels que :*
- *$|C_1| \leq t$ et $|C_2| \leq t$*
- *$\mathcal{G} \setminus C_1$ vérifie $\mathcal{S}$*
- *$\mathcal{G} \setminus C_2$ vérifie $\mathcal{S}$*

Si une composante connexe est réduite à un seul processus p, on pose $C_1 = \{p\}$ et $C_2 = \emptyset$.

Définition 5.2 *Pour un graphe $\mathcal{G}$ (t, S)bi-ambigu, on définit alors l'ensemble des clusters Γ de $\mathcal{G}$ (en utilisant le découpage de la définition précédente).*

Il est aisé de remarquer que Γ définit une partition de $\mathcal{G}$.

Définition 5.3 *On définit la fonction voisin : $\Gamma \to \Gamma$ qui à un cluster C_1, associe un autre cluster C_2, tel que si C est la composante connexe contenant C_1, alors $C_2 \cup C_2 = C$.*

Définition 5.4 *Soit un système de n processus et un graphe $\mathcal{G}$ de n sommets (on peut donc identifier chaque processus à un sommet du graphe). On note $E_{\mathcal{G},\Gamma}$ l'ensemble des exécutions vérifiant les quatre conditions suivantes :*
- *si p et q sont dans la même composante connexe, alors p et q sont synchrones ;*
- *si p et q sont dans deux composantes connexes distinctes, alors p et q sont asynchrones ;*
- *si p et q sont dans le même cluster, alors si p est fautif, q est en fautif ;*
- *si p et q sont dans deux clusters distincts, alors si p est fautif, q est correct.*

Il est intéressant de remarquer que contrairement à la définition correspondante dans le cas avec signatures, les composantes connexes définissent les composantes synchrones et les clusters définissent les composantes fautives.

Définition 5.5 *On note $E_{\mathcal{G},\Gamma}(t, \mathcal{S})$ l'ensemble des exécutions de $E_{\mathcal{G},\Gamma}$ qui possèdent moins de t processus fautifs et qui vérifient $\mathcal{S}$.*

Propriété 5.1 *Si $\mathcal{G}$ est un graphe $(t, \mathcal{S})$bi-ambigu, alors $E_{\mathcal{G},\Gamma} = E_{\mathcal{G},\Gamma}(t, \mathcal{S})$.*

Démonstration : Soit $\mathcal{G}$ un graphe (t, S)bi-ambigu. Considérons alors une exécution E de $E_{\mathcal{G},\Gamma}$.

Supposons qu'il y ait au moins un processus fautif lors de l'exécution. Par définition, l'ensemble des processus fautifs de E correspond exactement à un cluster. Or étant donné que $\mathcal{G}$ est $(t, \mathcal{S})$bi-ambigu, chaque cluster contient moins de t processus. Donc l'exécution E comporte moins de t processus fautifs.

Montrons maintenant que E vérifie la propriété $\mathcal{S}$. On note C_f le cluster fautif (si aucun processus n'est fautif, C_f désigne un cluster quelconque choisi au hasard). Le sous-graphe $\Pi \setminus C_f$ ne contient que des processus corrects et vérifie $\mathcal{S}$ et donc l'exécution E vérifie $\mathcal{S}$. □

Intuitivement ce résultat veut dire que si $\mathcal{G}$ est un graphe $(t, \mathcal{S})$bi-ambigu, toute exécution qui respecte les liens synchrones définis par $\mathcal{G}$ et dans laquelle une seule composante connexe peut être fautive, vérifie $\mathcal{S}$.

5.2 Une variante de FLP

Nous allons dans cette partie montrer que l'existence d'un graphe $(t, \mathcal{S})$bi-ambigu permet de montrer que le consensus ne possède pas de solution avec les hypothèses S et un nombre t de processus fautifs.

Le résultat étant relativement proche de celui de la partie précédente, la démonstration sera « relativement » semblable (bien que cela ne soit pas visible *a priori*). Plus précisément, la première démonstration consistait à se ramener à FLP, mais en ce qui concerne la seconde démonstration décrite ci-après, le résultat ne correspondant pas tout à fait, la preuve consiste en une modification légère de FLP.

Pour plus de détail sur la démonstration le lecteur est invité à lire initialement la démonstration de Fisher, Linch et Paterson [6] qui est sensiblement la même mais en légèrement plus simple (en particulier seule la démonstration du lemme 5.2 est changée).

5.2.1 Définitions

Avant de commencer la démonstration, nous commencerons par quelques définitions. Ces définitions peuvent être trouvées de manière plus détaillée dans l'article original de FLP [6].

Définition 5.6 *On appelle évènement la succession des trois étapes suivantes :*
- *la réception d'un message par un processus p (potentiellement le message vide $\perp$) ;*
- *un calcul local de ce même processus p ;*
- *accessoirement, un envoi de message de la part de p à tous les processus.*

Définition 5.7 *On appelle exécution d'un protocole une suite d'évènements $e_1, e_2, \ldots$ qui respecte le protocole.*

Définition 5.8 *On appelle état (du système) la donnée des états de chaque processus ainsi que l'ensemble des messages en transit (envoyés et non encore reçus).*

Définition 5.9 *On appelle état initial un état possible du système avant le début de l'exécution du protocole.*

Définition 5.10 *On dit qu'un état est 0-valent si toute exécution partant de cette état décide 0. On définit de même un état 1-valant.*

On dit qu'un état est bi-valent s'il existe une exécution qui partant de cette état décide 0 et une autre qui décide 1.

On remarque que dans le cas du consensus binaire, un état peut être soit uni-valent (c'est-à-dire 0-valent ou 1-valent), soit bi-valent.

5.2.2 Démonstration

Le principe de la démonstration est relativement simple et repose sur cette notion de valence. On suppose qu'il existe un protocole A qui résout le consensus binaire (qui est un cas particulier de consensus). On montre tout d'abord qu'il existe un état bi-valent, puis par récurrence on construit une exécution infinie ne comportant que des états bi-valents et cette exécution ne décide donc jamais. Le protocole A ne vérifiant pas alors la propriété de terminaison, cela nous permet de conclure qu'un tel protocole ne peut pas exister.

Lemme 5.1 *Il existe un état initial bi-valent.*

Démonstration : Supposons que cela ne soit pas le cas. On considère le tableau $T = [v_0, \ldots, v_n]$ des valeurs proposées. La valence de l'état initial dépendant uniquement de T, on associe donc à chaque état initial le tableau correspondant. On sait de plus qu'il existe une configuration initiale 0-valente (si tous les processus corrects proposent 0) et une configuration 1-valente (idem). On dit que deux configurations initiales sont adjacentes si les tableaux correspondants ne diffèrent que d'une valeur.

Il est aisé de constater que deux états initiaux peuvent être reliés par une chaîne d'états initiaux adjacent deux à deux. Ainsi en considérant une chaîne qui va de l'état 0-valent à l'état 1-valent, il existe deux états adjacents dont l'un est 0-valent et l'autre est 1-valent. On note alors p le processus à l'origine de la valeur qui diffère entre les deux processus.

Si p crash avant même le début de l'exécution, les processus ne peuvent faire la différence entre l'exécution partant de l'état initial 0-valent, et celle partant de l'état 1-valent, et donc les processus corrects autre que p décident la même valeur dans les deux cas, ce qui est absurde. □

Lemme 5.2 *Soit $\mathcal{G}$ un graphe $(t, \mathcal{S})$bi-ambigu. Si C est un état bi-valent et o un évènement qui peut s'appliquer sur C, alors il existe une exécution σ (compatible avec $\mathcal{S}$) tel que $o(\sigma(C))$ soit bivalent.*

Démonstration : Tout d'abord notons p_o le processus à l'origine de o et $\mathscr{C}$ l'ensemble des états atteignables depuis C par des exécutions dans lesquelles l'évènement o n'intervient pas (on considère que $C \in \mathscr{C}$). On définit alors l'ensemble $\mathscr{D} = o(\mathscr{C})$.

Montrons alors par l'absurde que $\mathscr{D}$ contient un état bi-valent. Supposons que $\mathscr{D}$ ne contienne que des états uni-valents.

Commençons par démontrer que $\mathscr{D}$ contient un état 0-valent et un état 1-valent. Soit $i \in \{0,1\}$, il existe une exécution σ_i, tel que $\sigma_i(C)$ soit i-valent (car C est bi-valent). Si $\sigma_i(C) \in \mathscr{C}$, dans ce cas $o(\sigma_i(C))$ est i-valent et appartient à $\mathscr{D}$. Sinon o est exécuté pendant σ_i, on pose alors $\sigma_i(C) = \sigma'(o(\sigma''(C)))$, ce qui nous permet de conclure que $o(\sigma''(C))$ est i-valent. Ainsi $\mathscr{D}$ contient un état 0-valent et un état 1-valent.

On suppose sans perdre en généralité que $o(C)$ est 0-valent. On peut alors montrer qu'il existe un état C_b et un évènement o' tel que $o(C_b)$ soit 0-valent et $o(o'(C_b))$ soit 1-valent. Le processus qui est à l'origine de o est forcément le même que celui qui est l'origine de o', en effet, si ce n'était pas le cas on aurait $o'(o(C_b)) = o(o'(C_b))$. Ce qui est absurde. Jusqu'alors, cette démonstration était exactement la même que celle de FLP. La partie qui suit par contre diffère de la preuve originale.

On note alors γ_0 le cluster contenant p_o et $\gamma_1 = voisin(\gamma_0)$. On suppose que $\gamma_1 \neq \emptyset$. Considérons maintenant trois situations :

1. Les processus de γ_0 et de γ_1 sont corrects, mais pour des raisons de synchronies, les processus de $\Pi \setminus (\gamma_0 \cup \gamma_1)$ ne reçoivent plus de messages de la part de γ_0. De plus p_o exécute o (ce qui mène dans un état 0-valent).
2. Les processus de γ_0 sont byzantins et n'envoient plus de messages aux processus de $\Pi \setminus (\gamma_0 \cup \gamma_1)$ et exécutent l'évènement o, mais les processus de γ_1 sont corrects.
3. Les processus de γ_0 sont corrects et p_o exécute o' puis o, mais les processus de γ_1 sont byzantins et se comportent comme dans les deux situations précédentes (ils font croire que p_o a exécuté o).

On remarque que les trois exécutions décrites ci-dessus sont indistinguables pour les processus de $\Pi \setminus (\gamma_0 \cup \gamma_1)$, or l'une d'entres elle mène dans un état 1-valent (la numéro 3) et l'autre (la première) mène dans un état 0-valent. Absurde. Si $\gamma_1 = \emptyset$, la démonstration est la même que celle de FLP (p_o est un processus asynchrone avec le reste du système). □

Propriété 5.2 *s'il existe un graphe $\mathcal{G}$ de n processus qui est $(t, \mathcal{S})$bi-ambigu, alors le problème du consensus sans signatures ne peut pas être résolu avec n processus dont t byzantins en présence des hypothèses $\mathcal{S}$.*

Démonstration : Ce résultat est une conséquence directe de la propriété précédente. La démonstration pourra être trouvée dans le papier [6]. □

Conjecture 5.1 *Le problème du consensus sans signatures ne peut pas être résolu avec n processus dont t byzantins avec les hypothèses de synchronie $\mathcal{S}$ si et seulement s'il existe un graphe $\mathcal{G}$ de n processus qui est $(t, \mathcal{S})$bi-ambigu.*

Cette conjecture se justifie par le résultat précédent. En fait nous venons de montrer que s'il existe un graphe bi-ambigu, alors le consensus ne possèderait pas de solution. La difficulté pour montrer la réciproque est que dans le cas précédant, le fait d'utiliser des signatures obligeait les processus byzantin à se comporter correctement. En effet, si l'exécution d'un processus byzantin déviait du protocole original, il suffisait qu'un processus s'en rende compte pour qu'il

puisse alerter les autres processus, preuves à l'appui (les messages signés). Une telle approche n'est par contre plus possible dans un modèle sans signatures et il faut que chaque erreur détectable soit détectée systématiquement par $t + 1$ processus.

Si ce résultat se révélait être vrai, une conséquence intéressante serait que l'on pourrait résoudre le problème du consensus byzantin sans signatures avec uniquement des $[2t]$bi-sources (au lieu d'une $2t + 1$ bi-source). Autrement dit la présence d'une $[2t + 1]$bi-source ne serait pas une condition nécessaire et suffisante. En particulier un système avec un cluster de $2t$ processus correct formant des liens synchrones entre eux (donc $2t[2t]$bi-sources) suffirait à résoudre le consensus pour $n = 3t + 1$.

5.3 Cas particulier des systèmes avec $b[x]$bi-sources

Tous comme dans la partie précédente, nous allons étudier plus particulièrement la signification de ce résultat avec la présence de $b[x]$bi-sources.

En particulier le tableau ci-dessous résume les résultats, en supposant la conjecture précédente et en posant $n = k \cdot (2t) + r$. Nous nous contenterons pas la suite de démontrer seulement les cas où le consensus est impossible, car les démonstrations, pour l'impossibilité de tracer des graphes bi-ambigus, manquent toujours.

$x \setminus b$	$1 \ \ldots \ (k-1) \cdot 2t$	$(k-1) \cdot 2t + 1 \quad \ldots \quad n - t$
$n - t$ $\vdots$ $2t + 1$	✓	
$2t$ $\vdots$ 1	(si $n < 4x$: ✓) ? si $n \geq 4x$: ×	(si $n - t < k \cdot 2x$: ✓) ? si $n - t \geq k \cdot 2x$ et $n \geq 4x$: ×

FIGURE 6 – Consensus byzantin avec signatures en présence de $b[x]$bi-sources

5.3.1 Impossibilité du consensus pour $n \geq 4x$ et $b \leq (k-1)2t$

Nous allons montrer ce résultat pour $x = 2t$, l'impossibilité pour de plus petites valeurs de x en est une conséquence immédiate.

Le graphe décrit dans la figure 7 ci-dessous montre un exemple de graphe $(t, \mathcal{S}(b, x))$bi-ambigu. En effet les composantes connexes sont soit de taille 1, soit de taille $2t$ (que l'on découpe alors en deux clusters de taille t). Et quelque soit le cluster C, on voit facilement que le sous graphe $\mathcal{G} \setminus C$ contient $(k-1) \cdot 2t$ $[2t]$bi-sources, et donc b $[2t]$bi-sources, car $b \leq (k-1) \cdot 2t$ par hypothèse.

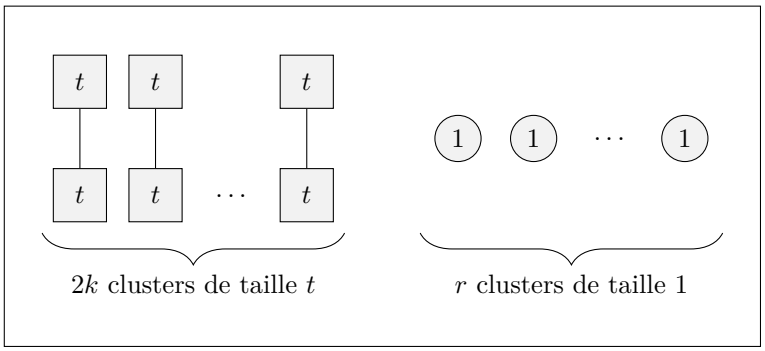

FIGURE 7 – $(k-1)2t$ $[2t]$bi-sources

Intuitivement ce résultat s'explique par le fait que si l'on considère deux clusters de taille t voisins, que nous nommerons C_1 et C_2. Si par exemple C_2 ne donne pas de nouvelle, les processus de $\Pi \setminus (C_1, C_2)$ ne peuvent pas savoir si :

1. les deux clusters C_1 et C_2 sont corrects ;
2. les processus du cluster C_2 sont byzantins et n'envoient des messages qu'au cluster C_1 (ce qui explique pourquoi les autres processus ne reçoivent pas de nouvelle de C_2 ;
3. les processus de C_1 sont byzantins et mentent sur le comportement de C_2.

Ainsi, si un cluster ne donne pas de nouvelles, il est impossible pour le reste du système de savoir si ce cluster est correct ou non. C'est cette ambiguïté permanente qui empêche le consensus, car il existe des exécutions dans lesquelles il est impossible de faire confiance à aucun processus.

5.3.2 Impossibilité du consensus pour $n \geq 4x$, $b > (k-1)2t$ et $n - t \geq k \cdot 2x$

Le raisonnement est sensiblement le même que dans le cas avec signatures.

On suppose dans cette partie que $n - t \geq kx$, ainsi que $b > (k-1) \cdot 2t$ et $n - t \geq k \cdot 2x$. Nous allons montrer qu'il est possible de trouver un graphe $\mathcal{G}$ qui soit $(t, \mathcal{S}(n-t, x))$bi-ambigu. Un tel graphe montre, en vertu du lemme 5.2, que le consensus ne peut être résolu en présence de $(n-t)$ $[x]$bi-sources et donc en présence de b $[x]$bi-sources ($\forall b \leq n - t$).

On pose $b = n-2t$ puis $b = \alpha k+\beta$ avec $0 \leq \beta < k$ (α et β sont respectivement le quotient et le reste de la division euclidienne de b par k). On partitionne alors le système de la manière suivante :

- $(\beta + 1)$ clusters de taille $(\alpha + 1)$;
- $(k - \beta)$ clusters de taille α ;
- $(n - (\alpha + 1) - b)$ clusters de taille 1.

Le raisonnement qui a amené à partitionner le graphe de cette façon est exactement le même que celui décrit dans la partie 4.3.3. Il nous reste alors à découper chaque composantes connexes en deux clusters (à l'exception bien sûr de celles réduites à un processus).

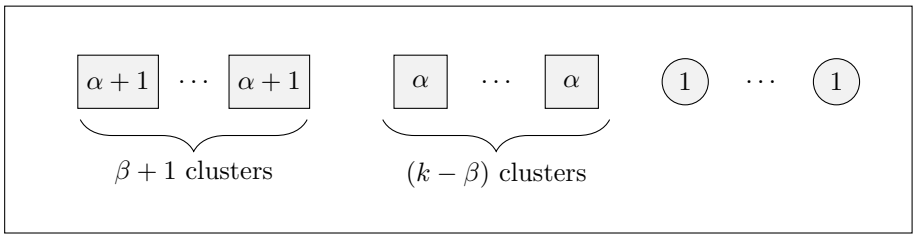

FIGURE 8 – Partitionnement pour $b \geq kx$ et $b > (k-1)t$

On découpe alors chaque partition de taille α (resp. $\alpha+1$) en deux clusters α_1 et α_2 (resp. α_1 et α_2+1) avec $\alpha_1 \leq t$ et $\alpha_2 < t$. Un tel découpage est toujours possible car $\alpha+1 \leq 2t$. En effet, sachant que $n < (k+1)\cdot(2t)$ (par définition de k), on a donc :

$$\alpha k \leq b = n - 2t < k(2t)$$

On obtient alors le graphe représenté dans la figure 9 :

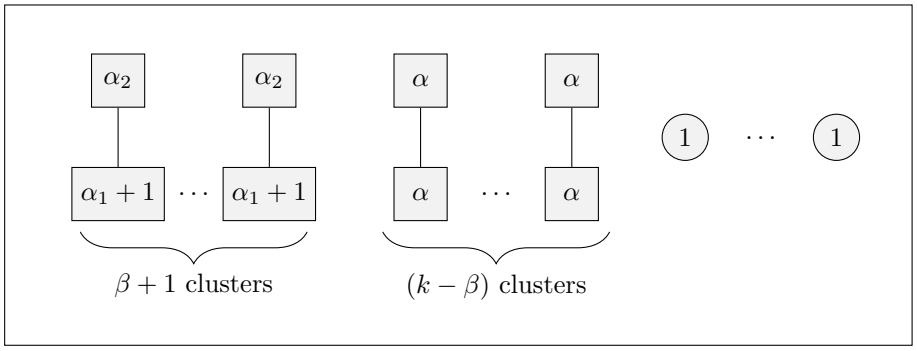

FIGURE 9 – Partitionnement pour $b \geq kx$ et $b > (k-1)t$

Pour montrer que ce graphe est bi-ambigu, il nous suffit de montrer la propriété suivante : « *Soit C un cluster quelconque, si C est byzantin, le système contient quand même b $[x]$bi-sources* ».

- si C est une composante connexe de taille $\alpha+1$ et contient un cluster de processus byzantin, alors que le reste du système $\Pi \setminus C$ est correct, il reste alors dans le système $k-\beta$ composantes connexes de taille α et β composantes connexes de taille $\alpha+1$, ce qui nous donne $b = \alpha k + \beta$ composantes connexes de taille supérieure ou égale à x et donc en particulier b $[x]$bi-sources.
- si C est une des composantes connexes de taille α et contient un cluster de processus byzantins alors que le reste du système est correct, il reste alors dans le système $k-\beta-1$ composantes connexes de taille α et $\beta+1$ clusters de taille $\alpha+1$, ce qui nous donne $b+1 = \alpha k+\beta+1$ composantes connexes de taille supérieure ou égale à x et donc en particulier $b+1$ $[x]$bi-sources.

– Enfin si C est une composante connexe réduite à un processus et que ce dernier est byzantin, et si tous les autres processus sont corrects, alors la présence de b $[x]$bi-sources est trivial.

On déduit alors que $\mathcal{G}$ est $(t, \mathcal{S}(b, x))$bi-ambigu. Ainsi le lemme 5.2 s'applique et le consensus ne peut être pas résolu pour tout b avec $n - t \leq k \cdot 2x$ et $b > (k-1) \cdot 2t$.

Conclusion

Le but de ce stage était avant tout d'améliorer et de compléter le résultat d'Ashour Mostefaoui, de Moumen Hamouma et de Gilles Trédan qui affirmait qu'une $\diamond[t+1]$bi-source était suffisante pour résoudre le problème du consensus. Nous avons pour cela tout d'abord amélioré le protocole de consensus proposé afin qu'il puisse avoir des performances correctes (en particulier qu'il termine rapidement dans le meilleur des cas) et donc qu'il soit utilisable en pratique. Il serait intéressant par la suite d'implémenter ce protocole afin de comparer les performances de ce dernier avec d'autres protocoles de consensus nécessitant des hypothèses plus fortes sur les liens synchrones.

Nous avons ensuite exhibé une condition nécessaire et suffisante sur les liens synchrones pour résoudre le problème du consensus byzantin avec signatures. La notion de graphe ambigu et la nouvelle définition de bi-source (qui ne prend plus en compte les liens avec les processus byzantins) ont permis de répondre à une question qui était posée depuis bientôt trois ans. Cependant, si nous avons montré que, dans le cas des hypothèses de la forme « il y a b $[x]$bi-sources dans le système », la présence d'une $[t+1]$bi-source était nécessaire et suffisante, il serait intéressant soit de démontrer ce résultat dans le cas général, soit de trouver un graphe ambigu dans lequel il n'y a pas de $[t+1]$bi-source.

Enfin, nous avons commencé à étudier le problème du consensus byzantin dans un système sans signatures. Alors que nous savions déjà qu'une $[2t+1]$bi-source était suffisante, nous avons montré que la présence d'une $[2t]$bi-source ne l'était pas, ce qui à notre connaissance n'avait jamais été démontré auparavant. Pour cela nous avons introduit la notion de graphe bi-ambigu.

Si l'existence d'un graphe bi-ambigu implique l'impossibilité de résoudre le problème du consensus byzantin sans signatures, il reste à montrer que l'absence d'un tel graphe suffit à résoudre le consensus.

Nous avons utilisé un modèle dans lequel les liens pouvaient être soit asynchrones, soit synchrones dans les deux sens (c'est-à-dire que si le lien $p \to q$ est synchrone alors le lien $q \to p$ l'est aussi). Cependant, il existe dans la littérature d'autres modèles plus fins, que ce soit les liens partiellement synchrones ou les IO-processus. Bien évidemment, les résultats d'impossibilités s'appliquent malgré tous dans ces modèles, car ces derniers généralisent le nôtre (que nous avons appelé « propriété de synchronie »), par contre, il serait intéressant de regarder si l'absence de graphe ambigu suffit à résoudre le problème du consensus avec ces nouvelles hypothèses de synchronie.

Références

[1] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier et S. Toueg : On implementing omega with weak reliability and synchrony assumptions. *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, page 314, 2003.

[2] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier et S. Toueg : Communication-efficient leader election and consensus with limited link synchrony. *In PODC '04 : Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 328–337, New York, NY, USA, 2004. ACM.

[3] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier et S. Toueg : Consensus with byzantine failures and little system synchrony. *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, pages 147–155, 2006.

[4] T. D. Chandra, V. Hadzilacos et S. Toueg : The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, 1996.

[5] C. Dwork, N. Lynch et L. Stockmeyer : Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[6] M. J. Fischer, N. A. Lynch et M. S. Paterson : Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[7] M. Hamouma, A. Mostefaoui et G. Trédan : Byzantine consensus with few synchronous links. *Principles of Distributed Systems*, pages 76–89, 2010.

[8] K. P. Kihlstrom, L. E. Moser et P. M. Melliar-Smith : Byzantine fault detectors for solving consensus. *The Computer Journal*, 46:2003, 2003.

[9] L. Lamport, R. Shostak et M. Pease : The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):401, 1982.

[10] M. Pease, R. Shostak et L. Lamport : Reaching agreement in the presence of faults. *Journal of the ACM*, 27:228–234, 1980.