Consensus : Un rapide tour d'horizon

Christophe Grégoire

Institut d'informatique
Facultés Universitaires Notre Dame de la Paix
5000 Namur, Belgique
cgregoir@info.fundp.ac.be

Résumé Les protocoles qui résolvent des problèmes d'agrément sont des éléments essentiels pour les applications distribuées tolérantes aux fautes. Parmi ceux-ci, nous pouvons mettre en évidence le plus important d'entre eux : le consensus. Celui-ci peut être considéré comme étant le bloc de base de tous les autres problèmes d'agrément. Dans cet article, nous décrirons le consensus et les difficultés sous-jacentes aux systèmes asynchrones tolérants aux pannes et nous présenterons quelques unes des solutions les plus importantes.

1 Introduction

Le consensus dans un système informatique distribué peut être, dans un premier temps, comparé à une réunion où différents acteurs proposent une solution à un problème. Le but de cette réunion est que tout le monde en sorte en ayant choisi la même solution. Nous simplifions le consensus en excluant toute forme de négociation. Ce qui implique que la solution décidée est une des solutions proposées.

La métaphore de la réunion, ne laisse pas transparaître toute la complexité de notre sujet. En effet, nous travaillons dans un système distribué. Les participants à la réunion ne se trouvant pas dans la même pièce, ils communiquent par l'échange d'emails[1]. Ce moyen de communication est dit asynchrone[2] et fiable[3]. De plus, nous voulons pouvoir tolérer les pannes franches[4], ce qui peut être vu dans une réunion par le départ d'un participant sans prévenir le reste du groupe. Ces deux extensions seront au coeur de ce papier.

Contrairement aux idées reçues, le consensus n'a pas qu'un intérêt purement théorique. Bien au contraire, étant à la base des autres problèmes d'agrément (plus complexes), tous les résultats théoriques découlant de son étude sont également applicables à ces derniers.

[1] L'utilisation d'une mémoire partagée, est également envisageable dans un système informatique, mais sa transposition dans la métaphore de la réunion est plus délicate.

[2] Nous ne connaissons pas le délai de transmission et de traitement d'un message.

[3] Du moins, nous faisons l'hypothèse que si le destinataire du message va consulter son courrier, alors il recevra, tôt ou tard, les messages lui ayant été envoyés.

[4] Une panne franche ou panne correspond à l'arrêt brutal d'un participant au consensus.

Dans la suite, nous définirons plus précisément le consensus (Sect. 2). Une fois le problème bien compris, nous montrerons et discuterons l'impossibilité de le résoudre de manière déterministe dans un système asynchrone avec des pannes (Sect. 3). Nous étudierons ensuite (Sect. 4) les moyens de contourner cette impossibilité. Cela nous permettra, dans la section 5, de décrire deux algorithmes résolvant le consensus. Après cette étude approfondie, nous explorerons les répercutions de ces résultats sur des problèmes plus pratiques (Sect. 6). Finalement, nous conclurons cet article par la section 7.

2 Le problème

Avant d'entrer dans les détails du consensus, nous allons commencer par poser clairement notre cadre de travail et les concepts associés.

2.1 Les concepts

Il est indispensable de bien poser le vocabulaire que nous allons utiliser tout au long de ce papier et qui est récurrent dans le domaine de la tolérance aux fautes (pannes). Nous allons donc définir les types de fautes ainsi que les concepts associés et le système asynchrone.

Les fautes Nous sommes dans un monde dans lequel les processus peuvent être sujets à des fautes de différents types. Elles vont de la panne (arrêt brutal du processus) aux comportements byzantins (arbitraire) en passant par les omission (un processus n'envoie ou ne reçoit pas un message) et les fautes dues au temps (un processus répond trop tôt ou trop tard à un message).

Nous allons nous concentrer uniquement sur les pannes (le type de faute le plus simple). Nous pouvons dès lors distinguer les processus corrects (ne tombant pas en panne durant une exécution infinie) des processus incorrects.

Notons dès à présent que la définition d'un processus incorrect est très large. En effet, elle nous dit qu'un processus est incorrect s'il tombe en panne durant une exécution infinie. Un processus fonctionnant correctement à un moment donné peut donc être incorrect car il tombera en panne dans un futur plus ou moins proche. Cette définition, difficilement vérifiable en pratique, est faite pour clarifier les autres définitions se basant sur cette dernière. Un processus correct pourrait être défini comme un processus ne tombant pas en panne pas durant l'exécution d'un algorithme, mais cela a pour inconvénient de dépendre du problème. C'est pourquoi nous utiliserons la définition générale pour les démonstrations théoriques et la deuxième pour les questions plus pragmatiques.

Remarquons enfin que la définition même d'un processus correct, nous place dans le modèle *crash/no recovery*. Cela signifie qu'un processus tombant en panne ne peut plus rejoindre le groupe des processus prenant part au consensus. Cet aspect est discuté plus en profondeur dans [21].

Le système Les processus ne peuvent communiquer entre eux que par l'échange de messages au travers de canaux *fiables*. Ceux-ci garantissent que si un processus envoie un message à un autre processus et que ce dernier est correct, alors il finira, tôt ou tard, par recevoir ce message. Les canaux fiables sont des hypothèses très exigeantes utilisées principalement pour démontrer certaines propriétés dans un cadre tout à fait général. Par facilité, certains auteurs travaillent avec des hypothèses plus faibles : les canaux *quasi-fiables*. Ils garantissent que si un processus p envoie un message m à un processus q et que p et q sont *tous les deux* corrects, alors q finira, tôt ou tard, par recevoir m. Ce qui correspond, en fait, au standard TCP [9].

De plus, nous considérons que nous travaillons dans un système asynchrone, i.e. le délai de transmission (vitesse du réseau) et le temps de traitement (vitesse relative des processus) d'un message n'ont pas de bornes.

2.2 Le consensus

Formellement, le consensus est défini par les primitives *propose*(v_i), par laquelle le processus p_i propose sa valeur initiale et *decide*(v), par laquelle un processus décide une valeur. Cette décision devant satisfaire les propriétés suivantes [4] :

- *Terminaison :* Tout processus correct finit par décider.
- *Validité :* Si un processus décide v, alors v a été proposé par un processus.
- *Agrément :* Deux processus corrects ne peuvent décider différemment.

La condition d'agrément autorise que des processus incorrects décident différemment des processus corrects. Dans ce papier, nous considèrerons le consensus *uniforme* dans lequel nous remplaçons la condition d'agrément par celle d'*agrément uniforme* :

- *Agrément uniforme :* Deux processus (corrects ou non) ne peuvent décider différemment.

Remarquons qu'aucune hypothèse n'est posée sur la nature des valeurs proposées par les participants. Cette valeur pourra tantôt être un ensemble d'identifiants de processus afin de composer dynamiquement un groupe et tantôt être un ensemble de messages (ou de leur identifiant) afin que tous les participants délivrent les messages dans le même ordre. Nous reviendrons plus en détail sur ces exemples dans la section 6.

3 L'impossibilité

Fischer, Lynch et Paterson ont prouvé que dans un système asynchrone (avec des canaux fiables), il n'existait aucun algorithme déterministe pouvant résoudre le consensus quand (au moins) un processus peut tomber en panne [16]. Cette impossibilité est communément appelée "*impossibilité FLP*" (FLP impossibility).

De manière intuitive, nous pouvons constater que ce résultat vient du fait qu'il est impossible de distinguer un processus lent d'un processus

tombant en panne. Autrement dit, quand un processus attend un message d'un autre processus, il ne peut pas fixer un délai après lequel il arrêtera d'attendre, sous peine de violer la propriété d'agrément. S'il attend alors que le processus est tombé en panne, il peut violer la propriété de terminaison.

Certains, ayant un regard plus pragmatique, objecteront que le modèle asynchrone n'est pas réaliste. Dans les réseaux réels, il y a moyen de fixer une borne au délai de transmission d'un message. Typiquement, un LAN avec un timeout de 30 secondes permet de détecter un processus en panne avec une probabilité proche de un. Ce qui revient, en fait, à un modèle synchrone.[5] Dans ce cas, il est vrai que l'impossibilité n'a plus de sens car nous sortons des hypothèses de départ.

Cette solution n'est cependant pas dépourvue d'inconvénients. En effet, nous devons faire un compromis entre les suspicions incorrectes (dues à un timeout trop court) et une réaction rapide face à la panne d'un processus. Un temps de réaction de 30 secondes est inacceptable pour une application temps réel. Or, si nous réduisons le timeout, nous augmentons la probabilité de suspicions incorrectes. Cette probabilité peut être faible en fixant le timeout à 15 secondes, mais devient non négligeable si l'on passe à 0,5 seconde. Dans ce cas, il n'est plus approprié de parler d'un système synchrone et l'impossibilité retrouve tout son sens.

En d'autres termes, ajouter des timeouts au système asynchrone n'est pas suffisant pour contourner l'impossibilité FLP [6]. Ce qui nous oblige à trouver une solution rigoureuse pour résoudre le consensus face à des suspicions incorrectes fréquentes.

4 Les modèles

Comme nous venons de le montrer, il est impossible de trouver un algorithme déterministe permettant de résoudre, dans *tous les cas*, le consensus dans un système asynchrone en la présence de pannes. L'impossibilité FLP nous oblige donc à définir l'ensemble des exécutions dans lesquelles un algorithme donné résoud le consensus [20].

Nous ne considèrerons pas ici l'approche non déterministe permettant de résoudre le problème en introduisant un aspect aléatoire au modèle asynchrone ou dans les processus [6,10]. De même que l'approche utilisant des segments de mémoire partagée avec, par exemple, des registres pouvant prendre les trois valeurs "*read-modify-write*" [15][6].

Nous nous attarderons, par contre, sur deux approches permettant de trouver des algorithmes déterministes fontionnant par échange de messages. L'une ajoutant des hypothèses restrictives au modèle asynchrone, alors que la seconde garde ce dernier intact et y ajoute une sorte d'oracle. Nous terminerons ce tour d'horizon en montrant une approche hybride.

[5] Il y a peu de fausses suspicions dues au temps, i.e. aucun signe de vie du processus avant le timeout.

[6] Cette approche ne fonctionne pas sur le principe de l'échange de messages adopté jusqu'à présent de ce papier.

4.1 Les hypothèses supplémentaires

Les *hypothèses supplémentaires sur le modèle asynchrone* rejettent les mauvaises exécutions par l'introduction d'une synchronisation limitée et n'émettent aucune hypothèse quant au nombre de pannes supportées [5,6,7,8,18]. Par exemple, le modèle *partial synchrony* [5], suppose qu'il existe un temps t après lequel il n'y a plus de suspicions incorrectes dues au temps ; alors que le modèle *timed asynchronous* [7] suppose que le système passe par des périodes *stables*[7] et *instables*.

4.2 Les détecteurs de défaillances

Les *détecteurs de défaillances*[8] sont des modules agissant comme des oracles en suspectant les processus qui sont susceptibles d'être tombés en panne[9] [22,23]. Cette approche est définie dans le modèle asynchrone, mais oblige à poser une limite sur le nombre de processus incorrects.[10]

Il existe deux grandes familles de détecteurs de fautes : ceux donnant une liste de processus suspectés, et ceux renvoyant un seul processus dans lequel ils ont confiance (i.e. n'est pas en panne). Cette deuxième famille peut être implémentée à partir de la première.

Un détecteur de pannes (renvoyant une liste de suspects) est communément décrit pas les propriétés de *complétude* (completeness) et d'*exactitude* (accuracy). La complétude portant sur la suspicion des processus incorrects, alors que l'exactitude porte sur la suspicion des processus corrects. Ces propriétés peuvent être nuancées comme suit :

- **Complétude forte** (Strong completeness) : Tout processus incorrect est finalement suspecté, pour toujours[11], par *tous* les processus corrects.
- **Complétude faible** (Weak completeness) : Tout processus incorrect est finalement suspecté, pour toujours, par (au moins) *un* processus correct.
- **Exactitude forte** (Strong accuracy) : Aucun processus n'est suspecté avant qu'il soit tombé en panne.
- **Exactitude faible** (Weak accuracy) : Au moins un processus correct n'est jamais suspecté.
- **Exactitude finalement forte** (Eventual strong accuracy) : Il existe un temps t après lequel les processus corrects ne sont plus suspectés par les autres processus corrects.

[7] Une période stable est une période durant laquelle aucune suspicion incorrecte due aux timeouts trop courts ne survient.

[8] Un détecteur de défaillances (Failure detector) pourra aussi être appelé détecteur de fautes ou de pannes.

[9] Il existe d'autres types de détecteurs de pannes. Par exemple un détecteur de défaillances byzantines, mais cela sort du cadre que nous nous sommes fixés.

[10] Afin de tolérer les partionnements du réseau et éviter que les processus de partitions différentes ne décident différemment (violent la propriété d'agrément).

[11] La notion de "pour toujours" peut ici encore être traduite par "jusqu'à la fin de l'exécution de l'algorithme".

- **Exactitude finalement faible** (Eventual weak accuracy) : Il existe un temps t après lequel au moins un processus correct n'est plus suspecté par les autres processus corrects.

Chandra et Toueg ont défini les classes de détecteurs de pannes suivantes :

- $\mathcal{P}$ (perfect) : complétude forte et exactitude forte.
- $\diamond\mathcal{P}$ (eventually perfect) : complétude forte et exactitude finalement forte.
- $\diamond\mathcal{S}$ (eventually strong) : complétude forte et exactitude finalement faible.
- $\diamond\mathcal{W}$ (eventually weak) : complétude faible et exactitude finalement faible.

Ces mêmes auteurs ont prouvé qu'il existait une hiérarchie entre ces différents détecteurs de pannes, et que le plus faible dont nous avons besoin pour résoudre le consensus est $\diamond\mathcal{W}$ [23]. Ils ont également prouvé que $\diamond\mathcal{S}$ est équivalent à $\diamond\mathcal{W}$[12] [22]. Vu que les algorithmes se basant sur $\diamond\mathcal{S}$ sont plus simples, nous allons désormais travailler avec ce dernier.

Notons que les détecteurs de pannes sont souvent mal compris et considérés, à tort, comme des artefacts de théoriciens. Certains diront que le détecteur de pannes $\diamond\mathcal{S}$ n'est pas implémentable dans un système asynchrone. Selon Guerraoui et Schiper [20], cette question est équivalente à se demander si les périodes stables du modèle timed asynchronous (cfr. Sect. 4.1) sont implémentables. Cette question n'a aucun sens car la stabilité d'une période dépend de l'utilisation du réseau, sur laquelle aucune hypothèse n'est faite. Il en est de même pour l'implémentabilité du détecteur de fautes $\diamond\mathcal{S}$ car sa définition doit être vue comme une spécification. Les timeouts permettant de l'implémenter doivent donc être choisis aussi petits que possible, mais pas trop petits, pour suivre la spécification avec une probabilité (proche) de un. Les implémentations peuvent donc être, par exemple, de type ping (requête/réponse) ou heartbeat (envoi périodique de messages).

Revenons un instant sur la distinction que nous avons faite au début de cette section. Parmi les détecteurs de défaillances ne renvoyant qu'un processus de confiance, nous pouvons mettre en évidence la famille de détecteurs de pannes Ω respectant la propriété suivante :

- **Eventual Leader** : Il existe un temps t après lequel tous les processus corrects font confiance, pour toujours, à exactement un processus correct p_l.

Il a été prouvé que cette famille est équivalente à $\diamond\mathcal{S}$ [22].

4.3 Approches hybrides

Certains algorithmes fonctionnent en combinant plusieurs modèles. Par exemple, l'approche hybride d'Aguilera et Toueg [11] résout le consensus très rapidement en utilisant un détecteur de pannes, et s'il n'y arrive pas, il passera à une approche probabiliste.

[12] $\diamond\mathcal{W}$ est réductible à $\diamond\mathcal{S}$ et vice versa.

Nous verrons dans la section 5, qu'en l'absence de suspicions incorrectes, les algorithmes basés sur les détecteurs de pannes sont très performants. Mais si le réseau se comporte de manière fortement asynchrone, ces derniers peuvent avoir du mal à décider. D'où l'intérêt de passer à une autre approche moins sensible à ce phénomène.

5 Les algorithmes

Nous allons poursuivre en nous focalisant sur le modèle asynchrone avec détecteurs de défaillances. Cette approche est la plus simple et la plus générale car elle n'émet aucune restriction sur le modèle asynchrone, mais bien sur le nombre de processus suspectés.

Il existe une multitude d'algorithmes déterministes résolvant le consensus grâce à l'aide fournie par un détecteur de pannes. Nous pouvons toutefois mettre en évidence deux familles : celle basée sur les communications de groupe et celle basée sur les quorums. Selon nous les communications de groupe sont plus générales que les quorums[13], c'est pourquoi nous ne décrirons que des algorithmes basés sur les communications de groupe. Pour information, l'article [2] traite de la résolution du consensus à l'aide de quorums.

Notons qu'il a été prouvé que tout algorithme résolvant le consensus en utilisant le détecteur de pannes $\diamond\mathcal{S}$ résout aussi le consensus uniforme [19].

Nous allons décrire deux algorithmes très similaires quant aux hypothèses, mais traitant le problème de manières légèrement différentes. Ces deux algorithmes fonctionnent en échangeant des messages au travers de canaux quasi-fiables dans un système asynchrone augmenté d'un détecteur de défaillances ($\diamond\mathcal{S}$ pour l'un et Ω pour l'autre). Ils imposent qu'une majorité de participants soient corrects. Le non-respect de cette condition causera une violation de la propriété de terminaison, mais en aucun cas de celle d'agrément (Ils ne garantissent plus que ce qui est bien arrivera (*liveness*), mais bien que ce qui n'est pas bien n'arrivera pas (*safety*)).

Les deux approches sont similaires en ce qu'elles procèdent par rondes asynchrones. Autrement dit, lors d'une ronde, un leader (coordinateur) va tenter d'imposer une décision. Ce leader ne peut pas être fixé pour toute l'exécution d'un consensus car ce processus peut être incorrect, chose qu'il est impossible à déterminer au commencement. Les détecteurs de pannes sont là pour suspecter le leader ($\diamond\mathcal{S}$) ou pour l'élire (Ω). Les rondes sont dites asynchrones car à un moment donné, les différents participants peuvent se trouver à des étapes différentes de l'algorithme, voire à des rondes différentes.

Le dernier point commun de ces deux algorithmes est qu'ils utilisent *Reliable Broadcast*, une opération de base des communications de groupe. Elle permet d'effectuer un broadcast fiable et est définie par les primitives *RBCAST(m)* et *R-deliver(m)* satisfaisant les propriétés suivantes [4] :

[13] Les quorums sont plus simples, mais ne peuvent fonctionner qu'au sein d'un modèle transactionnel.

- *Validité :* Si un processus correct exécute RBCAST(m), alors tous les processus corrects finiront, tôt ou tard, par effectuer R-deliver(m).
- *Agrément :* Si un processus correct effectue R-deliver(m), alors tous les processus corrects finiront, tôt ou tard, par effectuer R-deliver(m).
- *Intégrité :* Pour tout message m, tous les processus corrects effectueront au plus une fois R-deliver(m) et cela seulement si un processus a effectué RBCAST(m).

5.1 Chandra et Toueg [22]

L'algorithme de Chandra et Toueg se base sur le *paradigme du coordinateur tournant* (rotating coordinator paradigm), i.e. à la ronde (r), le coordinateur (c) sera le processus avec le numéro $((r \ mod \ n) + 1)$[14].

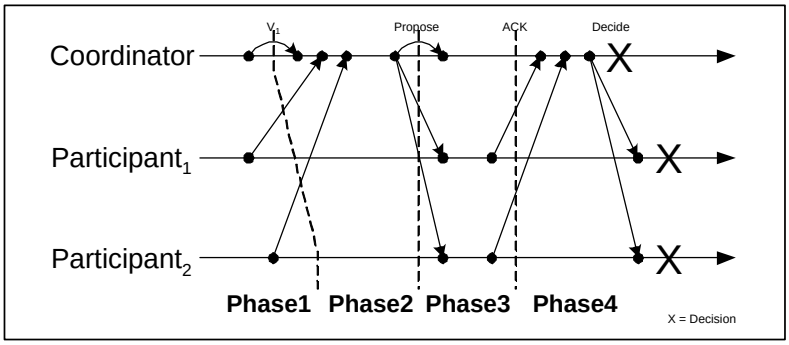

Fig. 1. Déroulement idéal d'une ronde de l'algorithme de Chandra et Toueg.

Une ronde est découpée en quatre phases (cfr. Fig. 1). Dans la **première phase**, chaque processus envoie son estimation courante au coordinateur courant (c), en y joignant le numéro de la ronde durant laquelle cette estimation a été mise à jour pour la dernière fois. Dans la **deuxième phase**, le coordinateur attend une majorité de ces estimations, sélectionne la plus récente et l'envoie à tous les processus. Dans la **troisième phase** quand p_i reçoit l'estimation du coordinateur, il lui envoie un acquit (ACK) pour indiquer au coordinateur qu'il adopte la nouvelle estimation. Si le détecteur de fautes de p_i suspecte le coordinateur avant de recevoir son estimation, p_i enverra un acquit négatif ($NACK$) au coordinateur et passera à la ronde suivante. Finallement, lors de la **quatrième phase**, le coordinateur attend une majorité d'ACKs ou un NACK. S'il reçoit une majorité d'ACKs, il décide l'estimation courante et effectue un Reliable Broadcast de cette décision. Quand les autres processus reçoivent cette

[14] En supposant que les processus soient numérotés de 1 à n

décision, ils décident immédiatement. Par contre, si le coordinateur reçoit un NACK, il passe à la ronde suivante sans décider et sans envoyer de messages.

Nous remarquons que l'algorithme se termine en une ronde s'il n'y a pas de suspicions incorrectes et que le coordinateur ne tombe pas en panne. De plus, une optimisation immédiate peut être obtenue en court-circuitant la première phase de la première ronde (pas des suivantes) comme représenté sur la figure 2.

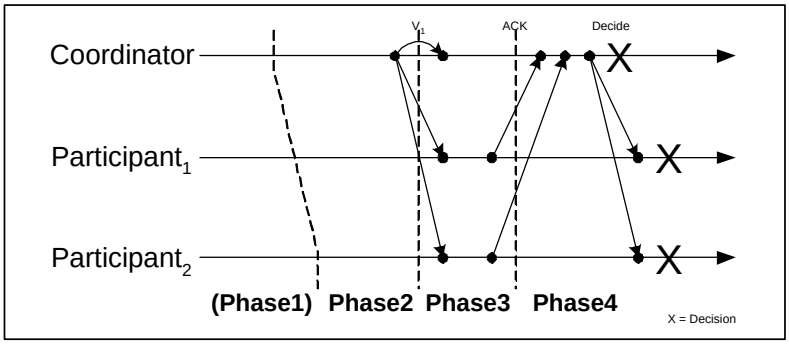

Fig. 2. Optimisation du premier tour du Consensus de Chandra et Toueg.

5.2 Paxos [13]

Paxos s'inscrit dans le paradigme "*leader-based*" et utilise un détecteur de pannes de type Ω. C'est d'ailleurs ce dernier qui est utilisé pour déterminer le leader. Une implémentation possible de ce détecteur serait d'utiliser un détecteur de pannes de type $\diamond\mathcal{S}$ et de renvoyer le processus qui n'est pas suspecté et ayant le numéro (identifiant) le plus petit.

Si le leader venait à tomber en panne, le détecteur de pannes finira, tôt ou tard, par en déterminer un nouveau.

Si un processus (p_i) se considère comme étant le leader, il commencera une nouvelle ronde. Ce processus utilisera un numéro de ronde du type $i,\ i+n,\ i+2n,\ \ldots$ (dans cet ordre)[15]. Il est évident que plusieurs processus peuvent se considérer comme étant le leader, les processus se trouveront donc dans des rondes de numéro différent. Mais la propriété du détecteur de défaillance Ω garantit qu'après un certain temps, tous les participants considèreront le même processus comme leader.

[15] De sorte que les numéros soient uniques. Avec n correspondant au nombre de processus prenant part au consensus et le numéro (identifiant) des processus compris entre 1 et n

Une ronde se compose de deux phases (Fig. 3) : dans la première le leader vérifie si les autres processus ont déjà décidé une valeur avec un numéro de ronde supérieur au sien ; et dans la seconde, il essaie de décider une valeur.

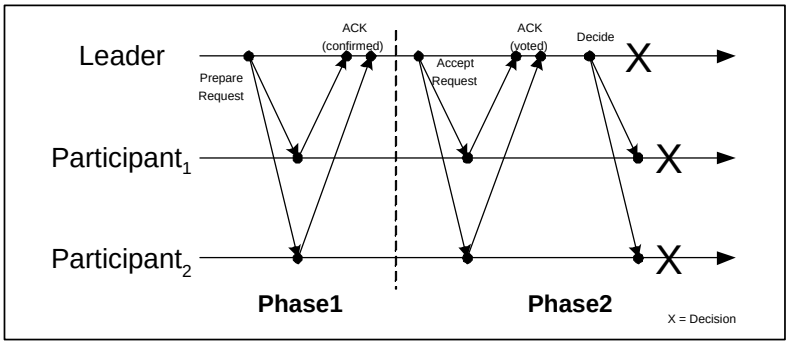

Fig. 3. Un tour de Paxos

Dans la **première phase**, le leader p_l envoie une *prepare request* avec un numéro de ronde rp_l à tous les autres processus. Quand un processus p_i reçoit la requête, il accepte la proposition si la proposition avec le numéro de ronde le plus élevé acceptée jusqu'à présent a un numéro de ronde inférieur à rp_l et n'a pas encore répondu à une requête avec un numéro de ronde supérieur à rp_l. Si p_i accepte la proposition, il envoie un acquit (ACK) avec son estimation courante de la décision et le numéro de la ronde durant laquelle l'estimation a été mise à jour pour la dernière fois. Sinon (si p_i reçoit une requête avec un numéro de ronde plus petit que le numéro de ronde de toute les requêtes reçues précédemment), p_i rejette la proposition et envoie un acquit négatif ($NACK$) au leader.

Le leader attend une majorité de réponses. Si toutes les réponses sont des ACKs, le leader met-à-jour son estimation courante de la décision à l'estimation la plus récente contenue dans les ACKs. Sinon, s'il reçoit au moins un NACK, la ronde est annulée immédiatement et le leader passe à la ronde suivante.

Dans la **deuxième phase**, le leader envoie une *accept request* à tous les autres participants avec l'estimation mise à jour. Cette requête est traîtée de la même manière que la *prepare request* : les autres processus peuvent mettre-à-jour leurs estimations et répondre avec un ACK, ou bien un NACK (à la différence près que l'ACK ne contient que le numéro de ronde, et pas l'estimation). Si le leader ne reçoit que des ACKs, il enverra un *decision message* avec la décision en utilisant Reliable Broadcast. Lors de la réception de ce message, le processus décide immédiatement. Si le leader reçoit au moins un NACK, il abandonne la ronde et passe à la suivante.

Une optimisation très simple de l'algorithme peut éviter à chaque ronde de devoir effectuer la première phase, comme dans la figure 4. En effet, dans la phase 2, quand un processus p envoie au leader p_l un NACK, il peut y adjoindre le numéro de la dernière ronde durant laquelle il a accepté une prepare request. Le leader peut donc ainsi passer directement à un numéro de ronde supérieur à ce dernier. Cette optimisation permet à p_l de résoudre directement le consensus en diminuant le nombre d'étapes de communication.

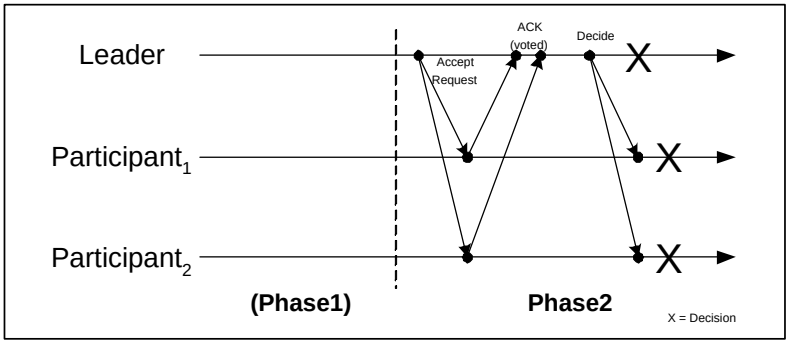

Fig. 4. Paxos optimisé.

5.3 Analyse

Nous venons de décrire des algorithmes similaires quant à leurs hypothèses de base, ce qui nous permet de pouvoir les comparer assez aisément d'un point de vue purement qualitatif. Notre but ici n'est pas de faire des tests de performance comme dans [17]. Notre démarche se rapprochera d'avantage de celle de Schiper dans [3] où l'auteur décrit un nouvel algorithme (Early Consensus) et le compare à celui de Chandra et Toueg en terme d'*étapes de communication* (*communication steps*).

Nous savons que ce qui coûte cher dans les algorithmes distribués, c'est l'envoi des messages sur le réseau. A côté du temps passé à envoyer et à attendre des messages, le temps de calcul semble dérisoire. C'est pourquoi, pour qu'un algorithme soit performant, il est important d'échanger le moins de messages possible.

Nous pouvons constater que le nombre d'étapes de communication pour les versions originales de l'algorithme de Chandra et Toueg est de 4, alors qu'il est de 5 pour Paxos. Les versions optimisées en comptent quant à elles 3 pour la permier ronde et 4 pour les suivantes dans le premier algorithme et 3 dans tous les cas dans le second. Paxos semblerait donc plus efficace si le consensus ne réussit pas lors de la première ronde. C'est ce que confirme [17] par des tests de performance.

L'élection du leader ne se passant pas de la même manière dans les deux cas, il pourrait être intéressant de regarder de plus près l'incidence de cet élément sur les performances. Cet aspect dépassant le cadre de ce papier, nous ne l'approfondirons pas.

Remarquons, pour conclure, qu'un détecteur de pannes est un module indépendant ayant pour seul but de suspecter des processus. Nous nous sommes limités à la détection des processus sujets à des pannes franches. Mais il existe des détecteurs de fautes byzantines (ou autres). Malheureusement, face à des comportement byzantins, le consensus a besoin d'une majorité des deux tiers de processus corrects pour fonctionner [14]. A titre d'exemple, SecureRing offre différents services en tolérant les fautes byzantines [12].

6 Exemple pratiques

Afin de montrer l'intérêt pratique du consensus, nous allons exposer ici deux problèmes intimement liés à ce dernier.

6.1 Atomic Broadcast

Atomic Broadcast (ou *Total Order Broadcast*) permet d'effectuer un broadcast fiable et ordonné[16].

Atomic Broadcast est formellement défini par les primitives *ABCAST(m)* et *A-deliver(m)* satisfaisant les propriété suivantes [4] :

- *Validité :* Si un processus correct exécute ABCAST(m), alors tous les processus corrects finiront, tôt ou tard, par effectuer A-deliver(m).
- *Agrément uniforme :* Si un processus (correct ou non) effectue R-deliver(m), alors tous les processus corrects finiront, tôt ou tard, par effectuer A-deliver(m).
- *Intégrité :* Pour tout message m, tous les processus corrects effectueront au plus une fois A-deliver(m) et cela seulement si un processus a effectué ABCAST(m).
- *Ordre total uniforme :* Si un processus (correct ou non) effectue un A-deliver(m) avant un A-deliver(m'), alors tous les processus effectueront A-deliver(m') seulement après avoir effectué A-deliver(m).

Atomic Broadcast est, par exemple, utilisé comme primitive de base pour la *réplication active*. La réplication consiste à faire des copies d'un service et de faire exécuter ces copies sur différentes machines de sorte que si l'une d'entre elles venait à tomber en panne, les données ne soient pas perdues et le service continue à être disponible. La difficulté réside dans le fait de garder un état cohérent des différentes copies. Une mise à jour d'une donnée sur l'une doit se faire sur toutes les autres et les différentes mises-à-jour doivent se faire dans le même ordre.

Atomic Broadcast peut facilement être implémenté en combinant Reliable Broadcast au consensus. Le consensus servant à déterminer l'ordre dans lequel les messages doivent être délivrés à l'application.

[16] Tous les processus délivrent les messages broadcastés dans le même ordre. Atomic Broadcast ajoute donc une propriété d'ordre à Reliable Broadcast.

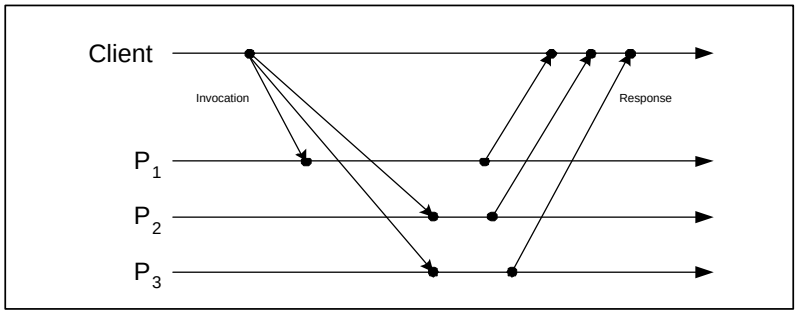

Fig. 5. La réplication active : toute la complexité se trouve dans la primitive broadcastant l'invocation de manière fiable et ordonnée.

6.2 Group Membership Problem

La gestion dynamique de l'appartenance d'un processus à un groupe permet de passer d'un modèle statique (aucune possibilité de joindre ou quitter le groupe après la création de ce dernier) à un modèle dynamique. Dans la section 6.1, nous avons défini Atomic Broadcast dans un modèle statique, mais une version dynamique existe [4].

L'évolution du groupe sera constituée d'une succession de vues. Durant une vue, le groupe ne changera pas. Si un processus veut rejoindre le groupe ou le quitter, cela se fera en installant une nouvelle vue. Nous pouvons voir le lien de ce problème avec le consensus en ce qu'il consiste à se mettre d'accord sur les membres faisant partie de la nouvelle vue.

Le Group Membership se compose de deux primitives. *join(p)* exécutée par un processus actuellement membre du groupe permet d'ajouter p au groupe. *leave(p)* exécutée par p ou tout autre processus membre du groupe permet d'enlever p du groupe[17]. Une dernière primitive *install(v)*, celle-ci étant appellée par le Group Membership, permet de signaler à l'application que la vue courante a changé et est devenue v.

Le Group Membership peut être défini par les propriétés suivante [4] :

- *Validité :* Considérons deux vues consécutives v_i et v_{i+1}. Si $p \in v_i \setminus v_{i+1}$, alors un processus a exécuté *leave(p)*. Si $p \in v_{i+1} \setminus v_i$, alors un processus a exécuté *join(p)*.
- *Agrément :* Si un processus p dans la vue v_i installe la vue v_{i+1}, et qu'un processus q dans la vue v_i installe la vue v'_{i+1} alors $v_{i+1} = v'_{i+1}$.
- *Terminaison :* Si un processus p dans la vue v, $p \in v$, exécute *join(q)*, alors, sauf si p tombe en panne, une vue v' telle que $q \in v'$ ou $p \notin v'$ sera finalement installée. Si un processus p dans la vue v, $p \in v$, exécute *leave(q)*, alors, sauf si p tombe en panne, une vue v' telle que $q \notin v'$ ou $p \notin v'$ sera finalement installée.

[17] Tout processus incorrect devra être retiré du groupe par l'application utilisant Group Membership

7 Conclusion

Nous avons vu tout au long de cet article que les problèmes, les plus simples en apparence, peuvent devenir extrêmement complexes une fois transposés dans un système distribué. Les résultats théoriques découlant de l'étude du consensus touchent également des problèmes pratiques fréquemment rencontrés. Ils nous permettent de connaître, à l'avance, les difficultés auxquelles nous serons confrontés, les approches pour les contourner, ainsi que les limites de ces dernières.

Les algorithmes écrits au-dessus du consensus sont plus simples à concevoir et à valider, mais peuvent être moins efficaces que des approches monolithiques. C'est pourquoi il est important de concevoir des algorithmes optimisés pour résoudre le consensus.

Malheureusement, les algorithmes disponibles dans la littérature sont décrits de manière informelle et souvent ambiguë. De plus, certains détails sont volontairement tus (exécutions multiples, ...) pour faciliter la compréhension de leur fonctionnement, mais cela rend leur implémentation d'autant plus difficile.

Enfin, les concepts utilisés ne sont compris que par la communauté assez restreinte de chercheurs travaillants sur le sujet. Il est donc impératif que ces derniers fassent un effort pour rendre ceux-ci accessibles à un plus grand nombre.

Ce papier ne se voulant pas exhaustif a pour but premier d'introduire le lecteur aux difficultés cachées des systèmes distribués tolérants aux pannes, dont le consensus est le bloc de base.

Références

1. Basu A., Charron-Bost B., and Toueg S. Solving problems in the presence of process crashes and lossy links. Technical Report TR96-1609, Cornell University, Computer Science Department, October 1996.
2. Mestafaoui A., Rajsbaum S., and Raynal M. A versatile and modular consensus protocol. Technical Report 1427, Institut en Recherche Informatique et Systèmes Aléatoires (IRISA), December 2001.
3. Schiper A. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3) :149–157, April 1997.
4. Schiper A. Lecture notes for the graduate school I&C (EPFL – LSR), October 2002.
5. Dwork C., Lynch N., and Stockmeyer L. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2) :288–323, April 1988.
6. Dolev D., Dwork C., and Stockmeyer L. On the minimal synchrony needed for distributed consensus. *Journal of the ACM*, 34(1) :77–97, January 1987.

7. Cristian F. and Fetzer C. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6) :642–657, June 1999.
8. Attiya H., Dwork C., Lynch N., and Stockmeyer L. Bounds on the time to reach agreement in the presence of timing uncertainty. *Journal of the ACM*, 41(1) :122–152, January 1994.
9. Information Sciences Institute. RFC 793, 1981. Edited by Jon Postel. Available at http ://rfc.sunsite.dk/rfc/rfc793.html.
10. Aspnes J. Randomized protocols for asynchronous consensus, September 2002.
11. Aguilera M. K. and Toueg S. Failure detection and randomization : A hybrid approach to solve consensus. *SIAM J. Comput.*, 28(3) :890–903, 1998.
12. Kihlstrom K., Moser L, and Melliar-Smith P. The SecureRing group communication system. *ACM Transactions on Information and System Security*, 4(4) :371–406, 2001.
13. Lamport L. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2) :133–169, 1998.
14. Lamport, Shostak, and Pease. The byzantine generals problem. In *Advances in Ultra-Dependable Distributed Systems, N. Suri, C. J. Walter, and M. M. Hugue (Eds.), IEEE Computer Society Press.* 1995.
15. Michael C. Loui and Hosame H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. In Franco P. Preparata, editor, *Advances in Computing Research*, volume 4, pages 163–183. JAI Press, 1987.
16. Fischer M., Lynch N., and Patterson M. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32 :374–382, April 1985.
17. Hayashibara N., Urbán P., Schiper A., and Katayama T. Performance comparison between the Paxos and Chandra-Toueg consensus algorithms. Technical Report IC-2002-61, Ecole polytechnique Fédérale de Lausanne (EPFL), Faculté d'Informatique et Communications (I&C), 2002.
18. Verissimo P. and Almeida C. Quasi-synchronism : a step away from the traditional fault-tolerant real-time system models. *IEEE TCOS bulletin*, 7(4) :35–39, December 1995.
19. Guerraoui R. Revisiting the relationship between non-blocking atomic commitment and consensus. In Jean-Michel Hélary and Michel Raynal, editors, *Proceedings of the 9th International Workshop on Distributed Algorithms (WDAG95)*, volume 972, pages 87–100, Le Mont-Saint-Michel, France, 1995. Springer-Verlag.
20. Guerraoui R. and Schiper A. Consensus : the big misunderstanding. In *Proceedings of the 6th IEEE Computer Society Workshop on Future Trends in Distributed Computing Systems (FTDCS-6)*, pages 183–188, Tunis, Tunisia, October 1997. IEEE Computer Society Press.

21. Oliviera R., Guerraoui R., and Schiper A. Consensus in the crash-recovery model. Technical Report TR-97/239, EPFL, Dept d'Informatique, July 1997.
22. Chandra T. and Toueg S. Unreliable failure detector for reliable distributed systems. *Communications of the ACM*, 43(2) :225–267, 1996.
23. Chandra T. and Toueg S. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4) :685–722, 1996.