Causal Mutual Byzantine Broadcast

Mathieu Féry, Vincent Kowalski, Florian Monsion, Achour Mostefaoui,
Samuel Pénault, Matthieu Perrin, Guillaume Poignant

Causal Mutual Byzantine Broadcast

Féry Mathieu
Nantes Université, Nantes, France
mathieu.fery@etu.univ-nantes.fr

Kowalski Vincent
LS2N, Nantes Université, Nantes, France
vincent.kowalski@etu.univ-nantes.fr

Monsion Florian
Nantes Université, Nantes, France
florian.monsion@etu.univ-nantes.fr

Mostéfaoui Achour
LS2N, Nantes Université, Nantes, France
achour.mostefaoui@univ-nantes.fr

Pénault Samuel
Nantes Université, Nantes, France
samuel.pénault@etu.univ-nantes.fr

Perrin Matthieu
LS2N, Nantes Université, Nantes, France
matthieu.perrin@univ-nantes.fr

Poignant Guillaume
Nantes Université, Nantes, France
guillaume.poignant@etu.univ-nantes.fr

Résumé

Récemment, la primitive de diffusion mutual-broadcast a été introduite pour capturer la propriété temporelle nécessaire et suffisant à l'implémentation d'un registre atomique dans un système tolérant les pannes. Plus précisément, mutual-broadcast permet à tout processus d'émettre des messages, qui seront reçus par tous les processus. La propriété fondamentale est qu'il est impossible que deux processus "s'ignorent mutuellement", c'est-à-dire que chacun émette un message, puis que chacun reçoive son propre message localement avant le message de l'autre processus.

Dans ce rapport, nous étendons le travail réalisé sur mutual-broadcast aux systèmes soumis aux fautes byzantines. Dans un premier temps, nous montrons qu'il est nécessaire d'ajouter la propriété de réception causale pour que mutual-broadcast reste équivalent aux registres atomiques dans un contexte byzantin, et nous spécifions plus précisément causal-mutual-broadcast. Dans un deuxième temps, nous proposons un algorithme pour implémenter causal-mutual-broadcast et pouvant supporter jusqu'à $t < \frac{n}{3}$ processus byzantins parmi n processus.

1 Introduction

1.1 Contexte

Les systèmes distribués sont des collections de processus communiquant par l'envoie et la réception de signaux utilisés pour résoudre un problème commun. La synchronisation de ces processus est habituellement faite au moyen d'objets partagés. Idéalement, ces objets se comportent comme si une unique copie de l'objet était physiquement partagé par chacun des processus. Ce genre d'abstraction est appelé cohérence forte. Malheureusement, d'après le CAP theorem [2], il est impossible d'assurer à la fois une cohérence forte et la terminaison de toutes les opérations d'un objet dans un système sujet au partitionnement du réseau. Les algorithmes respectant une cohérence forte s'appuient sur les quorums pour outrepasser les limites du partitionnement : chaque fois qu'un processus réalise une opération sur un objet partagé, il transmet un message et attends d'avoir une réponse de la part de la majorité des processus au sein du système.

Récemment, le mutual-broadcast [4] s'est fait remarquer comme une abstraction de broadcast équivalent aux registres atomiques. Autrement dit, le mutual broacast pourrait être apporté à des systèmes comme

un bloc de construction utilisable à la place de quorums pour implémenter une cohérence forte. Bien que le mutual broadcast puisse paraître comme étant la bonne abstraction pour s'occuper des problèmes de sûreté des systèmes distribués, ces systèmes peuvent aussi avoir des problèmes de sécurité. Lamport fut le premier à introduire le concept de byzantin [1]. Un participant byzantin ne respecte pas le code que l'on attend de sa part. Cela peut arriver quand des utilisateurs mal intentionnés essayent activement d'attaquer le système en altérant le code client d'une application, ou simplement en utilisant une version non à jour ou corrompu de cette application. L'objectif de ce projet de recherche est d'implémenter un mutual broadcast permettant à un système distribué de tolérer des byzantins.

1.2 Contribution

Le cmb-broadcast étant une adaptation du Mutual-Broadcast supportant un contexte causal, les services et propriétés seront similaires. Cette abstraction mettra donc à disposition une opération $cmb-broadcast(m)$ qui permettra d'envoyer un message m et son processus d'origine p, à l'ensemble des processus. Elle fournira également un évènement $cmb-deliver(m)$ qui sera appelé lorsque le processus concerné délivre le message m.

1.3 Modèle

— Toutes les canaux de communication entre processus corrects respectent un ordre FIFO et sont fiables.
— Un Message est un formatage d'un signal et d'une méthode d'identification de son processus originel.
— Au sein de l'algorithme toutes les phases doivent être déclenchées dans l'ordre même lorsque les conditions peuvent en activer deux en simultané.

1.4 Définitions

1.4.1 Processus Byzantin :

Un processus Byzantin est un terme couvrant plusieurs situations telles qu'un code client corrompu, plus à jour ou altéré par un utilisateur, un processus crashé, bugué lors de son exécution ou plus généralement un processus qui ne respecte pas le code que l'on attend de sa part.

1.4.2 Proportions :

Nous utiliserons certaines valeurs ci-dessous tout du long de ce document.
— n est le nombre de processus dans le système.
— t est le nombre de Byzantins dans le système. Ici on en supporte $t < (1/3) * n$
— $(n-t)$ est le nombre de processus correct dans le système.
— $(n-t)/2$ est la moitié des processus corrects, plus de $(n-t)/2$ est donc la majorité des processus corrects
— plus de $(n-t)/2+t$, soit $(n+t)/2$, est la majorité des processus corrects où l'on considère la présence potentielle de t Byzantins (Ainsi cette proportion s'approche de $(n-t)/2$ mais en ajoutant le support de jusqu'à t Byzantin)
— $t+1$ est un nombre pratique puisque si on a $t+1$ processus, au moins l'un d'entre eux est forcement correct, puisqu'il y a t Byzantins au pire.

2 Propriétés

2.1 Propriétés de Sûreté :

2.1.1 Validité :

Propriété : Si un processus correct cmb-deliver un message m de la part d'un processus correct p, alors p a cmb-broadcast le message m.

Utilité : Cette propriété assure que cmb-broadcast soit la seule façon d'envoyer un message.

2.1.2 Intégrité :

Propriété : Chaque processus correct ne peut cmb-deliver qu'une seule fois chaque message.

Utilité : Cette propriété permet d'empêcher les doublons de messages.

2.2 Propriétés de vivacité :

2.2.1 Progrès local :

Propriété : Si un processus correct cmb-broadcast un message m, alors il finira par cmb-deliver ce message m.

Utilité : Cette propriété permet d'assurer la terminaison de la transmission d'un message.

2.2.2 Fiabilité :

Propriété : Si un processus correct cmb-deliver un message m, alors tous les processus corrects finiront par cmb-deliver ce message m.

Utilité : Cette propriété permet d'assurer que soit tous les processus corrects reçoivent un message, soit aucun processus correct ne le reçoit.

2.3 Propriété d'ordre mutuel :

Propriété : Si un processus correct p cmb-broadcast un message m, et un autre processus correct p' cmb-broadcast un autre message m', alors il est impossible que p cmb-deliver m avant m' et p' cmb-deliver m' avant m.

Utilité : Cette propriété vérifie que deux écrivains ne s'ignorent pas.

2.4 Propriété de cohérence causale :

Propriété : Si un processus correct p cmb-deliver m avant de cmb-broadcast m', alors aucun processus correct ne cmb-deliver m' avant m.

Utilité : Cette propriété permet d'assurer qu'un message envoyé ait bien une dépendance d'ordonnancement par rapport aux messages précédemment acceptés.

2.5 Propriété FIFO :

Propriété : Si un processus correct cmb-deliver un message m de la part d'un processus p avant un message m' de la part du même processus p alors aucun processus correct ne cmb-deliver m' avant m.

Utilité : Cette propriété permet d'assurer qu'un message reçu ait bien une dépendance d'ordonnancement par rapport aux messages précédemment reçus du même processus.

3 Algorithme

3.1 Signaux

Cet algorithme utilise quatre types de signaux protocolaires :

3.1.1 INIT(m) :

Ce signal est constitué d'un unique élément, le message à transmettre m. Ce signal est utilisé par l'envoyeur p du message m afin d'envoyer m à tous les participants lors de la phase initiale de la communication, appelée Phase d'envoi.

3.1.2 ECHO(m, p) :

Ce signal est constitué de deux éléments : Le message à transmettre m ainsi que l'identifiant du processus ayant envoyé le signal, noté p. Ce signal permet principalement à un processus d'indiquer aux autres processus qu'il a reçu un message.

3.1.3 READY(m, p) :

Ce signal est constitué de deux éléments : Le message à transmettre m ainsi que l'identifiant du processus ayant envoyé le message, noté p. Ce signal permet principalement à un processus de signaler qu'il a remarqué que suffisamment de processus ont reçu le même message m que lui, et souhaite que le message soit cmb-deliver par tous.

3.1.4 ACK(m, p) :

Ce signal est constitué des mêmes éléments que les signaux READY et ECHO. Ce signal est utilisé par les processus pour indiquer aux autres processus avec lesquels il communique pour les prévenir que dans leur cas, ce message est désormais considéré comme accepté. Cela empêche alors de briser la causalité des messages en attendant d'avoir au moins accepté tous les messages acceptés par l'auteur avant de pouvoir accepter les nouveaux messages de sa part.

3.2 Déroulé

Ici l'algorithme se base sur le Reliable Broadcast de Bracha [3], cependant il utilise 4 signaux différents, les 3 de l'algorithme original et le signal ACK ajouté afin de permettre de préserver la causalité entre messages.

Si on se place d'un point de vue d'un processus émetteur, l'algorithme se déroule de la façon suivante.

— Le processus émetteur (Que l'on nommera p) va envoyer le signal protocolaire INIT avec son message à tous les processus du système, que nous nommerons $P*$.

— Lorsque les $P*$ processus recevront un INIT de la part du processus p, ils répondront à tout le système avec le signal protocolaire ECHO pour signaler la réception de cette demande d'envoi.
Cela n'est fait que si tous les ACK précédement émis par p ont déjà été acceptés, pour conserver la causalité des messages. En attendant cela, le message est mis en attente.

— Les autres processus $P*$ attendront de recevoir plus de $n + t/2$ ECHO concernant le message de p. Lorsqu'ils les auront reçu, cela signifiera qu'une majorité de processus corrects (Soit un quorum) auront relayé l'information d'envoi. À ce stade les autres processus qui se trouveront dans cette situation enverront un READY à tous les autres processus, pour indiquer qu'ils ont entendu une majorité de corrects ayant perçu le message initial.

— Dès que les $P*$ processus auront reçu $t + 1$ READY concernant le message de p, cela signifiera qu'au moins 1 correct a entendu une majorité de processus corrects répercuter le message via un ECHO, un READY sera envoyé par chacun des $P*$.

— Lorsque les $P*$ processus auront reçu $(n + t)/2$ READY, les processus non émetteurs peuvent cmb-deliver le message en interne et envoyer une confirmation du cmb-deliver via un ACK qu'ils envoient à $P*$.

— Lorsque le processus émetteur p aura reçu $(n + t)/2$ READY de m, il enverra un ACK à $P*$. Puis, lorsque que plus de $(n + t)/2$ files de ACK auront un ACK de m comme prochain à traiter, p cmb-deliver m.

3.3 Implémentation

3.3.1 Variables locales :

— $waitingAcks$ est une File (FIFO) des ACK non acceptés par processus.
Le but des ACK est d'indiquer la chose suivante : Un processus correct p quelconque envoie Ack(m). Cela indique que tous ses prochains envois auront besoin que m soit cmb-deliver avant d'être considéré. Cela permet que si un byzantin envoie un Ack d'un message qui n'existe pas, ses prochains messages seront ignorés. Tandis que si un processus correct envoi un ACK de m, c'est qu'il a considéré m. Tous ses futurs cmb-broadcast nécessiteront d'avoir cmb-deliver m.

— $isFirstAckOf(message, EmitterOfM)$ est une fonction ou un ensemble qui retourne tous les processus plaçant le ACK relatif au message émit par EmitterOfM en tant que plus ancien ack reçu pour un message encore non délivré par le processus courant.
Ici on considère que cette fonction considère les mises à jour faites lorsque l'on cmb-delivre un message.

3.3.2 Algorithme :

Algorithm: Algorithme principal

```
   /* Phase d'envoi                                                              */
 1 when cmb_broadcast(m) :
 2 |  Send (init(m)) to all the processes
 3 end when
   /* Phase de rebond                                                            */
 4 when p has received one (init(m)) message or more than (n+t)/2 (echo(m, EmitterOfM)) signals
     or (t + 1) (ready(m, EmitterOfM)) signals and waitingAcks[p] are cmb-delivered :
 5 |  Send (echo(m, EmitterOfM)) to all the processes.
 6 end when
   /* Phase de vérification                                                      */
 7 when p has received more than (n+t)/2 (echo(m, EmitterOfM)) or t+1 (ready(m, EmitterOfM))
     signals (including signals received in Phase de rebond) :
 8 |  Send (ready(m, EmitterOfM)) to all the processes
 9 end when
   /* Phase d'approbation récepteur                                              */
10 when non emitter p has received more than (n+t)/2 (ready(m, EmitterOfM)) signals (including
     signals received in Phase de rebond and Phase de vérification) :
11 |  cmb-deliver(message)
12 |  send(ack(m, EmitterOfM))
13 |  if m in waitingAcks then
14 |  |  remove m from waitingAcks
15 |  end if
16 end when
   /* Phase d'approbation émetteur                                               */
17 when emitter p has received more than (n+t)/2 (ready(m, EmitterOfM)) signals (including signals
     received in Phase de rebond and Phase de vérification) :
18 |  send(ack(m, EmitterOfM))
19 |  when length(isFirstAckOf(m, EmitterOfM)) > ((n+t)/2) :
20 |  |  cmb-deliver(message)
21 |  |  if m in waitingAcks then
22 |  |  |  remove m from waitingAcks
23 |  |  end if
24 |  end when
25 end when
   /* Phase de Gestion des Ack                                                   */
26 when receipt of (ack(m, EmitterOfM)) from p :
27 |  waitingAcks[p].enqueue(m)
28 end when
```

4 Lemmes

4.1 Lemme Validité

Validité :

Si un processus correct cmb-deliver un message m de la part d'un processus correct p, alors p a cmb-broadcast le message m.

Preuve :

Si un message m est cmb-deliver, alors le processus cmb-delivrant m doit avoir reçu plus de $(n+t)/2$ ACK. Donc, plus de $(n-t)/2$ processus corrects ont envoyés un ACK. Pour envoyer un ACK, un processus doit recevoir $(n+t)/2$ READY. (l.10 & l.19).

Pour avoir reçu plus de $(n+t)/2$ signaux READY, plus de $(n-t)/2$ processus corrects ont envoyés un signal READY.

Pour envoyer un signal READY, il faut avoir soi-même reçu plus de $(n+t)/2$ signaux ECHO ou $t+1$ signaux READY (l.7). Les premiers signaux READY ne peuvent avoir été envoyé qu'à la réception de signaux ECHO. Il y a donc au moins $t+1$ signaux READY envoyés par la réception de signaux ECHO, et donc au moins 1 bon processus ayant reçu plus de $(n+t)/2$ signaux ECHO.

Ces signaux ECHO ont donc été envoyés par plus de $(n-t)/2$ processus corrects, ce qui ne peut arriver que à la réception de $t+1$ signaux READY, ou plus de $(n+t)/2$ signaux ECHO, ou 1 signal INIT (l.4).

Comme montré plus haut, les premiers signaux READY ne peuvent avoir été envoyés qu'à la réception de signaux ECHO. Les premiers signaux ECHO ne peuvent avoir été envoyés qu'à la réception de 1 signal INIT.

Si l'expéditeur du message m est un processus correct, alors l'envoie des signaux INIT ne peut avoir été fait qu'en appelant cmb-broadcast (l.1).

Si l'expéditeur du message m est un processus correct, il a donc forcément cmb-broadcast le message m.

4.2 Lemme Intégrité

Intégrité :

Chaque processus correct ne peut cmb-deliver qu'une seule fois chaque message.

Preuve :

Pour chaque message, les conditions "when" ne peuvent être appelée qu'une seule fois. L'algorithme ne fait appel à la primitive cmb-deliver qu'une seule fois par processus (l.12 & l.22). Donc, chaque message ne peut être cmb-deliver qu'une seule fois.

4.3 Lemme 'Chrono'

'Définition' :

Si un processus correct p transmet un ACK à un autre processus correct p', alors le ACK qu'il aura transmit sera durant un temps le dernier ACK en attente transmit par p'.

Preuve :

Si un processus correct a émit un ACK cela signifie qu'il a reçu plus de $(n+t)/2$ READYs (l.10, l.17). Si il a reçus plus de $(n+t)/2$ READY, $(n-t)/2$ READY ont été émit par des processus corrects. Ainsi conformément à la ligne 7, tout les corrects recevant $t+1$ READY émettent à leur tour un READY, soit au moins $n-t$ READY émit. Si p est récepteur, alors il finira par cmb-deliver le message associé au ACK (l.11) et nettoiera les piles de ACK du message cmb-deliver (l.14). Si p est émetteur, il devra alors attendre $(n+t)/2$ ACK en tête de file. Hors comme nous venons de l'évoquer, jusqu'à $n-t$ processus corrects peuvent délivrer un ACK (l.12 pour au moins $n-t-1$ processus récepteurs et l.18 pour le processus émetteur). Donc, en tant que processus récepteur des ACK des autres messages, lorsque p recevra $(n+t)/2$ READY des autres messages, cela lui permettra de cmb-deliver ces messages, et donc de retirer de ses piles de ACK les ACK concernant ces messages (l.14). Ainsi p finira par atteindre $(n+t)/2$ ACK en tête de file, et conformément à la ligne 20, cmb-deliver en tant que processus émetteur, puis nettoyer les piles de ACK du message deliver, conformément à la ligne 22.

Donc, si un processus correct reçoit un ACK de la part d'un correct, il le délivrera un jour, ce qui permettra à tout les ACK reçu par ce correct d'être le dernier non considéré par le récepteur du ACK.

4.4 Lemme Progrès Local

Progrès Local :

Si un processus correct cmb-broadcast un message m, alors il finira par cmb-deliver ce message m.

Preuve :

Si un processus p correct a cmb-broadcast un message m, alors il a envoyé un signal INIT à tous les processus (l.2). Quand un processus p envoie un signal, puisque les canaux de communications sont fiables, le signal arrivera un jour à tous les processus. En ce cas, tous les processus corrects enverront à leur tour un signal ECHO à tous les processus (l.5).

Dans cette configuration on a l'envoi de 1 ECHO venant de chacun des $n-t$ corrects et potentiellement d'autres venant des byzantins, Soit au moins $n-t$ ECHO. Comme les canaux sont fiables alors tous les processus recevront un jour $n-t$ ECHO. Plus de $(n+t)/2$ étant inférieur à $n-t$, suffisamment de ECHO seront reçus, ils émettront alors un signal READY (l.8).

Comme les canaux sont fiables, tous les processus recevront un jour les $n-t$ READY envoyés par des processus corrects, ainsi tous les processus corrects entreront en Phase d'approbation récepteur (l.10), sauf le processus émetteur du message m, p, qui entrera en Phase d'approbation émetteur l.17.

Dans cette configuration, tous les récepteurs non byzantins (soit $n-t-1$, car p n'est pas un récepteur) vont cmb-deliver le message m (l.11) et enverront alors à leur tour $n-t-1$ ACK (l.12). Et l'émetteur p enverra à son tour 1 ACK (l.18), soit un total de $n-t$ ACK.

Nous avons ainsi p qui a reçu des ACK provenant de divers processus concernant ou non d'autres messages. Il attendra que plus de $((n+t)/2)$ ACK concernent son message m et que ces derniers soient les plus anciens ACK encore en attente d'être cmb-deliver par p.

Hors d'après le Lemme 'Chrono', si un processus correct, nommé p, transmet un ACK à un autre processus correct, nommé p', alors le ACK qu'il aura transmit sera durant un temps le dernier ACK en attente transmit par p. Ainsi, puisque $((n+t)/2) < n-t$, tous ACK relatifs à m finiront par être les derniers ACK en attente.

Donc p finira par avoir $(n+t)/2$ ACK de m traitables. p cmb-deliver alors m (l.20).

4.5 Lemme 'READY va deliver'

'Définition' :

Si $(n-t)/2$ processus corrects envoient un signal READY concernant un même message m, tous les processus correct finiront par cmb-deliver le message m.

Preuve :

Les canaux de communication étant fiables, si $(n-t)/2$ signaux READY sont envoyés par $(n-t)/2$ processus corrects, alors la totalité des processus finiront par recevoir $(n-t)/2$ signaux READY. Si un processus n'ayant pas déjà envoyé un signal READY reçoit $t+1$ signaux READY, alors il enverra lui-même un signal READY a tous les autres processus (l.7). Or, $(t+1) <= (n+t)/2$. Donc, la totalité des processus finiront par recevoir plus de $(n+t)/2$ signaux READY.

A la réception de plus de $(n+t)/2$ signaux READY, un processus correct n'étant pas l'émetteur va cmb-deliver le message correspondant (l.10). D'après le Lemme de progrès local, si le processus émetteur est correct, il finira par cmb-deliver.

Ainsi, tous les processus corrects auront cmb-deliver.

4.6 Lemme 'Fiabilité'

'Fiabilité' :

Si un processus correct cmb-deliver un message m, alors tous les processus corrects finiront par cmb-deliver ce message m.

Preuve :

Si un processus correct cmb-deliver un message m, c'est que ce processus à reçu plus de $(n+t)/2$ signaux ACK utilisable (l.11 et l.20). Pour avoir reçu plus de $(n+t)/2$ signaux ACK, il faut que plus de $(n-t)/2$ processus correct aient envoyés des signaux ACK. Si un processus correct envoie un message ACK, c'est que ce processus à reçu plus de $(n+t)/2$ signaux READY (l.10 & l.17). Pour avoir reçu plus de $(n+t)/2$ signaux READY, il faut que plus de $(n-t)/2$ processus correct aient envoyés des signaux READY.

D'après le LEMME de READY va deliver, si $(n-t)/2$ processus corrects ont envoyés un signal READY, tous les processus correct vont cmb-deliver le message m.

4.7 Lemme Ordre Mutuel

Ordre Mutuel :

Si un processus correct p cmb-broadcast un message m, et un autre processus correct p' cmb-broadcast un autre message m', alors il est impossible que p cmb-deliver m avant m' et p' cmb-deliver m' avant m.

Preuve :

Soit un processus correct p cmb-broadcastant un message m, et un autre processus p' cmb-broadcastant un autre message m'.

Dans le cas où p cmb-deliver m avant m' et p' cmb-deliver m' avant m, alors p à reçu parmi les plus anciens ACK non traités plus de $(n+t)/2$ concernant m (l.19), donc une majorité des processus corrects ont envoyés un ACK de m avant tout autre ACK.
p' à reçu parmi les plus anciens ACK non traités plus de $(n+t)/2$ concernant m', donc une majorité des processus corrects ont envoyés un ACK de m' avant tout autre ACK.
Donc une majorité de processus corrects ont envoyés un ACK de m avant un ACK de m' et une autre majorité de processus corrects ont envoyés un ACK de m' avant m. On se retrouve avec deux majorités de processus corrects contradictoires, ce qui n'est pas possible.

4.8 Lemme Cohérence causale

Cohérence causale :

Si un processus correct p cmb-deliver m avant de cmb-broadcast m', alors aucun processus correct ne cmb-deliver m' avant m.

Preuve :

Soit p un processus correct cmb-delivrant m avant de cmb-broadcast m'. Si p a cmb-deliver m, c'est qu'il est soit l'émetteur de m, auquel cas p a envoyé un ACK de m (l.18), soit p n'est pas l'émetteur de m, auquel cas p a envoyé un ACK de m (l.12). Donc, p a envoyé un ACK de m avant d'envoyer un message INIT de m'. Les canaux de communication étant FIFO et fiables, tout processus correct recevra ACK de m avant INIT de m'. Si un processus correct reçoit un ACK de m avant un INIT de m' de la part de p, alors il attendra de cmb-deliver m avant de valider la condition l.4 et envoyer un ECHO de m'.

Aucun processus corrects n'enverra de ECHO de m' avant d'avoir cmb-deliver m, donc au maximum t processus peuvent envoyer un ECHO de m'. La condition l.7 ne pourra donc pas être validée, aucun processus correct n'enverra de READY de m'. Au maximum t processus pourront envoyer un READY de m', $t < (n+t)/2$ donc aucun processus correct ne pourra valider la condition (l.10) et p ne validera pas la condition (l.17).

Tous processus correct attendra donc de cmb-deliver m avant de cmb-deliver m'.

4.9 Lemme FIFO

FIFO :

Si un processus correct cmb-deliver un message m de la part d'un processus p avant un message m' de la part du même processus p alors aucun processus correct ne cmb-deliver m' avant m.

Preuve :

Soit p un processus correct cmb-delivrant m avant de cmb-deliver un message m', deux messages cmb-broadcast part le processus p'.

Si m a été cmb-deliver avant m', c'est que plus de $(n+t)/2$ READY de m ont été reçus avant plus de $(n+t)/2$ READY de m'. Donc, plus de $(n-t)/2$ processus corrects ont envoyés un READY de m avant m'. Les canaux étant FIFO et fiables, plus de $(n-t)/2$ processus corrects ont envoyés un READY de m avant m'. Donc, plus de $(n-t)/2$ processus corrects ont validés la condition l.7 pour m avant m', ce qui signifie que plus de $(n+t)/2$ ECHO ont été reçu pour m avant m'. Donc plus de $(n-t)/2$ processus corrects ont validé la condition l.4 et donc reçu un INIT de m avant m'. Les canaux étant FIFO et fiables, INIT de m a été envoyé avant INIT de m'.

Si m et m' ont été cmb-broadcast, m' ne peut pas avoir été cmb-broadcast avant m.

5 Registre Simple Writer - Multiple Reader

5.1 Introduction

La primitive cmb-broadcast nous permet d'implémenter un registre atomique en écrivain unique et lecteurs multiples, dans un contexte Byzantin. Ci-dessous est présenté une implémentation possible d'un registre SWMR, utilisant le cmb-broadcast.

5.2 Algorithme

5.2.1 Déroulé

Le registre Simple Writer Multiple Reader (SWMR), se présente ainsi. Le processus écrivain ne peut-être byzantin, car ce poste est trop crucial pour pouvoir être géré par un processus non conforme à sa spécification.

Ainsi les byzantins ne pourront être que des processus lecteurs, qui n'ont alors aucun impact sur les données spécifiées par l'écrivain.

Le processus écrivain se comportent alors ainsi : Lorsque ce dernier écrit il émet un message X, il attends alors sa délivrance et il est soumis au système.

Les processus lecteurs se comporte ainsi : Lorsque que ce processus veut lire, il émet un message null pour indiquer sa volonté de lire afin de se synchroniser avec l'écrivain, avant d'émettre un autre message null pour se synchroniser avec le reste du système avec son nouvel état. Enfin il retourne la valeur qu'il a enregistré en local. Cette valeur étant mise à jour à chaque réception de message de la part de l'écrivain, le registre local est mis à jour.

5.2.2 Implémentation

Algorithm: Algorithme SWMR

```
   /* Reader and Writer processes                                   */
 1 read() :
   |  /* startReading                                               */
 2 |  cmb-broadcast(null)
 3 |  wait until cmb-deliver()
 4 |  currentVal = val
   |  /* endReading                                                 */
 5 |  cmb-broadcast(null)
 6 |  wait until cmb-deliver()
 7 |  return currentVal
 8 end read
   /* Writer process                                                */
 9 write(m) :
10 |  cmb-broadcast(m)
11 |  wait until cmb-deliver()
12 |  val = m
13 end write
```

6 Conclusion

Dans cet article, nous avons revisité le Reliable Broadcast de Bracha [3], de façon à l'adapter à l'abstraction de mutual broadcast, afin de garantir des propriétés résistantes à un contexte où des processus, appelés Bysantins, peuvent avoir un comportement déviant de l'algorithme et chercher à fausser les communications en s'associant.

L'algorithme original du Reliable Broadcast [3] permet en 3 étapes de communication d'assurer une livraison d'un message à tous les processus correct d'un système distribué, avec une proportion maximale de byzantins inférieure à 1/3 des processus.

Mutual Broadcast est une abstraction permettant de broadcast des messages entre processus, de façon ordonnée.

Notre implémentation en Byzantins de CMB Broadcast utilise et modifie les 3 étapes du Reliable Broadcast de Bracha [3], et ajoute une nouvelle étape afin de respecter un ordre mutuel.

Nous avons ainsi implémenté un Causal Mutual Byzantine Broadcast en 4 étapes, résistant à un contexte Byzantin.

En conclusion, on a réussi à prouver que Mutual Broadcast est implémentable avec une tolérance d'un maximum de processus Byzantins de moins d'un tier du nombre total de processus.

Remerciements

Ce travail a été partiellement supporté par les projets ANR ByBloS (ANR-20-CE25-0002-01) et PriCLeSS (ANR-10-LABX-07-81).

7 Bibliographie

[1] Lamport, L., Shostak, R. and Pease, M. *The Byzantine Generals Problem* ACM TOPLAS (1982)

[2] Gilbert S. and Lynch N. *Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services.* ACM SIGACT News (2002)

[3] Bracha G., Asynchronous Byzantine agreement protocols. Information Computation,75(2) :130-143 (1987)

[4] Déprés M., Mostéfaoui A., Perrin M., Raynal M. *Send/Receive Patterns Versus Read/Write Patterns in Crash-Prone Asynchronous Distributed Systems.* DISC (2023)