Registre atomique préservant la vie privée tolérant aux byzantins

Quentin Gomes dos Reis, Vincent Kowalski, Rodrigue Meunier, Matthieu Perrin, Gabriel Pouplin

Registre atomique préservant la vie privée tolérant aux byzantins

Gomes Dos Reis Quentin
Nantes Université, Nantes, France
quentin.gomes-dos-reis@etu.univ-nantes.fr

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

Meunier Rodrigue
Nantes Université, Nantes, France
rodrigue.meunier@etu.univ-nantes.fr

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

Pouplin Gabriel
Nantes Université, Nantes, France
gabriel.pouplin@etu.univ-nantes.fr

Résumé

Ce rapport traite de la sécurisation et de la confidentialité des données personnelles stockées dans des systèmes répartis. Il explore les défis algorithmiques nécessaires pour restreindre l'accès aux données à un ensemble de lecteurs autorisés tout en garantissant leur disponibilité. Pour résoudre ce défi, ce rapport propose un algorithme de registre lire/écrire partagé linéarisable qui résiste à un maximum de $t < \frac{n}{7}$ pannes byzantines, en exploitant l'algorithme d'Adi Shamir, permettant la division et la distribution des données entre entités de confiance.

1 Introduction

Contexte et motivation. Nous possédons tous des données personnelles, en particulier des données médicales. Ces données ne doivent pas être diffusées à tout le monde pour des raisons de sécurité et de vie privée. Il convient tout de même de les partager à un certain nombre de personnes autorisées. Par exemple, le propriétaire de ces données, ainsi que le personnel médical. Toutes ces personnes peuvent avoir besoin de lire ces données médicales tandis que le propriétaire doit être en capacité de modifier ces données. Afin d'éviter une architecture client serveur et les problèmes liés à celles-ci, notamment l'absence de disponibilité des données en cas de panne du serveur. C'est pourquoi nous avons choisi l'approche des systèmes distribués. Ces systèmes répartis sont des collections de processus communiquants par l'envoi et la réception de messages utilisés pour résoudre un problème commun. Dans l'exemple ci-dessus, la structure de données nécessaire pour partager des données est une case mémoire dans laquelle on peut écrire et lire (appelée registre). Idéalement, le registre partagé devrait se comporter comme s'il n'y avait qu'une seule copie physique partagée par tous les processus. Une telle abstraction est appelée linéarisabilité.
En partageant le registre, des problèmes de vie privée apparaissent. En effet, si le registre est partagé par tous les processus, chacun peut lire et écrire. Dans notre exemple, il ne faut pas que d'autres personnes, autres que les médecins et les propriétaires, puissent lire les données médicales.
L'asynchronisme et les plantages ne sont pas les seuls problèmes à résoudre dans les systèmes distribués. Ils peuvent également être sujet à des problèmes de sécurité. Les fautes byzantines ont été introduites pour la première fois par Lamport [2] en 1982 : un participant byzantin n'exécute pas le code attendu. Cela peut se produire lorsqu'il est malveillant ou simplement lorsqu'ils exécutent une version obsolète ou corrompue de l'application.
Des algorithmes de registre Lire/Écrire tolérants des processus byzantins ont également été conçus [3, 4]. Reposant sur la réplication complète, ces algorithmes ne sont pas du tout adaptés à la protection de la vie

privée. En effet, les processus byzantins doivent stocker la valeur du registre mais par conséquent, ils peuvent aussi lire le contenu du registre, et il est difficile de gérer les droits d'accès à ces algorithmes.

Travaux relatifs. Dans le domaine des registres atomiques Lire/Écrire partagés tolérant aux fautes byzantines, une multitude de travaux ont été proposés ces dernières années. Comme en 2017, dans le papier [4] de Mostéfaoui A., Petrolia M., Raynal M. et Jard C. qui aborde cette même problématique en proposant un système de mémoire atomique Lire/Écrire tolérant aux fautes byzantines. La limite admise est strictement inférieure à $\frac{n}{3}$ processus byzantins, limite précédemment prouvée comme nécessaire et suffisante dans le papier [8] datant de 2014, proposé par Imbs D., Rajsbaum S., Raynal M. et Stainer J.. Cependant, bien que ces deux papiers définissent une limite optimale de processus byzantins acceptables dans un système distribué, les algorithmes proposés n'incluent pas de solution pour sécuriser les valeurs partagées. De nombreux autres papiers traitent de sujets similaires, mais ne se centrent pas sur la sécurisation des valeurs partagées.

Problématique. Comment partager de l'information dans un système réparti tolérant les fautes byzantines en respectant la vie privée ?

Approche. Afin de garantir la vie privée des informations partagées, nous utiliserons l'algorithme d'Adi Shamir, exposé en 1979 [6], il fournit un mécanisme permettant de partager une information entre plusieurs entités de confiance. Cet algorithme permet de diviser les données en plusieurs sous-parties (shards) pour ensuite distribuer les parties aux autres processus. D'autres algorithmes de partage d'informations existent, mais leur complexité est plus élevée, notamment la cryptographie qui se base sur la puissance de calcul. Pour ce faire, un polynôme P est généré aléatoirement, avec $P(0)$, l'information à transmettre et à chaque entité du système est transmise $P(x)$ ($x \in [1, 2, ..., n]$ et n, le nombre d'entités du système). Alors, pour reconstituer l'information initiale, un lecteur devra collecter les parties d'un minimum d'entités afin d'avoir suffisamment de parties de l'information pour pouvoir réaliser l'interpolation de Lagrange et ainsi reformer le polynôme et retrouver l'information avec $P(0)$. La collaboration étant obligatoire pour récupérer l'information partagée, cela signifie qu'une entité ne peut pas récupérer les informations partagées sans collaborer avec les autres entités. Ainsi, si un processus exécute une lecture alors qu'il ne possède pas les droits de lecture, il ne recevra pas les sous-parties des processus corrects à la suite d'une demande. Cela garantit la vie privée, car il ne pourra pas retrouver l'information partagée initialement. Pour pouvoir garantir la linéarisabilité de notre registre Lire/Écrire, nous emploierons l'approche de consensus distribué tel qu'elle est exposée dans le papier [1] de 1990, écrit par Attiya H., Bar-Noy A. et Dolev D.. Ainsi, en employant cette même approche, la mise à jour de nos parties restera cohérente. Ce même partage d'information à l'aide de l'algorithme de Shamir ainsi que la présence d'entités byzantines nous forcera à mettre en œuvre des protections supplémentaires afin de garantir la linéarisabilité de chaque opération effectuée sur notre registre et rendre ce dernier atomique et résistant aux byzantins.

Contributions.

- Un algorithme qui lorsqu'il y a moins de $\frac{n}{7}$ processus byzantin permet d'implémenter un registre Lire/Écrire linéarisable tolérant aux fautes byzantines.
- Preuves de l'algorithme :
  - Terminaison
  - Linéarisabilité byzantine
  - Sécurité

Organisation. Le papier est composé de 6 sections. La section 2 présente les modèles utilisés. La section 3 ajoute des précisions sur le problème. La section 4 se consacre sur l'algorithme, et la section 5 prouve son exactitude. Enfin, la section 6 conclut l'article.

2 Modèles

Les entités : Le modèle est composé d'un ensemble de n processus séquentiels. Ces processus seront désignés comme ceci : p_1, p_2, ..., p_n. Ces processus sont asynchrones ce qui implique qu'ils vont tous à leur rythme propre. Aucun processus ne peut savoir où en est un autre dans son exécution du code.

Le modèle de communication : Les processus communiquent en envoyant et en recevant des messages par des canaux de communication bidirectionnels. Le réseau de communication est complet : tout processus p_i peut émettre un message à n'importe lequel des processus du système y compris lui-même. De plus, si un processus reçoit un message, il est certain de l'identité de l'émetteur de ce message. Les canaux sont fiables, ce qui implique qu'il n'y a pas de perte ou de corruption de message. Les canaux sont aussi asynchrones, ce qui implique que le temps de transmission des messages est fini, mais n'est pas borné supérieurement. Toutefois, si un processus p_i et un processus p_k envoient tous les deux un message à p_j, il n'y a aucune garantie dans l'ordre de réception de ces messages.

FIFO : Nous ajoutons en hypothèse supplémentaire que les canaux de communication sont FIFO (First In, First Out) entre processus corrects ce qui signifie que si un processus p_i envoie deux messages a puis b à un processus p_j, p_j recevra ces deux messages dans l'ordre de leur émission. Cela n'ajoute pas de complexité particulière à l'algorithme.

Les byzantins : L'algorithme tolère la présence de processus byzantin. Les processus byzantins ont un comportement qui ne correspond pas à l'algorithme qui leur est donné. Par exemple, ils peuvent tomber en panne, ne pas envoyer ou recevoir des messages, envoyer des messages arbitraires, exécuter de façon arbitraire du code. Un processus byzantin qui est censé envoyer un message m à tous les processus peut envoyer un message m_1 à un sous-ensemble de processus et un message m_2 à un autre sous-ensemble de processus et pas de messages aux processus restants. Cependant, ils ne peuvent pas altérer le contenu des messages émis par des processus non-byzantins. Plus généralement, les processus byzantins sont libres de produire toutes sortes d'actions pouvant nuire au bon déroulement de l'algorithme. On suppose que le nombre de byzantins dans une exécution est borné supérieurement par $t < \frac{n}{7}$. Un processus n'ayant pas un comportement byzantin est aussi appelé un processus correct.

Notation : L'acronyme $\mathcal{BAMP}_{n,t}[t < \frac{n}{7}]$ désigne le modèle Byzantine Asynchronous Message Passing où t processus peuvent avoir un comportement byzantin et la communication se fait à l'aide de registre Lire/Écrire (Read/Write).

3 Précision sur le problème à résoudre

Registres Lire/Écrire(R/W) : Ce registre permet deux opérations : une opération d'écriture que seul le processus écrivain peut utiliser qui ne peut rien renvoyer et une opération lire accessible à certains processus qui renvoient la dernière valeur écrite ou la valeur initial $\perp$ si aucune valeur n'a été écrite.

Vivacité : Soit p_i un processus correct.

- Chaque appel à la procédure WRITE() par p_i termine.
- Chaque appel à la procédure READ() par p_i termine.

Sécurité : Soit p_i un processus correct qui possède les droits de lecture.

- Chaque appel à la procédure READ() renvoie la valeur de la dernière écriture ou rien si aucune écriture correcte n'a eu lieu.
- Les processus n'ayant pas les droits de lecture ne doivent pas être capable de lire le contenu du registre.

Sûreté : L'article de Shir Cohen et Idit Keidar [7] décrit la linéarisabilité comme ceci :

Définition 1 (Linéarisabilité byzantine)**.** *Une histoire H est linéarisable par rapport à un objet O s'il existe une histoire séquentielle H' (appelée une linéarisation de H) telle que (1) après suppression de certaines opérations de H et en complétant les autres en ajoutant des réponses correspondantes, il contient les mêmes appels et réponses que H, (2) si une opération o retourne avant qu'une opération o' ne commence dans H alors o apparaît avant o' dans H', et (3) H' satisfait la spécification séquentielle de O.*
Une histoire H est linéarisable byzantine par rapport à un objet O s'il existe une histoire H' linéarisable par rapport à O, tel que $H'|_{correct} = H|_{correct}$ (où $H|_{correct}$ désigne l'histoire où seules les opérations effectuées par des processus corrects sont prises en compte). On dit qu'un objet est linéarisable byzantin, ou simplement linéarisable, si toutes ses exécutions sont linéarisables byzantines.

4 Algorithme

4.1 Présentation de l'algorithme

L'algorithme 1 présente le code suivi par un processus correct p_i dans le modèle $\mathcal{BAMP}_{n,t}[t < \frac{n}{7}]$.

Le processus écrivain commence par créer un polynôme afin de dissimuler son information puis il va distribuer des shards réalisés à l'aide de ce polynôme à chacun des processus du système avant d'attendre que suffisamment de processus ont reçu ses shards afin de permettre aux processus lecteurs de pouvoir retrouver le polynôme initial. Pour cela, les processus du système communiqueront entre eux afin de savoir si le processus écrivain a émis suffisamment de shards pour retrouver l'information qu'il cherche à dissimuler et si c'est le cas alors chaque processus émet le message ACK afin de signaler à l'écrivain qu'il est possible de retrouver l'information initiale car suffisamment de processus ont validé l'écriture. Lorsqu'un processus valide une écriture, cela signifie qu'il a reçu la shard émanant du processus écrivain et qu'il a envoyé le message ACK.
Un processus lecteur commence sa lecture du registre, il envoie une demande à tous les processus pour que chacun transmette ses shards. Après avoir reçu suffisamment de réponses de la part des processus, il exécute une recherche de reconstruction du polynôme P. À l'aide d'un système de numéro de séquence, on définit la version la plus récente comme les shards, avec pour numéro de séquence sn, qui sont en nombre suffisant pour pouvoir retrouver P.
Enfin, s'il trouve un polynôme valide, il renvoie la valeur de $P(0)$ sinon il renvoie 0 comme valeur initiale au registre.

Les variables locales. Chaque processus p_i s'occupe de variables locales indiquées par l'indice i dont la portée est la totalité de l'algorithme.

- $shards_i[]$: Il s'agit d'un tableau local de taille infinie contenant la liste des sous-parties transmises par le processus écrivain rangées dans l'ordre de leur réception. Si l'écrivain est correct alors, les valeurs stockées dans ce tableau pour le processus p_i sont soit égales à $P(i)$ suite à une écriture, soit à $\perp$.
- $acknoledged_i$: Un entier représentant le nombre de sous-parties validées par p_i
- $reg_i[][]$: Il s'agit d'un tableau de taille n contenant des tableaux de taille infinie. Cette variable enregistre les sous-parties validées (contenu dans $shard_j$) et transmises par les différents processus p_j lors de la demande de lecture. Plus précisément, $\forall j, k$, $reg_i[j][k]$ initialisé à $\perp$ contient $P(j)$ de l'écriture k

$\mathbb{Z}_t[X]$: l'ensemble des polynômes de degré t et de coefficient $\mathbb{Z}$

procedure `write`(v) *invoked by* p_w **is**

1 let $P \in \mathbb{Z}_t[X] : P(0) = v$;
2 **for** j **from** 1 **to** n **do**
3 **send** SHARE$(P(j), sn_w)$ **to** p_j;
4 **wait until** (p_w has received at least $n - t$ messages ACK(sn_w));
5 $sn_w \leftarrow sn_w + 1$;

procedure `read`() *invoked by any* p_i *with reading rights* **is**

6 $reg_i \leftarrow [[], ..., []]$;
7 **send** COLLECT(rsn_i) **to** all processes;
8 **wait until** (p_i has received at least $n - t$ messages SUPPLY$(_, rsn_i)$);
9 $rsn_i \leftarrow rsn_i + 1$;
10 **for** k **from** $\max_j |reg_i[j]|$ **to** 1 **do**
11 **if** $\exists P \in \mathbb{Z}_t[X] : |\{j : P(j) = reg_i[j][k]\}| > 2t$ **then**
12 **send** CONFIRM(k) **to** all processes;
13 **wait until** (p_i has received at least $n - 2t$ messages RATIFY(k));
14 **return** $P(0)$;
15 **return** 0;

when receive SHARE$(shard, sn)$ **from** p_j **:**

16 $shards_i[sn] \leftarrow shard$;
17 **send** ECHO(sn) **to** all processes;

when receive ECHO(sn) **from** $n - t$ *process or* **receive** READY(sn) **from** $5t + 1$ *process* **:**

18 **send** READY(sn) **to** all processes;

when receive COLLECT(rsn) **from** p_j *with reading rights* **:**

19 **send** SUPPLY$(shards_i[0..acknoledged_i], rsn)$ **to** p_j

when receive SUPPLY$(v[], rsn)$ **from** p_j **:**

20 **if** $rsn = rsn_i$ **then** $reg_i[j] \leftarrow v$;

when receive READY(sn) **from** $6t + 1$ *process* **:**

21 $acknoledged_i \leftarrow sn$;
22 **send** ACK(sn) **to** p_w;

when receive CONFIRM(k) **from** p_j **:**

23 **wait until** $(acknoledged_i \geq k)$;
24 **send** RATIFY(k) **to** p_j;

Algorithm 1: Implémentation d'un registre linéarisable respectueux de la vie privée dans le modèle $\mathcal{BAMP}_{n,t}[t < \frac{n}{7}]$

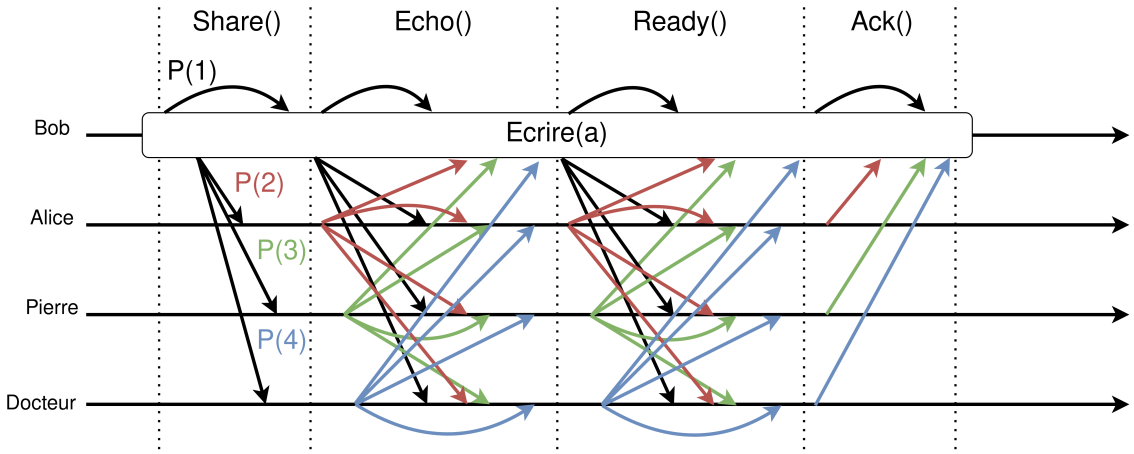

FIGURE 1 – Explication schématisée du processus de l'écriture

4.2 Les messages

Cet algorithme utilise 8 types de messages différents :

— SHARE($shard, sn$) : Transmets la *shard* de l'écrivain vers un processus.

— ECHO(sn) : Informe les autres processus du système qu'un processus a reçu une shard provenant de l'écrivain.

— READY(sn) : Informe les autres processus que le processus courant à reçu assez de messages ECHO(sn) pour pouvoir envoyer le message ACK(sn).

— ACK(sn) : Informe l'écrivain que la valeur qu'il a écrite peut être lue.

— COLLECT(rsn) : Demande les shards reçus par chaque processus.

— SUPPLY($v[], rsn$) : Envoie la liste des shards reçus au processus lecteur.

— CONFIRM(k) : Demande au processus s'ils possèdent au moins la valeur lue par le lecteur

— RATIFY(k) : Les processus envoient ce message s'ils ont au moins la valeur lue.

4.3 Déroulement de l'algorithme

4.3.1 Écriture

Le processus écrivain distribue à tous les processus du système, y compris lui-même, le message SHARE. Il permet de communiquer la shard correspondante à chaque processus et également le numéro de séquence (noté sn) de cette écriture. Dans le système, il y a n processus et chaque processus possède un numéro qui lui est attribué et grâce à ce principe, le processus p_k recevra la shard k correspondante à la valeur $P(k)$ où P est le polynôme généré par le processus écrivain. À la réception du message SHARE, les processus enregistrent la shard reçue dans leur variable $shards_i[sn]$ puis ils émettent le message ECHO(sn) afin d'informer tous processus de la réception de la shard. Lorsqu'un processus reçoit $n-t$ messages ECHO(sn) ou $5t+1$ messages READY(sn), il émet le message READY(sn) à la totalité des processus du système. Lorsqu'un processus correct p_i reçoit $6t+1$ messages READY(sn), il enregistre sn dans sa variable $acknoledged_i$ puis, il envoie le message ACK(sn). Le processus écrivain peut alors incrémenter son numéro de séquence.

4.3.2 Lecture

Le processus lecteur p_w commence par réinitialiser la variable reg_r. Ensuite, p_r envoie le message COLLECT(rsn_r) avec rsn_r le compteur de lecture effectué par le processus p_r. Ce compteur permet qu'il n'y ait

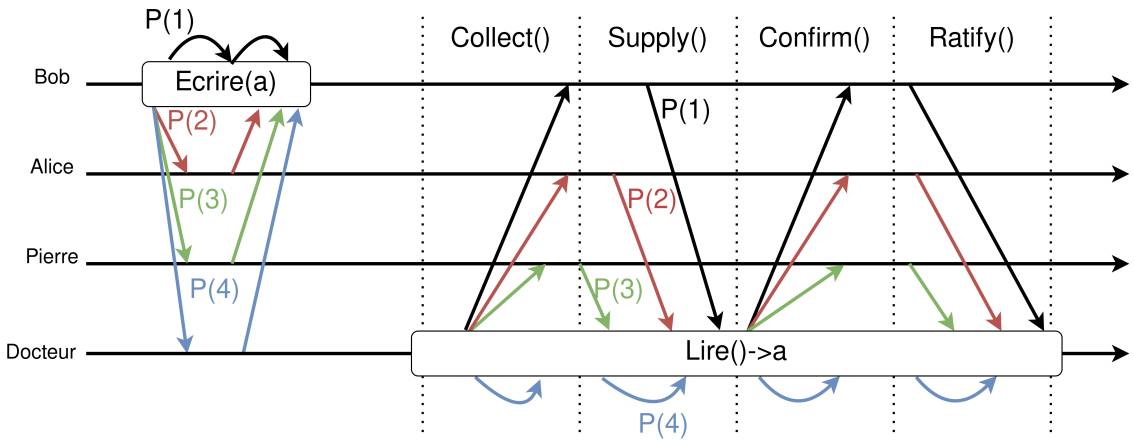

FIGURE 2 – Explication schématisée du processus de la lecture

pas de conflit entre les réceptions de ce message issu de différentes lectures. Ensuite p_w attend de recevoir $n-t$ messages SUPPLY$(shards_i, rsn_r)$. Lorsqu'un processus p_i reçoit le message COLLECT(rsn) il s'assure que p_r a les droits de lecture avant de de lui transmettre toutes les shards validées par p_i dans le message SUPPLY$(shards_i[0..acknoledged_i], rsn)$. À chaque message SUPPLY$(shards_i, rsn)$ envoyé par p_i reçu, p_r vérifie que le paramètre rsn soit égal à sa variable rsn_r (il vérifie que la réponse correspond à sa lecture courante) puis il enregistre $shard_i$ dans sa variable $reg_r[i]$. Une fois que le processus p_r a reçu $n-t$ message SUPPLY, il va ensuite incrémenter sa variable rsn_r puis en utilisant la variable k qui varie entre la taille la plus grande parmi les registres reçus jusqu'à 1, p_r essaie de retrouver le polynôme créer par p_w en utilisant l'interpolation de Lagrange avec $2t+1$ points. Si p_r ne trouve pas de polynôme alors il renvoie 0. Sinon il utilise le k pour lequel p_r a trouvé un polynôme afin d'envoyer le message CONFIRM(k) à tous les processus. Ce message permet au lecteur de s'assurer que les lectures futures liront au moins la valeur numéro k. Ensuite il attend de recevoir $n-2t$ messages RATIFY(k). Lorsque les processus p_i reçoivent le message CONFIRM(k) ils attendent que leur variable $acknoledged_i$ soit supérieure ou égale à k puis ils envoient le message RATIFY(k) à p_r. Lorsque p_r a reçu assez de messages RATIFY, il peut terminer et renvoie $P(0)$ pour retrouver la valeur initiale écrite par p_w.

5 Preuves

Définition 2 (Validation numéro de séquence)**.** *Soit $sn \in \mathbb{N}$, et p_i un processus correct. On dit que p_i* valide *sn quand p_i envoie un message* ACK(sn) *et $shards_i[sn] \neq \perp$.*

Définition 3 (Horodatage)**.** *Pour chaque opération o, nous définissons un horodatage $hd(o)$ de o comme suit. Si o est une écriture par p_w, alors $hd(o) = sn_w$ à la fin de la ligne 5. Si o est une lecture par p_i alors $hd(o) = k$ à la ligne 14 ou $hd(o) = 0$ à la ligne 15. En d'autres termes, $hd(o)$ est équivalent au numéro de séquence qui est lu ou écrit par o.*

Lemme 1. *Si un processus correct envoie le message* ACK*(sn) alors tous les processus corrects envoient le message* ACK*(sn).*

Démonstration. Supposons qu'un processus correct p_i envoie un message ACK(sn) à la ligne 22. Il a reçu au moins $6t+1$ messages READY(sn), dont au moins $5t+1$ ont été émis par des corrects. Donc tous les corrects recevront ces messages et émettront un message READY(sn) à la ligne 18. Comme $n-t \geq 6t+1$, au moins $6t+1$ messages READY(sn) sont envoyés par des corrects, tous les corrects envoient un message ACK(sn) à la ligne 22. □

Lemme 2 (Terminaison des écritures)**.** *Tout appel par un processus correct à l'opération write(v) termine.*

Démonstration. Supposons que p_w, un processus correct fasse appel à l'opération write(v) et que celle-ci ne termine pas. Les lignes 1-3 et 5 terminent par définition. Alors si l'exécution de write(v) par p_w ne termine pas, cela signifie que p_w a reçu moins de $n-t$ messages ACK(sn). À la ligne 3, p_w enverra des messages SHARE(sn) à tous les processus puisqu'il est correct. Ce qui implique que les $n-t$ processus corrects recevront le message SHARE(sn) et transmettront le message ECHO(sn) ligne 17. Donc au moins $t+1$ processus corrects recevra $n-t$ message ECHO(sn) et transmettront donc le message READY(sn) à la ligne 18. Tous les autres processus corrects finiront donc par transmettre READY(sn). Cela implique qu'au moins 1 processus correct recevra $6t+1$ messages READY(sn) et il enverra donc le message ACK(sn) ligne 22. Or, le lemme 1 nous dit que si un processus correct émet le message ACK(sn), alors tous les processus corrects finiront par l'envoyer. Le processus p_w recevra donc nécessairement $n-t$ message ACK(sn) et il pourra donc terminer son exécution de l'opération write(v). Contradiction avec la supposition de départ ce qui implique que la propriété est vraie. □

Lemme 3 (Terminaison des lectures)**.** *Tout appel par un processus correct à l'opération read() termine.*

Démonstration. Supposons que p_i, un processus correct, fasse appel à l'opération read() et que celle-ci ne se termine pas. Les lignes 6-7, 9-12 et 14-15 terminent toutes par définition. Donc, si l'exécution par p_i de read() ne termine pas c'est qu'il n'a pas reçu $n-t$ messages SUPPLY(-,rsn_i) ligne 8 ou alors $n-2t$ messages RATIFY(k) ligne 13.
S'il n'a pas reçu $n-t$ messages SUPPLY(-,rsn_i) c'est qu'au moins un processus correct n'a pas reçu le message COLLECT et n'a pas exécuté la ligne 19. Or, à la ligne 7, p_i a émis un COLLECT à tous les processus et comme les canaux sont sûrs, chaque processus correct a reçu le message. Alors chaque processus correct a envoyé SUPPLY à p_i et donc a exécuté la ligne 19, contradiction.
S'il n'a pas reçu $n-2t$ messages RATIFY, sachant qu'il a envoyé le message CONFIRM(k) à tous les processus alors c'est qu'au moins un processus correct est bloqué à la ligne 23. Sachant que si le processus p_i a envoyé le message CONFIRM(k) à la ligne 12 à tous les processus cela implique que d'après la ligne 11 nous avons forcément reçu $2t+1$ message SUPPLY qui signifie qu'au moins $2t+1$ processus ont validé k, donc au moins un processus correct a émis le message ACK(k). Or d'après le lemme 1, si un processus correct envoie le message ACK(k) alors tous les processus corrects enverront ACK(k). Donc $\forall p_j$ un processus correct, p_j aura $acknoledged_j \geq k$. Donc les p_j pourront envoyer le message RATIFY(k). Le processus p_i finira par recevoir $n-2t$ messages RATIFY(k), p_i ne restera donc pas bloqué à la ligne 13.
Les deux cas sont en contradiction avec l'hypothèse de départ.

□

Théorème 1 (Terminaison de l'algorithme)**.** *Toute utilisation de l'algorithme par un processus correct termine.*

Démonstration. Les lemmes 2 et 3 nous montrent que les appels aux procédures read() et write(v) terminent. Cela implique que l'algorithme termine. □

Théorème 2 (Sécurité)**.** *Si un processus byzantin essaye de lire le contenu d'une écriture, il sera empêché.*

Démonstration. Si un processus outrepasse la sécurité en exécutant une opération de lecture, quelle que soit sa méthode, il va avoir besoin de shards et nécessairement d'envoyer le message COLLECT pour recevoir des shards. Or, tous les processus corrects ignorent les messages COLLECT reçus de processus n'ayant pas les droits de lecture. Ainsi, le processus byzantin n'aura pas $2t+1$ shards et ne pourra pas retrouver la valeur de façon certaine en exécutant la ligne 11. □

Lemme 4. *Supposons l'écrivain correct. Si au moins $4t+1$ processus corrects ont validé le numéro de séquence $hd(o_1)$ avant qu'une lecture o_2 ne commence, alors $hd(o_2) \geq hd(o_1)$.*

Démonstration. Supposons l'écrivain correct, qu'au moins $n-3t$ processus corrects ont validé le numéro de séquence $hd(o_1)$ au moment où un processus p_i commence une lecture o_2. Lorsque p_i exécute la ligne 8, il reçoit $n-4t$ messages provenant de processus corrects p_j de la forme SUPPLY(v_j, rsn_i) qui ont déjà validé $hd(o_1)$. Notons Π_S l'ensemble de ces $n-4t$ processus.

Prouvons que, pour tout $p_j \in \Pi_S$, on a $|v_j| \geq a$. Soit $p_j \in \Pi_S$. Comme p_j a déjà validé $hd(o_1)$ (ligne 22), il a exécuté la ligne 21 précédemment. Pour pouvoir exécuter ces lignes, p_j doit avoir réceptionné $6t+1$ fois le message READY$(hd(o_1))$ provenant de processus différents, dont au moins $5t+1$ processus corrects. Cela implique qu'au moins $5t+1$ processus corrects ont reçu le message ECHO$(hd(o_1))$ de $n-t$ processus différents où ils ont reçu le message READY$(hd(o_1))$ de $5t+1$ processus différent pour pouvoir émettre READY$(hd(o_1))$ à la ligne 18. Cela implique pour qu'au moins $4t+1$ processus corrects émettent le message ECHO$(hd(o_1))$ ligne 17. Pour cela, ils ont dû recevoir le message ECHO$(hd(o_1))$ provenant de $n-t$ processus différents, dont au moins $n-2t$ processus corrects. Si ces $n-2t$ processus corrects ont émis ECHO$(hd(o_1))$ ligne 17, ils ont nécessairement reçu une shard provenant du processus écrivain et donc ils ont enregistré cette shard à la ligne 16. Donc $|v_j| \geq hd(o_1)$ d'après la ligne 19.

Lorsque p_i exécutera la ligne 11, il sera en mesure de retrouver la valeur de l'écriture ayant pour numéro de séquence $hd(o_1)$. En effet, p_i possède $n-5t$ messages provenant de processus corrects p_j de la forme SUPPLY(v_j, rsn_i) qui ont déjà validé $hd(o_1)$. Il faut que p_i parviennent à retrouver un polynôme avec $2t+1$ données. Or, si p_i possède $n-5t$ messages issu de processus corrects, il en a donc au moins $2t+1$ car $t < \frac{n}{7}$. L'écrivain étant correct, il a donc créé un polynôme ligne 1 puis il a envoyé une valeur associée à chaque processus à l'aide de ce polynôme dans le message SHARE$(P(j), sn_w)$. Comme montré précédemment, tous les p_j ont stocké cette valeur puis l'on transmit à p_i. Lorsque p_i terminera son opération de lecture (voir lemme 3), il renverra ligne 14 la valeur écrite par le processus écrivain.

□

Lemme 5. *Si l'écrivain est correct et qu'au moins $n-2t$ processus corrects ont envoyé le message* ACK*(sn), alors au moins $4t+1$ processus corrects ont validé sn.*

Démonstration. Soit p_i un processus correct qui termine une écriture ayant pour numéro de séquence sn. Le processus p_i a donc reçu $n-t$ messages ACK(sn) ligne 4. Cela implique que tous les processus corrects p_j ont envoyé le message ACK(sn) ligne 22 et ils ont donc stocké sn dans la variable $acknoledged_j$ à la ligne 21. Afin de pouvoir exécuter ces lignes, les processus p_j ont dû recevoir des messages READY(sn) provenant de $6t+1$ processus différents, dont au moins $5t+1$ corrects. Pour qu'un processus correct émette le message READY(sn) ligne 18, il doit avoir reçu $n-t$ message ECHO(sn) ou $5t+1$ messages READY(sn). Au minimum, il y a donc $4t+1$ processus corrects qui émettent le message READY(sn) après avoir reçu les messages ECHO(sn). Pour qu'un processus correct p_j envoie le message ECHO(sn) ligne 17, il doit avoir reçu le message SHARE$(shard, sn)$. Le processus p_i étant correct, il transmettra un message SHARE à la totalité des processus du système à la ligne 3. Donc $n-t$ processus ont envoyé le message ECHO(sn) dont au minimum $n-2t$ corrects. Tous les processus corrects seront en capacité d'envoyer le message READY(sn). Les processus corrects recevront donc au moins $6t+1$ message READY(sn). Ils seront donc en capacité de transmettre le message ACK(sn). Soit $Q1$ l'ensemble des processus ayant envoyé le message ECHO(sn), et soit $Q2$ l'ensemble des processus ayant envoyé le message ACK(sn). Nous avons $|Q1| \geq n-t$ et $|Q2| \geq n-t$. On a :

$$\begin{array}{rcll} |Q1 \cap Q2| & = & |Q1| + |Q2| - |Q1 \cup Q2| & \\ & \geq & n-t+n-t-n & \textbf{car } |Q1 \cup Q2| \leq n \\ & \geq & n-2t & \\ & > & 5t & \textbf{car } n > 7t \\ |Q1 \cap Q2| & \geq & 5t+1. & \end{array}$$

En particulier, $Q1 \cap Q2$ contient au moins $|Q1 \cap Q2| - t \geq 4t+1$ processus corrects p_j qui ont $shard_j[sn] \neq \perp$ et qui ont envoyé le message ACK(sn), soit la définition de validé.

□

Lemme 6 (écriture après lecture)**.** *Soient 2 processus correct p_i et p_j, si p_j termine la lecture o_1 avant que p_i ne commence une écriture o_2 alors $hd(o_1) < hd(o_2)$.*

Démonstration. Soit un processus correct p_j qui effectue une opération de lecture o_1 et un processus correct p_i qui effectue une opération d'écriture o_2 tel que $hd(o_1) < hd(o_2)$ et supposons que p_j retourne la valeur associée à l'écriture o_2.

Pour que p_j puisse retourner cette valeur, il faut qu'à la ligne 11 il ait réussi à retrouver un polynôme avec $2t+1$ shards de taille supérieure ou égale à $hd(o_2)$ émis par des processus corrects p_k. Les informations contenues dans $reg_j[k]$ étant différentes de $\perp$, on a $reg_j[k][hd(o_2)] = shards_k\,[hd(o_2)]$, qui a été édité à la ligne 16 quand p_k a reçu le message SHARE$(shard, hd(o_2))$ envoyé par le processus écrivain à la ligne 3. Cela implique donc que l'écriture $hd(o_2)$ a commencé si o_1 renvoie la valeur de l'écriture o_2 ce qui implique que $hd(o_1) \geq hd(o_2)$ par définition de hd.

□

Lemme 7 (lecture après écriture)**.** *Soit un processus correct p_i qui termine une écriture o_1 avant qu'un processus correct p_j ne commence une lecture o_2 alors $hd(o_2) \geq hd(o_1)$.*

Démonstration. Soient 2 processus correct p_i et p_j, si p_i termine une opération o_1 avant que p_j ne commence la lecture o_2. Pour que p_i termine son écriture, il a dû attendre de recevoir $n-t$ messages ACK(sn) (ligne 4) provenant de processus différents, dont au moins $n-2t$ processus corrects. Donc d'après le lemme 5, il y a au moins $4t+1$ processus corrects qui ont validé le numéro de séquence $hd(a)$. Donc d'après que le lemme 4, $hd(o_2) \geq hd(o_1)$. □

Lemme 8 (lecture après lecture)**.** *Soit p_i un processus correct qui termine une opération de lecture o_1 avant qu'un processus correct p_j commence une opération de lecture o_2. Alors $hd(o_1) \leq hd(o_2)$.*

Démonstration. Soient 2 processus corrects p_i et p_j, si p_l termine son opération de lecture o_1 ce qui implique qu'au moins $4t+1$ processus corrects ont validé l'écriture avec $hd(o_1)$ d'après le lemme 5. Si maintenant, p_j commence une lecture o_2, il a alors reçu les shards d'au moins $n-t$ processus et a déterminé ligne 10 que $hd(o_2)$ était la version la plus récente validée par le plus grand nombre. En effet, toujours d'après le lemme 4 $n-t$ processus garantissent qu'au moins $3t+1$ processus ont déjà validé $hd(o_1)$. Ainsi, en retirant t, les byzantins, $3t+1-t = 2t+1$ on peut encore retrouver la valeur, car au moins $2t+1$ shards sont nécessaires. Sachant qu'on a au moins $4t+1$ processus ont validé l'écriture o_1. Alors, $hd(o_2)$ est soit égal à $hd(o_1)$ soit plus grand, car ce serait une nouvelle valeur écrite. □

Lemme 9. *Soit une lecture o_1 et une écriture o_2, si $hd(o_1) = hd(o_2)$ alors o_1 renverra l'argument de o_2.*

Démonstration. Soit deux processus corrects p_i et p_j tel que p_i effectue l'opération d'écriture o_2 et p_j effectue l'opération de lecture o_1. Supposons que $hd(o_1) = hd(o_2)$. Soient P_1, le polynôme tel que $P_1(0)$ est renvoyé ligne 14 par p_i, et P_2, le polynôme choisi par p_j à la ligne 1.

Il existe au moins $t+1$ processus corrects p_k tel que $P_1(k)$ est égal à $reg_j[k][hd(o_1)]$ à la ligne 11. Les informations contenues dans $reg_j[k]$ étant différentes de $\perp$, on a $reg_j[k][hd(o_1)] = shards_k\,[hd(o_1)]$, qui a été édité à la ligne 16 quand p_k a reçu le message SHARE$(shard, hd(o_1))$ envoyé par le processus écrivain à la ligne 3. On a donc $P_1(k) = reg_j[k][hd(o_1)] = P_2(k)$.

Finalement, il existe au moins $t+1$ valeurs différentes de k telles que $P_1(k) = P_2(k)$. Par unicité de l'interpolation de Lagrange pour des polynômes de degré au plus t, on a $P_1 = P_2$, donc la valeur $P_1(0)$ retournée par la lecture o_1 est égale à l'argument $P_2(0)$ de l'écriture o_2. □

Lemme 10 (Linéarisabilité pour l'écrivain correct)**.** *Supposons l'écrivain correct. Il existe un ordre total $<$ sur toutes les écritures et toutes les lectures faites par les correctes, tel que :*

- *l'exécution des opérations dans l'ordre $<$ respecte la spécification séquentielle*
- *si e se termine avant que e' ne commence, alors $e < e'$*

Démonstration. Considérons une exécution de l'algorithme 1. Nous définissons la relation binaire $\rightarrow$ entre des opérations o_1 et o_2 par : $o_1 \rightarrow o_2$ si, soit 1) o_1 est terminé avant que o_2 ne commence (noté : $o_1 \rightarrow_1 o_2$), ou 2) $hd(o_1) < hd(o_2)$ (noté : $o_1 \rightarrow_2 o_2$), ou 3) o_1 est une écriture, o_2 est une lecture et $hd(o_1) \leq hd(o_2)$ (noté : $o_1 \rightarrow_3 o_2$), ou 4) $o_1 = o_2$ (noté : $o_1 \rightarrow_4 o_2$).

Remarquons que si $o_1 \to o_2$ alors $hd(o_1) \leq hd(o_2)$ et si de plus o_2 est une écriture différente de o_1, alors $hd(o_1) < hd(o_2)$. Ceci est vrai par définition pour $\to_2$ et $\to_3$. Pour $\to_1$, s'il s'agit d'une lecture suivie d'une lecture, c'est vrai par le lemme 8. S'il s'agit d'une écriture suivie d'une lecture, c'est vrai par le lemme 7. S'il s'agit d'une lecture suivie d'une écriture, c'est vrai par le lemme 6. S'il s'agit d'une écriture suivie d'une écriture alors c'est vrai la ligne 5 et la définition de $hd()$.

La relation $\to$ est transitive par définition de $\to_4$.

Prouvons la transitivité de $\to$**.** Soient trois opérations o_1, o_2 et o_3 telles que $o_1 \to o_2 \to o_3$. Prouvons que $o_1 \to o_3$. Il faut dissocier les cas suivants.

1. Cas $o_1 \to_4 o_2$ ou $o_2 \to_4 o_3$: on a $o_1 = o_2 \to o_3$ ou $o_1 \to o_2 = o_3$, donc $o_1 \to o_3$.
2. Cas $o_1 \to_2 o_2$ ou $o_2 \to_2 o_3$: on a $hd(o_1) < hd(o_2) \leq hd(o_3)$ ou $hd(o_1) \leq hd(o_2) < hd(o_3)$, donc $o_1 \to_2 o_3$.
3. Cas $o_1 \to_1 o_2 \to_1 o_3$: o_1 termine avant que o_2 ne commence et o_2 termine avant que o_3 ne commence. Donc o_1 est terminée avant que o_3 ne commence. Ainsi, $o_1 \to_1 o_3$.
4. Cas $o_1 \to_1 o_2 \to_3 o_3$: o_2 est une écriture. Si o_1 est une lecture, on a $hd(o_1) < hd(o_2)$ par le lemme 6. Si o_1 est une écriture alors $hd(o_1) < hd(o_2)$ par la ligne 5. Dans les deux cas, $hd(o_1) < hd(o_3)$, donc $o_1 \to_2 o_3$.
5. Cas $o_1 \to_3 o_2 \to_1 o_3$: On a $hd(o_1) \leq hd(o_2) \leq hd(o_3)$. Si $hd(o_1) < hd(o_3)$, alors $o_1 \to_2 o_3$. Sinon $hd(o_1) = hd(o_3)$. Comme o_1 est une écriture (par définition de $\to_3$) o_3 ne peut pas être une écriture à cause de la ligne 5. Donc o_3 est une lecture, donc $o_1 \to_3 o_3$.
6. Cas $o_1 \to_3 o_2 \to_3 o_3$: Impossible car o_2 serait à la fois lecture et écriture.

Prouvons l'antisymétrie de $\to$**.** Soient deux opérations o_1 et o_2 telles que $o_1 \to o_2 \to o_1$. Prouvons que $o_1 = o_2$. Il faut dissocier les cas suivants.

1. Cas $o_1 \to_4 o_2$ ou $o_2 \to_4 o_1$: on a bien $o_1 = o_2$.
2. Cas $o_1 \to_2 o_2$ ou $o_2 \to_2 o_1$: on a $hd(o_1) < hd(o_1)$, ce qui est absurde.
3. Cas $o_1 \to_1 o_2 \to_1 o_1$: Paradoxale car o_1 aurait terminé avant d'avoir commencé.
4. Cas $o_1 \to_1 o_2 \to_3 o_1$: Par définition de $\to_3$, o_2 est une écriture et o_1 est une lecture, et par définition de $\to_1$, o_1 se termine avant que o_2 ne commence. D'après le lemme 6, on a $hd(o_1) < hd(o2)$. Cela contredit la définition de $\to_3$ selon laquelle $hd(o_1) = hd(o_2)$.
5. Cas $o_1 \to_3 o_2 \to_1 o_1$: On a $o_2 \to_1 o_1 \to_3 o_2$, qui nous ramène au cas précédent.
6. Cas $o_1 \to_3 o_2 \to_3 o_1$: Impossible car o_2 serait à la fois lecture et écriture.

La relation $\to$ est donc une relation d'ordre partiel, qui contient le temps-réel, par définition de $\to_1$.

Supposons le processus écrivain correct nommé p_w, la totalité des lectures effectuées par processus p_k tel que $hd(o_{k0})$ soit égal à 0 sont placés avant la première écriture o_1 tel que $\forall o_{k0}, o_{k0} \to_2 o_1$. Ensuite, le lemme 9 nous dit que si une lecture o_k a lieu entre l'écriture o_n et l'écriture o_{n+1} alors $hd(o_n) = hd(o_k)$. Nous obtenons donc l'ordre $o_n \to_1 o_k$ et $o_k \to_2 o_{n+1}$. Cela nous créer un ordre partiel que nous pouvons étendre à un ordre total.

La relation binaire $\to$ peut être appliquée à un ordre total qui respecte le temps réel grâce à $\to_1$ et chaque lecture renvoie la valeur initiale si l'horodatage vaut 0 or la valeur écrite par l'écriture précédente puisque *sn* est mis à jour ligne 5 conjointement avec k ligne 11 grâce à $\to_2$ et $\to_3$. L'exécution considérée est donc linéarisable. □

Lemme 11 (Linéarisabilité pour l'écrivain byzantin)**.** *Supposons l'écrivain byzantin. Toute histoire acceptée par l'algorithme 1 est linéarisable byzantine pour un registre Lire/Écrire*

Démonstration. On construit une histoire H' constituée de toutes les lectures de H faites par des corrects dans laquelle on ajoute une écriture de la valeur lue avant chaque lecture. Clairement, H' est linéarisable et coïncide avec H sur toutes les lectures. □

Théorème 3 (Linéarisabilité)**.** *L'algorithme 1 implémente un registre respectant la linéarisabilité Byzantine.*

Démonstration. Les lemmes 10 et 11 détaillent les deux cas suivant le comportement de l'écrivain. □

6 Conclusion

Dans cet article, nous avons implémenté un registre Lire/Écrire linéarisable tolérant les fautes byzantines et préservant la vie privée. Pour ce faire, nous avons utilisé l'algorithme de Shamir qui permet de ne pas partager l'information entièrement et de garantir qu'un processus n'ayant pas les droits de lire la valeur, ne pourra pas la retrouver. Nous avons, par l'intermédiaire de théorèmes, démontré que notre registre est linéarisable. Pour les byzantins, notre registre peut en supporter un maximum n'atteignant pas un septième du nombre total de processus dans le système. Toutefois, il est légitime de se demander si la limite actuelle est optimale. Existe-t-il une possibilité d'améliorer cette limite tout en garantissant la sécurité et la confidentialité des données partagées ? De plus, serait-il possible d'accroître la sécurité en combinant plusieurs approches, comme l'ajout de techniques cryptographiques lors du passage des shards ? Ces questions restent en suspens et pourraient constituer une piste de recherche intéressante pour de futures études.

7 Remerciements

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

8 Références

[1] Attiya H., Bar-Noy A. and Dolev D. *Sharing memory robustly in message-passing systems.* Journal of the ACM (1995)

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

[3] Imbs D., Rajsbaum S., Raynal M., and Stainer J. *Read/write shared memory abstraction on top of asynchronous byzantine message-passing systems.* Journal of Parallel and Distributed Computing (2016)

[4] Mostéfaoui, A., Petrolia, M., Raynal, M., and Jard, C. *Atomic read/write memory in signature-free byzantine synchronous message-passing systems.* Theory of Computing Systems (2017)

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

[6] Shamir A. *How to share a secret.* Communications of the ACM 22 (1979)

[7] Shir Cohen and Idit Keidar. *Tame the Wild with Byzantine Linearizability : Reliable Broadcast, Snapshots, and Asset Transfer. In 35th International Symposium on Distributed Computing.* (DISC 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 209)

[8] Imbs D., Rajsbaum S., Raynal M. and Stainer J. *Reliable shared memory abstraction on top of asynchronous byzantine message-passing systems. In Proc. 21st Int. Colloquium on Structural Information and Communication Complexity* (SIROCCO'14) (pp. 37-53). Springer LNCS 8576 (2014).