Sorbonne Universités, UPMC Univ. Paris 06
Master Informatique – Systèmes et Applications Répartis

Rapport bibliographique

Gestion efficace des fautes byzantines dans les clouds

Bournat Marjorie

Encadrant : Olivier Marin

Enseignant référent : Julien Sopena

Table des matières

I. Introduction

Le *cloud* est de plus en plus utilisé pour effectuer des calculs, et du stockage. Cet environnement distribué et dynamique est très sujet aux fautes. Ces fautes peuvent être tout simplement des crashs ou des fautes byzantines plus complexes.

Les fautes byzantines correspondent à des comportements arbitraires des nœuds, et peuvent être dues à des erreurs logicielles, matérielles ou encore à des attaques. Le nom « fautes byzantines » est apparu suite à la métaphore utilisée par Lamport et al. [1] pour définir les comportements arbitraires des machines.

Les fautes byzantines pouvant avoir des répercussions assez lourdes sur les systèmes, et étant de plus en plus répandues du fait de la complexité croissante des applications (et donc des bugs potentiels permettant des intrusions); des techniques se sont développées pour les tolérer. Trois techniques majeures se distinguent. Certaines gèrent les fautes byzantines de manière matérielle, en utilisant des portes logiques [2]. D'autres utilisent la réplication de tâches au sein de plusieurs nœuds, puis effectuent un vote basé sur la majorité pour déterminer le résultat final. Dans la dernière méthode un nœud demande à un autre d'exécuter un ensemble de n tâches. Sur ces n tâches le demandeur connaît le résultat de n-1 d'entre elles. Lorsqu'il reçoit les réponses, il compare les résultats reçus à ceux qu'il connaît. Si ils sont identiques, le demandeur suppose que le résultat de la tâche dont il ne connaissait pas la solution est correct.

Nous nous intéressons au second type de technique.

Les algorithmes tolérant les fautes byzantines (BFT) ont plusieurs buts. Ils doivent assurer des propriétés de sûreté et de vivacité qui permettent respectivement que le système se comporte selon sa spécification, et que le système progresse dans son exécution. Certains algorithmes garantissent de plus la confidentialité des données. Ici cette dernière caractéristique ne nous intéresse pas, notre but n'étant pas de manipuler des données critiques mais d'assurer un service efficace qui tolère les fautes byzantines.

I.1. Principes de base de la tolérance aux fautes byzantines

I.1.a. Machine à états

Pour tolérer les fautes byzantines nous nous intéressons à la réplication de tâches. On utilise des machines à états pour répliquer les tâches : les machines possèdent des états, et des transitions permettant de passer d'un état à un autre. Ces transitions sont prises lorsque les entrées et événements extérieurs (comme une requête d'un client) le permettent. Pour parer aux fautes on réplique une tâche au sein de plusieurs machines à états. Les machines retournent le résultat qu'elles ont obtenu. Un vote est alors effectué pour déterminer le résultat final. Généralement, le résultat final correspond aux réponses identiques données par une majorité des machines interrogées (en espérant que cette majorité existe).

Pour un même traitement, les machines à états non fautives doivent rendre exactement le même résultat. Cela implique plusieurs caractéristiques que doivent respecter ces dernières. Tout d'abord elles doivent être initialisées avec la même valeur, ensuite le calcul qu'elles effectuent doit être déterministe (c'est à dire, étant donné un état et une suite d'opérations, elles doivent finir leur exécution dans un même état). Certains protocoles permettent de rendre déterministes des applications qui ne le sont pas, par exemple en insérant une valeur de graine qui sera passée aux répliquas, assurant alors que les nombres aléatoires générés soient identiques. Ou encore, lorsque l'action demandée concerne la demande de l'heure, certains protocoles font en sorte que la valeur médiane des réponses rendues soit choisie comme résultat final.

I.1.b. Indépendance des fautes

La réplication permet de tolérer les fautes byzantines. Cependant il faut que ces fautes soient indépendantes [3]. Pour que les fautes soient indépendantes, il faut que les répliquas soient géographiquement dispersés, que leurs architectures soient hétérogènes (évitant les failles matérielles), qu'ils s'exécutent sur des systèmes d'exploitation différents. Il faut également que le code exécuté soit différent tout en ayant le même but (et donc fournir la même réponse pour une requête identique), afin d'éviter les failles logicielles. Il est possible d'obtenir plusieurs implémentations différentes d'un même code de base [23].

En effet, supposons qu'une intrusion ait lieu à cause d'une faille dans le code. Vu que chacune des machines exécutent le même code, elles peuvent donc toutes subir la même intrusion. Tous les répliquas seront alors byzantins. Si les serveurs deviennent complices en retournant des résultats identiques mais incorrects, le système sera corrompu. Ce dernier choisira alors des mauvaises réponses, en pensant qu'elles sont correctes.

De la même manière, si les machines se trouvent sur un même site géographique, et que ce site est soumis à une panne, alors l'ensemble des nœuds seront défaillants. Le service ne sera alors plus disponible et faillira à sa fonction.

Cependant on peut nuancer ce discours. En effet, il est peu réaliste de réunir toutes ces conditions au sein d'un système distribué. Dans les systèmes répartis la dispersion géographique des machines est aisément réalisable. Cependant il est difficile d'avoir des systèmes d'exploitation et des codes différents pour chacune des machines mises à disposition. Malgré cela, faire l'hypothèse que les fautes sont indépendantes est raisonnable.

I.1.c Algorithme de base : PBFT (Practical Byzantine Fault Tolerance)

Avant de présenter la bibliographie nous allons nous intéresser à l'algorithme de Castro et Liskov [4] qui permet de tolérer les fautes byzantines. La plupart des protocoles ont utilisé ce protocole comme base de travail et cherchent à l'améliorer. C'est un des protocoles les plus reconnus dans le domaine de la tolérance aux fautes byzantines. Ce protocole peut être utilisé en pratique car il fait des hypothèses assez faibles et réalistes. La plupart des algorithmes BFT [5, 14, 17, 19, 20] sont basés sur cet algorithme et reprennent les mêmes principes (protocole d'accord, point de reprise, changement de vue).

Approche :

Le PBFT est un algorithme client serveur. Le client envoie une requête au service répliqué, puis se met en attente de la réponse. Le service répliqué retourne des réponses correctes tant qu'il y a au plus f fautes pour 3f+1 répliquas au minimum.

Le principe de l'algorithme est le suivant : Il y a plusieurs vues successives, dans chaque vue un nœud est désigné comme étant le *primary* et les autres sont les *backups*. On change de vue dès que l'on constate que le *primary* est en erreur.

Le client envoie une requête au *primary* qui transmet cette requête à tous les *backups*. Les répliquas exécutent la requête et transmettent la réponse au client. Le client attend d'avoir reçu f+1 réponses identiques signées correctement et prend alors ces réponses comme résultat.

Le client ne connaît pas forcément le *primary* courant. Il y a, inclus dans chaque réponse envoyée par les répliquas, le numéro de vue qui permet de connaître (s'il n'a pas été fautif depuis) le numéro du *primary*. Si la réponse est trop longue le client envoie sa requête à l'ensemble des répliquas. Les répliquas envoient alors la réponse au client si la requête a déjà été exécutée (il y a un cache). Sinon ils transfèrent la requête au *primary*, et si celui-ci ne diffuse pas la requête accompagnée d'un numéro de séquence à tous les *backups* à l'expiration d'un temporisateur, il sera suspecté d'être fautif par au moins une majorité de *backups* et la vue changera.

Lorsqu'il reçoit une requête du client, le *primary* lance un protocole *3-phase commit* (*pre-prepare*, *prepare*, *commit*) pour effectuer la diffusion de la requête du client vers les *backups*. Ce protocole établit un consensus sur l'ordre de réception des requêtes.

Le déroulement de ce consensus est le suivant (voir Figure 1) :

Le *primary* envoie un message *pre-prepare* à l'ensemble des *backups* avec un numéro de séquence pour la requête précisée dans le message. Un *backup* va accepter un message *pre-prepare* si la vue indiquée dans le message reçu correspond à la vue courante du répliqua, si les signatures de la requête et du message *pre-prepare* sont correctes, si le digest transmis dans le message *pre-prepare* correspond bien à la requête du client, et si il n'a pas déjà accepté une requête différente de la requête précisée dans le message avec le même numéro de séquence. Une fois que le *backup* a accepté un message il va alors envoyer un message *prepare* à l'ensemble des autres répliquas en incluant le numéro de séquence choisi. Lorsqu'un répliqua a reçu 2f messages *prepare* correspondant au message *pre-prepare* reçu il envoi un message *commit* à tous les autres répliquas. Le message *commit* sera accepté s'il correspond aux messages *prepare*. Lorsque les répliquas ont accepté 2f+1 *commit,* ils exécuteront la requête à la place indiquée par le numéro de séquence. Une fois la requête exécutée les répliquas envoient la réponse au client.

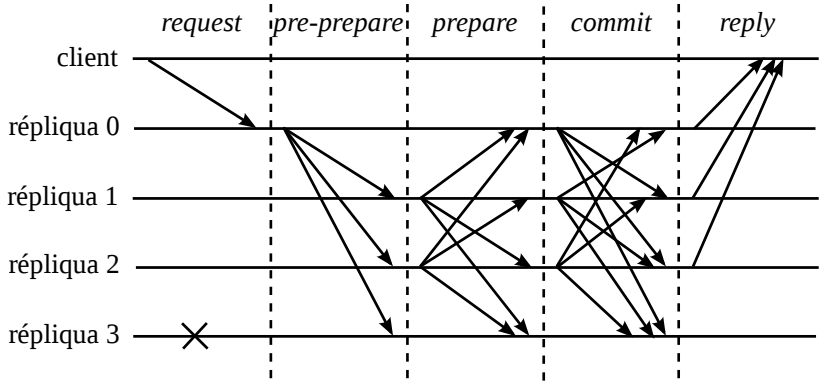

Figure 1- échanges de messages dans PBFT (f=1), avec le nœud 0 étant le *primary,* avec le nœud 3 fautif

Les répliquas stockent les messages qu'ils ont envoyés et reçus dans un journal. Mais ce journal n'étant pas infini ils effectuent des points de reprise qui leur permettent de ne pas conserver tous les messages. Dès que les répliquas effectuent un point de reprise ils envoient un message à l'ensemble des autres répliquas. Si les répliquas reçoivent 2f+1 points de reprise identiques correctement signés venant de différents répliquas, le point de reprise est stable. Les répliquas peuvent alors ne conserver dans le journal que le dernier point de reprise stable, leur état courant, ainsi que les messages non inclus dans le point de reprise.

Pour garantir la vivacité, PBFT s'appuie sur un mécanisme de changement de vue. Les *backups* ont des temporisateurs, quand un répliqua n'a toujours pas reçu de requête depuis trop de temps, il envoie un message *view-change* à tous les répliquas pour passer de la vue *v* à la vue *v+1*, et donc changer de *primary*. Lorsque le *primary* de la vue *v+1* a reçu 2f messages de ce type, il envoie un message *new-view* contenant la preuve que 2f répliquas voulaient changer de vue. Les répliquas changent de vue sur réception du message *new-view* si celui-ci contient des informations correctes et qu'il est correctement signé. Pour éviter les changements de *primary* dus à de fausses suspicions, à chaque changement de vue la valeur des temporisateurs est doublée.

Remarque :

Les 3f+1 répliquas sont nécessaires (et suffisants) pour effectuer le consensus en tolérant f fautes dans un environnement asynchrone augmenté avec des détecteurs de fautes. En effet, lors du consensus on attend à chaque étape des messages de n-f autres machines vu que f peuvent être fautives et donc ne pas répondre. Mais si les f répliquas dont on n'a pas pris en considération les messages sont corrects, cela signifie que parmi les n-f messages il y en a f qui seront possiblement faux. Pour pouvoir alors se mettre d'accord il faut que les répliquas corrects soient en majorité parmi les n-f messages pour être sûr d'avoir un ordonnancement final correct. Il faut donc que l'on ait n-2f > f, et donc n > 3f.

I.2. Contexte du stage

La bibliographie sera présentée suivant les objectifs que l'on souhaite atteindre dans ce stage.

L'objectif du stage est d'apporter une solution probabiliste efficace pour tolérer les fautes byzantines au sein du *cloud*. On veut réduire le nombre de répliquas et ne pas fixer le nombre maximum de fautes byzantines que le système est apte à tolérer. Il faut que le temps de réponse de cette solution soit faible. En offrant des garanties probabilistes et non déterministes, le nombre de répliquas utilisés pour effectuer les calculs sera réduit et adapté au dynamisme du *cloud* (c'est-à-dire on pourra facilement intégrer ou supprimer des nœuds qui participent aux calculs). De plus, notre solution doit être robuste face aux fautes, c'est à dire garder des performances correctes même lorsque le nombre de fautes augmente. Enfin, ce sujet se plaçant dans le *cloud* et pouvant donc potentiellement être très sollicité par les clients, il faut que notre solution passe à l'échelle.

Hypothèses de travail :

Dans le cadre de ce stage, nous ferons l'hypothèse que les fautes sont indépendantes. Nous nous appuierons de plus sur des entités fiables. Ces entités ordonnanceront les tâches. De plus nous considérerons un réseau partiellement synchrone. Ce réseau partiellement synchrone est nécessaire pour assurer la vivacité.

Parmi les algorithmes qui tolèrent les fautes byzantines on s'intéresse à ceux qui vont essayer d'améliorer (par rapport à PBFT) la latence (donc le temps de réponse) dans le cas sans faute, ceux qui vont utiliser un nombre de répliquas plus petit que 3f+1 (pour économiser des ressources), ceux qui vont s'adapter au réseau sous-jacent pour choisir les répliquas (et donc prendre en compte les nœuds qui entrent dans le système ou le quittent), puis ceux qui permettent d'être performants même lorsque le nombre de fautes augmente.

II. Techniques pour réduire la latence dans le cas sans fautes

Dans le cas sans faute, un protocole *3-phase commit* est effectué pour ordonnancer les requêtes dans PBFT. Son coût en nombre de messages est quadratique en fonction du nombre de machines impliquées dans ce consensus, et le temps de réponse correspond à 5 envois de message successifs. Des protocoles BFT utilisent des quorums, des chaînes, de la spéculation, ou encore de la détection d'intrusion pour diminuer le nombre de messages et la latence.

II.1. Gestion par groupes

Pour pouvoir améliorer la latence et le coût de PBFT, des protocoles BFT permettent au client d'interroger directement tous les serveurs ou un sous-ensemble des serveurs mis à disposition.

II.1.a. ensemble de serveurs

Ce procédé est utilisé au sein du protocole Aliph [5]. Aliph est un protocole composé de plusieurs protocoles BFT, qui sont exécutés à tour de rôle en fonction des conditions du système. Lorsqu'il n'y a pas de faute des répliquas, ni du réseau (pas de retard ni de perte de messages), ni de clients byzantins, et qu'il n'y a pas de concurrence, Aliph utilise un protocole appelé Quorum pour ordonnancer les requêtes qui lui sont soumises.

L'ordonnancement se fait de manière simpliste. Le client envoie sa requête à l'ensemble des répliquas (qui sont au moins 3f+1 pour pouvoir tolérer f fautes) et se met en attente d'une réponse. Les répliquas à la réception d'une requête l'exécutent puis renvoient la réponse au client (voir Figure 2). Si le client reçoit 3f+1 réponses identiques et correctement signées, alors il valide le résultat. Sinon c'est que le système ne se trouve plus dans les conditions énoncées précédemment il faut donc changer de protocole BFT pour exécuter cette requête.

Cette méthode est également utilisée au sein du protocole HQ [8]. En effet lorsqu'il n'y a pas de concurrence ce protocole utilise un quorum pour effectuer les requêtes de lecture, et 2 quorums pour effectuer les requêtes qui modifient l'état des répliquas.

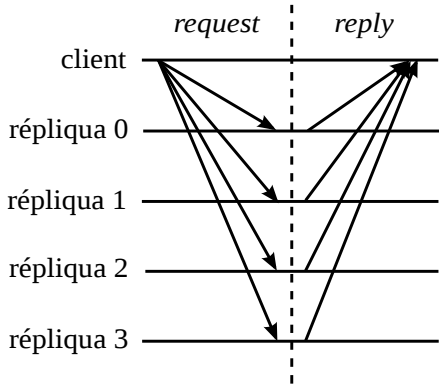

Figure 2 – échanges de messages de type « quorum » (f=1)

Le protocole décrit dans [10] peut être vu comme un dérivé de ce procédé. En effet, dans ce protocole un nœud supposé correct envoie une requête à un sous-ensemble de répliquas (dont il suppose la probabilité d'être fautif assez faible pour obtenir un résultat correct), la réponse qu'il reçoit de manière majoritaire (dans le sous-ensemble interrogé) est validée comme étant le résultat. S'il n'arrive pas à obtenir une réponse majoritaire, le nœud va interroger des nœuds supplémentaires.

Cette technique permet de réduire la latence puisque l'on n'a besoin que de 2 envois de message en termes de temps pour obtenir une réponse, et que le nombre de messages est linéaire en fonction du nombre de répliquas. C'est une réelle amélioration par rapport au PBFT. Cependant cette technique se place dans le cas très particulier où il n'y a ni faute ni concurrence.

II.1.b. Quorum

Le fait d'avoir introduit le protocole décrit dans [10] nous permet de constater un défaut majeur de cette technique. En effet, bien que tous les protocoles précédents prétendent utiliser des quorums, ils ne font en réalité qu'interroger l'ensemble des répliquas de manière directe. Le protocole [10] quant à lui n'interroge qu'un sous-ensemble de répliquas. Or un client est supposé pouvoir effectuer des écritures. Ainsi si deux clients effectuent des écritures en parallèle en interrogeant des sous-ensembles de répliquas différents alors nos machines à états seront dans des états incohérents.

On parle de quorums lorsque l'on divise un groupe de nœuds en plusieurs sous-groupes. L'intersection de ces sous-groupes deux à deux doit être non vide. Pour demander l'exécution d'une requête on va interroger un sous-groupe. Les répliquas dans l'intersection sont au courant des requêtes concurrentes, cela permet d'assurer la cohérence des machines à états. On retrouve cette technique au sein du protocole Q/U [12] (par contre cette technique nécessite 5f+1 répliquas).

Le nombre de messages et le temps de réponse sont identiques à ceux décrits dans la section II.1.a. le principe d’envoi de requête, exécution, réponse restant le même.

II.2. Chaînes

Ce procédé est utilisé au sein du protocole Chain [5]. Chain est utilisé au sein d'Aliph lorsque les répliquas ne sont pas fautifs, que les clients ne sont pas byzantins et qu'il n'y a ni perte ni retard de messages. La concurrence est cependant autorisée.

Dans Chain les répliquas (qui sont au minimum 3f+1 pour pouvoir tolérer f fautes) sont organisés en chaîne. Il y a donc une tête et une queue, et tous les autres nœuds possèdent deux voisins. Le client envoie sa requête à la tête de la chaîne qui va lui attribuer un numéro de séquence, l'exécuter et transmettre le message à son voisin en incluant le numéro de séquence et sa réponse. Chaque répliqua (sauf la queue) sur réception du message, exécute la requête, la transmet à son voisin accompagné de sa réponse et des réponses de ces ancêtres. La queue de la chaîne quant à elle lorsqu'elle reçoit le message, exécute la requête puis envoie l'ensemble des réponses au client (voir Figure 3). Si le client a reçu 3f+1 réponses identiques et correctement signées, il considère le résultat comme correct. Sinon (dans le cas où les réponses ne sont pas similaires, ou dans le cas où il n'a pas reçu 3f+1 réponses après expiration d'un temporisateur) il fait appel à un autre protocole pour exécuter sa requête car il y a des fautes dans le système.

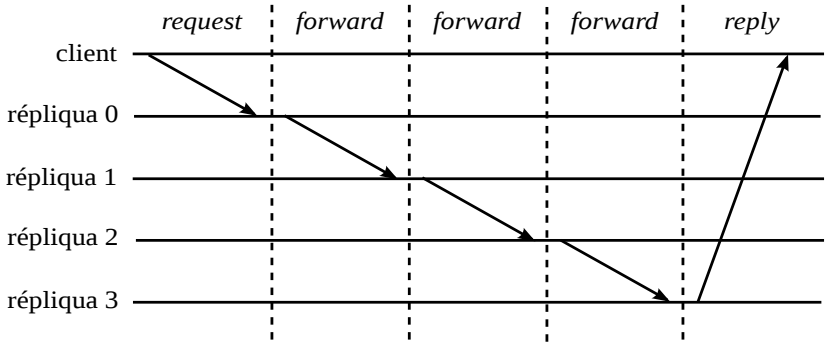

Figure 3 – échange de messages de type chaîne (f = 1), le nœud 0 est le *primary*

Le temps de réponse tout comme la complexité en messages sont linéaires en fonction du nombre de répliquas impliqués dans le calcul. Malgré le pipeline de l'exécution des requêtes qui est un point de vue intéressant, cette technique peut ralentir considérablement le système si un nœud est fautif et que le temporisateur du client n'est pas calibré de manière adéquate.

II.3. Spéculation

La spéculation est utilisée dans le protocole Zyzzyva [6]. Ce protocole, tout comme PBFT, possède un nœud principal (qui change en même temps que les vues). La spéculation fait l'hypothèse optimiste que tous les répliquas exécuteront les requêtes dans le même ordre.

Le client envoie une requête au nœud principal (appelé *primary*) qui sur réception de cette dernière lui attribue un numéro d'ordre et la transfère à l'ensemble des autres répliquas. Tous les serveurs exécutent la requête (si elle possède un numéro de séquence supérieur de 1 à la précédente requête qu'ils avaient exécutée) et envoient la réponse au client (voir Figure 4). On pourrait à ce point de la description se dire que la spéculation n'a pas de sens vu que le *primary* attribue un numéro de séquence et fournit la requête avec ce numéro de séquence aux répliquas. On se demande alors de quelle manière les serveurs pourraient ne pas être d'accord avec cet ordre imposé et pourquoi il n'exécuteraient pas les requêtes dans cet ordre (s'ils sont corrects). Cependant il ne faut pas oublier qu'on considère des protocoles BFT, et donc il est possible d'avoir un nœud principal fautif n'envoyant pas le même numéro de requête à tous les répliquas.

Si le client reçoit 3f+1 réponses identiques il valide le résultat.

Si le client reçoit entre 2f+1 et 3f+1 réponses identiques, dans ce cas soit certaines réponses ne sont pas identiques soit il n'a pas reçu toutes les réponses après expiration d'un temporisateur. Dans tous les cas il enverra un certificat (correspondant à un message haché des 2f+1 réponses identiques reçues) à l'ensemble des serveurs, puis si 2f+1 répliquas lui retournent un acquittement, il valide le résultat.

Si maintenant le client a reçu moins de 2f+1 réponses identiques, cela peut venir de deux phénomènes : le temporisateur du client n'est pas adapté au système (qui peut être dans une phase instable) ou le principal est fautif. Le client diffuse alors sa requête à tous les répliquas. Là plusieurs solutions sont possibles pour le répliqua qui reçoit une telle requête. Soit la requête possède une estampille inférieure ou égale à la dernière requête exécutée pour ce client, dans ce cas le répliqua envoie la réponse à cette requête qu'il possède dans son cache. Sinon il transfère la requête au *primary* en armant un temporisateur. S'il reçoit une demande d'exécution ordonnée par le *primary* il l'effectue comme décrit au-dessus sinon si le temporisateur expire avant une telle demande un changement de vue est initié. Si le client a reçu des réponses indiquant que les répliquas ont reçu des ordonnancements différents de requêtes le client le leur signale et un changement de vue est diffusé pour évincer le *primary*.

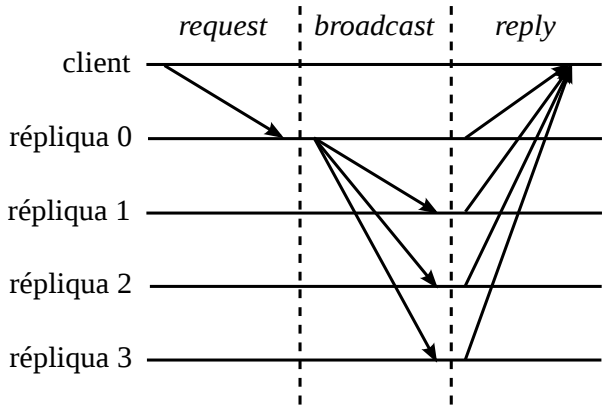

Figure 4 – échanges de messages de type spéculation (f = 1), le nœud 0 est le *primary*

De même dans le sous protocole Zlight du protocole AZyzzyva [5] la spéculation est utilisée. AZyzzyva est tout comme Aliph un protocole composé de plusieurs protocoles. Zlight est un protocole qui imite Zyzzyva mais qui n'est exécuté que lorsque les répliquas ne sont pas fautifs, que les clients ne sont pas byzantins et qu'il n'y a pas de perte ni de retard de messages. Lorsque les conditions précédentes ne sont pas satisfaites le protocole n'imite plus Zyzzyva. Il utilise PBFT qui en cas de fautes est plus performant que Zyzzyva. Ainsi dès lors que le client ayant invoqué Zlight ne reçoit pas 3f+1 réponses identiques il invoque PBFT.

Ce procédé est aussi utilisé au sein du protocole UpRight [7]. Cependant contrairement à Zyzzyva ce ne sont pas les clients qui décèlent les divergences au niveau des réponses. En effet, dans UpRight les serveurs d'exécution effectuent régulièrement des points de reprises qu'ils envoient aux nœuds qui s'occupent de l'ordonnancement. Ainsi ce sont ces derniers qui lors de la réception des points de reprise se rendent compte des potentielles incohérences et prennent les décisions nécessaires pour les corriger.

Le protocole BFTCloud décrit dans [9] est également un protocole basé sur la spéculation. Cependant par rapport à Zyzzyva, le nœud principal qui ordonne les requêtes n'est pas choisi en fonction de son identifiant et du numéro de vue, mais en fonction de ses caractéristiques (qualité de service et probabilité d'être fautif).

Avec cette méthode le temps de réponse correspond à 3 envois de messages, et le nombre de messages est linéaire en fonction du nombre de répliquas utilisés.

Remarques :

→ La spéculation ressemble beaucoup à l'utilisation de quorum mis à part qu'il y a une indirection de plus. Dans la spéculation le client interroge un *primary* tandis que dans l'utilisation d'un quorum de processus le client interroge directement les répliquas.

→ Le protocole UpRight utilise la spéculation pour la communication entre ses serveurs. Cependant le client utilise le quorum pour s'adresser au service. Le coût et le temps de réponse de UpRight ne correspondent pas au coût et latence des techniques utilisant la spéculation seule. Son coût est de $(u+r)^2$ où u correspond aux nombre de crashs, et r aux nombres nœuds byzantins que UpRight peux tolérer. En termes de latence il faut 7 envois de messages pour que le client connaisse le résultat de sa requête.

II.4. Détection d'intrusion

Pour réduire la latence on peut également utiliser des détecteurs. Le protocole ByzID [11] (qui tout comme PBFT fait une distinction entre les répliquas en désignant un nœud particulier comme étant le *primary* et les autres répliquas comme étant les *backups*) utilise un détecteur d'intrusion pour diminuer la latence.

Dans le principe ce mécanisme est identique à la spéculation dans le sens où le client envoie une requête au *primary* qui est chargé de l'ordonnancement et du transfert des requêtes aux *backups*. Tous les répliquas exécutent les requêtes et envoient la réponse au client.

Cependant les *backups,* en même temps que l'envoi de leurs réponses, envoient un acquittement au *primary* pour lui signaler qu'ils ont envoyé leurs réponses au client. Cela permet au *primary* d'être sûr que les *backups* ont reçu la requête.

De plus, à l'inverse de la spéculation, si la requête est arrivée jusqu'aux *backups* on est sûr que les répliquas ont reçu le même ordonnancement de requêtes. Ceci est assuré à l'aide de règles insérées au sein d'un détecteur d'intrusion (ID).

Tout d'abord l'ID vérifie que le *primary* transfère les requêtes dans le même ordre qu'il les a reçus. Le détecteur vérifie que le numéro de séquence que le *primary* donne aux requêtes est incrémenté de 1 à chaque nouvelle requête. Ensuite le détecteur vérifie que le nombre de messages que le *primary* envoie avec le même numéro de séquence est égal au nombre total de répliqua – 1 (on ne prend pas le *primary* en compte). Enfin le détecteur met en place des temporisateurs afin de vérifier que le *primary* n'est pas trop lent voir crashé. Les temporisateurs sont initialisés avec des détecteurs *anomaly-based* qui surveillent et maintiennent des statistiques sur le système. Cela permet d'avoir un temporisateur dont la valeur n'est pas aberrante. Si le détecteur s'aperçoit d'un problème il bloque les transferts de requêtes du *primary* vers les *backups* et lance une reconfiguration du *primary*. Le client attend de recevoir f+1 réponses identiques (sur les 2f+1 répliquas qui ont exécuté la requête) pour choisir le résultat.

Ici le coût de cette technique est linéaire en fonction du nombre de répliquas et correspond à 3 envois de message en termes de temps. Ainsi ces valeurs sont identiques à celles observées pour la spéculation. Cependant on est sûre qu'une fois la requête exécutée par les répliquas il n'y aura pas d'incohérence au sein de ces derniers. La spéculation elle, impose un mécanisme de retour en arrière dans l'exécution si l'on constate des incohérences.

III. Diminution du nombre de répliquas et élasticité

Au sein de PBFT on utilise 3f+1 répliquas pour tolérer f fautes. Cependant même si dans le cloud les ressources se trouvent en grande quantité, on voudrait pouvoir réduire le nombre de répliquas impliqués dans l'exécution d'une tâche. Dans le cas sans faute on gâche des ressources inutilement si on doit utiliser 3f+1 répliquas .

III.1. Diminution fixe du nombre de répliquas

Tout d'abord on s'intéresse aux protocoles qui permettent de diminuer le nombre de répliquas. Cependant le nombre de répliquas est fixe et le groupe de répliquas interrogé est figé (aucun répliqua n'est supprimé ou ajouté au groupe).

III.1.a. Séparation de la phase d'accord et de l'exécution, réduction du nombre de machines d'exécution

Pour diminuer le nombre de répliquas utilisés, certains protocoles comme [14] ont séparé la phase d'accord de la phase d'exécution. Ainsi pour tolérer f fautes ils ont toujours besoin de 3f+1 machines pour la phase d'accord mais ne nécessitent que 2g+1 machines pour l'exécution (pour tolérer g fautes).

La séparation entre les machines qui se mettent d'accord sur l'ordre d'exécution des requêtes et les machines qui exécutent les requêtes est possible car les premières peuvent fournir aux deuxièmes une preuve cryptée et vérifiable de l'ordre des requêtes.

Bien qu'utilisant 3f+1 machines pour l'accord, cette technique est tout de même moins coûteuse que les protocoles ne séparant pas l'accord de l'exécution. En effet, les machines d'exécution sont plus chères que les machines d'accord en termes de matériel de stockage, de calcul, mais également en termes de logiciel.

La phase d'accord s'effectue comme pour PBFT avec un protocole *3-phase commit*.

On a besoin pour élire le bon résultat qu'une majorité des machines retourne le même résultat. Ainsi on a juste besoin de 2g+1 machines pour exécuter les requêtes si l'on considère qu'il peut y avoir g fautes byzantines (crashs compris).

Le principe de l'algorithme est le suivant: le client envoie une requête certifiée au cluster d'accord. Le client se met alors en attente de la réponse certifiée. Cette réponse lui sera donnée lorsque g+1 répliquas auront fourni la même réponse.

Le protocole Zyzzyva [6] et UpRight [7] utilisent également cette méthode. (Dans sa forme originelle Zyzzyva ne sépare pas l'accord de l'exécution mais, les auteurs de ce protocole proposent une optimisation se basant sur ce principe réduisant alors le nombre de répliquas utilisé pour l'exécution).

Les méthodes présentées dans cette section permettent de réduire le nombre de répliquas liés à l'exécution à 2g+1 tandis que le nombre de répliquas permettant l'ordonnancement des requêtes est toujours de 3f+1. Ainsi ces techniques ne sont pas vraiment efficaces en termes de réduction de répliquas.

III.1.b. Protocole à base d'oracle

Le protocole décrit dans [16] utilise seulement 2f+1 répliquas au total (ordonnancement et exécution) en considérant f fautes. En effet au sein de chaque répliqua, un oracle est positionné dans le noyau. Cet oracle est supposé non malicieux (seuls les crashs sont autorisés). L'ensemble des oracles sont reliés par un réseau fiable. Le client envoie sa requête à n'importe quel serveur qui fera appel à son oracle local pour ordonnancer la requête. Lors de la réception d'un message les serveurs font alors appel à leur oracle qui leur donnera l'ordre dans lequel ils doivent exécuter les requêtes reçues.

Cette technique permet de réduire le nombre de répliquas total à 2f+1 pour tolérer f fautes. Cependant elle se base sur un réseau synchrone (qui relie les oracles) pour pouvoir obtenir un ordonnancement fiable des requêtes.

III.2. Élasticité du nombre de répliquas

Dans cette section nous allons nous intéresser aux protocoles qui n'utilisent pas toujours le même sous ensemble de répliquas pour effectuer la phase d'accord et/ ou d'exécution. Certains d'entre eux ne nécessitent plus l'utilisation de 3f+1 machines pour tolérer f fautes, tandis que d'autres ont toujours besoin de cette condition afin de donner un résultat correct.

III.2.a. Élasticité avec nombre fixe de répliquas

Le protocole [17] nécessite 3f+1 répliquas pour l'ordonnancement et l'exécution. Ce protocole est très similaire au PBFT de base. Cependant il permet l'ajout et la suppression de nœuds en cours d'exécution. Pour ce faire le protocole ajoute un nœud appelé *view manager* qui est digne de confiance. Ce *view manager* envoie une requête aux répliquas en indiquant l'identifiant des nœuds qu'il souhaite supprimer et ajouter au système. Le *view manager* peut également demander à tolérer plus ou moins de fautes f. La requête du *view manager* est ordonnancée de la même manière que les requêtes des autres clients. Cela garantit que la vue proposée par le *view manager* sera adoptée au même point de l'exécution par tous les répliquas corrects. Les répliquas ouvriront alors des liens de communications sécurisés avec les nouveaux nœuds et fermeront ceux des nœuds supprimés. Les répliquas envoient alors un message au *view manager* pour indiquer que la vue a changé. Le *view manager* envoie ensuite un message aux nœuds en attente d'être ajoutés pour leur signaler qu'ils peuvent participer à la vue courante. Les nœuds ainsi ajoutés vont effectuer une phase de recouvrement afin d'être à jour.

Le protocole ByzID [11] permet de s'abstraire de la phase d'accord grâce à des détecteurs. En effet, le protocole utilise un nœud particulier : le *primary*, qui effectue l'ordonnancement. Ce dernier est attentivement surveillé de telle sorte qu'il envoie bien le même ordre à l'ensemble des répliquas. Ainsi pour ordonnancer les requêtes on n'a besoin que d'un seul nœud (ainsi que le détecteur associé à ce nœud). La phase d'exécution ne nécessite que 2f+1 répliquas, le *primary* est inclus dans les nœuds exécutant les requêtes. Vu que l'on suppose f fautes, et sachant que tous les répliquas exécuteront bien les requêtes dans le même ordre, il suffit d'obtenir f+1 réponses identiques pour connaître le résultat final. Donc en considérant f fautes au maximum on doit interroger 2f+1 répliquas. On peut ajouter que chacun des nœuds est associé à un détecteur. Dés lors qu'un détecteur observe une défaillance sur un nœud (qu'il soit *primary* ou utilisé pour l'exécution), une reconfiguration est lancée. Cette reconfiguration consiste à remplacer un nœud détecté fautif par un autre nœud n'appartenant pas à l'ensemble originellement considéré.

Le protocole présenté par A. Agbaria et R. Friedman [15] utilise le principe de la séparation de l'accord et de l'exécution. Dans ce protocole les nœuds qui s'occupent de l'ordonnancement ne demandent qu'à g+1 (si on tolère g fautes) machines d'exécuter la requête. Si les g+1 répliquas ont fourni la même réponse alors on a utilisé seulement g+1 machines pour trouver le résultat. Si par contre les nœuds n'ont pas tous retourné le même résultat, le protocole interroge g autres répliquas pour pouvoir obtenir une majorité, on retombe alors dans le cas explicité par [14] avec l'interrogation de 2g+1 machines d'exécution. La différence avec [14] est qu'à l'issue de l'interrogation des 2g+1 répliquas, les nœuds d'accord identifient les nœuds fautifs et les suppriment.

Le temps de réponse de ce protocole peut cependant être long. Régulièrement les machines d'exécution effectuent des *checkpoints* qu'elles enverront aux répliquas d'accord. Ces derniers effectueront alors un protocole d'accord pour déterminer si les serveurs d'exécution sont cohérents entre eux. Cela peut être coûteux notamment si la fréquence de ces *checkpoints* est mal calibrée. Une fréquence élevée engendrerait trop d'exécution de protocole d'accord (parfois plusieurs protocole pour une même tâche). Et une fréquence trop peu élevée entraînerait en cas de fautes une perte de performance assez importante. En effet, il faudrait reprendre l'ensemble des exécutions depuis le dernier point de reprise stable.

Ces techniques permettent d'obtenir de l'élasticité en permettant l'ajout et la suppression de nœuds. Cependant ces techniques ont besoin d'un nombre fixe de répliquas pour s'exécuter.

III.2.b Élasticité avec nombre variable de répliquas

Certains protocoles utilisent uniquement une sous-partie des répliquas mis à disposition pour exécuter des requêtes. Les répliquas sont choisis soit de manière arbitraire soit en fonction de leurs caractéristiques.

Méthode de sélection des répliquas de manière arbitraire :

L'article [18] présente plusieurs techniques utilisant un sous-ensemble de répliquas pour exécuter les requêtes. Les répliquas sont choisis de manière aléatoire.

Voici le principe de ces méthodes :

→ *traditional redundancy* : consiste à répliquer des tâches sur k={3, 5, 7, …} nœuds de manière indépendante et sans prendre en compte la fiabilité de chaque nœud. La fiabilité de cette technique s'approche de 1 lorsque l'on augmente le nombre k de machines utilisées pour répliquer. Le coût de cette technique augmente linéairement lorsque k augmente. Pour déterminer le résultat final un vote est effectué : le résultat final correspond aux réponses identiques données par au moins (k+1) / 2 machines.

Une autre technique identique à *traditional redundancy* est introduite dans [10] et est appelée *fixed-size*. Cette méthode reprend la même idée que *traditional redundancy* à l'exception que le nombre k n'a pas besoin d'être impair, et que ce dernier est fixé (par un administrateur par exemple). Cette technique est notamment utilisée au sein de SETI@HOME.

→ *progressive redundancy* : va lancer (k + 1) / 2 tâches et si un consensus est atteint (tous les nœuds interrogés ont rendu le même résultat) alors le résultat est conservé. Sinon on va distribuer les calculs sur le minimum de nœuds supplémentaires qui pourrait nous permettre d'atteindre un consensus. Ce procédé est répété jusqu'à atteindre un consensus. Il est moins coûteux en termes de répliquas que le *traditional redundancy*.

→ *iterative redundancy* : distribue les tâches à des nœuds de telle sorte que l'on puisse atteindre une certaine fiabilité. Si tous les résultats des tâches attribuées à ces nœuds sont identiques alors on garde le résultat. Sinon il faudra que l'on redistribue la tâche à des nœuds additionnels jusqu'à atteindre la fiabilité voulue. Les auteurs ont montré que cette technique ne requiert pas de connaître la fiabilité des nœuds. Si toutes les machines ont la même fiabilité, il existe un calcul nous indiquant le nombre *d* de répliquas nécessaires pour que le résultat obtenu dépasse un certain seuil de confiance. Les auteurs se placent dans le cas critique où tous les nœuds fautifs sont complices et retournent le même résultat. On a donc deux résultats possibles, le correct et l'incorrect. Les auteurs ont montré que seule la différence de *d* nœuds entre le nombre de bonnes et mauvaises réponses est importante pour atteindre le seuil de confiance voulu.

Ainsi les auteurs proposent une méthode dans laquelle les utilisateurs peuvent paramétrer la valeur *d.*

Les deux dernières techniques ont le défaut de pouvoir être coûteuses au niveau du temps de réponse (dans le cas avec fautes), mais tendent à minimiser le nombre de machines utilisées.

Méthode de sélection des répliquas en fonction de leurs caractéristiques :

Nous allons nous intéresser maintenant à une technique d'allocation de tâches basée sur la probabilité des nœuds d'être fautifs. Cette technique est décrite dans [10]. Dans cette technique des groupes de nœuds sont formés pour exécuter les requêtes. Ces groupes sont formés en prenant en compte un seuil λ_{target} qui correspond à un seuil minimum de fiabilité souhaité pour un groupe. La fiabilité d'un groupe est notée λ, et est calculée en fonction de la réputation de chacun des nœuds. (La réputation des nœuds est initialisée à 1/2, puis à chaque fin de tâche elle est recalculée à l'aide de la formule : $\frac{\text{nombre de réponses correctes} + 1}{\text{nombre de tâches attribuées} + 2}$).

Les algorithmes de formation de groupes sont les suivants :

→ *first-fit* : on classe les nœuds par ordre décroissant de fiabilité. On forme les groupes en regroupant les nœuds (dans l'ordre de la liste formée précédemment) jusqu'à atteindre *λtarget* ou un nombre maximum de nœuds autorisés dans un groupe. Dans le pire des cas on se ramène donc à l'aide de ce nombre maximum à la méthode *fixed-size*. Il y a également un nombre de nœuds minimum autorisés dans chaque groupe.

→ *tight-fit* : cette méthode permet de former des groupes avec le moins de machines possibles et qui ont un λ le plus proche possible de *λtarget*.

→ *random-fit* : on ajoute des nœuds dans un groupe jusqu'à ce que le nombre maximum de nœuds autorisés dans un groupe soit atteint ou lorsque le λ des nœuds du groupe est supérieur ou égal à *λtarget*.

Ces techniques permettent de diminuer le nombre de répliquas utilisés. Elles sont de plus élastiques car pour deux demandes d'exécution de requête, ce n'est pas nécessairement le même sous-ensemble de répliquas qui les exécutera.

L'avantage des méthodes qui ne prennent pas en compte les caractéristiques du réseau, est qu'elles n'ont pas besoin d'effectuer de calculs pour mettre à jour les valeurs de fiabilité des nœuds.

Cependant, les méthodes qui calculent la réputation des nœuds sont plus aptes à retourner la bonne réponse rapidement.

Deux problèmes majeurs peuvent être relevés pour ces deux familles de techniques. Tout d'abord il n'est pas sûr que le résultat retourné au client soit correct. On ne peut que garantir que le résultat sera correct avec une certaine probabilité. De plus, comme indiqué section II.1.b, si des clients interrogent des sous-ensembles de serveurs différents et que leurs requêtes touchent les mêmes objets, cela peut mener à des incohérences.

IV. Efficacité en présence de fautes

Lorsque l'on parle de l'efficacité d'un protocole en présence de fautes on cherche à savoir si les performances de l’algorithme restent bonnes lorsque le nombre de fautes augmente.

Dans cette partie nous allons présenter pourquoi certaines techniques BFT sont peu efficaces en présence de fautes. Puis nous allons présenter des protocoles qui règlent ces problèmes.

IV.1. Inefficacité des protocoles BFT

IV.1.a. Inefficacité de PBFT (et protocoles similaires)

L'article [13] présente deux attaques qui rendent PBFT (ainsi que la plupart des algorithmes reprenant PBFT et donc s'appuyant sur un *primary* pour ordonner les requêtes) non performant. Tout d'abord un *primary* qui reçoit une requête et qui est malicieux peut très bien attendre un temps avant d'envoyer le message *pre-prepare* nécessaire pour ordonner les requêtes sans être pour autant détecté s'il fait attention à ne pas dépasser les délais imposés par les temporisateurs.

Le *primary* peut également ignorer les requêtes émanant des clients. Le client après expiration de son temporisateur va alors diffuser sa requête à tous les répliquas. Ces derniers transfèrent la requête au *primary* tout en la gardant dans une queue et en armant un temporisateur. Le problème est que les répliquas n'arment le temporisateur que pour la requête en tête de leur file d'attente, et non pour les n requêtes se trouvant dedans. Ainsi si tous les répliquas ont la même requête en tête, le *primary* peut attendre le dernier moment pour leur envoyer l'ordre. Si la requête en tête de chaque file est différente il devra envoyer plus de numéros de séquence dans le laps de temps donné par le temporisateur pour ne pas être suspecté. Si maintenant les répliquas mettent un temporisateur sur un ensemble de n requêtes, n étant grand, le débit sera peut être trop important pour le *primary.* Les répliquas le suspecteront alors qu'il est correct.

De plus lors d'un changement de vue PBFT va doubler la valeur d'initialisation de chaque temporisateur. Ainsi si un attaquant effectue un déni de service, que plusieurs changements de vue ont lieu, et que l'on reprend l'exécution avec un *primary* malicieux, dans ce cas le problème évoqué précédemment peut être lourd. En effet, la valeur du temporisateur étant grande, le leader peut répondre très lentement sans jamais être suspecté. Le problème est que PBFT n'a pas de mécanisme permettant de réduire la valeur des temporisateurs une fois les attaques finies.

IV.2.b. Inefficacité des protocoles faisant l’hypothèse d'un nombre fixe de fautes

La plupart des protocoles font des hypothèses sur le nombre de fautes et effectuent alors la réplication en fonction de ce nombre. Le calibrage de ce nombre est très important. S'il est trop faible alors il est possible que le système soit en présence de plus de fautes que prévues et donc que le système rende des résultats erronés. S'il est trop grand par contre le protocole tolérera bien les fautes, mais utilisera plus de ressources que nécessaire et, en fonction du protocole, pourra avoir des performances plus faibles (par exemple à cause du coût du consensus pour ordonnancer les requêtes).

Pour résoudre cela il suffit d'utiliser une méthode probabiliste qui est élastique et prend en compte la fiabilité des nœuds.

IV.2. Protocoles efficaces

Dans la littérature on trouve plusieurs protocoles visant à régler les problèmes d'inefficacité des techniques BFT. Les solutions pour rendre les protocoles BFT sont d'adapter la valeur du temporisateur qui surveille le *primary* en fonction de la stabilité ou l'instabilité du réseau sous-jacent. Une autre solution est de faire l'hypothèse d'un ordonnanceur fiable. Une autre encore est de ne pas utiliser de *primary*, et d'avoir donc une solution décentralisée. Enfin on peut changer très régulièrement de *primary*.

IV.2.a. Adaptation de la valeur du temporisateur

ByzID apporte une idée intéressante pour adapter la valeur du temporisateur en fonction du réseau. En effet, dans ce protocole on utilise un détecteur *anomaly-based*. Ce genre de détecteurs surveille le système et maintient des statistiques par rapport à celui-ci. Ainsi lorsque le détecteur constate que quelque chose a eu lieu qui ne rentre pas dans le cadre de ses précédentes observations il lance une alerte. En utilisant ce genre de détecteur on peut adapter la valeur du temporisateur pour qu'il soit au plus proche des conditions réelles et ainsi détecter très rapidement (en faisant très peu de fausses détections) un nœud qui aurait crashé ou un nœud qui tend à ralentir le système.

Tout comme PBFT le protocole Aardvark [19] possède un mécanisme de changement de vue qui est effectué lorsque le *primary* est suspecté d'être fautif. Cependant, le changement de vue est également appelé lorsque les répliquas trouvent que le débit auquel ils reçoivent les requêtes n'est pas assez élevé, ou lorsque les répliquas se rendent compte que le *primary* n'est pas équitable envers les clients.

Le protocole UpRight [7] a repris les principes de Aardvark en ce qui concerne les changements de vue lorsque le débit baisse.

Ces techniques permettent de régler le problème abordé dans la section IV.1.a.

Remarque :

Le protocole Aardvark permet également de résister aux attaques par déni de service. En effet, le protocole n'utilise que des communications point à point, et donc utilise un NIC (*Network Interface Controller*) différent pour communiquer avec chacun des répliquas. Cela permet de bloquer les entrées des répliquas qui émettent trop de messages par rapport aux autres répliquas. Celles-ci seront à nouveau ouvertes au bout de 10 minutes ou dès que f autres entrées seront bloquées.

IV.2.b. Ordonnancement

Pour s'abstraire des problèmes liés à un *primary* fautif, deux techniques sont utilisées : se reposer sur une entité fiable pour l'ordonnancement, ou effectuer un ordonnancement sans *primary*.

Ordonnancement à l'aide d'une entité fiable :

Certains protocoles comme [10] font l'hypothèse qu'une entité fiable est mise à leur disposition. Ils utilisent cette machine pour effectuer les ordonnancements.

Cette hypothèse n'est pas aberrante (même au sein d'un environnement byzantin). En effet, il n'est pas rare au sein du *cloud* d'avoir des machines fiables utilisées par des administrateurs pour des tâches délicates.

Ordonnancement sans primary :

Le protocole décrit dans [16] permet d'effectuer l'ordonnancement de manière distribuée plutôt que d'utiliser un nœud qui ordonnance toutes les requêtes.

Cette technique a été décrite en section III.1.b. ouvre une réflexion à la création de protocoles décentralisés pour ordonnancer les requêtes.

IV.2.c. Changement périodique du primary

Le protocole Spinning [20] change régulièrement de *primary*. Ce dernier est changé après avoir exécuté un lot de requêtes. A chaque changement de *primary*, le numéro de la vue est également changé : le *primary* d'une vue correspond au nœud d'identifiant (numéro de vue % nombre de répliquas).

Un mécanisme de *blacklist* est également mis en place pour éviter que des *primary* fautifs ne puissent redevenir *primary*. Chaque répliqua possède une *blacklist* contenant des identifiants de répliquas ne pouvant pas être *primary*. Lorsqu'un répliqua augmente le numéro de vue, il vérifie si le *primary* associé à la nouvelle vue est dans la *blacklist* ou non. Si c'est la cas, le répliqua augmente à nouveau le numéro de vue, et ainsi de suite jusqu'à trouver un *primary* hors de la *blacklist*. Suite à l'observation par plusieurs répliquas d'un *primary* fautif, le *primary* nouvellement élu envoie un message à l'ensemble des répliquas. Ces derniers insèrent alors l'ancien *primary* dans la *blacklist.* Les répliquas sont alors en accord avec les serveurs placés dans la *blacklist.* La *blacklist* a une taille maximum de f identifiants. Si un nœud doit être inséré alors que la liste est pleine, le plus ancien est supprimé de cette dernière (stratégie de remplacement FIFO).

Le temporisateur est doublé lorsqu'un nœud devient *primary* à cause d'un *primary* qui était fautif. Dans PBFT le même mécanisme existe. Cependant, ici, lors de phase stable la valeur du temporisateur est divisée par deux autant de fois que nécessaire.

Le protocole Spinning offre ainsi une solution pour parer aux *primary* fautifs.

V. Synthèse

Tableau récapitulatif des principaux algorithmes :

Algorithme	Latence (cas sans faute)		Nombre de répliquas d'accord	Nombre de répliquas d'exécution	élasticité
	Coût	Temps de réponses			
PBFT	$O(f^2)$	5	3f+1	3f+1	Non
Quorum de Aliph	O(f)	2	0	3f+1	Non
Quorum de HQ	O(f)	2 pour les lectures 4 pour les écritures	0	3f+1	Non
[10]	≤ O(f)	2 à 4[1]	1	Dépend de la fiabilité des nœuds et du seuil de confiance que le client souhaite pour sa réponse	Oui
Q/U	O(f)	2	0	4f+1 (mais besoin d'un total de 5f+1 répliquas)	Non
Chain de Aliph	O(f)	3f+2	1	3f+1	Non
Zyzzyva	O(f)	3	1	3f+1	Non
Zlight de AZyzzyva	O(f)	3	1	3f+1	Non
UpRight[2]	$O((u+r)^2)$	7	2u+r+1	u+r+1	Non
BFTCloud	O(f)	3	1	3f+1	Oui
ByzID	O(f)	3	1	2f+1	Non
[14][3]	$O(f^2 + g*f)$	7 (9 si on utilise un firewall)	3f+1	2g+1	Non
[15]	O(g*f)	5 en considérant un seul checkpoint sinon 5+4*nbCheckpoints	3f+1	Entre g+1 et 2g+1	Non
[16]	O(f)	3[4]	2f+1	2f+1	Non
[17]	$O(f^2)$	5	3f+1	3f+1	Oui

1 Dans l'algorithme les auteurs ne prennent pas en compte le client. Mais si on le considère il faut 2 envois de message supplémentaires (un pour l'envoi de la requête du client vers l'ordonnanceur et un pour l'envoi de la réponse de l'ordonnanceur vers le client).

2 Dans UpRight u correspond au nombre de crashs et r au nombre de comportements arbitraires tolérés par le protocole. ($u \geq r$)

3 Dans la suite du tableau f correspond au nombre maximum de fautes au sein des serveurs d'accord, et g correspond au nombre maximum de fautes au sein des serveurs d'exécution.

4 Pour effectuer l'accord les oracles effectuent des diffusion à intervalles réguliers. Le temps de réponse est de 3 si l'on considère une seule diffusion pour effectuer l'accord.

Aardvark	$O(f^2)$	5	3f+1	3f+1	Non
Spinning	$O(f^2)$	5	3f+1	3f+1	Non

On peut alors représenter sur un graphe les différentes techniques présentées. La Figure 5 montre le temps de réponse en fonction du nombre de répliquas utilisés pour la phase d'accord. La Figure 6 quant à elle montre le temps de réponse en fonction du nombre de serveurs utilisés pour l'exécution.

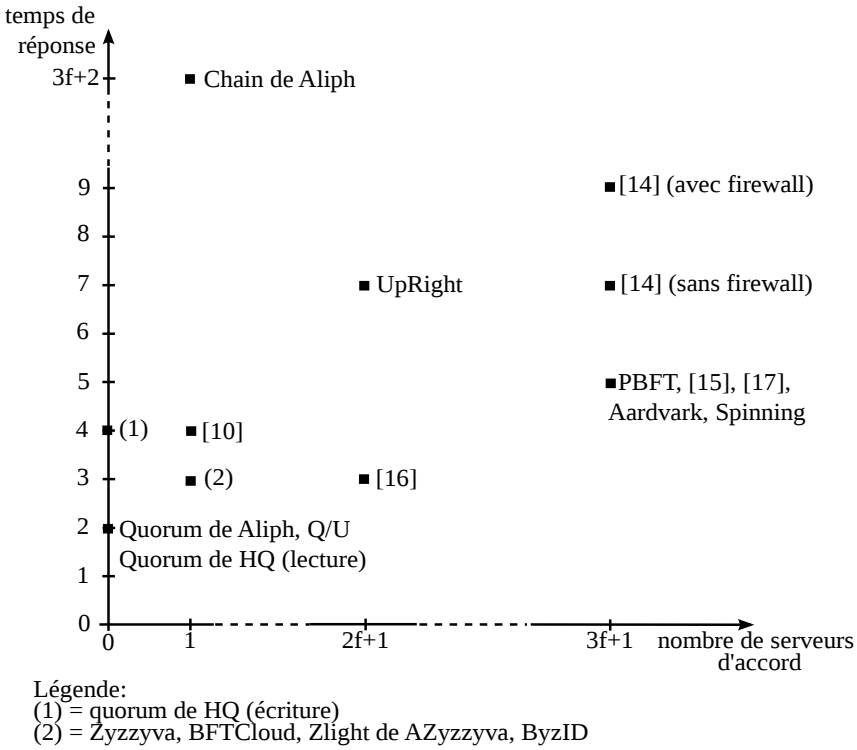

Figure 5 – représentation de la latence des protocoles BFT en fonction du nombre de répliquas utilisés pour la phase d'accord

Sur la Figure 5, le protocole UpRight a été placé en 2f+1. Cela est vrai uniquement si 2u+r = 2f. (avec u le nombre de crashs, et r le nombre de nœuds byzantins, tolérés par le protocole).

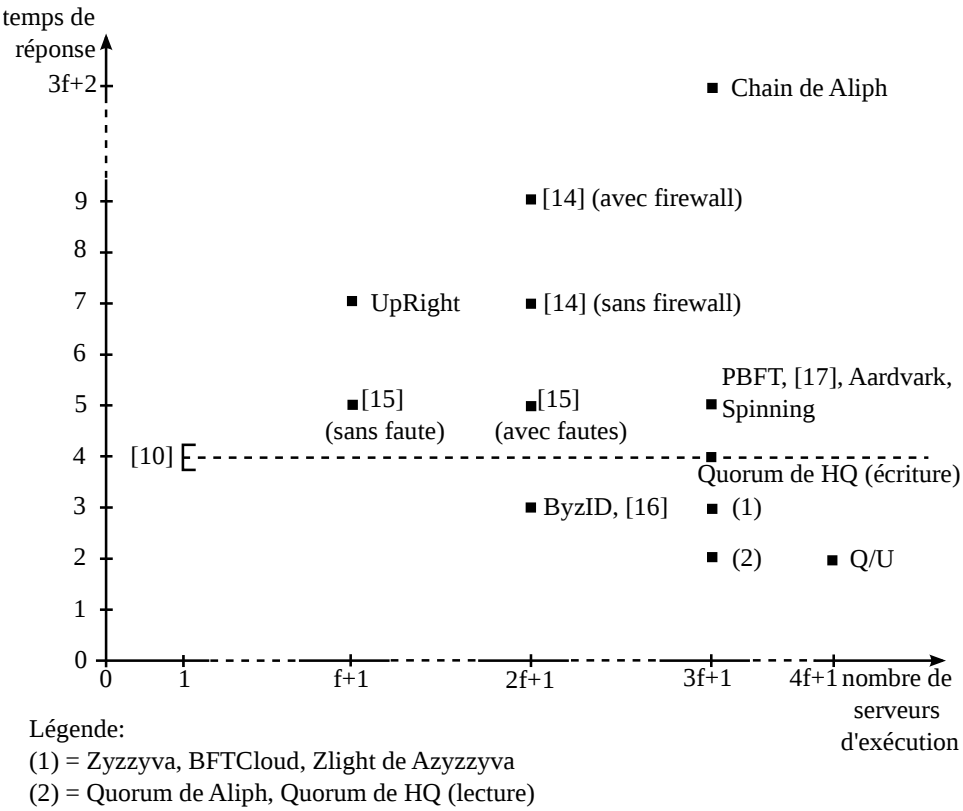

Figure 6 - représentation de la latence des protocoles BFT en fonction du nombre de répliquas utilisés pour l'exécution

De la même manière que précédemment, sur la Figure 6 le protocole UpRight a été représenté comme utilisant f+1 répliquas pour l'exécution. Cela est vrai si l'on a u+r = f.

La demie-droite en pointillés représente le nombre de répliquas qu'utilise le protocole [10] pour son exécution. Vu que cette dernière est basée sur la fiabilité des nœuds, le nombre de répliquas utilisés varie en fonction des propriétés des nœuds disponibles.

VI. Conclusion

Nous avons présenté l'état de l'art des protocoles tolérant les fautes byzantines en considérant les métriques qu'ils essaient d'améliorer.

La recherche a fait des progrès au niveau des protocoles BFT; L'article [21] l'indique d'ailleurs. Cependant cet article établit également qu'il est encore assez rare de voir les protocoles BFT mis en œuvre. Ce dernier se questionne d'ailleurs sur l'utilité des techniques BFT pour parer aux fautes byzantines. Il met en perspective l'utilisation peut-être plus raisonnable des détecteurs de défaillances. En effet, cet article se place dans le *cloud* et indique que cet environnement est assez sûr pour ne pas vouloir payer le coût élevé des phases d'accord pour des événements qui n'arriveront que très rarement. Il prône l'utilisation de phases d'accord pour des environnements moins stables comme le P2P.

L'article indique de plus que fixer le nombre de fautes lorsqu'on est face à un adversaire malicieux n'est pas une hypothèse raisonnable, de même que de se placer dans un environnement où les liens sont équitables et partiellement synchrones.

Le coût du consensus pour ordonner les tâches peut être rédhibitoire et l'hypothèse fixant le nombre de fautes qui peuvent avoir lieu dans le système limite la capacité de résistance aux attaques. Cependant comme nous l'avons dit, le détecteur seul ne semble pas approprié. En effet l'utilisation de ces derniers permet une bonne gestion des fautes mais implique de faire des hypothèses fortes sur la fiabilité de ces derniers. Une bonne alternative est donc d'utiliser les protocoles probabilistes qui affectent des tâches en fonction de la fiabilité des nœuds.

Un autre article [22] présente un ensemble de raisons expliquant que les techniques BFT n'ont pas d'intérêt au sein du *cloud*, mais seraient par contre appropriées pour l'intercloud (*cloud* de *cloud*).

Les raisons citées par ce papier sont les 2 coûts rédhibitoires suivants :

→ 3f+1 répliquas nécessaires pour tolérer les fautes byzantines. Alors que certains composants matériels peuvent permettre d'avoir des architectures dignes de confiance.

→ pour que les fautes soient indépendantes il faut que le code de chaque nœud soit implémenté d'une façon différente, que le système d'exploitation de chaque nœud soit différent… ce qui est coûteux.

De plus les auteurs ajoutent que le *cloud* est surtout sujet à des crashs et non à des fautes byzantines. Généralement les services les plus critiques sont dans des environnements sécurisés et ne nécessitent pas de protocole BFT. L'auteur cite l'exemple de Chubby (service de verrous) qui n'est interrogé au sein de google que par d'autres applications de google. Ces applications ne représentent donc pas une menace pour Chubby.

Cependant on peut critiquer cet article car les 3f+1 répliquas ne sont pas nécessaires si l'on n'utilise pas de consensus pour ordonner les tâches. Et donc si on utilise moins de répliquas on a besoin d'utiliser moins de codes différents. Aussi lorsque l'on se place dans un environnement tel que celui offert par BOINC [24] (*cloud* où les machines sont données de manière volontaire pour effectuer les calculs), les architectures des machines ont de grandes chances d'être hétérogènes, les systèmes d'exploitation différents… Aussi l'exemple donné (chubby) est un système sécurisé et qui effectivement est très peu soumis à des fautes byzantines mais lorsque l'on se place dans le *cloud* où les machines sont données de manière volontaire il n'est pas rare d'être en présence de fautes byzantines.

En prenant en compte ces remarques on peut alors présenter notre perspective de travail :

Notre objectif est d'implémenter une solution probabiliste dans le cloud. Quatre buts sont à atteindre : une faible latence, une utilisation des répliquas de manière élastique, une bonne robustesse face aux fautes, et une solution qui passe à l'échelle (qui puisse avoir un bon débit malgré les sollicitations importantes des clients).

Nous nous intéressons aux méthodes probabilistes. Elles permettent déjà d'avoir un temps de réponse faible, d'avoir un nombre de répliquas minimum. De plus, l'hypothèse d'un ordonnanceur fiable permet la robustesse. Ainsi pour répondre à notre objectif, il faut que notre solution passe à l'échelle.

Parmi les solutions de l'état de l'art peu passe à l'échelle. En effet, soit les solutions utilisent un seul répliqua pour l'ordonnancement des tâches, et dans ce cas ce dernier devient un goulot d'étranglement lorsque le nombre de requêtes augmente. D'autres solutions interrogent tous les répliquas; mais là encore si le nombre de requêtes augmente, l'ensemble des machines constitue un goulot d'étranglement.

On va donc utiliser une hiérarchie d'ordonnanceurs afin que les solutions probabilistes passent à l'échelle.

Les répliquas seront divisés en sous-groupes. Chaque sous-groupe sera sous la responsabilité d'un ordonnanceur fiable. Le but sera donc de déterminer le nombre d'ordonnanceurs que l'on doit utiliser pour maximiser les performances. On devra également étudier quelles stratégies adopter quant à la division des répliquas en sous-groupes.

Parmi ces stratégies on peut citer :

→ stratégie 1 : assembler les nœuds de manière round-robin. On aura alors le même nombre de répliquas dans chaque sous-groupe (à un nœud près).

→ stratégie 2 : regrouper les répliquas en fonction de leur réputation. Chaque ordonnanceur sera alors à la tête d'un ensemble de répliquas dont les réputations sont comprises entre une borne maximum et une borne minimum. Vu que les réputations des nœuds évoluent au cours du temps, les nœuds sont amenés à changer de groupe. Pour les nœuds qui arrivent dans le système on pourra utiliser plusieurs stratégies. On pourra les inclure initialement dans le groupe des nœuds qui ont une bonne réputation ou l'inverse...

On pourra également s'intéresser aux stratégies à appliquer pour faire évoluer les réputations des nœuds. On pourra utiliser une stratégie asymétrique qui augmente la réputation d'un nœud de x lorsque ce dernier a donné une bonne réponse, sinon la diminue de 2x. Une stratégie de gestion de la réputation de manière symétrique pourra aussi être effectuée. La stratégie donnée dans [10] pourra aussi être mise en œuvre.

Pour pouvoir tester notre solution par rapport à l'existant on va effectuer des tests.

Par rapport à la latence les tests à effectuer sont les suivants :
→ latence dans le cas sans faute.
→ Évolution de la latence lorsque le nombre de fautes augmente.

au niveau du nombre de répliquas :
→ nombre moyen de répliquas utilisés dans le cas sans faute
→ évolution du nombre de répliquas utilisés (le système étant basé sur la réputation des nœuds et apprenant cette dernière de manière dynamique, il y a une phase d'apprentissage durant laquelle le nombre de nœuds utilisés sera plus élevé)
→ évolution du nombre de répliquas utilisés en fonction du nombre de fautes

au niveau de la robustesse :
→ débit dans le cas sans faute avec un nombre de clients fixe
→ évolution du débit en fonction du nombre de fautes avec un nombre de clients fixe

au niveau du nombre de clients supportés (passage à l'échelle) :
→ évolution du débit lorsque le nombre de clients augmente
→ évolution du débit lorsque l'on augmente le nombre d'ordonnanceurs

Une autre métrique importante est à considérer vu que l'on utilise une méthode probabiliste. Il faut que l'on détermine le taux de bonnes réponses fournies par le système dans le cas sans faute. Il faut également déterminer l'évolution du taux de bonnes réponses lorsque le nombre de fautes augmente. (Pour cela il nous faudra un témoin correct qui puisse calculer les tâches et comparer la solution qu'il a obtenue avec celle obtenue par notre service répliqué).

L'ensemble de ces tests devront être effectués pour chacune des stratégies de division choisie, et suivant le nombre d'ordonnanceurs choisis.

Notre solution étant probabiliste, on se comparera aux autres méthodes probabilistes existantes. On pourra également se comparer au PBFT qui est l’algorithme de base. Cependant les algorithmes probabilistes et PBFT n'ont pas les mêmes objectifs (la concurrence des écritures est gérée dans PBFT, mais pas dans les méthodes probabilistes). La comparaison est donc non équitable.

Références :

[1] L. Lamport, R. Shostack, and M. Pease. The Byzantine Genereals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382-401, 1982

[2] K. Driscoll, B. Hall, H. Sivencroma, and P. Zumsteg. Byzantine Fault Tolerance, from Theory to Reality, *22nd International Conference, SAFECOMP,* 2003

[3] R. Guerraoui and M. Yabandeh. Independant Faults in the Cloud, *LADIS'10,* 2010

[4] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance, In *3rd Symp. on Operating Systems Design and Impl.,* February 1999

[5] R. Guerraoui, V. Quéma and M. Vukolić. The Next 700 BFT Protocols, *EPFL Technical Report,* 2009

[6] R. Kotla, L. Alvisi, M. Dahlin, A. Clement and E. Wong. Zyzzyva : Speculative Byzantine Fault Tolerance, *SOSP'07,* 2007

[7] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin and T. Riché. UpRight Cluster Services, *SOSP'09,* 2009

[8] J. Cowling, D. Myers, B. Liskov, R. Rodrigues and L. Shrira. HQ Replication : A Hybrid Quorum Protocol for Byzantine Fault Tolerance, *OSDI'06 : 7th USENIX Symposium on Operating Systems Design and Implementation,* 2006

[9] Y. Zhang, Z. Zheng, and M. Lyu. BFTCloud : A Byzantine Fault Tolerance Framework for Voluntary-Resource Cloud Computing, *IEEE Computer Society,* 2011

[10] J. Sonnek, A. Chandra, and Jon B. Weissman. Adaptative Reputation-Based Scheduling on Unreliable Distributed Infrastructures, *IEEE Computer Society,* 2007

[11] S. Duan, K. Levitt, H. Meling, S. Peisert, H. Zhang. ByzID : Byzantine Fault Tolerance from Intrusion Detection, *18th International Conference on Principles of Distributed Systems (OPODIS),* 2014

[12] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter and J. J. Wylie. Fault-Scalable Byantine Fault-Tolerant Services, *SOSP'05,* 2005

[13] Y. Amir, B. Coan, J. Kirsch, J. Lane, and J. Hopkins. Byzantine Replication Under Attack, *IEEE/IFIP International Conference on Dependable Systems and Networks,* 2008

[14] J. Yin, J-P. Martin, A. Venkataramani, L. Alvisi and M. Dahlin. Separating Agreement from Execution for Byzantine Fault Tolerant Services, *SOSP'03,* 2003

[15] A. Agbaria and R. Friedman. A Replication- and Checkpoint-Based Approach for Anomaly-Based Intrusion Detection and Recovery, *ICDCSW'05,* 2005

[16] M. Correia, N. Ferreira Neves and P. Veríssimo. How to Tolerate Half Less One Byzantine Nodes in Practical Distributed Systems, *23rd IEEE International Symposium on Reliable Distributed*, 2004

[17] A. Bessani, J. Sousa and E. Alchieri. State Machine Replication for the Masses with BFT-SMaRT, *Dependable Systems and Networks (DSN)*, 2013

[18] Y. Brun, G. Edwards, J. Young Bang, and N. Medvidovic. Smart Redundancy for Distributed Computation, *31st International Conference on Distributed Computing Systems*, 2011

[19] A. Clement, E. Wong, L. Alvisi, and M. Dahlin. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults, *NSDI'09 : 6th USENIX Symposium on Networked Systems Design and Implementation*, 2009

[20] G. Santos Veronese, M. Correia, A. Neves Bessani, and L. Cheuk Lung. Spin One's Wheels ? Byzantine Fault Tolerance with a Spinning primary, *SRDS'09*, 2009

[21] A. Neves Bessani, M. Correira, P. sousa, N. Ferreira Neves, and P. Verissimo. *Intrusion Tolerance* : The "killer app" for BFT Protocols(?), https://www.net.t-labs.tu-berlin.de/~petr/BFTW3/abstracts/position-bft.pdf

[22]M. Vukolić. The Byzantine Empire in the Intercloud, *ACM SIGACT News*, 2010

[23] C. Pu, A. P. Black, C. Cowan, J. Walpole and C. Consel. A Specialization Toolkit to Increase the Diversity of Operating Systems, *ICMAS Workshop on Immunity-Based Systems*, 1996

[24] D. P. Anderson. BOINC : A system for Public-Resource Computing and Storage, *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, 2004