Vers des protocoles de tolérance aux fautes byzantines efficaces et robustes

Lucas Perronne

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

Spécialité : **Informatique**

Arrêté ministériel : 25 mai 2016

Présentée par

Lucas PERRONNE

Thèse dirigée par **Pr. Sara Bouchenak**

préparée au sein du **Laboratoire d'Informatique de Grenoble**
et de **École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Vers des protocoles de tolérance aux fautes Byzantines efficaces et robustes

Thèse soutenue publiquement le **8 décembre, 2016**,
devant le jury composé de :

Pr. Didier Donsez
Université Grenoble Alpes, LIG, Président
Pr. François Taiani
Université Rennes 1, IRISA, Rapporteur
Pr. Romain Rouvoy
Université Lille 1, LIFL, Rapporteur
MdC. Vania Marangozova
Université Grenoble Alpes, LIG, Invitée
Pr. Sara Bouchenak
INSA Lyon, LIRIS, Directrice de thèse

Remerciements

Je remercie tout d'abord Didier Donsez, Professeur à l'Université de Grenoble 1, pour me faire l'honneur de présider le jury. Mes remerciements s'adressent également à François Taiani, Professeur à l'Université de Rennes 1, et Romain Rouvoy, Professeur à l'Université de Lille 1 qui ont gracieusement accepté d'être les rapporteurs de cette thèse. Merci également à Vania Marangozova, Maître de conférence à l'Université de Grenoble 1, d'avoir accepté d'y assister en tant qu'invitée.

Je souhaite remercier mon encadrante Sara Bouchenak, professeur à l'INSA Lyon, pour m'avoir soutenu pendant ces trois années de thèse, et sans qui je n'aurais pu me plonger dans l'univers de la recherche. Je tiens également à remercier Damian Serrano, qui m'a guidé avec ferveur durant ma première année de thèse.

Mes remerciements s'adressent également à Divya Gupta, pour une collaboration efficace et un travail ayant permis la publication de plusieurs travaux de recherche.

Je remercie chaleureusement l'ensemble du personnel de l'équipe ERODS et de l'école doctorale MSTII, qui a su me conseiller avec discernement sur de nombreux points techniques et administratifs.

Je tiens à remercier Nicolas Palix, Emmanuel Promayon, Pascal Sicard et le personnel de Polytech Grenoble, qui m'ont permis d'enseigner pendant ces deux dernières années, m'octroyant de fait la possibilité de vivre une expérience très enrichissante.

Je remercie mes collègues doctorants et docteurs, pour leur bonne humeur et la motivation qu'ils m'ont apportée lors de ces trois dernières années.

Je remercie ma famille qui n'a jamais cessé de me soutenir, et en laquelle je voue une confiance inébranlable.

Résumé

Au cours de la dernière décennie, l'informatique en nuage (*Cloud Computing*) suscita un important changement de paradigme dans de nombreux systèmes d'information. Ce nouveau paradigme s'illustre principalement par la délocalisation de l'infrastructure informatique hors du parc des entreprises, permettant ainsi une utilisation des ressources à la demande. La prise en charge de serveurs locaux s'est donc vue peu à peu remplacée par la location de serveurs distants, auprès de fournisseurs spécialisés tels que Google, Amazon, Microsoft.

Afin d'assurer la pérennité d'un tel modèle économique, il apparaît nécessaire de fournir aux utilisateurs diverses garanties relatives à la sécurité, la disponibilité, ou encore la fiabilité des ressources mises à disposition. Ces facteurs de qualité de service (QoS pour *Quality of Service*) permettent aux fournisseurs et aux utilisateurs de s'accorder sur le niveau de prestation escompté. En pratique, les serveurs mis à disposition des utilisateurs doivent épisodiquement faire face à des fautes arbitraires (ou byzantines). Il s'agit par exemple de ruptures temporaires du réseau, du traitement de messages corrompus, ou encore d'arrêts inopinés. Le contexte d'informatique en nuage s'est vu néanmoins propice à l'émergence de technologies telles que la virtualisation ou la duplication de machines à états. De telles technologies permettent de pallier efficacement à l'occurrence de pannes via l'implémentation de protocoles de *tolérance aux pannes*.

La tolérance aux fautes byzantines (BFT pour Byzantine Fault Tolerance) est un domaine de recherche implémentant les concepts de duplication de machines à états, qui vise à assurer la continuité et la fiabilité des services en présence de comportements arbitraires. Afin de répondre à cette problématique, de nombreux protocoles furent proposés. Ceux-ci se doivent d'être *efficaces* afin de masquer le surcoût lié à la duplication, mais également *robustes* afin de maintenir un niveau de performance élevé en présence de fautes. Nous constatons d'abord qu'il est délicat de relever ces deux défis à la fois : les protocoles actuels sont soit conçus pour être *efficaces* au détriment de leur *robustesse*, soit pour être *robustes* au détriment de leur *efficacité*. Cette thèse se focalise autour de cette problématique, l'objectif étant de fournir les instruments nécessaires à la conception de protocoles à la fois *robustes* et *efficaces*.

Notre intérêt se porte principalement vers deux types d'attaques (dénis de service) liés à la gestion des requêtes. La première attaque est causée par la corruption partielle d'une requête lors de son émission par un client. La deuxième est provoquée par l'abandon intentionnel d'une requête lors de sa réception par un réplica. Afin de faire face efficacement à ces deux compor-

tements byzantins, plusieurs mécanismes dédiés furent implémentés dans les protocoles de BFT *robustes*. En pratique, ces mécanismes engendrent d'importants surcoûts, et contribuent par conséquent au déclin des performances des protocoles *robustes* face aux protocoles *efficaces*. Ce constat nous permet d'introduire notre première contribution : la définition de plusieurs principes de conception génériques, applicables à de nombreux protocoles de BFT, et destinés à réduire ces surcoûts tout en assurant un niveau de robustesse équivalent.

La seconde contribution de cette thèse illustre ER-PBFT, un nouveau protocole appliquant ces principes de conception au consensus utilisé par PBFT, la référence en matière de tolérance aux fautes byzantines. S'ensuit une évaluation des performances de notre protocole, ainsi qu'une comparaison avec plusieurs protocoles *robustes* de l'état de l'art. Nous démontrons l'efficacité de notre nouvelle politique de robustesse, à la fois en présence de comportements byzantins mais également lors de scénarios sans faute.

La troisième contribution illustre ER-COP, un nouveau protocole orienté à la fois vers l'efficacité et la robustesse, implémentant nos principes de conception sur COP, le protocole de BFT fournissant les meilleures performances à l'heure actuelle dans un environnement sans faute. Nous présentons également une comparaison entre ER-COP et le prototype de COP original, afin d'évaluer le surcoût engendré par l'intégration de notre politique de robustesse. Nous réalisons finalement une évaluation d'ER-COP en présence de divers comportement byzantins, afin de démontrer sa capacité à tolérer l'occurrence de tels événements.

Abstract

Over the last decade, Cloud computing instigated an important switch of paradigm in numerous information systems. This new paradigm is mainly illustrated by the re-location of the whole IT infrastructures out of companies' warehouses. The use of local servers has thus being replaced by remote ones, rented from dedicated providers such as Google, Amazon, Microsoft.

In order to ensure the sustainability of this economic model, it appears necessary to provide several guarantees to users, related to the security, availability, or even reliability of the proposed resources. Such quality of service (QoS) factors allow providers and users to reach an agreement on the expected level of dependability. Practically, the proposed servers must episodically cope with arbitrary faults (also called byzantine faults), such as incorrect/corrupted messages, servers crashes, or even network failures. Nevertheless, the Cloud computing environment encouraged the emergence of technologies such as virtualization or state machine replication. These technologies allow cloud providers to efficiently face the occurrences of faults through the implementation of *fault tolerance protocols*.

Byzantine Fault Tolerance (BFT) is a research area involving state machine replication concepts, and aiming at ensuring continuity and reliability of hosted services in presence of any kind of arbitrary behaviors. In order to handle such threat, numerous protocols were proposed. These protocols must be *efficient* in order to counterbalance the extra cost of replication, and *robust* in order to lower the impact of byzantine behaviors on the system performance. We first noticed that tackling both these concerns at the same time is difficult : current protocols are either designed to be *efficient* at the expense of their *robustness*, or *robust* at the expense of their *efficiency*. We tackle this specific problem in this thesis, our goal being to provide the required tools to design both *efficient* and *robust* BFT protocols.

Our focus is mainly dedicated to two types of denial-of-service attacks involving requests management. The first one is caused by the partial corruption of a request transmitted by a client. The second one is caused by the intentional drop of a request upon receipt. In order to face efficiently both these byzantine behaviors, several mechanisms were integrated in *robust* BFT protocols. In practice, these mecanisms involve high overheads, and thus lead to the significant performance drop of *robust* protocols compared to *efficient* ones. This assessment allows us to introduce our first contribution : the definition of several generic design principles, applicable to numerous BFT protocols, and aiming at reducing these overheads while maintaining the same level of robustness.

The second contribution introduces ER-PBFT, a new protocol implementing these design principles on PBFT, the reference in terms of byzantine fault tolerance. We evaluate the performance of our protocol, and compare it with several *robust* BFT protocols, in order to demonstrate the efficiency of our new robustness policy, both in fault-free scenarios and in presence of byzantine behaviors.

The third contribution highlights ER-COP, a new BFT protocol dedicated to both efficiency and robustness, implementing our design principles on COP, the BFT protocol providing for now the best performances in a fault-free environment. We also present a comparison between ER-COP and the original COP implementation, in order to evaluate the additional cost introduced by the integration of our robustness policy. Finally, we evaluate ER-COP in presence of various byzantine behaviors, in order to demonstrate its ability to handle such events.

Table des matières

Chapitre 1

Introduction

Sommaire

1.1 Contexte et Motivation

L'apparition de la première vague de services hébergés sur le web remonte aux années 2000. C'est dans cet environnement que l'informatique en nuage (*Cloud Computing*) prit son essor. Le paradigme d'informatique en nuage repose sur la location et l'exploitation des capacités de stockage et de calcul de serveurs informatiques distants. Ces serveurs sont loués à la demande, en fonction des besoins exprimés par les clients. Le principal avantage d'un tel paradigme repose sur sa grande souplesse : le client peut se voir par exemple offrir la capacité de gérer lui-même son serveur, ou encore la possibilité de bénéficier de surcouches logicielles afin d'en faciliter l'utilisation [1, 56]. Une fois la délocalisation des services effectuée, les clients bénéficient de nombreux avantages, en particulier l'adaptation automatique de la quantité de ressources allouées. Les clients se voient ainsi facturés en fonction de l'utilisation effective des services hébergés [3]. L'adoption de ce paradigme s'est vue facilitée par une augmentation considérable de la puissance de calcul des équipements informatiques, permettant ainsi aux fournisseurs (*Cloud providers*) de proposer des tarifs de plus en plus compétitifs. En pratique, l'informatique en nuage repose sur l'adoption de technologies telles que la virtualisation, l'architecture orientée service (SOA), ou encore la parallélisation et la distribution des calculs.

Grâce à l'émergence de l'informatique en nuage, une utilisation plus générique de systèmes d'information distribués s'est rapidement propagée, tels que les réseaux sociaux, infrastructures bancaires, etc. Gérer de tels systèmes est rapidement devenu une tâche complexe. Ceux-ci impliquent bien souvent de nombreuses entités hétérogènes qui se doivent de coopérer dans des environnements difficiles à contrôler [17, 29, 61]. Il n'en demeure pas moins nécessaire que ces systèmes garantissent un haut niveau de disponibilité, malgré la fiabilité relative de l'environnement dans lequel ils se voient déployés. Il est en pratique très fréquent que les services hébergés sur le nuage soient victimes de pannes [2, 15, 35, 53]. Celles-ci peuvent se révéler bénignes [42], ou au contraire entraîner de fortes pertes financières, influençant à la fois les clients et les fournisseurs de services. Les pannes peuvent être causées par des phénomènes variés tels des erreurs logicielles ou des imprévus météorologiques : en 2012 par exemple, de violentes tempêtes en Virginie du Nord perturbèrent temporairement le service EC2 d'Amazon Web Services, entraînant des périodes d'indisponibilité sur les sites web de Reddit, Foursquare, Pinterest et GitHub. Des événements en apparence bénins, telle l'application d'une mise

à jour, peuvent déclencher l'occurrence d'une panne. Déterminer en amont les causes derrière chaque période d'indisponibilité n'est pas toujours possible. Les services hébergés sur le nuage doivent également faire face à des cyber-attaques, ou dénis de services, volontairement perpétrés afin de nuire aux utilisateurs.

La duplication de machine à états est une manière efficace de renforcer la fiabilité d'un système dans des environnements peu fiables [45, 63]. Cette technique consiste en l'utilisation de plusieurs copies du système, appelées réplicas, et permet de masquer l'occurrence de fautes de manière distribuée. Une telle technique permet de fournir une réponse adéquate à toute sortes de fautes, du crash à l'erreur logicielle, dès lors que la proportion de réplicas corrects demeure suffisante. Afin de bénéficier des propriétés fournies par la duplication de machines à états, le système doit répondre à plusieurs contraintes. Il doit tout d'abord être implémenté sous forme de machine à états ; autrement dit son état doit évoluer en fonction des instructions à exécuter. Il se doit également d'être déterministe, il fournira donc toujours la même réponse à une même requête. Ces pré-requis permettent de garantir que plusieurs copies d'un système répondront de la même manière lors de l'exécution d'une requête, tout en conservant des états cohérents. Il existe en pratique différentes manières d'implémenter la duplication. Celle-ci peut être active [44], auquel cas tous les réplicas exécutent manuellement les requêtes, ou passive [6], auquel cas un sous-ensemble de réplicas exécute les requêtes, puis transmet les modifications apportées aux réplicas restants. Le nombre de réplicas, associé à la manière dont ceux-ci communiquent sont les deux principaux facteurs permettant de déterminer quels types de fautes se verront tolérer. Ainsi, réduire le nombre de réplicas ou de messages échangés permettra au système d'être plus performant, mais le privera de sa capacité à tolérer certains types de fautes.

Cette thèse se focalise sur la tolérance aux fautes byzantines, ou arbitraires. Ce modèle de fautes générique assume la possible occurrence d'un quelconque comportement ne répondant pas aux spécifications du système. Un tel modèle considère par exemple l'occurrence d'événements tels l'arrêt brutal d'un réplica, l'envoi de réponses corrompues, ou même la prise de contrôle totale d'un réplica en cas d'attaque [13]. Nous nous focalisons donc sur les protocoles de duplication de machines à états capables de tolérer les fautes byzantines (également appelés protocoles de BFT, pour *Byzantine Fault Tolerance*) [4, 21, 24, 28, 37, 43, 59]. De tels protocoles ont pour dessein d'assurer la cohérence entre les états de chaque réplica dans un environnement byzantin, tout en conservant la capacité de progresser vers des états futurs. Cette cohérence entre réplicas est capitale afin d'assurer la pérennité du système dupliqué, il

serait par exemple rédhibitoire qu'un compte bancaire se voit attribuer des soldes différents selon les réplicas. Dans une même optique, le compte doit demeurer accessible afin d'autoriser des opérations de débit ou de crédit. D'une manière plus formelle, les protocoles de tolérance aux fautes byzantines se doivent d'assurer deux propriétés cruciales : (i) la propriété de *sûreté* (*safety*), qui stipule que l'état de tout réplica correct doit évoluer de manière cohérente, et (ii) la propriété de *vivacité* (*liveness*), qui stipule que chaque requête correcte doit finir par se voir exécutée, assurant ainsi la capacité du système à progresser.

L'utilisation pratique de tels protocoles reste marginale [25]. La duplication introduit tout d'abord un surcoût non négligeable comparé à un même système non dupliqué ; plus encore lorsqu'elle se doit d'offrir une tolérance aux fautes byzantines. Le surcoût dont il est question provient des multiples échange de messages entre réplicas. Ces messages se doivent d'être systématiquement authentifiés via l'utilisation de primitives cryptographiques dédiées, afin de garantir l'évolution cohérente de l'état des réplicas dans un environnement hostile. Les protocoles de BFT se doivent donc d'être *efficaces* pour masquer au mieux ce surcoût, tout en garantissant les propriétés de *sûreté* et de *vivacité*. D'autre part, ces protocoles doivent également être *robustes*, c'est-à-dire capables de fournir des performances acceptables en présence de comportements byzantins. Par conséquent, la capacité d'assurer les propriétés théoriques de *sûreté* et de *vivacité* reste insuffisante si le débit fourni en pratique par le protocole chute drastiquement lors d'une attaque [7, 10, 26].

1.2 Défis Scientifiques

La tolérance aux fautes byzantines a suscité l'intérêt de nombreux chercheurs durant les 15 dernières années. La plupart des contributions proposées peuvent être catégorisées selon deux principaux objectifs. Le premier consiste en l'amélioration des performances en absence de faute [14, 27, 37, 43, 65, 70], autrement dit parfaire l'efficacité des protocoles de BFT dans un environnement qui leur est favorable. Nous ferons par la suite référence aux protocoles de cette catégorie en tant que protocoles de BFT *efficaces*. Le second objectif est à contrario destiné à limiter l'impact des comportements byzantins sur les performances du système [7, 10, 26, 71], soit parfaire la robustesse des protocoles de BFT dans un environnement qui leur est hostile. Nous ferons par la suite référence à ces protocoles en tant que protocoles de BFT *robustes*.

Afin de tolérer efficacement l'occurrence de comportements byzantins de plus en plus agressifs, les protocoles *robustes* partagent tous le même objectif : limiter l'impact que des entités malveillantes pourraient engendrer sur les

performances du système dupliqué. Afin de répondre à cette problématique, plusieurs propositions mettent en scène des protocoles de BFT *robustes*, renforcés par de nombreux mécanismes de tolérance spécifiques. Au cours de la dernière décennie, de nouveaux comportements byzantins furent isolés, et plusieurs solutions proposées afin de leurs faire face efficacement. Ces comportements byzantins impliquent l'occurrence d'événements telles que l'introduction intentionnelle de délais (*pre-prepare delay*) [7], la saturation de bande passante (*flooding*) [10], ou encore une corruption partielle des messages [26]. Autant de nouveaux mécanismes à intégrer pour maintenir un niveau de performance satisfaisant. Inévitablement, ces mécanismes engendrent un surcoût supplémentaire, creusant l'écart de performance entre protocoles *efficaces* et protocoles *robustes* en absence de faute. Cette contradiction représente le cœur du problème abordé, nous cherchons donc à limiter les surcoûts engendrés par l'intégration de mécanismes de robustesse afin de permettre le design de protocoles à la fois *robustes* et *efficaces*.

Afin de répondre à cette problématique, notre attention s'est portée sur plusieurs attaques liées à la gestion des requêtes transmises par les clients. Parmi ceux-ci l'attaque MAC (Message Authentification Code) [26], qui s'est vue responsable de l'abandon des codes d'authentification de messages au profit de l'utilisation de signatures digitales par les protocoles *robustes*. L'analyse expérimentale de l'impact de ce changement de paradigme cryptographique nous a permis d'isoler l'importance de l'utilisation des codes d'authentification de messages afin d'assurer un haut niveau de performance. Un des enjeux étant dès lors de supporter l'occurrence des attaques considérées en évitant de recourir à l'utilisation systématique de signatures digitales.

1.3 Contributions

Cette thèse se focalise sur les protocoles de duplication de machines à états satisfaisant les propriétés de tolérance aux fautes byzantines. Nous débutons ce manuscrit par une présentation détaillée de l'état de l'art, nécessaire à la mise en exergue des quatre contributions suivantes :

1. **La définition de principes de conception génériques permettant le design de protocoles à la fois *efficaces* et *robustes*** : Ces principes facilitent la conception de protocoles *efficaces* et *robustes*, et sont également applicables à de nombreux protocoles de BFT existants, leurs octroyant ainsi la capacité de tolérer plus efficacement la présence de comportements byzantins. Ces principes de conception permettent par ailleurs aux protocoles auxquels ils sont appliqués d'afficher un niveau

de performance supérieur aux protocoles *robustes* contemporains.

2. **La mise en œuvre d'un nouveau protocole : ER-PBFT** : Nous illustrons l'impact de nos principes de conception sur le consensus utilisé par PBFT, la référence en matière de tolérance aux fautes byzantines [21]. le design de ER-PBFT nous permet ainsi de valider la pertinence de notre politique de tolérance aux pannes en pratique.
3. **La conception d'ER-COP : un amalgame *efficace* et *robuste*** : A l'heure actuelle, COP est le protocole de BFT fournissant les meilleures performances dans un environnement sans faute. Comme nombre de ses pairs, ce protocole privilégie l'optimisation des performances au détriment de sa robustesse, il est donc vulnérable face à l'occurrence de comportements byzantins. Nous avons choisi d'implémenter ER-COP afin de démontrer la fiabilité de nos mécanismes de tolérance aux pannes tout en évaluant leurs surcoûts face à l'implémentation originale de COP.
4. **L'évaluation comparative et expérimentale de plusieurs protocoles de BFT** : De nombreuses évaluations sont présentées au cours du manuscrit. Dans un premier temps, nous confrontons analytiquement plusieurs protocoles de l'état de l'art. Par la suite, nous comparons nos nouveaux protocoles en présence des diverses attaques considérées, ainsi qu'aux protocoles les plus *robustes* de l'état de l'art.

1.4 Organisation du Manuscrit

La suite du document est organisée sous différents chapitres, brièvement détaillés ci-dessous :

Chapitre 2 : Etat de l'Art. Ce chapitre introduit l'ensemble des concepts nécessaire à la compréhension du manuscrit. Parmi ceux-ci les concepts de duplication de machines à états et de tolérance aux fautes byzantines. S'en suit une présentation plus approfondie des principales directions de recherches empruntées au cours de la dernière décennie dans le domaine de la tolérance aux fautes byzantines.

Chapitre 3 : Problèmes Ouverts en Tolérance aux Fautes Byzantines. Ce chapitre expose la problématique abordée. Il décrit la nature des comportements byzantins considérés, ainsi que les solutions proposées par l'état de l'art afin de leur faire face. Nous évaluons de manière pratique le surcoût associé à ces solutions afin de démontrer la nécessité d'adapter les politiques de robustesse existantes.

Chapitre 4 : Principes de Conception de Protocoles *Efficaces* et *Robustes*. Nous proposons dans ce chapitre nos solutions dédiées, sous la forme de plusieurs principes de conception génériques. L'objectif de cette contribution est de faciliter le design de protocoles de tolérances aux fautes byzantines à la fois *efficaces* et *robustes*.

Chapitre 5 : Conception du protocole ER-PBFT. Ce chapitre présente ER-PBFT, un nouveau protocole de tolérance aux fautes byzantines implémentant nos principes de conception au consensus original utilisé par le protocole PBFT [21].

Chapitre 6 : Evaluation du protocole ER-PBFT. Ce chapitre est dédié à l'évaluation de notre protocole ER-PBFT. Nous effectuons plusieurs comparaisons face aux protocoles de BFT *robustes* existants, afin de démontrer l'efficacité et la robustesse d'ER-PBFT. Nous confrontons par la suite notre protocole aux diverses attaques considérées dans le chapitre 4.

Chapitre 7 : Conception du protocole ER-COP. Ce chapitre présente ER-COP, un nouveau protocole de BFT implémentant nos principes de conception, ainsi que de nombreuses optimisations dédiées à la performance. ER-COP s'inspire respectivement des concepts de parallélisation utilisés par le protocole COP, proposé par Behl et al. [16].

Chapitre 8 : Evaluation du protocole ER-COP. Dans ce chapitre, nous comparons ER-COP au protocole COP original, afin d'évaluer le surcoût de notre politique de robustesse. Finalement, une évaluation d'ER-COP en présence de divers comportements byzantins nous permet de démontrer la capacité de notre protocole à tolérer ces attaques.

Chapitre 9 : Conclusions et Perspectives. Pour clore ce manuscrit, nous dressons un bilan général, puis nous présentons plusieurs perspectives envisageables quant à la poursuite de ces travaux.

Chapitre 2

Etat de l'Art

Sommaire

Nous introduisons dans ce chapitre les concepts de duplication de machines à états et de tolérance aux fautes byzantines (section 2.1). Ces concepts sont mis en pratique via des protocoles de tolérance aux pannes dont la fonction consiste à masquer l'occurrence de fautes arbitraires. S'en suit une caractérisation des différentes contributions de l'état de l'art. Nous répartissons ces contributions selon leurs deux principaux objectifs : l'optimisation des performances en absence de faute (cf. section 2.2.1), et la dépréciation de l'impact des comportements byzantins (cf. section 2.2.2). Nous présentons les principales caractéristiques de nombreux protocoles en détail, ainsi que les directions de recherches empruntées dans le contexte générique de la tolérance aux fautes byzantines.

2.1 Contexte

Nous présentons dans ce chapitre les concepts de duplication de machines à états et de tolérance aux fautes byzantines. Nous introduisons d'abord le concept de machine à états, ainsi que son utilisation pratique dans le cadre des protocoles de duplication. Nous décrivons par la suite le problème du consensus, qui témoigne des principaux enjeux abordés par la duplication, ainsi que son adaptation au modèle de fautes byzantines. Nous clôturons ce chapitre par l'illustration des concepts associés au protocole PBFT [21], le premier protocole destiné à fournir une solution pratique et efficace face à l'occurrence de fautes byzantines.

2.1.1 Duplication de Machines à Etats

La duplication de machines à états (SMR pour *State Machine Replication*) est une technique permettant l'implémentation de services tolérants les pannes via leurs duplications sur un ensemble de serveurs [45, 63, 64]. Spécifiquement, plusieurs copies d'un service sont déployées sur différents nœuds qui communiquent entre eux par l'intermédiaire de protocoles dédiés. Cette approche repose essentiellement sur la coopération des différents réplicas, connectés par un réseau de communications. Une telle technique permet de masquer l'occurrence de fautes en tout genre, afin d'assurer le disponibilité du service. Néanmoins, la duplication de machine à états à elle seule ne permet pas d'assurer une tolérance aux pannes ; elle doit en effet être associée à un protocole de communication, chargé d'assurer la cohérence entre les états des réplicas. Le protocole de communication, associé au nombre de réplicas utilisés permet de définir le degré de tolérance du système.

2.1.1.1 Machines à Etats

Une machine à états consiste en la représentation d'un système sous forme d'automate. Une telle représentation facilite la modélisation de nombreux systèmes, et permet d'en simplifier l'étude. Le fonctionnement d'une machine à états repose sur deux concepts essentiels : celui d'*état* et celui de *transition*. Un *état* décrit la configuration d'un système. Une *transition* est une séquence d'instructions à exécuter lorsqu'une condition particulière est remplie, ou lorsqu'un événement spécifique survient. Une machine à états est donc caractérisée par un ensemble d'états et un ensemble de transitions (cf. figure 2.1).

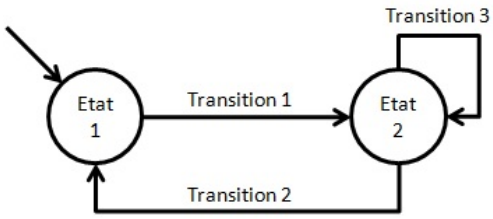

FIGURE 2.1 – Illustration d'une machine à états, composée de deux états distincts

Une machine à états peut être utilisée pour modéliser des systèmes déterministes ou non. Dans le cadre d'une tolérance aux pannes via duplication, notre focus se tourne vers les machines à états déterministes. Une machine de ce type empruntera toujours la même transition lors de l'occurrence d'un événement spécifique depuis un état donné. Cette caractéristique permet de garantir la cohérence des états de plusieurs copies du système lorsque ceux-ci empruntent les mêmes transitions depuis les mêmes états.

2.1.1.2 Vue d'Ensemble

La duplication de machines à états consiste donc en l'utilisation de plusieurs copies d'un service (cf. figure 2.2). Chacune de ces copies, appelée réplica, est implémentée sous la forme d'une machine à états déterministe. Le nombre de réplicas requis dépend de la nature des fautes à tolérer, ainsi que de leur nombre d'occurrences simultanées.

Considérons N le nombre total de réplicas présents dans le système. Intuitivement, la duplication de machines à états ne saurait tolérer plus de $N-1$ réplicas fautifs simultanément. En pratique, pour un nombre f de réplicas fautifs, le nombre total de réplicas N dépend de la nature des fautes à tolérer. Par exemple, $N = 2f+1$ réplicas sont suffisants pour tolérer l'arrêt inopiné (*crash*) de f réplicas [49], alors que $N = 3f + 1$ réplicas sont nécessaires pour tolérer

l'occurrence de f fautes byzantines [21, 22, 49]. Ainsi, tolérer l'occurrence de plusieurs fautes simultanément requiert l'utilisation de davantage de réplicas. Dans le contexte de la duplication de machines à états, le nombre total de réplicas N est fonction du nombre maximal de réplicas fautifs simultanément f.

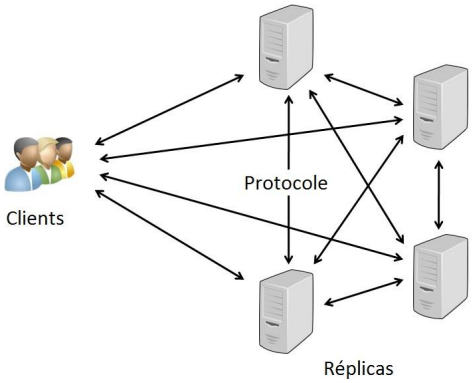

FIGURE 2.2 – Duplication de machines à états. Le système est composé de 4 réplicas (implémentant des machines à états déterministes) et d'un protocole de communication

Afin d'assurer l'indépendance de l'occurrence de fautes entre réplicas, il est préconisé de recourir au paradigme de *N-version programming* [11, 12, 23]. Ce paradigme implique le recours à différentes implémentations du protocole utilisé sur chacune des réplicas. Une telle technique est destinée à disséminer l'impact d'hypothétiques erreurs de programmation, afin d'éviter à un bug de se voir déclencher sur plusieurs réplicas à la fois. Dans une même optique, l'utilisation de matériels divers, associés à différents systèmes d'exploitation, contribue à minimiser l'improbable occurrence de plusieurs fautes simultanément. Lorsqu'un réplica se voit victime de l'occurrence d'une faute, plusieurs mécanismes lui permettent de retrouver un état cohérent, parmi lesquelles la mise en place de points de sauvegarde (*checkpoints*), ou encore le partage de l'historique des événements locaux (*local histories*) de chaque réplica [22].

L'état du service dupliqué évolue suite à la réception de requêtes, transmises par les utilisateurs (également nommés clients). Ces requêtes engendrent l'exécution de séquences d'instructions, qui permettent aux réplicas d'évoluer vers des états futurs. Afin d'assurer la pérennité du système dupliqué, la duplication de machines à états doit assurer deux propriétés distinctes : la *sûreté* (safety) et la *vivacité* (liveness), définie ci-dessous :

— **Sûreté.** L'état de chaque réplica correct doit évoluer en adéquation avec l'état des autres réplicas corrects. Le système dupliqué dans sa globalité doit donc évoluer comme le ferait un même système non-dupliqué exempt de faute.

— **Vivacité.** Toute requête instanciée par un utilisateur correct doit être exécutée. L'indubitable exécution de chaque requête valide permet au système dupliqué de conserver sa capacité à progresser dans des états futurs.

Plus spécifiquement, Schneider détermine que la duplication de machines à états doit répondre à trois impératifs [64], nécessaires afin d'assurer une évolution cohérente de l'état des différents réplicas ; (i) La cohérence de l'état partagé initialement entre chaque réplica se doit d'être garantie. (ii) Chaque réplica doit générer le même résultat suite à l'exécution d'une même instruction. (iii) Chaque réplica doit exécuter les instructions dans le même ordre. Afin de forcer l'exécution des requêtes dans le même ordre, nombre de protocoles requièrent la présence d'un réplica spécifique, dénommé *primary* ou *leader*. Ce *primary* sera tout d'abord chargé d'associer à chaque requête l'ordre dans lequel elle se devra d'être exécutée, puis diffusera cette information aux autres réplicas. S'accorder sur la validité de cet ordre d'exécution représente l'essence même du problème, la diffusion d'une telle information requiert donc l'utilisation d'une primitive de communication dédiée, nommée diffusion à ordre total (*total-order broadcast* ou *atomic broadcast*) [30, 39]. Cette primitive doit assurer la réception fiable des messages par tous les participants. Pour ce faire, elle doit fournir plusieurs propriétés : (i) Si un réplica correct diffuse un message, tous les réplicas corrects recevront ce message. (ii) Si un réplica correct exécute une séquence d'instructions suite à la réception d'un message, tous les réplicas corrects exécuteront cette séquence d'instructions. (iii) Chaque séquence d'instructions n'est exécutée qu'une seule fois, et seulement suite à la réception du message associé. (iv) Si deux réplicas $r1$ et $r2$ exécutent deux séquences d'instructions $i1$ et $i2$ suite à la réception de deux messages distincts $m1$ et $m2$, $r1$ n'exécute $i1$ avant $i2$ si et seulement si $r2$ exécute lui aussi $i1$ avant $i2$.

Il existe deux approches fondamentales quant à l'exécution des instructions par les réplicas. Dans le premier cas, tous les réplicas exécutent eux-même les requêtes, ce type de duplication est appelé duplication *active* (cf. figure 2.3) [44]. Dans le second cas, un sous-ensemble de réplicas exécute les requêtes, puis transmet les modifications apportées aux réplicas restants, il s'agit de duplication dite *passive* (cf. figure 2.4) [6]. L'utilisation de la duplication passive peut se révéler problématique selon la nature des fautes auxquelles le système est confronté. Si par exemple un unique réplica est responsable de l'éxécution

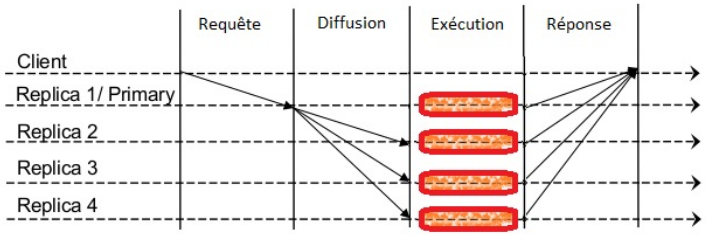

FIGURE 2.3 – Exemple de duplication de machines à états active

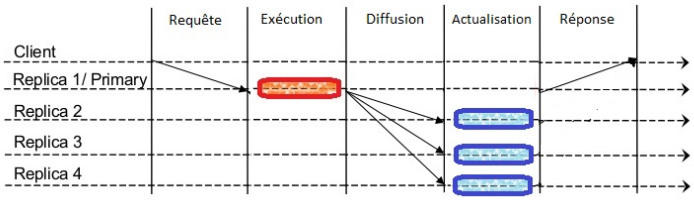

FIGURE 2.4 – Exemple de duplication de machines à états passive

des requêtes, et décide de transmettre des modifications erronées aux autres réplicas, la cohérence du système tout entier se voit remise en question. Nous discutons ce genre de comportements byzantins en détail dans la section 3.1.

2.1.1.3 Le Problème du Consensus

Nous avons mentionné dans la section précédente la nécessité d'une primitive de communication dédiée à la diffusion de l'ordre d'exécution des requêtes. En pratique, le choix et la diffusion de l'ordre dans lequel les requêtes doivent être exécutées peut être assimilé au problème du *consensus*. Ce problème représente un challenge fondamental dans le contexte générique des systèmes distribués : plusieurs processus (il s'agit dans notre cas des réplicas) doivent s'accorder sur le choix d'une valeur unique. Afin d'assurer la fiabilité d'un protocole s'attaquant au problème du consensus, il doit fournir les propriétés suivantes : (i) la valeur choisie doit avoir été proposée par un réplica correct, (ii) deux réplicas corrects ne peuvent pas choisir une valeur différente, (iii) un réplica correct ne se prononce qu'en faveur d'une unique valeur, (iv) une valeur doit être choisie à terme.

Le théorème énoncé par Fischer, Lynch et Patterson [34], connu en tant que *FLP impossibility result* détermine que la résolution du problème de consensus est impossible dans un environnement asynchrone en présence d'un processus fautif. Ce constat est une conséquence de l'incapacité à différencier un processus fautif d'un réseau défectueux. Considérons par exemple un processus p

en attente d'un message m qui lui est nécessaire pour progresser dans un état futur. Il est impossible pour p de déterminer si l'émetteur de m est fautif, ou si le réseau est responsable de l'absence de m. Afin de résoudre ce problème, p peut soit décider d'attendre davantage, soit décider de choisir une valeur sans le message m. Quelle que soit la décision prise par p, les propriétés nécessaires pour garantir le succès du consensus ne sont plus respectées. Afin d'apporter des solutions pratiques au problème du consensus, de nouvelles hypothèses relatives à la fiabilité du réseau et à la nature des fautes rencontrées furent considérées [33, 50]. Les premières tentatives pour répondre au problème de consensus d'un point de vue pratique furent proposées il y a de cela 25 ans, dans le contexte des bases de données distribuées [36, 69]. Depuis, de nombreux travaux on été effectués dans ce domaine, parmi lesquels les protocoles de la famille Paxos tolérants les arrêts inopinés (*crashs*)[19, 46, 54], et les protocoles dédiés à la tolérance aux fautes byzantines [4, 21, 28]. C'est vers cette seconde catégorie de protocoles que se tourne notre intérêt.

2.1.2 Tolérance aux Fautes Byzantines

Alors que la duplication offre une solution adéquate pour assurer un haut degré de fiabilité dans les services d'informatique en nuage, la nature des menaces auxquelles ceux-ci se voient confrontés ne cesse de se diversifier. Il est primordial pour les fournisseurs de garantir le fonctionnement des services hébergés quelque soit la nature des pannes rencontrées. Ainsi, il est nécessaire de disposer de mécanismes permettant de tolérer l'occurrence de fautes logicielles, *crashs*, pertes de messages ou encore de comportements malveillants remettant en cause la qualité de service fournie par le système. Alors que les protocoles comme Paxos [47] permettent de faire face à des fautes bénignes (*fail-stop*) [48, 58], tolérer un panel de fautes plus large nécessite le recours à une classe de protocoles particuliers, nommés protocoles de tolérance aux fautes byzantines. Ces protocoles sont en effet destinés à assurer la pérennité du service dupliqué face à toutes sortes de dysfonctionnements. Nous ferons par la suite référence à ces dysfonctionnements en tant que fautes arbitraires, ou byzantines.

2.1.2.1 Le Problème des Généraux Byzantins

Le problème abordé par ces protocoles fut présenté en 1982 par Lamport sous l'appellation de *problème des généraux byzantins* [50]. Il fait référence à une instance particulière du problème de consensus et s'énonce de la manière suivante : A l'orée d'une bataille, tous les généraux d'une armée doivent s'accorder sur une décision commune pour remporter la victoire : attaque ou

retraite. Ceux-ci ne peuvent communiquer que par l'intermédiaire de messagers, et un certain nombre de généraux peuvent s'avérer être des traîtres, dont l'objectif est de fausser le plan de bataille. Concrètement, considérons qu'un premier général loyal estime que l'attaque doit être menée, et qu'un second général loyal estime que la retraite serait préférable. Si un troisième général malveillant se prononce en faveur de l'attaque auprès du premier, et en faveur de la retraite auprès du second, seul le premier général attaquera, alors que les deux généraux loyaux penseront s'être accordés sur la marche à suivre. Ce problème est d'autant plus difficile à résoudre compte tenu du mode de communication par messagers, car ceux-ci peuvent potentiellement échouer à délivrer les messages, ce qui ne serait pas le cas si les généraux s'accordaient de vive voix. La figure 2.5 illustre une instance du problème des généraux byzantins impliquant trois généraux : si un général malveillant remet en cause la parole d'un autre général, il est impossible pour le troisième de déterminer qui dit la vérité.

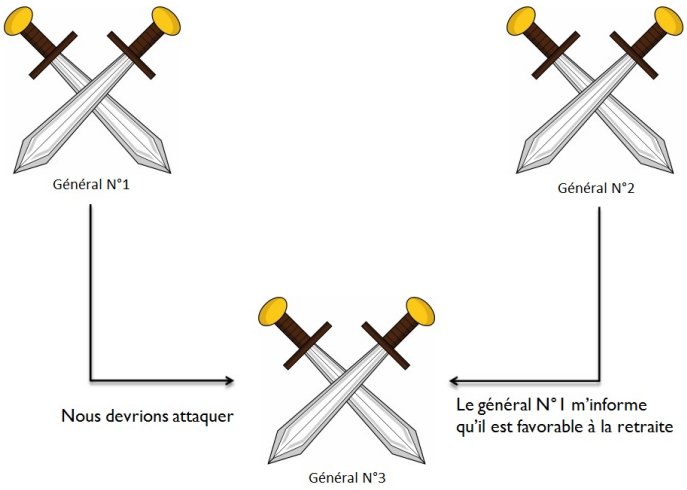

FIGURE 2.5 – Le dilemne des généraux byzantins : au vu des informations qui lui parviennent, le général N°3 ne sait pas si le traître est le général N°1 ou le général N°2

Dans de telles conditions, obtenir un consensus en présence de f traîtres n'est possible que si le nombre total de généraux N est suffisamment élevé, respectivement si $N > 3f$, et si les généraux s'échangent les propositions des autres protagonistes. Une logique analogue s'applique au contexte de duplication de machine à états, où une proportion de réplicas malveillants cherche à corrompre la cohérence des réplicas corrects. Un tel comportement est donc reconnu en tant que comportement byzantin. Dans le cadre d'un système dis-

tribué, est considéré comme comportement byzantin tout comportement ne répondant pas aux spécifications du système, tels l'envoi de messages corrompus, la génération de réponses incorrectes, ou toutes tentatives de déni de service telle la saturation de bande passante (*flooding*).

2.1.2.2 Modèle de Fautes Byzantines

Pour résumer, le nombre de réplicas requis afin d'assurer la fiabilité d'un système dépend intrinsèquement des hypothèses induites par l'environnement considéré. Ainsi, la nature des fautes envisagées ou encore le degré de fiabilité du réseau influencent considérablement la conception des protocoles de tolérance aux pannes [52]. Dans le cadre même de la tolérance aux fautes byzantines, un large éventail d'hypothèses peut être établi. Le protocole OBFT [66], contrairement à ses pairs, considère que les clients ne peuvent être instigateurs de comportements byzantins. Dans une même optique, certains protocoles requièrent la présence de composants fiables sur chaque réplica [27, 40, 72], remaniant ainsi la nature du problème traité.

Nous décrivons ci-dessous le modèle considéré dans ce manuscrit, ce modèle présente les hypothèses communément admises dans l'état de l'art. Le système est composé de N réplicas, dont au plus f peuvent être responsables de comportements byzantins. Une quantité indéterminée de clients peut elle aussi être l'instigatrice de comportements byzantins. L'ensemble de ces nœuds malveillants, qu'il s'agisse de clients ou de réplicas, peuvent collaborer afin d'endommager le système dupliqué. Il est toutefois impossible pour un nœud malveillant d'usurper l'identité d'un nœud correct. Autrement dit, un nœud malveillant est incapable de contrefaire un code d'authentification de message, une signature, ou tout autre mécanisme d'authentification utilisé. Le réseau est ultimement synchrone, soit asynchrone avec des intervalles de synchronie durant lesquels les messages se verront délivrés. Ces périodes de synchronies sont nécessaires afin d'assurer la propriété de *vivacité*. Le réseau respecte également les propriétés du modèle d'Aguilera et al. [5] : les canaux de communication ne peuvent pas créer de message, mais peuvent dupliquer, corrompre, ou perdre les messages échangés.

2.1.2.3 PBFT : une Tolérance aux Fautes Byzantines Pratique

En 1999, Miguel Castro et Barbara Liskov proposèrent PBFT[21, 22], la première tentative pratique destinée à offrir un protocole de duplication tolérant l'occurrence de fautes byzantines en conditions réelles. PBFT implémente en effet tout un ensemble de techniques permettant de réduire le surcoût lié à la duplication. Ce protocole garantit les propriétés de *sûreté* et de *vivacité* dans

un environnement ultimement synchrone. Il nécessite pour ce faire $N = 3f + 1$ réplicas pour tolérer la présence de f réplicas fautifs simultanément. Ce protocole fait partie de la classe des protocoles de duplication active (tous les réplicas exécutent manuellement chaque requête) et requiert la présence d'un *primary* en charge d'assurer l'exécution cohérente des requêtes sur chaque réplica. Le *primary* associe à chaque requête un ordre dans lequel elle se devra d'être exécutée. Les réplicas restants (*backups*) doivent ensuite s'accorder sur l'ordre proposé. Si un consensus est obtenu, les requêtes seront donc toutes exécutées dans le même ordre sur chaque réplica correct. Afin d'assurer la pérennité du système en présence d'un *primary* malveillant, PBFT implémente également un mécanisme de changement de vues (*view-change*) permettant de remplacer le *primary* courant. Enfin, chacun des messages échangés se voit sauvegardé par les réplicas, afin de permettre à ceux-ci de recouvrir un état cohérent en cas de problème.

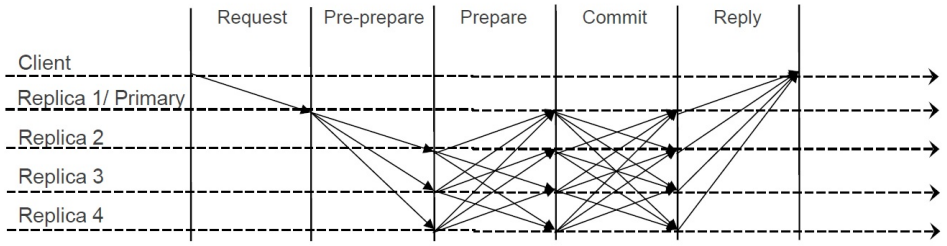

FIGURE 2.6 – Modèle de communication de PBFT

Consensus. Le consensus effectué par PBFT est présenté dans la figure 2.6. Celui-ci est initié par l'envoi d'une requête de la part d'un client au *primary* (REQUEST). Cette requête doit être authentifiée afin d'assurer un contrôle d'accès au service dupliqué. Le consensus débute lorsque le *primary* associe à la requête reçue un numéro de séquence unique, qu'il diffuse ensuite aux *backups* par l'intermédiaire d'un message PRE-PREPARE. Lors de la réception d'un message PRE-PREPARE, les *backups* prennent dès lors connaissance de la requête envoyée par le client, ainsi que du numéro de séquence qui lui fut affilié par le *primary*. La prochaine étape du consensus consiste en la diffusion de ce numéro de séquence par l'ensemble des *backups* (PREPARE). Cette étape permet à chacun des réplicas de posséder une vision globale des numéros de séquence locaux associés à la requête. Elle permet donc à chaque réplica de connaître d'hypothétiques divergences quant aux numéros de séquence envoyés par le *primary*. Lorsqu'un réplica reçoit suffisamment de messages PREPARE ($2f$) corroborant le numéro de séquence contenu dans le PRE-PREPARE reçu

précédemment, il peut conclure que $2f + 1$ réplicas se sont accordés sur un même numéro de séquence, et diffuse un message COMMIT à tous les réplicas. Finalement, lorsqu'un réplica reçoit $2f + 1$ messages COMMIT, il peut dès lors exécuter la requête et répondre au client. Un quorum de $2f + 1$ réplicas s'accordant sur un unique numéro de séquence est nécessaire. En effet, un quorum de cette taille implique la présence d'au moins $f + 1$ réplicas corrects qui considèrent ce numéro de séquence comme étant valide, rendant ainsi impossible le choix d'un numéro de séquence différent par d'autres réplicas corrects (cf. figure 2.7).

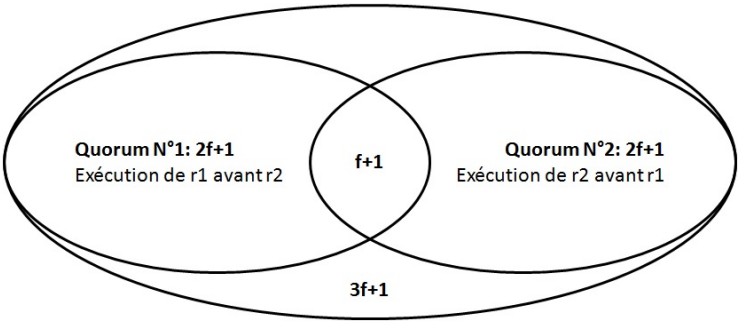

FIGURE 2.7 – l'utilisation de quorums de taille $2f + 1$ rend impossible la validation de deux requêtes différentes associées à un même numéro de séquence. Seuls f parmi les $f + 1$ réplicas peuvent être fautifs : le réplica correct ne peux pas se prononcer en faveur des deux requêtes

Changement de vues. Une vue représente la configuration du protocole de BFT à un instant donné, elle nous renseigne sur l'identité du *primary*, et par conséquent des *backups*. Afin d'assurer la continuité du service dupliqué en présence d'un *primary* malveillant, PBFT a recours à l'utilisation d'un mécanisme de changement de vues. Ce mécanisme a pour fonction de remplacer un *primary* dès lors que ce dernier ne remplit plus son rôle d'ordonnanceur.

Un changement de vue se voit déclenché lorsque plusieurs conditions sont remplies. Tout d'abord, si un client ne reçoit pas une réponse attendue, il retransmet sa requête, mais la destine cette fois ci à tous les réplicas (à la fois au *primary* et aux *backups*). Lorsqu'un *backup* reçoit une requête, il fait suivre cette requête au *primary* et démarre un minuteur dédié. Si le *primary* est innocent, la requête se verra exécutée avant l'expiration du minuteur. Dans le cas contraire, le *backup* déclenche un changement de vue en diffusant un message VIEW-CHANGE à tous les réplicas, et cesse de participer aux consensus instigués par le *primary* actuel. Si suffisamment de *backups* estiment que le comportement du *primary* ne répond plus aux spécifications du système, il leur

sera possible de collecter $2f + 1$ messages VIEW-CHANGE, soit le nombre suffisant pour révoquer le *primary* actuel en faveur d'un nouveau *primary*.

Optimisations. Si PBFT fut reconnu comme le premier protocole de tolérance aux fautes byzantines pratique, ceci est dû au faible surcoût engendré par le protocole de duplication face à ses prédécesseurs [41, 59]. PBFT implémente l'optimisation de *batching* (traitement de lots de requêtes), soit la capacité d'ordonner plusieurs requêtes via l'utilisation d'un unique message PRE-PREPARE. Cette optimisation permet de créer, d'authentifier, de transmettre et de vérifier moins de messages. Contrairement à ses prédécesseurs, l'authentification des messages échangés par PBFT n'est plus réalisée grâce à l'utilisation de signatures digitales mais de codes d'authentification de messages (MACs). Les processus de génération et de vérification d'un MAC nécessitent une quantité de calcul très inférieure à ces mêmes opérations réalisées via signatures digitales, ce qui permet au protocole de fournir de meilleures performances pratiques. Si la taille des requêtes est trop importante, le client lui-même est chargé de les faire parvenir aux différents réplicas, afin d'éviter une surcharge du réseau. De plus, dès lors qu'une requête est parvenue à l'ensemble des réplicas, celle-ci est remplacée par son empreinte (digest). L'utilisation d'empreintes est également plébiscitée lors de l'envoi des réponses aux clients. Ainsi, un unique réplica correct peut envoyer la réponse complète, alors que les autres réplicas peuvent se contenter d'envoyer l'empreinte concordante. PBFT utilise également le protocole de communication UDP, ainsi que la primitive de multi-diffusion (*multicast*), ce qui contribue également à réduire le trafic réseau. Enfin, PBFT propose l'exécution provisoire de requêtes (*tentative execution*). Il autorise ainsi les réplicas à exécuter les requêtes dès la fin de l'étape PREPARE. Cette optimisation permet de s'affranchir de l'étape de COMMIT, mais peut parfois impliquer le retour à un état précédent (*rollback*).

2.2 Catégorisation des Protocoles de Tolérance Aux Fautes Byzantines

L'émergence de PBFT suscita l'intérêt de nombreux chercheurs, et de multiples contributions furent proposées au fil des ans. Celles-ci répondent à deux principales thématiques : le besoin d'*efficacité* et le besoin de *robustesse*.

2.2.1 Optimisation des Performances en Absence de Faute

La tolérance aux fautes byzantines reposant fondamentalement sur les concepts de duplication de machines à états, une des principales préoccupations des chercheurs demeure le fait de masquer au mieux le surcoût lié à cette duplication. Il s'agit en pratique de proposer des solutions permettant d'améliorer la performance des protocoles de BFT, tout en conservant les propriétés de *sûreté* et de *vivacité*. Lampson préconise qu'un système se doit d'être rapide dans des conditions normales, et qu'il se doit de réaliser des progrès dans des conditions hostiles [51]. C'est en partageant cette optique que nombre de chercheurs se proposèrent d'améliorer la performance des protocoles de BFT en se focalisant sur les périodes pendant lesquelles le système n'est pas confronté à l'occurrence de comportements byzantins [4, 18, 27, 28, 37, 38, 40, 43, 57, 68, 70, 73]. Les protocoles *efficaces* se fixent donc pour objectif d'optimiser les performances du système en absence de faute. Cela se caractérise généralement par le raffinement du modèle de communication inter-réplicas, ou par l'intégration d'optimisations liées à l'émergence de nouvelles technologies telles la virtualisation ou la parallélisation. Les protocoles mentionnés dans cette section s'inspirant de PBFT, ceux-ci implémentent nombre de ses optimisations, en particulier l'utilisation de codes d'authentification de messages (MACs) afin de se décharger du coût des signatures digitales.

2.2.1.1 Protocoles Spéculatifs

les protocoles spéculatifs exécutent les requêtes avant de s'accorder sur l'ordre dans lequel elles se doivent d'être exécutées [4, 55]. En d'autres termes, un réplica exécutera localement les requêtes reçues sans avoir la certitude que les autres réplicas les exécuteront également. Un tel paradigme remet en cause la propriété de *sûreté*, car l'état de différents réplicas peut se révéler temporairement incohérent. Afin de remédier à ce problème, les protocoles spéculatifs implémentent un mécanisme de *rollback*, permettant à l'ensemble

des réplicas de recouvrir un état cohérent.

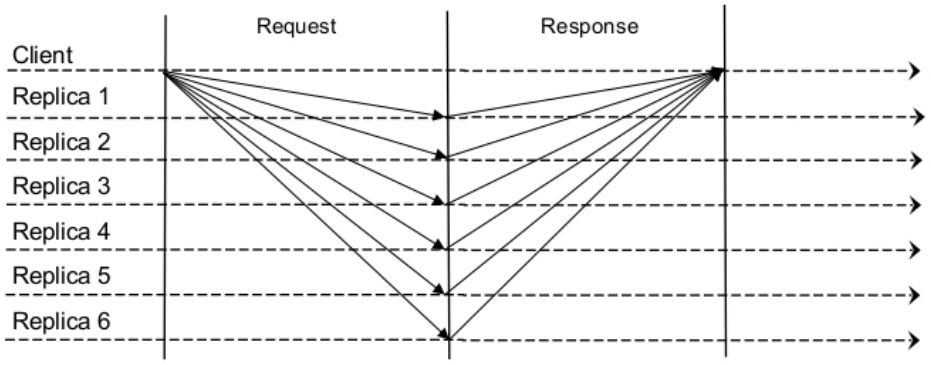

FIGURE 2.8 – Modèle de communication de Q/U

Query/ Update (Q/U). Le protocole Query/Update (Q/U) fut présenté en 2005 et se destine à fournir un haut niveau de performance en absence de contention [4]. Contrairement à la plupart de ses pairs, Q/U requiert l'utilisation de $5f + 1$ réplicas pour tolérer f fautes byzantines. La figure 2.8 présente son modèle de communication, nous y constatons que Q/U n'utilise pas de *primary*, et qu'il s'affranchit de toute forme de consensus précédant l'exécution d'une requête. Les requêtes sont en effet directement envoyées aux réplicas par les clients : les réplicas les exécutent puis répondent aux clients sans se concerter. Un tel modèle de communication implique peu d'échanges de messages et peu d'opérations d'authentification, il se montre donc particulièrement performant si les conditions demeurent propices. En présence de contention (deux clients distincts qui envoient leurs requêtes au système de manière concurrente) ou de tout autre événement remettant en cause la propriété de *sûreté*, les réplicas peuvent rétablir la cohérence de leurs états via une procédure de synchronisation. Celle-ci se voit déclenchée lorsqu'un client ne parvient pas à récolter suffisamment de réponses concordantes. Alors que Q/U parvient à réduire considérablement le nombre de messages requis par PBFT, il n'en reste pas moins très vulnérable face à la contention ou à la présence de clients byzantins. Un client byzantin peut intentionnellement provoquer des divergences entre les états des réplicas corrects via l'émission de requêtes contradictoires.

Zyzzyva. C'est en 2009 que Kotla et al. introduisirent Zyzzyva. Bien que spéculatif, Zyzzyva requiert la présence d'un *primary* contrairement à Q/U [43]. Ce *primary* lui permet de tolérer efficacement l'occurrence de contention, sans pour autant imposer le recours à un consensus systématique (cf. figure

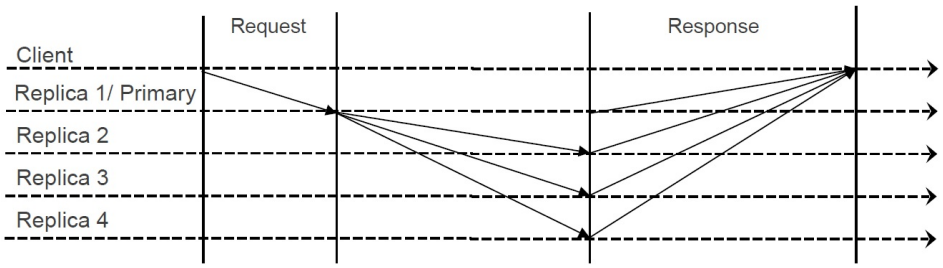

FIGURE 2.9 – Modèle de communication de Zyzzyva

2.9). Le modèle de communication de Zyzzyva évolue en présence d'éléments fautifs, afin de garantir les propriétés de *sûreté* et de *vivacité*. Tout comme son prédécesseur Q/U, Zyzzyva s'affranchit de l'étape de consensus. La présence d'un *primary* permet néanmoins d'ordonner chaque requête de manière centralisée et résout de ce fait les problèmes associés à l'émergence de contention (PRE-PREPARE). Il convient de remarquer que, contrairement à PBFT (section 2.1.2.3), les étapes PREPARE et COMMIT ne succèdent pas à l'étape d'ordonnancement de Zyzzyva. Il en découle une exécution spéculative des requêtes ordonnées par le *primary*, suivie par l'envoi immédiat des réponses aux clients. Un tel modèle de communication s'astreint de tout échange de messages entre *backups*, les clients se voient donc responsables de maintenir une cohérence entre les états des différents réplicas. Suite à l'émission d'une requête, un client s'attendra à recevoir $3f+1$ réponses identiques de la part des réplicas. Si le client ne reçoit pas suffisamment de réponses concordantes, il les collecte puis les retourne aux réplicas par l'intermédiaire d'un message COMMIT, permettant ainsi aux réplicas de prendre connaissance de l'état de leurs pairs et d'évoluer en accord. Tout comme PBFT, Zyzzyva implémente lui aussi un mécanisme de changement de vue destiné à remplacer un *primary* fautif. La nature spéculative de Zyzzyva implique plusieurs variations conceptuelles quant à sa procédure de changement de vue. En particulier, lorsqu'un réplica se prononce en faveur d'un changement de vue, celui-ci continue d'exécuter les requêtes ordonnées par le *primary* actuel jusqu'à son éviction. De plus, l'absence d'un consensus systématique impose le choix d'un état commun afin d'assurer l'initialisation cohérente de la vue à venir. Alors que Zyzzya parvient à tolérer efficacement la présence de contention, il demeure particulièrement vulnérable face aux attaques engendrées par des clients byzantins. Un client malveillant peut par exemple forcer la génération de messages COMMIT afin d'entraîner le protocole dans une succession d'échanges de messages inutiles. La présence de clients byzantins est d'autant plus dangereuse si ceux-ci collaborent avec un *primary* malveillant. L'absence de consensus permet à un *primary* fautif de provoquer intentionnellement une divergence entre les états

des réplicas, que les clients byzantins peuvent se garder de résoudre.

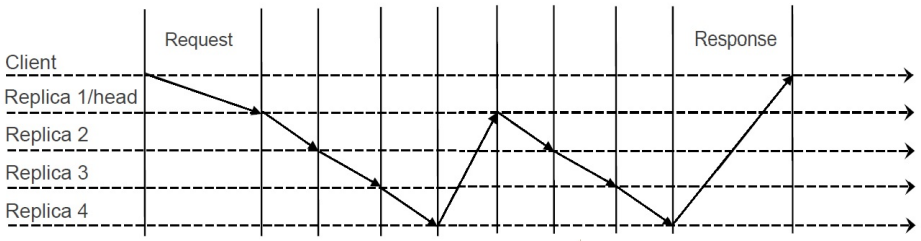

FIGURE 2.10 – Modèle de communication de Ring

Ring. Ring apparaît en 2011 [38], et comme son nom l'indique utilise un modèle de communication sous forme d'anneau (cf. figure 2.10), il exploite le plein potentiel de l'optimisation de *batching*. Son design repose principalement sur la constatation selon laquelle la charge des réplicas n'est pas uniformément répartie. En prenant l'exemple de Zyzzyva, nous constatons que son *primary* est amené à authentifier et envoyer davantage de messages que les *backups*, ce qui provoque en pratique l'apparition d'un goulet d'étranglement (*bottleneck*) sur ce nœud spécifique. Ring parvient à uniformiser le nombre d'authentifications effectuées par chaque réplica tout en optimisant la consommation de bande passante. Néanmoins, chaque communication ne mettant en scène que deux nœuds, traverser l'ensemble de l'anneau requiert un nombre d'étapes important. Ring coordonne l'exécution des requêtes via l'utilisation d'un *primary*. Cependant, contrairement à ses pairs, Ring permet aux clients d'envoyer leurs requêtes à n'importe quel réplica, celles-ci se voient par la suite transportées jusqu'au *primary* courant, contribuant ainsi à réduire la charge de ce dernier. Intuitivement, la présence d'un réplica byzantin (qui se refuserait par exemple à transmettre les messages au réplica suivant) résulte indubitablement en une rupture de l'anneau. Afin de pallier à ce problème, Ring implémente un modèle de communication alternatif. Si un client n'obtient pas une réponse attendue, il provoque le passage du protocole en mode *résilient*. Dans ce mode, l'authentification des messages est réalisée via signatures digitales et non MACs, afin d'isoler plus facilement le/les élément(s) fautif(s). Les messages ne sont plus envoyés à un unique successeur, mais aux $f + 1$ successeurs, permettant ainsi de tolérer une rupture de l'anneau. Bien que moins vulnérable que Zyzzyva ou Q/U, un client fautif peut intentionnellement forcer le passage de Ring en mode *résilient*, écartant de fait nombre de ses optimisations.

***h*BFT.** Duan et al. proposèrent en 2014 *h*BFT, un protocole spéculatif inspiré du design de PBFT et de Zyzzyva [31]. Le client envoie une requête au

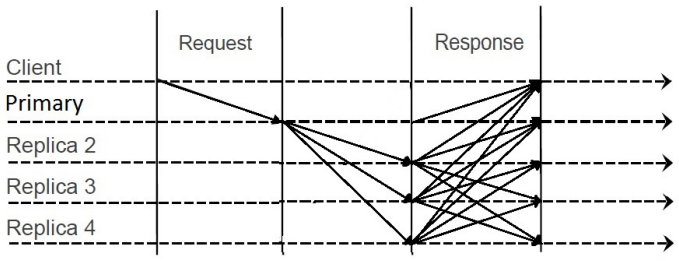

FIGURE 2.11 – Modèle de communication de hBFT

primary qui lui associe l'ordre dans lequel elle devra être exécutée. hBFT réalise ensuite un amalgame entre les étapes REPLY et COMMIT des protocoles Zyzzyva et PBFT. L'exécution est déclenchée suite à la réception d'un numéro de séquence, puis la réponse est directement envoyée (Zyzzyva). En parallèle, l'exécution d'une requête implique également l'échange d'informations relatives à l'évolution des états de chaque réplica (PBFT). Si un réplica collecte $2f + 1$ messages COMMIT concordants, il peut conclure que tous les réplicas corrects partagent le même état. Dans le cas contraire, il est possible que l'état des réplicas corrects diverge, mais contrairement à Zyzzyva, l'échange d'information imposé par hBFT lui permet de résoudre les conflits sans recourir aux clients. hBFT implémente néanmoins un mécanisme permettant à un client conscient d'une incohérence d'en informer les réplicas. La spéculation implémentée par hBFT le rend moins efficace que Zyzzyva en absence de faute, mais lui permet de tolérer plus efficacement leur présence. Les divergences entre réplicas corrects sont plus facilement résolues par hBFT, mais demeurent possibles dû à l'exécution spéculative des requêtes.

2.2.1.2 Protocoles Evolutifs

Les protocoles évolutifs se reposent essentiellement sur des instances de protocoles pré-existants, ils font varier ces dernières en fonction des événements auxquels le système se voit confronté [14, 37].

HQ. Le protocole Hybrid Quorum (HQ), présenté en 2006, repose sur les protocoles Q/U et PBFT [28]. Il bénéficie du haut niveau de performance fourni par Q/U en l'absence de contention, tout en tolérant efficacement celle-ci par l'intermédiaire du protocole PBFT. Contrairement à Q/U, HQ ne nécessite que $3f + 1$ réplicas. La première interaction entre un client et les réplicas a pour fonction de convenir de l'ordre dans lequel la requête du client doit être exécutée. En absence de contention, les réplicas répondent au client de

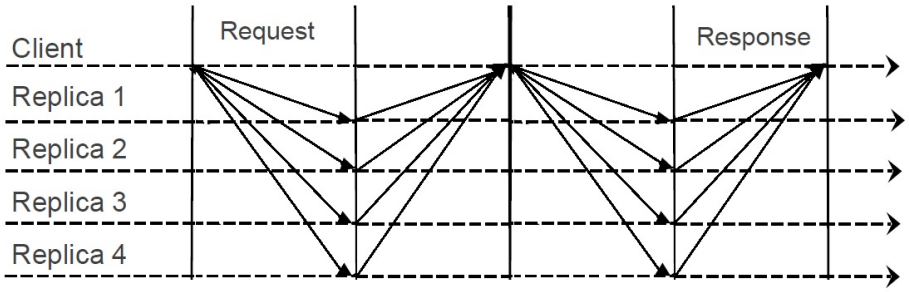

FIGURE 2.12 – Modèle de communication de HQ

manière concordante, le client interagit de nouveau avec les réplicas afin d'instancier l'exécution de sa requête selon le modèle de communication de Q/U. Si en revanche les réplicas ne s'accordent pas sur l'ordre d'exécution de la requête, le protocole remplace l'étape d'exécution par un consensus (PBFT) permettant d'obtenir un ordre d'exécution définitif. Le protocole HQ s'inspire donc de deux protocoles de tolérance aux fautes byzantines existants, permettant ainsi d'assurer un haut niveau de performance en absence de contention tout en se reposant sur la fiabilité de PBFT afin de résoudre les conflits.

Aliph. Aliph nécessite $3f+1$ réplicas et fut proposé par Guerraoui et al. en 2010 [37] . Il est l'amalgame de trois protocoles différents, entre lesquels il bascule de manière statique, selon les conditions auxquelles il se voit confronté :

— **Quorum.** Destiné à fournir un haut niveau de performance en absence de contention.

— **Chain.** Destiné à fournir un haut niveau de performance en absence de faute.

— **Backup/PBFT.** Destiné à tolérer l'occurrence de fautes.

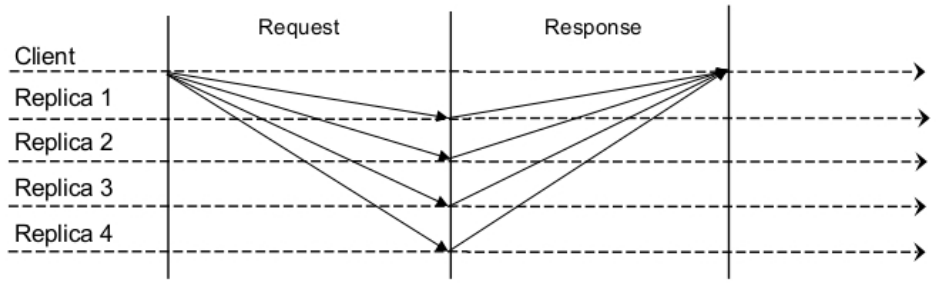

FIGURE 2.13 – Modèle de communication de Quorum

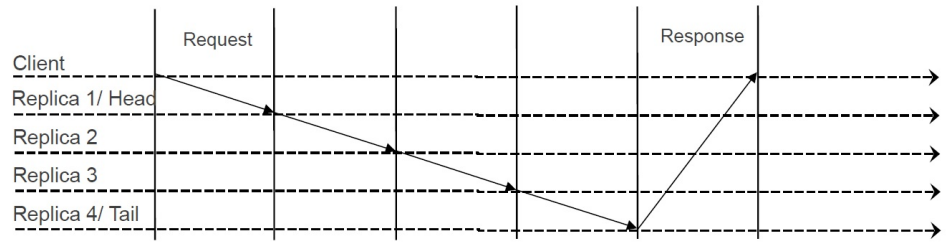

FIGURE 2.14 – Modèle de communication de Chain

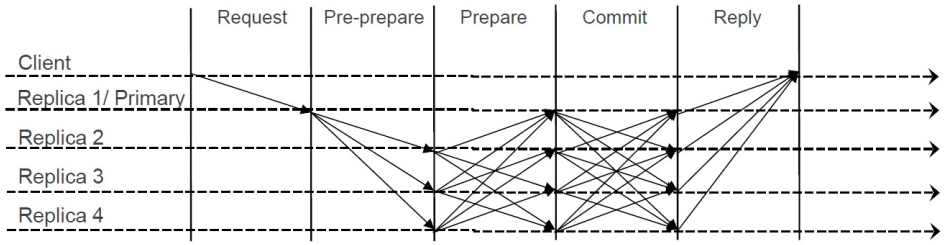

FIGURE 2.15 – Modèle de communication de Backup

Quorum utilise un modèle de communication identique à Q/U mais nécessite moins de réplicas et n'implémente pas de mécanisme de synchronisation, il ne peut donc fonctionner qu'en l'absence de contention. L'exécution d'une requête ne nécessite que deux échanges de messages, comme décrit dans la figure 2.13. Si l'instance de Quorum est active, un client requiert l'exécution d'une requête par son envoi à l'ensemble des réplicas. Chaque réplica l'exécute de manière spéculative et répond au client. Si le client reçoit $3f + 1$ réponses identiques, l'exécution est terminée et valide. Dans le cas contraire, le client en informe le système, et initie un changement d'instance vers Chain. Chain fonctionne d'une manière similaire à Ring, chaque nœud n'envoie ses messages qu'à son successeur direct, ce qui permet de réduire le nombre d'authentifications nécessaires, ainsi que la charge du réseau (figure 2.14). Contrairement à Ring en revanche, les requêtes ne peuvent être transmises qu'au *primary* et non aux nœuds en milieu de chaîne. Si un client ne reçoit pas de réponse valide (envoyée par le réplica en fin de chaîne et contenant $f+1$ confirmations de réplicas différents), celui-ci suspecte une rupture de la chaîne, et initie un changement d'instance vers Backup. Backup peut implémenter tout protocole capable d'assurer les propriétés de *sûreté* et de *vivacité* en présence d'éléments byzantins. Guerraoui et al. utilisent PBFT dans l'implémentation originale d'Aliph en tant qu'instance de Backup [37]. Si l'élément fautif responsable du changement d'instance retrouve un comportement normal, Backup laissera la place à Quorum, initialisant ainsi un nouveau cycle. La mutualisation opérée

par Aliph lui permet de faire face efficacement à de nombreux scénarios. Néanmoins, la responsabilité des changements d'instances incombant aux clients, ceux-ci peuvent délibérément provoquer un recours systématique à l'instance de Backup, rendant de ce fait obsolète l'optimisation des performances associée à Quorum et Chain.

Adapt. Quatre ans plus tard, Adapt s'inspire de la conception d'Aliph en intégrant davantage d'instances à sa palette [14]. Adapt utilise respectivement les instances des protocoles PBFT, Zyzzyva, Quorum, Ring et Chain. Contrairement à Aliph, Adapt évalue quelle instance est la plus adaptée via l'utilisation de techniques d'apprentissage automatique (*machine learning*), il se repose sur plusieurs critères parmi lesquels le nombre de clients, la taille des requêtes et des réponses, ou encore la présence de réplicas fautifs. Adapt évalue également en temps réel trois indicateurs de performance afin de déterminer l'instance à utiliser. Ces indicateurs sont respectivement le débit courant, le temps de réponse, et la présence de contention. L'utilisation de plusieurs protocoles en coordination semble prometteuse. Une politique efficace quant au choix de l'instance la plus adaptée est la clé du succès de ces protocoles évolutifs. Adapt est la première tentative visant à choisir dynamiquement l'instance la plus adaptée à une situation en temps réel. Son concept peut être implémenté en intégrant davantage de prototypes pour faire face à un plus large éventail de situations.

2.2.1.3 Virtualisation et Parallélisme

L'émergence de nouvelles technologies lors des 15 dernières années a façonné l'évolution des prototypes de BFT. Alors que les nombreuses contributions présentées dans les sections précédentes se focalisent sur la manière dont le consensus lui-même est orchestré, l'application pratique d'optimisations externes au consensus permet d'engendrer un gain de performance non négligeable. L'implémentation originale de PBFT n'intègre pas les notions d'architecture multi-coeurs ou de virtualisation, et se révèle par conséquent peu adaptée à une exécution sur des machines modernes. Plusieurs chercheurs se sont intéressés à l'impact des avancées technologiques sur la performance des protocoles de BFT, et comment en tirer profit en pratique.

Séparer le Consensus de l'Exécution. Yin et al. proposèrent les fondamentaux d'une nouvelle architecture dédiée aux protocoles de duplication de machines à états tolérants les fautes byzantines [74]. Cette nouvelle architecture s'illustre par la séparation des étapes de consensus et d'exécution. En

pratique, alors que le nombre de réplicas nécessaires au consensus demeure $3f + 1$, l'exécution peut être effectuée sur seulement $2f + 1$ réplicas, sans remettre en cause les propriétés de *sûreté* et de *vivacité*. La séparation du consensus et de l'exécution permet également l'intégration d'un pare-feu (*firewall*), composé des réplicas affectés au consensus [74]. Ces réplicas ont pour responsabilité de vérifier la fiabilité des réponses renvoyées par les réplicas en charge de l'exécution, avant de les retransmettre aux clients. Quelques années plus tard, Wood et al. présentent ZZ, un raffinement de la politique de séparation du consensus et de l'exécution sous fond de virtualisation [73]. ZZ nécessite lui aussi $3f + 1$ réplicas pour effectuer le consensus, mais n'utilise que $f + 1$ réplicas lors d'une exécution en absence de faute. Afin de tolérer l'occurrence de pannes lors de l'étape d'exécution, ZZ se contente de réveiller d'autres réplicas si le besoin s'en fait sentir. La capacité d'activer rapidement de nouveaux réplicas est cruciale pour permettre une utilisation pratique de cette politique. La virtualisation offre une solution adaptée à un tel problème, tout en s'intégrant parfaitement à l'environnement d'informatique en nuage.

BFT-SMaRt. BFT-SMaRt, proposé en 2014, est une protocole de BFT actualisant l'implémentation de PBFT [18]. Celui-ci améliore la fiabilité du code et permet de tirer profit des architectures multi-coeurs. L'objectif de BFT-SMaRt est de fournir un prototype pratique et adapté aux machines actuelles tout en conservant la rigueur du consensus original de PBFT. Le prototype décompose les fonctionnalités du protocole en plusieurs modules : (i) l'envoi et la gestion des requêtes, (i) le consensus, (iii) le transfert d'états, et (iv) la reconfiguration du système (changement de vue). Il permet également l'utilisation de plusieurs *threads* afin de mieux répartir la charge liée à la génération/vérification des authentifications. BFT-SMaRt peut être configuré pour permettre l'authentification des requêtes via signatures digitales ou via codes d'authentification de messages. BFT-SMaRt est à l'heure actuelle un des seuls prototypes fiables implémentant l'ensemble des mécanismes requis à une utilisation pratique des protocoles de BFT. Le code de BFT-SMaRt est disponible à cette adresse : https ://github.com/bft-smart/library.

COP. C'est finalement en 2015 que Behl et al. proposèrent COP, une nouvelle optimisation nommée *Consensus-Oriented Parallelization* [16]. Cette optimisation est présentée par ses auteurs comme l'introduction de la superscalarité aux protocoles de consensus. COP permet aux utilisateurs d'adapter la configuration du système en fonction du matériel sur lequel il se voit déployé. Tout comme BFT-SMaRt, COP implémente également le consensus de PBFT, mais exploite les capacités d'une architecture multi-coeur à un autre

niveau de parallélisation. COP se propose de paralléliser l'ensemble du consensus, sans se restreindre à la gestion concurrente des messages. En pratique, le goulet d'étranglement d'un protocole de BFT se situe au niveau du *primary*, qui se doit d'attribuer des numéros de séquence cohérents à chaque requête entrante. COP propose donc de paralléliser l'attribution de ces numéros de séquence via l'implémentation de plusieurs *threads* dédiés nommés *pillars*. En pratique, bien qu'utilisant le même modèle de communication que PBFT, COP fournit de loin les meilleures performances de l'état de l'art. Réduire le nombre de messages échangés pour parvenir au consensus ne semble donc pas être la meilleure façon d'optimiser les performances des protocoles de BFT.

2.2.1.4 Reformulation du Problème

Le modèle de faute byzantines présenté en section 2.1.2.2 s'est vu remanié par diverses hypothèses au fil du temps. Nous illustrons dans cette section plusieurs directions de recherche impliquant une redéfinition du modèle de faute byzantines considéré.

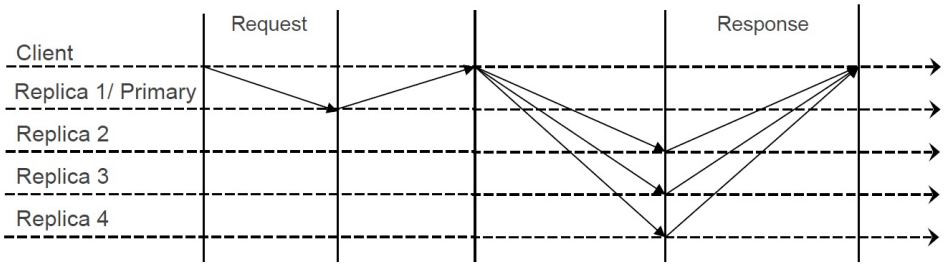

FIGURE 2.16 – Modèle de communication de OBFT

Fiabilité des clients. Le design du protocole OBFT (Obfuscated BFT) [66], présenté en figure 2.16 repose sur l'hypothèse selon laquelle le système n'est pas confronté à des clients byzantins. Il délègue ainsi la majeure partie du consensus au côté client du protocole. Cela se traduit en pratique par une indépendance complète entre réplicas, ainsi qu'une réduction du nombre de réplicas requis. En effet, seuls $2f + 1$ réplicas sont nécessaires pour assurer les propriétés de *sûreté* et de *vivacité*, car le *primary* n'est plus en mesure d'envoyer des informations incohérentes aux réplicas. Le protocole débute par l'envoi d'une requête au *primary*, celui-ci y associe un numéro de séquence puis renvoie la requête au client. Le client étant fondamentalement correct, il garantit la cohérence de tous les numéros de séquence transférés aux *backups*. Le protocole nécessite néanmoins la présence de f réplicas passifs, dont la

fonction consiste à remplacer les réplicas byzantins. Le remplacement d'un replica fautif s'effectue lui aussi du côté client du protocole. Si un client ne reçoit pas suffisamment de réponses concordantes, il collecte l'état des réplicas et déclenche le remplacement de celui dont l'état diverge.

Fiabilité des réplicas. Les protocoles présentés dans cette section nécessitent le concours de composants présumés fiables sur chaque réplica [27, 40, 72]. Cette hypothèse permet de réduire le coût de la duplication à $2f + 1$.

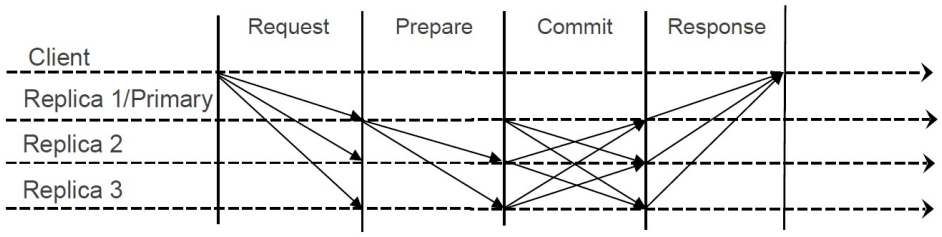

FIGURE 2.17 – Modèle de communication de MinBFT

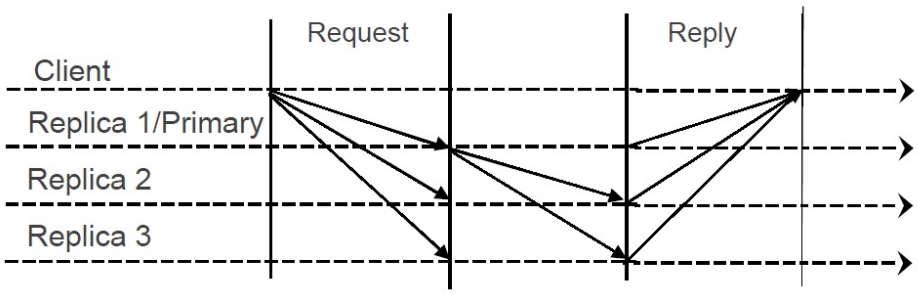

FIGURE 2.18 – Modèle de communication de MinZyzzyva

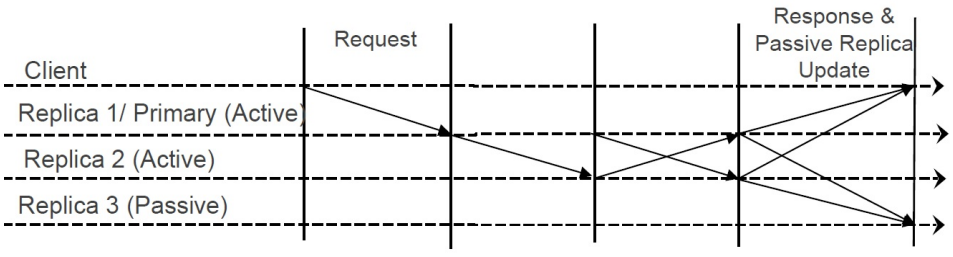

FIGURE 2.19 – Modèle de communication de CheapTiny

MinBFT et MinZyzzyva requièrent la présence du service *USIG* (*Unique Sequential Identifier Generator*) sur tous les réplicas [72]. Ce service associe à

chaque message la valeur d'un compteur authentifié. Il garantit respectivement les propriétés suivantes : (i) il est impossible d'associer le même identifiant à deux messages différents. (ii) La valeur d'un nouvel identifiant succède à la valeur de l'identifiant précédemment assigné. L'utilisation d'un tel service facilite la résolution du problème de consensus car il prohibe l'envoi de numéros de séquence incohérents de la part du *primary*. Le design de MinBFT et MinZyzzyva s'inspire des protocoles originaux PBFT et Zyzzyva, leurs modèles de communication se trouvent toutefois réorganisés par l'intégration du service USIG. La figure 2.17 présente le consensus implémenté par minBFT, et la figure 2.18 présente celui de minZyzzyva. le protocole BFT-TO utilise quant à lui le service *TO wormhole* (*Trusted Ordering Wormhole*) [27], en charge d'assister le processus de diffusion des messages. Le service *TO wormhole* tempère l'évolution du consensus en fonction de la réception effective des messages échangés. Il permet d'assurer une réception collective et ordonnée des messages diffusés. CheapBFT applique les principes illustrés en section 2.2.1.2 concernant l'adaptation dynamique du modèle de communication [40]. Il requiert la présence du service *CASH* (*Counter Assignment Service in Hardware*), destiné à assurer la cohérence et l'authentification des messages échangés. Comme Aliph ou Adapt, CheapBFT utilise plusieurs instances de protocoles pour améliorer ses performances. CheapBFT est l'amalgame de deux protocoles de duplication, respectivement CheapTiny et MinBFT. La transition d'un protocole à l'autre s'effectue par l'intermédiaire de la procédure CheapSwitch et permet de maintenir la cohérence des réplicas. Similairement aux autres protocoles présentés dans cette section, CheapBFT ne nécessite que $2f + 1$ réplicas. Il parvient même à réduire ce nombre à $f+1$ en absence de faute, via le protocole CheapTiny présenté en figure 2.19. Celui-ci bénéficie des avantages associés à une duplication passive partielle (section 2.1.1.2). Les réplicas passifs ne participent pas au consensus, leurs états ne sont donc mis à jour que lorsque le consensus est terminé. En présence de fautes, CheapSwitch réveille les réplicas passifs et effectue la transition vers MinBFT. Lorsque que la fiabilité du système est restaurée, MinBFT laisse de nouveau place à CheapTiny.

Relaxe des Garanties. Les contributions présentées dans cette section se proposent d'adapter le degré de tolérance aux fautes byzantines en fonction de l'environnement réel du système dupliqué. Ils relaxent la propriété de *sûreté* tout en considérant de nouvelles hypothèses.

Li et al. s'intéressent au devenir des protocoles de duplication lorsque ceux-ci sont confrontés à un nombre de fautes supérieur à f [52]. Ils proposent BFT2F, une extension de PBFT garantissant les propriétés de *sûreté* et de *vivacité* en présence de f réplicas fautifs, mais permettant également de ga-

rantir un degré de fiabilité moindre jusqu'à $2f$ réplicas fautifs. Dans le cas où $2f$ réplicas sont fautifs, BFT2F peut soit interdire une évolution incohérente des états au prix de la propriété de *vivacité* ; soit permettre l'évolution parallèle d'états incohérents tout en garantissant la propriété de *Fork* consistency*. Cette propriété permet à un réplica correct de prendre conscience de l'incohérence de son état sous certaines conditions (interactions multiples avec un client correct). En d'autres termes, la propriété de *Fork* consistency* permet d'éviter une corruption complète du système dès lors que plus de f réplicas sont fautifs.

Scrooge effectue une distinction entre réplicas byzantins et réplicas inactifs [65]. Il requiert $2f + 2b$ réplicas (f et $b > 0$) , f étant le nombre maximal de réplicas fautifs, et b le nombre de réplicas byzantins parmi ces réplicas fautifs. Scrooge utilise le même modèle de communication que Zyzzyva, à la différence que seul un sous ensemble de réplicas (*replier quorums*) est chargé d'envoyer les réponses aux clients. Effectuer une distinction entre réplicas inactifs et réplicas byzantins permet de déterminer plus efficacement l'impact de pannes potentielles. Ce modèle interdit par exemple à un réplica inactif de collaborer avec un réplica byzantin, limitant ainsi les dégâts que le nœud byzantin pourrait causer sur le système.

Zeno reproduit le modèle de communication utilisé par Zyzzyva [68]. Il relaxe la propriété de *sûreté* au profit de la propriété d'*Eventual consistency*. La propriété d'*Eventual consistency* garantit la cohérence à terme de l'état des réplicas, mais n'émet pas d'hypothèse sur le temps requis pour y parvenir. Le protocole Zeno autorise donc l'évolution parallèle de réplicas dans des états différents. Néanmoins, afin d'assurer l'obtention d'une cohérence à terme, il implémente une primitive permettant la mise en commun (*merge*) de deux historiques distincts. Pour ce faire, Zeno distingue deux type de requêtes, *weak* et *strong*, selon les relations causales qui leurs sont associées. Une requête *weak* peut se voir exécutée sur un quorum de taille $f+1$, alors qu'une requête *strong* nécessite le quorum habituel de taille $2f + 1$.

Arantes et al. proposent de traiter le problème de tolérance aux fautes byzantines de manière probabiliste [8], contrairement au modèle communément considéré par la littérature impliquant la présence d'au plus f réplicas fautifs. Le modèle proposé par les auteurs associe à chaque réplica une probabilité de démontrer un comportement byzantin. Cette probabilité nommée *réputation* évolue dynamiquement selon l'environnement du système. Dans un tel contexte, la politique de gestion des réputations remplace le problème original du consensus. Le protocole éveille et endort des nœuds dynamiquement selon leurs propensions à fournir une réponse correcte.

2.2.2 Minimisation de l'Impact des Comportements Byzantins

Parallèlement à l'optimisation des performances en absence de faute, une seconde problématique fut rigoureusement traitée dans l'état de l'art. Il s'agit de la minimisation de l'impact des comportements byzantins sur le système dupliqué. La fragilité des protocoles présentés en section 2.2.1 n'est plus à démontrer [7, 26, 71]. Les performances de ceux-ci se voient bien souvent réduites à néant face aux attaques instiguées par des nœuds malveillants. Clement et al. introduisirent la notion de protocoles *robustes* dans le cadre d'une tolérance aux fautes byzantines [26]. Ceux-ci sont destinés à maintenir un degré de performance constant, à la fois en présence et en absence de nœuds byzantins. Ils implémentent pour ce faire nombre de mécanismes destinés à traiter efficacement l'occurrence de comportements byzantins spécifiques.

2.2.2.1 Comportements Byzantins

Classification des comportements fautifs. Avizienis et al proposèrent une classification hiérarchique des différents types de fautes envisageables [13] :

- **Arrêt inopiné.** Un arrêt inopiné peut être engendré par de nombreuses causes différentes (panne de courant, surchauffe, etc.). Un processus affecté par un arrêt inopiné cesse de participer au protocole, il peut être considéré comme inactif.
- **Omission.** Une omission se traduit par l'échec de réception ou d'émission d'un message. Une telle faute peut être causée par un dysfonctionnement affectant le réseau.
- **Exécution.** Un processus commet une faute d'exécution lorsqu'il exécute une opération de manière incorrecte. Une faute d'exécution peut entraîner un réplica dans un état incohérent. Ce genre de faute peut se voir causée par un bug logiciel.
- **Byzantine/Arbitraire.** Une faute byzantine peut être affiliée à une faute d'omission, d'exécution, un arrêt inopiné, ou un quelconque comportement ne respectant pas les spécifications du système. Cette catégorie générique comprend par exemple la prise de contrôle d'un réplica par une entité malveillante ou la corruption intentionnelle des messages transitant sur le réseau.

Dégradation des Performances. L'impact de nombreux comportements byzantins fut évalué au fil des ans [67]. Parmi le large éventail de fautes envisageables, certaines fautes bénignes tel le *crash* d'un réplica passent inaperçues.

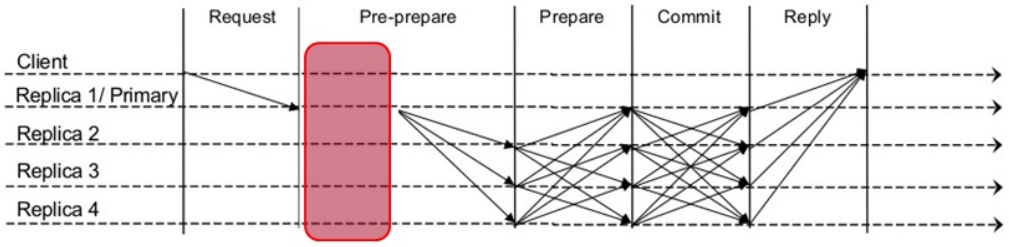

FIGURE 2.20 – Illustration de l'attaque de *Pre-Prepare Delay* sur le protocole PBFT

D'autres, en revanche, peuvent engendrer de fortes dégradations de performance sur nombre de protocoles de BFT. Il existe plusieurs failles, conceptuelles et logicielles, permettant à un attaquant malveillant de dégrader les performances du système. L'utilisation des codes d'authentification de messages, qui permirent jadis à PBFT de fournir un solution pratique à la tolérance aux fautes byzantines, introduisent l'opportunité pour un client byzantin de remplacer volontairement un *primary* correct [26]. Cette attaque, nommé *MAC-Attack* sera présentée plus en détails en section 3.1.1. Amir et al. définissent l'attaque de *Pre-Prepare Delay*, qui s'est vue considérée par nombre de chercheurs car capable de dégrader la performance des protocoles *efficaces* de manière considérable [7]. En pratique, elle consiste en l'introduction délibérée d'un délai par le *primary* lors de l'envoi d'un message PRE-PREPARE. Cette attaque, illustrée en figure 2.20 sur le protocole PBFT, est applicable à tout protocole nécessitant le concours d'un *primary* pour ordonner l'exécution des requêtes. L'attaque de *Pre-Prepare Delay* permet à un *primary* malveillant de profiter de son statut pour feindre un problème de réseau. Elle met en exergue les difficultés liées à l'identification des éléments fautifs. Comment distinguer en pratique un *primary* byzantin instigateur d'une attaque de *Pre-Prepare Delay* d'un réseau particulièrement lent ? Afin de faire face aux tentatives de saturation de bande passante (*flooding*), les protocoles *robustes* implémentent généralement un mécanisme de *Blacklisting*. De plus, de nombreux protocoles *robustes* forcent la transmission des requêtes à l'ensemble des réplicas avant d'initier le consensus. Une telle mesure permet de savoir avec exactitude quelles sont les requêtes qui doivent être ordonnées par le *primary*, afin de le remplacer plus facilement s'il ne les ordonne pas en temps voulu.

2.2.2.2 Protocole Robustes

Un protocole de BFT est considéré *robuste* s'il manifeste la capacité de maintenir un degré de performance suffisant en présence de nœuds malveillants.

Pour y parvenir, de nombreux mécanismes furent proposés afin de tempérer l'impact d'autant d'attaques différentes.

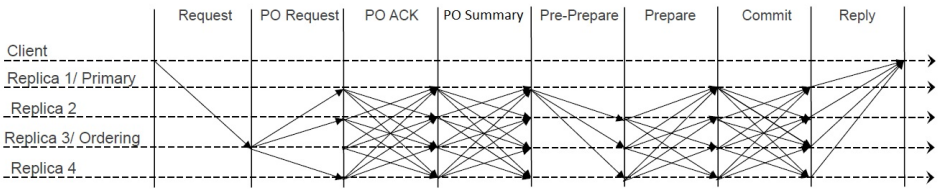

FIGURE 2.21 – Modèle de communication de Prime

Prime. Prime fut proposé en 2008 par Amir et al. [7]. C'est dans cet article que l'attaque de *Pre-Prepare Delay* fut instaurée. En déclenchant cette attaque sur le protocole PBFT, Amir et al. parviennent à faire chuter son débit à 0 requête/seconde. Prime implémente le consensus de PBFT, enrichi par plusieurs mécanismes de robustesse destinés à faire face à l'attaque de *Pre-Prepare Delay*. D'abord, tous les messages échangés sont authentifiés via l'utilisation de signatures digitales et non de codes d'authentification de messages (MACs). Cette pratique n'est pas courante, même dans le cadre des protocoles *robustes* (cf. reste de la section). La plupart d'entre eux n'utilisent les signatures digitales que pour l'authentification des requêtes afin de se défaire de la possible occurrence de *MAC-Attacks*. La signature de tous les messages échangés entre réplicas n'est pas nécessaire et demeure coûteuse. Prime échange systématiquement les requêtes reçues avant d'initier un consensus. Il permet aux clients d'envoyer leurs requêtes à un réplica quelconque et non plus systématiquement au *primary*. Lorsqu'un réplica reçoit une requête valide de la part d'un client, il lui associe un numéro de séquence puis transmet l'ensemble dans un message PO-REQUEST. lors de la réception d'un message PO-REQUEST, un réplica répond par l'émission d'un message PO-ACK contenant l'empreinte associée à la requête. Finalement, lorsqu'un réplica collecte $2f$ PO-ACK correspondants au PO-REQUEST, il peut générer un certificat de pré-ordonnancement. Périodiquement, les réplicas diffusent un message PO-SUMMARY qui contient, pour chaque réplica, le numéro de séquence maximal associé aux certificats de pré-ordonnancement. De son côté, le *primary* collecte les messages PO-SUMMARY et envoie périodiquement un message PRE-PREPARE contenant les PO-SUMMARY reçus depuis le dernier PRE-PREPARE émis (le message PRE-PREPARE est donc potentiellement vide). La réception d'un PRE-PREPARE initie le consensus tel qu'effectué par PBFT, les requêtes sont exécutées et les réponses transmises aux clients. L'envoi périodique de messages PRE-PREPARE, même vides,

permet aux *backups* de repérer un *primary* malveillant. Ceux-ci évaluent les performances du réseau afin d'estimer la fréquence à laquelle les messages du *primary* devraient être reçus. Si le *primary* est trop lent, les *backups* le considéreront malveillant et procéderont à un changement de vue.

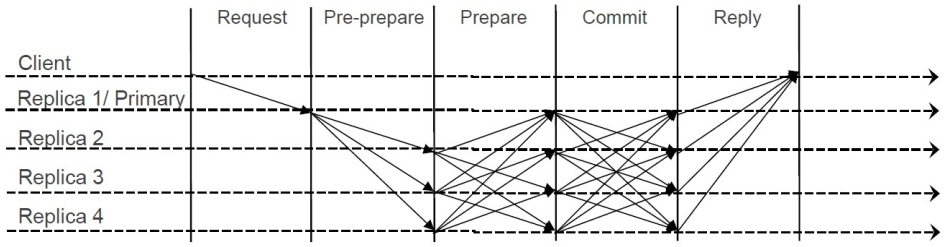

FIGURE 2.22 – Modèle de communication d'Aardvark

Aardvark. Clement et al. démontrèrent en 2009 que la plupart des protocoles de BFT reconnus (PBFT, Q/U, HQ et Zyzzyva) ne parviennent pas à tolérer l'occurrence de comportements malveillants. Afin de pallier à ce problème, ils présentèrent un nouveau protocole : Aardvark, conçu pour faire face efficacement à toutes sortes de comportements byzantins [26]. Conceptuellement, ce protocole est très similaire à PBFT, il requiert $3f + 1$ réplicas, et partage le même modèle de communication. Il intègre toutefois plusieurs mécanismes additionnels afin de renforcer la robustesse du système. Afin d'interdire aux clients de déclencher des *MAC-Attacks*, il raffine le mécanisme d'authentification dédié aux requêtes. Aardvark combine l'utilisation des MACs et des signatures pour l'authentification de chaque requête. L'utilisation de signatures est suffisante afin d'interdire l'occurrence de *MAC-Attacks* (cf. section 3.1.1.2). Aardvark décide néanmoins de ré-authentifier chaque requête signée avec un code d'authentification de message. La vérification d'un MAC étant bien plus rapide que celle d'une signature, cela permet au protocole de se prémunir efficacement contre les dénis de services via la mise en place d'une politique de *Blacklisting*. Spécifiquement, lorsqu'une requête est reçue par un réplica, celui-ci va d'abord vérifier le MAC, puis, s'il est valide, la signature. Si le MAC est valide mais non la signature, le client est considéré malveillant et ses futures requêtes seront ignorées. Ce processus permet d'éviter d'avoir à systématiquement vérifier les signatures des clients considérés malveillants. Aardvark utilise également plusieurs interfaces réseau (NIC pour *Network Interface Controller*), afin de séparer le trafic induit par les réplicas de celui des clients. Chaque réplica possède respectivement $3f + 1$ interfaces : une pour la communication avec les clients, et une dédiée à chaque autre réplica. Une telle mesure permet de garantir que le trafic généré par un élément fautif ne

perturbera pas la collaboration des éléments corrects. En d'autres termes, l'impact d'un réplica malveillant qui tenterait de monopoliser la consommation de bande passante s'en trouve amoindri. Dans une même optique, Aardvark associe à chaque réplica plusieurs files de réception, une pour les clients, et une pour chaque autre réplica. Cela permet de prioriser les messages envoyés par les réplicas, nécessaires au progrès du système. Aardvark parallélise l'étape de vérification liée à l'authentification des messages reçus. Il implémente respectivement deux *threads*, l'un dédié à la vérification des requêtes (MACs et signatures), et l'autre dédié à la vérification des messages restants (MACs). Une telle technique permet d'éviter l'apparition d'un goulet d'étranglement sur le processus principal, la vérification des requêtes signées étant plus coûteuse. Finalement, afin d'éviter à un *primary* malveillant de saper les performances du système, Aardvark impose le recours à des changements de vues réguliers. En pratique, ces changements de vues sont déclenchés de la manière suivante : le protocole exige d'un *primary* qu'il fournisse un débit au moins égal à 90% du débit maximal obtenu lors des $N = 3f + 1$ vue précédentes. Ce seuil évolue à la hausse périodiquement, jusqu'à ce que le *primary* courant ne puisse plus fournir le débit attendu. Lorsque cette situation apparaît, un changement est déclenché et un nouveau réplica prend la place du *primary* courant.

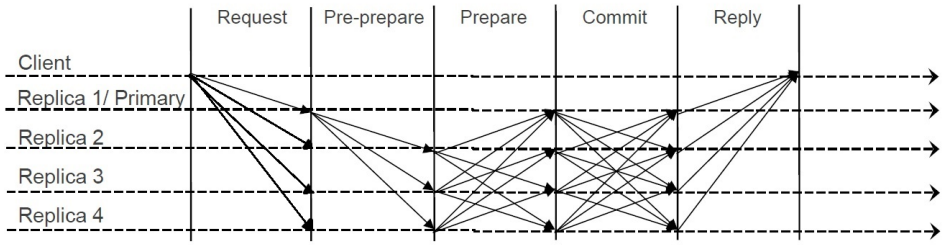

FIGURE 2.23 – Modèle de communication de Spinning

Spinning. Spinning fut présenté en 2009 [71], et se base comme ses prédécesseurs sur le modèle de communication de PBFT. Il effectue tout comme Aardvark des changements de vue réguliers, mais ceux-ci se voient déclenchés automatiquement suite à l'ordonnancement de chaque lot de requêtes. Une telle politique permet le remplacement systématique d'un *primary* sans engendrer de communications additionnelles entre réplicas. Spinning impose l'envoi des requêtes à tous les réplicas (figure 2.23). Les changements de vues étant automatiques, le prochain réplica amené à devenir *primary* peut accoler son futur message PRE-PREPARE au message COMMIT de la vue courante. Cette optimisation permet d'améliorer les performances globales de

Spinning, autorisant les consensus de plusieurs vues à être instanciés en parallèle. Le changement perpétuel de *primary* permet à Spinning de disperser l'impact d'un *primary* fautif sans recourir aux coûteux mécanismes de Prime ou d'Aardvark. Néanmoins, contrairement à ses deux prédécesseurs, Spinning n'utilise pas de signatures digitales pour l'authentification des requêtes clients. Cela le rend donc vulnérable aux attaques MACs (cf. section 3.1.1).

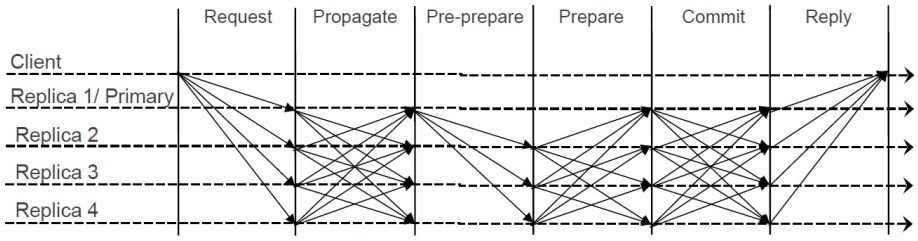

FIGURE 2.24 – Modèle de communication de RBFT

RBFT. C'est en 2013 que RBFT fut proposé par Aublin et.al. [10]. Tout comme Prime, RBFT s'intéresse à la fréquence à laquelle un *primary* devrait proposer ses messages PRE-PREPARE. Comme tous les protocoles présentés ci-dessus, le consensus effectué par RBFT est identique à celui utilisé par PBFT. RBFT impose néanmoins l'envoi de chaque requête à l'ensemble des réplicas, suivi d'un échange systématique des requêtes reçues avant d'initialiser le consensus. RBFT exécute $f + 1$ instances du protocole de duplication, chacune mettant en scène un *primary* différent. Chacune de ces instances effectue le consensus, mais seules les requêtes ordonnées par l'instance principale sont exécutées. Un dispositif de surveillance (*monitoring*) permet d'évaluer la différence de performance entre l'instance principale et les autres instances, et, si cette différence est trop importante, un changement de vue se voit déclenché. Cette technique n'est efficace que sur les systèmes fonctionnant en boucle ouverte (*open loop*), où un client à la possibilité d'envoyer une nouvelle requête sans attendre de réponse. RBFT utilise comme Aarvark plusieurs interfaces réseau pour isoler le trafic entre réplicas et clients. Il mesure également la latence associée à chaque requête afin de s'assurer que le *primary* demeure équitable envers chaque client.

R-Aliph. R-Aliph fur proposé en 2015 et s'inspire des concepts originaux d'Aliph [9]. Il nécessite lui aussi $3f + 1$ réplicas et est l'amalgame des trois protocoles suivants :

— **Quorum.** Destiné à fournir un haut niveau de performance en absence de contention.

— **Chain.** Destiné à fournir un haut niveau de performance en absence de faute.

— **Backup/Aardvark.** Destiné à tolérer l'occurrence de fautes.

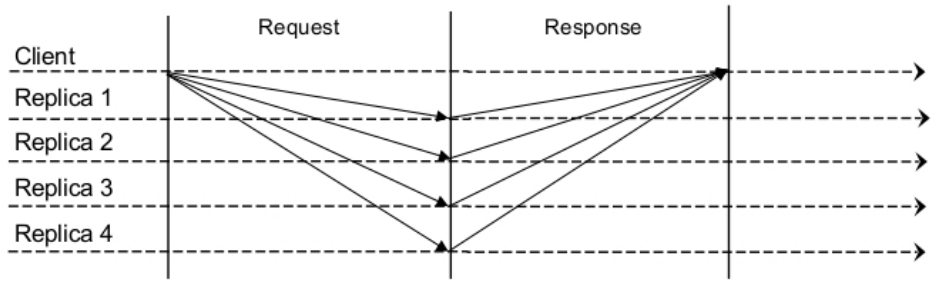

FIGURE 2.25 – Modèle de communication de Quorum

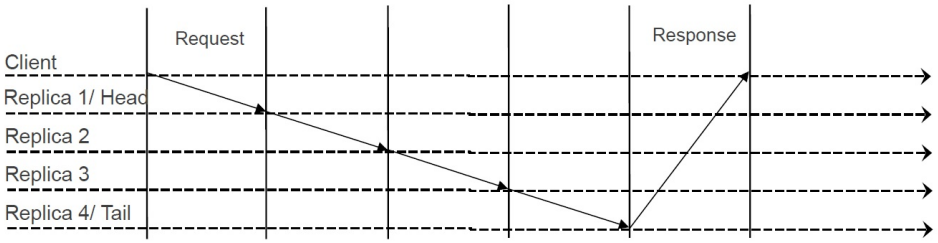

FIGURE 2.26 – Modèle de communication de Chain

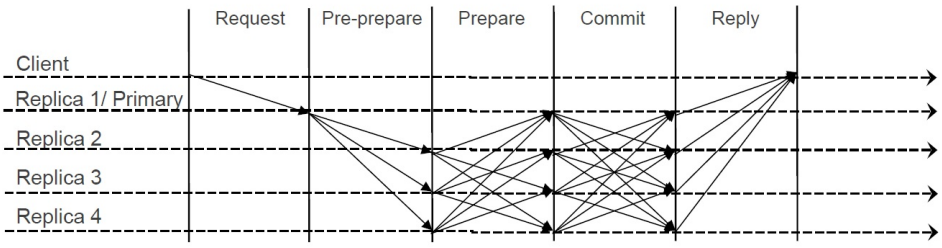

FIGURE 2.27 – Modèle de communication de Backup

Tout comme Aliph, R-Aliph utilise les instances de Quorum et Chain afin d'offrir des performances satisfaisantes en absence de contention et/ou de faute. Pour tolérer efficacement l'occurrence de comportements byzantins, R-aliph remplace l'instance originale de Backup (PBFT) par Aardvark. Contrairement à Aliph, Les instances Chain et Quorum de R-Aliph évaluent le débit courant, et le compare avec le débit fourni précédemment par l'instance d'Aardvark. Si Chain ou Quorum fournissent un débit trop faible face à celui

d'Aardvark, le protocole suspecte une tentative de déni de service et remplace l'instance courante par celle d'Aardvark. Chaque instance de R-aliph utilise plusieurs interfaces réseaux dédiées aux communications entre réplicas et avec les clients. R-aliph retire aux clients la capacité de provoquer un changement d'instance, ceux-ci sont désormais déclenchés par les réplicas eux-mêmes. En pratique, un réplica suspectant un problème va endosser lui-même le rôle d'un client en envoyant une requête vide, initiant ainsi le processus de changement d'instance si nécessaire.

2.3 Synthèse

Nous avons présenté dans ce chapitre les principales directions de recherche dans le cadre de la tolérance aux fautes byzantines. Les contributions associées peuvent être classées en deux catégories distinctes : (i) l'optimisation des performances en absence de faute, dont l'objectif est de masquer au mieux le surcoût lié à la duplication, et (2) la minimisation de l'impact des comportements fautifs, afin de garantir un niveau suffisant de performance quelle que soit les conditions du système. Le tableau 2.1 résume les caractéristiques de plusieurs protocoles de BFT contemporains. Il s'agit d'une classification théorique utilisée dans de nombreux articles [27, 32, 37, 43, 65, 66, 72], et destinée à offrir un aperçu de la performance des protocoles de BFT. Nous avons intégré davantage de protocoles, et divisé leurs évaluations selon les modèles de communication utilisés lors de l'occurrence de fautes. La colonne 'Nombre de réplicas requis' détermine le degré de duplication nécessaire au protocole afin de tolérer la présence de f fautes simultanées [1]. Les colonnes 'Opérations d'authentification par requête sur un réplica' permettent de déterminer le débit maximal théorique des différents protocoles : plus le nombre d'authentifications à effectuer sur un réplica est faible, plus le débit maximal est élevé [2]. Nous séparons les opérations d'authentification selon les primitives utilisées : σ pour les codes d'authentification de messages, et θ pour les signatures digitales. Les colonnes 'Messages échangés dans le chemin critique' introduisent un indicateur de latence : un faible nombre de messages échangés implique théoriquement une latence plus faible [3].

1. Il ne s'agit que des réplicas *actifs*, certains protocoles réveillent des réplicas *passifs* en présence de fautes

2. Cet indicateur ne prend pas en compte la présence d'un quelconque degré de parallélisation

3. Aucune différence n'est effectuée entre les échanges clients-réplicas et réplicas-réplicas. Un échange entre client et réplica est d'ordinaire plus long, en raison de la proximité des réplicas

TABLE 2.1 – Comparaison de plusieurs protocoles de tolérance aux fautes byzantines, lorsque ceux-ci génèrent des lots de b requêtes

	Absence de faute		**Présence de faute(s)**		Nombre de réplicas requis
	Opérations d'authentification par requête sur un réplica	Messages échangés dans le chemin critique	Opérations d'authentification par requête sur un réplica	Messages échangés dans le chemin critique	
PBFT [21]	$\sigma(2+\frac{8f+1}{b})$	4	$\sigma(2+\frac{8f+1}{b})$	4	$3f+1$
Q/U [4]	$\sigma(2+8f)$	2	$\sigma(2+8f)$	2	$5f+1$
Zyzzyva [43]	$\sigma(2+\frac{3f}{b})$	3	$\sigma(4+5f+\frac{3f}{b})$	5	$3f+1$
Ring [38]	$\sigma(\frac{2}{3f+1}+\frac{2f+2}{b})$	$6f+3$	$\theta(2+\frac{4f+2}{b})$	$3f+3$	$3f+1$
hBFT [32]	$\sigma(2+\frac{3f}{b})$	3	$\sigma(2+\frac{3f}{b})$	3	$3f+1$
HQ [28]	$\sigma(4+4f)$	4	$\sigma(2+\frac{8f+1}{b})$	4	$3f+1$
Quorum [37]	$\sigma(2)$	2	-	-	$3f+1$
Chain [37]	$\sigma(1+\frac{f+1}{b})$	$3f+2$	-	-	$3f+1$
BFT-SMaRt [18]	$\sigma(2+\frac{8f+1}{b})$	4	$\sigma(2+\frac{8f+1}{b})$	4	$3f+1$
COP [16]	$\sigma(2+\frac{8f+1}{b})$	4	$\sigma(2+\frac{8f+1}{b})$	4	$3f+1$
OBFT [66]	$\sigma(2)$	4	$\sigma(2)$	4	$2f+1$
MinBFT [72]	$\sigma(1+\frac{f+3}{b})+\theta(1)$	4	$\sigma(1+\frac{f+3}{b})+\theta(1)$	4	$2f+1$
MinZyzzyva [72]	$\sigma(1)+\theta(1+1/b)$	3	$\sigma(3+f)+\theta(2+1/b)$	5	$2f+1$
CheapTiny [40]	$\sigma(1+\frac{f+1}{b})+\theta(1)$	4	-	-	$f+1$
BFT2F [52]	$\theta(1)+\sigma(1+\frac{8f+1}{b})$	4	$\theta(1)+\sigma(1+\frac{8f+1}{b})$	4	$3f+1$
Scrooge [65]	$\sigma(2+\frac{3f}{b})$	3	$\sigma(5+10f+\frac{3f}{b})$	7	$2f+2B$
Prime [7]	$\theta(\frac{1}{3f+1}+2+\frac{20f}{b})$	6	$\theta(\frac{1}{3f+1}+2+\frac{20f}{b})$	6	$3f+1$
Aardvark [26]	$\theta(1)+\sigma(1+\frac{8f+1}{b})$	4	$\theta(1)+\sigma(1+\frac{8f+1}{b})$	4	$3f+1$
Spinning [71]	$\sigma(2+\frac{8f+1}{b})$	4	$\sigma(2+\frac{8f+1}{b})$	4	$3f+1$
BFTMencius [57]	$\sigma(2+\frac{8f+1}{b})$	4	$\sigma(2+\frac{8f+1}{b})$	4	$3f+1$
RBFT [10]	$\theta(1)+\sigma(2+\frac{14f+1}{b})$	5	$\theta(1)+\sigma(2+\frac{14f+1}{b})$	5	$3f+1$

Chapitre 3

Problèmes Ouverts en Tolérance aux Fautes Byzantines

Sommaire

Ce chapitre introduit la problématique abordée. Il décrit en détail la nature des comportements byzantins considérés, ainsi que les solutions proposées par l'état de l'art afin de leur faire face.

3.1 Attaques Considérées

Cette section présente en détail la nature des comportements byzantins abordés. Nous avons illustré en section 2.2.2.1 plusieurs types d'attaques (déni de services) perpétrées par des éléments malveillants. Notre intérêt se porte particulièrement sur deux de ces attaques : (i) L'attaque MAC, et (ii) l'indifférence sélective du *primary* aux requêtes.

3.1.1 L'Attaque MAC

L'attaque MAC consiste en la corruption partielle d'une requête. Spécifiquement, un client malveillant restreint l'authentification d'une requête à un sous-ensemble de réplicas, qui la considéreront donc valide. Les réplicas restants, incapables de l'authentifier, rejetteront par conséquent cette requête. Dans le cadre d'une attaque MAC, la requête corrompue se trouve authentifiable par les *backups* et non le *primary*. Les *backups* s'attendent à ce que cette requête d'apparence correcte soit proposée pour ordonnancement, ce qui ne peut pas arriver car le *primary*, responsable de l'ordonnancement, rejettera cette requête invalide. Il en résulte le déclenchement d'un changement de vue initié par les *backups* qui suspecteront le *primary* d'avoir intentionnellement ignoré la requête.

3.1.1.1 Codes d'Authentification de Messages

Un code d'authentification de message (MAC) est un code associé à des données dans le but d'assurer l'*intégrité* et l'*authenticité* de ces dernières. Il permet donc d'assurer au destinataire que les données transmises n'ont subi aucune altération, tout en garantissant l'identité de leur émetteur. Contrairement aux fonctions de hachage classiques capables d'assurer la propriété d'*intégrité*, un code d'authentification de message requiert l'utilisation d'une clé secrète afin d'également garantir la propriété d'*authenticité*. Cette clé secrète confidentielle doit être partagée entre l'émetteur et le récepteur. Le génération puis la vérification d'un MAC s'effectue par l'intermédiaire d'une unique fonction appliquée sur le message. Il suffit au récepteur de comparer le code généré par l'émetteur au code généré localement (figure 3.1). Le principal

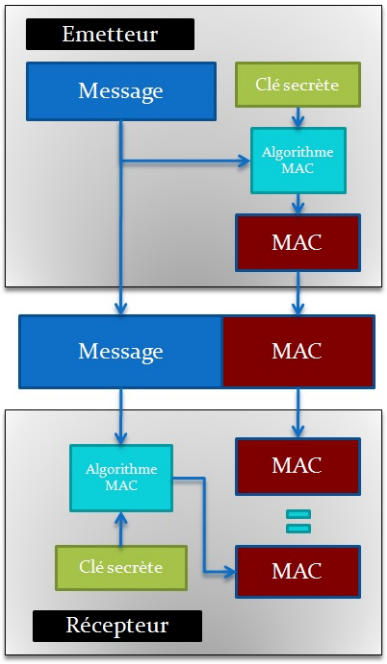

FIGURE 3.1 – Fonctionnement d'un code d'authentification de message

avantage d'une authentification via MACs réside dans la faible puissance de calcul nécessaire à leurs générations et leurs vérifications.

Authenticator. Un *Authenticator* est un vecteur contenant plusieurs codes d'authentification de messages. Il consiste donc à associer à un unique message plusieurs MACs, permettant ainsi à différents destinataires d'authentifier le message en question. Chacun des destinataires partage une clé secrète différente avec l'émetteur, afin d'assurer la propriété d'*authenticité*. Les *Authenticators* sont utilisés afin de permettre a un client d'envoyer une *unique* requête au *primary*, qui sera par la suite authentifiable par l'ensemble des réplicas.

Corruption partielle d'un *Authenticator*. Clement et al. rapportèrent qu'un unique client malveillant était capable de faire tomber la plupart des prototypes testés en générant des *Authenticators* corrompus [26]. L'attaque MAC, illustrée en figure 3.2 implique la corruption sélective de certains MACs présents à l'intérieur de l'*Authenticator*. Elle peut par exemple prendre l'aspect suivant : Le MAC dédié au *primary* est délibérément corrompu par un client, alors que le reste de l'*Authenticator* est valide. Lorsqu'un réplica reçoit une requête qu'il considère valide de la part d'un client, il la retransmet au *primary* et déclenche un minuteur. Si la requête n'est pas proposée pour exécution avant l'expiration du minuteur, le réplica suspecte le *primary* puis initie un

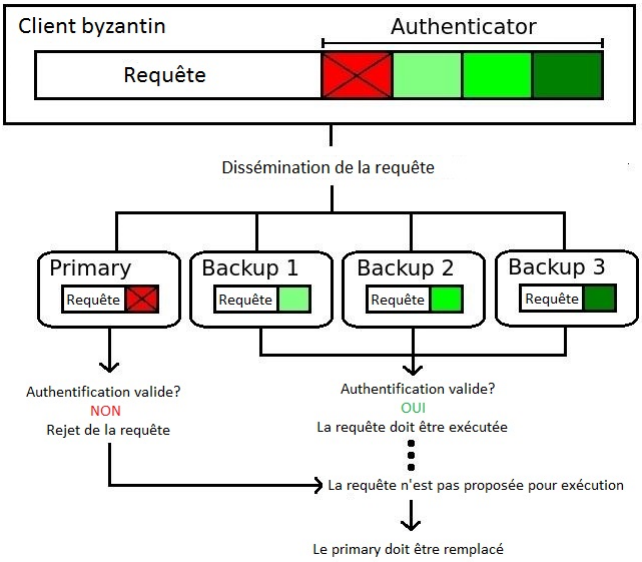

FIGURE 3.2 – Illustration de l'attaque MAC

changement de vue.

3.1.1.2 Signatures Digitales

Les signatures digitales permettent tout comme les MACs d'authentifier les messages échangés. Néanmoins, contrairement aux MACs qui ne nécessitent qu'une unique clé secrète, les signatures digitales implémentent le paradigme de cryptographie à clé publique [60]. Chaque nœud possède deux clés : une clé publique et une clé privée. La clé privée, utilisée pour générer la signature, n'est connue que par le processus en question, la clé publique quant à elle, utilisée pour vérifier la signature, se voit partagée entre tous les destinataires. Les signatures digitales permettent d'assurer les propriétés d'*intégrité*, d'*authenticité*, ainsi que la propriété de *non-repudiation*. La propriété de *non-repudiation* permet à un nœud quelconque en possession de la clé publique d'attester de l'*authenticité* d'un message signé, même s'il n'en est pas le destinataire explicite. L'avantage d'une signature digitale face à un MAC réside donc dans cette propriété de *non-repudiation*. Contrairement à un *Authenticator* composé d'autant de MACs que de destinataires, une unique signature peut être vérifiée par l'ensemble des destinataires. En revanche, les étapes de génération et de vérification d'une signature sont plus coûteuses que celles d'un MAC.

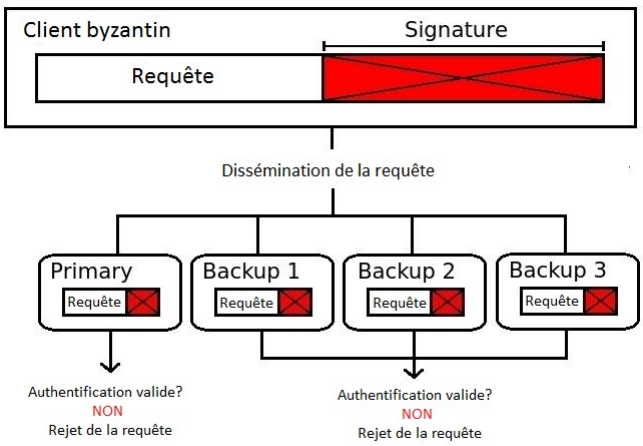

FIGURE 3.3 – Résoudre l'attaque MAC en utilisant les signatures digitales

L'attaque MAC peut donc être facilement résolue grâce à la propriété de *non-repudiation* fournie par les signatures digitales. En pratique, l'envoi d'un unique message signé interdit à un nœud fautif de restreindre l'authentification du message à un sous-ensemble de nœuds. Autrement dit, une signature correcte sera considérée valide par l'ensemble des réplicas alors qu'une signature corrompue sera unanimement rejetée (figure 3.3). Cette propriété de *non-repudiation* illustre la raison pour laquelle nombre de protocoles robustes remplacent systématiquement l'utilisation de MACs par celle de signatures digitales afin d'authentifier les requêtes.

3.1.2 L'indifférence Sélective du *Primary*

La seconde attaque considérée implique l'indifférence sélective du *primary* aux requêtes. En pratique, un *primary* malveillant peut intentionnellement ignorer certaines requêtes sans déclencher de changements de vues. Les protocoles *efficaces* ont pour politique d'imposer la transmission des requêtes directement au *primary* afin d'éviter une surcharge inutile du réseau en absence de faute. Cette décision offre à un *primary* malveillant l'opportunité de forcer la retransmission systématique des requêtes.

3.1.2.1 Flouer le Changement de Vue

Afin de déclencher un changement de vue, les *backups* évaluent la propension du *primary* à arborer un comportement byzantin. En pratique, un *backup* considère qu'un *primary* est fautif si celui-ci n'attribue pas de numéro

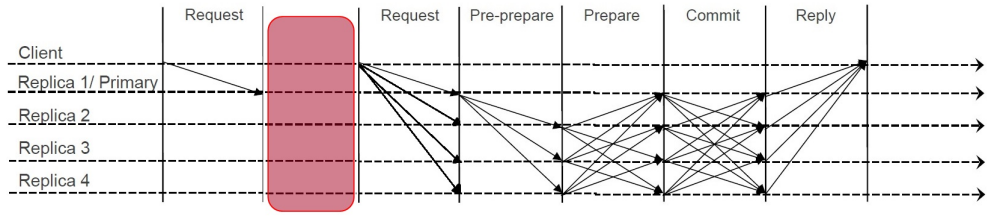

FIGURE 3.4 – Un *primary* malveillant force la retransmission systématique des requêtes sur le protocole PBFT

de séquence à une requête valide pendant l'intervalle de temps qui lui est imparti. Afin d'éviter une consommation inutile de bande passante, les protocoles *efficaces* n'imposent la transmission des requêtes qu'au *primary*. Dès lors qu'un client ne reçoit pas de réponse à temps, il retransmet la requête à l'ensemble des réplicas (*primary* et *backup*). Lorsqu'un *backup* reçoit une requête de la part d'un client, il déclenche un minuteur puis retransmet la requête en question au *primary*. Si le *primary* n'attribue pas à cette requête un numéro de séquence avant l'expiration du minuteur, le *backup* suspectera le *primary* d'être inactif, et déclenchera la procédure de changement de vue. Alors qu'un tel mécanisme permet de faire face efficacement à l'arrêt inopiné des réplicas, il introduit pour un *primary* malveillant l'opportunité de forcer une retransmission systématique des requêtes (figure 3.4).

Afin d'empêcher un *primary* fautif de forcer la retransmission des requêtes, les protocoles robustes comme RBFT ou Prime effectuent un échange systématique des requêtes dès leurs réceptions [7, 10]. Prime réalise cet échange en deux étapes, PO-Request et PO-ACK (cf. figure 2.21), RBFT s'assure d'une réception cohérente des requêtes grâce au round Propagate (cf. figure 2.24). Ces échanges se révèlent particulièrement coûteux si le service dupliqué implique l'utilisation de requêtes de grande taille.

3.2 Le Prix d'une Tolérance aux Fautes Byzantines *Robuste*

La tolérance aux fautes byzantines ne suffit pas à garantir un niveau de performance constant. En pratique, bien que les propriétés de *sûreté* et de *vivacité* soient assurées par les protocoles de BFT, il est possible qu'un déni de service rigoureusement planifié parvienne à faire drastiquement chuter le débit de ces protocoles. La *robustesse* est la capacité de certains protocoles à minimiser l'impact de ces dénis de services (cf. section 2.2.2). Elle implique l'intégration de mécanismes dédiés, engendrant inévitablement des surcoûts.

3.2.1 Des Authentifications Onéreuses

Cette section présente l'évaluation du surcoût engendré par la substitution des codes d'authentification de messages par les signatures digitales. L'utilisation de signatures digitales, destinées à interdire l'occurrence d'attaques MAC (cf. section 3.1.1.2), engendre un surcoût non négligeable sur les clients mais également sur les réplicas. Afin de démontrer la véracité de cette déclaration, nous avons effectué plusieurs séries d'expériences, mettant en scène trois protocoles de BFT exécutant le micro-benchmark x/y. Ce benchmark permet d'évaluer le coût des protocole de duplication en s'affranchissant de l'étape d'exécution. Autrement dit, les clients n'envoient que des requêtes vides, puis le *primary* associe à chaque requête un numéro de séquence afin d'initier le consensus. Les réplicas s'accordent sur ce numéro de séquence puis retournent des réponses vides aux clients. Un tel benchmark permet donc d'isoler le surcoût lié aux échanges de messages requis par les protocoles de duplication. La dénomination x/y correspond respectivement à la taille des requêtes (x Kb) et des réponses (y Kb) transitant sur le réseau. Les expériences sont réalisées en boucle close, les clients n'envoient par conséquent une nouvelle requête qu'après réception d'une réponse escomptée. Nous utilisons les mêmes primitives cryptographiques que celles implémentées par défaut dans le prototype de BFT-smart, respectivement Hmac :MD5 pour les MACs, and RSA :Sha1 pour les signatures digitales.

Les figures 3.5 et 3.6 présentent la latence et le débit des protocoles Aardvark, COP, et BFT-smart lorsque ceux-ci utilisent des MACs ou des signatures pour l'authentification des requêtes. Ces expériences sont effectuées sur le micro-benchmark 0/0. Nous constatons que l'utilisation de signatures dégrade à la fois le débit et la latence, quel que soit le protocole impliqué.

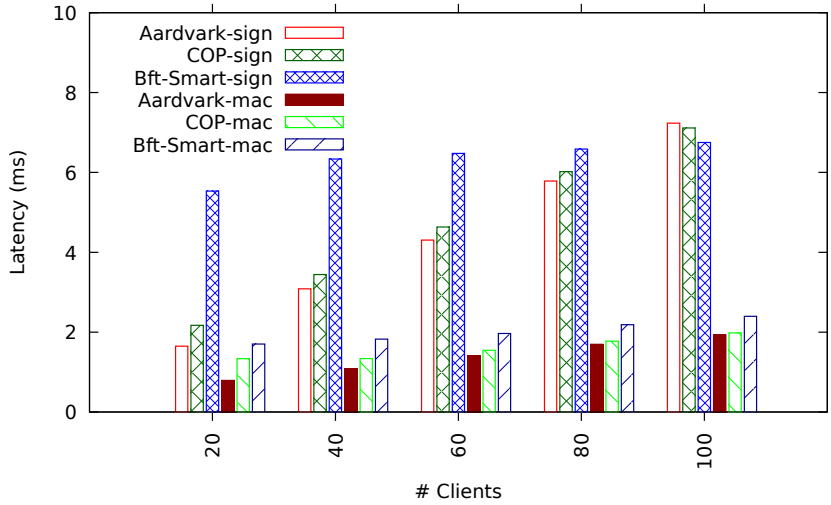

FIGURE 3.5 – Latence de plusieurs protocoles de BFT exécutant le micro-benchmark 0/0 en boucle close [21]. L'authentification des requêtes est effectuée en utilisant des MACs ou des signatures digitales.

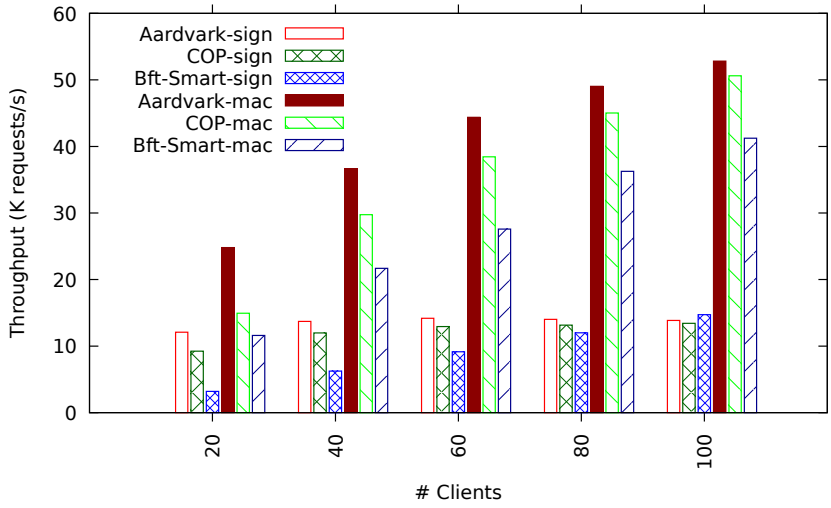

FIGURE 3.6 – Débit de plusieurs protocoles de BFT exécutant le micro-benchmark 0/0 en boucle close [21]. L'authentification des requêtes est effectuée en utilisant des MACs ou des signatures digitales.

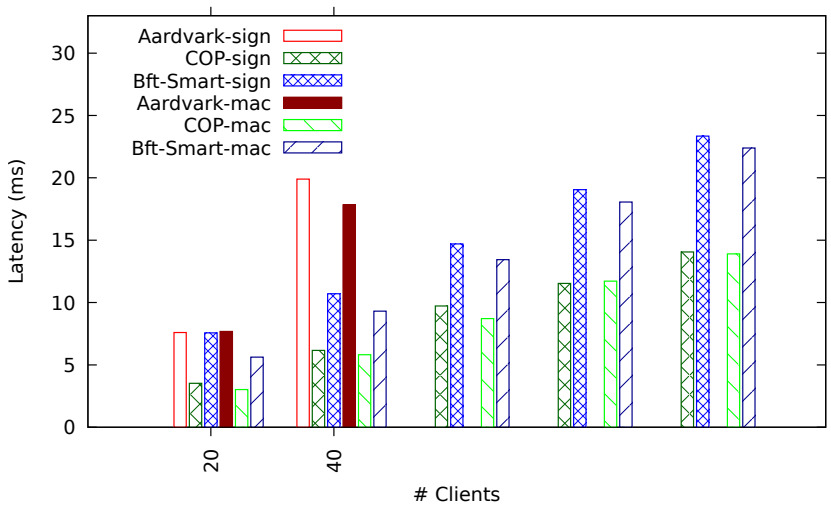

FIGURE 3.7 – Latence de plusieurs protocoles de BFT exécutant le micro-benchmark 4/4 en boucle close [21]. L'authentification des requêtes est effectuée en utilisant des MACs ou des signatures digitales.

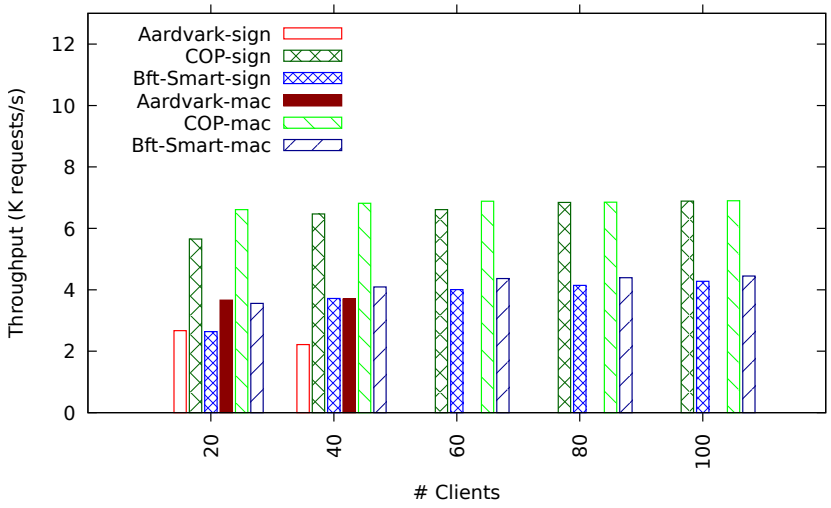

FIGURE 3.8 – Débit de plusieurs protocoles de BFT exécutant le micro-benchmark 4/4 en boucle close [21]. L'authentification des requêtes est effectuée en utilisant des MACs ou des signatures digitales.

Les figures 3.7 et 3.8 présentent la latence et le débit des protocoles Aardvark, COP, et BFT-smart lorsque ceux-ci utilisent des MACs ou des signatures pour l'authentification des requêtes. Ces expériences sont effectuées sur le micro-benchmark 4/4. Nous constatons que la consommation supplémentaire de bande passante liée à la taille des requêtes et des réponses limite le débit maximal des trois protocoles. Dans de telles conditions, la performance se voit davantage influencée par la taille des messages que par la méthode d'authentification sousjacente.

3.2.2 La Tolérance aux Fautes Byzantines en Pratique

De nombreux protocoles n'intègrent aucun mécanisme de robustesse. Le tableau 3.1 illustre la capacité des protocoles de BFT contemporains à tolérer l'occurrence de 5 comportements byzantins différents.

TABLE 3.1 – Comparaison de plusieurs protocoles de BFT selon leur capacité à tolérer efficacement l'occurrence de comportements byzantins

	Attaques				
	Réplica inactif	Délai intentionnel	Inondation du réseau (*flooding*)	Attaque MAC	indifférence sélective du *primary*
PBFT [21]	oui	non	non	non	non
Q/U [4]	oui	-	non	non	-
Zyzzyva [43]	oui	non	non	non	non
Ring [38]	oui	non	non	oui	non
*h*BFT [32]	oui	non	non	non	non
HQ [28]	oui	non	non	non	non
Quorum [37]	non	-	non	non	-
Chain [37]	non	non	non	non	non
Aliph [37]	oui	non	non	non	non
Adapt [14]	oui	non	non	non	non
BFT-SMaRt [18]	oui	non	non	oui	non
COP [16]	oui	non	non	non	non
OBFT [66]	oui	non	non	-	non
MinBFT [72]	oui	non	non	oui	non
MinZyzzyva [72]	oui	non	non	oui	non
BFT-TO [27]	oui	-	non	non	-
CheapTiny [40]	non	non	non	non	non
CheapBFT [40]	oui	non	non	oui	non
BFT2F [52]	oui	non	non	oui	non
Scrooge [65]	oui	non	non	non	non
Zeno [68]	oui	non	non	non	non
Prime [7]	oui	oui	oui	oui	oui
Aardvark [26]	oui	oui	oui	oui	non
Spinning [71]	oui	oui	non	non	oui
BFTMencius [57]	oui	oui	non	non	non
RBFT [10]	oui	oui	oui	oui	oui
R-Aliph [9]	oui	oui	oui	oui	non

Chapitre 4

Principes de Conception de Protocoles *Efficaces* et *Robustes*

Sommaire

Cette section présente nos contributions sous la forme de plusieurs principes de conception. L'objectif de ceux-ci est de faciliter le design de protocoles de BFT à la fois *robustes* et *efficaces*. Nous nous proposons de tolérer les deux attaques présentées dans la section 3.1 sans recourir aux solutions coûteuses de l'état de l'art. Premièrement, nous retirons aux clients la capacité de déclencher des changements de vues, en évitant le recours systématique aux signatures digitales. Deuxièmement, nous empêchons un *primary* malveillant de forcer la retransmission des requêtes sans introduire d'étape additionnelle. Nous proposons pour ce faire une nouvelle politique de gestion des requêtes et de changement de vues. Nous ne modifions pas le coeur du consensus, permettant ainsi à notre approche d'être adaptée à la plupart des protocoles de BFT existants.

4.1 Associer Efficacité et Robustesse

Nous présentons brièvement nos trois contributions majeures quant au design de protocoles à la fois *robustes* et *efficaces*.

4.1.1 Désactiver les Attaques MACs

1. Les changements de vues ne sont plus déclenchés lorsqu'une requête authentifiée via MAC n'est pas proposée pour exécution.

Habituellement, les changements de vues sont déclenchés par des *backups* corrects lorsqu'un consensus n'est pas obtenu à temps. En pratique, un minuteur est démarré sur chaque réplica s'attendant à l'exécution d'une requête. Lorsque ce minuteur est écoulé, les réplicas diffusent un message de VIEW-CHANGE aux autres réplicas. Si plusieurs réplicas transmettent ce message, ceux-ci seront en mesure de collecter suffisamment de VIEW-CHANGE pour remplacer le *primary*. Supprimer la capacité des protocoles à déclencher des changements de vues basés sur les requêtes authentifiées via MAC résout par conséquent le problème des attaques MACs, mais retire également aux *backups* corrects la possibilité de remplacer un *primary* inactif.

4.1.2 Remplacer les *Primaries* Inactifs

2. Les clients utilisent les signatures digitales pour la **retransmission** des requêtes. Les changements de vues sont déclenchés par les *backups* lorsqu'une requête *signée* n'est pas proposée pour exécution.

Les clients peuvent être amenés à retransmettre leurs requêtes pour plusieurs raisons. Le réseau peut échouer à distribuer les messages (cf. section 2.1.2.2), ou le *primary* peut refuser de faire suivre les requêtes aux *backups* (cf. section 3.1.2). En pratique, un minuteur est démarré sur chaque client en attente d'une réponse. Si le minuteur s'écoule avant réception de la réponse escomptée, le client retransmet la requête à l'ensemble des réplicas, qui la font suivre au *primary*. Cette solution, associée a la procédure de changement de vue, permet aux protocoles de BFT d'assurer la propriété de *vivacité* dans un environnement ultimement synchrone. Afin de fournir les mêmes garanties, nous utilisons également le mécanisme de retransmission. Nous authentifions néanmoins les requêtes retransmises en utilisant des signatures digitales. Enfin, contrairement au requêtes authentifiées via MACs, les requêtes *signées* retransmises peuvent déclencher des changements de vues si les réplicas corrects ne parviennent pas à obtenir de consensus. Un *primary* malveillant peut toutefois forcer la retransmission des requêtes authentifiées via MACs en requêtes *signées* (cf. section 3.1.2).

4.1.3 Substituer les *Primaries* Malveillants

3. Les réplicas surveillent le nombre de requêtes *signées* exécutées, en fonction des client et des *primaries*. Un changement de vue est déclenché si trop de clients retransmettent leurs requêtes.

Afin d'interdire aux *primaries* malveillants de forcer la retransmission des requêtes, les réplicas enregistrent plusieurs informations en temps réel (*monitoring*). Ceux-ci contrôlent la proportion de requêtes *signées* sur l'ensemble des requêtes récemment exécutées. Si cette proportion devient trop élevée, le *primary* sera suspecté et remplacé.

4.2 Gestion des Requêtes

Cette section décrit en détail la nouvelle politique de gestion des requêtes proposée. L'évolution du service dupliqué s'effectue par l'intermédiaire de requêtes transmises par les clients. Respectivement, un client c requiert l'exécution d'une opération via l'envoi d'un message $< request > \overrightarrow{\mu}_c$ au *primary*. L'authentification est effectuée en utilisant un *authenticator* $\overrightarrow{\mu}_c$ (cf. section 3.1.1.1), et la requête n'est transmise qu'au *primary*. Lors de l'émission d'une requête, c démarre un minuteur dédié T_c. Si le client c ne reçoit pas de réponse avant que le minuteur T_c n'arrive à son terme, c retransmet le message $< request >_{\sigma c}$ à tous les réplicas. Lors de la retransmission, la requête est envoyée à l'ensemble des réplicas, et l'*authenticator* $\overrightarrow{\mu}_c$ précédemment utilisé

est remplacé par une signature digitale σc. Ce mécanisme, combiné à la politique de changement de vue décrite dans la section 4.3, permet aux clients corrects de s'assurer de l'exécution des requêtes.

Primaries et *backups* ne réagissent pas de la même manière à la réception d'une requête transmise par un client. Lorsqu'une requête est reçue par le *primary* p, p lui associe un numéro de séquence afin d'initier le consensus. les *backups* b rejettent systématiquement les requêtes authentifiées via MACs lors de leurs réceptions. Les requêtes *signées* sont en revanche transférées au *primary* p, et un minuteur T_v est démarré. Ce minuteur est nécessaire afin de remplacer les *primaries* inactifs, le mécanisme en question est exposé plus en détail dans la section suivante.

4.3 Changement de Vues

En raison de sa responsabilité dans l'affectation des numéros de séquence aux requêtes, le *primary* possède une capacité unique de contrôle sur les progrès du système, et d'équité envers les clients. La procédure de changement de vue est destinée à assurer la propriété de *vivacité*, en permettant au système de poursuivre son évolution si le *primary* n'est plus apte à remplir sa fonction.

4.3.1 *Primaries* Inactifs

La procédure de changement de vue est déclenchée par des minuteurs sur les *backups*, empêchant ceux-ci d'attendre indéfiniment l'exécution des requêtes. Lors de la réception d'une requête *signée* par un *backup*, un minuteur T_v est démarré afin de s'acquitter de deux fonctions respectives. T_v permet d'abord d'assurer au *backup* que le *primary* recevra une instance de la requête *signée*. Il permet par la suite de remplacer le *primary* s'il est inactif. Si le minuteur T_v s'écoule avant l'exécution de la requête *signée* associée, cette requête est transférée au *primary* par le *backup* lui-même. Ce dispositif permet au *backup* de s'assurer que le *primary* possède bel et bien une instance de la requête en question, même si un client malveillant ignore intentionnellement le *primary* lors de la diffusion d'une requête. D'autre part, cela nous permet d'éviter le transfert systématique de toutes les requêtes reçues par les *backups*. Si la retransmission d'une requête par un client n'a pas pour moteur un comportement malveillant (e.g. perte de message liée à une défaillance du réseau), celle-ci se verra exécutée avant l' expiration de T_v, évitant ainsi un gaspillage inutile de bande passante.

Une fois le minuteur T_v écoulé et la requête signée transmise au *primary*, T_v se voit réinitialisé afin de pouvoir assurer sa seconde fonction : remplacer le

primary s'il est inactif. Si T_v s'écoule pour la deuxième fois avant l'exécution de la requête *signée* qui lui est associée, le *backup* conclut que le *primary* est inactif, et déclenche localement la procédure de changement de vue. Il convient de remarquer que seules les requêtes *signées* peuvent désormais déclencher cette procédure. Autrement dit, contrairement à PBFT [21], les minuteurs liés aux changements de vues ne sont plus démarrés lors de la réception de requêtes authentifiées via MACs, mais seulement à la suite de la retransmission de ces requêtes, authentifiées via signatures digitales. Cette solution fournit de bonnes performances en absence de faute grâce à l'utilisation des codes d'authentification de messages, en conservant la capacité de remplacer les *primaries* inactifs. Nous bénéficions de la propriété de non-répudiation comme les protocoles *robustes* [10, 26], sans recourir à l'utilisation systématique des signatures digitales.

4.3.2 *Primaries* Malveillants

Afin d'interdire aux *primaries* malveillants de forcer la retransmission systématique des requêtes, et ce sans imposer la présence d'étapes additionnelles liées à l'échange des requêtes en question, les *backups* doivent posséder la capacité de remplacer ces *primaries* malveillants. Dès lors que le nombre de clients corrects retransmettant leurs requêtes dépasse un seuil défini, le *primary* est suspecté et remplacé. Pour y parvenir, nous contrôlons le nombre de requêtes *signées* exécutées sur les réplicas, par clients et *primaries*. Fixer une valeur appropriée au seuil est une tâche difficile, car le modèle considéré implique une quantité indéterminée de clients byzantins. Ces clients byzantins peuvent intentionnellement émettre des requêtes *signées* aux *primaries* corrects afin de rendre ces derniers suspects (au lieu de seulement signer les requêtes retransmises). Nous décrivons dans cette section une manière distribuée de prédire la proportion de clients byzantins auxquels chaque *primary* devra faire face. La valeur des seuils évolue dynamiquement en accord avec ces prédictions, permettant ainsi aux protocoles de faire face efficacement à la fois aux *primaries* et aux clients malveillants.

4.3.2.1 Seuils de confiance

Les réplicas maintiennent $n = 3f + 1$ seuils de confiance $T_P(p)$. Ces seuils permettent de déterminer la proportion de clients malveillants auxquels chaque *primary* p sera supposé faire face. Lors de l'initialisation du système, une valeur est attribuée par défaut aux seuils $T_P(p)$, selon les estimations de l'utilisateur. Après l'exécution d'un nombre de requêtes S dans la vue v, où pv est le *primary* de v, tous les *backups* b mettent à jour la valeur de leurs

seuils $T_P(pv)$ de la manière suivante : $T_P(pv) = \lceil (x + T_P(pv))/2 \rceil$.

x est respectivement égal à la plus faible valeur parmi les n $T_P(p)$ associés à chaque p sur b, ou 0 si pv est *primary* pour la première fois. Le nombre de requêtes S est un paramètre lié à l'application dupliquée et doit être choisi en fonction du débit du service, afin de contrôler la vitesse à laquelle les seuils seront amenés à évoluer. Le consensus garantissant la propriété de *sûreté* (cohérence de l'état des réplicas corrects), notre politique garantit également que chaque réplica correct possède une valeur cohérente pour les seuils $T_P(p)$. Les seuils de confiance, combinés à la politique de suspicion définie ci-dessous, nous permettent de déterminer de manière distribuée la proportion de clients byzantins auxquels les *primaries* doivent faire face. Une fois ces informations disponibles, il est possible d'interdire aux *primaries* malveillants de forcer la retransmission des requêtes transmises par les clients corrects.

4.3.2.2 Politique de suspicion

Afin d'évaluer l'intégrité des clients, nous utilisons T_C, le pourcentage présumé de messages perdus lorsque les clients et le *primary* sont corrects. T_C est nécessaire en raison des hypothèses liées à la fiabilité du réseau, définies en section 2.1.2.2. Si l'on considère que le réseau peut être responsable de la perte d' 1% des messages en pratique, la valeur de T_C est attribuée à 1. Afin d'évaluer la propension des clients à être fautifs, chaque réplica b surveille, par client c et *primary* p, la proportion de requêtes *signées* $Signed_r(c, p)$ parmis les D dernières requêtes exécutées (ne sont considérées dans D que les requêtes transmises par c et ordonnées par p). Dans les expériences présentées en sections 6 et 8, nous considérons $D = 100$ afin de conserver notre attention sur l'occurrence d'attaques en temps réel. Dans ces scénarios, $Signed_r(c, p)$ fait donc référence au nombre de requêtes **signées**, transmises par c, ordonnées par p, et correctement exécutées parmis les (au maximum) 100 dernières requêtes transmises par c et ordonnées par p.

En pratique, le choix de la valeur D doit être effectué en accord avec la nature de l'application dupliquée. L'utilisation de signatures pouvant être induite à la fois par des *primaries* malveillants ou par les clients eux-mêmes, notre politique de suspicion doit être suffisamment fiable pour faire la distinction entre ces deux cas. Afin d'y parvenir, nous distinguons trois statuts différents liés aux comportements des clients et des *primaries*, respectivement *correct*, *suspect* et *fautif*.

— Un **couple** < Client c, Primary p > se comporte correctement depuis la perspective d'un réplica b si $Signed_r(c, p)$ est égal ou inférieur au seuil

T_C. Autrement dit, si moins de T_C pour cent des requêtes exécutées furent transmises par c et authentifiées via signatures digitales sous la direction du *primary* p.

— Un **couple** < Client c, Primary p > devient *suspect* depuis la perspective d'un réplica b si $Signed_r(c, p)$ dépasse le seuil T_C.

— Enfin, un **client** c est considéré *fautif* depuis la perspective d'un réplica b si plus de f $Signed_r(c, p)$ dépassent le seuil T_C. Autrement dit, lorsque que le nombre de *primaries* ordonnant les requêtes signées d'un client c devient supérieur au nombre maximal de *primaries* potentiellement fautifs [1].

le nombre de requêtes signées $Signed_r(c, p)$ peut potentiellement impliquer des requêtes exécutées par le couple < Client c, Primary p > dans des vues différentes. Un tel cas peut se produire si p devient *primary* à plusieurs reprises et que le client c envoie un nombre de requêtes inférieur à D dans une vue spécifique. Il est important de conserver la valeur de $Signed_r(c, p)$ au fil des différentes vues traversées afin de borner l'impact d'éventuels éléments byzantins.

4.3.2.3 Substitution des *primary* suspects

Au cours de la vue v, ou p est le *primary* de v, lorsque le ratio S_c de couples *suspects* < Client c, Primary p > dépasse le seuil $T_P(p)$ sur le *backup* b, celui-ci diffuse un message signé *suspicion-view-change* à l'ensemble des réplicas. Ce mécanisme est ajouté aux mécanismes de changement de vues pré-existants, et n'interdit pas à b de progresser davantage dans la vue v après l'envoi des messages *suspicion-view-change*, contrairement aux mécanismes de changements de vues habituels dans les protocoles s'inspirant de PBFT [21]. Lorque le *primary* responsable de la vue $v+1$ reçoit $2f+1$ messages *suspicion-view-change* valides (incluant potentiellement son propre message), il diffuse un message *suspicion-new-view* contenant les $2f+1$ messages *suspicion-view-change* signés. Ces $2f+1$ messages fournissent une preuve de validité quant au message *suspicion-new-view* auquel ils sont associés. Lors de la réception d'un message *suspicion-new-view* valide, les réplicas entrent dans la vue $v+1$ et mettent à jours leurs seuils $T_P(p)$ à la valeur $\lceil S_c \rceil$. Seuls les **couples** < Client c, Primary p > *suspects* sont considérés par le mécanisme additionnel de changement de vues, afin d'interdire aux **clients** reconnus *fautifs* d'impacter le seuil $T_P(p)$.

1. La borne f est justifiée par l'hypothèse selon laquelle au plus f réplicas peuvent être fautifs parmi les $3f+1$, comme décrit en section 2.1.2.2.

4.3.3 Interactions entre Clients et *Primary* Corrects

Nous avons présenté en section 4.2 le mécanisme de transmission des requêtes au système dupliqué. Celui-ci implique l'envoi exclusif des requêtes au *primary* courant. D'autre part, nos changements de vues se voient effectué sans l'intervention des clients (cf. section 4.3). Il est par conséquent envisageable qu'un client n'ait pas connaissance de l'identité du *primary* courant, et envoie une requête correctement authentifiée via MAC à un réplica qui n'est plus *primary*. Ce réplica se débarrassera de la requête, et forcera ainsi sa retransmission *signée* avant que le client ne prenne connaissance de l'identité du nouveau *primary* - une information qu'il obtiendra via la réception de la réponse associée à sa requête.

Si la première requête transmise de la part d'un client c dans la nouvelle vue est signée et exécutée, elle n'engendre pas la mise à jour du seuil de confiance $Signedr(c, p)$. Lorsqu'un réplica répond à un client suite à l'exécution d'une requête, ce réplica adjoint à sa réponse l'identifiant de la vue actuelle. Ce dispositif permet ainsi aux clients de connaître le *primary* courant, auquel ils devront adresser leurs futures requêtes.

4.4 Synthèse

Nous avons présenté dans ce chapitre les diverses contributions proposées afin de permettre la conception de protocoles de BFT à la fois *robustes* et *efficaces*. Nous traitons respectivement deux types de dénis de services : l'attaque MAC, et l'indifférence sélective du *primary* aux requêtes. Afin d'éviter le recours aux solutions coûteuses de l'état de l'art, nous avons proposé plusieurs principes de conception génériques. Concrètement, nous résolvons le problème de l'attaque MAC sans recourir à l'utilisation systématique des signatures digitales, et le problème du *primary* malveillant sans pratiquer l'échange systématique des requêtes. Les principes de conception résultants ne s'appliquent pas au cœur même du consensus, ce qui permet leur intégration à nombre de protocoles existants.

Chapitre 5

Conception du protocole ER-PBFT

Sommaire

Ce chapitre introduit le premier cas d'usage exploré. Nous implémentons nos différents principes de conception sur le consensus de PBFT, initialement proposé par Castro et al. en 1999 [21]. Nous décrivons en détail le fonctionnement d'ER-PBFT, un nouveau protocole à la fois *robuste* et *efficace*.

5.1 Motivation

ER-PBFT implémente le consensus original de PBFT [21]. Il s'agit de notre premier cas d'usage lié à l'intégration et l'évaluation des principes de conception décrits en section 4. Le consensus originellement implémenté par PBFT est le cœur même de nombreux protocoles de tolérance aux fautes byzantines. Tous les protocoles *robustes* présentés en section 2.2.2.2 utilisent ce même consensus, enrichi de plusieurs mécanismes dédiés à minimiser l'impact des comportements byzantins. Notre choix s'est donc rationnellement porté vers ce même consensus, nous permettant ainsi d'effectuer des comparaisons pertinentes avec d'autres protocoles *robustes* de l'état de l'art.

5.2 Description du Protocole

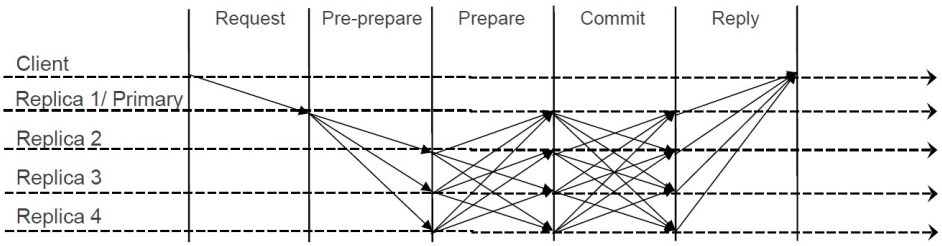

FIGURE 5.1 – Modèle de communication d'ER-PBFT

La figure 5.1 présente le consensus qu'ER-PBFT partage avec PBFT et ses nombreux successeurs [7, 10, 26, 27, 32, 57, 71, 72]. Tout comme ceux-ci, nous fournissons la propriété de tolérance aux fautes byzantines par l'intermédiaire de trois procédures distinctes, respectivement la *transmission des requêtes*, le *consensus*, et le *changement de vue*.

5.2.1 Transmission des Requêtes

La mise à jour de l'état de l'application dupliquée s'effectue par l'intermédiaire de *requêtes* transmises par les clients :

1. Un client envoie un message *requête* au *primary*

Un client c requiert l'exécution d'une opération o sur le système dupliqué en envoyant un message *requête* $< REQUEST, o, t, c > \overrightarrow{\mu}_c$ au réplica qu'il présume être le *primary* (cf. section 4.3.3). t est l'horodatage de la requête et permet d'assurer son unicité. t permet par exemple de distinguer deux requêtes différentes émises par le même client et impliquant la même opération o. L'authentification de la requête est effectuée grâce à un *authenticator* $\overrightarrow{\mu}_c$.

Si le client c ne reçoit pas de réponse en temps requis, il retransmet la requête $< REQUEST, o, t, c >_{\sigma c}$ à tous les réplicas. Lors de la retransmission, l'*authenticator* $\overrightarrow{\mu}_c$ précédemment utilisé est remplacé par une signature digitale σc, et la requête est cette fois ci destinée à l'ensemble des réplicas. Ce dispositif permet de garantir l'exécution à terme de toutes les requêtes (cf. section 4.3).

Comme énoncé en section 4.2, les *primaries* et *backups* ne réagissent pas de la même manière à la réception d'une requête transmise par un client. Lorsque l'authentification des requêtes est réussie, le *primary* associe celles-ci à un message *pre-prepare*, nécessaire au consensus décrit en section 5.2.2. Les *backups* rejettent systématiquement les requêtes authentifiées via MACs, et peuvent être amenés à transférer les requêtes *signées* au *primary* si la procédure de changement de vue est instanciée (cf. section 4.3). Si une requête doit être transférée au *primary* p par un *backup* b, celle-ci est ré-authentifiée avec le MAC $\overrightarrow{\mu_{b,p}}$ partagé entre b et p.

5.2.2 Consensus

2. Le *primary* génère un message *pre-prepare* associé à un ensemble de requêtes valides, puis diffuse le message *pre-prepare* à tous les *backups*

Le *primary* p créé et transmet un message $< PRE-PREPARE, v, n, < REQUEST, o, t, c > \overrightarrow{\mu}_c > \overrightarrow{\mu}_p$ à tous les *backups*, où v fait référence au numéro de vue courant, et n est le numéro de séquence associé au *pre-prepare*. Comme toute communication inter-réplicas, les messages *pre-prepare* reposent sur l'utilisation d'*authenticators*. Pour des questions de commodité, nous n'avons associé qu'une seule requête au message *pre-prepare*, mais de multiples requêtes peuvent être associées à un unique *pre-prepare* (*batching*). Ces requêtes peuvent indépendamment être authentifiées via l'utilisation de MACs ou de signatures digitales par les clients.

3. Les *backups* reçoivent et authentifient les messages *pre-prepare* en provenance du *primary*, puis transmettent des messages *prepare* aux autres réplicas

Lors de la réception d'un message $< PRE-PREPARE, v, n, < REQUEST, o, t, c > \overrightarrow{\mu}_c > \overrightarrow{\mu}_p$ de la part du *primary* p, un *backup* b vérifie d'abord qu'aucune autre requête (ou lot de requêtes) avec $n' = n$ n'a déjà été exécutée durant la vue v. Si c'est le cas, b se défausse du message *pre-prepare*. Dans le cas contraire, b valide l'authenticité du message en vérifiant le MAC qui lui est dédié parmi l'*authenticator* $\overrightarrow{\mu}_p$, puis vérifie la validité de la/des requête(s) associée(s).

Si les authentifications sont réussies, b conserve le message *pre-prepare* puis génère un message $< PREPARE, v, n, h, b > \overrightarrow{\mu}_b$, diffusé à tous les autres réplicas, où h est l'empreinte (*digest*) de l'ensemble des requêtes contenues dans le message *pre-prepare*. L'utilisation de cette empreinte permet d'éviter la retransmission des requêtes originales, afin de réduire la taille des messages *prepare* transitant sur le réseau.

4. Les replicas reçoivent $2f$ messages *prepare*, cohérents avec le message *pre-prepare* associé au numéro de séquence n, puis diffusent un message de *commit* aux autres réplicas

Suite a la réception de $2f$ messages *prepare* de la part de *backups*, cohérents avec le message *pre-prepare* précédemment accepté, le réplica r diffuse un message *commit* $< COMMIT, v, n, r > \overrightarrow{\mu}_r$ à tous les autres réplicas.

5.2.3 Exécutions et Réponses

5. Les réplicas qui reçoivent $2f + 1$ messages *commit* exécutent la requête, et transmettent finalement leurs *réponses* au client

La réception de $2f + 1$ messages $< COMMIT, v, n, r > \overrightarrow{\mu}_r$ cohérents de la part de réplicas distincts permet l'exécution de la/les requête(s). La *réponse* $< REPLY, v, t, c, i, r > \overrightarrow{\mu_{r,c}}$ est par la suite transmise au client c, où t est l'horodatage assurant l'unicité de la requête, i le résultat de l'exécution, et v l'identifiant de la vue courante. Ajouter l'information v à l'intérieur des *réponses* permet aux clients de connaître l'identité du *primary* courant (cf. section 4.3.3). Lorsqu'une requête est exécutée, plusieurs informations sont collectées afin de permettre l'implémentation de notre politique de suspicion, détaillée en section 4.3. Respectivement, nous mémorisons l'identité de l'expéditeur de la requête, et du *primary* responsable de son exécution. Nous

enregistrons également le mode d'authentification utilisé pour chacune des requêtes exécutées. Seules les informations relatives aux D dernières requêtes de chaque client sont nécessaires afin d'évaluer l'intégrité des clients et des réplicas.

6. Le client reçoit $f+1$ *réponses* cohérentes, qui garantissent l'exécution de sa requête

Dès lors qu'un client reçoit $f+1$ *réponses* $< REPLY, v, t, c, i, r > \overrightarrow{\mu_{r,c}}$ cohérentes, il peut conclure que sa requête fut exécutée par au moins 1 réplica correct, ce qui garantit son exécution à terme sur l'ensemble des réplicas corrects.

5.2.4 Changement de Vues

Afin d'assurer la propriété de *vivacité*, le protocole ER-PBFT implémente les mécanismes de changements de vues détaillés en section 4.3.

Chapitre 6

Evaluation du protocole ER-PBFT

Sommaire

Dans ce chapitre, nous confrontons expérimentalement ER-PBFT aux attaques présentées en section 3.1. Nous effectuons également plusieurs comparaisons mettant en scène d'autres protocoles *robustes* de l'état de l'art implémentant eux-aussi le consensus de PBFT, afin d'évaluer la performance de nos politiques de robustesse respectives.

6.1 Configuration expérimentale

Les expériences présentées dans cette section furent réalisées sur le *cluster* Grid5000 [20]. Chaque nœud possède deux processeurs Quad-Core Intel Xeon E5520, cadencés à 2.27 Ghz, 24 Go de RAM, et 64 GB de mémoire. Les machines communiquent via Ethernet 1GB et utilisent toutes le noyau Debian wheezy. Nous utilisons le micro-benchmark 0/0 [21], les requêtes et réponses ont donc pour tailles respectives 0 KB. Les comparaisons réalisées dans cette section mettent en scène les protocoles aardvark [26], et RBFT [10]. Ces comparaisons sont effectuées via les implémentations originales des deux prototypes. Une phase de préparation (*warmup*) de 180 secondes est effectuée préalablement à chacune de nos expériences. Nous employons les mêmes primitives cryptographiques que celles utilisées en section 3.2 sur l'ensemble des prototypes, respectivement Hmac :MD5 pour les codes d'authentification de messages, et RSA :Sha1 pour les signatures digitales. Toutes les expériences sont effectuées avec $n = 4$ réplicas, soit la configuration requise pour tolérer la présence d'un réplica byzantin. Seules les $D = 100$ dernières requêtes exécutées par les couples $<$ Client c, Primary $p >$ sont considérées pour la mise en pratique de notre politique de suspicion (cf. section 4.3.2). La valeur de T_C est attribuée à 1 : on considère donc que seul 1% des messages seront involontairement perdus.

6.2 Evaluation

Cette section est dédiée à l'évaluation du protocole ER-PBFT. Nous présentons tout d'abord nos comparaisons face à deux protocoles robustes de l'état de l'art. Nous évaluons ensuite ER-PBFT en présence de comportements byzantins, respectivement d'attaques MAC (cf. section 3.1.1) et de *primaries* malveillants (cf. section 3.1.2).

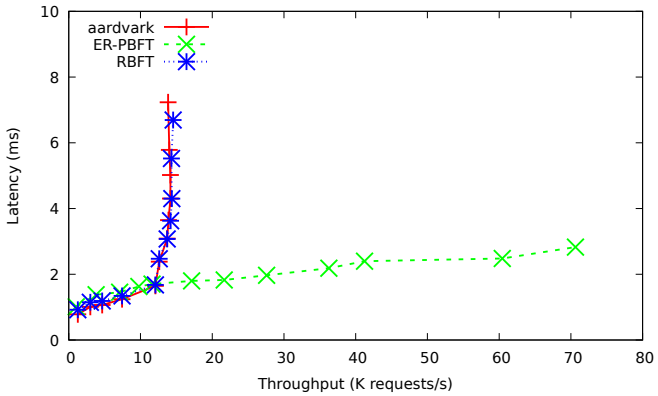

FIGURE 6.1 – Latence en fonction du débit pour plusieurs protocoles *robustes*

6.2.1 En absence de faute

Les expériences présentées ci-dessous impliquent ER-PBFT, ainsi que deux protocoles *robustes*, aardvark et RBFT [10, 26] en absence de faute. Ces expériences furent réalisées en boucle ouverte par égard au design de RBFT [1]. En boucle ouverte, chaque client peut envoyer plusieurs requêtes de manière concurrente. Contrairement à une boucle close, les clients ne sont pas tenus d'attendre la réponse associée à la précédente requête pour en émettre une nouvelle. Effectuer les expériences en boucle ouverte permet de contrôler le flux de requêtes entrantes quelque soit le nombre de clients impliqués, et d'atteindre ainsi facilement un débit élevé.

Similairement aux résultats présentés en section 3.2, nous constatons dans le figure 6.1 que l'utilisation de signatures digitales réduit considérablement les performances d'aardvark et RBFT face à ER-PBFT. Nous observons que le débit maximal obtenu dans nos expériences pour les prototypes d'aardvark et RBFT est d'environ 15K requêtes/s, alors que celui d' ER-PBFT atteint 70K requêtes/s. Aardvark et RBFT partagent de nombreuses similitudes architecturales et démontrent le même comportement en absence de faute, ce qui est cohérent avec les résultats présentés dans [10].

6.2.2 En présence d'attaques MAC

Cette section est dédiée à l'évaluation d'ER-PBFT en présence de clients malveillants provoquants des attaques MACs (cf. section 3.1.1). Les expériences réalisées mettent en scène une proportion spécifique de clients fautifs.

1. Les mécanismes de robustesse de RBFT reposent sur l'imprévisibilité de la fréquence des requêtes reçues, or, dans le cas d'un système fonctionnant en boucle close, le *primariy* peut imposer lui-même cette fréquence

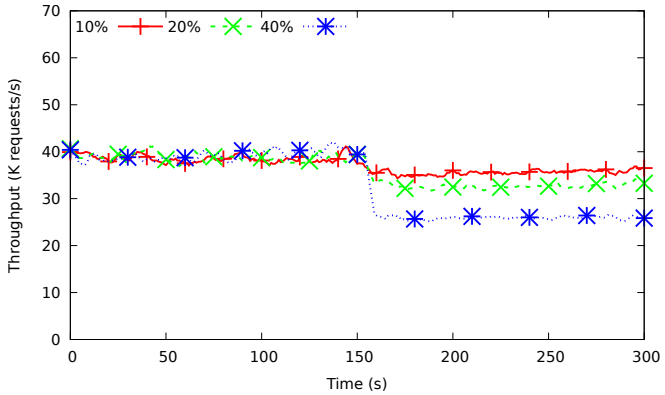

FIGURE 6.2 – Evaluation du débit sur le prototype d'ER-PBFT en présence d'attaques MAC

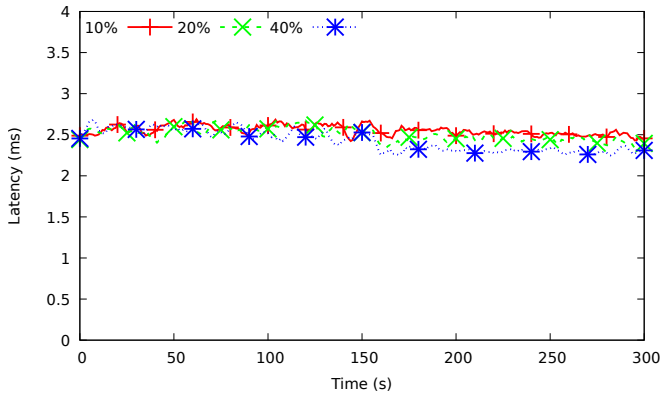

FIGURE 6.3 – Evaluation de la latence sur le prototype d'ER-PBFT en présence d'attaques MAC

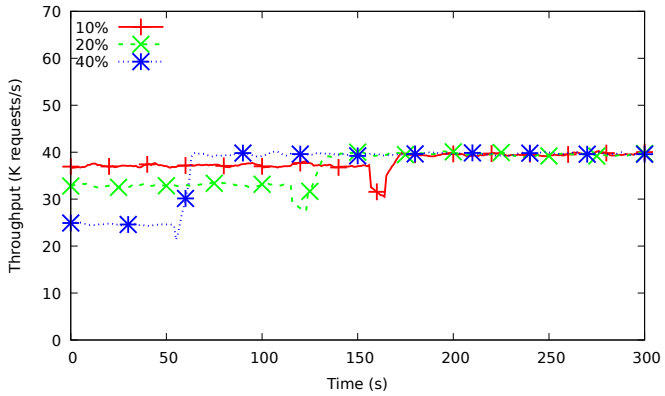

FIGURE 6.4 – Evaluation du débit sur le prototype d'ER-PBFT en présence d'un *primary* malveillant

Ces clients effectuent une corruption partielle de leurs *authenticators*, de telle sorte à rendre l'authentification des requêtes possibles pour les *backups*, mais impossible pour le *primary*. Afin de concentrer nos efforts d'évaluation sur l'attaque MAC et non l'inondation de messages corrompus (flooding), les clients fautifs ne transmettent leurs requêtes qu'une fois toutes les 10 secondes. Les expériences sont effectuées en boucle close et impliquent la présence de 100 clients distincts. A partir de t=150s, une partie des clients commence à provoquer des attaques MAC, respectivement 10%, 20%, ou 40%. Les figures 6.2 et 6.3 présentent le débit et la latence du prototype d'ER-PBFT. Nous observons qu'aucun changement de vue n'est déclenché par les clients fautifs, grâce à la mise en pratique de notre politique (cf section 4.3). D'autre part, nous constatons que le débit diminue lorsque la proportion de clients fautifs augmente, car moins de clients continuent d'envoyer des requêtes correctes durant l'attaque. La latence diminue en fonction de la proportion de clients fautifs, car les clients corrects reçoivent leurs réponses plus rapidement lorsque le système se voit confronté à une charge plus faible.

6.2.3 En présence de *primaries* malveillants

Nous évaluons dans ces expériences la performance d'ER-PBFT en présence de *primaries* malveillants. Ces *primaries* forcent la retransmission des requêtes envoyées par les clients (cf. section 3.1.2). Nos expériences se présentent de la manière suivante : un *primary* fautif ignore intentionnellement les requêtes authentifiées via MACs d'une portion spécifique de clients, et force ainsi leurs retransmissions en requêtes *signées*. Les expériences sont effectuées en boucle close et impliquent la présence de 100 clients distincts. Le

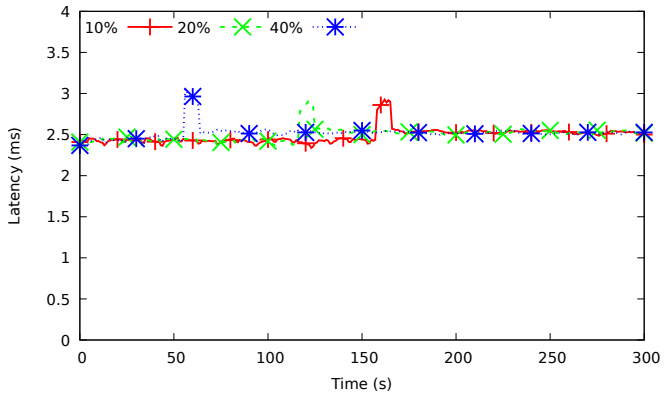

FIGURE 6.5 – Evaluation de la latence sur le prototype d'ER-PBFT en présence d'un *primary* malveillant

primary fautif ignore respectivement 10%, 20%, ou 40% des clients corrects. Afin d'exposer les effets d'une telle attaque sur notre protocole, nous ne permettons aux seuils $T_P(p)$ de n'évoluer que lorsque la phase de préparation est terminée. C'est dans cette même optique que nous choisissons les valeurs par défaut $T_P(p) = 50\%$ et $S = 2.000.000$. Autrement dit, dès lors que 2 millions de requêtes sont exécutées, la valeur de $T_P(p)$ se voie divisée par 2. Selon le nombre de clients ciblés et le débit exhibé par le prototype, les seuils $T_P(p)$ sont finalement dépassés et le *primary* remplacé. Après le changement de vue, le système recouvre sa performance originale (cf. figures 6.4 et 6.5).

6.3 Synthèse

Les chapitres 5 et 6 nous ont permis d'introduire ER-PBFT, soit l'implémentation de nos principes de conception sur le consensus original proposé par Castro et.al. [21]. Nous justifions tout d'abord le choix de PBFT comme premier cas d'usage car son consensus s'est vu réutilisé par l'ensemble des protocoles *robustes* présentés en section 2.2.2.2. Nous avons ensuite décrit le protocole en détail en section 5.2, puis nous avons présenté nos résultats expérimentaux en section 6. Nous confirmons l'efficacité d'ER-PBFT face à deux protocoles *robustes* : aardvark et RBFT [10, 26]. Finalement, nous avons également démontré sa capacité à tolérer les deux dénis de services considérés, respectivement l'attaque MAC (cf. section 3.1.1), et l'indifférence sélective du *primary* (cf. section 3.1).

Chapitre 7

Conception du protocole ER-COP

Sommaire

Ce chapitre introduit le second cas d'usage exploré. Nous implémentons nos différents principes de conception en s'inspirant de COP, un protocole très *efficace* proposé par Behl et al. en 2015 [16], qui introduit un nouveau degré de parallélisation. Nous décrivons en détail le fonctionnement d'ER-COP, et l'intégration de sa surcouche dédiée à la robustesse.

7.1 Motivation

ER-COP adapte notre politique de robustesse au design du protocole COP, présenté brièvement en section 2.2.1.3 [16]. COP repose sur la parallélisation de l'ensemble du consensus afin d'éviter une attribution séquentielle des ordres d'exécution. En pratique, la parallélisation telle qu'implémentée par COP lui permet d'afficher un niveau de performance jusqu'alors inégalé. COP propose à l'heure actuelle l'implémentation la plus efficace de tolérance aux fautes byzantines, et la plus à même d'assurer un passage à l'échelle des systèmes dupliqués (*scalability*). La parallélisation proposée par COP est implémentée sur le consensus original de PBFT [21]. Cette optimisation permet d'améliorer l'efficacité des protocoles de BFT, et peut théoriquement être appliquée à toutes sortes de consensus. Elle s'adapte donc particulièrement bien à notre politique de robustesse, permettant ainsi l'implémentation d'un protocole à la fois très *efficace* et *robuste* : ER-COP.

7.1.1 *Task-Oriented Parallelization*

Les approches traditionnelles ne permettent pas d'optimiser l'utilisation des architectures multi-cœurs modernes. En pratique, la parallélisation telle qu'implémentée par les protocoles de l'état de l'art se focalise sur les tâches à effectuer (*task oriented parallelization*) [62], et non les dépendances entre celles-ci. La figure 7.1 présente une parallélisation possible des tâches réalisées par un protocole de BFT contemporain. La séparation du consensus et de l'exécution du service dupliqué, illustrée en section 2.2.1.3 permet d'implémenter un premier degré de parallélisation. La parallélisation peut également être implémentée à l'intérieur même du consensus via sa scission en plusieurs modules, chargés d'exécuter des tâches spécifiques. Ces modules permettent par exemple de paralléliser la gestion des messages, en séparant les connections avec les clients de celles destinées aux autres réplicas (cf. CC et RC en figure 7.1). Des *threads* peuvent également être mis en place afin d'effectuer plusieurs opérations d'authentification en parallèle, permettant ainsi de rendre les protocoles plus *efficaces*. Behl et al. présentent un autre degré de parallélisation, plus adapté au contexte de duplication de machine à états : il s'agit

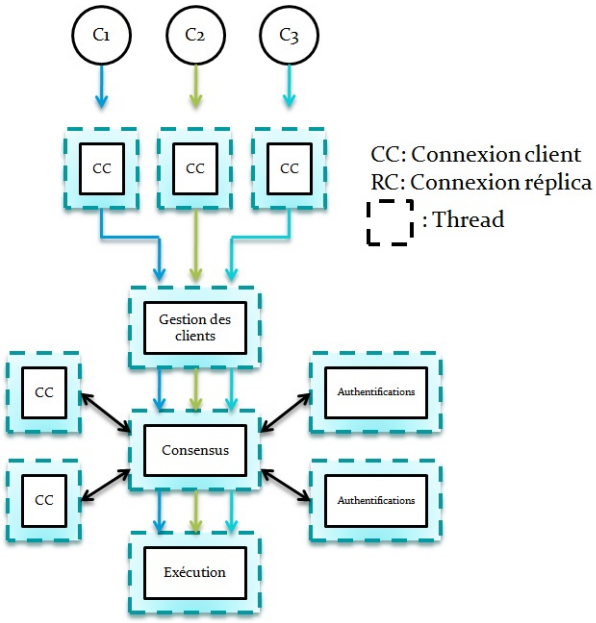

FIGURE 7.1 – Architecture d'un réplica mettant en scène les concepts de *task oriented parallelization*, tels qu'implémentés par les protocoles de BFT contemporains

de la parallélisation du consensus dans son ensemble [16].

7.2 Description du Protocole

7.2.1 Parallélisme et Robustesse

7.2.1.1 La Parallélisation du Consensus

La figure 7.2 présente le degré de parallélisation qu'ER-COP partage avec COP [16]. Cette figure illustre l'architecture interne d'un réplica. Dans cet exemple, le réplica en question utilise trois *threads* : deux *pillars*, et un *thread* dédié à l'exécution des requêtes. Seul le *thread* dédié à l'exécution est à même de modifier l'état du réplica. La fonction des *pillars* réside dans l'administration de l'ordre d'exécution des requêtes. Chaque *pillar* implémente pour ce faire l'ensemble des modules nécessaires au consensus. Un *pillar* fonctionne de manière asynchrone, sans interférer avec les autres *pillars* présents sur un même réplica (cf. figure 7.2).

Chaque *pillar* est associé à un panel de clients spécifiques. Au sein d'un réplica, seul un unique *pillar* sera chargé de gérer les requêtes transmises par les clients qui lui sont associés. Lorsqu'un *pillar* implémenté sur le *primary*

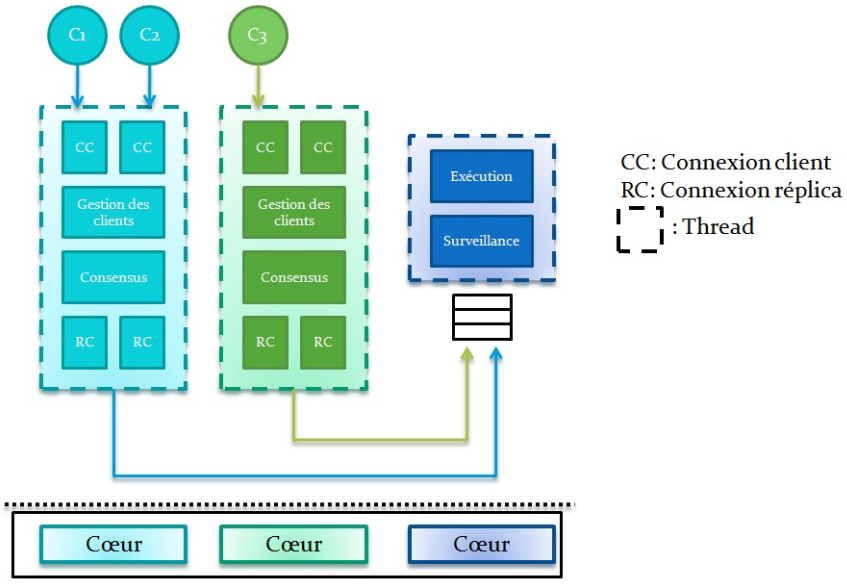

FIGURE 7.2 – Réplica implémentant la parallélisation orientée consensus d'ER-COP, contenant deux *pillars*

reçoit une requête de la part d'un client [1], il instancie un nouveau consensus. ER-COP implémente le consensus de PBFT, et suit la même logique que celle précédemment décrite en section 5.2.2, que nous ne détaillerons plus par la suite. Lorsqu'une instance du consensus est complétée sur un *pillar* (autrement dit lors de la réception de $2f + 1$ messages *commit*), celui-ci peut propager le numéro de séquence associé à la requête au *thread* dédié à l'exécution. Chaque *pillar* initiant des consensus de manière asynchrone, le *thread* dédié à l'exécution doit s'assurer de maintenir une cohérence quant à l'ordre global d'exécution des requêtes.

7.2.1.2 Préserver la Propriété de *Sûreté*

Les protocoles de BFT contemporains requièrent l'attribution d'un numéro de séquence à chaque requête (ou lots de requêtes - *batching*). Afin d'assurer la propriété de *sûreté*, l'exécution des requêtes doit s'effectuer en accord avec les numéros de séquence qui leurs sont associés. Il n'est en revanche pas nécessaire de s'accorder sur l'ordre dans lequel les numéros de séquence sont associés aux requêtes pour conserver la propriété de *sûreté*. Considérons par exemple 3 consensus instanciés en parallèle, $c1$, $c2$, et $c3$. $c1$ associe à sa requête l'ordre d'exécution 1, $c2$ l'ordre d'exécution 2, et $c3$ l'ordre d'exécution

1. ou cumule suffisamment de requêtes pour générer un lot (*batching*)

3. Si le consensus $c3$ se termine avant les consensus $c1$ et $c2$, la propriété de *sûreté* n'est pas remise en question. Il suffit au réplica d'attendre l'achèvement des consensus $c1$ et $c2$ pour pouvoir exécuter les trois requêtes dans l'ordre adéquat.

Afin de permettre à plusieurs *pillars* d'instancier leurs consensus indépendamment, deux contraintes doivent être respectées. Premièrement, il doit être possible pour chaque *pillar* de déterminer le prochain numéro de séquence à utiliser, et ce sans se concerter avec les autres *pillars*. Deuxièmement, chaque consensus doit indépendamment préserver la propriété de *sûreté*, autrement dit assurer l'exécution ordonnée et cohérente des requêtes sur l'ensemble des réplicas.

L'utilisation d'un compteur global, partagé entre les *pillars*, permettrait de résoudre ces deux contraintes. Cependant, son utilisation créerait un point de contention entre les *pillars*, ce que nous souhaitons éviter. Ces derniers devraient en effet faire référence au compteur à chaque instanciation d'un nouveau consensus. Afin de maintenir l'indépendance des *pillars*, nous utilisons le modèle de partitionnement implémenté dans COP. Celui-ci consiste en une distribution des numéros de séquence entre les *pillars* de manière prédéfinie. Si N_P *pillars* sont utilisés, chacun d'entre eux peut déterminer le numéro de séquence $seq(p, i)$ associé à son prochain consensus tel que $seq(p, i) = p + iN_P$, ou p fait référence à l'identifiant du *pillar*, et i est un compteur local incrémenté à chaque nouvelle instance de consensus.

7.2.1.3 L'Exécution des Requêtes

Afin de permettre l'exécution des requêtes sur le *thread* dédié, celui-ci doit disposer d'une suite continue de numéros de séquence. Les consensus étant instanciés de manière asynchrone, le *thread* dédié à l'exécution peut ne pas recevoir les requêtes dans l'ordre dans lequel elles se devront d'être exécutées. Ainsi, ce *thread* sera parfois tenu d'attendre la réception d'une requête pour poursuivre l'exécution. Ce point précis soulève un important problème compte tenu de l'affectation des clients à leurs *pillars* respectifs. Considérons l'exemple suivant : dans la figure 7.2, si le client C_3 cesse de transmettre des requêtes au second *pillar*, le *pillar* en question n'instanciera plus de consensus. Le *thread* dédié à l'exécution ne recevra donc pas le numéro de séquence manquant, et se verrait par conséquent incapable d'exécuter les requêtes transmises par les clients C_1 et C_2.

Afin de résoudre ce problème, nous permettons à un *pillar* d'instancier un consensus sans y associer de requête. Les réplicas peuvent donc s'accorder sur un numéro de séquence qui se devra d'être ignoré par les *threads* dédiés à

l'exécution. Néanmoins, afin d'éviter le gaspillage inutile des ressources disponibles, le système peut désactiver les *pillars* obsolètes, ou réaffecter certains clients à des *pillars* sous-utilisés.

Lorsqu'une requête est exécutée sur un réplica par l'intermédiaire du *thread* dédié, ce réplica collecte les informations nécessaires à notre politique de robustesse. Respectivement, nous mémorisons l'identité de l'expéditeur de la requête, et du primary responsable de son exécution. Nous enregistrons également le mode d'authentification utilisé pour chacune des requêtes exécutées. Seules les informations relatives aux D dernières requêtes de chaque client sont nécessaires afin d'évaluer l'intégrité des clients et des réplicas. Ces informations ne sont conservées que par le *thread* dédié à l'exécution, et ne sont pas partagées avec les *pillars*.

7.2.2 les Caractéristiques d'ER-COP

7.2.2.1 Passage à L'Échelle

Lorsque la parallélisation est implémentée via les concepts de *task oriented parallelization* (cf. figure 7.1), le débit du protocole est déterminé par le débit du module le plus lent, et ce même si des ressources supplémentaires demeurent disponibles. L'ordonnancement des requêtes étant effectué de manière séquentielle par les protocoles contemporains, le débit de ces protocoles n'est pas limité par le matériel mais bel et bien par l'implémentation du consensus. La parallélisation du consensus permet en revanche d'accroître le débit maximal en ajoutant des *pillars* supplémentaires, dès lors que l'exécution des requêtes ne borne pas ce débit. De même, si le flux de requêtes entrantes diminue, le nombre de *pillars* peut être réduit afin d'éviter une consommation inutile des ressources. Chaque *pillar* fonctionnant de manière indépendante, le surcoût lié à la synchronisation entre *threads* est minime, ce qui contribue à une meilleure utilisation du cache des processeurs. Enfin, chaque *pillar* incorporant les mêmes modules et réalisant les mêmes tâches, le design d'ER-COP assure une utilisation homogène des différents cœurs.

7.2.2.2 Applicabilité

Nos principes de conception définis en section 4, tout comme la parallélisation proposée par COP, sont applicables indépendamment au consensus sous-jacent. La parallélisation du consensus n'entraîne pas de variation conceptuelle quant aux consensus en question. Elle se révèle même plus simple à réaliser que la parallélisation par tâches, car chaque *pillar* implémente l'ensemble du consensus au sein d'un unique *thread*. La gestion des messages et des opé-

rations d'authentification associées est effectuée sur place, et n'implique donc pas de synchronisation entre les différents *threads*. Nos principes de conception sont parfaitement adaptables à la parallélisation proposée par COP. La mise à jour des informations requises par notre politique de robustesse ne concernant que les requêtes dont l'exécution est effective, notre surcouche peut être implémentée sur le *thread* dédié à l'exécution (cf. surveillance en figure 7.2), et n'interfère donc pas avec avec le fonctionnement des nombreux *pillars*.

Chapitre 8

Evaluation du protocole ER-COP

Sommaire

Dans ce chapitre, nous confrontons expérimentalement ER-COP aux diverses attaques présentées en section 3.1. Nous effectuons également plusieurs comparaisons mettant en scène le prototype original de COP face à ER-COP, afin d'évaluer le surcoût engendré par notre politique de robustesse.

8.1 Configuration expérimentale

Les expériences présentées dans cette section furent réalisées sur le *cluster* Grid5000 [20]. Chaque nœud possède deux processeurs Quad-Core Intel Xeon E5520, cadencés à 2.27 Ghz, 24 Go de RAM, et 64 GB de mémoire. Les machines communiquent via Ethernet 1GB et utilisent toutes le noyau Debian wheezy. Nous utilisons le micro-benchmark 0/0 [21], les requêtes et réponses ont donc pour tailles respectives 0 KB. Les comparaisons réalisées dans cette section mettent en scène le prototype original de COP [16]. L'implémentation d'ER-COP est effectuée par intermédiaire de ce même prototype, enrichi de notre politique de robustesse. Nous utilisons également une troisième version du prototype, COP-sign, ou l'authentification des requêtes est effectuée via signatures digitales. Nous employons les mêmes primitives cryptographiques que celles utilisées en section 3.2 sur l'ensemble des prototypes, respectivement Hmac :MD5 pour les codes d'authentification de messages, et RSA :Sha1 pour les signatures digitales. Une phase de préparation (*warmup*) de 180 secondes est effectuée préalablement à chacune de nos expériences. Tous les prototypes utilisés dans nos évaluations disposent de 16 *pillars* et appliquent l'optimisation de *batching*. Dans ces prototypes, la taille b attribuée aux lots de requêtes est un paramètre défini de manière statique. En d'autres termes, la valeur de b n'est pas dynamiquement déterminée en fonction du nombre de requêtes reçues. Toutes les expériences sont effectuées avec $n = 4$ réplicas, soit la configuration requise pour tolérer la présence d'un réplica byzantin. Seules les $D = 100$ dernières requêtes exécutées par les couples $<$ Client c, Primary $p >$ sont considérées pour la mise en pratique de notre politique de suspicion (cf. section 4.3.2). La valeur de T_C est attribuée à 1 : on considère donc que seul 1% des messages seront involontairement perdus.

8.2 Evaluation

Cette section est dédiée à l'évaluation du protocole ER-COP. Nous présentons tout d'abord nos comparaisons face au protocole COP original (cf. section 8.2.1). Nous évaluons ensuite ER-COP en présence de comportements byzantins, respectivement d'attaques MAC (cf. section 3.1.1) et de *primaries*

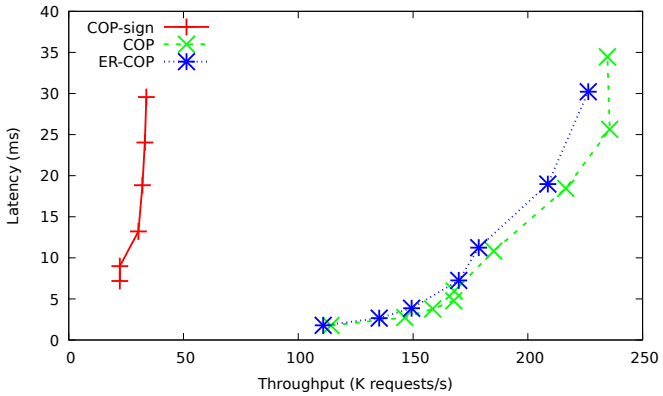

FIGURE 8.1 – Latence en fonction du débit pour COP, COP avec requêtes signées, et ER-COP avec des lots de taille $b = 10$

malveillants (cf. section 3.1.2).

8.2.1 En absence de faute

Les expériences présentées ci-dessous ont pour objectifs de quantifier le surcoût introduit par nos mécanismes de robustesse. Le protocole COP réalise l'authentification de ses requêtes via l'utilisation d'*authenticators*. Ces expériences furent réalisées en boucle ouverte, afin de permettre d'atteindre plus facilement le débit maximal offert par les prototypes. Dans les figures 8.1 et 8.2, COP fait référence à l'implémentation originale du protocole, et COP-sign au même prototype, où l'authentification des requêtes est effectuée grâce à des signatures digitales. b étant un paramètre statique, nous présentons deux batteries d'expériences, ou $b = 10$ (cf. figure 8.1), et $b = 100$ (cf. figure 8.2). L'utilisation des signatures digitales réduit drastiquement les performances de COP-sign, son débit maximal étant de $36K$ requêtes/s lors de l'utilisation de lots de 100 requêtes, face aux $388K$ requêtes/s fournit par COP. La comparaison entre ER-COP et COP révèle que l'intégration de nos principes de conception n'impliquent qu'une atténuation raisonnable des performance du prototype original.

8.2.2 En présence d'attaques MAC

Cette section est dédiée à l'évaluation de ER-COP en présence de clients malveillants provoquant des attaques MACs (cf. section 3.1.1). Les expériences réalisées mettent en scène une proportion définie de clients fautifs. Ces clients réalisent une corruption partielle de leurs *authenticators*, de telle sorte à rendre

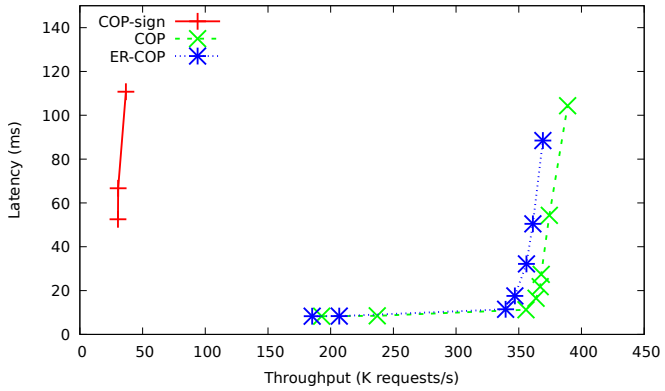

FIGURE 8.2 – Latence en fonction du débit pour COP, COP avec requêtes signées, et ER-COP avec des lots de taille $b = 100s$

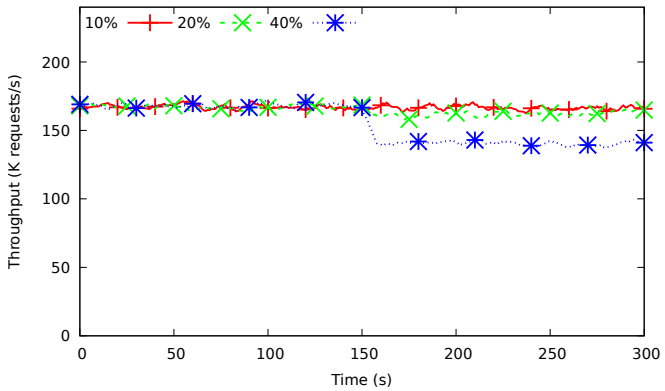

FIGURE 8.3 – Evaluation du débit sur le prototype d'ER-COP en présence d'attaques MAC

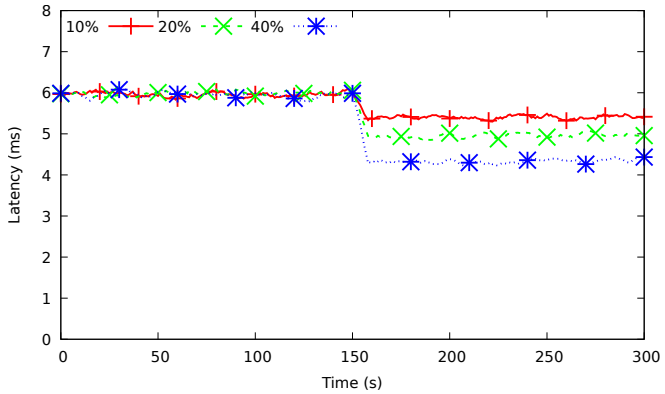

FIGURE 8.4 – Evaluation de la latence sur le prototype d'ER-COP en présence d'attaques MAC

l'authentification des requêtes possible pour les *backups*, mais impossible pour le *primary*. Afin de concentrer nos efforts d'évaluation sur l'attaque MAC et non l'inondation de messages corrompus (flooding), les clients fautifs ne transmettent leurs requêtes qu'une fois toutes les 10 secondes. La taille des lots de requêtes générées est $b = 10$. Les expériences sont effectuées en boucle close et impliquent la présence de 100 clients distincts. A partir de t=150s, une partie des clients commence à provoquer des attaques MAC, respectivement 10%, 20%, ou 40%.

Les figures 8.3 et 8.4 présentent le débit et la latence du prototype d'ER-COP. Les comportements observés sont cohérents avec ceux exhibés par le protocole ER-PBFT en présence d'attaques MAC : aucun changement de vue n'est déclenché par les clients fautifs, et le débit diminue lorsque la proportion de clients fautifs augmente, car moins de clients envoient des requêtes correctes durant l'attaque. Il convient en revanche de remarquer que la présence de 10% de clients fautifs n'a qu'un impact négligeable sur le débit, alors qu'à contrario la latence s'en trouve fortement impactée. Ces résultats sont cohérents avec ceux présentés en section 8.2.1, ou l'on observe une corrélation plus importante entre le flux de requêtes entrantes et la latence lorsque le débit dépasse $150K$ requêtes/s. La configuration du système implique l'obtention d'un tel débit dès le début de l'expérience, c'est pourquoi l'impact des attaques MAC est davantage perceptible sur la latence.

8.2.3 En présence de *primaries* malveillants

Nous évaluons dans ces expériences la performance d'ER-COP en présence de *primaries* malveillants. Ces *primaries* forcent la retransmission des requêtes envoyées par les clients (cf. section 3.1.2). Nos expériences se présentent de la manière suivante : un *primary* fautif ignore intentionnellement les requêtes authentifiées via MACs d'une portion spécifique de clients, et force ainsi leurs retransmissions en requêtes *signées*. Les expériences sont effectuées en boucle close et impliquent la présence de 100 clients distincts. Le *primary* fautif ignore respectivement 10%, 20%, ou 40% des clients corrects. Afin d'exposer les effets d'une telle attaque sur notre protocole, nous ne permettons aux seuils $T_P(p)$ de n'évoluer que lorsque la phase de préparation est terminée. C'est dans cette même optique que nous choisissons les valeurs par défaut $T_P(p) = 50\%$ et $S = 10.000.000$. Dès lors que 10 millions de requêtes sont exécutées, la valeur de $T_P(p)$ se voie divisée par 2. Selon le nombre de clients ciblés et le débit exhibé par le prototype, les seuils $T_P(p)$ sont finalement dépassés et le *primary* remplacé. Après le changement de vue, le système recouvre sa performance originale (cf. figures 8.5 et 8.6).

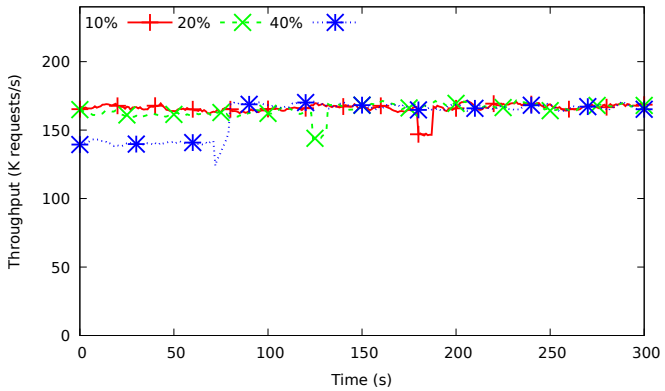

FIGURE 8.5 – Evaluation du débit sur le prototype ER-COP en présence d'un *primary* malveillant

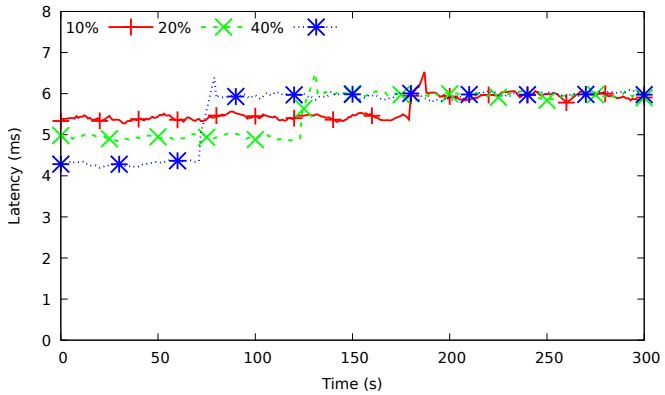

FIGURE 8.6 – Evaluation de la latence sur le prototype d'ER-COP en présence d'un *primary* malveillant

8.3 Synthèse

Les chapitres 7 et 8 nous permirent de présenter ER-COP, soit l'implémentation de nos principes de conception sur le consensus original proposé par Castro et.al. [21], en lui appliquant le degré de parallélisation proposé par COP [16]. Nous présentons tout d'abord le degré de parallélisation tel qu'appliqué par les prototypes contemporains, puis nous détaillons les spécificités conceptuelles d'ER-COP (cf. section 7.2.1). Nous décrivons ses principales caractéristiques en section 7.2.2 puis nous proposons l'évaluation expérimentale du prototype en section 8. Nous quantifions le surcoût introduit par nos mécanismes de robustesse via une comparaison entre ER-COP, COP, et une version alternative de COP ou les requêtes sont authentifiées via signatures digitales (cf. section 8.2.1). Finalement, nous avons également démontré dans les sections 8.2.2 et 8.2.3 la capacité d'ER-COP à tolérer les deux dénis de services considérés, respectivement l'attaque MAC (cf. section 3.1.1), et l'indifférence sélective du *primary* (cf. section 3.1).

Chapitre 9

Conclusions et Perspectives

Sommaire

9.1 Conclusions

Cette thèse se concentre sur la conception de protocoles de duplication de machines à états tolérant les fautes byzantines. Nous avons d'abord distingué les deux principales directions de recherche empruntées par l'état de l'art : les protocoles *efficaces* se fixent pour objectif d'optimiser les performances du système en absence de faute, alors que les protocoles *robustes* sont destinés à maintenir un degré de performance constant, à la fois en présence et en absence de nœuds byzantins. Relever ces deux défis à la fois est un véritable challenge : les protocoles actuels sont soit conçus pour être *efficaces* au détriment de leur *robustesse*, soit pour être *robustes* au détriment de leur *efficacité*.

L'étude des nombreux protocoles de l'état de l'art nous a permis de constater que les mécanismes impliqués dans la minimisation de l'impact des comportements malveillants se révèlent bien souvent très coûteux. Afin de d'atténuer les surcoûts associés à ces mécanismes de *robustesse*, nous avons porté notre intérêt vers deux types de dénis de service liés à la gestion des requêtes. Le premier de ces dénis de service est causé par la corruption partielle d'une requête lors de son émission par un client. Le deuxième est causé par l'abandon intentionnel d'une requête lors de sa réception par un réplica. Les protocoles *robustes* contemporains tolèrent ces deux dénis de services via la mise en place de solutions dédiées. Ils réalisent respectivement la substitution des codes d'authentification de messages au profit de signatures digitales pour l'authentification des requêtes, et l'échange systématique des requêtes dès leurs réceptions. En pratique, ces solutions impliquent d'importants surcoûts, c'est pourquoi nous nous proposons de définir plusieurs principes de conception génériques, permettant d'assurer un niveau de robustesse équivalent à moindre coût.

Nous implémentons ces principes de conception à travers deux nouveaux protocoles de tolérance aux fautes byzantines, respectivement ER-PBFT et ER-COP. ER-PBFT illustre l'efficacité de notre politique de robustesse associée au consensus proposé par Castro et al., largement réutilisé par de nombreux protocoles. Nous décrivons en détail le fonctionnement d'ER-PBFT, puis nous le confrontons expérimentalement aux deux dénis de services considérés. Nous effectuons également plusieurs comparaisons mettant en scène d'autres protocoles *robustes* de l'état de l'art implémentant eux-aussi le consensus de Castro, afin d'évaluer la performance de nos politiques de robustesse respectives. ER-COP consiste en l'implémentation de nos principes de conception lorsque associés au degré de parallélisation proposée par COP, un protocole très *efficace* proposé par Behl et al. Nous décrivons en détail le fonctionne-

ment d'ER-COP et l'intégration de notre surcouche dédiée à la robustesse, puis nous le confrontons expérimentalement aux deux dénis de services considérés. Nous effectuons également plusieurs comparaisons mettant en scène le prototype original de COP face à ER-COP, afin d'évaluer le surcoût engendré par notre politique de robustesse.

9.2 Perspectives

Ces travaux ouvrent la voie à une nouvelle manière de concevoir le design de protocoles de tolérance aux fautes byzantines. Alors que les technologies de virtualisation et de parallélisation permettent de produire des protocoles de plus en plus performants en pratique [16], il convient de conserver un degré de robustesse suffisant afin de permettre aux protocoles de BFT de tolérer efficacement l'occurrence de comportements byzantins. A l'heure actuelle, il demeure difficile de marier *efficacité* et *robustesse* au sein d'un unique protocole. Afin d'y parvenir, il est nécessaire de considérer ces deux contraintes dès l'étape de conception. Fournir des solutions efficaces pour faire face aux dénis de services à moindre coût permettra de faciliter l'adoption des protocoles de BFT par les fournisseurs de services du *Cloud*.

La parallélisation orientée consensus proposée par Behl et al. permit l'obtention d'un débit jusqu'alors inédit. ER-COP parvient à conserver l'indépendance des *pillars* tout en assurant l'intégration d'une surcouche dédiée à la robustesse. Cette surcouche pourrait à l'avenir se voir enrichie par des mécanismes de robustesse supplémentaires, permettant ainsi de tolérer un plus large panel de dénis de services.

Dans le cadre plus général de la tolérance aux fautes byzantines, raffiner les modèles de communication ne permet plus d'afficher de véritables progrès en termes de performance. Lorsque associées au consensus originel proposé par Castro et al., la parallélisation et la virtualisation semblent désormais à elles seules pourvoir au besoin d'*efficacité*. Il convient désormais d'adapter les solutions dédiées à la *robustesse* à ces nouvelles architectures, afin de délivrer le plein potentiel de la tolérance aux fautes byzantines, et de permettre son adoption systématique.

9.3 Publications

Plusieurs de nos travaux furent publiés dans des conférences internationales.

1. Divya Gupta, Lucas Perronne, Sara Bouchenak. BFT-Bench : Framework to Evaluate Robustness and Effectiveness of BFT Protocols in Practice, 7th ACM/SPEC International Conference on Performance Engineering, Delft, The Netherlands, March 12-18, 2016.
2. Divya Gupta, Lucas Perronne, Sara Bouchenak. BFT-Bench : Framework to Evaluate Robustness and Effectiveness of BFT Protocols in Practice, 6th ACM Symposium on Cloud Computing, Hawai'i, USA, August 27-29, 2015. Poster.
3. Divya Gupta, Lucas Perronne, Sara Bouchenak. BFT-Bench : Towards a Practical Evaluation of Robustness and Effectiveness of BFT Protocols, 16th IFIP International Conference on Distributed Applications and Interoperable Systems, Heraklion, Crete, June 6-9, 2016.
4. Lucas Perronne, Sara Bouchenak. Towards Efficient and Robust BFT Protocols, 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Toulouse, France, June/July 28-1,2016.
5. Lucas Perronne, Sara Bouchenak. Towards Efficient and Robust BFT Protocols, 18th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2016), Lyon, France, November 7-10,2016.

Ces travaux furent soutenus par AMADEOS (Architecture for Multi-criticality Agile Dependable Evolutionary Open System-of-Systems), un projet collaboratif financé par la commission européenne FP7 (FP7-ICT-2013-610535), l'université Grenoble Alpe étant partenaire. Toutes nos expériences furent effectuées sur le cluster Grid'5000, mis à disposition par INRIA ALADDIN, avec le support du CNRS, RENATER, plusieurs universités ainsi que d'autres organismes de financement.

Liste des Figures

Liste des Tableaux

Bibliographie

[1] Cloud Computing. https://en.wikipedia.org/wiki/Cloud_computing.

[2] Cloud outage collection. http://ventures.tpedersen.net/errata/cloudstatus/cloud-outage-collection.

[3] e-fiscal project state of the art repository.

[4] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable byzantine fault-tolerant services. In *SOSP*, pages 59–74, 2005.

[5] M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. *Distributed computing*, 13(2) :99–125, 2000.

[6] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, pages 562–570. IEEE Computer Society Press, 1976.

[7] Y. Amir, B. A. Coan, J. Kirsch, and J. Lane. Byzantine replication under attack. In *DSN*, pages 197–206, 2008.

[8] L. Arantes, R. Friedman, O. Marin, and P. Sens. Probabilistic byzantine tolerance for cloud computing. In *Reliable Distributed Systems (SRDS), 2015 IEEE 34th Symposium on*, pages 1–10. IEEE, 2015.

[9] P.-L. Aublin, R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić. The next 700 bft protocols. *ACM Transactions on Computer Systems (TOCS)*, 32(4) :12, 2015.

[10] P.-L. Aublin, S. B. Mokhtar, and V. Quéma. Rbft : Redundant byzantine fault tolerance. In *ICDCS*, pages 297–306, 2013.

[11] A. Avizienis. The n-version approach to fault-tolerant software. *IEEE Transactions on software engineering*, 11(12) :1491, 1985.

[12] A. Avizienis and L. Chen. On the implementation of n-version programming for software fault tolerance during execution. In *Proc. IEEE COMPSAC*, volume 77, pages 149–155, 1977.

[13] A. Avizienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1) :11–33, 2004.

[14] J.-P. Bahsoun, R. Guerraoui, and A. Shoker. Making bft protocols adaptive.

[15] M. Balduzzi, J. Zaddach, D. Balzarotti, E. Kirda, and S. Loureiro. A security analysis of amazon's elastic compute cloud service. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 1427–1434. ACM, 2012.

[16] J. Behl, T. Distler, and R. Kapitza. Consensus-oriented parallelization : how to earn your first million. In *Proceedings of the 16th Annual Middleware Conference*, pages 173–184. ACM, 2015.

[17] D. Bernstein, E. Ludvigson, K. Sankar, S. Diamond, and M. Morrow. Blueprint for the intercloud - protocols and formats for cloud computing interoperability. In *Internet and Web Applications and Services, 2009. ICIW '09. Fourth International Conference on*, pages 328–336, May 2009.

[18] A. Bessani, J. Sousa, and E. E. Alchieri. State machine replication for the masses with bft-smart. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 355–362. IEEE, 2014.

[19] R. Boichat, P. Dutta, S. Frølund, and R. Guerraoui. Deconstructing paxos. *ACM Sigact News*, 34(1) :47–67, 2003.

[20] F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jégou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, et al. Grid'5000 : A large scale and highly reconfigurable grid experimental testbed. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, pages 99–106. IEEE Computer Society, 2005.

[21] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *OSDI*, pages 173–186, 1999.

[22] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4) :398–461, 2002.

[23] B.-G. Chun, P. Maniatis, and S. Shenker. Diverse replication for single-machine byzantine-fault tolerance. In *USENIX Annual Technical Conference*, pages 287–292, 2008.

[24] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. Upright cluster services. In *SOSP*, pages 277–290, 2009.

[25] A. Clement, M. Marchetti, E. Wong, L. Alvisi, and M. Dahlin. Bft : The time is now. In *Proceedings of the 2Nd Workshop on Large-Scale Distributed Systems and Middleware*, LADIS '08, pages 13 :1–13 :4, New York, NY, USA, 2008. ACM.

[26] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *NSDI*, pages 153–168, 2009.

[27] M. Correia, N. F. Neves, and P. Veríssimo. Bft-to : Intrusion tolerance with less replicas. *Comput. J.*, 56(6) :693–715, 2013.

[28] J. A. Cowling, D. S. Myers, B. Liskov, R. Rodrigues, and L. Shrira. Hq replication : A hybrid quorum protocol for byzantine fault tolerance. In *OSDI*, pages 177–190, 2006.

[29] V. Cunsolo, S. Distefano, A. Puliafito, and M. Scarpa. Volunteer computing and desktop cloud : The cloud@home paradigm. In *Network Computing and Applications, 2009. NCA 2009. Eighth IEEE International Symposium on*, pages 134–139, July 2009.

[30] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms : Taxonomy and survey. *ACM Computing Surveys (CSUR)*, 36(4) :372–421, 2004.

[31] S. Duan, S. Peisert, and K. Levitt. hbft : speculative byzantine fault tolerance with minimum cost. *IEEE Transactions on Dependable and Secure Computing*, 2014.

[32] S. Duan, S. Peisert, and K. Levitt. hbft : Speculative byzantine fault tolerance with minimum cost. 2014.

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

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

[35] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in globally distributed storage systems. In *OSDI*, pages 61–74, 2010.

[36] J. N. Gray. Notes on data base operating systems. In *Operating Systems*, pages 393–481. Springer, 1978.

[37] R. Guerraoui, N. Knezevic, V. Quéma, and M. Vukolic. The next 700 bft protocols. In *EuroSys*, pages 363–376, 2010.

[38] R. Guerraoui, N. Knezevic, V. Quema, and M. Vukolic. Stretching bft. Technical report, Technical Report EPFL-REPORT-149105, EPFL, 2011.

[39] R. Guerraoui, R. R. Levy, B. Pochon, and V. Quéma. Throughput optimal total order broadcast for cluster environments. *ACM Transactions on Computer Systems (TOCS)*, 28(2) :5, 2010.

[40] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel. Cheapbft : Resource-efficient byzantine fault tolerance. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 295–308. ACM, 2012.

[41] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. The securering protocols for securing group communication. In *System Sciences, 1998., Proceedings of the Thirty-First Hawaii International Conference on*, volume 3, pages 317–326. IEEE, 1998.

[42] J. Kohlas, B. Meyer, and A. Schiper, editors. *Dependable Systems : Software, Computing, Networks, Research Results of the DICS Program*, volume 4028 of *Lecture Notes in Computer Science*. Springer, 2006.

[43] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. L. Wong. Zyzzyva : Speculative byzantine fault tolerance. *ACM Trans. Comput. Syst.*, 27(4), 2009.

[44] L. Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2 :95–114, 1978.

[45] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7) :558–565, 1978.

[46] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2) :133–169, 1998.

[47] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2) :133–169, 1998.

[48] L. Lamport. Paxos made simple. *SIGACT News*, 32(4) :51–58, 2001.

[49] L. Lamport. Lower bounds for asynchronous consensus. In *Future Directions in Distributed Computing*, pages 22–23, 2003.

[50] L. Lamport, R. E. Shostak, and M. C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3) :382–401, 1982.

[51] B. W. Lampson. Hints for computer system design. *IEEE Software*, 1(1) :11–28, 1984.

[52] J. Li and D. Maziéres. Beyond one-third faulty replicas in byzantine fault tolerant systems. In *NSDI*, 2007.

[53] Z. Li, M. Liang, L. O'brien, and H. Zhang. The cloud's cloudy moment : A systematic survey of public cloud service outage. *arXiv preprint arXiv :1312.6485*, 2013.

[54] P. J. Marandi, M. Primi, and F. Pedone. Multi-ring paxos. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–12. IEEE, 2012.

[55] J.-P. Martin and L. Alvisi. Fast byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3) :202–215, 2006.

[56] P. M. Mell and T. Grance. Sp 800-145. the nist definition of cloud computing. Technical report, Gaithersburg, MD, United States, 2011.

[57] Z. Milosevic, M. Biely, and A. Schiper. Bounded delay in byzantine-tolerant state machine replication. In *SRDS*, pages 61–70, 2013.

[58] B. M. Oki and B. H. Liskov. Viewstamped replication : A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 8–17, New York, NY, USA, 1988. ACM.

[59] M. K. Reiter. The rampart toolkit for building high-integrity services. In *Theory and Practice in Distributed Systems*, pages 99–110. Springer, 1995.

[60] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2) :120–126, 1978.

[61] M. Rouse. What is a multi-cloud strategy, 2014.

[62] D. C. Schmidt and T. Suda. Transport system architecture services for high-performance communications systems. *IEEE Journal on Selected Areas in Communications*, 11(4) :489–506, 1993.

[63] F. B. Schneider. Implementing fault-tolerant services using the state machine approach : A tutorial. *ACM Computing Surveys (CSUR)*, 22(4) :299–319, 1990.

[64] F. B. Schneider. Implementing fault-tolerant services using the state machine approach : A tutorial. *ACM Comput. Surv.*, 22(4) :299–319, Dec. 1990.

[65] M. Serafini, P. Bokor, D. Dobre, M. Majuntke, and N. Suri. Scrooge : Reducing the costs of fast byzantine replication in presence of unresponsive replicas. In *DSN*, pages 353–362, 2010.

[66] A. Shoker, J.-P. Bahsoun, and M. Yabandeh. Improving independence of failures in bft. In *Network Computing and Applications (NCA), 2013 12th IEEE International Symposium on*, pages 227–234, Aug 2013.

[67] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe. Bft protocols under fire. In *NSDI*, volume 8, pages 189–204, 2008.

[68] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis. Zeno : Eventually consistent byzantine-fault tolerance. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 169–184, Berkeley, CA, USA, 2009. USENIX Association.

[69] M. D. Skeen. *Crash recovery in a distributed database system*. University of California, Berkeley, 1982.

[70] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, pages 91–104, 2004.

[71] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung. Spin one's wheels ? byzantine fault tolerance with a spinning primary. In *SRDS*, pages 135–144, 2009.

[72] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Veríssimo. Efficient byzantine fault-tolerance. *IEEE Trans. Computers*, 62(1) :16–30, 2013.

[73] T. Wood, R. Singh, A. Venkataramani, P. Shenoy, and E. Cecchet. Zz and the art of practical bft execution. In *Proceedings of the sixth conference on Computer systems*, pages 123–138. ACM, 2011.

[74] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for byzantine fault tolerant services. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 253–267. ACM, 2003.