Vers des protocoles de tolérance aux fautes Byzantines efficaces et robustes

Pierre-Louis Aublin

▶ **To cite this version:**

Pierre-Louis Aublin. Vers des protocoles de tolérance aux fautes Byzantines efficaces et robustes. Performance et fiabilité [cs.PF]. Université de Grenoble, 2014. Français. NNT : 2014GRENM006 . tel-01549111

HAL Id: tel-01549111
https://theses.hal.science/tel-01549111

Submitted on 28 Jun 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Pierre-Louis Aublin

Thèse dirigée par **Vivien Quéma**
et co-encadrée par **Sonia Ben Mokhtar**

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

Vers des protocoles de tolérance aux fautes Byzantines efficaces et robustes

Thèse soutenue publiquement le **21 janvier 2014**,
devant le jury composé de :

Prof. Noël De Palma
Professeur à l'Université de Grenoble 1, Président
Prof. Xavier Défago
Associate Professor au Japan Advanced Institute of Science and Technology, Rapporteur
Dr Gilles Muller
Directeur de Recherche à Inria, Rapporteur
Prof. Françoise Baude
Professeur à l'Université de Nice Sophia-Antipolis, Examinatrice
Prof. Pierre Sens
Professeur à l'Université Pierre et Marie Curie - Paris VI, Examinateur
Prof. Vivien Quéma
Professeur à Grenoble INP, Directeur de thèse
Dr Sonia Ben Mokhtar
Chargée de Recherche au CNRS, Co-Encadrante de thèse

Résumé

Les systèmes d'information deviennent de plus en plus complexes et il est difficile de les garantir exempts de fautes. La réplication de machines à états est une technique permettant de tolérer les fautes, quelque soit leur nature, qu'elles soient logicielles ou matérielles. Cette thèse traite des protocoles de réplication de machines à états tolérant les fautes arbitraires, également appelées Byzantines. Ces protocoles doivent relever deux défis : (i) ils doivent être efficaces, c'est-à-dire que leurs performances doivent être les meilleurs possibles, afin de masquer le coût supplémentaire dû à la réplication et (ii) ils doivent être robustes, c'est-à-dire qu'une attaque ne doit pas faire baisser leurs performances de manière importante.

Dans cette thèse nous observons qu'aucun protocole ne relève ces deux défis en même temps : les protocoles que nous connaissons aujourd'hui sont soit conçus pour être efficaces au détriment de leur robustesse, soit conçus pour être robustes au détriment de leurs performances. Une première contribution de cette thèse est la conception d'un nouveau protocole qui réunit le meilleur des deux mondes. Ce protocole, R-Aliph, combine un protocole efficace mais peu robuste avec un protocole robuste afin de fournir un protocole à la fois efficace et robuste. Nous évaluons ce protocole de manière expérimentale et montrons que ses performances en cas d'attaque sont égales aux performances du protocole robuste sous-jacent. De plus, ses performances dans le cas sans faute sont très proches des performances du protocole connu le plus efficace : la différence maximale de débit est inférieure à 6%.

Dans la seconde partie de cette thèse nous observons que les protocoles conçus pour être robustes sont peu robustes en réalité. En effet, il est possible de concevoir une attaque dans laquelle leur perte de débit est supérieure à 78%. Nous identifions le problème de ces protocoles et nous concevons un nouveau protocole plus robuste que les précédents : RBFT. L'idée de base de ce protocole est d'exécuter en parallèle plusieurs instances d'un même protocole. Les performances de ces différentes instances sont surveillées de près afin de détecter tout comportement malicieux. Nous évaluons RBFT dans le cas sans faute et en cas d'attaque. Nous montrons que ses performances dans le cas sans faute sont comparables aux performances des protocoles considérés comme robustes. De plus, nous observons que la dégradation maximale de performance qu'un attaquant peut causer sur le système est inférieure à 3%, même dans le cas d'attaques où les clients malicieux et les réplicas malicieux complotent.

Mots-clés. Systèmes distribués, tolérance aux fautes Byzantines, performance, robustesse.

Abstract

Information systems become more and more complex and it is difficult to guarantee that they are bug-free. State Machine Replication is a technique for tolerating faults, regardless their nature, whether they are software or hardware faults. This thesis studies Fault Tolerant State Machine Replication protocols that tolerate arbitrary, also called Byzantine, faults. These protocols face two challenges : (i) they must be efficient, i.e., their performance have to be the best ones, in order to mask the cost of the replication and (ii) they must be robust, i.e., an attack should not cause an important performance degradation.

In this thesis, we observe that no protocol addresses both of these challenges : current protocols are either designed to be efficient but fail to be robust, or designed to be robust but exhibit poor performance. A first contribution of this thesis is the design of a new protocol which achieves the best of both worlds. This protocol, R-Aliph, combines an efficient but not robust protocol with a protocol designed to be robust. The result is a protocol that is both robust and efficient. We evaluate this protocol experimentally and show that its performance under attack equals the performance of the underlying robust protocol. Moreover, its performance in the fault-free case is close to the performance of the best known efficient protocol : the maximal throughput difference is less than 6%.

In the second part of this thesis we analyze the state-of-the-art robust protocols and demonstrate that they are not effectively robust. Indeed, one can run an attack on each of these protocols such that the throughput loss is at least equal to 78%. We identify the problem of these protocols and design a new, effectively robust, protocol called RBFT. The main idea of this protocol is to execute several instances of a robust protocol in parallel and closely monitor their performance, in order to detect a malicious behaviour. We evaluate RBFT in the fault-free case and under attack. We observe that its performance in the fault-free case is equivalent to the performance of the other so-called robust BFT protocols. Moreover, we show that the maximal throughput degradation, under attacks where malicious clients and malicious replicas collude, is less than 3%.

Keywords. Distributed systems, Byzantine Fault Tolerance, performance, robustness.

Remerciements

Je tiens tout d'abord à remercier Noël De Palma, Professeur à l'Université de Grenoble 1, de m'avoir fait l'honneur de présider ce jury.

Je remercie également Xavier Défago, Associate Professor au Japan Advanced Institute of Science and Technology, et Gilles Muller, Directeur de recherche à Inria, d'avoir accepté d'être rapporteurs de cette thèse. Merci également à Françoise Baude, Professeur à l'Université de Nice Sophia-Antipolis, et Pierre Sens, Professeur à l'Université Pierre et Marie Curie - Paris VI, d'avoir accepté d'être examinateurs de ce travail.

Je souhaite également remercier mes encadrants de thèse sans qui rien de cela n'aurait été possible. En particulier, je remercie Vivien Quéma, Professeur à Grenoble INP, pour m'avoir encadré durant mon stage de magistère, mon stage de master puis ma thèse. J'ai pu apprendre énormément de choses durant toutes ces années grâce à ton encadrement. Je tiens également à remercier Sonia Ben Mokhtar, Chargée de Recherche au CNRS, pour m'avoir co-encadré depuis Lyon et pour toute l'énergie dépensée tout au long de ma thèse, notamment lors des deadlines.

Je remercie également les personnes suivantes sans qui la Recherche n'aurait pas eu les mêmes saveurs :

- Renaud, grâce à qui j'ai pu effectuer un stage au sein de l'équipe ;
- Didier et Nicolas, pour l'énergie investie dans les cours de Polytech ;
- mes "frères de thèse", Baptiste et Gautier, pour tous les moment que nous avons pu partager au cours de ces trois dernières années ;
- les anciens membres de l'équipe : Alessio, Bertrand, Fabien, Fabien, Sylvain et Willy ;
- les doctorants qui n'ont pas encore fini : Ahmed, Amadou, Jérémie et Soguy. Je continuerai à prendre soin du Gingko ;
- tous les membres des équipes Sardes, ERODS et DRIM.

Je tiens également à remercier Decathlon pour sa gamme de vêtements Quechua et pour tous ses magasins autour de Grenoble. Il aurait été difficile de venir au labo en vélo ou d'aller respirer dans les montagnes par tous temps sans eux.

Enfin, je remercie ma famille et mes amis pour leur support tout au long de mon parcours. Je tiens tout particulièrement à remercier Cécile, Cyril et la Bande à LauLo. Le nombre d'atomes dans l'univers n'est pas suffisant pour vous exprimer ma gratitude.

Préface

Ce manuscrit présente la recherche conduite dans les équipes Sardes (INRIA Grenoble – Rhône-Alpes / Laboratoire d'Informatique de Grenoble) puis ERODS (Laboratoire d'Informatique de Grenoble) lors de ma thèse de doctorat qui s'est déroulée dans la spécialité "Informatique" de l'école doctorale "Mathématiques, Sciences et Technologies de l'Information, Informatique", au sein de l'Université de Grenoble. Ces recherches ont été menées sous la direction de Vivien Quéma (Grenoble INP/ERODS) et Sonia Ben Mokhtar (CNRS/DRIM).

Cette thèse se concentre sur la création de protocoles de réplication de machines à états tolérant les fautes arbitraires (aussi appelées *Byzantines*) efficaces et robustes. Ces protocoles exécutent un service en parallèle sur plusieurs machines distinctes et coordonnent ces copies. Ces protocoles doivent être robustes, c'est-à-dire qu'une attaque ne doit que faiblement impacter leurs performances. De plus, la coordination des copies ainsi que les mécanismes utilisés par ces protocoles sont coûteux, c'est pourquoi il est important que ces protocoles soient le plus efficace possible.

Cette thèse a donné lieu à un article de journal en cours d'examination et à un article accepté dans une conférence, tous les deux de rang international :

1 P.-L. Aublin, R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The next 700 BFT protocols", ACM Transactions on Computer Systems, en cours d'examination.

2 P.-L. Aublin, S. Ben Mokhtar, and V. Quéma, "RBFT : Redundant Byzantine Fault Tolerance," in Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems, IEEE Computer Society.

Terminologie

Ce manuscrit fait référence à plusieurs concepts. Pour chacun de ces concepts, nous précisons le terme correspondant dans la Table I. Nous fournissons également le ou les termes équivalents en Anglais.

Concept	**Terme utilisé**	**Équivalent(s) Anglais**
Instance d'un programme en cours d'exécution	processus	process
Machine physique	nœud	node
Copie du système	réplica	replica
Réplica qui propose un ordre des requêtes	principal	leader, primary
Le système se comporte de manière correcte	propriété de sûreté	safety property
Le système progresse à terme	propriété de vivacité	liveness property
Réplication de machines à états	RME	State Machine Replication (SMR)
Tolérance des comportements malicieux/arbitraires	tolérance aux fautes Byzantines (TFB)	Byzantine Fault Tolerance (BFT)
Code d'authentification de message	MAC	Message Authentication Code (MAC)
Valeur retournée par une fonction de hachage	empreinte	digest
point de contrôle	point de contrôle	checkpoint
mécanisme de récupération	mécanisme de récupération	recovery mechanism
lot de requêtes	lot de requêtes	batch
basculement	basculement	switching
avortement	avortement	abort

TABLE I – Terminologie utilisée dans ce document.

Table des matières

Introduction

Contexte et problématique

Les systèmes d'information distribués sont aujourd'hui omniprésents, que ce soient les infrastructures bancaires, les réseaux de distribution d'énergie, l'informatique dans le nuage, les réseaux sociaux, etc. Ces systèmes deviennent de plus en plus complexes. Ils mettent en jeu un grand nombre d'entités hétérogènes qui doivent coopérer entre elles dans des environnements potentiellement peu fiables et difficiles à contrôler. Toutefois, notre dépendance envers ces systèmes demande à ce qu'ils soient hautement disponibles, malgré la présence de fautes.

De multiples études ont montré que les fautes logicielles et les erreurs dans les systèmes d'information sont nombreuses [1, 2, 3, 4, 5, 6]. Par exemple, Li et al. ont récemment présenté une étude d'environ vingt-neuf mille bugs dans les logiciels de Mozilla et le serveur web Apache [5]. Dans cette étude, il a été découvert que 10% des bugs d'Apache conduisent à un arrêt du système et que les bugs relatifs à la sécurité des applications sont de plus en plus nombreux au fil des années. Des outils conçus pour améliorer la qualité du code existent. Par exemple, Dimmunix [7] est un outil conçu pour être résistant face aux inter-blocages ; la fiabilité des pilotes peut être améliorée avec Nooks [8] ; certaines erreurs de programmation peuvent être trouvées statiquement [9, 10] ; etc. Toutefois, aucun de ces outils ne peut certifier qu'un système sera exempt de bugs. Pire encore, certaines erreurs ne sont pas dues à la qualité du code mais proviennent de l'environnement physique, telles que les inversions de bits dûs au rayonnement cosmique [11] ou la section d'un câble réseau [12]. De plus il est important de se protéger contre les attaques, intrusions et comportements malicieux. Par exemple un ver informatique a désactivé le système de surveillance d'une centrale nucléaire dans l'Ohio en 2003. Par conséquent devoir tolérer les fautes logicielles et matérielles est aujourd'hui nécessaire.

La réplication est une manière d'améliorer la fiabilité des systèmes. La réplication consiste en l'utilisation de plusieurs copies du système afin de permettre de masquer les fautes, de quelque nature que ce soit (p. ex. un arrêt brutal d'une machine ou un bug logiciel). De cette façon le système continue d'être disponible malgré la présence de fautes. Dans le contexte des protocoles de réplication, le système répliqué doit répondre aux contraintes suivantes. Tout d'abord, son interface doit être telle qu'il reçoit en entrée des requêtes de la part de clients et retourne en sortie des réponses. Ensuite, il doit être implanté sous forme de machine à états, ce qui permet de simplifier son étude et sa

conception. Enfin, il doit être déterministe, c.à.d. qu'il retournera toujours la même sortie pour une même entrée. Cela permet de garantir que les différentes copies du système répondront de la même manière à une même requête.

La réplication peut être implantée de différentes manières. En particulier, elle peut être *active*, auquel cas tous les réplicas exécutent les requêtes, ou *passive*, auquel cas un seul réplica exécute les requêtes et transmet les modifications aux autres réplicas.

Dans cette thèse, nous considérons le modèle de fautes le plus général possible : les réplicas peuvent s'arrêter brutalement tout comme être contrôlés par un attaquant et présenter un comportement différent de leur spécification. Nous parlons alors de fautes arbitraires, ou *Byzantines*. Comme un seul réplica exécute les requêtes dans le cas de la réplication passive, nous ne pouvons utiliser cette technique. En effet, ce réplica pourrait ne pas exécuter les requêtes, transmettre de faux résultats aux autres réplicas ou transmettre des réponses incorrectes aux clients sans être détecté. C'est pourquoi nous nous intéressons à la réplication active uniquement.

En particulier, nous nous intéressons aux algorithmes distribués (appelés *protocoles*) de réplication de machines à états tolérant les fautes Byzantines (que nous abrégeons en *protocoles de TFB* par souci de clarté). Le but premier de ces protocoles et de garantir que l'état interne des différentes copies ne divergera pas. Par exemple, dans le cas d'un compte bancaire répliqué, il serait dommageable que les différents réplicas ne possèdent pas la même balance. Étant donné la nature peu fiable des liens de communication et des machines physiques, ils doivent fournir differentes propriétés : (i) des propriétés de *sûreté*, qui stipulent que le système se comporte de manière correcte, en accord avec sa spécification et (ii) des propriétés de *vivacité*, qui stipulent qu'à terme, l'exécution du système progresse.

Ces protocoles posent de nombreux défis. Tout d'abord, la réplication induit un surcoût non négligeable par rapport à un système non répliqué. Ce surcoût provient des multiples échanges de messages entre les copies afin de garantir le même ordre d'exécution sur toutes les copies et la cohérence de leurs états. De ce fait, les protocoles doivent être efficaces afin de ne pas impacter les performances des systèmes répliqués. De plus, ils doivent être robustes, c'est-à-dire qu'ils doivent fournir des performances acceptables en cas d'attaque. En effet, un protocole tolérant les fautes dont le débit chute à zéro en cas d'attaque ne présente pas d'intérêt. Enfin, les propriétés de sûreté et de vivacité ne garantissent en rien les performances du système en cas d'attaque. Comme nous le détaillons dans la suite de ce manuscrit, les protocoles conçus pour offrir les meilleures performances possibles, proches des performances d'un système non répliqué, peuvent fournir un débit nul en cas d'attaque.

Objectifs et contributions de cette thèse

Cette thèse se place dans le domaine des protocoles de réplication de machines à états tolérant les fautes Byzantines. Nous présentons les protocoles existants et observons qu'ils peuvent être classés en deux catégories : (i) les protocoles *patrimoniaux*, qui adressent le problème de l'efficacité en cherchant à fournir de très bonnes performances dans le cas sans faute et (ii) les protocoles *conçus pour être robustes*, qui cherchent à

limiter la perte de performance en cas d'attaque. Toutefois, il n'existe à ce jour pas de protocole conçu pour être à la fois efficace et robuste.

Une première contribution de cette thèse est un nouveau protocole, R-Aliph, qui combine le meilleur des deux mondes en étant à la fois efficace et robuste. Ce protocole combine un protocole efficace avec un protocole conçu pour être robuste. Ce protocole fournit des performances égales au plus rapide des protocoles patrimoniaux dans le cas sans faute, et des performances au moins égale au protocole conçu pour être robuste en cas d'attaque.

Une seconde contribution de cette thèse concerne la robustesse des protocoles existants. Nous montrons qu'il est possible de réduire les performances de tous les protocoles tolérant les fautes Byzantines, même ceux conçus pour être robustes. Nous proposons un nouveau protocole, RBFT, plus robuste que les protocoles existants. Le concept de base de ce protocole est l'exécution en parallèle de plusieurs instances d'un même protocole. Les performances de ces instances sont surveillées en permanence afin de détecter tout comportement malicieux. Nous montrons que la perte de performance de RBFT en cas d'attaque est très faible, tandis que ses performances dans le cas sans faute sont équivalentes aux performances des protocoles existants conçus pour être robustes.

Nous avons conduit une évaluation expérimentale des principaux protocoles de tolérance aux fautes Byzantines présentés dans cette thèse sur un cluster de dix machines : Aliph, Prime, Aardvark, Spinning, R-Aliph et RBFT. Également, en nous basant sur les travaux d'Aardvark [13] et de PBFT [14], nous fournissons en annexe une étude plus formelle de Prime, Aardvark, Spinning et RBFT.

Organisation du document

La suite de ce manuscrit est organisée en différents chapitres, dont voici un bref résumé.

Chapitre 1 - Réplication de machines à états tolérant les fautes Byzantines (TFB) Ce chapitre introduit les fautes Byzantines et la réplication de machines à états. Il décrit également le modèle du système et la méthodologie utilisée lors de l'évaluation des protocoles présentés dans ce manuscrit.

Chapitre 2 - État de l'art des protocoles de TFB Ce chapitre présente l'état de l'art des protocoles de tolérance aux fautes Byzantines. Nous y présentons deux grandes familles de protocoles : les protocoles patrimoniaux, qui fournissent de bonnes performances dans le cas sans faute, et les protocoles dits robustes, conçus pour fournir de bonnes performances en cas d'attaque.

Chapitre 3 - R-Aliph : un protocole de TFB conçu pour être efficace et robuste Ce chapitre présente un nouveau protocole, R-Aliph, qui est efficace dans le cas sans faute et en cas d'attaque, tout en étant au moins aussi robustes que les protocoles dits robustes en cas d'attaque. R-Aliph combine deux protocoles : un efficace mais non robuste, avec un robuste mais moins efficace, afin de construire un nouveau protocole à la fois efficace et robuste.

Chapitre 4 - RBFT : un nouveau protocole de TFB plus robuste que les précédents Ce chapitre montre que les protocoles existants dits robustes ne le sont pas en

réalité. Pour chacun d'eux il est possible de trouver une attaque qui fait baisser leur performance de manière importante. Ce chapitre présente un nouveau protocole, RBFT, qui adresse ce problème. En particulier, RBFT est plus robuste que les protocoles existants tout en fournissant des performances comparables dans le cas sans faute.

Enfin, nous dressons un bilan général des travaux présentés, et nous donnons un ensemble de perspectives et d'évolutions possibles à ces travaux.

1

Réplication de machines à états tolérant les fautes Byzantines (TFB)

Sommaire

Nous introduisons dans ce chapitre les concepts de base de la réplication de machines à états tolérant les fautes Byzantines. Ces concepts sont utilisés par les protocoles (également appelés *algorithmes distribués*) de réplication de machines à états dont l'objectif est de construire un système tolérant les fautes. Ce manuscrit se concentre sur la conception de tels protocoles. Nous définissons dans un premier temps différentes abstractions utilisées pour modéliser un système distribué (Section 1.1). Puis nous présentons la réplication de machines à états (Section 1.2). Enfin, nous présentons le modèle du système que nous considérons tout au long de ce manuscrit (Section 1.3).

1.1 Abstractions

La réplication de machines à états tolérant les fautes Byzantines s'inscrit dans le domaine des systèmes distribués. En effet, cette approche consiste en la coopération

de différentes entités connectées entre elles via un réseau de communications. Nous présentons donc dans cette section différentes abstractions et définitions qui permettent de modéliser un système distribué et de raisonner sur la réplication de machines à états.

1.1.1 Définitions

Dans un système distribué, une collection d'entités, appelées *processus*, communiquent entre eux afin de former une application que l'on nomme *service*. Par exemple, dans le domaine du web, un client et un serveur (les *processus*) communiquent entre eux pour accéder à des pages web (le *service*). La communication dans un système distribué se fait via des canaux de communication uniquement : les processus ne partagent pas de mémoire. Il est bien sûr possible de construire un système distribué ayant une notion de mémoire partagée (tel qu'un registre atomique [15]), mais cela ne peut se faire qu'au dessus des canaux de communication.

La définition de *processus* est suffisamment large pour pouvoir observer, analyser et représenter le service à de multiples niveaux. Plus précisément, un processus peut aussi bien désigner une fonction dans un programme en cours d'exécution qu'une machine physique sur laquelle s'exécutent de nombreux programmes. Dans le contexte de ce manuscrit, nous considérons généralement la totalité de la machine physique et parlons alors de *nœud*.

Leslie Lamport a écrit qu'"un système distribué est un système dans lequel la défaillance d'un ordinateur dont vous ignoriez l'existence peut rendre votre propre ordinateur inutilisable". Cette définition illustre un problème central des systèmes distribués : la gestion des défaillances. Un processus est *défaillant* lorsque son comportement n'est pas conforme à sa spécification. La déviation du processus de sa spécification est une *erreur*, dont la cause est une *faute* [16].

1.1.2 Catégories de fautes

Comme détaillé par Avizienis et al [16], les fautes peuvent être classées en différentes catégories, organisées de manière hiérarchique.

Arrêt inopiné Un arrêt inopiné a pour conséquence l'arrêt du processus en cours d'exécution. Cela se produit par exemple suite à une panne de courant. Dans un tel cas le processus ne peut ni répondre aux messages qui lui sont destinés ni en envoyer.

Omission Un processus commet une faute d'omission lorsqu'il échoue lors de la réception ou de l'émission d'un message. Par exemple, un processus qui s'exécute sur une machine dont la carte réseau présente des dysfonctionnements peut commettre des fautes par omission.

Validation Un processus commet une faute de validation lorsqu'il exécute une opération inutile ou incorrecte. Par exemple, le programme d'une banque qui exécute deux fois la même transaction commet une faute de validation.

Arbitraire / Byzantine Un processus commet une faute arbitraire lorsqu'il dévie de sa spécification de manière arbitraire. Cela peut être une faute d'omission, validation, parce que le processus a été compromis par un attaquant, etc. Cette catégorie de faute est la plus générale et regroupe les catégories présentées ci-dessus.

Historiquement, les fautes arbitraires sont appelées *fautes Byzantines*, dû au travail séminal de Lamport et al. intitulé *The Byzantine generals problem* [17].

Dans la suite de ce manuscrit, nous désignons par *en panne* un processus qui s'arrête de manière inopinée et par *malicieux* un processus qui montre un comportement Byzantin. À l'inverse, un processus qui s'exécute correctement est dit *correct*.

1.1.3 Cryptographie

La cryptographie peut être utilisée dans un environnement peu fiable pour garantir l'authenticité et la protection des messages. En effet, une erreur dans le système, que ce soit au niveau logiciel ou matériel, peut corrompre les échanges. Un processus malicieux peut également tenter d'attaquer le système en modifiant des messages ou en se faisant passer pour quelqu'un d'autre. Nous présentons dans cette section trois mécanismes de cryptographie utilisés par les protocoles de réplication de machines à états.

1.1.3.1 Fonctions de hachage

Le mécanisme le plus simple est l'utilisation de fonctions de hachage, tels que MD5 ou SHA-2. Ces fonctions associent une empreinte à un ensemble de données en entrée. La taille de l'empreinte dépend de la fonction de hachage utilisée. Elle est par exemple de 128 bits pour MD5 et jusqu'à 512 bits pour SHA-2. Idéalement, l'association entre l'entrée et l'empreinte doit être unique, c.à.d. qu'il n'existe pas deux entrées différentes qui ont la même empreinte.

L'empreinte est utilisée afin de vérifier l'intégrité des données. Plus précisément, lorsqu'un processus souhaite envoyer un message m à un autre processus, il calcule son empreinte δ et envoie les deux informations. Lorsque le processus destinataire reçoit le message, il calcule son empreinte d' et la compare avec d. Si $d = d'$, alors le message n'a pas été corrompu lors de son transit.

Une fonction de hachage permet de vérifier l'intégrité d'un message, mais ne permet pas de le réparer. Lorsqu'un message corrompu est reçu, le processus peut tenter de le réparer en utilisant des codes correcteurs d'erreurs ou demander une retransmission du message. De plus, une empreinte ne garantit pas l'authenticité du message : un processus malicieux peut envoyer des messages en se faisant passer pour un autre sans être détecté. Cependant, calculer une fonction de hachage est peu coûteux et très rapide.

1.1.3.2 Code d'authentification de message

Un code d'authentification de message (*Message Authentication Code* ; noté MAC) est un champ de bits qui permet d'authentifier un message. CBC-MAC et HMAC sont des exemples de fonctions de calculs de MAC. La taille d'un MAC est relativement faible (au plus quelques centaines de bits). Le calcul d'un MAC sur un message se fait via une fonction de cryptographie par clé secrète, telle qu'AES. Pour cela, les différents processus doivent auparavant s'échanger leurs clés secrètes en toute confidentialité.

Un MAC est relativement peu coûteux et rapide à calculer. Lorsqu'un processus doit valider un message possédant un MAC, il vérifie que le MAC déchiffré (en utilisant la clé secrète) correspond bien au message reçu. À cause de l'utilisation de clés secrètes, il faut calculer autant de MACs que de destinataires pour envoyer un message à plusieurs

processus (un MAC par destinataire). Lorsqu'un message possède plusieurs MACs, nous parlons d'*authentificateur MAC*.

Un MAC authentifie un message entre deux processus p_1 et p_2. Un processus malicieux, p_3, ne peut se faire passer pour p_1 ou p_2, sauf s'il connaît la clé secrète. Toutefois, un MAC ne garantit pas la propriété de non-répudiation : un message envoyé à plusieurs destinataires et contenant un authentificateur MAC ne garantit pas que le message sera correctement vérifié par tous les destinataires. En effet, un sous-ensemble des MACs peut être invalide, ce qui amènera certains processus à accepter le message et d'autres processus à le rejeter.

1.1.3.3 Signature

Les signatures fournissent un moyen puissant d'authentifier un message. Les signatures se basent sur un algorithme de cryptographie à clé publique, tel que RSA. Les signatures sont un ordre de grandeur plus coûteuses que les MACs ou les empreintes, et sont généralement calculées sur une empreinte des données à signer (la taille de la signature est alors égale à la taille de l'empreinte).

Chaque processus possède deux clés : une clé publique et une clé privée. La clé privée n'est connue que de ce processus, tandis que la clé publique peut être divulguée aux autres processus. Lorsqu'un processus souhaite envoyer un message, alors il le signe en utilisant sa clé privée. Lorsqu'un processus reçoit un message signé, alors il vérifie sa signature en utilisant la clé publique de l'émetteur. Plus précisément, il déchiffre la signature d en utilisant la clé publique, puis vérifie que le message correspond bien à la signature déchiffrée.

Contrairement aux MACs, les signatures garantissent la propriété de non-répudiation : si un processus accepte un message donné, alors tous les processus qui recevront le même message l'accepteront également (tant que le message ou la signature n'a pas été corrompu).

1.1.4 Temps

Il est important dans un système distribué de pouvoir raisonner sur le passage du temps et la causalité entre les évènements. Cela est à la fois valable pour les échanges de messages sur les canaux de communication ainsi que pour le temps de calcul des différents processus. Le système peut être modélisé de trois manières différentes, que nous détaillons ci-dessous.

1.1.4.1 Système asynchrone

L'abstraction la plus simple est un système asynchrone. Dans un tel système, il n'existe aucune hypothèse temporelle sur les processus et les canaux de communication. Plus précisément, il n'existe pas de délai borné connu sur les temps de calcul ainsi que la transmission des messages. De plus, les processus peuvent ne pas avoir accès à une horloge, et donc peuvent n'avoir aucune notion du temps qui s'écoule. Malgré cela, les processus peuvent avoir une notion de causalité entre les évènements par le biais d'horloges logiques [18]. Plus précisément, chaque processus p maintient un compteur l_p appelé *horloge logique*, initialisé à 0. Chaque fois qu'un nouvel évènement

se produit chez le processus p, l'horloge logique est incrémentée de 1. Lorsque p envoie un message, il envoie également la valeur de l_p. Lorsqu'un processus q reçoit un message provenant de p avec l'horloge logique l_p, alors il met à jour sa propre horloge logique l_q de la façon suivante : $l_q = max(l_p, l_q) + 1$. Les horloges logiques peuvent également être utilisées afin d'avoir une notion de passage du temps : la valeur de l'horloge logique d'un processus fournit le nombre d'évènements qui se sont produits depuis le démarrage du système.

Cette abstraction peut être utilisée pour décrire un système dans lequel les différents processus n'ont pas la même puissance de calcul ou les canaux de communication ne sont pas fiables (p. ex. s'ils peuvent perdre des messages).

1.1.4.2 Système synchrone

À l'opposé se trouve le système synchrone. Dans un tel système, il existe une hypothèse temporelle forte sur les processus et canaux de communication. Plus précisément, il existe des délais bornés connus sur les temps de calcul et la transmission des messages. De plus, les processus ont accès à une horloge et connaissent le rythme auquel les horloges des différents processus dévient d'une horloge commune (p. ex. une horloge atomique). Les processus peuvent ainsi raisonner sur le passage du temps, que ce soit sur leur propre exécution ou sur celle d'autres processus, ainsi qu'avoir une notion de causalité entre les évènements : un évènement e_1 se produit avant un autre évènement e_2 si et seulement si l'heure à laquelle e_1 s'est produit précède l'heure à laquelle e_2 s'est produit.

En utilisant cette abstraction les processus peuvent détecter des erreurs (telle qu'un processus en panne qui n'émet plus aucun message) avec une grande précision. Les systèmes synchrones sont souvent utilisés dans les systèmes embarqués critiques. Toutefois, ils ne permettent pas de modéliser correctement des systèmes faiblement couplés tels qu'Internet.

1.1.4.3 Système ultimement synchrone

Une troisième abstraction est le système *ultimement synchrone* [19]. Tout comme dans un système synchrone, il existe un délai borné sur les temps de calcul et la transmission des messages. Toutefois, ces bornes ne sont pas connues. Une autre manière de définir un tel système est qu'un système ultimement synchrone est asynchrone avec des périodes synchrones qui se répètent à l'infini. Un système ultimement synchrone n'étant pas toujours synchrone, le passage du temps et la causalité entre les évènements peuvent être observés de la même façon que dans un système asynchrone, en utilisant des horloges logiques.

Cette abstraction correspond aux systèmes distribués que l'on trouve sur Internet, qui s'exécutent de manière synchrone la plupart du temps mais qui peuvent parfois rencontrer des délais de communication importants.

1.1.5 Réseau

Comme nous l'avons expliqué auparavant, les différents processus d'un système distribué communiquent via des canaux de communication uniquement. Ces canaux de

communication sont généralement représentés par des liens réseaux. Nous supposons que les canaux de communication sont point-à-point, c.à.d. que la communication est possible entre chaque paire de processus.

Chaque processus a accès à trois primitives de communication réseau : `envoyer(`m`,` p`)`, qui permet d'envoyer un message m au processus p ; `recevoir(`m`,` p`)`, qui permet de recevoir un message m provenant du processus p ; `livrer(`m`)`, qui permet au processus ayant reçu le message m de le traiter. La distinction entre `recevoir()` et `livrer()` est nécessaire. Dans le premier cas, le message est reçu par le processus, mais est placé dans une file d'attente des messages à traiter. Dans le second cas, le message est traité par l'algorithme qu'exécute le processus, potentiellement après le traitement d'autres messages. Par exemple, un processus peut avoir besoin de recevoir plusieurs messages identiques provenant de sources différentes avant de les livrer à l'algorithme distribué qu'il exécute et prendre une décision. Notons que nous supposons que chaque message peut être identifié de manière unique. Une manière de garantir cette hypothèse est d'identifier chaque message avec un numéro de séquence unique.

Les liens réseaux peuvent exposer différentes propriétés, que nous détaillons ci-dessous.

Perte de messages Les messages peuvent être perdus par le réseau. Cela arrive par exemple lorsqu'un tampon d'écriture est plein et ne peut plus accepter de nouveaux messages. Afin de contourner la perte des messages, un processus peut les retransmettre tant qu'il n'a pas reçu une confirmation de leur bonne réception.

Modification et création de messages Les messages peuvent être modifiés, dupliqués ou créés par le réseau ou par une entité tierce. Cela peut être le cas lorsqu'un composant réseau, tel qu'un commutateur, est défectueux. Afin d'empêcher la création ou la modification des messages en transit, les processus peuvent avoir recours aux mécanismes de cryptographie présentés dans la Section 1.1.3.

Duplication et réception désordonnée de messages Les messages peuvent être dupliqués par le réseau, ou alors l'ordre de réception des messages peut être différent de l'ordre d'envoi. Un message peut se retrouver dupliqué suite à des erreurs de transmission au niveau des couches basses des protocoles réseaux. L'ordre de réception des messages peut être différent de leur ordre d'envoi si deux messages n'utilisent pas la même route pour parvenir à la même destination. Dans les deux cas, un processus peut avoir à gérer ces messages. Une manière simple est d'attribuer un numéro de séquence qui définit un ordre de livraison des messages et d'utiliser un tampon de sauvegarde des messages reçus mais non encore livrés.

Aujourd'hui, deux protocoles de communication sont majoritairement utilisés pour implanter des systèmes distribués : UDP et TCP. Le premier, très simple, fournit un moyen d'échanger des messages entre plusieurs processus dans un mode déconnecté, c.à.d. que l'émetteur et le destinataire du message n'ont pas besoin d'établir une connexion entre eux afin de communiquer ; l'émetteur a seulement besoin de connaître l'adresse du destinataire. En utilisant UDP, les messages peuvent être perdus et leur réception peut être désordonnée. Enfin, UDP fournit une primitive de multidiffusion, qui permet à un même message d'être reçu par plusieurs processus mais envoyé qu'une seule fois. Le second fournit un moyen d'échanger des flux de données entre deux processus seulement dans un mode connecté, c.à.d. que l'émetteur et le destinataire

du message ont besoin d'établir une connexion entre eux afin de communiquer. TCP fournit un canal de communication sans perte et dans lequel la réception des messages se fait dans le même ordre que leur émission. Pour cela il implante un mécanisme de retransmission et de notification des messages. TCP implante également un mécanisme de gestion de la congestion, qui permet de partager un lien réseau entre plusieurs processus de manière équitable.

1.2 Réplication de machines à états

Une manière d'améliorer la fiabilité des systèmes est d'utiliser la réplication. La réplication de machines à états consiste en l'utilisation de plusieurs copies d'un système, implanté sous forme d'une machine à états, afin de masquer les fautes [18, 20], quelque soit leur type, de façon à ce que le système reste disponible. Chaque copie du système, appelée *réplica*, est placée sur un nœud différent.

Dans la suite de cette section nous considérons un système distribué S possédant une entrée et une sortie (cf. Figure 1.1). Ce système distribué fournit un *service* et possède une interface très simple : il reçoit en entrée des *requêtes* et retourne en sortie des *réponses*. Les processus qui souhaitent accéder au service sont appelés *clients* et communiquent avec ce dernier via des requêtes et réponses uniquement. Cette

FIGURE 1.1 – Représentation d'un système distribué S. Ce système possède une entrée, qui reçoit des *requêtes*, et une sortie, qui renvoie des *réponses*.

représentation du système s'accorde avec les systèmes distribués existants (p. ex. un serveur web) et permet de simplifier leur conception et leur étude.

1.2.1 Machines à états

Une machine à états est une représentation d'un système sous forme d'automate. Cette représentation est très utilisée dans les domaines de la modélisation de systèmes, de la vérification de programmes et de l'étude des langages formels, car elle permet de représenter et d'étudier facilement un système complexe.

Une machine à états est composée d'un ensemble d'états et d'un ensemble de transitions. La transition entre deux états s'effectue selon une condition. L'évaluation de cette condition dépend de l'état dans lequel se trouve l'automate ainsi que des données en entrée. Suite à une transition, la machine à états se retrouve dans un nouvel état et peut renvoyer une donnée en sortie.

Prenons pour exemple un serveur web, illustré dans la Figure 1.2. Les données en entrée correspondent aux demandes de connexion, de déconnexion et aux requêtes de pages web. Les données en sortie correspondent aux pages web demandées. Le serveur web est composé de trois états, e_0 à e_2. Le premier état, e_0, est l'état initial, c.à.d. l'état

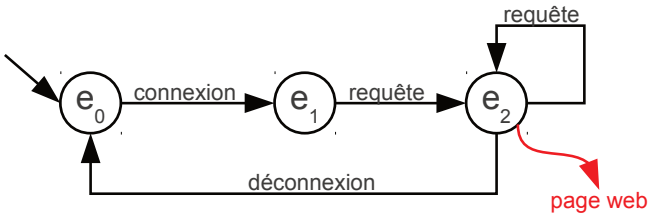


FIGURE 1.2 – Représentation d'un serveur web implanté sous forme de machine à états, composé de trois états, e_0 à e_2. La sortie du système, représenté en rouge, correspond aux pages web retournées.

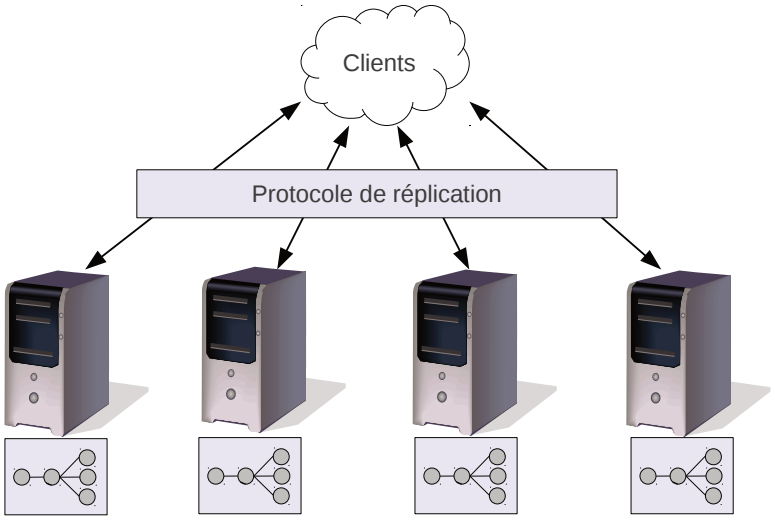


FIGURE 1.3 – Réplication de machines à états. Le système est composé de 4 réplicas (implantés par des machines à états déterministes) et d'un protocole de réplication, qui permet d'assurer la cohérence du système.

dans lequel se trouve le serveur web au début de son exécution. Lorsque le serveur web reçoit une requête, que ce soit une demande de (dé)connexion ou une demande de page web, alors il change d'état. Dans le dernier cas il produit un résultat, qui est la page web demandée.

Les machines à états peuvent être déterministes ou non déterministes. Dans le premier cas, leurs sorties ne dépendent que de leurs entrées. Dit autrement, une machine à états déterministe retournera toujours la même sortie pour une même entrée. Cela n'est pas le cas des machines non déterministes, dont les sorties dépendent des entrées ainsi que d'un état interne, tel qu'une valeur aléatoire. Dans le cadre de la réplication, nous ne nous intéressons qu'aux machines à états déterministes. En effet, celles-ci permettent de garantir l'état de plusieurs copies du système sera identique lorsque soumis aux mêmes entrées et que leurs états ne divergeront pas.

1.2.2 Vue d'ensemble de la réplication

Comme nous l'avons défini, la réplication de machines à états consiste en l'utilisation de plusieurs copies du système. Chaque copie du système, appelée *réplica*, est implantée sous la forme d'une machine à états déterministe. Soit N le nombre total de réplicas présents dans le système, la réplication permet de tolérer au maximum $f < N$ fautes simultanées. Le nombre total de réplicas, N, dépend du type et du nombre de fautes que l'on veut tolérer. Par exemple, pour résoudre le problème du consensus présenté dans la Section 1.2.4, $N = 2f + 1$ réplicas sont suffisants pour tolérer f fautes simultanées de type arrêt inopiné [21] et $N = 3f + 1$ réplicas sont suffisants pour tolérer f fautes simultanées de type Byzantines [14, 21]. Notons que chaque réplica possède un identifiant unique $i \in \{0, N - 1\}$.

Les protocoles de réplication de machines à états doivent maintenir l'hypothèse qu'il n'y aura jamais plus de f fautes simultanées à n'importe quel instant. Si les réplicas sont identiques, alors un bug ou une faille de sécurité présente sur l'un sera également présent sur les autres, ce qui viole cette hypothèse. Afin de la maintenir valide, les protocoles de réplication de machines à états doivent assurer que les fautes sont indépendantes entre les copies [22]. Cette diversité des copies peut être obtenue via des techniques de *N-version programming*, où différentes implantation du système, potentiellement par des entreprises différentes, sont écrites [23, 24, 25, 26]. Il est également possible d'utiliser des systèmes d'exploitation différents [27] ou encore du matériel différent.

Également, les protocoles de réplication de machines à états doivent fournir des propriétés appartenant aux deux classes suivantes [28, 29, 30] :

Propriétés de sûreté Quelque chose de "mal" n'arrivera jamais durant l'exécution. Une propriété de sûreté est par exemple le fait qu'un processus correct, initialement dans un état valide (c.à.d. en accord avec sa spécification), terminera son exécution dans un état valide.

Propriétés de vivacité Quelque chose de "bien" arrivera durant l'exécution. Des exemples de propriétés de vivacité sont l'exécution du système progresse à terme ou encore l'exécution du système va terminer (p. ex. le système ne rentrera pas dans une boucle infinie dont il ne sortira jamais).

D'après Schneider [20], répliquer une machine à états tolérant f fautes demande à ce que les réplicas soient coordonnés : tous les réplicas corrects doivent recevoir et traiter la même séquence de requêtes. Cette propriété peut être décomposée en deux sous-propriétés : (i) chaque réplica correct doit recevoir chaque requête (*accord*) et (ii) chaque réplica correct doit traiter les requêtes qu'il reçoit dans le même ordre (*ordre*). Très souvent, les protocoles de réplication de machine à états garantissent la propriété d'accord en implémentant une primitive de communication spéciale, appelée *diffusion à ordre total* (*"total-order broadcast"*) ou *diffusion à ordre uniformément total* (*"atomic broadcast"*) [31, 32]. Cette primitive garantit les quatre propriétés suivantes : (i) si un processus correct p diffuse un message m, alors p délivre m à terme (*OT-validité*) ; (ii) si un processus correct p délivre m, alors chaque processus correct q délivre m à terme (*OT-accord*) ; (iii) pour chaque message m, chaque processus correct p délivre m au plus une fois. De plus, si l'émetteur de m est correct, alors il a auparavant diffusé le message m (*OT-intégrité*) et (iv) si les processus corrects p et q délivrent les messages m et m', alors p a délivré m avant m' si et seulement si q a délivré m avant m' (*OT-ordre*).

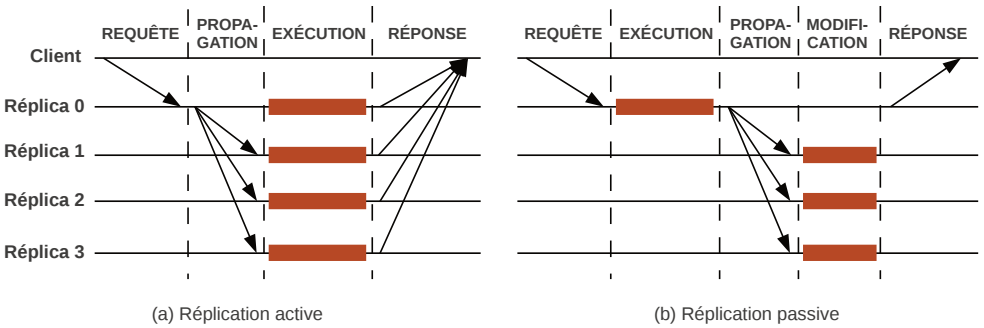


FIGURE 1.4 – Réplication de machines à états active et passive.

Nous présentons dans la section suivante différentes manières permettant de s'assurer que tous les réplicas corrects traitent les requêtes dans le même ordre afin de garantir la propriété d'*ordre*.

Il est important de noter que les protocoles de réplication de machines à états ne résolvent pas le problème de la confidentialité des données : une machine compromise peut transmettre des données confidentielles [33,34,35,36]. Ce problème est orthogonale aux problèmes adressés dans ce manuscrit.

La Figure 1.3 présente une vue d'ensemble de la réplication de machines à états. Dans cette exemple, le système est implanté sous la forme d'une machine à états déterministe et est répliqué sur 4 machines. Afin de garder un état cohérent, les réplicas doivent exécuter les mêmes requêtes, de manière déterministe et dans le même ordre. Pour cela, il est nécessaire que les réplicas implantent des machines à états déterministes. De plus, les réplicas communiquent entre eux à travers un protocole de réplication, qui est en charge de garantir le fait que le système se comportera en accord avec sa spécification et que l'état entre les réplicas sera cohérent, malgré la présence d'au plus f fautes. La communication entre les clients et les réplicas, ainsi qu'entre les réplicas, peut être implantée de différentes manières, que nous détaillons dans la section suivante.

1.2.3 Différentes formes de réplication

Il existe plusieurs manières d'implanter le protocole de réplication. La réplication peut être transparente, auquel cas le client n'a pas conscience d'utiliser un service répliqué, ou non, auquel cas le client est conscient de la réplication du système. Généralement, un intergiciel est placé entre le client et le service afin de masquer la réplication.

De plus, il existe deux approches pour l'exécution des requêtes : soit tous les réplicas exécutent les requêtes (on parle alors de *réplication active*), soit un seul réplica exécute les requêtes et transmet les modifications aux autres réplicas (on parle alors de *réplication passive*). La Figure 1.4 présente l'échange de message qui se déroule au sein d'un protocole de réplication active et d'un protocole de réplication passive afin d'exécuter une requête. Dans le premier cas, tous les réplicas exécutent la requête et répondent au client. Dans le second cas seul le réplica 0 exécute la requête, après quoi il envoie les modifications aux autres réplicas et répond au client.

Dans le cas de la tolérance aux fautes Byzantines, seule la réplication active peut être

utilisée. En effet, avec la réplication passive, un seul réplica, le coordinateur, exécute les requêtes. Les autres réplicas ne font que recevoir les modifications à prendre en compte sur leur état. Si le coordinateur est malicieux, alors il peut décider de ne pas exécuter les requêtes correctement. Or, ni le client ni les autres réplicas ne s'en rendront compte. Cela n'est pas le cas avec la réplication active, où chaque réplica exécute la requête et répond au client. Le client peut ainsi comparer les réponses des différents réplicas afin de s'assurer de leur validité.

Enfin, pour que l'état du système reste cohérent, les réplicas doivent choisir l'ordre d'exécution des requêtes reçues de la part des clients. Cela peut être fait de manière coopérative ou non. Dans le premier cas, il n'y a pas de distinction entre les rôles des différents réplicas. L'algorithme FloodSet [37], qui tolère des fautes de type arrêt inopiné, et l'algorithme présenté dans [38], qui tolère des fautes de type Byzantines, sont des exemples de protocoles de réplication où l'ordre d'exécution des requêtes est définit de manière coopérative. Dans le second cas, un réplica spécial, appelé *principal*, est chargé de définir un ordre d'exécution. Le protocole Paxos [39], qui tolère des fautes de type arrêt inopiné, et le protocole PBFT [40], qui tolère des fautes de type Byzantines, sont des exemples de protocoles de réplication basés sur un principal.

1.2.4 Le problème du consensus

Comme nous l'avons vu dans la section précédente, un problème important des protocoles de réplication de machines à états est de devoir définir un ordre d'exécution des requêtes. Ce problème peut être ramené au problème du *consensus*, qui est un problème fondamental dans le domaine des systèmes distribués.

Le problème du consensus est le suivant : plusieurs processus proposent différentes valeurs et collaborent afin de décider une valeur unique. Un protocole qui résout le problème du consensus assure les quatre propriétés suivantes : (i) tous les processus corrects décident à terme une valeur (*terminaison*) ; (ii) si un processus décide v, alors v a été proposée par certains des processus (*validité*) ; (iii) aucun processus ne décide deux fois (*intégrité*) ; (iv) il n'existe pas deux processus corrects qui décident une valeur différente (*accord*).

Indépendamment du modèle de fautes du système, certains processus peuvent ne pas participer à l'exécution du système distribué. Les systèmes distribués utilisent une notion de sous-ensemble de processus, appelée *quorum*, et prennent des actions une fois qu'ils ont récolté un quorum de messages identiques. Par exemple, les protocoles de réplication tolérant les fautes Byzantines utilisent des quorums de taille $2f + 1$, c.à.d. attendent $2f + 1$ messages de la part de processus différents, afin de garantir qu'une majorité de processus corrects va prendre la même action.

Le protocole de validation en deux phases [41] et le protocole de validation en trois phases [42], présentés respectivement par J. Gray en 1978 et M.D. Skeen en 1982, font partie des premiers protocoles à avoir résolu le problème du consensus. Ces protocoles ont donné lieu à de nombreux travaux dans le domaine, tels que les protocoles de la famille Paxos [39, 43, 44] ou les protocoles de réplication tolérant les fautes Byzantines [40, 45].

1.3 Modèle du système

Nous décrivons dans cette partie le modèle du système que nous considérons dans la suite de ce manuscrit.

Nous considérons le modèle Byzantin dans la suite de ce manuscrit. Le système est composé de N nœuds. N'importe quel quantité finie de clients peut se comporter de manière arbitraire, et au plus $f = \lfloor \frac{N-1}{3} \rfloor$ nœuds sont malicieux (ce chiffre est la valeur inférieure théorique [21]). Les nœuds malicieux et les clients peuvent coopérer afin de compromettre le service répliqué. Néanmoins, ils ne peuvent pas casser les opérations de cryptographie (p. ex. les signatures, les codes d'authentification de message, et les méthodes de hachage qui résistent aux collisions).

Les réplicas possèdent chacun une clé secrète ainsi qu'une paire de clé privée/publique afin de calculer respectivement les MACs et les signatures. Nous supposons que l'échange de ces clés entre les réplicas s'est déroulée en toute confidentialité et qu'un attaquant ne peut pas obtenir la clé secrète ou privée d'un réplica ou d'un client (et par conséquent ne peut se faire passer pour un autre nœud).

Nous notons un message m signé avec la clé publique du nœud i par $\langle m \rangle_{\sigma_i}$, un message m authentifié par le nœud i avec un MAC pour le nœud j par $\langle m \rangle_{\mu_{i,j}}$, et un message m authentifié par le nœud i avec un authentificateur MAC, c.à.d un tableau contenant un MAC par nœud, par $\langle m \rangle_{\vec{\mu}_i}$. Notre modèle s'accorde avec les hypothèses des autres papiers dans le domaine, par exemple [40, 46].

Nous supposons que le réseau est *ultimement synchrone*, c.à.d qu'il est asynchrone avec des intervalles synchrones, durant lesquels les messages sont délivrés avant un délai borné inconnu, qui arrivent infiniment souvent. Fischer, Lynch et Patterson ont prouvé [47] que dans un système complètement asynchrone, aucun algorithme de consensus asynchrone déterministe ne peut tolérer ne serait-ce que la moindre faute par arrêt inopiné. Les périodes synchrones sont nécessaires afin d'assurer les propriétés de vivacité. De plus, les canaux de communication respectent les propriétés du modèle *Stop-Reprise* d'Aguilera et al. [48] : les canaux de communication ne peuvent pas créer de messages, mais peuvent dupliquer ou perdre des messages un nombre fini de fois. Il est possible de rendre ces canaux fiables, en retransmettant les messages ou en utilisant le protocole TCP.

Dans la suite de ce manuscrit, nous abrégeons *Réplication de machines à états tolérant les fautes Byzantines* par **TFB** (*Tolérance aux Fautes Byzantines ; "Byzantine Fault Tolerance"*). Cette abréviation est largement répandue dans la littérature du domaine (p. ex. [49]).

2

État de l'art des protocoles de TFB

Sommaire

Nous présentons dans ce chapitre l'état de l'art des protocoles de tolérance aux fautes Byzantines. Ces protocoles peuvent être séparés en deux catégories : les protocoles patrimoniaux, conçus pour offrir les meilleurs performances possibles, proches des performances d'un système non répliqué, dans le cas sans faute, et les protocoles dits robustes, conçus pour offrir des garanties de performance en cas d'attaque.

2.1 Protocoles de TFB patrimoniaux

Nous présentons dans cette section cinq protocoles dits patrimoniaux. Un protocole patrimonial peut être vu comme un protocole conçu pour être plus rapide que les protocoles précédents dans le cas sans faute. En effet, tour à tour, les protocoles que nous présentons dans cette section ont fourni de meilleures performances que les protocoles précédents dans le cas sans fautes. Le premier protocole présenté, PBFT, est le travail séminal de Castro et Liskov. Ce protocole est le premier protocole de réplication tolérant les fautes Byzantines réellement utilisable dans un système réel. Les protocoles suivants, Q/U, HQ, Zyzzyva et Aliph, ont été conçus pour réduire le coût de la réplication et améliorer les performances des protocoles de TFB dans le cas sans faute.

2.1.1 PBFT

En 1999, Miguel Castro et Barbara Liskov ont présenté PBFT [14,40,50], le premier protocole de réplication tolérant les fautes Byzantines réellement utilisable dans un système réel. En effet, contrairement aux protocoles précédents, PBFT implante tout un ensemble de techniques qui permettent de réduire le coût de la réplication, que nous décrivons dans la suite de cette section. PBFT a donné naissance à de nombreux protocoles par la suite (p.ex., Q/U, Aardvark). Ce protocole garantit des propriétés de sûreté et de vivacité dans un environnement ultimement synchrone et nécessite $N = 3f + 1$ réplicas pour tolérer f fautes Byzantines simultanées. Ce protocole est un protocole de réplication active (c.à.d. que tous les réplicas exécutent la requête) et utilise un réplica spécial, le principal, qui propose un ordre d'exécution des requêtes. Les réplicas se mettent ensuite d'accord sur cet ordre. Cela garantit que les requêtes sont exécutées dans le même ordre par tous les réplicas corrects. Tous les messages échangés lors de l'exécution de PBFT sont sauvegardés dans un journal spécial. Enfin, une exécution de PBFT est découpée en *vue*s, une vue étant définie comme une configuration du système dans laquelle l'un des réplicas est le principal.

Nous détaillons dans la suite de cette section le protocole d'ordonnancement des requêtes puis le mécanisme de changement de vue. Enfin, nous présentons les différentes optimisations dont PBFT bénéficie afin de réduire le coût de la réplication.

Ordonnancement d'une requête

Nous détaillons maintenant les étapes du protocole dans le cas sans faute. Ces étapes sont décrites dans la Figure 2.1. Dans ce protocole, le client envoie sa requête au principal. Ce principal créé un message PRÉ-PRÉPARATION, dont le but est de proposer aux autres réplicas l'ordonnancement de la requête. Ce message contient un numéro de séquence unique. Les réplicas corrects répondent alors au PRÉ-PRÉPARATION avec un message PRÉPARATION, qui est envoyé à tous les réplicas (notons que le principal n'envoie pas ce message). De manière intuitive, un réplica qui reçoit un tel message et le PRÉ-PRÉPARATION correspondant sait qu'il existe au moins un réplica, différent de lui-même, qui a accepté le PRÉ-PRÉPARATION. Une fois que les réplicas ont reçu $2f$ PRÉPARATION et le PRÉ-PRÉPARATION correspondant, alors ils sont d'accord sur l'ordre de la requête du client. Dit autrement, chaque réplica sait qu'il existe $2f + 1$

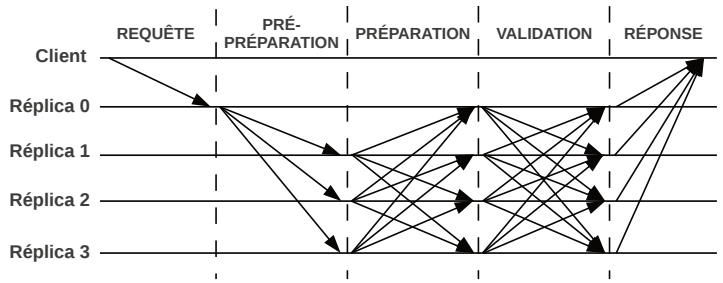


FIGURE 2.1 – Échange de messages avec un client dans PBFT ($f = 1$). Le réplica 0 est le principal.

réplicas qui sont d'accord sur l'ordre de la requête. Ils envoient alors, à tous les réplicas, un message VALIDATION. Une fois qu'un réplica a reçu $2f + 1$ VALIDATION, alors il exécute la requête et répond au client. Les $2f + 1$ VALIDATION constituent un *certificat de validation*. De manière intuitive, un réplica qui possède un certificat de validation sait qu'il existe $2f + 1$ réplicas (lui inclus) qui savent qu'il existe $2f + 1$ réplicas qui ont accepté l'ordre proposé par le principal. En l'absence de cette phase, les réplicas pourraient ne pas avoir suffisamment d'information pour finir d'ordonner des requêtes en cours lors d'un changement de vue.

L'exemple suivant illustre le problème causé par l'absence d'une phase de validation. Soit $f = 2$, le réplica 0 est le principal de la vue courante v et les réplicas 0 et 5 sont malicieux. Le principal envoie un PRÉ-PRÉPARATION pour la requête r_1 aux réplicas 2 à 5 avec le numéro de séquence n. Le réseau étant asynchrone, seuls les réplicas 2, 3 et 4 ont pu constituer un certificat de validation et ont ordonné la requête. Le principal, malicieux, envoie un PRÉ-PRÉPARATION pour la requête r_2 au réplica 1. À ce moment un changement de vue, pour la vue $v + 1$, se déclenche. Le réplica 1 devient principal. Il décide d'ordonner la requête r_2 avec le numéro de séquence n. En effet, il n'est pas au courant que ce numéro de séquence a déjà été attribué dans la vue précédente. Les réplicas 0, 1, 5 et 6 ordonnent alors la requête r_2 (les réplicas "oublient" les anciens messages qui n'ont pas donné lieu à un ordonnancement lors du changement de vue). Dans cet exemple, les réplicas sont dans un état inconsistant : pour le même numéro de séquence n, certains réplicas corrects ont ordonné la requête r_1 tandis que d'autres réplicas ont ordonné la requête r_2. Grâce à l'ajout de la phase de commit, aucun réplica correct n'ordonne la requête r_1 dans la vue v pour le numéro de séquence n.

Enfin, après que la requête ait été validée, le client attend $f + 1$ réponses identiques, provenant de réplicas différents, avant d'accepter la réponse comme résultat de l'exécution de sa requête.

Changement de vue

Si le client ne reçoit pas de réponse après une période de temps prédéfinie, alors il retransmet sa requête, cette fois ci à tous les réplicas. Lorsqu'un réplica reçoit une requête, il démarre un minuteur (appelé *minuteur de changement de vue*). Dans le cas sans faute la requête est exécutée avant l'expiration du minuteur. Toutefois, si le minuteur expire avant que la requête ne soit exécutée, alors il double la durée du

minuteur afin de garantir la propriété de vivacité du protocole. Puis il déclenche un *changement de vue* en envoyant à tous les réplicas un message CHANGEMENT-VUE qui contient, entre autres, les certificats de validation que le réplica possède depuis le dernier point de contrôle stable.

Lors d'un changement de vue, de la vue v à la vue $v + 1$, le réplica qui était le principal de la vue v quitte son rôle de principal, et un nouveau réplica est élu principal. Comme expliqué dans la Section 1.2.2, un identifiant unique est associé à chaque réplica dans l'ensemble $\{0, N - 1\}$. Dans PBFT, le principal de la vue v est défini comme le réplica $v\%N$. Notons que le protocole de changement de vue est un protocole distribué : un changement de vue n'intervient que si une majorité de réplicas (plus précisément, $2f + 1$) sont d'accord. Ce protocole de changement de vue garantit la propriété de vivacité. De plus, les réplicas s'échangent et synchronisent leur état durant ce protocole, ce qui permet par exemple de réparer un réplica dont l'état aurait divergé. En particulier, le protocole de changement de vue se termine par une phase dans laquelle le nouveau principal envoie un message NOUVELLE-VUE qui contient les dernières requêtes à avoir été ordonnées ou qui étaient en cours d'ordonnancement lors du changement de vue. Lorsque le principal est malicieux et dévie de son protocole, alors le protocole de changement de vue garantit qu'il sera ultimement remplacé par un réplica correct. Une déviation du protocole peut par exemple être un principal qui cesse d'ordonner les requêtes qu'il reçoit, ou un principal qui n'est pas équitable et qui privilégie les requêtes d'un ou plusieurs clients au détriment des autres.

Les réplicas stockent les messages qu'ils reçoivent et envoient dans un journal. Cela leur permet de les retransmettre si besoin, par exemple si un message se perd sur le réseau, ou si un réplica est redémarré. Toutefois, ils ne peuvent pas stocker une quantité infinie de message, et doivent périodiquement le tronquer. Pour cela, ils échangent périodiquement (toutes les 128 requêtes ordonnées) des messages de POINT-DE-CONTRÔLE qui contiennent le numéro de séquence n de la dernière requête ordonnée. Lorsqu'un réplica reçoit $f + 1$ POINT-DE-CONTRÔLEs identiques, il possède la preuve qu'au moins un réplica correct a reçu ce POINT-DE-CONTRÔLE. Ce point de contrôle est dit *stable*. Il tronque alors son journal en supprimant les messages reçus et envoyés jusqu'à la requête pour laquelle le numéro de séquence n a été attribué.

Implantation et optimisations

Tous les messages du protocole sont authentifiés par des MACs. La génération et la vérification d'un MACs est un ordre de grandeur moins coûteuse que pour une signature, ce qui permet au protocole de fournir de meilleures performances que les précédents, qui utilisaient exclusivement des signatures [51,52]. En effet, les signatures, bien que fournissant d'avantage de garantie, sont un ordre de grandeur plus coûteuses que les MACs. De plus, plusieurs requêtes peuvent être ordonnées en parallèle. Plus précisément, un PRÉ-PRÉPARATION peut servir à ordonner un lot de requêtes plutôt qu'une seule. Cela permet de réduire la charge CPU et réseau, en créant, transmettant et vérifiant moins de messages. De plus, afin de réduire le trafic réseau, les requêtes dépassant une certaine taille ne sont pas stockées dans le PRE-PEPARE. Au lieu de cela, il ne contient que les identifiants et empreintes des requêtes. C'est le client qui est chargé

de transmettre ses requêtes de taille importante à tous les réplicas [1]. Enfin, l'implantation de PBFT utilise le protocole de communication UDP avec la multidiffusion. Les réplicas souscrivent à un groupe de multidiffusion. Dès lors, lorsqu'un client ou réplica souhaite envoyer un message aux réplicas, il l'envoie à l'adresse du groupe de multidiffusion. Le message n'est émis qu'une seule fois sur le réseau mais bien reçu par tous les membres du groupe, ce qui permet d'économiser de la bande passante.

Dans cette section nous avons présenté le protocole PBFT, qui a été à l'origine d'une renaissance dans la conception de protocoles de TFB. En effet, PBFT, contrairement aux protocoles précédents, PBFT assure des propriétés de sûreté et de vivacité même dans un environnement asynchrone. Également, PBFT fournit de meilleures performances que les protocoles précédents grâce à l'utilisation de MACs uniquement, qui sont un ordre de magnitude moins coûteux à générer et vérifier que les signatures, utilisées par les protocoles précédents.

2.1.2 Q/U

Le protocole Q/U [53] a été présenté en 2005 par Abd-El-Malek et al. suite au constat suivant : les protocoles de TFB traditionnels sont basés sur l'accord entre les réplicas : tous les réplicas corrects s'échangent des messages afin d'ordonner toutes les requêtes. Or, cela ne passe pas à l'échelle lorsque le nombre de fautes tolérées augmente. Par exemple, le débit de PBFT dans le cas sans faute baisse de 83% lorsque la configuration du système passe d'une à cinq fautes tolérées. L'approche utilisée par Q/U est différente, et permet, selon les auteurs, un meilleur passage à l'échelle : il n'est pas nécessaire d'impliquer tous les réplicas du système afin d'ordonner une requête. Impliquer un sous-ensemble des réplicas, appelé *quorum*, est suffisant.

Nous présentons dans la suite de cette section plus en détails les raisons du meilleur passage à l'échelle de Q/U. Puis nous détaillons le protocole d'ordonnancement des requêtes. Enfin, nous présentons brièvement quelques détails sur l'implantation du protocole.

Passage à l'échelle de Q/U

Une différence notable avec PBFT est que cette approche requiert $5f + 1$ réplicas au total et des quorums composés de $4f + 1$ réplicas. Bien que Q/U passe moins bien à l'échelle lorsqu'on considère le nombre de serveurs utilisés lorsque le nombre de fautes tolérées augmente, le modèle de communication employé par les deux protocoles est différent. Plus précisément, une requête dans Q/U est ordonnée en deux phases, contre 5 pour PBFT. Le nombre de messages que chaque réplica reçoit et envoie est également plus faible dans Q/U que dans PBFT : 2 contre $4f + 3$. Nous pouvons observer que ce nombre ne dépend pas du nombre de fautes tolérées dans Q/U, contrairement à PBFT, ce qui lui permet un meilleur passage à l'échelle.

Une autre raison pour laquelle Q/U passe mieux à l'échelle est que l'état du système n'est pas nécessairement répliqué entièrement sur tous les réplicas. Cela permet de répartir la charge équitablement entre les différents réplicas lorsque les requêtes des

1. Dans notre implantation de PBFT, la taille d'une requête est qualifiée d'importante lorsqu'elle dépasse 1ko.

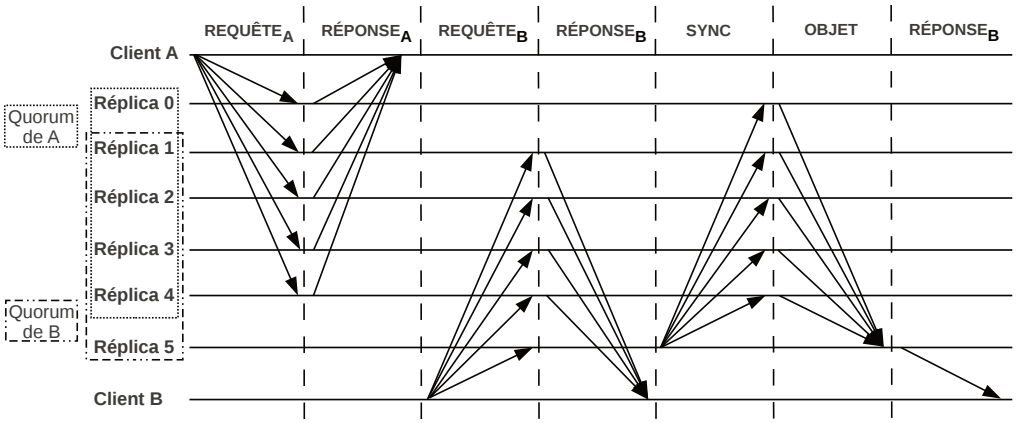


FIGURE 2.2 – Échange de messages avec deux clients dans Q/U ($f = 1$). Les deux clients accèdent à un même objet, répliqué sur les réplicas du quorum du client A.

clients nécessitent des objets distincts. Ainsi, le système peut fournir un meilleur débit que dans le cas où tous les réplicas corrects doivent traiter toutes les requêtes. Par exemple, il est possible d'avoir les objets du client a stockés sur le réplica 1, les objets du client b stockés sur le réplica 2, et les objets du client c stockés sur les réplicas 0 et 1. Plus précisément, chaque objet possède un quorum préféré : les réplicas de ce quorum ont une copie à jour de l'objet. Les réplicas peuvent transmettre des objets au besoin, lorsqu'un réplica tente d'accéder à un objet qu'il ne possède pas. Pour cela, les objets sont associés à un numéro de version. Un réplica peut posséder plusieurs versions d'un même objet. Les clients doivent envoyer, avec leurs requêtes, un historique des objets. Cela permet d'assurer que les réplicas répondent bien avec la bonne version des objets et permet aux différents réplicas d'ordonner les requêtes dans le même ordre.

Ordonnancement d'une requête

La figure 2.2 détaille les étapes du protocole. Dans cette figure, deux clients, A et B, accèdent à un objet répliqué uniquement sur les réplicas du quorum de A. Dans le cas normal, traiter une requête ne prend qu'un seul aller-retour : le client A envoie sa requête à tous les réplicas de son quorum. Chaque réplica reçoit la requête, l'exécute, puis répond au client. Le client attend une réponse de chacun des réplicas de son quorum avant de l'accepter comme résultat de l'exécution de sa requête. Dans le cas où l'objet n'est pas répliqué sur tous les réplicas du quorum, un échange de messages supplémentaire est nécessaire. Dans la figure ci-dessus, le client B souhaite accéder à un objet qui se trouve sur les réplicas du quorum de A uniquement. Une fois sa requête reçue par les réplicas de son quorum, les réplicas 1 à 4 possèdent l'objet et peuvent lui répondre directement. Toutefois, ça n'est pas le cas du réplica 5. Il envoie alors un message SYNC aux réplicas 0 à 4, qui possèdent l'objet, leur demandant de bien vouloir le lui envoyer. Les réplicas répondent alors avec l'objet en question. Le réplica 5 attend d'avoir reçu $f + 1$ OBJET identiques avant d'accepter le message. Enfin, maintenant qu'il possède la bonne copie de l'objet, il peut exécuter la requête et répondre au client.

Dans Q/U, le client n'accepte une réponse à sa requête que si $4f + 1$ réplicas, c.à.d. tous les réplicas du quorum, ont répondu de manière identique. En cas de fautes, il peut arriver que le client ne reçoive pas des réponses identiques ou aucune réponse de la part des réplicas malicieux. Pour éviter cela le client peut, lorsqu'il détecte un problème, *sonder* le système et changer les quelques réplicas défectueux de son quorum par d'autres réplicas. Ainsi, même en cas d'attaque, un client recevra une réponse de chacun des réplicas du quorum à terme.

Implantation

L'implantation de Q/U utilise le protocole de communication TCP, qui fournit un canal de communication sans pertes et une livraison des paquets dans le même ordre que l'ordre d'envoi. À l'inverse d'UDP, utilisé dans PBFT, il n'est pas possible de faire de multidiffusion : un message envoyé à n destinataires est transmis n fois sur le réseau. Toutefois, TCP implante des mécanismes de retransmission des messages et de contrôle de la congestion qui permettent de limiter le nombre de retransmission des messages lorsque le réseau est surchargé ou peu fiable, contrairement à UDP où la retransmission est gérée par l'application, sans prendre en compte l'état du réseau. L'utilisation de TCP permet un meilleur passage à l'échelle lorsque le nombre de fautes tolérées augmente.

Comme nous avons pu le voir, Q/U est un protocole de TFB construit autour de la notion de quorums afin d'offrir un meilleur passage à l'échelle lorsque le nombre de fautes tolérées augmente. Plus précisément, exécuter une requête dans Q/U n'implique qu'un sous-ensemble des réplicas, moins de phases de communication et d'échange de messages, et l'état du système peut être partitionné entre les différents réplicas afin de répartir la charge entre tous les réplicas de manière équitable.

Dans le cas sans faute et en l'absence de contention, c.à.d. lorsque plusieurs clients tentent d'accéder simultanément à un même objet, Q/U fournit de meilleures performances que PBFT. Toutefois, ce n'est pas le cas en présence de contention. Le protocole HQ, que nous présentons dans la section suivante, identifie et corrige ce problème.

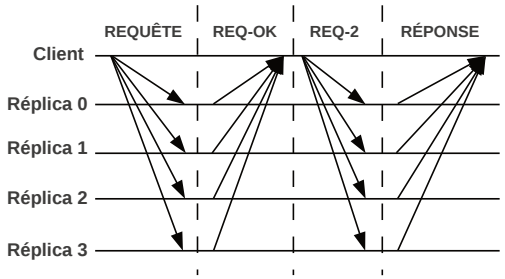
2.1.3 HQ

Nous avons présenté dans les sections précédentes PBFT et Q/U. Nous avons vu qu'un défaut de PBFT est de ne pas passer à l'échelle lorsque le nombre de fautes augmente. Ce problème est résolu par Q/U. Toutefois, il introduit un nouveau problème qui n'est pas présent dans PBFT : Q/U fournit de mauvaises performances lorsqu'il y a de la contention, c.à.d. lorsque plusieurs clients tentent d'accéder simultanément à un même objet. Plus précisément, deux réplicas qui reçoivent deux requêtes en parallèle peuvent les exécuter dans un ordre différent. De plus, Q/U requiert davantage de réplicas que PBFT : $5f + 1$ contre $3f + 1$. C'est pourquoi, en 2006, Cowling et al. ont présenté le protocole HQ [54]. Ce protocole utilise le meilleur de PBFT et Q/U afin de fournir les meilleures performances possibles dans le cas sans faute : HQ utilise un protocole basé sur la notion de *quorum* en l'absence de contention, et un protocole basé sur l'accord entre les réplicas en présence de contention.

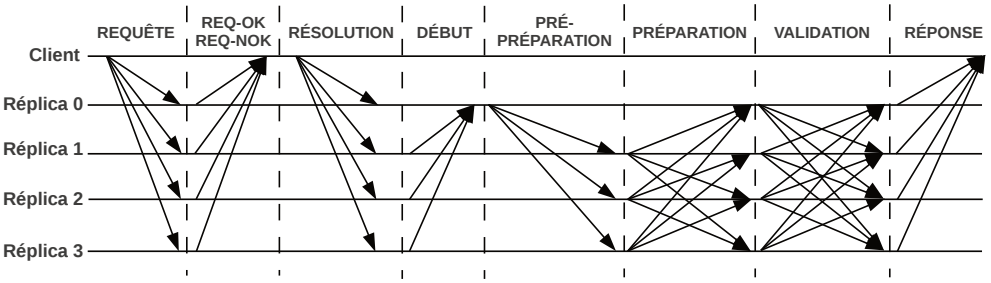
HQ est un protocole de TFB qui requiert $3f + 1$ réplicas, dont un principal, et des quorums de taille $2f + 1$ réplicas. En l'absence de contention, une requête est exécutée

en deux phases : la première phase permet d'associer un numéro de séquence à la requête, et donc de définir l'ordre d'exécution des requêtes ; la seconde phase est la phase durant laquelle la requête est exécutée. En présence de contention, les réplicas ordonnent et exécutent la requête en utilisant PBFT. Tout comme Q/U, chaque objet a un quorum préféré, c.à.d. un ensemble de réplicas sur lesquels il est répliqué. De plus, les clients envoient leurs requêtes à tous les réplicas, mais seuls ceux appartenant au quorum sont obligés de répondre.

La figure 2.2 détaille les étapes du protocole en l'absence et en présence de contention. Afin de simplifier l'étude du protocole, nous avons modifié le nom des étapes telles que décrites dans le papier [54]. Par exemple, l'étape RÉPONSE se nomme en réalité WRITE-2-ANS. Lorsqu'un client souhaite exécuter une opération, il envoie sa



(a) En l'absence de contention.



(b) En présence de contention.

FIGURE 2.3 – Échange de messages avec un client dans HQ ($f = 1$). Le réplica 0 est le principal.

requête à tous les réplicas. Chaque réplica doit associer un numéro de séquence à cette requête. S'il n'autorise pas ce client à obtenir le prochain numéro de séquence, car il a été attribué à un autre client, alors il répond avec un message REQ-NOK. Sinon il répond avec un message REQ-OK ou, dans le cas où il a déjà exécuté la requête, avec la réponse. Le client a la preuve que sa requête peut être ordonnée lorsqu'il reçoit un quorum de REQ-OK pour un même numéro de séquence. C'est ce qui arrive en l'absence de contention. Le client envoie alors cette preuve dans un message REQ-2. À la réception d'un tel message, un réplica vérifie la validité de la preuve puis exécute la requête et répond au client. Enfin, le client attend un quorum de réponses identiques avant d'accepter la réponse. En présence de contention, le client recevra des réponses différentes lors de la première phase, p. ex. des REQ-OK de la part de certains réplicas et de REQ-NOK de la part d'autres réplicas. Cela arrive lorsque plusieurs clients sont en

compétition sur le même numéro de séquence. Dans ce cas, le client envoie un message RÉSOLUTION à tous les réplicas, qui contient les différentes réponses reçues lors de la première phase. Notons qu'un réplica ne peut résoudre qu'un seul conflit à la fois. S'il n'est pas déjà en train de résoudre un conflit, alors il envoie un message DÉBUT au principal de PBFT (rappelons que HQ utilise PBFT afin de résoudre les conflits). Lorsque le principal reçoit un quorum de DÉBUT valides (incluant le sien), alors il démarre le validation en trois phases de PBFT en envoyant un PRÉ-PRÉPARATION. Une fois que la requête a été ordonnée par les réplicas, alors ils l'exécutent et répondent au client. Enfin, lorsque le client reçoit $f + 1$ réponses identiques, il accepte la réponse.

Nous avons présenté HQ dans cette section, un protocole de TFB qui combine deux protocoles afin de fournir de meilleures performances et un meilleur passage à l'échelle que les protocoles précédents dans le cas sans faute. En l'absence de contention, HQ exécute un protocole ressemblant à Q/U qui lui permet de fournir de meilleures performances que TFB et un meilleur passage à l'échelle. Toutefois, ce n'est pas le cas en présence de contention. HQ bascule alors sur PBFT, qui fournit de meilleures performances dans ce cas.

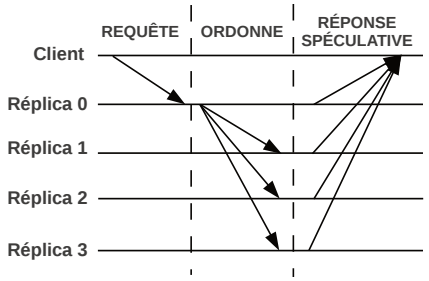
2.1.4 Zyzzyva

Zyzzyva [55] est un protocole de TFB présenté en 2009 par Kotla et al. et qui est basé sur la spéculation : les réplicas n'ont pas besoin de se mettre d'accord sur l'ordre d'exécution des requêtes, dans le cas sans faute. Zyzzyva fournit de très bonnes performances (supérieures aux performances des protocoles présentés précédemment) dans le cas sans faute au prix d'une réparation du système plus coûteuse lorsqu'une faute est détectée.

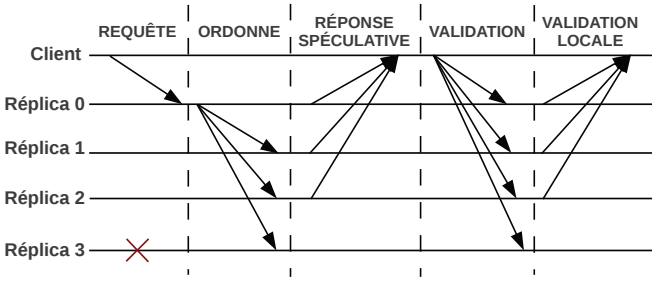
Zyzzyva nécessite $3f + 1$ réplicas, dont un principal, tout comme PBFT. Toutefois, contrairement à PBFT, les requêtes sont ordonnées de manière spéculative en seulement deux phases. Les clients attendent des réponses identiques de la part de chacun des réplicas (c.à.d. $3f + 1$) avant d'accepter le résultat de leurs requêtes. Les réponses renvoyées par les réplicas contiennent un historique qui permet aux clients de vérifier si les réponses sont stables, c.à.d. si les réponses font suite à la même séquence d'ordonnancement sur tous les réplicas, ou non, auquel cas l'état des réplicas a divergé. Nous pouvons comparer le protocole d'ordonnancement de Zyzzyva à celui de PBFT de la manière suivante : les réplicas de Zyzzyva sont chargés d'exécuter les étapes PRÉ-PRÉPARATION et PRÉPARATION, tandis que l'étape de VALIDATION est déléguée aux clients.

La Figure 2.4 détaille les étapes du protocole. Le client envoie sa requête au principal et démarre un minuteur. Une fois que le principal reçoit la requête, il envoie un message ORDONNE à tous les réplicas. Lorsqu'un réplica reçoit un tel message, il exécute la requête et répond de manière spéculative au client. Dans le cas sans faute le client reçoit $3f + 1$ réponses identiques avant l'expiration du minuteur et accepte le résultat de la requête. Dans le cas avec fautes (par exemple un réplica qui ne répond plus, ou un réplica qui répond de manière incorrecte) le client peut n'avoir reçu qu'entre $2f + 1$ et $3f$ réponses identiques [2] lorsque le minuteur arrive à expiration. Le client envoie alors

2. Le client retransmet sa requête périodiquement à tous les réplicas tant qu'il n'a pas obtenu au moins



(a) Exécution spéculative.



(b) Exécution en cas de fautes : le réplica 3 est malicieux et n'envoie aucun message.

FIGURE 2.4 – Échange de messages avec un client dans Zyzzyva ($f = 1$). Le réplica 0 est le principal.

un message VALIDATION à tous les réplicas, qui contient $2f + 1$ réponses identiques reçues. Lorsqu'un réplica reçoit un tel message valide, il répond au client avec un message VALIDATION LOCALE. Cet échange de messages est équivalent à la phase de validation de PBFT. Enfin, lorsque le client reçoit $2f + 1$ VALIDATION LOCALE valides, de la part de réplicas différents, alors il considère que la requête a été correctement ordonnée et son historique est valide : les réplicas corrects vont toujours exécuter cette requête au même instant dans leur historique.

La nature spéculative de Zyzzyva modifie le protocole de changement de vue. En effet, par rapport à PBFT, l'absence de phase de validation pose de nouveaux défis. Le but du protocole de changement de vue est d'élire un nouveau principal et de garantir que les réplicas corrects vont démarrer la nouvelle vue avec le même historique de requêtes. Le protocole de changement de vue de Zyzzyva possède deux différences notables avec celui de PBFT : (i) une nouvelle phase de communication, nommée JE-DÉTESTE-LE-PRINCIPAL, est introduite et (ii) les réplicas de Zyzzyva doivent savoir quelles requêtes exécutées de manière spéculative doivent être placées dans l'historique des requêtes au démarrage de la nouvelle vue.

Rappelons que la phase de validation de PBFT permet de garantir la vivacité du protocole lors d'un changement de vue : les réplicas vont pouvoir continuer à ordonner les requêtes qui étaient en cours d'ordonnancement lors du changement de vue. Zyzzyva doit pouvoir assurer la vivacité du protocole malgré l'absence de cette phase. Pour cela, un réplica correct ne change de vue que s'il a une garantie que tous les réplicas

$2f + 1$ requêtes, afin de s'assurer que sa requête sera ordonnée à terme.

corrects vont faire de même. S'il suspecte le principal, alors il envoie un message JE-DÉTESTE-LE-PRINCIPAL à tous les réplicas et continue à traiter les messages dans la vue courante. Si un réplica reçoit $f + 1$ tels messages, alors il initie le protocole de changement de vue en envoyant un message CHANGEMENT-VUE, à tous les réplicas, qui contient la preuve que $f + 1$ réplicas n'ont pas confiance dans le principal actuel. Ainsi, il existe au moins $f + 1$ réplicas qui sont prêts à initier un changement de vue.

Le second problème auquel le protocole de changement de vue de Zyzzyva fait face est que les réplicas peuvent ne pas savoir quelles requêtes ont complétées (c.à.d. quelles sont les requêtes pour lesquelles le client a accepté la réponse) mais doivent tout de même s'échanger un historique de requêtes valide afin de démarrer la nouvelle vue dans un état cohérent. Cela est vrai pour les requêtes ordonnées de manière spéculative (c.à.d., pour les requêtes pour lesquelles le client a reçu $3f + 1$ réponses). Les réplicas n'ont pas reçu de certificat prouvant que $2f + 1$ réplicas ont exécuté la requête, puisqu'il n'y a pas eu la phase de validation. Pour cela, lorsque les réplicas corrects initient le protocole de changement de vue, ils attachent tous les messages ORDONNE reçus depuis le dernier point de contrôle stable au message CHANGEMENT-VUE et l'envoient à tous les réplicas. De plus, l'historique envoyé par le nouveau principal, dans le message NOUVELLE VUE, contient toutes les requêtes dont le numéro de séquence est plus grand que le plus grand numéro de séquence des certificats de validation qui apparaît dans au moins $f + 1$ des messages CHANGEMENT-VUE que le nouveau principal collecte. Cela permet de garantir que tous les réplicas corrects vont commencer la nouvelle vue avec le même historique des messages.

Dans cette section nous avons présenté Zyzzyva, un protocole de TFB basé sur la *spéculation* afin de fournir de très bonnes performances dans le cas sans faute. En particulier, ses auteurs ont montré que son débit est jusqu'à 45% supérieur au débit de PBFT. En présence de fautes, Zyzzyva exécute un nouvel algorithme de changement de vue afin de rétablir un état cohérent entre les réplicas. Toutefois, ses auteurs précisent que son implantation est très compliquée, ce qui rend le protocole facilement vulnérable en cas d'attaque [13].

2.1.5 Aliph

Aliph est un protocole de TFB spéculatif construit autour d'une nouvelle abstraction, Abstract [49], qui permet de simplifier la conception de protocoles de TFB. Aliph combine trois protocoles de TFB : (i) Quorum, un protocole spéculatif qui ne fonctionne que lorsqu'il n'y a pas de contention et lorsque la charge est faible, (ii) Chain, un protocole spéculatif très rapide lorsque la charge est élevée, et (iii) Backup, un protocole de TFB patrimonial (en l'occurrence PBFT). Dans cette section, nous présentons tout d'abord Abstract. Nous présentons ensuite une vue générale d'Aliph. Enfin, nous présentons plus en détails Quorum et Chain.

2.1.5.1 Abstract

Abstract [49] (*Abortable Byzantine faulT toleRant stAte maChine replicaTion* ; *réplication de machines à état tolérant les fautes Byzantines qui peut avorter*) est une nouvelle abstraction qui permet de simplifier la conception de protocoles de TFB. L'idée

principale derrière Abstract est de combiner plusieurs protocoles de TFB, appelés *instances*, afin de construire un nouveau protocole. Chaque instance peut être implantée, testée et vérifiée indépendamment des autres. De plus, la modularité d'Abstract permet de réutiliser une grande partie du code source d'un protocole existant (notamment la partie réseau) afin d'implanter un nouveau protocole plutôt que de partir de zéro. Enfin, la composition de deux instances d'Abstract est idempotente : combiner deux instances d'Abstract qui assurent des propriétés de sûreté et de vivacité retourne une nouvelle instance d'Abstract qui assure les mêmes propriétés de sûreté et vivacité.

Chaque instance d'Abstract ordonne les requêtes des clients sauf si certaines conditions du système ne sont pas satisfaites (p. ex. de la contention apparaît, le réseau est asynchrone, un réplica est malicieux). Dans une telle situation, les protocoles de TFB patrimoniaux procéderaient à un changement de vue. Dans Abstract, le changement de vue est remplacé par un basculement vers la prochaine instance. Plus précisément, chaque instance d'Abstract possède un numéro d'identification unique i et a accès à une fonction $suivant(i)$, qui retourne l'identifiant de l'instance sur laquelle basculer. Cette fonction peut être statique (le choix de la prochaine instance est fixe) ou dynamique (le choix de la prochaine instance dépend de l'état actuel du système).

Nous présentons maintenant le protocole de basculement. Lorsqu'un client détecte un problème au niveau de l'instance courante (p. ex. il ne reçoit pas de réponse après une période donnée), alors il envoie un message PANIQUE à tous les réplicas. Lorsqu'un réplica reçoit un tel message valide, il arrête de traiter de nouvelles requêtes et envoie un message AVORTE au client. Ce message contient un historique des requêtes qui ont étés exécutées par l'instance courante jusqu'ici, ainsi que l'identifiant de la prochaine instance à utiliser. Lorsque le client reçoit $2f + 1$ AVORTE valides pour la même prochaine instance, alors il extrait un *historique d'avortement* à partir des historiques présents dans les $2f+1$ AVORTE. Enfin, le client envoie sa prochaine requête à l'instance suivante en même temps que l'historique d'avortement ainsi que les $2f + 1$ AVORTE. Un réplica qui reçoit cette nouvelle requête vérifie la validité de l'historique d'avortement. Le réplica doit également posséder, dans son propre historique, toutes les requêtes présentes dans l'historique d'avortement. Au cas où certaines requêtes manquent, il les demande auprès des autres réplicas.

2.1.5.2 Présentation générale d'Aliph

Aliph est un protocole de TFB nécessitant $3f + 1$ nœuds. Il est la composition de trois instances d'Abstract, entre lesquelles il bascule de manière statique :

Quorum : Fonctionne lorsque les réplicas et les clients sont corrects, le réseau est synchrone, et il n'y a pas de contention.

Chain : Fonctionne lorsque les réplicas et les clients sont corrects et le réseau est synchrone.

Backup : Fonctionne en présence de fautes. PBFT est le protocole implanté dans Backup.

Quorum et Chain permettent de fournir de très bonnes performances dans le cas sans faute : selon les auteurs, Aliph est plus performant que les protocoles existants (p. ex. Zyzzyva) dans le cas sans faute à la fois en terme de latence (jusqu'à 30%) et en terme de débit (jusqu'à 360%). En l'absence de contention, les requêtes sont ordonnées par

Quorum. Le protocole bascule vers Chain lorsque de la contention est détectée, c.à.d. lorsqu'un réplica reçoit deux nouvelles requêtes en parallèle sans qu'aucune des deux ne soit encore exécutée. En effet, ces deux requêtes pourraient être exécutées dans un ordre différent par les différents réplicas. Enfin, en cas de fautes, Aliph bascule vers Backup (c.à.d. PBFT) pour un nombre défini de requêtes, après quoi il bascule à nouveau dans Quorum, et ainsi de suite. Nous présentons Quorum et Chain plus en détails dans les sections suivantes.

2.1.5.3 Quorum

Quorum est une instance d'Abstract qui fournit de très bonnes performances en exécutant les requêtes de manière spéculative. Exécuter une requête ne nécessite que deux échanges de messages, comme décrit dans la Figure 2.5. Lorsqu'un client souhaite

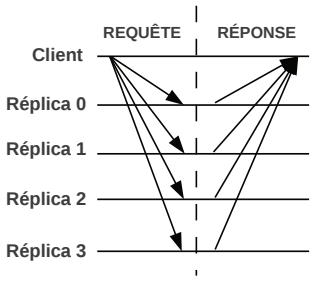


FIGURE 2.5 – Échange de messages dans Quorum ($f = 1$).

soumettre une requête au système, il l'envoie à tous les réplicas. Chaque réplica, après avoir reçu cette requête, l'exécute de manière spéculative et répond au client. Enfin, le client attend $3f + 1$ réponses identiques avant d'accepter le résultat de sa requête. Si tel n'est pas le cas, alors le client panique et initie un basculement de protocole vers Chain. Il est important de noter que Quorum ne peut fonctionner qu'en l'absence de contention. Sans cette propriété, c.à.d. en présence de contention, les clients peuvent recevoir des réponses différentes et l'état des réplicas peut diverger.

2.1.5.4 Chain

Chain est une instance d'Abstract qui fournit de très bonnes performances en organisant les réplicas à la manière d'un pipeline et en exécutant les requêtes de manière spéculative. Un réplica, appelé *tête*, est le point d'entrée de la chaîne et reçoit les requêtes, tandis qu'un autre réplica, appelé *queue*, est le point de sortie de la chaîne et renvoie les réponses aux clients. De plus, contrairement à Quorum, Chain est capable d'ordonner des requêtes en présence de contention sans que l'état des réplicas ne diverge. La particularité de Chain est qu'il nécessite un nombre minimal d'opérations de cryptographie au niveau du nœud qui fait office de goulot d'étranglement. Étant donné que les performances des protocoles de réplication sont bornées par le coût des opérations de cryptographie plutôt que par la vitesse du réseau, cela permet à Chain de fournir un débit supérieur au débit des protocoles de BFT patrimoniaux existants. La Figure 2.6 présente les étapes du protocole. Le client créé une requête qui contient un

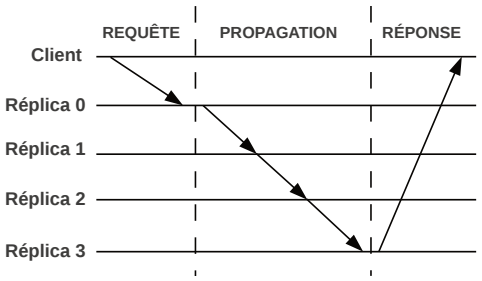


FIGURE 2.6 – Échange de messages dans Chain ($f = 1$). Le réplica 0 (resp. 3) est la tête (resp. queue) de la chaîne.

authentificateur de chaîne, c.à.d. un tableau de $f + 1$ MACs, qui permet à un ensemble de $f + 1$ réplicas, appelés successeurs, de vérifier un message. Cet authentificateur contient un MAC pour chacun des $f + 1$ premiers réplicas de la chaîne. Ensuite, le client envoie sa requête à la tête de la chaîne uniquement. Lorsque la tête reçoit une requête, elle (i) vérifie le MAC de ses prédécesseurs dans la chaîne (en l'occurrence le client), (ii) modifie l'authentificateur MAC afin d'enlever ceux de ses prédécesseurs (en l'occurrence celui du client) et d'inclure celui de ses successeurs, (iii) exécute la requête, puis (iv) transmet la requête au prochain réplica. Le processus est similaire pour les autres réplicas de la chaîne sauf la queue. Lorsque le réplica en queue de la chaîne reçoit la requête, il vérifie les $f + 1$ MACs de ses prédécesseurs puis en rajoute un pour le client, avant de transmettre la réponse au client. Notons que les f prédécesseurs de la queue ajoutent une empreinte de la réponse dans la requête qu'ils propagent, afin que le client puisse vérifier la validité de la réponse. Enfin, lorsque le client reçoit une réponse valide, il accepte le résultat de la requête. Sinon, il panique et initie un basculement vers Backup.

En résumé, Aliph est un protocole spéculatif conçu autour d'une abstraction permettant de simplifier la conception de protocoles de TFB. Dans le cas sans faute, Aliph minimise le nombre de phases de communications et d'opérations de cryptographie, ce qui lui permet de fournir de meilleures performances que Zyzzyva. En présence de fautes, Aliph bascule vers PBFT afin de garantir les propriétés de sûreté et de vivacité du protocole.

2.2 Protocoles de TFB conçus pour être robustes

Nous présentons dans cette section trois protocoles dits robustes, qui ont été conçus pour offrir des garanties de performance en cas d'attaque : Spinning [56], Prime [57], et Aardvark [13].

Les protocoles patrimoniaux que nous avons présentés dans la section précédente utilisent tous un réplica spécial : le principal. Le principal est le seul réplica qui choisit dans quel ordre ordonner les requêtes ; les autres réplicas ne font qu'être d'accord ou non avec cet ordre. Étant donné son rôle central, il peut dégrader les performances du système s'il est malicieux en retardant l'envoi des messages d'ordonnancement. Il est donc une source de non-robustesse, adressée par les protocoles conçus pour être

robustes.

Nous présentons dans la Section 2.2.5 des protocoles robustes qui ne nécessitent pas la présence d'un principal. Enfin, nous présentons une approche orthogonale à la conception de protocoles robustes qui est la tolérance aux intrusions (Section 2.2.6) : dans un protocole de tolérance aux intrusions, plutôt que de détecter et réparer les réplicas défectueux, les réplicas sont remis à neuf de manière périodique.

2.2.1 Définition de la robustesse

Le fait que les protocoles de TFB patrimoniaux sont fragiles (c.à.d. n'offrent aucune garantie de performance en cas d'attaque) est bien connu [13,56,57,58]. Cela a motivé le développement de protocoles dits *robustes* : Spinning [56], Prime [57], et Aardvark [13]. Le but de ces protocoles est d'offrir des garanties de performance lorsque le réseau est synchrone et que le système subit des attaques de la part de clients et réplicas Byzantins.

De manière plus formelle, nous définissons qu'un protocole est X-robuste lorsque sa chute de débit en cas d'attaque est d'au plus $X\%$ du débit dans le cas sans faute. Par exemple, un protocole de TFB dont la chute de débit est d'au plus 10% est 10-robuste. Dans ce manuscrit, nous considérons qu'un protocole n'est *robuste* que s'il est au plus 10-robuste. Cette valeur nous semble correspondre aux besoins de performances en cas d'attaque des applications existantes.

Les protocoles patrimoniaux présentés dans la section précédente ne répondent pas à cette propriété car, comme montré par Clement et al. [13], leur débit en cas d'attaque peut chuter à zéro [3]. Par exemple, un client malicieux dans PBFT peut forger une requête telle qu'un changement de vue sera déclenché à chaque fois qu'elle sera reçue par la majorité des réplicas. Un réplica malicieux peut également inonder le réseau de messages invalides, ce qui va très fortement ralentir l'exécution du système.

2.2.2 Prime

Prime [57], présenté en 2008 par Amir et al., part du constat que dans les protocoles de TFB patrimoniaux un principal malicieux peut retarder l'envoi des messages d'ordonnancement afin de dégrader les performances du système sans être détecté. Par exemple, dans le cas de PBFT, la durée du minuteur de changement de vue est augmentée par les réplicas à chaque changement de vue. Un attaquant peut forcer des changements de vue en envoyant des requêtes invalides, ce qui a pour conséquence d'augmenter la durée du minuteur. Un réplica malicieux peut alors profiter d'une durée de minuteur de changement de vue très élevée pour retarder l'envoi des messages d'ordonnancement, ce qui a pour conséquence de faire chuter le débit du protocole à zéro.

Prime, tout comme PBFT, est un protocole de TFB basé sur un principal et nécessite $3f + 1$ réplicas. Deux mécanismes sont au cœur de sa conception pour être robuste : (i) les réplicas s'échangent les requêtes entre eux afin de les pré-ordonner et (ii) les réplicas surveillent les performances du réseau afin d'avoir une idée du délai maximal entre deux messages d'ordonnancement envoyés par le principal.

Dans Prime, les clients peuvent envoyer leurs requêtes à n'importe quel réplica. Les réplicas s'échangent les requêtes qu'ils reçoivent de la part des clients, de la

3. L'étude de Clement et al. porte sur PBFT, Q/U, HQ et Zyzzyva. Nous montrons qu'Aliph n'est pas robuste dans le Chapitre 3.

façon suivante (cf. Figure 2.7). Un réplica qui reçoit une requête valide, r, lui attribut un numéro de séquence s puis envoie un message PO-REQUÊTE qui contient r et s. Lorsqu'un réplica reçoit un message PO-REQUÊTE pour une requête r et un numéro de séquence s de la part du réplica n, il envoie un message PO-ACK à tous les réplicas, qui contient l'empreinte de la requête, s et n. Un réplica qui a reçu un message PO-REQUÊTE et $2f$ messages PO-ACK correspondants (c.à.d. pour les mêmes valeurs d'empreinte de r, de s et de n) possède un *certificat de pré-ordonnancement*, c.à.d. une preuve comme quoi les réplicas se sont mis d'accord sur le pré-ordonnancement de la requête r avec l'identifiant (n, s). De manière périodique, les réplicas envoient un message PO-RÉSUMÉ, qui contient, pour chaque réplica n, le numéro de séquence maximal pour lequel ils possèdent un certificat de pré-ordonnancement. Ces messages sont agrégés par les réplicas et leurs permettent de se tenir au courant des requêtes qui ont été pré-ordonnées. Le principal peut ne pas recevoir de certificat de pré-ordonnancement. C'est pourquoi chaque réplica envoie périodiquement un message MATRICE DE RÉSUMÉ au principal. Lorsque le principal reçoit ce message, alors il sait quelles requêtes ont été pré-ordonnées et peuvent maintenant être ordonnées. Le principal doit envoyer périodiquement un message PRÉ-PRÉPARATION, potentiellement vide. Ce message contient, pour chaque réplica, le numéro de séquence de la dernière requête à avoir été pré-ordonnée. Cet identifiant est calculé à partir des messages PO-RÉSUMÉ et MATRICE DE RÉSUMÉ reçus durant la phase de pré-ordonnancement des requêtes. Les requêtes sont ensuite ordonnées de la même manière que dans PBFT, c.à.d. en suivant un protocole de validation en trois phases. Enfin, les réplicas exécutent les requêtes qui viennent d'être ordonnées puis répondent au client.

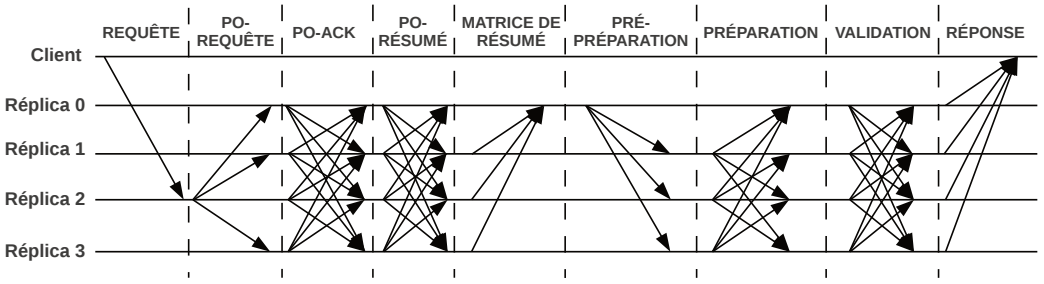


FIGURE 2.7 – Échange de messages dans Prime ($f = 1$). Le client peut envoyer sa requête à n'importe quel réplica (en l'occurrence le réplica 2).

Notons que les périodes d'envoi des messages PO-RÉSUMÉ, MATRICE DE RÉSUMÉ et PRÉ-PRÉPARATION peuvent être différentes. Par exemple, un message MATRICE DE RÉSUMÉ peut être envoyé après plusieurs messages PO-RÉSUMÉ et un message PRÉ-PRÉPARATION peut être envoyé à la même fréquence qu'un message PO-RÉSUMÉ. Ces périodes sont des paramètres de configuration définis manuellement par l'administrateur du système.

Le principal doit envoyer périodiquement des messages d'ordonnancement, même vides, afin de permettre aux réplicas de repérer un principal malicieux. En effet, ils s'attendent à recevoir des messages d'ordonnancement à une certaine fréquence. Afin d'améliorer la précision de la fréquence attendue, les réplicas surveillent les perfor-

mances du réseau. Plus précisément, les réplicas mesurent périodiquement le temps de réponse entre chaque paire de réplicas. Cette mesure leur permet de calculer le délai maximal qui doit séparer l'envoi de deux messages d'ordonnancement de la part d'un principal correct. Ce délai est calculé en fonction de trois paramètres : le temps de réponse entre les réplicas, la fréquence à laquelle un principal correct doit envoyer les messages d'ordonnancement, et une constante, fixée par l'administrateur système, qui prend en compte la variabilité de la latence réseau. Si le principal est plus lent que ce qui est attendu par les réplicas, alors ils le considèrent comme malicieux et procèdent à un changement de vue.

Enfin, notons que Prime n'utilise pas de MACs mais uniquement des signatures. Bien que plus coûteuses, les signatures fournissent davantage de garanties que les MACs, et en particulier la propriété de non-répudiation, décrite dans la Section 1.1.3.2 : si un message signé est correctement vérifié par un nœud correct, alors il le sera également pour tous les nœuds corrects. Ce n'est pas le cas avec les MACs : un authentificateur MAC peut être vérifié correctement par un sous-ensemble des nœuds seulement.

En résumé, Prime intègre une nouvelle phase de pré-ordonnancement des requêtes au protocole de validation en trois phases de PBFT ainsi qu'un mécanisme de surveillance des performances du réseau afin de surveiller le principal et de le remplacer s'il montre un comportement malicieux. Toutefois, Prime fournit des performances moindres que celles des protocoles de TFB patrimoniaux car tous les messages du protocole sont signés, ce qui induit un coût non-négligeable.

2.2.3 Aardvark

Clement et al. ont montré en 2009 que les protocoles de TFB patrimoniaux, PBFT, Q/U, HQ et Zyzzyva, ne sont pas robustes, et ont présenté un nouveau protocole conçu pour être plus robuste : Aardvark [13]. Ce protocole est similaire à PBFT : il requiert $3f + 1$ nœuds, dont un principal, et les étapes du protocole sont identiques. Toutefois, contrairement à PBFT, il implante toute une série de mécanismes pour améliorer sa robustesse. Nous détaillons ces mécanismes ci-dessous.

Premièrement, Aardvark implante un mécanisme intelligent d'authentification des requêtes des clients, qui limite l'impact des requêtes mal formées sur le système : les requêtes des clients possèdent un authentificateur MAC et une signature. Comme nous l'avons détaillé dans la section précédente, les signatures fournissent davantage de garanties que les MACs. En particulier, un nœud correct qui valide un MAC ne garantit pas que tous les nœuds corrects vont le valider, ce qui n'est pas le cas avec les signatures. Lors de la vérification d'une requête, le MAC est vérifié, puis, s'il est valide, alors la signature est également vérifiée. Dans le cas où le MAC est valide mais la signature est invalide, alors le client est considéré comme malicieux et plus aucune de ses requêtes ne sera vérifiée. Étant donné que vérifier une signature est beaucoup plus coûteux que vérifier un MAC, authentifier une requête avec ces deux mécanismes et vérifier le MAC dans un premier temps permet au protocole de se protéger d'une attaque par déni de service dans lequel un client malicieux enverrait des requêtes dont la signature est invalide.

Deuxièmement, Aardvark sépare le trafic réseau des clients et des réplicas en utilisant plusieurs interfaces réseau (*Network Interface Controller ; NIC*). En particulier, chaque réplica possède $3f + 1$ interfaces réseau : une pour la communication avec

les clients, et une pour la communication avec chaque autre réplica. Cela garantit que le trafic réseau des clients ne ralentira pas le trafic des réplicas. De plus, cela permet d'isoler un réplica qui inonderait le réseau avec des messages invalides. Plus précisément, si un réplica envoie beaucoup plus de messages que les autres réplicas en moyenne, alors son interface réseau est coupée pendant un certain temps, durant lequel il ne peut plus pénaliser les performances du système et peut être réparé.

Troisièmement, Aardvark isole les ressources de calcul. Plus précisément, chaque réplica utilise plusieurs files de messages : une pour les clients, et une pour chaque autre réplica. Cela permet au besoin de donner la priorité aux messages des réplicas, nécessaires à l'avancement du protocole. De plus, les réplicas utilisent la technique du tourniquet pour lire dans ces files de messages, ce qui permet de traiter les messages des différents réplicas de manière équitable, et de ne lire un message invalide d'un réplica que tous les $3f + 1$ messages. Aardvark tire également parti du parallélisme inhérent des machines multicœurs actuelles, en exécutant deux processus légers : un en charge de la vérification des requêtes des clients uniquement, et un en charge du reste du protocole. Cela permet d'amortir le coût de la vérification des signatures.

Enfin, un principe important de la robustesse d'Aardvark est la présence de changements de vue réguliers. Les auteurs d'Aardvark affirment que ces changements de vue réguliers permettent de ne pas tomber sous la dictature d'un principal malicieux. Les changements de vue, et donc de principal, sont accomplis de la manière suivante (cf. Figure 2.8). Le protocole exige d'un principal qu'il fournisse un débit au moins

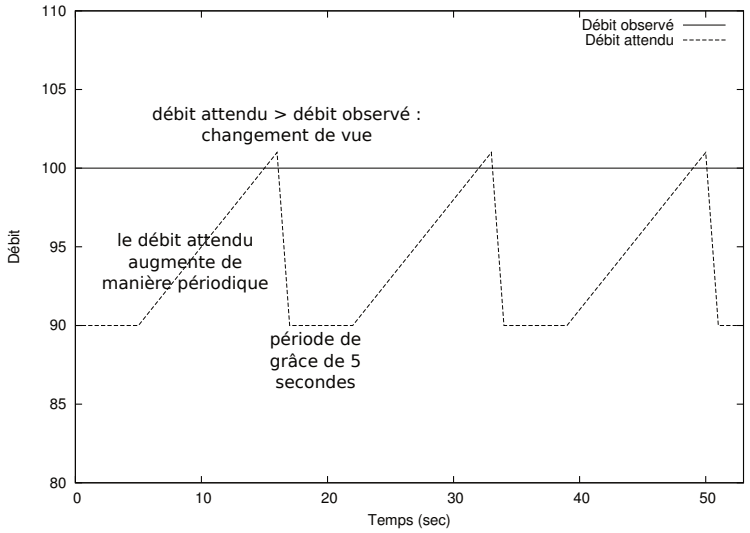


FIGURE 2.8 – Mécanisme de calcul du débit attendu et changements de vue dans Aardvark. Nous supposons que la charge est statique.

égal à 90% du débit maximal atteint lors des N vues précédentes (rappelons que N est le nombre total de réplicas). Après une période de grâce de 5 secondes qui suit un début de vue, et pendant laquelle le débit attendu est stable, les réplicas augmentent périodiquement le débit attendu par un facteur de 0,01, jusqu'à ce que le principal ne puisse plus fournir le débit attendu. Dans ce cas, un changement de vue est déclenché

et un nouveau réplica devient principal. Notons que ces changements de vue réguliers induisent un faible coût sur les performances du système : nous avons observé une perte de performances d'au plus 10% lorsque les changements de vue sont actifs.

En résumé, Aardvark est un protocole de TFB conçu pour être robuste via l'implantation de divers mécanismes : signature des requêtes, séparation des ressources de calcul et réseau, changements de vue réguliers. Ces mécanismes induisent un surcoût comparé à PBFT, à partir duquel Aardvark est conçu. De ce fait, le protocole Aardvark fournit des performances inférieures aux performances de PBFT.

2.2.4 Spinning

Le troisième protocole de TFB conçu pour être robuste à avoir été présenté est Spinning [56], en 2009. Ce protocole est basé sur PBFT et, de manière similaire à Aardvark, exécute des changements de vue réguliers. La particularité de Spinning est que ces changements de vue sont exécutés automatiquement après l'ordonnancement de chaque lot de requêtes, sans aucune communication entre les réplicas.

Dans ce protocole, les requêtes sont envoyées par les clients à tous les réplicas (cf. Figure 2.9). Dès qu'un réplica, différent du principal, reçoit une requête, il démarre un

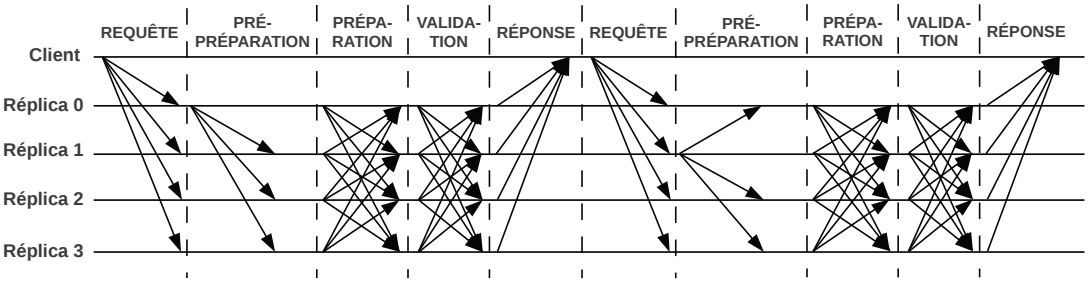


FIGURE 2.9 – Échange de messages dans Spinning ($f = 1$). Le réplica 0 est le principal pour la première requête, puis le réplica 1 devient automatiquement le principal pour la seconde requête, et ainsi de suite.

minuteur et attend un message d'ordonnancement (qui contient cette requête) de la part du principal. Si le minuteur expire, après une durée S_t, alors le principal courant est mis sur liste noire : il ne pourra plus devenir principal dans le futur [4], un autre réplica devient le principal, et la valeur de S_t est doublée. Les requêtes sont ordonnées comme dans PBFT, en 3 phases : PRÉ-PRÉPARATION, PRÉPARATION et VALIDATION. Enfin, les réplicas exécutent la requête, répondent au client, changent automatiquement de principal (c.à.d. sans échanger de messages), et la valeur de S_t est remise à sa valeur initiale. Notons que la valeur de S_t est un paramètre du système défini statiquement : il ne dépend pas de l'observation en temps-réel du comportement du système.

Lorsque le minuteur expire, un algorithme similaire à celui de changement de vue de PBFT se déclenche. En effet, étant donné que les réplicas ne sont pas synchronisés, le minuteur pourrait se déclencher sur certains réplicas mais pas sur d'autres, qui

4. Si f réplicas sont déjà sur liste noire, alors le plus ancien est enlevé de la liste, afin d'assurer la propriété de vivacité du système

pourraient recevoir le message d'ordonnancement avant la fin du minuteur. Dans ce cas, certains réplicas corrects exécuteraient la requête mais pas d'autres. Cette phase, appelée *fusion*, évite ce cas et garantit la propriété de sûreté du système lorsque le minuteur expire. En particulier, lors de cette phase, le nouveau principal sélectionne les prochaines requêtes à ordonner en se basant sur les requêtes qu'il a reçu mais qu'il n'a pas encore ordonné.

Notons également que, dans Spinning, les clients ne signent pas leurs requêtes : elles sont authentifiées avec un authentificateur MAC uniquement. Cela permet au protocole de fournir de bonnes performances. Toutefois, les MACs seuls ne garantissent pas la propriété de non-répudiation, comme nous l'avons vu précédemment. En particulier, Spinning est vulnérable à une attaque dans laquelle un client malicieux envoie une requête qui n'est valide que pour un sous-ensemble des réplicas. En effet, les réplicas, différents du principal et pour lesquels la requête est valide, déclenchent la phase de fusion. Si le client malicieux envoie de telles requêtes suffisamment souvent, alors le débit de Spinning peut devenir nul car le protocole passe la majorité de son exécution à déclencher et traiter des phases de fusion. Ceci étant, il est possible de modifier le protocole afin de forcer les clients à signer leurs requêtes, au détriment des performances.

Enfin, étant donné que le changement de vue est automatique et se déclenche après l'ordonnancement de chaque lot de requêtes, le réplica qui va devenir le prochain principal peut envoyer son prochain lot de requêtes en même temps que son message *validation*. Cette optimisation permet d'améliorer les performances de Spinning.

En résumé, Spinning est un protocole de TFB conçu pour être robuste dont la conception est relativement simple. En effet, contrairement aux protocoles précédents, les changements de vue sont automatiques, sans échange de messages, et se déclenchent après l'exécution de chaque requête. Quant aux performances, en utilisant exclusivement des MACs, elles sont meilleures que celles de Prime et Aardvark.

2.2.5 Protocoles de TFB sans principal

Comme nous avons vu précédemment, un problème des protocoles dits robustes est qu'ils ont besoin d'un réplica ayant un rôle spécial : le principal. Dans les protocoles conçus pour être robustes, Prime, Aardvark et Spinning, les réplicas surveillent les performances du principal afin de détecter un comportement malicieux. Afin de construire un protocole robuste, une idée qui vient à l'esprit est de construire un protocole sans principal. À notre connaissance, deux tels protocoles ont été proposés. Nous les présentons ci-après.

Correia et al. ont présenté en 2004 un nouveau protocole [59] qui ne nécessite pas de principal et qui ne nécessite que $2f + 1$ réplicas. De plus, ce protocole tolère un système asynchrone : il n'y a pas de borne sur les temps de traitements et de communications. Pour cela, ce protocole nécessite un composant spécial, la *base de calcul en temps borné sécurisée* (*Trusted Timely Computing Base* ; *TTCB*). Ce composant est distribué sur tous les réplicas et utilise pour ses communications un réseau secondaire dédié et synchrone. Grâce à ce réseau synchrone, il est possible de résoudre le résultat d'impossibilité FLP [47] : aucun protocole déterministe ne peut résoudre le problème du consensus dans un environnement asynchrone si un seul processus peut s'arrêter inopinément. De plus, ce composant est fiable, c'est-à-dire qu'il est résistant à n'importe quelle

attaque, et peut être défaillant seulement en s'arrêtant (c.à.d., il ne peut pas montrer un comportement arbitraire). Enfin, ce composant fournit un nombre limité d'opérations et borne les temps de calcul. Cela facilite sa vérification et lui permet d'être synchrone. Dans ce protocole, ce composant est utilisé pour émuler le rôle du principal : il fournit aux réplicas l'ordre dans lequel exécuter les requêtes. Si l'on fait un parallèle avec PBFT, nous pouvons dire que ce composant est l'équivalent du principal qui envoie un message PRÉ-PRÉPARATION, qui propose un ordre des requêtes. Après cette étape, les réplicas peuvent, dans un environnement asynchrone, ordonner et exécuter les requêtes. Bien qu'intéressante, cette approche nous semble peu réaliste. En effet, l'expérience montre qu'il est très difficile de concevoir des composants sûrs. Même s'il est parfois possible de vérifier de manière formelle la spécification des programmes (par exemple [60]), le matériel peut être sujet à des erreurs arbitraires (p. ex. inversion de bits, erreur disque, etc.) [1, 11, 61] qui invalident la preuve formelle.

Plus récemment, Boran et Schiper ont présenté un nouveau protocole de TFB sans principal [62] qui suppose un réseau ultimement synchrone et ne nécessite pas de composant spécial. Les auteurs affirment qu'étant donné qu'il existe des protocoles sans principal tolérant les arrêts inopinés, alors il doit être possible de construire des protocoles sans principal tolérant les fautes Byzantines. Ce protocole est basé sur le modèle des tours : chaque étape du protocole est divisée en un tour d'échange de messages. Chaque tour consiste en une étape d'envoi de messages, et une étape de transition vers le prochain tour, qui commence dès qu'un sous-ensemble des messages envoyés a été reçu. Une hypothèse du protocole est qu'après un certain tour inconnu, tous les messages envoyés par des réplicas corrects sont reçus par leurs destinataires dans le même tour. Pour cela, la durée d'un tour doit être suffisamment grande. De manière simplifiée, ce protocole combine deux protocoles : (i) un protocole sans principal qui fonctionne dans un environnement synchrone et qui assure que chaque réplica correct décidera ultimement une valeur et (ii) un protocole sans principal qui fonctionne dans un environnement asynchrone et qui assure que tous les réplicas corrects vont se mettre d'accord sur la même valeur. Ce protocole a un intérêt théorique, mais n'a pas d'intérêt pratique. En effet, le prix à payer pour ne pas utiliser de principal est que, avant d'ordonner chaque requête, les réplicas doivent s'assurer qu'ils vont recevoir un message de tous les autres réplicas corrects. Étant donné que les réplicas ne savent pas quels réplicas sont corrects, ils doivent attendre une certaine période (qui est étendue si elle n'est pas assez longue). Cela est source de mauvaises performances, ce qui pourrait expliquer pourquoi ce protocole n'a jamais été implanté.

2.2.6 Protocoles de tolérance aux intrusions

Les systèmes critiques (réseaux de télécommunications, distribution d'énergie, etc.) doivent pouvoir être disponible 24h/24 et 7j/7. Les protocoles tolérant les fautes Byzantines garantissent des propriétés de sûreté et de vivacité pour des systèmes répliqués aussi longtemps qu'au plus f réplicas parmi les N que composent le système sont malicieux simultanément. Toutefois, en pratique, il n'est pas possible de garantir qu'il y aura toujours moins de $f + 1$ fautes simultanées, d'autant plus pour des systèmes s'exécutant durant une période arbitraire, potentiellement longue.

Plusieurs protocoles ont étés conçu pour assurer la tolérance aux intrusions [63, 64, 65, 66, 67, 68, 69], c.à.d. que le système est capable de traiter (détecter, masquer

et récupérer) un large ensemble de fautes qui pourraient aboutir à une défaillance du système si aucune action n'est prise, qu'elles soient malicieuses ou non. Ces protocoles sont donc très proches des protocoles de TFB.

Les protocoles de tolérance aux intrusions se basent sur les trois principes suivants. Premièrement, les réplicas sont fréquemment rajeunis (p. ex. les clés de cryptographie sont changées, une version propre du système est réinstallée sur certains nœuds, etc). Cette remise à jour peut être périodique, auquel cas on parle de *récupération proactive*, ou en fonction des conditions du système, auquel cas on parle de *récupération réactive*. Dans ce dernier cas, les réplicas surveillent le comportement des autres réplicas. S'ils observent un comportement anormal de la part d'un réplica, alors ils le suspectent et exécutent un consensus afin de décider s'il faut le rajeunir, auquel cas le réplica sera réinstallé. Si exécuté suffisamment souvent, la remise à jour des nœuds rend la tâche plus difficile à un attaquant qui voudrait corrompre suffisamment de nœuds pour porter atteinte au système. Deuxièmement, comme dans le protocole de Correia et al. présenté dans la section précédente [59], chaque réplica doit exécuter un composant spécial fiable et insensible à n'importe quelle attaque, le TTCB. Ce composant est utilisé pour remettre un réplica en service dans un temps borné. Troisièmement, les TTCB doivent pouvoir communiquer entre eux dans un environnement synchrone. Pour cela, ils doivent posséder des canaux de communication fiables et indépendants des canaux asynchrones utilisés par les réplicas et les clients pour envoyer, recevoir et ordonner les requêtes.

Ces protocoles nécessitent au moins $3f + 2k + 1$ réplicas pour assurer les propriétés de vivacité et de sûreté, en présence de f fautes simultanées et de k réplicas qui peuvent être rajeunis en même temps. Une propriété clé est la définition d'une période de temps durant laquelle il n'y aura pas plus de f fautes simultanées. Cette période de temps doit être supérieure ou égale au temps nécessaire pour rajeunir un nœud. Dès lors, le système peut tolérer un nombre illimité de fautes sur toute la durée de son exécution.

Cette solution est complémentaire aux mécanismes de robustesse étudiés dans ce manuscrit et peut être appliquée aux travaux présentés dans cette thèse.

2.3 Résumé

Nous avons présenté dans ce chapitre différents protocoles de TFB existants. Ces protocoles peuvent être classés en deux catégories : les protocoles patrimoniaux, dont le but est de réduire le coût de la TFB et les protocoles dits robustes, dont le but est de fournir des garanties de performance en cas d'attaque.

La Table 4.1 présente une analyse des différents protocoles de TFB présentés dans ce chapitre. Pour chacun d'eux, nous montrons le nombre de réplicas nécessaires pour tolérer f fautes Byzantines simultanées, le nombre d'opérations (génération et vérification) de cryptographie (MAC et signatures) effectuées par le réplica qui fait office de goulot d'étranglement, et le nombre de phases d'échange de messages pour ordonner une requête. Nous présentons également les valeurs théoriques minimales [21,38,70,71].

Nous pouvons observer que tous les protocoles sauf Q/U requièrent le nombre minimal de réplicas. De plus, le nombre d'opérations de cryptographie des protocoles patrimoniaux est inférieur à celui des protocoles dits robustes. Ces derniers effectuent davantage d'opérations de cryptographie afin d'améliorer leur robustesse. Les protocoles

		Nb réplicas	Nb MACs	Nb signatures	Nb phases
	Borne théorique minimale	$3f+1$	2	0	2
Protocoles patrimoniaux	PBFT	$3f+1$	$2+\frac{8f}{b}$	0	5
	Q/U	$5f+1$	$2+4f$	0	2
	HQ	$3f+1$	$2+4f$	0	4
	Zyzzyva	$3f+1$	$2+\frac{3f}{b}$	0	3
	Aliph	$3f+1$	$1+\frac{f+1}{b}$	0	2
Protocoles dits robustes	Prime	$3f+1$	0	$2+\frac{11f+5}{b}$	7
	Aardvark	$3f+1$	$2+\frac{10f}{b}$	1	5
	Spinning	$3f+1$	$2+\frac{13f}{b}$	0	5

TABLE 2.1 – Analyse des différents protocoles de TFB présentés dans ce chapitre. Le nombre d'opérations de cryptographie effectuées prend en compte les générations et vérifications et est celui du réplica qui est goulot d'étranglement, pour des lots de requêtes de taille b. Le nombre de phases correspond au nombre de phases d'échanges de messages pour ordonner un lot de requêtes.

Prime et Aardvark nécessitent même des signatures, qui sont plus coûteuses à générer et vérifier que des MACs. Dans le cas d'Aliph, le nombre de génération et de vérification de MACs tend vers 1, qui est inférieur à la valeur théorique. De manière intuitive, cela peut s'expliquer par le fait que ce sont deux réplicas différents qui reçoivent la requête et répondent au client. Enfin, nous pouvons observer que la latence des protocoles patrimoniaux tend vers la borne minimale (et est atteinte par Q/U et Aliph), tandis qu'elle est au moins égale au plus lent des protocoles patrimoniaux dans le cas des dits protocoles robustes. Les protocoles dits robustes ne peuvent optimiser la latence d'ordonnancement des requêtes sans sacrifier une partie de leur robustesse. Notons que, dans le cas de Prime, le nombre de phases d'échanges de messages est de 7 lorsque le protocole est configuré de telle sorte qu'un message PRÉ-PRÉPARATION est envoyé plus fréquemment qu'un message MATRICE DE RÉSUMÉ.

Nous avons vu que les protocoles patrimoniaux sont, au fil des années, de plus en plus rapides. Ces protocoles optimisent le cas sans faute afin de réduire le coût de la réplication et, idéalement, de fournir des performances comparables aux performances d'un système non répliqué. Pour cela, ils réduisent le nombre d'opérations de cryptographie ainsi que le nombre de messages échangés afin d'exécuter une requête. Toutefois ces protocoles fournissent des performances dégradées en cas d'attaque. Cela peut être dû à un défaut inhérent à leur conception ou à un algorithme de traitement des fautes complexe. Par exemple, d'après les auteurs de Zyzzyva, certaines étapes du protocole pour garantir la propriété de vivacité du protocole sont suffisamment complexes pour être difficiles à implanter [13].

Nous avons présenté dans une seconde partie 3 protocoles conçus pour être robustes : Prime, Aardvark et Spinning. Le point clé de leur robustesse est une surveillance permanente du comportement du réplica principal. Cela leur permet de détecter un principal malicieux et d'en changer, sans impacter les performances du système de manière significative. D'autres mécanismes permettent également d'améliorer la robustesse, tel que l'isolation des traitements des messages et des communications réseau. Toutefois, ces protocoles nécessitent davantage de phases de communication que les protocoles patrimoniaux (p. ex. Prime nécessite 7 phases, contre 2 pour Q/U ou Aliph), génèrent et vérifient davantage de MACs (p. ex. Spinning exécute $\frac{5f}{b}$ opérations sur les MACs de plus que PBFT) ou utilisent des signatures qui sont plus coûteuses à générer et vérifier que les MACs.

Nous avons également présenté deux protocoles qui ne nécessitent pas de principal, et qui évitent donc les attaques qu'un principal pourrait exécuter afin de dégrader les performances du système. Toutefois, ces deux protocoles ont un intérêt théorique mais peu d'intérêt pratique étant donné leurs hypothèses : ils nécessitent un réseau secondaire synchrone et un composant fiable et résistant à n'importe quelle attaque.

Enfin, nous avons présenté une approche complémentaire afin d'assurer la robustesse d'un protocole de TFB : la tolérance aux intrusions. Dans cette approche, les réplicas sont remis à jour fréquemment. Cela rend difficile pour un attaquant de contrôler suffisamment de nœuds, durant suffisamment longtemps, pour dégrader les performances du système. Cette approche nous semble intéressante et peut être utilisée en parallèle par les travaux que nous présentons dans ce manuscrit.

2.4 Présentation générale des contributions de cette thèse

Dans cette thèse, nous nous intéressons à la conception de nouveaux protocoles de TFB efficaces et robustes. Plus précisément, nous souhaitons concevoir de nouveaux protocoles fournissant de très bonnes performances dans le cas sans faute tout en affichant une faible perte de performance, bornée, en cas d'attaque.

2.4.1 R-Aliph : un protocole de TFB conçu pour être efficace et robuste

Un premier problème que nous adressons dans cette thèse est la conception de protocoles à la fois efficaces et robustes. C'est-à-dire de protocoles fournissant de bonnes performances dans le cas sans faute comme en cas d'attaque.

Nous avons présenté dans ce chapitre une nouvelle approche, Abstract, permettant de simplifier la conception de protocoles de TFB. Les auteurs d'Abstract ont utilisé cette approche afin de concevoir Aliph, un nouveau protocole de TFB spéculatif fournissant de très bonnes performances dans le cas sans faute. En particulier, ce protocole est le plus performant des protocoles que nous avons présenté dans l'état de l'art. Toutefois, comme nous le montrons dans le Chapitre 3, Aliph n'est pas robuste : son débit peut chuter à zéro en cas d'attaque.

Nous souhaitons donc combiner les très bonnes performances d'Aliph dans le cas sans faute avec un protocole de TFB robuste qui fournit un débit non nul en cas d'attaque. Nous utilisons Abstract afin de concevoir R-Aliph. Tout comme Aliph, R-Aliph est la combinaison de deux protocoles spéculatifs, fournissant de très bonnes

performances dans le cas sans faute, Quorum et Chain, avec un troisième protocole patrimonial, Backup. Toutefois, contrairement à Aliph qui implante PBFT dans Backup et qui n'est pas robuste, R-Aliph implante Aardvark. Cela lui permet de résister aux mêmes attaques qu'Aardvark, tout en fournissant de meilleures performances que ce dernier en l'absence d'attaques. R-Aliph implante également trois autres principes, qui lui permettent de garantir (i) que son débit sera au moins égal au débit qu'aurait fournit Aardvark, (ii) que les réplicas seront équitables vis-à-vis des clients, et (iii) que les clients Byzantins ne peuvent impacter le mécanisme de basculement entre chaque protocole. R-Aliph réutilise certains mécanismes d'Aardvark qui lui permettent d'être robustes, tels que la présence de plusieurs interfaces réseau ou la séparation des ressources de calcul.

Nous évaluons R-Aliph dans le cas sans faute et en cas d'attaque. Nous montrons tout d'abord que le débit de R-Aliph est légèrement plus faible que le débit d'Aliph dans le cas sans faute. Ce surcoût d'au plus 6% est nécessaire afin de garantir la robustesse du protocole. Nous montrons également que le débit de R-Aliph est au moins égal au débit d'Aardvark en cas d'attaque. Enfin, nous montrons que le temps de basculement entre les instances d'Abstract n'est que faiblement impacté par des clients et réplicas Byzantins. En particulier, ce temps de basculement ne peut être augmenté par une quelconque attaque.

2.4.2 RBFT : un nouveau protocole de TFB plus robuste que les précédents

Un second problème que nous adressons dans cette thèse est la conception de protocoles réellement robustes. En effet, bien que conçus pour être robustes, les protocoles de TFB dits robustes n'offrent aucune garantie de performance en cas d'attaque. Pire encore, comme nous le montrons dans le Chapitre 4, les protocoles dits robustes présentent des défauts de conception majeurs qui permettent à un attaquant de faire baisser dramatiquement leurs performances, d'au moins 78%. Nous montrons que le problème de ces protocoles est que le réplica principal, qui propose un ordre d'exécution des requêtes, peut être malicieux de manière intelligente et réduire les performances jusqu'au seuil de détection. Il n'est jamais considéré comme malicieux, et donc jamais remplacé, mais fait tout de même décroître le débit du système de manière importante.

Nous souhaitons donc construire un nouveau protocole de TFB plus robustes que les précédents et qui garantisse une faible perte de performances même en cas d'attaque. Ce nouveau protocole, RBFT, repose sur le principe suivant : plusieurs protocoles dits robustes s'exécutent en parallèle. Les performances de ces différentes instances sont surveillées de près. Toutes les instances ordonnent les mêmes requêtes, mais seuls celles exécutées par une instance spéciale, appelée *instance maître*, sont exécutées. Les autres instances, appelées *instances de surveillance*, permettent de vérifier les performances de l'instance maître. En particulier, si le ratio entre le débit de l'instance maître et le débit des instances de surveillance est inférieur à une certaine valeur, alors le principal de l'instance maître est considéré comme malicieux et est remplacé par un autre réplica. Tout comme Aardvark, RBFT implante plusieurs mécanismes pour garantir sa robustesse contre toutes les attaques possibles. Par exemple, les différentes étapes du protocole sont exécutées en parallèle et de manière isolée et différentes interfaces réseau sont utilisées afin d'isoler les ressources réseaux. Enfin, RBFT tire

parti des architectures multicœurs, en exécutant les différentes étapes du protocole et les réplicas des différentes instances en parallèle, afin de fournir un débit acceptable dans le cas sans faute.

Nous évaluons les protocoles dits robustes et RBFT dans le cas sans faute et en cas d'attaque. Nous montrons que ses performances dans le cas sans faute sont comparables aux performances des protocoles dits robustes. Nous montrons également qu'en cas d'attaque, dans le pire cas possible, lorsqu'une proportion arbitraire de clients et f réplicas malicieux complotent, ses performances ne baissent que de 3%. Enfin, nous analysons ses performances de manière théorique et nous prouvons certaines propriétés du protocole en annexe A.3.

3

R-Aliph : un protocole de TFB conçu pour être efficace et robuste

Sommaire

Le protocole Aliph présenté dans la Section 2.1.5 fournit de très bonnes performances dans le cas sans faute, c.à.d lorsque le réseau est synchrone et lorsque les clients et les réplicas sont corrects. Toutefois, comme nous le montrons dans ce chapitre, un client ou un réplica Byzantin peut attaquer le protocole et réduire dramatiquement les performances. Cette fragilité des protocoles de TFB est bien connue et a motivé le développement de trois protocoles dits robustes : Prime [58], Aardvark [13] et Spinning [56]. Bien que plus robustes que les protocoles patrimoniaux, nous avons vu

dans le Chapitre 2 que ces protocoles sont moins performants. Nous présentons dans ce chapitre un nouveau protocole, R-Aliph, qui est une version robuste d'Aliph [1]. R-Aliph réunit le meilleur des deux mondes en utilisant Aliph, un protocole très efficace dans le cas sans faute et Aardvark, un protocole conçu pour être robuste. Nous évaluons R-Aliph dans le cas sans faute et en cas d'attaque. Nous montrons que ses performances sont proches des performances d'Aliph dans le cas sans faute et au moins égales aux performances d'Aardvark en cas d'attaque.

Dans la suite de ce chapitre, nous étudions tout d'abord les performances d'Aliph et des protocoles robustes en cas d'attaque (Section 3.1). Nous présentons ensuite R-Aliph (Section 3.2). Enfin, nous évaluons les performances de R-Aliph, dans le cas sans faute et en cas d'attaque (Section 3.3), avant de conclure (Section 3.4).

3.1 Motivations

Nous étudions dans cette section les performances d'Aliph, Prime, Aardvark et Spinning avec les quatre attaques suivantes, inspirées des attaques présentées dans [13]. Cette évaluation s'est faite sur les machines 8 cœurs présentées dans la section 3.3.2.

Déni de service par inondation par un client Dans cette attaque, l'un des clients est Byzantin et attaque les réplicas avec un déni de service, en envoyant de manière répétée des messages de 9ko à tous les réplicas.

Requêtes clientes mal formées Dans cette attaque, l'un des clients est Byzantin et envoie des requêtes contenant un authentificateur invalide, qui ne peut être vérifié que par un sous-ensemble des réplicas (incluant le principal dans Backup et la tête dans Chain).

Traitement supplémentaire Dans cette attaque, un réplica Byzantin (le principal dans Backup, la tête dans Chain et un réplica choisi de manière aléatoire dans Quorum) ajoute un temps de traitement supplémentaire de 10ms à tous les messages qu'il reçoit. Étant donné que les requêtes sont exécutées de manière séquentielle, cela impact le débit des protocoles. Notons que nous avons testé différentes valeurs de temps de traitement supplémentaire et avons observé des résultats cohérents.

Déni de service par inondation par un réplica Dans cette attaque, l'un des réplicas, choisi de manière aléatoire, est Byzantin. Il ne traite plus les messages du protocole et implante une attaque par déni de service en envoyant de manière répétée des messages de 9ko à tous les autres réplicas.

Nous montrons qu'Aliph, qui est un protocole spéculatif, fournit de très bonnes performances dans le cas sans faute, mais est énormément impacté durant une attaque. Au contraire, les protocoles robustes (Prime, Aardvark et Spinning), fournissent un débit plus faible dans le cas sans faute, mais restent peu impactés au cours des attaques.

3.1.1 Aliph en cas d'attaque

Dans cette section, nous étudions les performances d'Aliph dans le cas sans faute et avec les quatre attaques présentées ci-dessus. Les performances d'Aliph pour des

1. Ces travaux ont donné lieu à un article de journal en cours d'examination [72].

requêtes et des réponses nulles sont reportées dans la Table 3.1. Les résultats avec des tailles de requêtes et réponses différentes sont similaires.

	Sans attaque	Inondation par un client	Requêtes mal formées	Traitement supplémentaire	Inondation par un réplica
Aliph	55575	30733 (-44.7%)	0 (-100%)	98 (-99,8%)	0 (-100%)

TABLE 3.1 – Débit maximal (en req/s) d'Aliph dans le cas sans faute et en cas d'attaque, lorsque les requêtes et les réponses sont de taille nulle. Le traitement supplémentaire est de 10ms.

Nous pouvons observer que le débit d'Aliph chute à 0 lorsque des requêtes mal formées sont envoyées ou lorsqu'un réplica inonde les autres réplicas. Cela s'explique par le fait que, dans les deux cas, Quorum et Chain sont incapables d'ordonner les requêtes, ce qui induit un basculement vers Backup. Ce dernier se repose sur PBFT, qui est incapable de soutenir un débit non nul dans ces cas, comme déjà observé par [13].

Concernant l'attaque de traitement supplémentaire, le débit d'Aliph baisse à environ 98 requêtes par seconde. Ce résultat s'explique par le fait que le réplica malicieux ajoute un temps de traitement supplémentaire de 10ms avant de traiter chaque requête. Étant donné que le traitement des requêtes s'effectue séquentiellement, cela réduit le débit à un peu moins de environ 100 requêtes par second. La latence globale perçue par les clients est en dessous du seuil au-delà duquel les clients paniquent et basculent vers Backup. Par conséquent, le débit reste à cette valeur basse.

Enfin, concernant l'attaque du client qui inonde les réplicas, le débit d'Aliph décroît, mais de manière moins significative que dans le cas des autres attaques. Cela s'explique de la façon suivante. Lors de cette attaque, Quorumn'est pas capable d'ordonner les requêtes, ce qui induit un basculement vers Chain. Ce dernier utilise TCP, qui implante des mécanismes de gestion du débit et de partage équitable des ressources d'un lien réseau. Par conséquent, les clients corrects peuvent continuer à envoyer leurs requêtes à un débit élevé malgré le fait qu'un client malicieux essaye d'inonder le réseau.

3.1.2 Protocoles dits robustes en cas d'attaque

Nous présentons dans la Table 3.2 les performances de Prime, Aardvark et Spinning, lorsque les requêtes et les réponses ont une taille nulle, dans le cas sans faute et avec les quatre attaques décrites précédemment. Étant donné que Prime et Aardvark utilisent des signatures pour authentifier les requêtes plutôt que des authentificateurs MAC, nous avons implanté une version légèrement différente de l'attaque avec des requêtes mal formées : dans cette section, une requête malformée est une requête contenant une signature invalide.

Nous pouvons faire les observations suivantes. Tout d'abord, en l'absence d'attaque, les protocoles dits robustes sont moins efficaces que les protocoles spéculatifs (dans le cas présent, Chain. Le débit de Chain est 76% plus élevé que le débit d'Aardvark. Cette différence de performance entre les protocoles spéculatifs et les protocoles dits robustes est attendue et connue [13] : elle est due au surcoût des mécanismes que les protocoles dits robustes implantent afin de limiter l'impact des clients et réplicas Byzantins.

	Sans attaque	Inondation par un client	Requêtes mal formées	Traitement supplémentaire	Inondation par un réplica
Prime	14801	3201 (-78,4%)	14611 (-1,3%)	8253 (-44,2%)	0 (-100%)
Aardvark	31510	30280 (-3.9%)	31336 (-0,1%)	24749 (-21.5%)	28599 (-9,2%)
Spinning	41999	0 (-100%)	41505 (-1,2%)	11696 (-72,1%)	0 (-100%)

TABLE 3.2 – Débit maximal (en req/s) de Prime, Aardvark et Spinning dans le cas sans faute et en cas d'attaque, lorsque les requêtes et les réponses sont de taille nulle. Le traitement supplémentaire est de 10ms.

Deuxièmement, Prime et Spinning sont plus impactés qu'Aliph par l'attaque par inondation d'un client. Cela vient du fait qu'ils utilisent UDP, et, par conséquent, le client malicieux a un impact bien plus grand sur les autres clients et réplicas que Chain, qui utilise TCP.

Troisièmement, nous observons que, pour les autres attaques, les performances des trois protocoles dits robustes sont moins impactées par les attaques que celles d'Aliph. Néanmoins, il y a des différences significatives entre les performances des trois protocoles dits robustes : sous certaines attaques, Prime et Spinning peuvent souffrir de pertes de performances importantes. Par exemple, lorsqu'un réplica inonde les autres réplicas, le débit de Prime et Spinning chute à 0. Cela vient du fait que, dans ces deux protocoles, tous les réplicas partagent le même réseau. Cela les rend inefficaces face à une attaque par déni de service par inondation.

Au contraire, Aardvark est moins impacté par les attaques. Dans le pire des cas, la perte de performance est de 21,5% (lorsqu'un principal malicieux ajoute un temps de traitement à chaque requête). La raison pour laquelle Aardvark fournit de meilleures performances que Prime et Spinning en cas d'attaque est qu'il combine une surveillance précise de la progression du principal avec une isolation des communications réseau entre chaque paire de réplicas, tandis que Prime et Spinning ne font que surveiller la progression du principal.

3.1.3 Résumé

La conclusion que nous pouvons tirer des deux sections précédentes est que les concepteurs de systèmes distribués ont de toute évidence le choix entre (i) un protocole de TFB qui atteint de très bonnes performances dans le cas sans faute, mais qui fournit un débit très bas ou nul en cas d'attaque (p. ex. Aliph), ou (ii) un protocole de TFB qui fournit un débit plus faible lorsqu'il n'y a pas d'attaque (Aardvark est par exemple environ 43% moins efficace qu'Aliph pour des requêtes de taille nulle), mais qui est peu impacté en cas d'attaque.

3.2 R-Aliph : une version robuste d'Aliph

Dans cette section, nous montrons comment, en utilisant Abstract [49] (présenté dans la Section 2.1.5.1), nous avons conçu R-Aliph, un protocole qui atteint presque le meilleur des deux mondes. R-Aliph est à la fois aussi efficace qu'Aliph dans le cas

sans faute, et aussi robuste qu'Aardvark en cas d'attaque. Afin d'atteindre ce but, nous avons construit R-Aliph en suivant quatre principes :

Principe P1 Backup est implanté au-dessus d'Aardvark (et est ainsi résistant aux attaques).

Principe P2 Quorum et Chain ne sont exécutés que s'ils fournissent un débit meilleur que celui que Backup (c.à.d. Aardvark) aurait fourni.

Principe P3 Quorum et Chain ne sont exécutés que s'ils sont équitables vis-à-vis des clients.

Principe P4 Le temps requis afin de basculer entre ces protocoles n'est impacté ni par la présence de clients Byzantins, ni par la présence de réplicas Byzantins.

Il est trivial d'appliquer le **Principe P1**. Nous décrivons ci-dessous comment nous avons appliqué les trois autres principes.

3.2.1 Principe P2 : Le débit fourni doit toujours être au moins égal au débit du protocole conçu pour être robuste

Afin de s'assurer que R-Aliph exécute Quorum et Chain seulement s'ils atteignent un débit meilleur que celui que Backup aurait atteint (**Principe P2**), les réplicas qui exécutent Quorum et Chain surveillent le débit de ces protocoles périodiquement. Ils vérifient qu'il est supérieur à un débit attendu. Si un réplica détecte que ça n'est pas le cas, alors il devient *mécontent*, arrête de traiter les requêtes, et déclenche un basculement vers le protocole suivant (nous expliquons ce basculement plus en détail dans le **Principe P4**, dans la Section 3.2.3). Dans la suite de cette section, nous expliquons comment les réplicas de R-Aliph (i) définissent le débit attendu et (ii) surveillent le débit courant.

3.2.1.1 Définition du débit attendu

Pour régler le débit attendu, R-Aliph tire parti du mécanisme d'attente du débit fournit par Backup (c.à.d. Aardvark) lorsqu'il s'exécute. Plus précisément, lorsque R-Aliph bascule à Quorum ou Chain, le débit auquel chaque réplica s'attend est le débit maximum observé sur tous les débits attendus calculés lors de l'exécution de Backup. Par conséquent, le débit que doivent fournir Quorum et Chain doit être supérieur ou égal au débit maximal attendu lors de l'exécution de Backup. Dit autrement, les protocoles spéculatifs doivent fournir un débit supérieur au débit auquel le protocole conçu pour être robuste s'attendrait.

3.2.1.2 Surveillance du débit courant

Lors de l'exécution de Quorum ou Chain, un réplica ne peut pas calculer le débit de manière précise basé sur ce qu'il observe localement : il ne peut pas se baser sur les requêtes qu'il ordonne ou sur les messages de point de contrôle qu'il reçoit de la part des autres réplicas. Par exemple, un réplica Byzantin dans Quorum peut envoyer un message de point de contrôle prétendant qu'il a exécuté une requête, mais retarder l'envoie de la réponse au client. Cela a pour conséquence de réduire le débit global sans que le réplica Byzantin ne soit repéré par le client (p. ex. si le client reçoit la requête

avant le seuil après lequel il panique et bascule vers Backup, comme lors de l'attaque où le réplica Byzantin ajoute un traitement supplémentaire décrite dans la Section 3.1).

Par conséquent, la seule manière pour les réplicas de mesurer le débit de manière précise est de demander aux clients d'envoyer aux réplicas des notifications qui confirment qu'ils ont bien reçu une réponse pour les requêtes envoyées précédemment. Les notifications sont envoyées à tous les nœuds. Cela met davantage de pression sur les liens réseaux, ce qui peut impacter les performances du système. Nous montrons l'impact des notifications sur les performances dans la Section 3.3.3. Dans notre prototype, afin de limiter le surcoût des notifications, les clients ne les envoient que toutes les 5 réponses reçues, comme illustré dans la Figure 3.1.

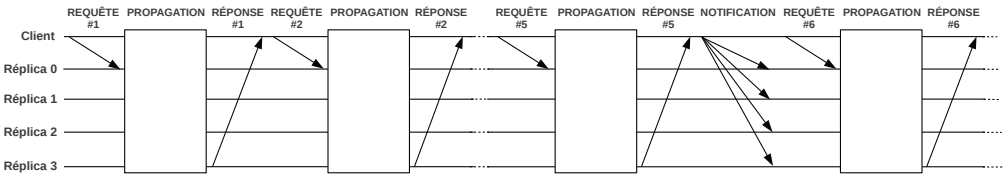


FIGURE 3.1 – Échange de messages avec un client dans Chain lorsqu'il envoie une notification de réponse toutes les 5 requêtes validées (c.à.d. toutes les 5 réponses) ($f = 1$).

Dans le cas de Quorum, ces messages sont intégrés dans les requêtes, ce qui permet d'éviter l'envoi de messages supplémentaires. Dans R-Aliph, Quorum et Chain calculent le débit de cette façon toutes les 128 requêtes validées (c.à.d. toutes les 128 réponses reçues). Si un réplica ne calcule pas le débit à temps, c.à.d. s'il n'a pas validé 128 requêtes alors que Backup l'aurait fait, alors le réplica devient *mécontent* immédiatement, arrête de traiter les requêtes suivantes, et initie un basculement de protocole.

3.2.2 Principe P3 : les réplicas doivent être équitables vis-à-vis des clients

Afin de s'assurer que R-Aliph exécute Quorum et Chain seulement lorsque les clients sont traités de manière équitable (**Principe P3**), les réplicas implantent le mécanisme suivant (inspiré d'Aardvark) : les réplicas suivent les requêtes des clients. Après avoir reçu une requête r ils vérifient qu'aucun client n'a envoyé de notifications pour deux requêtes reçues après r (ce qui peut être un signe comme quoi l'un des réplicas n'est pas équitable et retarde r). Si un réplica détecte que ça n'est pas le cas, alors il devient *mécontent*, arrête de traiter les requêtes et déclenche un basculement vers le protocole suivant (comme détaillé dans la section suivante).

Les réplicas de Quorum reçoivent toutes les requêtes directement depuis les clients, et peuvent donc les suivre de manière précise. Toutefois, ça n'est pas le cas de Chain : la tête peut transmettre les requêtes reçues dans un ordre arbitraire, ce qui empêche le suivi précis des requêtes. La seule manière de s'assurer que les réplicas suivent les requêtes de manière précise est que les clients envoient une notification à tous les réplicas à chaque fois qu'ils envoient une requête. Afin de limiter le surcoût de ces messages dans Chain, ils sont inclus dans les notifications de réponses (cf. Figure 3.1).

3.2.3 Principe P4 : les clients et réplicas Byzantins ne peuvent impacter le temps de basculement entre les protocoles

Afin de s'assurer que le temps nécessaire pour un basculement de protocole n'est pas impacté par des clients ou réplicas Byzantins, (**Principe P4**), R-Aliph se repose sur les trois idées suivantes :

1. La taille de l'historique des requêtes est bornée ;
2. les ressources CPU et réseau sont isolées ;
3. Les clients ne prennent pas part au protocole de basculement.

Nous détaillons chacun de ces points dans la suite de cette section.

3.2.3.1 La taille de l'historique des requêtes est bornée

La taille de l'état qui doit être transféré durant un basculement de protocole dépend de la taille de l'historique des requêtes, comme expliqué dans la Section 2.1.5.1. Afin de limiter la quantité de données à transférer, les réplicas bornent la taille de l'historique. Dans notre implantation, la borne est de 384 requêtes, ce qui ne limite pas les performances : une borne plus élevée n'augmente pas le débit maximal de R-Aliph. En effet, la différence maximale de débit entre un historique borné à 384 requêtes et sans borne, pour des requêtes de taille nulle, est de seulement 1,4%.

3.2.3.2 Les ressources CPU et réseau sont isolées

R-Aliph tire parti de l'isolation des ressources réseau et CPU qu'Aardvark (utilisé dans Backup) demande. Plus précisément, comme illustré dans la Figure 3.2 :

1. chaque réplica utilise une interface réseau (*Network Interface Controller*, NIC) dédiée à la communication avec les clients – notons que cette interface réseau est également utilisée dans Quorum et Chain pour l'échange des messages du protocole dans le cas normal ;
2. chaque réplica utilise un ensemble d'interfaces réseau dédiées pour la communication avec les autres réplicas. Il y a $3f$ telles interfaces par nœud : une pour chaque autre réplica ;
3. chaque réplica lit les messages dans ces différentes interfaces réseau de manière cyclique, en utilisant la méthode du tourniquet (*round-robin*). Cela permet ainsi de lire environ autant de message de la part de chacun des réplicas, ce qui par exemple limite une attaque de déni de service ;
4. chaque réplica peut désactiver une interface réseau dédiée à la communication avec un autre réplica si celui-ci envoie des messages invalides ou beaucoup plus de messages que les autres réplicas en moyenne.
5. la gestion des points de contrôle et du basculement de protocole s'effectue en parallèle du traitement des messages du protocole dans le cas normal de Quorum et Chain (module *Basculement robuste* dans la figure),

Cette combinaison de mécanisme permet d'avoir une implantation robuste des canaux de communication point-à-point entre chaque paire de réplicas : même si un client ou un réplica Byzantin attaque le réseau, cela n'empêchera pas les paires de

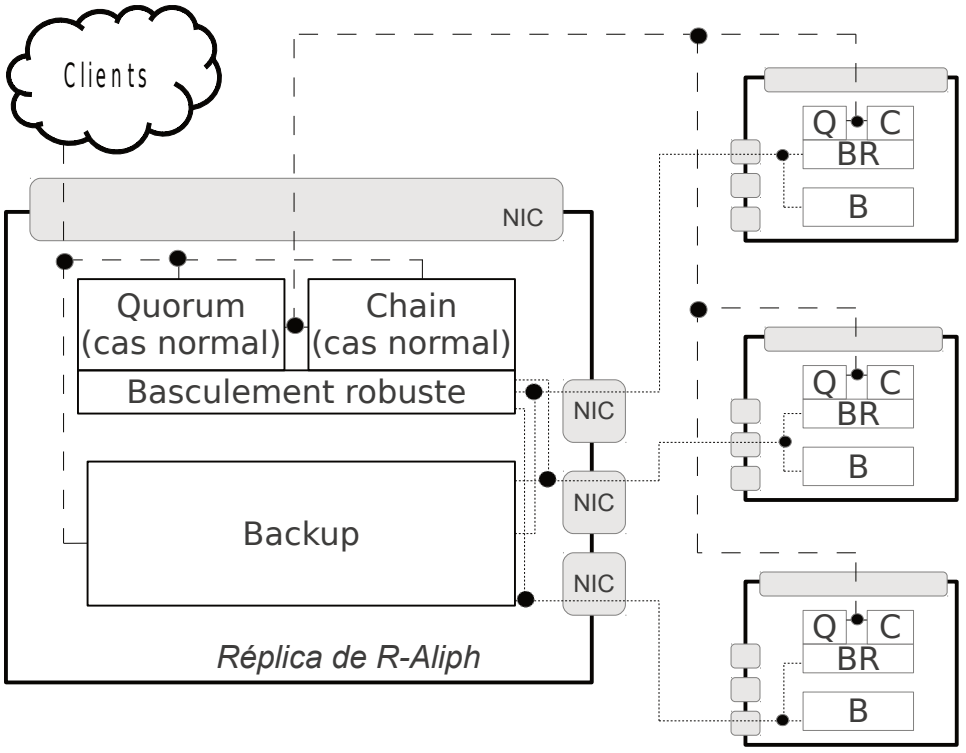


FIGURE 3.2 – Architecture d'un réplica de R-Aliph ($f = 1$).

réplicas corrects de continuer à communiquer de manière efficace. De plus, le traitement des point de contrôle et des messages de basculement en parallèle permet de ne pas retarder le basculement robuste, par exemple si des clients et réplicas malicieux envoient des messages qui mettent beaucoup de temps à être traités.

3.2.3.3 Les clients ne prennent pas part au protocole de basculement

Les clients ne prennent pas part au protocole de basculement (bien qu'ils puissent paniquer si besoin). Sans cela, étant donné qu'un réplica est connecté à tous les clients via une unique interface réseau, un client Byzantin pourrait retarder de manière arbitraire le protocole de basculement.

Un réplica souhaitant déclencher un basculement de protocole (parce qu'il détecte que Quorum ou Chain ne fournissent pas un débit adéquat, ou qu'ils ne sont pas équitables, ou si un client panique) se comporte comme un client : il invoque une requête nulle sur l'instance courante d'Abstract (c.à.d. Quorum ou Chain) et panique immédiatement. Le réplica complète alors sous-protocole de panique/avortement et bascule, comme s'il était un client, à la prochaine instance d'Abstract. Ensuite, étant donné que (i) la taille de l'historique est limité, (ii) les clients ne sont pas impliqués dans le protocole d'avortement et (iii) chaque pair de réplicas est connectée via une interface réseau dédiée, les clients et/ou réplicas Byzantins ne peuvent pas impacter le temps maximal nécessaire pour compléter un basculement de protocole.

3.3 Évaluation

Dans cette section nous évaluons R-Aliph. Nous présentons tout d'abord les objectifs de cette évaluation (Section 3.3.1) puis les paramètres expérimentaux (Section 3.3.2). Nous jugeons ensuite son surcoût par rapport à Aliph (Section 3.3.3). Puis nous étudions son comportement en cas d'attaque (Section 3.3.4). Enfin, nous évaluons le temps de basculement dans le pire des cas (Section 3.3.5).

3.3.1 Objectifs de l'évaluation

Nous souhaitons répondre à l'interrogation suivante : R-Aliph est-il un protocole de TFB robuste et efficace ? Nous considérons que c'est le cas s'il répond de manière positive à chacune des conditions suivantes :

1. Le mécanisme de basculement d'instance de R-Aliph doit préserver les propriétés d'exactitude des protocoles composés, c.à.d. Aliph et Aardvark ;
2. R-Aliph doit être efficace dans le cas sans fautes : son débit doit être au pire inférieur de 10% comparé au débit d'Aliph dans le cas sans faute ;
3. R-Aliph doit être robuste en cas d'attaque : il doit fournir des performances au moins égales aux performances fournies par le protocole robuste implanté dans Backup, c.à.d. Aardvark ;
4. R-Aliph doit pouvoir basculer vers Backup en cas d'attaque dans un temps court et borné.

La première condition est validée grâce aux propriétés de la composition d'instance dans Abstract. Comme montré par ses auteurs [73], combiner deux instances d'Abstract qui assurent des propriétés de sûreté et de vivacité retourne une nouvelle instance d'Abstract qui assure les mêmes propriétés de sûreté et vivacité. Nous montrons que les conditions suivantes sont respectées par R-Aliph dans la suite de cette section.

3.3.2 Paramètres expérimentaux

Nous avons évalué les performances de R-Aliph dans le cas sans faute et en cas d'attaque, de manière expérimentale. Cette évaluation s'est faite sur un cluster composé de huit Dell PowerEdge T610 et deux Dell Precision WorkStation T7400, connectées entre elles via un réseau Gigabit Ethernet local. Les T610 possèdent deux processeurs quad-cœurs Intel Xeon E5620 cadencés à 2,4GHz, avec 16Go de RAM et 10 cartes réseau Gigabit. Les T7400 possèdent deux processeurs quad-cœurs Intel Xeon E5410 cadencés à 2,33GHz, avec 8Go de RAM et cinq cartes réseau Gigabit. Toutes ces machines utilisent un noyau Linux version 2.6.32. Les nœuds s'exécutent toujours sur les T610 ; les machines restantes sont utilisées pour exécuter les clients. Nous avons lancés les expériences avec au plus une faute Byzantine tolérée, c.à.d. $f = 1$.

Dans notre implantation, seul Quorum communique via UDP et utilise son mécanisme de multidiffusion. Chain et Backup communiquent via TCP. De plus, Backup et le basculement robuste utilisent des interfaces réseau dédiées.

Nous avons soumis le protocole à une charge statique, lorsque le système arrive à saturation, afin de mesurer le débit maximal. De plus, les clients envoient leurs requêtes

en boucle fermée, comme définit dans [74], c.à.d. qu'ils attendent la réponse d'une requête avant d'en envoyer une nouvelle. Cette configuration s'accorde avec elles utilisées dans les autres papiers du domaine, p. ex. [13, 40, 49].

3.3.3 surcoût de R-Aliph

Le surcoût de R-Aliph est principalement dû aux messages de notification, envoyés par les clients afin d'informer les réplicas de Chain qu'ils ont envoyé des requêtes et reçu des réponses. Afin d'évaluer ce surcoût, nous avons lancé une expérience sans attaque et avons comparé le débit de R-Aliph au débit d'Aliph. La taille des requêtes varie d'une taille nulle à 10ko. Les réponses sont de taille nulle (nous avons observé des résultats similaires avec des réponses de taille supérieure). Les résultats sont reportés dans la Figure 3.3. Nous pouvons faire les deux observations suivantes. Premièrement,

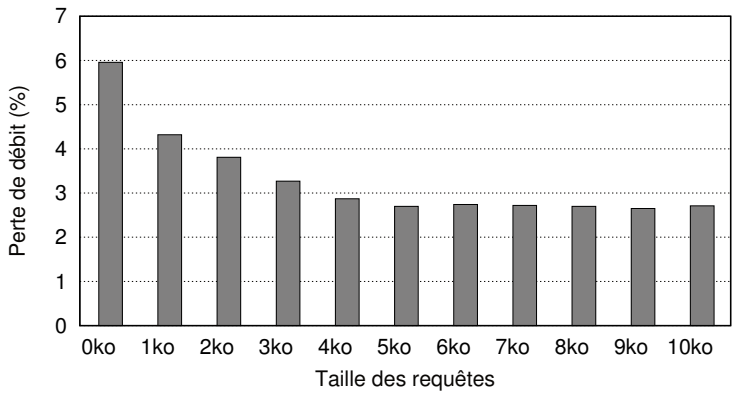


FIGURE 3.3 – Perte de performance de R-Aliph comparé à Aliph pour des requêtes de différentes tailles et des réponses nulles.

la perte de débit maximale est inférieure à 6%, ce qui est raisonnable : dans le cas normal, R-Aliph est toujours 65% plus efficace qu'Aardvark. Deuxièmement, nous observons que le surcoût décroît lorsque la taille des requêtes augmente. Par exemple, avec des requêtes de 4ko, il est inférieur à 3%. La raison pour laquelle le surcoût décroît lorsque la taille des requêtes augmente est dû au fait que la taille relative des messages de notification des clients devient plus petite.

Dans une seconde expérience, nous avons évalué le surcoût des messages de notification en fonction de leur fréquence pour des requêtes de différentes tailles (0, 4 et 10ko). Nous avons fait varier la fréquence de notification comme suit : une notification été envoyée tous les 1, 2, 5 ou 10 messages. Les résultats sont reportés dans la Figure 3.4. Nous pouvons observer que, quelque soit la taille des requêtes, la perte de performance décroît lorsque la fréquence de notification augmente. Par exemple, pour des requêtes de taille nulle, elle est de 40,5% lorsqu'une notification est envoyée après chaque réponse, contre seulement 6% lorsqu'une notification est envoyée toutes les 5 réponses. De plus, augmenter la période au-delà de toutes les 5 requêtes ne réduit pas énormément la perte de débit : pour des requêtes nulles, la différence entre la perte avec une notification toutes les 5 requêtes et avec une notification toutes les 10 requêtes est de seulement 1,5%.

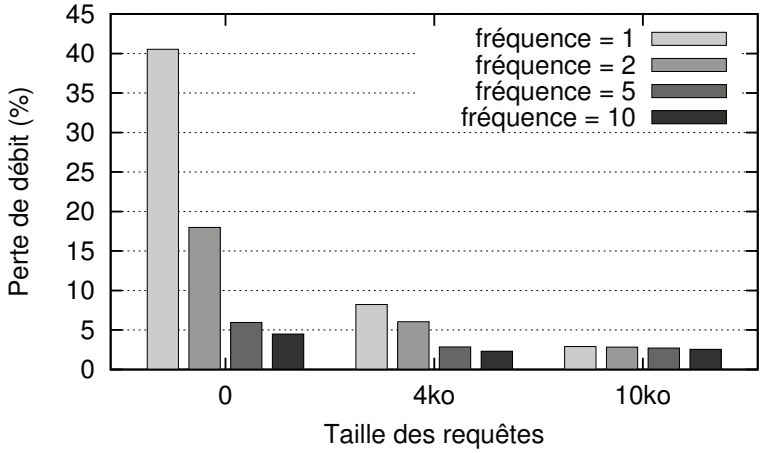


FIGURE 3.4 – Perte de performance de R-Aliph pour des requêtes de différentes tailles et des réponses nulles en fonction de la fréquence de notification.

3.3.4 Comportement de R-Aliph en cas d'attaque

Nous évaluons dans cette section les performances de R-Aliph durant les attaques décrites dans la Section 3.1. La figure 3.5 présente le comportement de R-Aliph durant l'attaque de traitement supplémentaire. Nous avons observé des résultats comparables avec les autres attaques. C'est pourquoi, afin d'éviter toute redondance, nous ne montrons pas les figures pour les autres attaques. Sur l'axe des X, nous reportons le temps (en secondes). Sur l'axe des Y, nous reportons le débit attendu par les réplicas (ligne pointillée) ainsi que le débit soutenu par R-Aliph (ligne solide). Les lignes verticales en pointillés représentent des basculements de protocole.

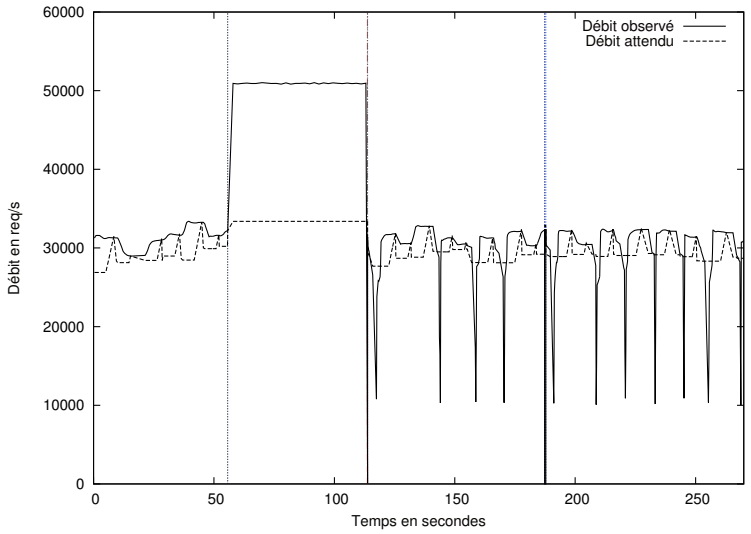


FIGURE 3.5 – Comportement de R-Aliph lors de l'attaque de traitement supplémentaire.

L'expérience se déroule de la manière suivante. Cent clients envoient des requêtes

de taille nulle en boucle fermée et reçoivent des réponses de taille nulle, durant toute la durée de l'expérience. Initialement, R-Aliph exécute Backup, configuré pour exécuter un nombre fixe de requêtes. Il n'y a pas d'attaque en cours et Backup soutient un débit d'environ $31,5\ kreq/s$, ce qui correspond au résultat observé précédemment, dans la Table 3.2. Au temps $55s$, R-Aliph bascule vers Quorum. Étant donné qu'il y a de la contention, ce dernier n'est pas capable d'ordonner des requêtes et panique immédiatement. C'est alors que R-Aliph bascule vers Chain. Ce dernier soutient un débit bien plus élevé que Backup (et donc un débit plus élevé que le débit auquel les réplicas s'attendent), aux alentours de $55,6\ kreq/s$. Au temps $114s$, l'attaque débute. Le réplica qui joue le rôle de la tête de Chain ajoute un temps de traitement supplémentaire de 10ms à tous les messages qu'il reçoit. Nous avons profilé le système et observé qu'un réplica correct commence à observer que Chain ne se comporte pas correctement et déclenche un basculement environ $7ms$ après le début de l'attaque. De même, le profilage du système nous a montré qu'il faut environ $63ms$ pour que Chain bascule vers Backup. Ce dernier exécute les requêtes malgré l'attaque. Son débit est légèrement impacté (-21% en moyenne). En effet, nous pouvons observer des baisses de performance sporadiques qui s'expliquent par le fait que le protocole utilisé dans Backup, Aardvark, change le principal régulièrement. Le principal élu est malicieux un quart du temps, et ajoute un temps de traitement de $10ms$. Cela induit de courtes pertes de performance, avant qu'un nouveau principal ne soit élu. Au temps $187s$, après que Backup ait exécuté un nombre fixe de requêtes, R-Aliph bascule vers Quorum. À nouveau, ce dernier n'est pas capable d'ordonner les requêtes (à cause de la contention). R-Aliph bascule ensuite vers Chain, qui fournit un débit inférieur au débit requis, à cause de l'attaque. Le profilage du système révèle qu'il faut environ $5ms$ pour qu'un réplica correct remarque que Chain est lent, et environ $20ms$ pour basculer vers Backup. Notons que le basculement à environ $187s$ est plus rapide comparé à celui à environ $114s$, lorsque l'attaque a débutée. Cela est dû au fait que l'historique des requêtes des réplicas contient bien moins de requêtes lors du second basculement.

3.3.5 Temps de basculement dans le pire des cas

Dans cette section nous évaluons le temps de basculement dans le pire des cas, dans le cas sans faute et avec chacune des attaques décrites dans la Section 3.1. Le temps que nous mesurons correspond au temps écoulé entre la création du premier message de panique et le temps lorsque les réplicas ont basculé au protocole suivant. Afin d'évaluer le temps de basculement, nous avons exécuté plusieurs basculements en boucle, avec la moitié des réplicas dont l'historique de requêtes était plein, et l'autre moitié pour laquelle il était vide. Ces paramètres induisent le plus grand transfert d'état possible entre les réplicas et sont identiques aux paramètres utilisés pour évaluer le coût du basculement dans [49].

Les résultats sont reportés dans la Table 3.3. Nous observons que le temps de basculement dans le pire des cas est bas (au plus $64ms$) et qu'il n'est impacté par les attaques que de façon marginale. Cela s'explique facilement par le fait que (i) le traitement des messages et les ressources réseaux nécessaires au basculement sont isolées et (ii) les clients ne sont pas impliqués dans le protocole de basculement. Ainsi, les réplicas corrects peuvent basculer vers le protocole suivant sans être impactés par la présence de clients ou réplicas Byzantins.

Sans attaque	Inondation par un client	Requêtes mal formées	Traitement supplémentaire	Inondation par un réplica
60.36	62.49	60.52	63.92	63.24

TABLE 3.3 – Temps de basculement (en ms) de R-Aliph dans le pire des cas, dans le cas sans faute et en cas d'attaque.

Dans le pire des cas, pour l'attaque de traitement supplémentaire et l'attaque d'inondation par un réplica, il augmente de $6,5\%$. Dans ces expériences, le réplica Byzantin et un réplica correct sont les seuls réplicas qui stockent les requêtes. Étant donné que le réplica Byzantin ne participe pas au protocole, il n'y a qu'un seul réplica qui enverra les requêtes manquantes, à deux réplicas, ce qui explique cette légère augmentation.

3.4 Conclusion

Dans ce chapitre, nous avons constaté que les concepteurs de systèmes distribués ont aujourd'hui le choix entre utiliser un protocole de TFB spéculatif, rapide mais qui fournit de très mauvaises performances en cas d'attaque, ou un protocole de TFB conçu pour être robuste, plus lent mais dont la perte de débit est inférieure en cas d'attaque. Nous avons alors montré comment utiliser Abstract pour combiner un protocole efficace mais peu robuste, Aliph, avec un protocole moins efficace mais conçu pour être robuste, Aardvark. Nous avons montré que R-Aliph fournit de très bonnes performances dans le cas sans faute : son débit est au plus 6% inférieur au débit d'Aliph. De plus, en cas d'attaque, R-Aliph peut basculer rapidement vers Backup, sans être impacté par les clients ou réplicas Byzantins. Une fois dans Backup, les performances de R-Aliph sont égales aux performances du protocole robuste utilisé dans Backup : Aardvark.

4

RBFT : un nouveau protocole de TFB plus robuste que les précédents

Sommaire

Nous avons présenté, dans le Chapitre 2, trois protocoles spécifiquement conçus pour être robustes et offrir des garanties de performance en cas d'attaque : Prime, Aardvark et Spinning. Ces protocoles fournissent en effet de meilleurs performances en cas d'attaque que les protocoles patrimoniaux. Cela vient principalement du fait que les réplicas de ces différents protocoles surveillent en permanence les performances du réplica principal. Cela leur permet de détecter un principal malicieux qui dégraderait les performances du système. Toutefois, nous montrons dans ce chapitre qu'ils ne sont en réalité pas robustes : ils peuvent souffrir d'une perte de performance importante, supérieure ou égale à 78%, en cas d'attaque (Section 4.1). Nous présentons dans ce chapitre un nouveau protocole, RBFT, qui adresse ce problème (Section 4.2) [1]. L'idée de base de RBFT est d'exécuter plusieurs instances d'un protocole conçu pour être robuste en parallèle et de surveiller leurs performances. Nous évaluons ce protocole de manière théorique (Section 4.3 et Annexe A.3) et expérimentale (Section 4.4), dans le cas sans faute et en cas d'attaque. Notre évaluation montre que le débit de RBFT dans le cas sans faute est comparable au débit des protocoles dits robustes que nous avons étudié, c.à.d. Prime, Aardvark et Spinning. De plus, RBFT est plus robuste en cas d'attaque. De manière expérimentale, la perte de débit maximale que nous avons pu observer est de seulement 3%. De manière théorique, nous montrons dans la Section 4.3 que le débit en cas d'attaque est un facteur constant près du débit dans le cas sans faute (les détails se trouvent en Annexe A.3).

4.1 Motivations

Nous présentons dans cette section une analyse des protocoles de TFB présentés comme robustes : Prime, Aardvark et Spinning. Nous montrons dans cette section que bien que conçus pour être robustes, ces protocoles ne le sont en fait pas. La Table 4.1 présente, pour chacun de ces protocoles, la perte maximale de performance en cas d'attaque. Nous pouvons observer qu'elle est au minimum de 78%, ce qui n'est pas acceptable. Dans la suite de cette section nous présentons, pour chacun de ces protocoles, la source de leur manque de robustesse ainsi que l'attaque dans laquelle la perte de performance est la plus importante.

	Prime	Aardvark	Spinning
Dégradation maximale du débit	78%	87%	99%

TABLE 4.1 – Dégradation des performances des protocoles dits robustes en cas d'attaque.

4.1.1 Prime en cas d'attaque

Dans Prime, les réplicas pré-ordonnent les requêtes et mesurent les performances du réseau. Cela leur permet de savoir quelles requêtes doivent être ordonnées et au

1. Ces travaux ont donné lieu à un rapport technique [75] et à une publication dans une conférence de rang international [76].

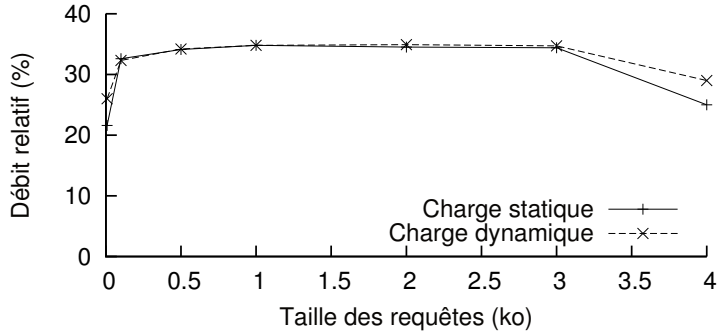


FIGURE 4.1 – Débit de Prime lors d'une attaque causé par un principal malicieux relatif au débit sans faute.

bout de combien de temps le principal doit envoyer un message d'ordonnancement. Plus précisément, les réplicas mesurent périodiquement le temps de réponse entre eux. Plus précisément, chaque réplica envoie périodiquement un message PING. Lorsqu'un réplica reçoit un tel message, il répond avec un message PONG. Le temps de réponse est alors le temps écoulé entre l'envoi du message PING et la réception du message PONG correspondant. Cette mesure est utilisée pour calculer le délai maximal qui doit séparer l'envoi de deux PRÉ-PRÉPARATION consécutifs lorsque le principal est correct. Ce délai est calculé en fonction de 3 paramètres : le temps de réponse entre les réplicas, le temps requis pour exécuter un lot de requêtes et une constante, définie par le développeur, qui permet de prendre en compte la variabilité de la latence du réseau. Si le principal devient plus lent que ce qui est attendu par les réplicas, alors il est remplacé.

Le protocole Prime n'est pas robuste car le mécanisme de surveillance peut être faussé. Le délai après lequel le principal est considéré comme malicieux s'il n'envoie pas de messages d'ordonnancement peut être arbitrairement long. Par exemple, un réplica peut ne pas répondre immédiatement à un message PING car il est occupé à une autre tâche, telle qu'exécuter un lot de requêtes. Cela donne l'opportunité à un principal malicieux de retarder les messages d'ordonnancement. Afin d'illustrer cela, nous avons lancé l'expérience suivante : un réplica malicieux complote avec un client malicieux. Le client malicieux envoie une requête qui est plus longue à s'exécuter que les requêtes des autres clients (10ms au lieu de 0,1ms). Étant donné que le temps d'envoi du prochain PRÉ-PRÉPARATION est supérieur au délai maximal qu'ont calculé les réplicas, cela provoque un changement de vue et le nouveau réplica, qui est malicieux, devient le principal. De plus cela augmente le délai maximal entre deux messages PRÉ-PRÉPARATION, ce qui permet au principal malicieux de retarder l'ordonnancement des requêtes des clients corrects. La Figure 4.1 montre le débit de Prime lors de cette attaque relatif au débit dans le cas sans attaque, avec une charge statique et une charge dynamique (les détails sur ces charges sont donnés dans la Section 4.4.2). Nous pouvons observer que le principal est capable de dégrader le débit jusqu'à 22% du débit sans faute. En d'autres termes, la perte de débit que peut causer un principal malicieux est d'au plus 78%.

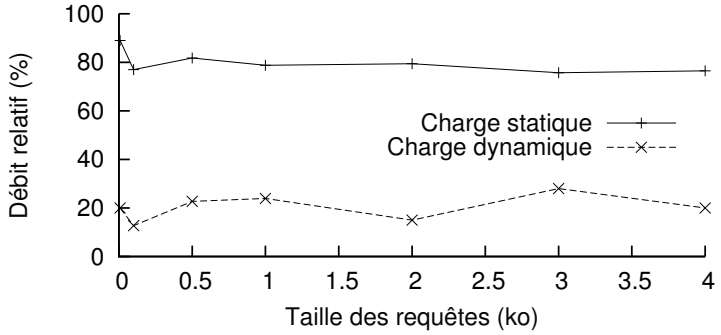


FIGURE 4.2 – Débit d'Aardvark lors d'une attaque causé par un principal malicieux relatif au débit sans faute.

4.1.2 Aardvark en cas d'attaque

Un principe important de robustesse implanté par Aardvark est la présence de changements réguliers de principal. Les auteurs affirment que changer régulièrement le principal permet de limiter la perte de débit qu'un principal malicieux peut causer. Nous rappelons qu'un principal dans Aardvark doit fournir au début de sa vue un débit au moins égal à 90% du débit maximal observé sur les N dernières vues. Le débit minimal qu'il doit fournir est augmenté périodiquement, d'un facteur de 0,01. Dès que le débit observé est inférieur au débit attendu, alors le principal est changé et un nouveau réplica est élu principal.

Aussi longtemps que le système reçoit une charge constante de requêtes, un principal malicieux ne peut ralentir le protocole Aardvark que de manière limitée. En effet, avec une charge statique, le débit auquel les réplicas s'attendent à ordonner des requêtes est proche du débit maximal que les clients peuvent fournir. Nous avons lancé une expérience sous une charge statique, dans laquelle un réplica donné, lorsqu'il est principal, ralentit les requêtes autant qu'il peut. Les résultats, représentés dans la Figure 4.2, montrent que le débit observé durant l'attaque est d'au moins 76% le débit observé dans le cas sans faute.

Toutefois, lorsque la charge est dynamique, la dégradation des performances peut être beaucoup plus importante. Plus précisément, lorsque la charge est basse, le débit attendu par les réplicas est également bas. Si, soudainement, la charge entrante augmente, un principal malicieux peut profiter du débit attendu bas afin de retarder les requêtes. Nous avons lancé des expériences qui illustrent ce cas, avec la charge dynamique décrite dans la Section 4.4.2. Les résultats, représentés dans la Figure 4.2, montrent que le débit peut baisser à 13% du débit qui aurait été observé si le réplica malicieux avait été détecté. En d'autres termes, la perte de débit que peut causer un principal malicieux est d'au plus 87%.

4.1.3 Spinning en cas d'attaque

La particularité de Spinning est que le principal est automatiquement changé après l'exécution de chaque lot de requêtes. De plus, les réplicas surveillent le principal et s'attendent à ce qu'il envoie un message d'ordonnancement au plus après un délai S_t.

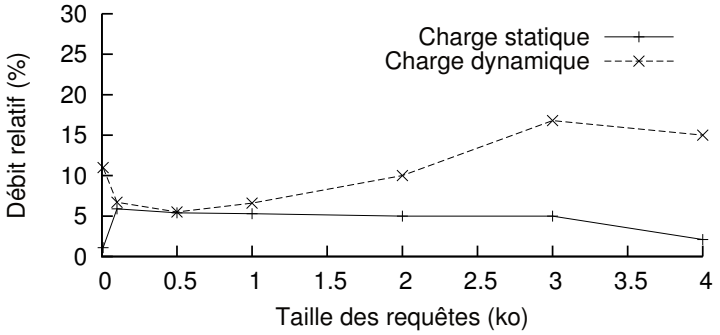


FIGURE 4.3 – Débit de Spinning lors d'une attaque causé par un principal malicieux relatif au débit sans faute.

Enfin, un réplica détecté comme malicieux est mis sur liste noire et ne pourra plus devenir principal dans l'avenir. Ainsi, un réplica malicieux qui retarde trop les messages d'ordonnancement n'impacte le système que pour un seul lot de requêtes.

Le protocole Spinning n'est pas robuste pour la raison suivante. Un principal malicieux peut retarder les requêtes par un peu moins que le délai après lequel un principal est considéré comme malicieux, S_t. Cela va lui permettre de réduire de manière drastique les performances, sans être détecté. Ainsi, il ne sera pas mis sur liste noire et va pouvoir continuer à retarder les futurs messages d'ordonnancement, la prochaine fois qu'il sera principal. Nous avons lancé plusieurs expériences où le principal malicieux retardait les messages d'ordonnancement de 40ms (cette valeur est la valeur de S_t utilisée par les auteurs de Spinning [56]), pour une charge statique et une charge dynamique. La Figure 4.3 montre le débit de Spinning lors de cette attaque de manière relative au débit dans le cas sans attaque, pour différentes tailles de requêtes, sous une charge statique et une charge dynamique. Nous pouvons observer que dans le pire des cas le débit chute à 1% du débit sans attaque (pour une charge statique). Cette dégradation de performances de 99% n'est clairement pas acceptable.

4.1.4 Résumé

Nous avons vu dans cette section que les protocoles de TFB dits robustes, c.à.d. Prime, Aardvark et Spinning, ne sont pas robustes en réalité. Un principal peut être malicieux de manière intelligente et peut causer une grande perte de performance sans être détecté. Spécifiquement, nous avons vu que Prime est robuste tant que le réseau fournit un certain niveau de synchronie. Si la variance du réseau est trop grande, alors un principal malicieux peut causer d'importants dommages sur les performances du système. De plus, Aardvark est robuste tant que la charge est statique. Dès que la charge est dynamique (par exemple si le système répliqué est un site Internet, dont le nombre de connexions varie de manière significative en fonction du temps), alors le protocole ne fournit plus de bonnes performances. Enfin, Spinning est robuste seulement pour $2f + 1$ sur $3f + 1$ requêtes, lorsque le principal courant est un réplica correct. Toutefois, lorsque le principal est malicieux, il peut retarder les messages d'ordonnancement jusqu'à la limite maximale autorisée et causer d'importants dégâts.

4.2 Le protocole RBFT

Nous avons vu dans la section précédente que les protocoles existants conçus pour être robustes ne le sont pas en réalité. Dans cette section, nous présentons *Redundant Byzantine Fault Tolerance* (*Tolérance aux Fautes Byzantines Redondante* ; abrégé RBFT), un nouveau protocole de TFB dont l'idée principale est la suivante : plusieurs instances d'un protocole de TFB conçu pour être robuste sont exécutées simultanément, chacune avec un principal qui s'exécute sur une machine différente. Les performances de ces différentes instances sont continuellement surveillées afin de détecter un principal malicieux.

Nous présentons tout d'abord une vue d'ensemble de RBFT (Section 4.2.1). Nous détaillons ensuite les différentes étapes du protocole (Section 4.2.2). Puis nous présentons le mécanisme de surveillance des performances (Section 4.2.3), qui est au cœur de la robustesse du protocole. Ensuite nous présentons le mécanisme de changement d'instance de protocole (Section 4.2.4), qui permet de remplacer un principal malicieux par un principal correct afin de garantir les performances maximales même en cas d'attaque, ainsi que le mécanisme de liste noire (Section 4.2.5), qui permet de se prémunir de certaines attaques par déni de service par inondation, et nous abordons la retransmission des messages (Section 4.2.6), nécessaire lorsque le réseau devient asynchrone, afin de garantir la propriété de longévité. Enfin, nous présentons notre implantation du protocole et certaines optimisations dans la Section 4.2.7.

4.2.1 Vue d'ensemble

Comme pour les autres protocoles robustes, RBFT nécessite $3f + 1$ nœuds (c.à.d. $3f + 1$ machines physiques). Chaque nœud exécute $f + 1$ instances d'un protocole de TFB en parallèle (cf. Figure 4.4). Comme nous le montrons de manière théorique en

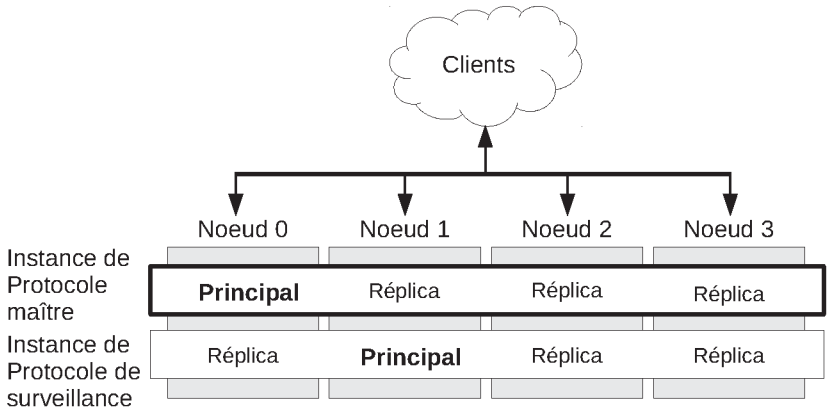


FIGURE 4.4 – Vue d'ensemble de RBFT ($f = 1$).

Annexe A.3, $f + 1$ instances de protocoles est nécessaire et suffisant afin de détecter un principal malicieux et de garantir la robustesse du protocole. Cela signifie que chacun des n nœuds dans le système exécute localement un réplica pour chaque instance de protocole. Notons que ces différentes instances ordonnent les requêtes suivant un

protocole de validation en trois phases, similaire à PBFT [40]. Les principaux des différentes instances sont placés sur les nœuds de façon telle que, à chaque instant, il y ait au plus un principal par nœud. L'une des $f + 1$ instances de protocole est appelée l'*instance maître*, tandis que les autres sont appelées *instances de surveillance*.

Toutes les instances ordonnent les requêtes des clients. Toutefois, seules les requêtes ordonnées par l'instance maître sont exécutées par les nœuds. Les instances de surveillance ne font qu'ordonner les requêtes afin de surveiller les performances de l'instance maître. Pour cela, chaque nœud exécute un module de surveillance des performances qui calcule le débit des $f + 1$ instances de protocole. Si $2f + 1$ nœuds observent que le ratio entre les performances de l'instance maître et les performances de la meilleure instance de surveillance est plus bas qu'un seuil prédéfini, alors le principal de l'instance maître est considéré comme défectueux et un nouveau principal est élu. De manière intuitive, cela signifie qu'une majorité de nœuds corrects se mettent d'accord sur le fait qu'un changement d'instance de protocole est nécessaire (les preuves complètes se trouvent en annexe A.3). Une alternative pourrait être de changer l'instance maître pour l'instance qui fournit le meilleur débit. Cela requiert un mécanisme de synchronisation de l'état des différentes instances lors du changement, similaire au mécanisme de commutation d'Abstract [49].

L'objectif à haut niveau de RBFT est le même que celui des protocoles dits robustes que nous avons étudié précédemment : les réplicas surveillent les performances du principal et déclenchent le mécanisme de récupération lorsque le principal est lent. Cependant l'approche que nous prenons est radicalement différente. Les réplicas ne peuvent pas deviner quel est le débit que devrait fournir un principal non malicieux. Par conséquent, l'idée clé est de tirer parti des architectures multicœurs afin d'exécuter plusieurs instances d'un même protocole en parallèle. Les nœuds comparent le débit fournit par les différentes instances afin de savoir si un changement d'instance de protocole est nécessaire ou non.

Pour que RBFT puisse fonctionner correctement, il est important que les $f + 1$ instances reçoivent les mêmes requêtes des clients. Dans cette optique, lorsqu'un nœud reçoit une requête provenant d'un client, il ne la transmet pas directement aux $f + 1$ réplicas qu'il exécute localement. Au lieu de cela, il envoie la requête aux autres nœuds. Lorsqu'un nœud a reçu $f + 1$ copies d'une requête donnée (potentiellement en incluant sa propre copie), il sait que tous les nœuds corrects vont à terme recevoir cette requête (car cette requête a été reçue par au moins un nœud correct ; cf. preuve en annexe A.3.4). Il transmet alors cette requête aux $f + 1$ réplicas qui s'exécutent localement.

RBFT est destiné à des systèmes en boucle ouverte (tels que les APIs asynchrones de Zookeeper [77], Boxwood [78] ou Chubby [79]), c.à.d. des systèmes dans lesquels un client peut envoyer plusieurs requêtes en parallèle, sans devoir attendre la réception des réponses des requêtes antérieures. En effet, dans un système en boucle fermée, la vitesse à laquelle les requêtes sont injectées par les clients dans le système serait conditionnée par la vitesse à laquelle l'instance maître ordonne les requêtes (qui est la seule instance dont les requêtes ordonnées sont exécutées). Dit autrement, les instances de surveillance ne seraient jamais plus rapides que l'instance maître.

Notons que chaque instance de protocole implante un protocole d'ordonnancement des requêtes très similaire à celui d'Aardvark. Il y a néanmoins une importante différence : une instance de protocole ne procède pas à un changement de vue d'elle-même. En effet, les changements de vue dans RBFT sont contrôlés par le mécanisme de

surveillance des performances et s'appliquent sur toutes les instances de protocole en même temps.

4.2.2 Étapes du protocole

Les étapes de RBFT sont décrites ci-après et représentées dans la Figure 4.5. La numérotation des étapes est la même que celle utilisée dans la figure.

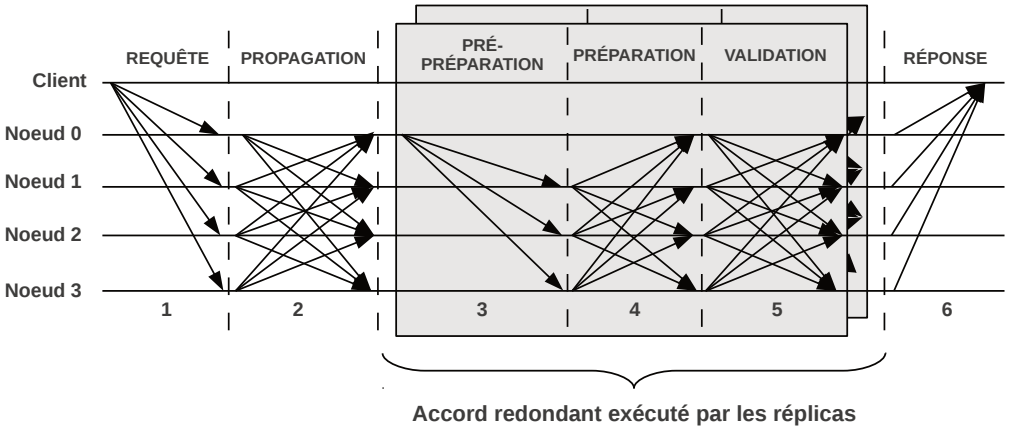


FIGURE 4.5 – Étapes du protocole RBFT ($f = 1$).

1. Le client envoie une requête à tous les nœuds.

Un client c envoie un message $\langle\langle \text{REQUÊTE}, o, rid, c\rangle_{\sigma_c}, c\rangle_{\vec{\mu}_c}$, de type REQUÊTE, à tous les nœuds (étape 1 de la figure). Ce message contient l'opération demandée o, l'identifiant de la requête rid et l'identifiant du client c. Ce message est signé avec la clé privée de c, puis authentifié avec un authentificateur MAC pour tous les nœuds. À la réception d'un message REQUÊTE, un nœud i vérifie l'authentificateur MAC. Si le MAC est valide, alors il vérifie la signature de la requête. Si la signature est invalide, alors le client est placé sur liste noire : les prochaines requêtes qu'il envoie ne seront pas traitées [2]. Si la requête a déjà été exécutée, alors i renvoie la réponse au client. Sinon, il passe à l'étape suivante. La requête est signée car elle doit être transmise entre les nœuds et utiliser un authentificateur MAC seul ne pourrait pas garantir la propriété de non-répudiation, comme décrit dans la Section 1.1.3.2. De manière similaire, utiliser des signatures seulement permettrait aux clients malicieux de surcharger les nœuds en envoyant des messages invalides avec de mauvaises signatures, qui sont plus couteuses à vérifier que des MACs.

2. Les nœuds corrects propagent la requête à tous les nœuds.

Dès que la requête a été vérifiée, le nœud i envoie un message $\langle \text{PROPAGATION}, \langle \text{REQUÊTE}, o, s, c\rangle_{\sigma_c}, i\rangle_{\vec{\mu}_i}$ à tous les nœuds. Cette étape permet de s'assurer que chaque

2. Les déviations exécutées par les clients ou les nœuds sur les protocoles de sécurité, par exemple un client qui initierait de manière continue un échange de clés avec un réplica, sont considérés comme hors de portée de cette thèse.

nœud correct va ultimement recevoir la requête, à condition que la requête ait été envoyée à au moins un nœud correct. Lorsque le nœud i reçoit un message PROPAGATION de la part du nœud j, il vérifie dans un premier temps l'authentificateur MAC. Si le MAC est valide, et si c'est la première fois qu'il reçoit cette requête, alors il vérifie sa signature. Si la signature est valide, i envoie un message PROPAGATION à tous les autres nœuds. Dès qu'un nœud a reçu $f + 1$ messages PROPAGATION pour une requête donnée, la requête est prête à être transmise aux $f + 1$ instances de protocole qui s'exécutent sur la même machine. Comme montré en annexe A.3, $f + 1$ PROPAGATION sont suffisants pour garantir que si un principal correct peut ordonner une requête, alors tous les principaux corrects vont à terme être capables d'ordonner cette même requête.

Afin de réduire le coût de l'ordonnancement, les réplicas n'ordonnent que les identifiants des requêtes, c.à.d. l'identifiant du client c, l'identifiant de la requête rid, et l'empreinte de la requête d. Non seulement la requête entière n'est pas nécessaire pour pouvoir l'ordonner (en particulier il est inutile de transmettre l'opération à exécuter étant donné que les instances de protocole n'exécutent pas les requêtes), mais en plus cela améliore les performances, car les réplicas ont moins de données à traiter et à transmettre.

À partir du début de cette phase, la requête ne contient plus l'authentificateur MAC, mais uniquement la signature. En effet, le MAC n'est plus nécessaire pour authentifier la requête. De plus, cela permet d'éviter un cas complexe à traiter lorsqu'un nœud traite un PROPAGATION avec une requête signée correctement mais avec un authentificateur MAC invalide pour un nœud donné. Un réplica ne peut faire la différence entre un client malicieux et un réplica émetteur malicieux. Tandis que si la requête ne contient que la signature, et si elle est invalide, alors seul le réplica qui a émis le PROPAGATION peut être blâmé, car il doit vérifier la requête dès sa réception depuis le client.

3. 4. et 5. Les réplicas de chaque instance de protocole ordonnent la requête.

Quand le principal d'une instance de protocole, p, reçoit une requête, il envoie un message de type PRÉ-PRÉPARATION, $\langle \text{PRÉ-PRÉPARATION}, v, n, c, rid, d \rangle_{\vec{\mu}_p}$, authentifié avec un authentificateur MAC pour chaque réplica de son instance de protocole (étape 3 de la figure). Les valeurs v (numéro de vue) et n (numéro de PRÉ-PRÉPARATION) identifient ce message de manière unique. Un réplica qui n'est pas le principal de son instance de protocole stocke la requête et attend un PRÉ-PRÉPARATION correspondant. Lorsqu'un réplica r reçoit un message PRÉ-PRÉPARATION venant du principal de son instance de protocole, il vérifie la validité du MAC. Il répond ensuite au PRÉ-PRÉPARATION avec un PRÉPARATION, $\langle \text{PRÉPARATION}, v, n, d, r \rangle_{\vec{\mu}_r}$, envoyé à tous les réplicas de son instance de protocole. L'envoi n'est effectué que si le nœud sur lequel s'exécute ce réplica a déjà reçu $f + 1$ copies de la requête. Sans cette vérification, un principal malicieux pourrait comploter avec des clients malicieux qui n'enverraient leurs requête qu'à lui. L'instance de protocole dont le principal est malicieux ordonnerait davantage de requêtes que les autres instances de protocoles, ce qui augmenterait ses performances au détriment des autres instances de protocoles. Après la réception de $2f$ messages PRÉPARATION pour le même PRÉ-PRÉPARATION et provenant de différents réplicas de la même instance de protocole, un réplica r envoie un message de type VALIDATION, $\langle \text{VALIDATION}, v, n, d, r \rangle_{\vec{\mu}_r}$, authentifié avec un authentificateur MAC (étape 5 dans la figure), à tous les réplicas. Enfin, après la réception de $2f + 1$

VALIDATION pour le même PRÉ-PRÉPARATION et provenant de différents réplicas de la même instance de protocole, un réplica envoie la requête, maintenant ordonnée, au nœud sur lequel il s'exécute.

6. Les nœuds exécutent la requête ordonnée et répondent au client.

À chaque fois qu'un nœud reçoit une requête ordonnée par un réplica r, la requête est prise en compte par le mécanisme de surveillance des performances. Si r est un réplica d'une instance de surveillance, alors la requête est jetée. Sinon, c.à.d. si r est un réplica de l'instance maître, alors l'opération demandée par le client est exécutée. Après l'exécution, le nœud i répond au client c en lui envoyant un message de type RÉPONSE (étape 6 de la figure) : $\langle \text{RÉPONSE}, u, rid, i \rangle_{\mu_{i,c}}$. Ce message, qui contient le résultat u de l'opération exécutée, est authentifié avec un MAC à destination du client. Lorsque le client c reçoit $f + 1$ RÉPONSES $\langle \text{RÉPONSE}, u, rid, i \rangle_{\mu_{i,c}}$ valides, avec des réponses u identiques, pour une même requête rid et de la part de différents nœuds i, alors il accepte u comme résultat de l'exécution de la requête rid.

4.2.3 Mécanisme de surveillance des performances

RBFT implante un mécanisme de surveillance des performances qui permet de détecter si le principal de l'instance de protocole maître est défectueux ou non. Ce mécanisme permet de s'assurer que les performances de l'instance de protocole maître correspondent bien à ce qu'elles devraient être dans le cas sans faute. Chaque nœud possède un compteur $nbreqs_p$, pour chaque instance de protocole p, qui correspond au nombre de requêtes qui ont été ordonnées par le réplica de l'instance p (c.à.d. qui a collecté $2f + 1$ messages VALIDATION). De manière périodique, le nœud utilise ces compteurs pour calculer le débit de chaque réplica qui s'exécute sur la même machine, et considère ce débit comme le débit de l'instance de protocole de ce réplica. La période de surveillance des performances est un paramètre de configuration. Plus elle est petite et plus les différences de performance sont visibles, tout comme le nombre de faux positifs. Plus elle est grande et moins les différences de performance sont visibles, tout comme le nombre de faux positifs. Autrement dit, si la période de surveillance est petite, alors de légères variations des performances des différentes instances de protocoles peuvent amener un réplica à suspecter un principal qui est en réalité correct. Ces variations de performance peuvent être dues par exemple à un lien réseau temporairement surchargé ou à un temps de calcul légèrement plus long en présence de contention entre un processeur et sa mémoire vive associée. Nous évaluons la précision du mécanisme en fonction de la période de surveillance dans la Section 4.4.4. Les débits sont ensuite comparés : chaque réplica calcule le ratio r entre le débit de l'instance maître t_{matre} et le débit maximal des instances de surveillance $t_{surveillance}$: $r = \frac{t_{maitre} - t_{surveillance}}{t_{maitre}}$. Notons que ce ratio est négatif lorsque le débit de l'instance maître est inférieur au débit de la meilleure instance de surveillance, auquel cas le principal de l'instance maître est suspecté d'être malicieux, et positif dans le cas contraire [3]. Si ce ratio est inférieur à un seuil Δ, alors le principal de l'instance maître est suspecté d'être malicieux et le nœud initie un changement d'instance de protocole, comme détaillé dans la section suivante.

3. Il est tout à fait possible de considérer la valeur absolue de cette valeur. En effet, le débit de l'instance maître ne peut être supérieur au débit de la meilleure des instances de surveillance

La valeur de Δ dépend du ratio entre le débit observé dans le cas sans faute et le débit observé durant une attaque. Elle est calculée de manière théorique en annexe A.3.

En plus de surveiller le débit, le mécanisme de surveillance des performances suit également le temps requis par les réplicas pour ordonner les requêtes. Ce mécanisme permet de s'assurer que le principal de l'instance maître est équitable entre les différents clients et ne privilégie pas certains clients au détriment d'autres. Plus précisément, chaque nœud mesure, pour chaque réplica p s'exécutant sur le même nœud, la latence de chaque requête req (notée $lat_{p,req}$) et la latence moyenne pour chaque client c (notée $lat_{p,c}$). Deux paramètres de configuration permettent de définir les latences maximales acceptables :

- Λ définit la latence maximale acceptable pour n'importe quelle requête.
- Ω définit la différence maximale acceptable entre les latences moyennes d'un client sur les différentes instances de protocoles.

Ces deux paramètres dépendent de la charge d'entrée du système et des paramètres expérimentaux (principalement la latence du réseau et le coût des opérations de cryptographie). Lorsqu'un nœud envoie une requête req aux réplicas qui s'exécutent sur la même machine afin de l'ordonner, il enregistre la date courante. Ensuite, lorsqu'il reçoit la requête ordonnée par le réplica de l'instance de protocole p, il calcule sa latence $lat_{p,req}$, ainsi que la latence moyenne pour toutes les requêtes de ce client $lat_{p,c}$. Si la latence pour cette requête est supérieure à Λ pour le réplica de l'instance maître et que ça n'est pas le cas pour les f réplicas des instances de surveillance, alors le nœud initie un changement d'instance de protocole. En effet, une requête pourrait exceptionnellement mettre davantage de temps à s'ordonner. Toutefois, si cela n'est pas dû à un comportement malicieux, alors le temps sera supérieur pour toutes les instances de protocoles. De manière similaire, si la différence des latences moyennes entre la latence moyenne pour le réplica de l'instance maître $lat_{m,c}$ et les latences des réplicas des instances de surveillance $lat_{p,c}$ est supérieure à Ω, alors le nœud initie un changement d'instance de protocole.

4.2.4 Mécanisme de changement d'instance de protocole

Dans cette section nous décrivons le mécanisme de changement d'instance de protocole utilisé pour remplacer un principal malicieux au niveau de l'instance maître. Étant donné qu'il y a au plus un principal par nœud dans RBFT, cela implique également de remplacer tous les principaux de toutes les instances de protocole.

Chaque nœud i possède un compteur cpi_i qui identifie de manière unique un message de changement d'instance de protocole (ce compteur est équivalent au numéro de vue des protocoles de TFB). Lorsque le nœud détecte un problème de performance, c.à.d. une trop grande différence entre les performances de l'instance maître et des instances de surveillance ou un problème d'équité dans la manière de traiter les requêtes des clients, comme détaillé dans la section précédente, alors il initie un changement d'instance de protocole. Pour cela, il envoie un message $\langle \text{CHANGEMENT\_INSTANCE}, cpi_i, i \rangle_{\vec{\mu}_i}$, contenant un authentificateur MAC, à tous les autres nœuds.

Lorsqu'un nœud j reçoit un message CHANGEMENT_INSTANCE de la part du nœud i, il vérifie le MAC puis traite le message de la manière suivante. Si $cpi_i < cpi_j$, alors le message était destiné à un changement d'instance précédent et est supprimé. Dans le cas contraire, si $cpi_i \geq cpi_j$, alors le nœud vérifie s'il doit également envoyer un

message CHANGEMENT_INSTANCE. Plus précisément, le nœud vérifie la différence de performance entre l'instance maître et les instances de surveillance et envoie un message CHANGEMENT_INSTANCE si besoin.

Après la réception de $2f + 1$ CHANGEMENT_INSTANCE valides et correspondants, le nœud i incrémente cpi_i et initie un changement de vue sur chaque réplica qui s'exécute localement, comme décrit dans la figure 4.6. Par conséquent, chaque instance de protocole élit un nouveau principal et le réplica malicieux n'est plus le principal [4].

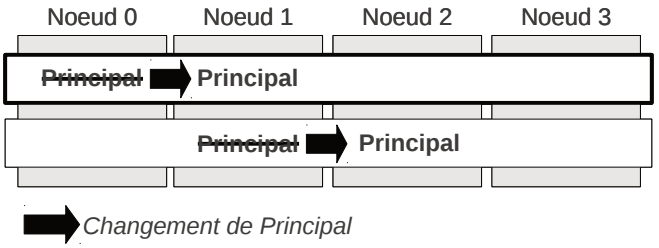


FIGURE 4.6 – Changement d'instance de protocole de RBFT : un changement de vue est initié sur toutes les instances de protocole en parallèle ($f = 1$).

4.2.5 Mécanisme de liste noire

RBFT isole un client ou un nœud malicieux qui tenterait d'attaquer le système par déni de service en utilisant un mécanisme de liste noire similaire à celui d'Aardvark. Plus spécifiquement, si un client envoie une requête mal formée contenant un MAC valide mais une signature invalide, alors il est mis sur liste noire. La raison est que la corruption du message ne peut pas briser le MAC sans briser la signature. De manière similaire, si un nœud envoie une majorité de messages invalides, beaucoup plus de messages que la moyenne des messages reçus depuis les autres nœuds, ou un message PROPAGATION valide mais dont la requête incluse contient une signature invalide (un nœud correct ne peut montrer un tel comportement : il aurait dû vérifier la requête et se rendre compte que sa signature était invalide), alors il est placé sur liste noire.

Lorsqu'un nœud est placé sur liste noire, alors est isolé du réseau grâce à l'utilisation de plusieurs interfaces réseau pour les communications entre les nœuds : chaque nœud communique avec chacun des autres nœuds via une interface réseau dédiée (cf. Section 4.2.7). Plus précisément, en cas de comportement malicieux, l'interface réseau qui permet de communiquer avec ce nœud est désactivée durant une certaine période (p. ex. 10 minutes, comme dans le cas d'Aardvark). Après cette période, il sort de la liste noire et le nœud traite à nouveau ses messages. Cela permet de réintroduire dans le système un réplica malicieux qui aurait été réparé.

Lorsqu'un client est placé sur liste noire, alors les messages qu'il envoie sont traitées à une fréquence plus faible que celle des messages des clients corrects (p. ex. un message d'un client sur liste noire est lu tous les 100 messages de clients corrects). Si, sur ses derniers m messages, le client sur liste noire envoie une majorité de messages

4. Dans le pire des cas f changements d'instance de protocole sont votés avant que le principal de l'instance maître ne soit correct.

corrects, alors il est considéré correct et sort de la liste noire. Cela permet de réintroduire dans le système un client invalide qui aurait été réparé.

4.2.6 Retransmission des messages

Afin d'assurer la progression de l'exécution du protocole (propriété de vivacité) dans un environnement *ultimement synchrone*, il peut être nécessaire de retransmettre des messages. Nous détaillons dans cette section quels sont les messages qui peuvent être retransmis.

Premièrement, les clients retransmettent périodiquement leurs requêtes tant qu'ils n'ont pas reçu de réponse. Si un nœud a déjà exécuté la requête, alors il renvoie la réponse au client. Tout comme dans PBFT, le nœud limite le taux de retransmission des réponses afin de se prémunir d'un client malicieux qui enverrait d'anciennes requêtes en boucle et empêcherait le nœud d'exécuter de nouvelles requêtes.

Deuxièmement, les messages de PROPAGATION sont périodiquement retransmis tant que la requête n'a pas été ordonnée. Cela permet de s'assurer qu'à terme, tous les nœuds corrects recevront les $f + 1$ messages de PROPAGATION nécessaires pour que la requête soit envoyée aux réplicas des instances de protocole et ordonnée. Il n'est pas possible de stopper la retransmission des messages PROPAGATION avant, par exemple lorsque la requête est envoyée aux réplicas. En effet, tant que la requête n'a pas été ordonnée un nœud ne peut pas savoir si les autres nœuds corrects ont reçu un quorum de PROPAGATION.

Troisièmement, il n'y a pas de retransmission des requêtes entre le nœud et ses réplicas, dans les deux sens. Cela n'est pas nécessaire car nous supposons que le mécanisme de communication utilisé pour les communications intra-nœud est fiable. De plus, la plus petite unité de faute dans notre modèle est la machine physique : soit tout le nœud est correct, soit tout le nœud est malicieux. En particulier, si un réplica est incorrect, alors toute la machine sur laquelle il s'exécute (c.à.d. les autres réplicas et le nœud) est considérée incorrecte.

Quatrièmement, les messages échangés entre les instances de protocole sont retransmis en utilisant le mécanisme de retransmission de PBFT : un réplica peut envoyer un message spécial demandant une retransmission d'une partie des messages qu'il n'a pas reçu.

Enfin, il n'y a pas de retransmission des messages de changement d'instance de protocole. Ils sont implicitement retransmis tant qu'un nœud observe une différence de performance trop importante entre les performances de l'instance de protocole maître et des instances de protocole de surveillance.

4.2.7 Implantation et optimisations

Nous avons implanté RBFT en C++, à partir de la base de code d'Aardvark. De manière similaire à Aardvark, RBFT utilise des interfaces réseau séparées (*Network Interface Controller*, NIC). Non seulement cela permet de séparer le trafic des clients et des réplicas et d'empêcher le trafic des clients d'interférer avec le trafic des réplicas, mais cela permet également de se prémunir d'une attaque par déni de service par inondation de la part des nœuds malicieux. Dans une telle situation, RBFT coupe

l'interface réseau du nœud malicieux pour une certaine période de temps qui permet au nœud malicieux d'être réparé, sans pénaliser les performances du système.

Le protocole de communication réseau utilisé par RBFT est TCP, comme pour notre implantation d'Aardvark. TCP permet de faciliter le développement : à l'inverse d'UDP, TCP fournit un canal de communication sans pertes de données de type *premier entré, premier sorti*. De plus, des travaux précédents (p. ex. Zyzzyva [55] ou PBFT [40]) ont montré que le goulot d'étranglement des protocoles de TFB n'est pas l'utilisation réseau mais les opérations de cryptographie. Il est intéressant de noter que le protocole de TFB qui fournit à l'heure actuelle le meilleur débit est Chain [49], présenté dans la Section 2.1.5.4. Ce protocole est basé sur TCP et fournit de très bonnes performances grâce à un nombre minimal d'opérations de cryptographie effectuées par les réplicas. À titre de comparaison, nous avons également implanté une version UDP de RBFT. Nous montrons dans la Section 4.4.3 que ces deux implantations fournissent les mêmes performances.

La Figure 4.7 décrit l'architecture d'un nœud dans RBFT, dans le cas où une faute est tolérée (c.à.d. $f = 1$). Nous pouvons observer que deux réplicas, appartenant à deux

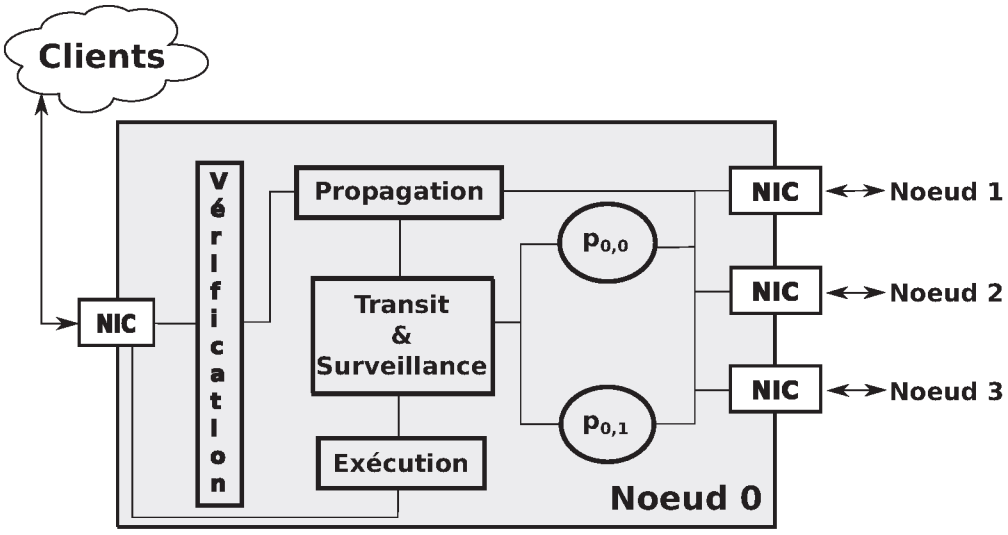


FIGURE 4.7 – Architecture d'un nœud ($f = 1$). Les réplicas (dans des cercles) sont des processus qui sont indépendants des différents modules du protocole, implantés sous forme de processus légers (dans des rectangles).

différentes instances de protocole, sont hébergés par le nœud : le réplica 0 de l'instance de protocole 0 (noté $p_{0,0}$ dans la figure) et le réplica 0 de l'instance de protocole 1 (noté $p_{0,1}$ dans la figure). Nous observons également que le nœud possède $3f + 1 = 4$ interfaces réseau, tout comme Aardvark : une interface dédiée à la communication avec les clients, et une interface par autre nœud.

Utiliser de nombreuses interfaces réseau sur une machine multicœurs peut poser un problème de passage à l'échelle. À chaque fois qu'un message est reçu depuis le réseau, il génère une interruption. Cette interruption est traitée par un cœur, qui ne peut pas effectuer d'autres tâches en parallèle. Lorsque le nombre d'interruption

est très élevé (soit parce que le trafic réseau est important, soit parce que beaucoup d'interfaces réseau sont utilisées), ce cœur devient surchargé, ce qui fait baisser les performances du processus qui s'y exécute. Pour empêcher ce problème, nous forçons l'affinité entre les interruptions des différentes cartes réseaux et les différents cœurs. Cela permet de balancer la charge des interruptions sur les différents cœurs de la machine. Cette technique a déjà été utilisée pour le passage à l'échelle de serveurs web sur machines multicœurs [80] ou pour construire un routeur logiciel fournissant un débit de 35Gb/s [81].

La figure 4.7 présente les différents modules (dans des rectangles) de RBFT qui sont exécutés sur chaque nœud. Nous décrivons le comportement de ces différents modules ci-dessous. Lorsqu'une requête est reçue depuis un client, elle est vérifiée par le module de *Vérification*. Si la requête est valide, alors le module de *Propagation* la dissémine aux autres nœuds via les interface réseau et attend des messages similaires de leur part. Une fois qu'il a reçu $f + 1$ messages de *Propagation*, le module envoie la requête au module de *Transit & Surveillance*. Ce module sauvegarde la requête et la date courante (utilisée comme date de réception afin d'évaluer les performances), et la donne aux réplicas qui s'exécutent localement (c.à.d. $p_{0,0}$ et $p_{0,1}$ dans la figure). Les réplicas interagissent avec les autres réplicas de la même instance de protocole, qui s'exécutent sur les autres nœuds, grâce aux interfaces réseau séparées. Une fois que la requête a été ordonnée par une instance de protocole, le réplica la renvoie au module de *Transit & Surveillance*, qui mesure les performances de l'instance de protocole, puis, si la requête ordonnée provient du réplica de l'instance maître, la transmet au module d'*Exécution*. Ce dernier exécute la requête avant de répondre au client.

Les modules de *Vérification*, *Propagation*, *Transit & Surveillance* et *Exécution* sont implantés sous forme de processus légers séparés. Les réplicas, quant à eux, sont implantés comme des processus indépendants. Les différents processus (légers ou non) sont déployées sur différents cœurs (nos machines de tests possèdent 8 cœurs). En tirant parti des machines multicœurs, nous pouvons améliorer les performances du système. En effet, les différents composants et réplicas peuvent s'exécuter de manière concurrente.

Dans ce chapitre nous avons présenté le fonctionnement du protocole lorsqu'une seule requête est ordonnée à la fois. Toutefois, afin d'améliorer les performances du système, RBFT supporte les lots de requêtes [82] pour les messages de type PROPAGATION et PRÉ-PRÉPARATION. Plus précisément, au lieu d'envoyer un tel message ne contenant qu'une seule requête, il est possible d'en envoyer un contenant plusieurs requêtes. Cela permet de réduire la pression sur le système (à la fois sur le processeur et le réseau) lorsqu'il est surchargé, pour des requêtes de petite taille. Par exemple, avec des requêtes de 8 octets et une charge constante, dans les conditions expérimentales décrites dans la Section 4.4.2, le débit maximal de RBFT est de 35kreq/s avec des lots de requêtes, alors qu'il est inférieur à 15kreq/s sans cette optimisation. Notons que cette optimisation est déjà utilisée par la plupart des protocoles de TFB (p. ex. PBFT, Zyzzyva, Prime, Aardvark, Spinning).

4.3 Étude analytique des performances

Nous fournissons dans cette section une étude analytique des performances des protocoles de TFB conçus pour être robustes, c.à.d. Prime, Aardvark et Spinning ainsi que RBFT. Notre étude analytique nous permet de comparer les performances de ces protocoles sans considérer leur implantation. Nous montrons que le débit calculé de manière théorique est proche du débit mesuré expérimentalement. Enfin, nous présentons le débit théorique de ces différents protocoles en cas d'attaque ainsi que la perte de performance maximale. En particulier nous montrons que cette perte est très importante pour les protocoles considérés comme robustes, mais inférieure à 1% dans le cas de RBFT. Le détail des formules théoriques est fourni en Annexe A.

4.3.1 Méthodologie

Notre analyse théorique consiste à calculer le débit, en nombre de requêtes par seconde, pour chacun des protocoles de TFB conçus pour être robustes considérés dans ce chapitre.

Notre analyse des protocoles de TFB conçus pour être robustes se base sur le coût des opérations de cryptographie [5]. Plus précisément, nous mesurons le coût de génération et vérification d'un MAC (α), le coût de vérification d'une signature (θ) et le coût de création d'une empreinte (δ). Nous prenons également en compte la taille des lots de requêtes, b. Notons que nous supposons que la taille des lots de requêtes est fixe et, dans le cas de RBFT, est la même pour les messages de PROPAGATION et les messages de PRÉ-PRÉPARATION. Enfin, nous ne prenons pas en compte le coût de la génération de la réponse envoyée au client. Ce coût est le même pour tous les protocoles excepté Prime, où une signature remplace le MAC calculé par Aardvark, Spinning et RBFT.

Nous considérons une machine multicœurs. Chaque cœur a une vitesse d'horloge de κ Hz. Dans le cas d'Aardvark, la vérification des requêtes et l'ordonnancement sont exécutés en parallèle. Dans le cas de RBFT, chaque réplica et chaque module (c.à.d. les modules de vérification, propagation, transit et surveillance et exécution) s'exécutent en parallèle sur des cœurs distincts. Nous ignorons le coût de l'exécution des requêtes car nous souhaitons mesurer le débit maximal (dans le cas de Prime, nous supposons que ce temps d'exécution est non-nul en cas d'attaque : la précision du mécanisme de surveillance des performances de Prime dépend du temps d'exécution des requêtes). De plus, nous ne prenons pas en compte le coût du traitement des points de contrôle et des changements de vue. Cela permet de simplifier la modélisation tout en gardant une bonne précision.

4.3.2 Performances dans le cas sans faute

Dans cette section nous présentons les débits théoriques des différents protocoles conçus pour être robustes dans le cas sans fautes. Nous analysons ces formules dans le but de comparer les performances de ces protocoles.

5. Les formules présentées en Annexe A prennent également en compte le coût des opérations de communication. Nous les avons omis dans cette section dans un souci de clarté ; le coût des opérations de communication est bien pris en compte dans nos calculs.

La Table 4.2 présente les formules de débit de chaque protocole. Nous pouvons

Prime :	$\frac{\kappa}{2\theta+((11f+5)\theta+(6f+2)\delta+P)/b}$
Aardvark :	$\frac{\kappa}{max(\theta+\alpha+\delta,(\alpha(10f+b)+\delta)/b)}$
Spinning :	$\frac{b\kappa}{\alpha(13f+2b)+\delta(b+1)}$
RBFT :	$\frac{\kappa}{max(\theta+\alpha(n+f)+\delta,\alpha(2n+4f+1)/b+\delta)}$

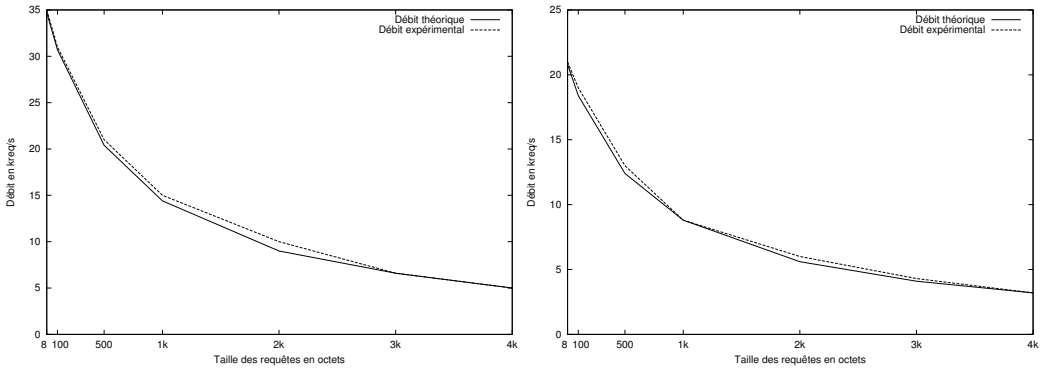
TABLE 4.2 – Débit de Prime, Aardvark, Spinning et RBFT dans le cas sans fautes pour ordonner une requête dans un lot de requêtes de taille b. Dans le cas de Prime, P est la période d'envoi des messages PRÉ-PRÉPARATION.

faire les observations suivantes. Premièrement, le débit de Prime est plus faible que le débit des autres protocoles, étant donné qu'il n'utilise que des signatures pour authentifier ses messages, qui sont par nature plus couteuses que des MACs. Par conséquent Prime fournit les moins bonnes performances des quatre protocoles étudiés. Deuxièmement, en parallélisant la vérification des signatures, Aardvark et RBFT peuvent en éliminer le coût supplémentaire. Leur coût revient donc au coût d'ordonnancement d'une requête. Troisièmement, nous remarquons qu'en terme d'opérations de cryptographie, en ne prenant en compte que la phase d'ordonnancement, Aardvark, Spinning et RBFT fournissent un débit équivalent : pour un lot de requêtes de taille b, ils génèrent et vérifient respectivement $10f + b$, $13f + 2b$ et $10f + 1$ MACs. Ceci est normal étant donné que ces trois protocoles ordonnent les requêtes globalement de la même manière, en suivant le protocole de validation en trois phases.

4.3.3 Précision des formules théoriques

Dans cette section nous souhaitons savoir si les formules théoriques donnent des résultats proches des résultats expérimentaux. Par souci de clarté nous prenons comme exemple RBFT seulement et montrons que son débit expérimental approche son débit théorique. Nous sommes arrivés à la même conclusion avec les autres protocoles (ceci n'est pas étonnant étant donné que les formules des autres protocoles ont été obtenues de la même manière que pour RBFT). Les conditions expérimentales sont celles décrites dans la Section 4.4.2.

Les figures 4.8a et 4.8b montrent le débit maximal du protocole dans le cas sans fautes en fonction de la taille des requêtes, respectivement lorsque $f = 1$ et $f = 2$. Comme nous pouvons le constater, le débit théorique est très proche du débit expérimental. En effet, la différence est d'au plus 10%, pour $f = 1$ et des requêtes de 2ko, tandis que la différence moyenne est de 2,8%. Nous pouvons donc valider les formules de débit et de latence théoriques. Nous observons que le débit théorique est légèrement inférieur au débit expérimental. Nous émettons l'hypothèse que cela vient d'une imprécision dans les valeurs avec lesquelles nous avons instancié les formules. En particulier, cette différence pourrait être dûe à une imprécision dans la mesure de chaque opération (génération et vérification de messages, taille des batchs, etc).



(a) $f = 1$ **(b)** $f = 2$

FIGURE 4.8 – Comparaison des débits théoriques et expérimentaux de RBFT dans le cas sans faute.

4.3.4 Performances en cas d'attaque

Dans cette section nous présentons les formules théoriques permettant de calculer le débit des différents protocoles en cas d'attaque. En particulier, nous considérons les attaques présentées dans la Section 4.1 pour Prime, Aardvark et Spinning. Concernant RBFT, nous présentons le débit minimal que peut fournir le principal malicieux de l'instance de protocole maître sans être détecté.

La Table 4.3 présente les formules de débit de chaque protocole en cas d'attaque.

Prime :	$\kappa/(2\theta + \frac{(11f+5)\theta+(6f+2)\delta+P}{b} + E + TAR_{attaque} - TAR_{sf})$
Aardvark :	$\kappa/max(\theta + \alpha + \delta, \frac{\alpha(10f+b)+\delta}{b}) + A_{delai}(obs, exp)$
Spinning :	$\frac{(n-f)*T_{sf}+f\kappa/(L_{sf}+S_t)}{n}$
RBFT :	$\frac{(1-\Delta)\kappa}{max(\frac{\theta}{g}+\alpha(\frac{1}{g}+n+f-1)+\frac{\delta}{g}, \alpha\frac{2n+4f-1}{b}+\delta)}$

TABLE 4.3 – Latence de Prime, Aardvark, Spinning et RBFT en cas d'attaque pour ordonner une requête dans un lot de requêtes de taille b.

Dans le cas de Prime, E est le temps d'exécution d'un lot de requêtes de taille b, tandis que TAR_{sf} et $TAR_{attaque}$ sont respectivement le temps de réponse maximal (c.à.d. le temps maximal entre l'envoi de deux messages PRÉ-PRÉAPARATION) dans le cas sans faute et le temps de réponse maximal en cas d'attaque. Dans le cas d'Aardvark, $A_{delai}(obs, exp)$ est une formule qui, à partir du débit observé obs et du débit requis exp, calcule le délai maximal que le principal malicieux peut appliquer aux messages d'ordonnancement sans être détecté. Dans le cas de Spinning, T_{sf} est le débit observé dans le cas sans faute et S_t est le le délai après lequel un principal est considéré comme malicieux. Dans le cas de RBFT, g est la proportion de requêtes valides envoyées par les clients malicieux et Δ est le seuil de détection d'un principal de l'instance de protocole maître malicieux.

Comme nous l'avons présenté dans la Section 4.2.1, la perte de performance théorique de Prime, Aardvark et Spinning en cas d'attaque est respectivement de 73%, 86% et 97% (notons que ces valeurs sont proches des valeurs mesurées expérimentalement présentées dans la Section 4.2.1). Ces valeurs ne sont clairement pas acceptables. Nous pouvons observer que le débit de RBFT en cas d'attaque dépend de la valeur de Δ. Nous précisons dans l'Annexe A.3 comment calculer cette valeur de manière théorique. En particulier, la valeur théorique est de zéro. Cenpendant, cette valeur théorique ne prend pas en compte les légères fluctuations de performance et de mesure des performances que l'on observe en pratique. Durant notre analyse expérimentale, nous avons observé $\Delta \geq -0,005$. Avec $\Delta = -0,005$, la perte de performance théorique maximale de RBFT est de 0,5%. Bien évidemment, une valeur plus faible de Δ conduirait à une perte maximale de performance plus élevée. C'est pourquoi, comme nous le montrons dans la Section 4.4.4, il est important de bien choisir la période de surveillance des performances.

4.4 Évaluation expérimentale

Dans cette section nous évaluons RBFT. Nous présentons tout d'abord les objectifs de cette évaluation (Section 4.4.1) puis les paramètres expérimentaux (Section 4.4.2). Puis nous comparons ses performances dans le cas sans faute avec les performances de Prime, Aardvark et Spinning (Section 4.4.3). Nous analysons ensuite la précision du mécanisme de surveillance des performances (Section 4.4.4). Enfin nous étudions les performances de RBFT en cas d'attaque (Section 4.4.5).

4.4.1 Objectifs de l'évaluation

Nous souhaitons répondre à l'interrogation suivante : RBFT est-il plus robuste que les protocoles précédents ? Nous considérons que c'est le cas s'il répond de manière positive à chacune des conditions suivantes :

1. RBFT fournit des propriétés de sûreté et de vivacité ;
2. RBFT fournit des performances comparables aux performances des protocoles considérés comme robustes dans le cas sans fautes. En effet, un protocole aussi robuste soit-il n'est pas intéressant s'il fournit de très mauvaises performances qui empêche son adoption ;
3. RBFT est robuste. D'après notre définition de la robustesse (cf. Section 2.2.1), cela signifie qu'il doit être 10-robuste, c.à.d. que la chute de débit en cas d'attaque est d'au plus 10%.

Le protocole RBFT se base sur PBFT, qui fournit des propriétés de sûreté et de vivacité. Ainsi RBFT doit fournir les mêmes propriétés. Par rapport à PBFT, nous avons rajouté une phase de communication supplémentaire. Nous montrons dans l'Annexe A.3 que cette phase maintient les propriétés de sûreté et de vivacité du système. Par conséquent la première condition est respectée. Nous montrons que les conditions suivantes sont respectées par RBFT dans la suite de cette section.

4.4.2 Paramètres expérimentaux

Nous avons évalué les performances de RBFT et des protocoles dits robustes, Prime, Aardvark et Spinning, dans le cas sans faute et en cas d'attaque, de manière expérimentale. Cette évaluation s'est faite sur un cluster composé de huit Dell PowerEdge T610 et deux Dell Precision WorkStation T7400, connectées entre elles via un réseau Gigabit Ethernet local. Les T610 possèdent deux processeurs quad-cœurs Intel Xeon E5620 cadencés à 2,4GHz, avec 16Go de RAM et 10 cartes réseau Gigabit. Les T7400 possèdent deux processeurs quad-cœurs Intel Xeon E5410 cadencés à 2,33GHz, avec 8Go de RAM et cinq cartes réseau Gigabit. Toutes ces machines utilisent un noyau Linux version 2.6.32. Les nœuds s'exécutent toujours sur les T610 ; les machines restantes sont utilisées pour exécuter les clients. Nous avons lancés les expériences avec au plus deux fautes Byzantines tolérées, c.à.d. $f \leq 2$.

Sauf indication contraire les protocoles étaient configurés pour tolérer une faute Byzantine (cas $f = 1$) et nous avons utilisé la version TCP de RBFT.

Nous avons soumis les différents protocoles à deux charges différentes : une charge statique, où le système est saturé et les clients envoient leurs requêtes à une vitesse constante, et une charge dynamique, où le débit des clients varie au cours du temps. Nous présentons la charge utilisée avec des requêtes de taille nulle. Des charges similaires ont été utilisées pour les autres tailles de requêtes. Notons que les clients envoient tous leurs requêtes au même débit. Nous avons donc dû adapter le nombre de clients en fonction de la taille des requêtes, étant donné que le nombre de clients nécessaire pour atteindre le débit maximal de chaque protocole dépend de la taille des requêtes. L'expérience commence avec un seul client. Nous augmentons progressivement le nombre de clients jusqu'à 10. Ensuite nous simulons un pic de charge avec 50 clients. Enfin, le nombre de clients est réduit progressivement jusqu'à ce qu'il n'y en est plus qu'un seul d'actif. Une charge similaire a déjà été utilisée dans [49]. Lorsque la charge est dynamique, nous considérons que le débit de l'expérience est le débit moyen observé sur toute la durée de celle-ci.

Les clients envoient leurs requêtes en boucle ouverte, comme définit dans [74], c.à.d. qu'ils n'attendent pas la réponse d'une requête avant d'en envoyer une nouvelle. Cette condition est nécessaire étant donné que, dans RBFT, seules les requêtes ordonnées par l'instance maître sont exécutées et amènent le nœud à envoyer une réponse au client. Bien que les protocoles de TFB soient généralement évalués en boucle fermée, ce type de banc de test a du sens pour les services asynchrones qui pourraient bénéficier de la tolérance aux fautes Byzantines. Par exemple, Zookeeper [77], Boxwood [78] ou Chubby [79] fournissent une version asynchrone de leurs APIs. Cela permet de concevoir des programmes plus efficaces dans certains cas ou de supporter d'autres modèles de programmation tels que la programmation événementielle.

4.4.3 Performances dans le cas sans faute

Nous présentons dans cette section les performances dans le cas sans faute de RBFT, Prime, Aardvark et Spinning. Nous présentons également les performances de RBFT lorsque le protocole de communication utilisé est UDP au lieu de TCP. Les figures 4.9 et 4.10 présentent la latence des différents protocoles en fonction du débit, pour respectivement des requêtes nulles et des requêtes de 4ko.

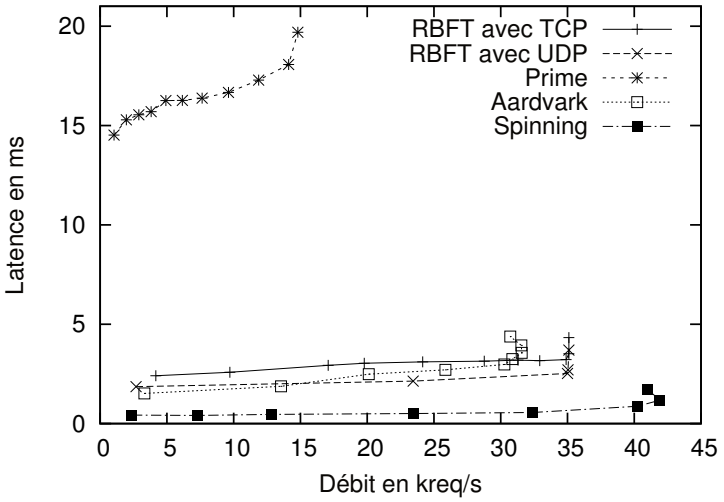


FIGURE 4.9 – Latence en fonction du débit pour des requêtes de taille nulle ($f = 1$).

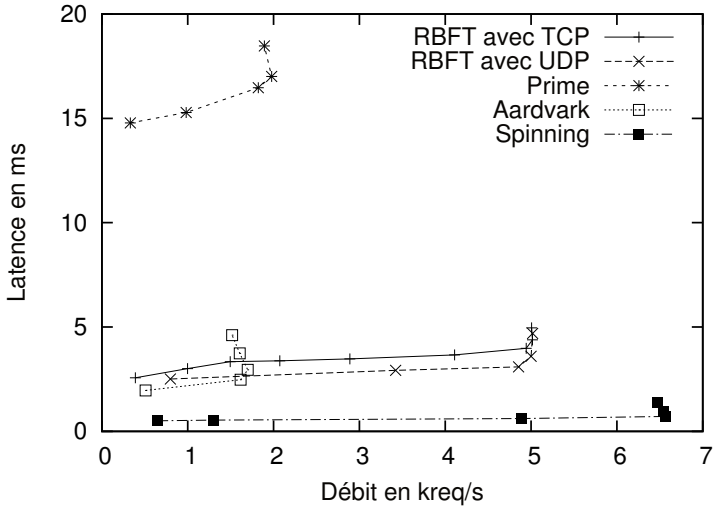


FIGURE 4.10 – Latence en fonction du débit pour des requêtes de 4ko ($f = 1$).

Tout d'abord, nous observons que Spinning fournit le meilleur débit et la plus faible latence. Plus précisément, avec des requêtes nulles, son débit est 20% plus élevé que le débit de RBFT et Aardvark, et 183% plus élevé que le débit de Prime. De manière similaire, avec des requêtes de 4ko, son débit est 30% plus élevé que le débit de RBFT. Ses bonnes performances sont principalement dues à l'utilisation d'UDP et de la multidiffusion, ce qui lui permet de fournir une latence basse. De plus, contrairement à Aardvark et RBFT, Spinning n'ajoute aucun surcoût au protocole de validation en trois phases : les changements de vue sont automatiques, il n'y a pas de phases additionnelle d'échange de messages, et il n'y a pas de vérification supplémentaire du comportement et des performances des réplicas (à part bien évidemment un minuteur pour détecter un principal malicieux qui retarderait les messages d'ordonnancement). Enfin, Spinning ne génère et vérifie aucune signature, contrairement aux autres protocoles conçus pour être robustes.

La seconde observation que nous pouvons faire est que les performances de RBFT sont supérieures aux performances d'Aardvark. Pour des requêtes nulles, le débit maximal de RBFT est de 35 kreq/s, tandis que le débit maximal d'Aardvark est de $31,6$ kreq/s. De même, pour des requêtes de 4ko, le débit maximal de RBFT est de 5 kreq/s, tandis que le débit maximal d'Aardvark est de $1,7$ kreq/s. Cela peut sembler surprenant étant donné que le protocole exécuté par les réplicas de RBFT imite Aardvark et utilise la même base de code. La raison pour laquelle RBFT est plus efficace est qu'il n'effectue pas de changement de vues périodiques (nous rappelons que dans RBFT les changements de vues sont remplacés par des changements d'instance de protocole, exécutés seulement en présence de fautes). Afin de valider cette hypothèse, nous avons désactivé les changements de vue périodique d'Aardvark et avons observé le même débit que celui de RBFT pour des requêtes de petite taille. Pour de plus grosses requêtes, RBFT fournit un meilleur débit parce que les instances de protocole n'ordonnent que les identifiants des requêtes, non pas les requêtes complètes. Comme montré dans PBFT [14] et UpRight [83], ordonner des requêtes de grande taille conduit à une latence plus élevée et à un débit plus faible. De plus, la taille des lots de requêtes est bornée, ce qui limite le nombre de requêtes qu'il est possible d'ordonner avec un seul message PRÉ-PRÉPARATION. Nous avons lancé une expérience dans laquelle les instances de protocole de RBFT ordonnent la requête complète. Dans ce cas, le débit maximal, pour des requêtes de 4ko, chute à $1,8$ kreq/s.

Troisièmement, nous observons que Prime fournit les moins bonnes performances, notamment en terme de latence. En effet, sa latence est un ordre de magnitude plus élevée que la latence des autres protocoles, pour les deux tailles de requêtes présentées. Cette latence élevée est due au fait que Prime n'utilise que des signatures, qui sont connues pour être plus lentes que les MACs. De plus, dans Prime, la plupart des étapes du protocole sont déclenchées périodiquement plutôt que suivant l'arrivée d'un message. Bien qu'il soit possible de réduire la latence en envoyant les messages plus fréquemment, nous avons configuré Prime de façon telle qu'il fournisse le meilleur débit. Enfin, Prime a été conçu pour des réseaux étendus (*Wide-area network*), dans lesquels la latence entre les différents nœuds du système est un ordre de grandeur plus élevée que dans un réseau local : Prime a été évalué dans un réseau dans lequel le temps de réponse était de 50ms [45].

Enfin, nous observons que les implantations UDP et TCP de RBFT fournissent le même débit maximal. La seule différence est que l'implantation UDP fournit une

latence 22% plus basse (respectivement 18%) que l'implantation TCP, pour des requêtes nulles (respectivement de 4ko). Cette différence de latence est due aux mécanismes de TCP qui assurent sa robustesse (acquittement, contrôle de flux, etc.). Notons que le choix du mécanisme de communication n'a pas d'impact sur les performances de RBFT en cas d'attaque. Nous avons observés des résultats similaires, que ce soit avec TCP ou UDP.

L'analyse théorique des performances présentée dans la Section 4.3.2 est en accord avec notre analyse expérimentale. Cependant, nous n'avions pas observé de grandes différences entre Aardvark, Spinning et RBFT en comparant seulement le coût des opérations de cryptographie. Les différences importantes que nous observons dans cette analyse expérimentale sont dûes à l'implantation des protocoles. En particulier, Aardvark effectue des changements de vue réguliers, ce qui impacte son débit, et Spinning utilise UDP avec multicast, ce qui lui permet de fournir de meilleures performances que les autres protocoles.

4.4.4 Précision du mécanisme de surveillance des performances

RBFT détecte que le principal de l'instance de protocole maître est malicieux en surveillant périodiquement les performances des instances de protocole. Dans cette section, nous analysons la précision de cette surveillance, pour une charge statique et une charge dynamique. Plus précisément, nous sommes intéressés par l'impact de la période de surveillance sur le ratio minimal observé par les nœuds entre les performances de l'instance maître et des instances de surveillance, et sur la distribution des ratios. Le ratio minimal permet à un nœud de détecter si le principal de l'instance maître est malicieux ou non : un nœud considère que le principal de l'instance de protocole maître est malicieux dès que le ratio qu'il observe est inférieur à un seuil pré défini. Si le ratio minimal est trop bas, alors il sera plus difficile de détecter une attaque. Au contraire, si le ratio minimal est trop élevé, alors il y aura des changements d'instance de protocole causés par des faux positifs. De plus, plus la période de monitoring est grande et plus un nœud malicieux peut attaquer le système longtemps avant d'être détecté. Enfin, la distribution des ratios observés rend compte de leur volatilité : plus les valeurs observées sont distribués, et plus le protocole risque d'initier des changements d'instance de protocole sans qu'aucun nœud ne soit malicieux.

La Table 4.4 présente le ratio minimal observé, sous une charge statique et une charge dynamique, dans le cas sans faute, pour des requêtes de taille nulle et de 4ko, en fonction de la période de surveillance : de 1ms à 1 seconde. Notons qu'augmenter davantage la période de monitoring, à plus d'une seconde, n'améliore pas la précision du mécanisme de manière significative. Nous observons que, pour une période de 1ms, les nœuds ne peuvent pas détecter une attaque car le ratio minimal observé est $-\infty$. Plus on augmente la période de surveillance, et plus le ratio minimal devient proche de 0. Dit autrement, plus la période de monitoring est grande et moins les nœuds observent de différence entre les performances des différentes instances de protocole. Par exemple, avec des requêtes de taille nulle et sous une charge statique, le ratio est de -7 avec une période de 10ms et de seulement -0,003 avec une période de monitoring d'une seconde. Enfin, le ratio observé dépend de la charge pour des périodes de surveillance jusqu'à 100ms (par exemple, pour des requêtes de taille nulle, il est de -7 sous une charge statique et de -1 sous une charge dynamique). Toutefois, le ratio est assez similaire

	1ms	10ms	100ms	1s
0ko	$-\infty$	-7	-0.2	-0.003
4ko	$-\infty$	-4	-0.1	-0.002

(a) Charge statique.

	1ms	10ms	100ms	1s
0ko	$-\infty$	-1	-0.07	-0.004
4ko	$-\infty$	-0.7	-0.05	-0.005

(b) Charge dynamique.

TABLE 4.4 – Ratio minimal observé par le mécanisme de surveillance des performances de RBFT en fonction de la période de monitoring et de la taille des requêtes, pour une charge statique et une charge dynamique. Les expériences ont étés lancées dans le cas sans faute et avec $f = 1$.

quelque soit la charge et la taille des requêtes lorsque la période de surveillance est d'une seconde.

Nous analysons maintenant la distribution des ratios observés par les différents réplicas, pour différentes périodes de surveillance et sous différentes charges. La Figure 4.11 présente la courbe du pourcentage cumulé des ratios observés, pour des périodes de monitoring de 1ms à 1 seconde, sous une charge statique, avec des requêtes de 4ko. Les résultats avec des requêtes nulles et/ou une charge statique sont similaires. Nous

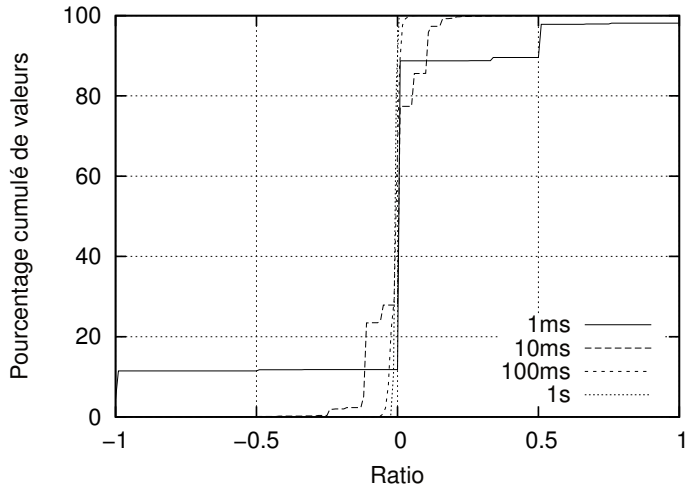


FIGURE 4.11 – Pourcentage cumulé des ratios observés pour différentes périodes de surveillance, dans le cas sans faute, pour des requêtes de 4ko et sous une charge statique.

observons que plus la période de surveillance est grande, plus la distribution des ratios est proche de 0. C'est-à-dire que la différence entre les performances observées par les différentes instances devient plus petite. Ainsi, un principal malicieux au niveau de

l'instance maître aura moins de libertés lorsqu'il voudra attaquer le système, car il sera détecté avec une plus grande précision.

Dans la suite de l'évaluation, nous avons configuré RBFT avec une période de surveillance de 1s.

4.4.5 Performances en cas d'attaque

Nous présentons dans cette section les performances de RBFT en cas d'attaque. Nous présentons tout d'abord deux attaques que nous considérons comme les pires attaques qui puissent être exécutées contre le protocole. Ces attaques montrent les dommages qu'un attaquant peut causer au protocole dans deux situations différentes : (1) lorsque le principal de l'instance maître est correct (attaque *déstitution d'un principal correct*, Section 4.4.5.1), et (2) lorsque le principal de l'instance maître est malicieux (attaque *complot d'un principal malicieux*, Section 4.4.5.2). Nous considérons ces attaques comme les pires attaques dans leurs contextes respectifs, car, dans chaque cas, les f nœuds malicieux et n'importe quel proportion de clients malicieux complotent afin de réduire les performances du système. Nous avons lancé les expériences avec une charge statique et une charge dynamique, dans deux configurations : $f = 1$ et $f = 2$. Nous montrons que la perte de performance maximale est de seulement 3%. Enfin, nous montrons également que la surveillance de la latence permet de détecter un principal qui ne traiterait pas les requêtes des différents clients de manière équitable (Section 4.4.5.3).

4.4.5.1 Attaque *déstitution d'un principal correct*

Dans cette attaque, f nœuds et n'importe quelle proportion de clients sont malicieux. Le principal de l'instance maître est correct (c.à.d. qu'il s'exécute sur un nœud correct). Le but de cette attaque est de décroître les performances de l'instance maître le plus possible afin de provoquer un changement d'instance de protocole. Cela peut permettre de placer un réplica malicieux comme principal de l'instance maître, ce que nous souhaitons éviter.

Soit p le nœud sur lequel s'exécute le principal de l'instance maître. L'attaque est la suivante :

- les clients malicieux envoient des requêtes invalides au nœud p ;
- les f nœuds malicieux attaquent le nœud p par déni de service par inondation, en envoyant des messages PROPAGATION invalides de la taille maximale [6] ;
- les réplicas malicieux de l'instance de protocole maître attaquent les réplicas corrects par déni de service par inondation, en envoyant des messages invalides de la taille maximale et ne participent pas au protocole.

Nous avons lancé des expériences avec des requêtes de taille nulle jusqu'à à 4ko. Les figures 4.12 et 4.13 montrent le débit durant l'attaque relatif au débit dans le cas sans faute pour une charge statique et une charge dynamique, respectivement lorsque $f = 1$ et $f = 2$. Ces figures montrent le cas lorsque tous les clients étaient corrects. Les résultats sont équivalents lorsque la proportion de clients malicieux augmente. Nous rappelons que la vérification des requêtes des clients s'effectue en parallèle du traitement des messages du protocole. Par ailleurs, les requêtes peuvent également être

6. La taille maximale des messages dans notre implantation est 9ko.

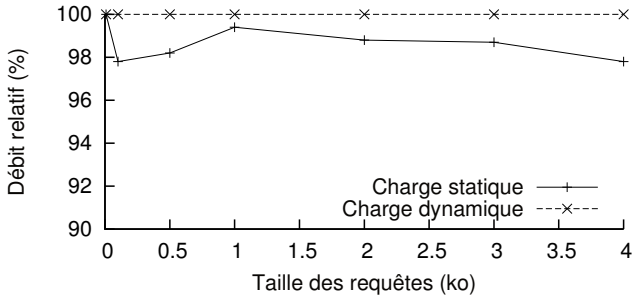


FIGURE 4.12 – Débit de RBFT relatif au débit sans faute lors de l'attaque *déstitution d'un principal correct* ($f = 1$).

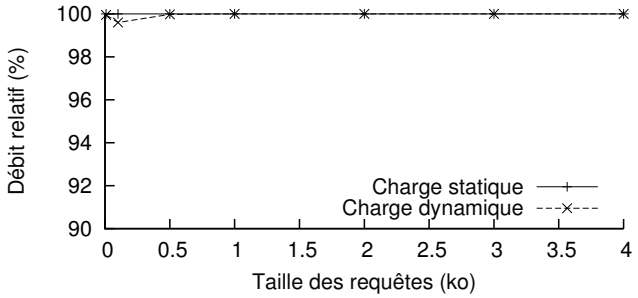


FIGURE 4.13 – Débit de RBFT relatif au débit sans faute lors de l'attaque *déstitution d'un principal correct* ($f = 2$).

reçues lors de la phase de PROPAGATION, pour laquelle un second processus léger est dédié. Par conséquent, les clients malicieux, qui augmentent la charge de travail seulement du processus léger de vérification des requêtes, ne peuvent pas faire baisser les performances du protocole. De plus les nœuds malicieux qui attaquent p par déni de service se retrouvent isolés sont détectés par le mécanisme de liste noire. Étant donné que le protocole utilise différentes interfaces réseau, une par réplica, les interfaces réseau qui permettent de communiquer avec les nœuds malicieux sont désactivés. Ces derniers ne peuvent pas impacter les nœuds corrects. En résumé, nous observons que RBFT est robuste : sa perte de débit maximale est inférieure à $2,2\%$ sous une charge statique, et est nulle lorsque la charge est dynamique (lorsque $f = 1$). De manière similaire, lorsqu'au plus deux fautes sont tolérés, c.à.d. lorsque $f = 2$, nous observons que la perte de débit est d'au plus $0,4\%$.

Afin d'illustrer le comportement de RBFT, nous avons enregistré le débit mesuré par le mécanisme de surveillance des différents nœuds. Les résultats, pour une charge statique, avec des requêtes de 4ko et lorsqu'au plus une faute est tolérée, sont représentés dans la Figure 4.14. Nous avons observés des résultats similaires dans les autres configurations. Cette figure montre tout d'abord que chaque nœud mesure le même débit. De plus, elle montre que le débit mesuré pour l'instance de protocole maître est très proche du débit mesuré pour les instances de surveillance, la différence étant de 2%. Notons que nous ne représentons pas les valeurs reportées par le nœud 3 car ce nœud était malicieux dans notre expérience, et pouvait donc reporter des valeurs arbitraires.

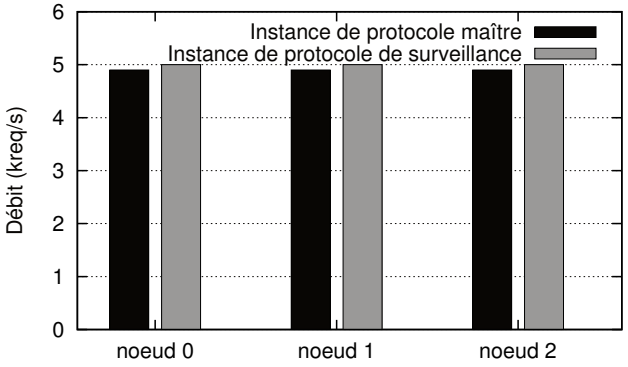


FIGURE 4.14 – Débit mesuré par les différents nœuds l'attaque *déstitution d'un principal correct* ($f = 1$, charge statique, requêtes de 4ko.).

4.4.5.2 Attaque *complot d'un principal malicieux*

Dans cette attaque, f nœuds et n'importe quelle proportion de clients sont malicieux. Le principal de l'instance maître est malicieux (c.à.d. s'exécute sur un nœud malicieux). Les clients et les nœuds malicieux tentent de réduire les performances des instances de protocoles de surveillance afin de permettre au principal malicieux de l'instance maître de réduire les performances du système sans être détecté. Pour cela, les nœuds et les clients malicieux complotent et prennent les actions suivantes :

- les clients malicieux envoient des requêtes invalides aux nœuds corrects ;
- les f nœuds malicieux attaquent les nœuds corrects par déni de service par inondation, en envoyant des messages PROPAGATION invalides de la taille maximale, et ne participent pas au protocole ;
- les réplicas des instances de protocole de surveillance qui s'exécutent sur les f nœuds malicieux attaquent les réplicas corrects par déni de service par inondation en envoyant des messages invalides de la taille maximale et ne participent pas au protocole ;

Nous pourrions choisir les réplicas malicieux de telle façon qu'il est nécessaire d'exécuter f changements d'instance de protocole avant que le principal de l'instance de protocole maître ne soit correct à nouveau. Dans un tel cas, la baisse de débit dure plus longtemps mais n'est pas plus importante. De plus, étant donné que le mécanisme de surveillance des performances détecte très rapidement un principal malicieux, nous considérons que le temps passé à exécuter f principaux malicieux à la suite est négligeable en comparaison de la durée d'exécution du protocole dans le cas sans fautes. C'est pourquoi nous avons choisi d'évaluer la perte de performances dans une situation où le principal de l'instance maître est correct après le premier changement d'instance de protocole.

Nous avons lancé des expériences avec des requêtes de taille nulle jusqu'à à 4ko. Les figures 4.15 et 4.16 montrent le débit durant l'attaque relatif au débit dans le cas sans faute pour une charge statique et une charge dynamique, respectivement lorsque $f = 1$ et $f = 2$. Le principal malicieux de l'instance de protocole maître fait décroître le débit de son instance en retardant l'émission des messages d'ordonnancement, jusqu'à la valeur minimale autorisée telle que le ratio observé par les nœuds corrects est

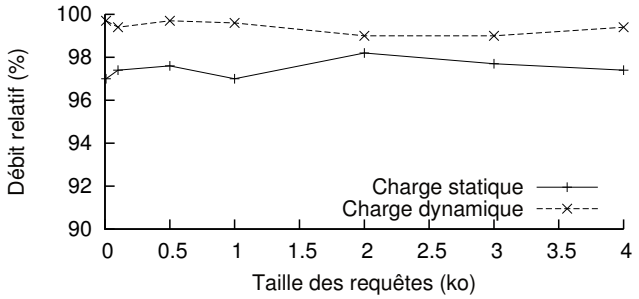


FIGURE 4.15 – Débit de RBFT relatif au débit sans faute lors de l'attaque *complot d'un principal malicieux* ($f = 1$).

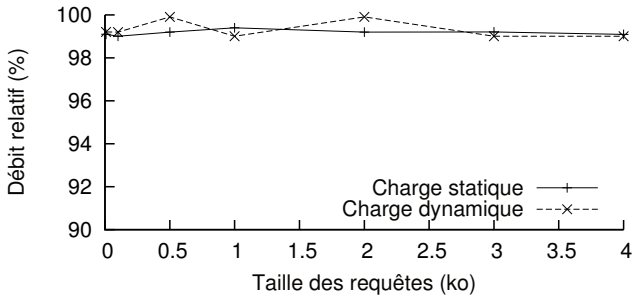


FIGURE 4.16 – Débit de RBFT relatif au débit sans faute lors de l'attaque *complot d'un principal malicieux* ($f = 2$).

supérieure ou égale à Δ. En effet, un ratio inférieur implique un changement d'instance de protocole. Grâce au traitement des requêtes des clients en parallèle du traitement des autres messages du protocole, les clients malicieux ne peuvent réduire les performances du système, quelque soit leur proportion. De plus, grâce au mécanisme de liste noire et à l'utilisation de plusieurs interfaces réseau séparées, les nœuds malicieux qui attaquent le système par déni de service par inondation se retrouvent isolés des nœuds corrects. Une fois de plus, nous observons que RBFT est robuste : la perte de performance maximale est inférieure à 3% lorsque $f = 1$ et inférieure à 1% lorsque $f = 2$.

Comme pour la première attaque, nous présentons dans la Figure 4.17 le débit mesuré par le mécanisme de surveillance des performances des différents nœuds. De manière similaire à l'attaque précédente, nous pouvons constater que tous les nœuds mesurent le même débit. De plus, nous observons que le débit observé par l'instance de protocole maître est similaire au débit mesuré par les instances de protocole de surveillance. Ce débit similaire explique pourquoi RBFT est robuste.

4.4.5.3 Attaque *principal non équitable*

Dans cette section, nous présentons une attaque dans laquelle le principal de l'instance maître ne traite pas les requêtes des clients de manière équitable : il propose un ordonnancement pour les requêtes d'un client donné moins fréquemment que pour les requêtes des autres clients. Nous montrons que le principal malicieux ne peut pas augmenter la latence des requêtes de ce client au-delà d'une limite basse. Nous avons

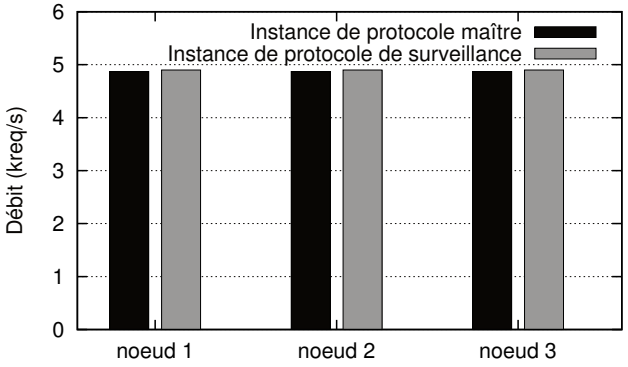


FIGURE 4.17 – Débit mesuré par les différents nœuds durant l'attaque *complot d'un principal malicieux* ($f = 1$, charge statique, requêtes de 4ko.).

lancé une expérience dans le cas $f = 1$ avec deux clients et des requêtes de 4ko. La latence maximale acceptable, Λ, est de 1,5ms. Nous avons donné une valeur élevée à Ω, qui est la différence maximale acceptable entre la latence moyenne d'un client sur les différentes instances de protocoles. Nous montrons que le principal de l'instance maître ne peut pas augmenter la latence d'un client au-delà de Λ sans être détecté, ce qui a pour conséquence d'initier un changement d'instance de protocole.

La figure 4.18 montre la latence de chaque requête, pour les deux clients, lors de l'expérience. Au début de l'expérience et durant 500 requêtes, le principal malicieux,

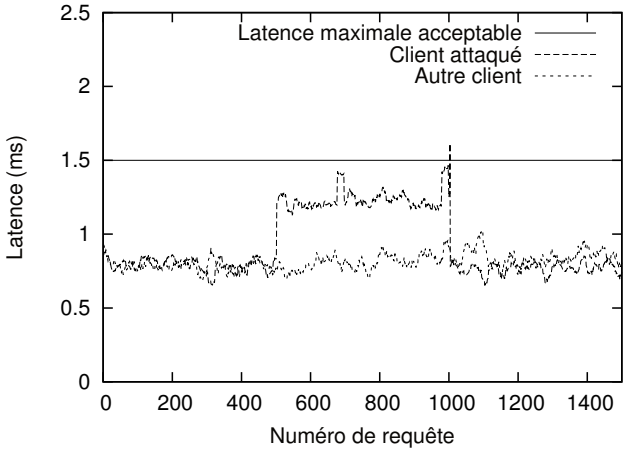


FIGURE 4.18 – Latence d'ordonnancement des requêtes de 2 clients sur l'instance de protocole maître lorsque le principal n'est pas équitable et commence à retarder les requêtes de l'un des clients après 500 requêtes.

sur l'instance de protocole maître, se comporte normalement. La latence moyenne observée est de $0,8ms$. Puis il propose un ordonnancement des requêtes du premier client moins souvent, afin que la latence observée soit de $1,3ms$, durant 500 requêtes. Étant donné que cette latence est inférieure à la valeur de Λ, le principal est considéré comme correct par les nœuds corrects. À la requête 1000, il augmente davantage la latence observée pour le client attaqué. Il envoie une seule requête, dont la latence

d'ordonnancement est de $1,6ms$. Étant donné que cette valeur est supérieure à la valeur de Λ, les nœuds corrects votent un changement d'instance de protocole. Par conséquent, le principal malicieux de l'instance maître est remplacé par un réplica correct. Ce réplica est équitable et fournit la même latence pour les requêtes des deux clients.

Notons que durant cette expérience le mécanisme de surveillance des performances n'a pas observé de différence de débit entre les différentes instances. Cette attaque a pu être détectée grâce au mécanisme de vérification de l'équité du principal uniquement.

4.5 Conclusion

Dans ce chapitre nous avons vu que les protocoles dits robustes, Prime, Aardvark et Spinning, ne le sont pas. Ils présentent tout trois un défaut de conception qui permet à un principal malicieux de retarder les messages d'ordonnancement de manière intelligente et de faire baisser dramatiquement les performances du système. Pour résoudre ce problème de robustesse, nous proposons une nouvelle approche qui consiste à exécuter plusieurs instances d'un protocole robuste en parallèle et de surveiller leurs performances afin de détecter un principal malicieux. Nous avons conçu, implanté et évalué le protocole RBFT. Nous avons montré que ses performances dans le cas sans faute sont comparables aux performances des protocoles dits robustes. De plus, RBFT est plus robuste que les autres protocoles en cas d'attaque étant donné qu'il borne la perte de débit maximale à 3% dans des conditions drastiques, lorsque f nœuds et une proportion arbitraire de clients sont malicieux et complotent. Nous avons montré que la perte de performance reste faible même lorsque le nombre de fautes tolérées augmente (de $f = 1$ à $f = 2$).

Conclusion

Ce chapitre conclut ce manuscrit. Nous fournissons tout d'abord un résumé des contributions de cette thèse, avant d'illustrer des axes de recherche futurs.

Contributions de cette thèse

Cette thèse se concentre sur la création de protocoles de réplication de machines à états tolérant les fautes arbitraires (aussi appelées *Byzantines*) efficaces et robustes. Nous avons pu observer deux problèmes posés par les protocoles actuels. Premièrement, les protocoles actuels sont soit conçus pour être efficace, c.à.d. qu'ils cherchent à afficher les meilleures performances possibles dans le cas sans faute, soit conçus pour être robustes, c.à.d. qu'ils cherchent à minimiser leur perte de performance en cas d'attaque. Toutefois, il n'existe pas de protocole conçu pour être à la fois efficace et robuste, c.à.d. un protocole qui fournit les meilleures performances possibles dans le cas sans faute et pour lequel la perte de performance en cas d'attaque est faible. Deuxièmement, nous avons pu observer que les protocoles conçus pour être robustes ne le sont pas en réalité : Prime, Aardvark et Spinning présentent chacun un défaut de conception important qui permet à un attaquant de faire baisser leur débit d'au moins 78%, ce qui n'est pas acceptable. Cette thèse apporte deux contributions au domaine, en répondant à chacun de ces deux problèmes.

Tout d'abord, nous avons présenté un nouveau protocole de réplication efficace et robuste : R-Aliph. Ce protocole se base sur Abstract afin de combiner un protocole efficace mais peu robuste, Aliph, avec un protocole moins efficace mais conçu pour être robuste, Aardvark. De plus, quatre principes importants permettent à ce protocole d'être plus robuste. Premièrement, R-Aliph implante un protocole conçu pour être robuste, Aardvark. Deuxièmement, R-Aliph implante un mécanisme de surveillance du débit qui lui permet de fournir un débit toujours au moins égal au débit que fournirait le protocole conçu pour être robuste. Troisièmement, R-Aliph implante un mécanisme de surveillance de l'équité du principal. Cela lui permet de détecter des attaques dans lesquelles un réplica Byzantin privilégierait certains clients au détriment des autres. Quatrièmement, R-Aliph isole les ressources de calcul et réseau afin de garantir que le temps de basculement entre les protocoles ne peut être impacté par des clients ou réplicas Byzantins. Nous avons évalué R-Aliph dans le cas sans faute et en cas d'attaque. Nous avons montré que R-Aliph fournit de très bonnes performances dans le cas sans faute : son débit est au plus 6% inférieur au débit d'Aliph. De plus, en cas d'attaque, R-Aliph peut basculer rapidement vers Backup (qui implante Aardvark) sans être

impacté par les clients ou réplicas Byzantins. Une fois dans Backup, les performances de R-Aliph sont égales aux performances du protocole conçu pour être robuste utilisé dans Backup, c.à.d. Aardvark.

Dans un second temps nous avons présenté un nouveau protocole plus robuste que les précédents : RBFT. Ce protocole se base sur une nouvelle approche qui consiste à exécuter plusieurs instances d'un protocole conçu pour être robuste en parallèle et de surveiller leurs performances afin de détecter un principal malicieux. L'une de ces instances est appelée *instance maître*, tandis que les autres instances sont appelées *instances de surveillance*. Périodiquement, les nœuds mesurent les performances des différentes instances (plus précisément des réplicas des différentes instances qui s'exécutent sur la même machine). Si le ratio entre les performances de l'instance maître et les performances des instances de surveillance est inférieur à un seuil prédéfini, alors le principal de l'instance maître est considéré comme malicieux. Si une majorité de nœuds considère le principal de l'instance maître comme malicieux, alors ils votent pour un changement d'instance de protocole. Plus spécifiquement, les nœuds déclenchent un changement de principal sur chacune des instances de protocole. Ainsi le réplica malicieux qui était principal de l'instance maître perd son rôle de principal. Un nouveau réplica, potentiellement correct, devient principal à la place du réplica malicieux. RBFT implante également d'autres mécanismes qui lui permettent d'être robuste, comme l'isolation des ressources de calcul et réseau ou un mécanisme de liste noire qui permet de se prémunir d'un client ou nœud malicieux qui attaquerait le système en envoyant des messages invalides. Nous avons montré que ses performances dans le cas sans faute sont comparables aux performances des protocoles dits robustes. De plus, RBFT est plus robuste que les autres protocoles en cas d'attaque : nous avons observé une perte maximale de débit de 3%, même dans des conditions drastiques lorsque f nœuds et une proportion arbitraire de clients sont malicieux et complotent afin de réduire les performances du système. Nous avons observé que la perte de performance reste faible même lorsque le nombre de fautes tolérées augmente (de $f = 1$ à $f = 2$).

Bien que ces deux protocoles adressent le problème de la robustesse des protocoles tolérant les fautes Byzantines, ils n'ont pas tout à fait le même objectif : R-Aliph est un protocole construit dans le but d'être efficace dans le cas sans fautes et robuste en cas d'attaque, tandis que RBFT est un protocole conçu pour être le plus robuste possible. Alors que R-Aliph se base sur la spéculation afin d'améliorer ses performances dans le cas sans fautes, RBFT cherche à réduire la marge de manoeuvre d'un attaquant qui voudrait faire baisser les performances du système. Même s'il est important que RBFT fournisse de bonnes performances, afin de faciliter son adoption, l'accent est mis sur la robustesse du protocole. Une idée simple permettant d'améliorer les performances de RBFT dans le cas sans fautes est de combiner R-Aliph avec RBFT. Nous pourrions par exemple implanter RBFT à la place d'Aardvark dans Backup.

Perspectives

Les travaux menés dans cette thèse ouvrent des pistes pour de futurs travaux, aussi bien des travaux d'extensions à court terme que des études plus profondes à long terme. Nous présentons dans cette section quelques directions futures possibles.

R-Aliph aussi rapide que possible

R-Aliph est aussi efficace qu'un protocole spéculatif dans le cas sans faute et aussi robuste qu'un protocole conçu pour être robuste en cas d'attaque. Cependant, les performances des protocoles conçus pour être robustes sont inférieures aux performances des protocoles spéculatifs (p. ex., le débit maximal d'Aardvark est un peu moins de deux fois inférieur au débit maximal d'Aliph). Il nous semble intéresser de diminuer l'écart de performance entre les protocoles conçus pour être robustes et les protocoles spéculatifs.

Rappelons qu'un basculement de protocole a lieu uniquement lorsque le débit fourni par le protocole spéculatif est inférieur au débit de Backup. Les réplicas malicieux des protocoles spéculatifs peuvent faire chuter le débit au débit de Backup, bien inférieur au débit maximal que peuvent fournir les protocoles spéculatifs, sans être détectés. Nous pourrions envisager d'avoir à notre disposition tout un ensemble de protocoles spéculatifs, chacuns pouvant résister à un sous-ensemble d'attaques. Par exemple, considérons un réplica malicieux qui retarde l'ordonnancement des requêtes. Chain n'est pas robuste, contrairement à Zyzzyva : il lui suffit d'initier un changement de vue afin de remplacer le principal malicieux par un réplica correct. De plus, Zyzzyva fournit des performances proches de Chain dans certaines configurations [49]. Nous pourrions construire un détecteur d'attaque, qui surveille de près le comportement des différents réplicas. Ce détecteur pourrait être utilisé afin de changer le protocole spéculatif utilisé actuellement, voir d'initier une reconfiguration (telle qu'un changement de vue) dans le protocole spéculatif. Ainsi, en présence de certaines attaques, les performances fournies par ce nouveau protocole seraient supérieures aux performances fournies par R-Aliph, et en particulier proches des performances fournies par R-Aliph dans le cas sans faute.

De plus, R-Aliph ne fournit pas nécessairement son meilleur débit lorsque la charge est fluctuante. En effet, supposons une exécution dans laquelle la charge de départ est maximale et R-Aliph exécute Chain. Puis la charge fluctue et devient inférieure au débit requis. R-Aliph bascule alors vers Backup. Enfin, alors que le protocole est toujours en train d'exécuter Backup, la charge devient à nouveau maximale. Cependant il n'y a pas de basculement de protocole vers R-Aliph. Nous envisageons de surveiller la charge en entrée fournie par les clients ainsi que les réplicas afin de savoir si le débit fourni par le protocole est faible car la charge est insuffisante, ou si cela est dû à un comportement malicieux. Dans le premier cas, contrairement au second, il est inutile d'initier un basculement de protocole. Avec un tel mécanisme, R-Aliph ne quitterait l'exécution d'un protocole spéculatif qu'en présence d'un comportement malicieux, même lorsque la charge est fluctuante.

RBFT en boucle fermée

Le mécanisme de surveillance des performances de RBFT peut détecter un principal malicieux au niveau de l'instance maître uniquement lorsque le système est utilisé en boucle ouverte, c.à.d. lorsqu'un client n'a pas besoin d'attendre la réponse à sa dernière requête avant d'en envoyer une nouvelle. En effet, en boucle fermée, lorsque le client doit attendre la réponse à sa dernière requête avant d'en envoyer une nouvelle, le débit des instances de surveillance est limité par le débit de l'instance maître, pour qui les requêtes sont exécutées. Ainsi on ne peut se baser sur le débit pour détecter un principal

malicieux au niveau de l'instance maître dans le cas de la boucle fermée.

Une solution serait de se baser sur la latence d'ordonnancement des requêtes. Toutefois, cela n'est pas trivial. Étant donné qu'il n'y a aucune synchronisation entre l'ordre dans lequel les instances ordonnent les requêtes, deux instances de protocole correctes peuvent ordonner deux requêtes différentes dans un ordre différent. Leurs latences seront différentes alors qu'aucune attaque n'est en cours. Afin de résoudre ce problème, nous pourrions rajouter une phase de pré-ordonnancement, similaire à celle de Prime. Cette phase permettrait aux nœuds de définir un ordre d'ordonnancement des requêtes. Toutefois, définir cet ordre nécessite un nœud spécial, équivalent au principal. Or c'est justement à cause de ce nœud spécial, qui peut devenir malicieux de manière intelligente, que les protocoles dits robustes ne le sont pas.

Très récemment (octobre 2013), Milosevic et al. ont présenté un nouvel algorithme de TFB, BFT-Mencius [84], qui garantit que la latence d'ordonnancement des requêtes valides est à terme bornée. Dans ce protocole, plusieurs consensus sont initiés en parallèle, chaque réplica étant le principal d'un consensus. Cela permet aux différents réplicas de surveiller la vitesse d'ordonnancement des différents principaux et de détecter un comportement malicieux. Cette idée est similaire à RBFT, qui exécute plusieurs instances de protocole en parallèle et surveille leurs performances. De plus, chaque consensus implante une nouvelle primitive de communication, la *diffusion annoncée en temps borné qui peut avorter* (*"Abortable Timely Announced Broadcast"*), similaire au protocole de validation en 3 phases. En outre, les réplicas calculent la latence maximale d'un consensus en utilisant une formule similaire à celle utilisée dans Prime. Si un consensus met plus de temps à ordonner une requête que cette latence maximale, alors son principal est placé sur liste noire par les réplicas corrects, de manière similaire au mécanisme de liste noire de Spinning. Les prochaines requêtes ordonnées par ce principal ne seront pas prises en compte pour exécution. Cela permet à BFT-Mencius de détecter un principal malicieux qui retarderait l'ordonnancement des requêtes puis de l'empêcher de nuire dans le futur. D'après l'évaluation faite par les auteurs de BFT-Mencius, ce protocole fournit des performances similaires à Spinning. Cependant BFT-Mencius n'utilise pas de signatures. Un client malicieux peut envoyer des requêtes invalides pour un sous-ensemble des réplicas, ce qui conduira à leur ajout sur la liste noire. Il est trivial d'obliger les clients à authentifier leurs messages avec des signatures, au détriment d'une baisse des performances du système. En résumé, BFT-Mencius partage une idée commune avec RBFT, qui est l'exécution en parallèle de plusieurs consensus. Grâce à une surveillance de la latence du réseau, il est capable de détecter un principal malicieux qui retarderait l'ordonnancement des requêtes. Contrairement à RBFT, BFT-Mencius est capable de détecter un principal malicieux en boucle fermée, mais n'implante pas certains mécanismes nécessaires à la robustesse tels que l'utilisation de plusieurs interfaces réseau. Ce protocole nous semble très intéressant et nous sommes en train de l'étudier afin de construire une version en boucle fermée de RBFT.

Exécuter plusieurs protocoles différents dans RBFT

Actuellement les instances de protocole de RBFT exécutent le même protocole. Nous pensons à une extension de RBFT dans laquelle ces protocoles seraient différents. L'instance maître serait alors le protocole fournissant les meilleurs performances. Par

exemple RBFT pourrait utiliser un protocole rapide en l'absence de contention, tel que Quorum, un protocole rapide en présence de contention, tel que Chain, et un protocole conçu pour être robuste, tel qu'Aardvark.

Aussi longtemps que les protocoles des instances de surveillance fournissent des performances plus faibles que le protocole de l'instance maître, alors le système reste dans son état actuel. Dès que l'une des instances de surveillance devient plus rapide que le protocole de l'instance maître, alors RBFT bascule le protocole utilisé au niveau de l'instance maître vers ce protocole de surveillance. Cela pose deux défis. Tout d'abord, un protocole efficace tel que Quorum ou Chain peut afficher un débit artificiellement augmenté ou baissé. Un nœud malicieux pourrait profiter de cette faille afin de déclencher un basculement d'instance de protocole. Nous avons résolu ce problème dans l'implantation de R-Aliph en forçant les clients à envoyer des messages de notification de manière périodique. Un second défi est la nécessité de synchroniser l'état des différentes instances lors du changement, Ce mécanisme serait similaire au mécanisme de basculement d'Abstract. Une différence notable est que les différents protocoles n'ordonneraient pas forcément les requêtes à la même vitesse. Cela pose la question de savoir quelles requêtes considérer lors du basculement depuis / vers un protocole en avance / en retard. Pour cela, nous pourrions nous baser sur ce qui est fait dans le mécanisme de changement de vue de Zyzzyva : les réplicas s'échangent des messages contenant la liste des dernières requêtes exécutées. Ces listes sont utilisées par le nouveau principal afin de définir les requêtes, et leur ordre d'exécution, que tous les réplicas doivent adopter avant de finaliser le changement de vue.

Machines virtuelles justifiables

Haeberl et al. ont présenté en 2010 les *machines virtuelles justifiables* (*Accountable Virtual Machines*, AVM) [85]. Une AVM est une machine virtuelle sur laquelle un audit est périodiquement créé par un composant spécial : le moniteur d'AVM (AVMM). Le but de cette approche est de pouvoir vérifier un système qui tourne dans la machine virtuelle, afin de détecter un comportement malicieux ou un problème de sécurité. De manière intéressante pour la tolérance des fautes Byzantines, l'audit ne requiert pas que l'utilisateur du système ait confiance dans le matériel ou l'AVMM.

L'audit d'une AVM fonctionne de la manière suivante : l'AVMM enregistre tous les messages reçus et envoyés par l'AVM dans un registre. Périodiquement, l'AVMM vérifie qu'il existe bel et bien une exécution correcte produisant les messages contenus dans le registre. Si c'est le cas, alors l'AVM se comporte de manière correcte. Sinon l'AVM se comporte de manière malicieuse.

Les auteurs ont utilisé une AVM afin de détecter des triches dans un jeu en ligne. Utiliser une AVM induit un surcoût qui, d'après les auteurs, est faible en comparaison des avantages de cette approche. Nous pensons que ce surcoût est tout de même important. L'utilisation d'une AVM implique de signer tous les messages qui sont transmis sur le réseau. Comme nous l'avons vu lors de l'évaluation de RBFT, le protocole Prime fournit de mauvaises performances en parti à cause de l'utilisation de signatures uniquement. De plus, le système surveillé doit être exécuté dans une machine virtuelle, ce qui est moins rapide qu'une exécution directement sur le matériel. Enfin, l'AVMM doit pouvoir rejouer une exécution de manière déterministe, sans quoi il deviendrait très complexe de vérifier que le registre est correct. De ce fait, bien que

cette approche semble intéressante, elle nous semble, de part sa conception, plus lente que l'approche utilisée dans RBFT. Un second problème de cette approche est qu'elle ne permet pas de détecter un réplica qui retarderait l'envoi d'un message. Pour cela, il faudrait pouvoir ajouter un mécanisme de surveillance supplémentaire qui vérifie que le temps d'exécution est "correct".

Nous pensons toutefois que ce concept d'AVM pourrait donner lieu à de nouveaux mécanismes de surveillance du comportement de nœuds dans un système distribué, voir à de nouveaux protocoles de TFB robustes, dans lesquels les réplicas surveillent mutuellement leur comportement.

Exécution - vérification : une nouvelle approche pour la conception de protocoles de réplication de machines à états

Kapritsos et al. ont récemment présenté Eve [86], un protocole de réplication de machines à états (RME) basé sur une nouvelle architecture dont le but est le passage à l'échelle sur les machines multicœurs. Cette architecture change le schéma "ordonnancement - exécution" des protocoles de RME traditionnels en "exécution - vérification". Plus précisément, au lieu d'ordonner puis d'exécuter les requêtes, les réplicas exécutent les requêtes dans un premier temps en parallèle en tirant parti des architectures multicœurs. L'un des réplicas a le rôle de principal. Il reçoit les requêtes des clients et forme des lots de requêtes à exécuter (en parallèle), qu'il envoie ensuite à tous les réplicas. Puis, dans un second temps, les réplicas vérifient qu'une majorité de réplicas est bien dans un état cohérent. Si ça n'est pas le cas, alors les dernières modifications sur l'état du système sont annulées et les requêtes du dernier lot de requêtes sont à nouveau exécutée, mais cette fois de manière séquentielle. De plus, cela peut causer un changement de principal.

Du point de vue de l'efficacité, Eve fournit des performances nettement meilleures que les protocoles de RME traditionnels. Plus précisément, les auteurs d'Eve ont montré, sur le banc de test TPC-W [87] en utilisant le moteur de base de données H2 [88], que cette nouvelle approche permet d'améliorer les performances des protocoles de RME traditionnels d'un facteur 6,5 et d'approcher les performances d'un système non répliqué.

Du point de vue de la robustesse, Eve n'est pas robuste. Tout d'abord, un client ou réplica malicieux pourrait inonder le réseau afin de faire chuter le débit du protocole. Cela n'est toutefois qu'un problème d'implantation et se corrige facilement. Par contre, si le principal est Byzantin, alors il peut retarder l'envoi des lots de requêtes pour exécution. Eve se base sur le même mécanisme de surveillance du débit qu'Aardvark, et est de ce fait autant vulnérable. Un principal malicieux pourrait profiter d'une charge fluctuante afin de retarder l'envoi des lots de requêtes sans être détecté, ce qui a pour conséquence une chute du débit.

En ce sens, en présence de réplicas Byzantins, Eve ne fournit aucune garantie de performance et n'est par conséquent pas robuste. Étant donné les bonnes performances fournies par l'approche "exécution - vérification", nous pensons qu'il serait intéressant d'améliorer la robustesse des protocoles basés sur cette approche. Nous pourrions utiliser l'approche de RBFT, c.à.d. exécuter plusieurs instances de protocoles (donc plusieurs réplicas ayant le rôle de principal) en parallèle et surveiller la fréquence à laquelle ils envoient les lots de requêtes.

Bibliographie

[1] E. B. Nightingale, J. R. Douceur, and V. Orgovan, "Cycles, cells and platters : an empirical analysisof hardware failures on a million consumer pcs," in *Proceedings of the sixth conference on Computer systems*, ser. EuroSys '11. New York, NY, USA : ACM, 2011, pp. 343–356.

[2] 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 *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, ser. OSDI'10. Berkeley, CA, USA : USENIX Association, 2010, pp. 1–7.

[3] B. Schroeder, E. Pinheiro, and W.-D. Weber, "Dram errors in the wild : a large-scale field study," *Commun. ACM*, vol. 54, no. 2, pp. 100–107, Feb. 2011.

[4] L. N. Bairavasundaram, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, G. R. Goodson, and B. Schroeder, "An analysis of data corruption in the storage stack," *Trans. Storage*, vol. 4, no. 3, pp. 8 :1–8 :28, Nov. 2008.

[5] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, "Have things changed now ? : an empirical study of bug characteristics in modern open source software," in *ASID '06 : Proceedings of the 1st workshop on Architectural and system support for improving software dependability*. New York, NY, USA : ACM, 2006, pp. 25–33.

[6] 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*, ser. SAC '12. New York, NY, USA : ACM, 2012, pp. 1427–1434.

[7] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea, "Deadlock Immunity : Enabling Systems To Defend Against Deadlocks," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[8] M. M. Swift, S. Martin, H. M. Levy, and S. J. Eggers, "Nooks : an architecture for reliable device drivers," in *EW 10 : Proceedings of the 10th workshop on ACM SIGOPS European workshop*. New York, NY, USA : ACM, 2002, pp. 102–107.

[9] D. L. Detlefs, K. R. M. Leino, K. Rustan, M. Leino, G. Nelson, and J. B. Saxe, "Extended static checking," 1998.

[10] J. L. Lawall, J. Brunel, N. Palix, R. R. Hansen, H. Stuart, and G. Muller, "WYSIWIB : A declarative approach to finding API protocols and bugs in linux

code," in *Proceeding of the International Conference on Dependable Systems and Networks*, Estoril (Lisbon), Portugal, Jun. 2009, pp. 43–52.

[11] A. A. Hwang, I. A. Stefanovici, and B. Schroeder, "Cosmic rays don't strike twice : understanding the nature of dram errors and the implications for system design," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. New York, NY, USA : ACM, 2012, pp. 111–122.

[12] AFP, "Un câble internet sous-marin coupé en egypte," *Libération*, 2013. [Online]. Available : http://www.liberation.fr/monde/2013/03/28/un-cable-internet-sous-marin-coupe-en-egypte_891919

[13] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Making byzantine fault tolerant systems tolerate byzantine faults," in *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, ser. NSDI'09. Berkeley, CA, USA : USENIX Association, 2009, pp. 153–168.

[14] M. Castro, "Practical byzantine fault tolerance," Ph.D., MIT, Jan. 2001, also as Technical Report MIT-LCS-TR-817.

[15] R. Guerraoui, D. Kostic, R. R. Levy, and V. Quema, "A high throughput atomic storage algorithm," *2012 IEEE 32nd International Conference on Distributed Computing Systems*, vol. 0, p. 19, 2007.

[16] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. Dependable Secur. Comput.*, vol. 1, no. 1, pp. 11–33, Jan. 2004.

[17] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, Jul. 1982.

[18] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[19] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *J. ACM*, vol. 35, no. 2, pp. 288–323, Apr. 1988.

[20] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach : a tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, Dec. 1990.

[21] L. Lamport, "Lower bounds for asynchronous consensus," in *Future Directions in Distributed Computing*, ser. Lecture Notes in Computer Science, A. Schiper, A. Shvartsman, H. Weatherspoon, and B. Zhao, Eds. Springer Berlin Heidelberg, 2003, vol. 2584, pp. 22–23.

[22] I. Gashi, P. T. Popov, and L. Strigini, "Fault tolerance via diversity for off-the-shelf products : A study with sql database servers," *IEEE Trans. Dependable Sec. Comput.*, vol. 4, no. 4, pp. 280–294, 2007.

[23] B.-G. Chun, P. Maniatis, and S. Shenker, "Diverse replication for single-machine byzantine-fault tolerance," in *ATC'08 : USENIX 2008 Annual Technical Conference on Annual Technical Conference*. Berkeley, CA, USA : USENIX Association, 2008, pp. 287–292.

[24] A. Avižienis and L. Chen, "On the implementation of N-version programming for software fault tolerance during execution," in *Proceedings of the IEEE*

International Computer Software and Applications Conference, 1977, pp. 149–155.

[25] A. Avizienis, "The n-version approach to fault-tolerant software," *IEEE Trans. Softw. Eng.*, vol. 11, no. 12, pp. 1491–1501, Dec. 1985.

[26] J. C. Knight and N. G. Leveson, "An experimental evaluation of the assumption of independence in multiversion programming," *IEEE Trans. Softw. Eng.*, vol. 12, no. 1, pp. 96–109, Jan. 1986.

[27] M. Garcia, A. Bessani, I. Gashi, N. Neves, and R. Obelheiro, "Os diversity for intrusion tolerance : Myth or reality ?" in *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks*, ser. DSN '11. Washington, DC, USA : IEEE Computer Society, 2011, pp. 383–394.

[28] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Trans. Softw. Eng.*, vol. 3, no. 2, pp. 125–143, Mar. 1977.

[29] M. W. Alford, L. Lamport, and G. P. Mullery, "Basic concepts," in *Advanced Course : Distributed Systems*, 1984, pp. 7–43.

[30] B. Alpern and F. B. Schneider, "Defining liveness," Ithaca, NY, USA, Tech. Rep., 1984.

[31] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms : Taxonomy and survey," *ACM Comput. Surv.*, vol. 36, no. 4, pp. 372–421, Dec. 2004.

[32] R. Guerraoui, R. R. Levy, B. Pochon, and V. Quéma, "Throughput optimal total order broadcast for cluster environments," *ACM Trans. Comput. Syst.*, vol. 28, no. 2, pp. 5 :1–5 :32, Jul. 2010. [Online]. Available : http://doi.acm.org/10.1145/1813654.1813656

[33] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer, "Farsite : federated, available, and reliable storage for an incompletely trusted environment," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 1–14, Dec. 2002.

[34] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "Future directions in distributed computing," A. Schiper, A. A. Shvartsman, H. Weatherspoon, and B. Y. Zhao, Eds. Berlin, Heidelberg : Springer-Verlag, 2003, ch. Towards a practical approach to confidential Byzantine fault tolerance, pp. 51–56.

[35] R. Padilha and F. Pedone, "Belisarius : Bft storage with confidentiality," in *Proceedings of the 2011 IEEE 10th International Symposium on Network Computing and Applications*, ser. NCA '11. Washington, DC, USA : IEEE Computer Society, 2011, pp. 9–16.

[36] X. Dong, J. Yu, Y. Luo, Y. Chen, G. Xue, and M. Li, "Achieving secure and efficient data collaboration in cloud computing," in *Quality of Service (IWQoS), 2013 IEEE/ACM 21st International Symposium on*, 2013, pp. 1–6.

[37] N. A. Lynch, *Distributed Algorithms*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1996.

[38] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *J. ACM*, vol. 27, no. 2, pp. 228–234, Apr. 1980.

[39] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, May 1998.

[40] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *OSDI '99 : Proceedings of the third symposium on Operating systems design and implementation*, 1999.

[41] J. Gray, "Notes on data base operating systems," in *Operating Systems, An Advanced Course*. London, UK, UK : Springer-Verlag, 1978, pp. 393–481.

[42] M. Skeen, "Crash recovery in a distributed database system," Ph.D., University of California at Berkeley, 1982.

[43] R. Boichat, P. Dutta, S. Frølund, and R. Guerraoui, "Deconstructing paxos," *SIGACT News*, vol. 34, no. 1, pp. 47–67, Mar. 2003.

[44] P. J. Marandi, M. Primi, and F. Pedone, "Multi-ring paxos," in *Proceedings of the 2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, ser. DSN '12. Washington, DC, USA : IEEE Computer Society, 2012, pp. 1–12.

[45] Y. Amir, B. Coan, J. Kirsch, and J. Lane, "Byzantine replication under attack," in *In Proceedings of the 38th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '08*, 2008, pp. 105–114.

[46] A. Clement, "Upright fault tolerance," Ph.D. dissertation, University of Texas at Austin, Austin, Texas, december 2010.

[47] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, Apr. 1985.

[48] M. K. Aguilera, W. Chen, and S. Toueg, "Failure detection and consensus in the crash-recovery model," Ithaca, NY, USA, Tech. Rep., 1998.

[49] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The next 700 bft protocols," in *Proceedings of the 5th European conference on Computer systems*, ser. EuroSys '10. New York, NY, USA : ACM, 2010, pp. 363–376.

[50] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst.*, vol. 20, pp. 398–461, November 2002.

[51] M. K. Reiter, "The rampart toolkit for building high-integrity services," in *Selected Papers from the International Workshop on Theory and Practice in Distributed Systems*. London, UK, UK : Springer-Verlag, 1995, pp. 99–110.

[52] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith, "The securering protocols for securing group communication," in *Proceedings of the Thirty-First Annual Hawaii International Conference on System Sciences - Volume 3*, ser. HICSS '98. Washington, DC, USA : IEEE Computer Society, 1998, pp. 317–.

[53] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, "Fault-scalable Byzantine fault-tolerant services," in *Proceedings of the Symposium on Operating Systems Principles (SOSP)*. ACM, 2005.

[54] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira, "HQ replication : a hybrid quorum protocol for Byzantine fault tolerance," in *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2006.

[55] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva : Speculative byzantine fault tolerance," *ACM Trans. Comput. Syst.*, vol. 27, no. 4, pp. 1–39, 2009.

[56] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung, "Spin one's wheels ? byzantine fault tolerance with a spinning primary," in *Reliable Distributed Systems, 2009. SRDS '09. 28th IEEE International Symposium on*. Los Alamitos, CA, USA : IEEE Computer Society, 2009, pp. 135–144.

[57] Y. Amir, B. Coan, J. Kirsch, and J. Lane, "Byzantine replication under attack," in *In Proceedings of the 38th IEEE/IFIP International Conference on Depe ndable Systems and Networks (DSN '08)*, 2008, pp. 105–114.

[58] ——, "Prime : Byzantine replication under attack," *IEEE Trans. Dependable Secur. Comput.*, vol. 8, no. 4, pp. 564–577, Jul. 2011.

[59] M. Correia, N. F. Neves, and P. Verissimo, "How to tolerate half less one byzantine nodes in practical distributed systems," in *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*, ser. SRDS '04. Washington, DC, USA : IEEE Computer Society, 2004, pp. 174–183.

[60] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "sel4 : formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ser. SOSP '09. New York, NY, USA : ACM, 2009, pp. 207–220.

[61] B. Schroeder and G. A. Gibson, "Disk failures in the real world : what does an mttf of 1,000,000 hours mean to you ?" in *Proceedings of the 5th USENIX conference on File and Storage Technologies*, ser. FAST '07. Berkeley, CA, USA : USENIX Association, 2007.

[62] F. Borran and A. Schiper, "Brief announcement : a leader-free byzantine consensus algorithm," in *Proceedings of the 23rd international conference on Distributed computing*, ser. DISC'09, 2009.

[63] P. Sousa, A. N. Bessani, M. Correia, N. F. Neves, and P. Verissimo, "Highly available intrusion-tolerant services with proactive-reactive recovery," *IEEE Trans. Parallel Distrib. Syst.*, vol. 21, no. 4, pp. 452–465, Apr. 2010.

[64] P. E. Veríssimo, N. F. Neves, and M. P. Correia, "Architecting dependable systems," R. De Lemos, C. Gacek, and A. Romanovsky, Eds. Berlin, Heidelberg : Springer-Verlag, 2003, ch. Intrusion-tolerant architectures : concepts and design, pp. 3–36.

[65] P. Sousa, N. F. Neves, and P. Veríssimo, "Proactive resilience through architectural hybridization," in *Proceedings of the 2006 ACM symposium on Applied computing*, ser. SAC '06. New York, NY, USA : ACM, 2006, pp. 686–690.

[66] P. Veríssimo, N. F. Neves, and M. Correia, "Crutial : the blueprint of a reference critical information infrastructure architecture," in *Proceedings of the First international conference on Critical Information Infrastructures Security*, ser. CRITIS'06. Berlin, Heidelberg : Springer-Verlag, 2006, pp. 1–14.

[67] A. N. Bessani, P. Sousa, M. Correia, N. F. Neves, and P. Veríssimo, "The crutial way of critical infrastructure protection," *IEEE Security and Privacy*, vol. 6, no. 6, pp. 44–51, Nov. 2008.

[68] P. E. Veríssimo, "Travelling through wormholes : a new look at distributed systems models," *SIGACT News*, vol. 37, no. 1, pp. 66–81, Mar. 2006.

[69] P. Sousa, N. Neves, and P. Verissimo, "How resilient are distributed f fault/intrusion-tolerant systems ?" in *Proceedings of the 2005 International*

Conference on Dependable Systems and Networks, ser. DSN '05. Washington, DC, USA : IEEE Computer Society, 2005, pp. 98–107.

[70] P. Dutta, R. Guerraoui, and M. Vukolic, "Best-case complexity of asynchronous byzantine consensus," Technical Report EPFL/IC/200499, EPFL, Tech. Rep., 2005.

[71] J.-P. Martin and L. Alvisi, "Fast byzantine consensus," *IEEE Trans. Dependable Secur. Comput.*, vol. 3, no. 3, pp. 202–215, Jul. 2006.

[72] P.-L. Aublin, R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The next 700 bft protocols," *ACM Trans. Comput. Syst.*, in review.

[73] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The next 700 bft protocols," EPFL, Tech. Rep. LPD-REPORT-2008-008, 2008, http ://infoscience.epfl.ch/record/121590/files/TR-700-2009.pdf.

[74] B. Schroeder, A. Wierman, and M. Harchol-Balter, "Open versus closed : a cautionary tale," in *Proceedings of the 3rd conference on Networked Systems Design & Implementation - Volume 3*, ser. NSDI'06. Berkeley, CA, USA : USENIX Association, 2006, pp. 18–18.

[75] P.-L. Aublin, S. B. Mokhtar, and V. Quéma, "RBFT : Redundant Byzantine Fault Tolerance," INRIA Rhône-Alpes, Tech. Rep., 2012, http ://membres-liglab.imag.fr/aublin/rbft/.

[76] P.-L. Aublin, S. Ben Mokhtar, and V. Quéma, "RBFT : Redundant Byzantine Fault Tolerance," in *Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems*, ser. ICDCS '13. Washington, DC, USA : IEEE Computer Society, 2013.

[77] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper : wait-free coordination for internet-scale systems," in *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, ser. USENIXATC'10, 2010.

[78] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou, "Boxwood : abstractions as the foundation for storage infrastructure," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA : USENIX Association, 2004, pp. 8–8.

[79] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *Proceedings of the 7th symposium on Operating systems design and implementation*, ser. OSDI '06, 2006.

[80] B. Veal and A. Foong, "Performance scalability of a multi-core web server," in *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, ser. ANCS '07. New York, NY, USA : ACM, 2007, pp. 57–66.

[81] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "Routebricks : exploiting parallelism to scale software routers," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ser. SOSP '09. New York, NY, USA : ACM, 2009, pp. 15–28.

[82] T. Friedman and R. Van Renesse, "Packing messages as a tool for boosting the performance of total ordering protocls," in *Proceedings of the 6th IEEE*

International Symposium on High Performance Distributed Computing, ser. HPDC '97. Washington, DC, USA : IEEE Computer Society, 1997, pp. 233–.

[83] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche, "Upright cluster services," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ser. SOSP '09. New York, NY, USA : ACM, 2009, pp. 277–290.

[84] Z. Milosevic, M. Biely, and A. Schiper, "Bounded delay in byzantine-tolerant state machine replication," in *Reliable Distributed Systems (SRDS), 2013 IEEE 32nd International Symposium on*, 2013, pp. 61–70.

[85] A. Haeberlen, P. Aditya, R. Rodrigues, and P. Druschel, "Accountable virtual machines," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, ser. OSDI'10. Berkeley, CA, USA : USENIX Association, 2010, pp. 1–16.

[86] M. Kaptritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin, "Eve : Execute-verify replication for multi-core servers," in *OSDI 2012*, Oct 2012.

[87] Transaction Processing Performance Council, "The TPC-W home page." [Online]. Available : http://www.tpc.org/tpcw

[88] H2, "The H2 home page." [Online]. Available : http://www.h2database.com

Table des figures

Liste des tableaux

Analyse théorique des protocoles de TFB conçus pour être robustes

Nous présentons dans cette annexe une analyse théorique des protocoles de TFB conçus pour être robustes considérés dans le Chapitre 4, c.à.d. Prime, Aardvark, Spinning et RBFT. Pour chacun, nous fournissons ses formules de débit et de latence dans le cas sans faute et dans le cas des attaques présentées dans la Section 4.1 (Section A.2). Nous fournissons également les esquisses de preuve du bon fonctionnement des mécanismes clés de RBFT (Section A.3). La section suivante présente la méthodologie utilisée dans cette analyse.

A.1 Méthodologie

Notre analyse théorique consiste à calculer la latence, en cycles, et le débit, en nombre de requêtes par seconde, pour chacun des protocoles de TFB conçus pour être robustes considérés dans le Chapitre 4.

Notre analyse des protocoles de TFB conçus pour être robustes se base sur le coût des opérations de cryptographie et des communications réseau. Concernant le coût des opérations de cryptographie, notre méthodologie est celle détaillée dans la Section 4.3.1.

Concernant le coût des communications réseau, nous mesurons le coût d'envoi (σ) et de réception (ρ) d'un message en fonction de sa taille en octets, l. Les coûts de ces différentes opérations sont mesurés en cycles. Les coûts d'envoi et de réception d'un message correspondent au coût des appels systèmes *send()* et *recv()*. Enfin, de la même façon que nous ne prenons pas en compte le coût de la génération de la réponse au client, nous ne prenons pas en compte le coût de son envoi.

Nous considérons une machine multicœurs. Chaque cœur a une vitesse d'horloge de κ Hz. Dans le cas d'Aardvark, la vérification des requêtes et l'ordonnancement sont exécutés en parallèle. Dans le cas de RBFT, chaque réplica et chaque module (c.à.d. les modules de vérification, propagation, transit et surveillance et exécution) s'exécutent en parallèle sur des cœurs distincts. Nous ignorons le coût de l'exécution des requêtes car nous souhaitons mesurer le débit maximal (dans le cas de Prime, nous supposons

que ce temps d'exécution est nul dans le cas sans faute et non-nul en cas d'attaque). De plus, nous ne prenons pas en compte le coût du traitement des points de contrôle et des changements de vue. Cela permet de simplifier la modélisation tout en gardant une bonne précision. En outre, nous ne prenons pas en compte le coût de la propagation des messages sur le réseau car ce temps est très faible en comparaison des opérations de cryptographie et de communication : nous utilisons un réseau Gigabit et une taille maximale de messages de 9ko. Toutefois, nous prenons en compte le protocole de communication utilisé par les différents protocoles de TFB afin de compter le nombre d'envoi et réception de messages : Prime (pour la communication entre les réplicas) et Spinning utilisent la multidiffusion d'UDP, dans lequel un message à destination de n nœuds est envoyé 1 fois, tandis que Prime (pour la communication avec les clients), Aardvark et RBFT utilisent TCP, dans lequel un message à destination de n nœuds requiert n envois. Enfin, nous ne prenons pas en compte le coût de la propagation des messages sur le réseau car ce temps est très faible en comparaison des opérations de cryptographie et de communication : nous utilisons un réseau Gigabit et une taille maximale de messages de 9ko. Toutefois, nous prenons en compte le protocole de communication utilisé par les différents protocoles de TFB afin de compter le nombre d'envoi et réception de messages : Prime (pour la communication entre les réplicas) et Spinning utilisent la multidiffusion d'UDP, dans lequel un message à destination de n nœuds est envoyé 1 fois, tandis que Prime (pour la communication avec les clients), Aardvark et RBFT utilisent TCP, dans lequel un message à destination de n nœuds requiert n envois.

La Table A.1 récapitule ces différents coûts, exprimés en cycles. Le modèle utilisé pour chacune des opérations prend en compte la taille des messages l et se compose d'un coût fixe C_f et d'un coût par octet C_v. La formule générique permettant de calculer le coût d'une opération D en cycles est $D(l) = C_f + C_v * l$. Ce modèle a déjà été utilisé dans l'analyse de PBFT [14]. Pour chacune des opérations nous avons mesuré les coûts fixes et par octet de manière expérimentale, sur les machines présentées dans la Section 4.4.2. Cette modélisation s'est révélée imprécise dans le cas des MACs. Étant donné que dans les implantations d'Aardvark, Spinning et RBFT les MACs sont calculés sur 3 tailles de messages seulement, nous avons mesuré le coût, en nombre de cycles, pour ces trois tailles de messages seulement : 16, 72 et 88 octets. Le coût pour chacune de ces tailles est respectivement 644, 795 et 827 cycles.

A.2 Formules de débit et latence des protocoles conçus pour être robustes

Nous présentons dans cette section les formules de débit et de latence pour Prime, Aardvark, Spinning et RBFT dans le cas sans faute et en cas d'attaque. Pour chacun de ces protocoles, l'attaque correspond à l'attaque présentée dans la Section 4.1 (Section 4.4.5 dans le cas de RBFT).

A.2.1 Prime

Soient E le temps nécessaire pour exécuter un lot de b requêtes et P la période d'envoi des messages PRÉ-PRÉPARATION. La latence et le débit de Prime dans le cas

	Variable	Valeur
Génération/vérification d'un MAC	α	entre 644 et 827 cycles
Vérification d'une signature	Θ	$11221 + 5.8 * l$ cycles
Création d'une empreinte	δ	$1907 + 23 * l$ cycles
Envoi d'un message	σ	$643 + 19 * l$ cycles
Réception d'un message	ρ	$619 + 20 * l$ cycles
Taille des lots de requêtes	b	1
Vitesse du processeur	κ	2,4GHz

TABLE A.1 – Liste des variables utilisées dans les formules. Le coût des opérations de cryptographie et de réseau est pour des messages de taille l octets. Dans les implantations des protocoles robustes, les MACs sont calculés sur 3 tailles de messages uniquement : 16, 72 et 88 octets. Le coût pour chacune de ces tailles est respectivement 644, 795 et 827 cycles.

sans faute sont donnés par les formules suivantes :

$$L_{sf} = \rho + 2\theta + \sigma + \frac{(11f+5)\theta + (6f+2)\delta + (n+5)\sigma + (n+8f+5)\rho + P}{b} + E$$

$$T_{sf} = \frac{\kappa}{L_{sf}}$$

Soient E_{sf} le temps nécessaire pour exécuter un lot de b requêtes dans le cas sans faute, et $E_{attaque}$ le temps nécessaire pour exécuter un lot de b requêtes lors de l'attaque, lorsqu'une requête met davantage de temps à s'exécuter. Le mécanisme de surveillance de Prime calcule le temps de réponse maximal $TAR_{sf} = (2L_{opportun} + E_{sf})K_{Lat} + pp$ dans le cas sans faute et $TAR_{attaque} = (2L_{opportun} + E_{attaque})K_{Lat} + pp$ lors de l'attaque. Un principal malicieux peut alors retarder les PRÉ-PRÉPARATION de $TAR_{attaque} - TAR_{sf}$. Par conséquent, la latence et le débit de Prime en cas d'attaque sont :

$$L_{attaque} = L_{sf} + TAR_{attaque} - TAR_{sf}, \qquad T_{attaque} = \frac{\kappa}{L_{attaque}}$$

La Table A.2 recense les variables spécifiques à Prime et les valeurs que nous avons utilisé. La parte de débit théorique sous une charge statique est calculée à partir de la

E_{sf}	0,1ms
$E_{attaque}$	10ms
$L_{opportun}$	0,78ms
K_{Lat}	5
pp	1ms

TABLE A.2 – Variables de Prime et valeurs utilisées lors de notre analyse théorique.

différence en pourcentage entre $T_{attaque}$ et T_{sf}.

A.2.2 Aardvark

La latence et le débit d'Aardvark dans le cas sans faute sont donnés par les formules suivantes :

$$L_{sf} = max(\theta + \alpha + \delta + \rho, \frac{\alpha(10f + b) + \delta + \sigma(2n + b) + \rho(4f + 2)}{b})$$

$$T_{sf} = \frac{\kappa}{L_{sf}}$$

Nous rappelons que les réplicas s'attendent à recevoir au moins P PRÉ-PRÉPARATION toutes les M ms. Nous devons considérer deux cas avec attaques. Dans le premier cas, la charge est statique. Dans ce cas, un principal malicieux est en place au maximum $L_{max} = \frac{\kappa * P}{1000 * M}$ cycles. Les performances moyennes du protocole sont donc :

$$L_{attaque} = \frac{((n - f) * L_{sf} + f * L_{max})}{n}, \qquad T_{attaque} = \frac{\kappa}{L_{attaque}}$$

La parte de débit théorique sous une charge statique est calculée à partir de la différence en pourcentage entre $T_{attaque}$ et T_{sf}.

Dans le second cas, lorsque la charge est dynamique, la formule est plus complexe. En effet, nous devons prendre en compte le calcul du débit attendu. La latence durant une attaque avec charge dynamique est calculée par la formule $L_{attaque} = L_{sf} + A_{delai}(obs, exp)$, avec $A_{delai}(obs, exp)$ une formule qui, à partir du débit observé obs et du débit requis exp, calcule le délai maximal que le principal malicieux peut appliquer aux messages d'ordonnancement sans être détecté. Ce délai est calculé de la manière suivante. Soit u le temps écoulé (en ms) depuis la création du dernier point de contrôle, et r le nombre de requêtes qu'il a ordonné depuis le dernier point de contrôle. Alors le délai d qu'il peut appliquer doit vérifier $\frac{r+b}{u+d} \geq exp$. Par conséquent, $d = \frac{r+b}{exp} - u$. De plus, le principal doit envoyer au moins P PRÉ-PRÉPARATION toutes les M ms. Étant donné ces deux conditions, le principal malicieux peut retarder le prochain PRÉ-PRÉPARATION de $d = min(\frac{r+b}{exp} - u, \frac{t}{n}) - \lambda$ ms, avec λ une constante qui prend en compte le temps de communication entre l'envoi d'un PRÉ-PRÉPARATION et sa réception [1]

La perte de débit théorique sous une charge dynamique est calculée à partir de la différence en pourcentage entre l'intégrale de la courbe du débit en cas d'attaque et l'aire du rectangle formé par la courbe dans le cas sans faute.

A.2.3 Spinning

La latence et le débit de Spinning dans le cas sans faute sont donnés par les formules suivantes :

$$L_{sf} = \frac{\alpha(13f + 2b) + \delta(b + 1) + \sigma(3 + b) + \rho(4f + b + 3)}{b}$$

1. Cette constante dépend de la puissance de calcul des machines et du réseau. Dans nos expériences, $\lambda = 10$ ms.

$$T_{sf} = \frac{\kappa}{L_{sf}}$$

Soit S_t le temps maximal avant que le minuteur de surveillance du principal n'expire, le considère comme malicieux et le mette sur liste noire, alors le débit de Spinning en cas d'attaque est donné par la formule suivante :

$$T_{attaque} = \frac{(n-f) * T_{sf} + \frac{f\kappa}{(L_{sf}+S_t)}}{n}$$

De manière intuitive, cette formule correspond au débit moyen lorsque f réplicas sont malicieux et retardent l'ordonnancement des requêtes de S_t.

La perte de débit théorique de Spinning est calculée à partir de la différence en pourcentage entre $T_{attaque}$ et T_{sf}, lorsque $S_t = 40ms$. Cette valeur a été utilisée par les auteurs de Spinning dans leur évaluation [56].

A.2.4 RBFT

La latence et le débit de RBFT dans le cas sans faute sont donnés par les formules suivantes :

$$L_{sf} = max(\theta + \alpha(n+f) + \delta + n\sigma + (2f+2)\rho, \alpha\frac{2n+4f-1}{b} + \delta + 2n\sigma + (4f+2)\rho)$$

$$T_{sf} = \frac{\kappa}{L_{sf}}$$

En cas d'attaque, les performances fournies lorsque le principal de l'instance maître n'est pas malicieux sont les suivantes (plus de détails sont présentés dans la Section A.3.1 ; g est la proportion de requêtes valides envoyées par les clients malicieux).

$$L_{incivil} = max(\frac{\theta}{g} + \alpha(\frac{1}{g} + n + f - 1) + \frac{\delta}{g} + n\sigma + \rho(\frac{1}{g} + 2f + 1), \alpha\frac{2n+4f-1}{b} + \delta + 2n\sigma + (4f+2)\rho)$$

$$T_{incivil} = \frac{\kappa}{L_{incivil}}$$

le débit minimal que peut fournir le système est le débit que peut fournir le principal de l'instance maître lorsqu'il est malicieux et lorsqu'il ne dépasse pas le seuil de détection. Plus précisément, le ratio entre son débit $T_{attaque}$ et le débit en cas d'attaque $T_{incivil}$ doit être supérieur ou égal à Δ :

$$\frac{T_{attaque} - T_{incivil}}{T_{attaque}} \geq \Delta \Leftrightarrow T_{attaque} \geq \frac{T_{incivil}}{1-\Delta}$$

Ainsi le débit minimal en cas d'attaque, dans le pire des cas, est :

$$T_{attaque} = \frac{T_{incivil}}{1-\Delta}$$

A.3 Formules théoriques de RBFT

Nous présentons dans cette section une analyse théorique de RBFT. Nous présentons tout d'abord une analyse du débit dans la Section A.3.1. Grâce à cette analyse, nous montrons comment calculer le ratio minimal acceptable entre les performances de l'instance maître et les performances des instances de surveillance, dans la Section A.3.2. Dans la Section A.3.3, nous montrons que $f + 1$ instances de protocoles est nécessaire et suffisant. Nous montrons ensuite que, grâce à la phase de PROPAGATION, tous les nœuds corrects finiront par envoyer la requête à l'ordonnancement (Section A.3.4). Enfin, nous montrons que si $2f + 1$ nœuds corrects décident d'exécuter un changement d'instance de protocole, alors un changement d'instance de protocole va finir par être exécuté par chacun des nœuds corrects (Section A.3.5).

Comme définit par Clement et al. [13], une exécution est *sans faute* si et seulement si (i) l'exécution est synchrone avec une borne faible et dépendante de l'implantation sur le temps de parcours des messages sur le réseau et (ii) tous les clients et les nœuds se comportent de manière correcte. À contrario, une exécution est *incivile* si et seulement si (i) l'exécution est synchrone avec une borne faible et dépendante de l'implantation sur le temps de parcours des messages sur le réseau, (ii) jusqu'à f nœuds et un nombre arbitraire de clients sont Byzantins et (iii) les nœuds et clients restants se comportent de manière correcte.

A.3.1 Analyse du débit

Nous analysons dans cette section le débit de RBFT dans le cas sans faute et lors d'une exécution incivile. Nous calculons les performances maximales de RBFT dans le Théorème 1 et montrons, dans le Théorème 2, que les performances lors d'une exécution incivile sont proches des performances dans le cas sans faute d'un facteur constant. Notons que, dans un souci de clarté, nous ne prenons en compte dans cette partie que le coût des opérations de cryptographie. En effet, les performances de RBFT sont principalement limitées par le coût des opérations de cryptographie.

Théorème 1. Considérons une exécution sans faute dans laquelle le système est saturé, toutes les requêtes proviennent de clients corrects, tous les nœuds sont corrects et les réplicas ordonnent des lots de requêtes de taille b. Pour chaque instance de protocole, le débit est au plus $t_{pic} = \frac{\kappa}{max(\theta+\alpha(n+f),\ \alpha\frac{2n+4f-1}{b})}$ req/s.

Esquisse de preuve. Nous calculons dans un premier temps le nombre d'opérations exécutées par chaque nœud pour envoyer une requête à l'étape d'ordonnancement. Pour une requête, il y a au plus la vérification d'un MAC et d'une signature, faits lorsque la requête est reçue depuis le client, ou lorsque la requête est reçue durant la phase de PROPAGATION. Ensuite, chaque nœud génère $n - 1$ MACs pour créer le message PROPAGATION. Chaque nœud attend pour f tels messages, pour lesquels il vérifie le MAC. Ainsi, le coût pour une requête jusqu'à l'étape d'ordonnancement est $\theta + \alpha(n + f)$.

Nous calculons ensuite le nombre d'opérations exécutées pour ordonner une requête. Nous différencions les opérations effectuées par le principal d'une instance de protocole et par les réplicas non principal. Le principal génère $n - 1$ MACs pour le

PRÉ-PRÉPARATION, tandis qu'un réplica non principal vérifie le PRÉ-PRÉPARATION et génère $n-1$ MACs pour le PRÉPARATION. Ensuite, chaque réplica vérifie $2f$ PRÉPARATION, après quoi il génère $n-1$ MACs pour le VALIDATION, et finalement vérifie $2f+1$ VALIDATION. Par conséquent, pour chaque requête, le coût au niveau du principal est $\alpha\frac{2n+4f-1}{b}$ et le coût au niveau d'un réplica non principal est $\alpha\frac{2n+4f}{b}$.

Étant donné que le module de propagation et les réplicas s'exécutent en parallèle, le débit est au plus $t_{pic} = \frac{\kappa}{max(\theta+\alpha(n+f),\ \alpha\frac{2n+4f-1}{b})}$ req/s. □

Lemme 1. Considérons une exécution incivile durant laquelle le système est saturé. Seule une fraction g des requêtes générées par les clients est correcte. De plus, le nœud sur lequel s'exécute le principal de l'instance maître est correct et au plus f nœuds sont Byzantins. Tous les nœuds reçoivent la même proportion de requêtes correctes et les réplicas ordonnent des lots de requêtes de taille b. Pour chaque instance de protocole, le débit est $t_{incivil} = \frac{\kappa}{max(\frac{\theta}{g}+\alpha(\frac{1}{g}+n+f-1),\ \alpha\frac{2n+4f-1}{b})}$ req/s.

Esquisse de preuve. Nous calculons le nombre d'opérations effectuées par chaque nœud pour envoyer une requête à l'étape d'ordonnancement. Notons que, grâce au mécanisme de liste noire (cf. Section 4.2.5), un nœud qui envoie trop de messages invalides ou trop de messages est isolé du système. Pour une requête correcte, il y a au plus une vérification de MAC et une vérification de signature. Pour envoyer un PROPAGATE, $n-1$ MACs sont générés. Chaque nœud correct attend au plus f PROPAGATE, pour lesquels il vérifie le MAC.

Le coût pour une requête jusqu'à la phase d'ordonnancement est $\frac{\theta}{g}+\alpha(\frac{1}{g}+n+f-1)$. Le coût de l'étape d'ordonnancement est le même que dans le cas sans faute : il est, au niveau du principal, de $\alpha\frac{2n+4f-1}{b}$ et, au niveau d'un réplica non principal, de $\alpha\frac{2n+4f}{b}$.

Par conséquent, le débit est $t_{incivil} = \frac{\kappa}{max(\frac{\theta}{g}+\alpha(\frac{1}{g}+n+f-1),\ \alpha\frac{2n+4f-1}{b})}$ req/s. □

Théorème 2. Considérons une exécution incivile durant laquelle le système est saturé. Seule une fraction g des requêtes générées par les clients est correcte. De plus, le nœud sur lequel s'exécute le principal de l'instance maître est correct et au plus f nœuds sont Byzantins. Tous les nœuds reçoivent la même proportion de requêtes correctes et les réplicas ordonnent des lots de requêtes de taille b. Alors les performances sont un facteur constant $C \leq 1$ des performances d'une exécution dans le cas sans faute dans laquelle la taille des lots de requêtes est la même.

Esquisse de preuve. Le Théorème 1 et le Lemme 1 nous donnent les performances lors d'une exécution sans faute et durant une exécution incivile. De plus, étant donné que les opération sur les signatures sont beaucoup plus coûteuses que les opérations sur les MACs, le ratio $\frac{\alpha}{\theta}$ tend vers 0. Soient $v1 = \frac{\theta/\alpha+1-1/b}{10/b-4}$ et $v2 = \frac{\theta/(g\alpha)+1/g-1/b}{10/b-4}$ ($v1 \leq v2$). D'après les formules de t_{pic} et $t_{incivil}$, soient $A = \theta+\alpha(n+f)$, $B = \alpha\frac{2n+4f-1}{b}$ et $C = \frac{\theta}{g}+\alpha(\frac{1}{g}+n+f-1)$. Nous devons distinguer trois cas [2] :

1. $A > B$ et $C > B$. Cela se produit lorsque $f < v_1$. Dans ce cas, le ratio entre le débit en cas d'attaque et sans faute $\frac{t_{incivil}}{t_{pic}}$ est $C = g$.

2. La quatrième combinaison, $A \leq B$ et $C > B$, ne peut pas se produire étant donné que cela signifierait $v_1 > f > v_2$.

2. $A > B$ et $C \leq B$. Cela se produit lorsque $f \geq v_1$ et $f < v_2$. Dans ce cas, le ratio entre le débit en cas d'attaque et sans faute $\frac{t_{incivil}}{t_{pic}}$ tend vers $C = 0$.

3. $A \leq B$ et $C \leq B$. Cela se produit lorsque $f \geq v_2$. Dans ce cas, le ratio entre le débit en cas d'attaque et sans faute $\frac{t_{incivil}}{t_{pic}}$ est $C = 1$.

□

A.3.2 Surveillance des instances de protocole

Nous définissons dans cette section la manière de calculer le ratio minimal acceptable entre les performances de l'instance maître et les performances des instances de surveillance (Définition 1). Nous prouvons également qu'un changement d'instance de protocole est initié si le débit fournit par l'instance maître est inférieur au débit durant une exécution incivile (Lemme 2).

Définition 1 Nous avons vu dans la Section 4.2.3 que le débit de l'instance maître est comparé avec le débit maximal des instances de surveillance. Si la différence est plus grande que Δ, alors les nœuds initient un changement d'instance de protocole.

Plus précisément, les nœuds surveillent le débit des différentes instances. Ils calculent le ratio entre le débit de l'instance maître (noté t_{maitre}) et le débit maximal des instances de surveillance (noté $t_{surveillance}$). Nous savons que la différence maximale telle qu'aucun changement d'instance de protocole est entre le débit durant une exécution sans faute (Théorème 1) et le débit durant une exécution incivile (Lemme 1). En cas d'attaque, ils s'attendent à $\frac{t_{maitre}-t_{surveillance}}{t_{maitre}} < \Delta$. Par conséquent, $\Delta = \frac{t_{incivil}-t_{pic}}{t_{incivil}}$.
□

Lemme 2. Si le principal de l'instance maître de protocole est malicieux et fournit un débit inférieur au débit durant une exécution incivile, alors il est détecté et un changement d'instance de protocole est initié.

Esquisse de preuve. Soit t_{delai} le débit de l'instance maître. D'après le Lemme 1, le débit des instances de surveillance dans le pire des cas est $t_{incivil}$. D'après la définition 1, un changement d'instance de protocole est initié si $\frac{t_{maitre}-t_{surveillance}}{t_{maitre}} < \Delta$. Nous calculons maintenant t_{delai} : $\frac{t_{maitre}-t_{surveillance}}{t_{maitre}} < \Delta \Leftrightarrow \frac{t_{delai}-t_{incivil}}{t_{delai}} < \frac{t_{incivil}-t_{pic}}{t_{incivil}} \Leftrightarrow t_{incivil}(t_{delai} - t_{incivil}) < t_{delai}(t_{incivil} - t_{pic}) \Leftrightarrow t_{incivil}t_{incivil} > t_{delai}t_{pic}$
$\Leftrightarrow t_{delai} < \frac{t_{incivil}}{t_{pic}}t_{incivil}$.

D'après le Théorème 2, la fraction $\frac{t_{incivil}}{t_{pic}}$ tend vers une constante $C \leq 1$. Ainsi un changement d'instance de protocole est initié si et seulement si $t_{delai} < C * t_{incivil}$. Étant donné que $C \leq 1$, un changement d'instance de protocole est initié si le débit durant l'attaque est inférieur au débit durant une exécution incivile. En d'autres termes, le principal malicieux de l'instance de protocole maître ne peut pas fournir un débit inférieur au débit durant une exécution incivile sans être remplacé. □

A.3.3 Nombre d'instances de protocoles

Dans cette section nous prouvons que $f + 1$ instances de protocole est nécessaire et suffisant (Théorème 3).

Lemme 3. Une instance de protocole ne peut pas fournir un débit plus élevé que le débit observé dans le cas sans faute tant que le nombre de fautes simultanées est au plus égal à f.

Esquisse de preuve. Le Théorème 1 nous apprend que le débit dépend du nombre d'opérations de cryptographie : génération et vérification de MACs et vérification de signatures. Afin de fournir un débit plus élevé que le débit maximal dans le cas sans faute, une instance de protocole doit effectuer moins d'opérations de cryptographie. Toutefois, la formule de t_{pic} considère le cas lorsque, avec au plus f réplicas malicieux (c.à.d. lorsque la propriété de sûreté est valide). Par conséquent, une instance de protocole ne peut fournir un débit supérieur au débit dans le cas sans faute. t_{pic}. □

Invariant 1. Considérons une exécution incivile dans laquelle le système est saturé et au plus f nœuds sont Byzantins. Tant que le principal de l'instance maître est correct, alors un changement d'instance de protocole n'arrivera jamais.

Esquisse de preuve. Nous considérons une exécution incivile, c.à.d. qu'une proportion g des requêtes est correcte et au plus f nœuds sont Byzantins. De plus, le principal de l'instance maître est correct. Nous supposons qu'un changement d'instance de protocole va à terme se produire et montrons que nous arrivons à une contradiction.

Un changement d'instance de protocole se produira à terme signifie qu'il y aura à terme au moins $2f+1$ nœuds qui vont recevoir $2f+1$ messages CHANGEMENT_INSTANCE. Il y a au plus f nœuds Byzantins, qui peuvent envoyer des messages CHANGEMENT_INSTANCE valides en continu. Ainsi il est suffisant que $f + 1$ nœuds corrects envoient un message CHANGEMENT_INSTANCE. Pour cela, ils doivent observer de meilleures performances au niveau des instances de surveillance qu'au niveau de l'instance maître par un seuil pré-défini Δ (cf. Définition 1). Plus précisément, les nœuds malicieux doivent augmenter les performances des instances de surveillance et réduire les performances de l'instance maître.

Nous considérons l'attaque suivante, qui est celle dans laquelle la différence entre le débit de l'instance maître et le débit des instances de surveillance est maximale. Les clients malicieux envoient une proportion $g < 1$ de requêtes correctes à l'instance maître seulement et une proportion $g = 1$ de requêtes correctes aux instances de surveillance. De plus, les nœuds malicieux attaquent le nœud sur lequel s'exécute le principal de l'instance maître par déni de service par inondation (ils n'attaquent pas les réplicas des autres instances de protocole qui s'exécutent sur ce nœud). Ainsi le débit de l'instance maître, noté t_{maitre}, est donné par le Lemme 1 et le débit maximal des instances de surveillance, noté $t_{surveillance}$, est donné par le Théorème 1.

Un message CHANGEMENT_INSTANCE est envoyé par un nœud correct si le ratio entre t_{maitre} et $t_{surveillance}$ est inférieur à Δ. Comme montré dans la Définition 1, Δ est précisément calculé comme le ratio entre le débit dans le cas sans faute et le débit dans une exécution incivile. De plus, d'après le Lemme 3, $t_{surveillance} \leq t_{pic}$. Par

conséquent aucun messages CHANGEMENT_INSTANCE n'est jamais envoyé, ce qui contredit notre postulat de départ. □

Invariant 2. Considérons une exécution incivile dans laquelle le système est saturé et au plus f nœuds sont Byzantins. Si le principal de l'instance maître est Byzantin et si ses performances ne sont pas acceptables (c.à.d. que le ratio entre son débit et le débit maximal des instances de surveillance est inférieur à Δ ; cf. Définition 1), alors un changement d'instance de protocole finira par se déclencher.

Esquisse de preuve. Nous considérons une exécution incivile : une proportion g de requêtes sont correctes et au plus f nœuds sont Byzantins. De plus, le principal de l'instance maître est malicieux. Nous supposons qu'un changement d'instance de protocole ne se produira jamais et montrons que nous arrivons à une contradiction.

Nous considérons l'attaque suivante, qui est celle dans laquelle la différence entre les performances de l'instance maître et des instances de surveillance est maximale. Les clients envoient une proportion $g < 1$ de requêtes correctes aux instances de surveillance. De plus, les nœuds malicieux attaquent les nœuds corrects par déni de service par inondation. Le débit de l'instance maître, noté t_{maitre}, est le débit résultant de l'attaque ; le débit fournit par les autres instances pour lesquelles le principal est malicieux est $t_{incivil}$ (cf. Lemme 1). Le nombre de réplica par nœud est $f + 1$. Par conséquent, il y a f instances de surveillance, parmi lesquelles au plus $f - 1$ sont malicieuses et au moins une est correcte. Étant donné que $t_{maitre} < t_{incivil}$, le débit maximal des instances de surveillance est au moins égal à $t_{incivil}$.

Un message CHANGEMENT_INSTANCE est envoyé par un nœud correct si le ratio entre t_{maitre} et $t_{incivil}$ est inférieur à Δ (cf. Définition 1). Plus précisément, si le ratio entre t_{maitre} et $t_{incivil}$ est plus petit que le ratio entre le débit durant une exécution incivile et une exécution sans faute. D'après le Lemme 2, si $t_{maitre} < t_{incivil}$, alors tous les nœuds corrects vont détecter l'attaque et vont envoyer un message CHANGEMENT_INSTANCE. Par conséquent, chaque nœud correct va à terme voter pour un changement d'instance de protocole. Cela contredit notre hypothèse selon laquelle un changement d'instance de protocole ne se produira jamais. □

Théorème 3. Considérons une exécution incivile dans laquelle le système est saturé et au plus f nœuds sont Byzantins. Dans ce cas, $f + 1$ instances de protocole sont nécessaires et suffisantes pour que les nœuds et les clients malicieux ne puissent pas altérer les performances des différentes instances tel que les performances fournies par le système ne soient pas adéquates.

Esquisse de preuve. À partir des Invariants 1 et 2, nous savons que si le nombre d'instances de protocole est de $f + 1$, alors les nœuds et les clients malicieux ne peuvent forcer un changement d'instance de protocole ou altérer les performances du système au-delà d'un seuil toléré (définit par le ratio Δ) sans déclencher un changement d'instance de protocole. Nous montrons maintenant que l'un des invariants est violé lorsque le nombre d'instances de protocole est inférieur à $f + 1$.

Nous considérons f instances de protocole et l'attaque présentée dans la preuve de l'Invariant 2 et montrons que cet invariant est violé, c.à.d. que le principal de l'instance maître n'est pas détecté. La preuve est triviale : si l'on considère f nœuds

malicieux, alors il est possible de trouver une exécution dans laquelle le principal de chaque instance de protocole est malicieux. Par conséquent, ils peuvent tous fournir un débit nul. Dans ce cas le principal de l'instance maître n'est pas détecté, ce qui viole l'Invariant 2. □

A.3.4 Phase de propagation

Dans cette section nous prouvons que, grâce à la phase de PROPAGATION, tous les nœuds corrects envoient à terme les requêtes valides à la phase d'ordonnancement.

Invariant 3. Si un nœud correct envoie un message PROPAGATION, alors chaque nœud va à terme recevoir $f + 1$ messages PROPAGATION.

Esquisse de preuve. Par la contraposée : nous supposons qu'il existe un nœud correct n_1 qui envoie un message PROPAGATION à tous les nœuds, et qu'il existe un nœud correct n_2 qui ne recevra jamais $f + 1$ messages PROPAGATION. Nous montrons que nous arrivons à une contradiction.

Le nœud n_1 envoie un message PROPAGATION à tous les nœuds. Ce message sera à terme reçu par tous les nœuds corrects, étant donné que le système est *ultimement synchrone* et que les messages de PROPAGATION sont retransmis à tous les nœuds tant que la requête n'a pas été exécutée. Soit n_3 un nœud correct qui reçoit le message PROPAGATION depuis n_1. Nous devons distinguer trois cas :

1. il a déjà reçu la requête. D'après la spécification du protocole (cf. Section 4.2.2), il a déjà envoyé un message PROPAGATION à tous les nœuds ;
2. il n'a jamais reçu la requête et a déjà reçu un message PROPAGATION valide pour cette requête. D'après la spécification du protocole, il a déjà envoyé un message PROPAGATION à tous les nœuds ;
3. il n'a jamais reçu la requête ni de message PROPAGATION valide pour cette requête. D'après la spécification du protocole, il envoie un message PROPAGATION à tous les nœuds.

Dans chacun de ces cas le nœud n_3 envoie un message PROPAGATION.

Étant donné qu'il y a au moins $2f + 1$ nœuds corrects, au moins $2f + 1$ PROPAGATION valides sont envoyées et finalement reçus par chacun des nœuds corrects. Ainsi, chaque nœud correct, dont n_2, recevra à terme au moins $2f + 1 > f + 1$ messages PROPAGATION. Cela contredit l'hypothèse de départ qui spécifie que n_2 ne recevra jamais $f + 1$ messages PROPAGATION. □

Invariant 4. Si une requête est envoyée à l'ordonnancement par un nœud correct, alors elle sera à terme envoyée à l'ordonnancement par tous les nœuds corrects.

Esquisse de preuve. Par la contraposée : nous supposons qu'une requête est envoyée à l'ordonnancement par un nœud correct n_1 et qu'il existe un nœud correct n_2 qui n'enverra jamais la requête. Nous montrons que nous arrivons à une contradiction.

Dire que le nœud n_2 ne va jamais envoyer la requête à l'ordonnancement revient à dire qu'il ne va jamais recevoir $f + 1$ messages PROPAGATION valides.

Dire que le nœud n_1 envoie la requête à l'ordonnancement revient à dire qu'il a reçu $f + 1$ messages PROPAGATION valides, ce qui signifie qu'il il a $f + 1$ nœuds (lui

inclus) qui ont envoyé un message PROPAGATION valide. D'après la spécification du protocole, un nœud correct qui reçoit un message PROPAGATION valide ou une requête valide envoie un message PROPAGATION. Par conséquent il existe un nœud correct (p. ex. n_1) qui a envoyé un message PROPAGATION valide. D'après l'Invariant 3, si un nœud correct envoie un message PROPAGATION valide, alors chaque nœud correct va à terme recevoir $f + 1$ messages PROPAGATION.

Ainsi, n_2, qui est correct, va à terme recevoir $f + 1$ messages PROPAGATION. Cela contredit l'hypothèse de départ, qui est que n_2 ne recevra jamais $f + 1$ messages PROPAGATION. □

A.3.5 Changement d'instance de protocole

Nous prouvons dans cette section que si $2f + 1$ nœuds corrects décident d'effectuer un changement d'instance de protocole, alors un changement d'instance de protocole sera finalement effectué par chacun des nœuds corrects.

Invariant 5. Si $2f + 1$ nœuds corrects décident d'effectuer un changement d'instance de protocole, alors un changement d'instance de protocole sera à terme effectué par chacun des nœuds corrects.

Esquisse de preuve. Par la contraposée : nous supposons que $2f + 1$ nœuds corrects décident d'effectuer un changement d'instance de protocole et qu'il existe un nœud qui n'en effectuera jamais un. Nous montrons que nous arrivons à une contradiction.

D'après l'algorithme présenté dans la Section 4.2.4, dire que $2f + 1$ nœuds corrects décident d'effectuer un changement d'instance de protocole revient à dire que $2f + 1$ nœuds envoient un message CHANGEMENT_INSTANCE valide à tous les nœuds. Ainsi, chaque nœud correct va à terme recevoir $2f + 1$ messages CHANGEMENT_INSTANCE valides.

Nous supposons qu'il existe un nœud correct qui ne va jamais effectuer un changement d'instance de protocole. Cela signifie qu'il existe un nœud qui ne va jamais recevoir $2f + 1$ messages CHANGEMENT_INSTANCE valides. Cela contredit le fait que chaque nœud correct va à terme recevoir $2f + 1$ messages CHANGEMENT_INSTANCE valides. □