Vers une prise en charge des comportements rationnels dans les systèmes distribués

Amadou Diarra

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Amadou Diarra

Thèse dirigée par **Vivien Quéma**
et coencadrée par **Sonia Ben Mokhtar**

préparée au sein de **LIG**
et de **Ecole Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Vers une prise en charge des comportements rationnels dans les systèmes distribués

Thèse soutenue publiquement le **23 Septembre 2015**,
devant le jury composé de :

Prof. Didier Donsez
Professeur à l'Université de Grenoble 1, Président
Prof. David Bromberg
Professeur à l'Université de Rennes 1, Rapporteur
Dr. Gilles Muller
Directeur de Recherche à l'Inria, Rapporteur
Dr. Sébastien Monnet
Maître de Conférences à l'Université Pierre et Marie Curie, Examinateur
Prof. Vivien Quéma
Professeur á Grenoble INP, Directeur de thèse
Dr. Sonia Ben Mokhtar
Chargée de Recherche au CNRS, Co-Encadrante de thèse

Résumé

De nos jours, la notion de responsabilité dans un système distribué est devenue quasiment incontournable dans les techniques de détection de fautes. Elle permet non seulement de détecter les fautes mais aussi de fournir des preuves de dysfonctionnement contre les nœuds fautifs dans un système distribué. Les nœuds dits rationnels, c'est-à-dire des nœuds qui essayent de tirer profit du système en maximisant leur bénéfice sans y contribuer, en sont un exemple. Dans la littérature, il existe deux types de solutions exploitant cette notion : les solutions spécifiques et les solutions génériques.
Les solutions spécifiques sont relatives à un type de système distribué donné et se construisent en tenant compte de la structure du système et de l'application qui s'y exécute. Les solutions génériques quant à elles, sont indépendantes du système.
Dans cette thèse nous nous intéressons au second type de solutions c'est à dire les solutions génériques. Dans cette classe de solutions, il existe deux approches pour mettre en place la notion de responsabilité : l'approche matérielle et l'approche logicielle. Actuellement le seul protocole logiciel, générique qui permet d'assurer la notion de responsabilité dans un système distribué, est le protocole *PeerReview*. Ce protocole n'est basé sur aucune configuration matérielle. Cependant, il n'est pas robuste aux comportements dits rationnels au sein de ses propres étapes.

Notre objectif est de fournir une solution logicielle sous-jacente renforçant la notion de responsabilité au niveau d'une application qui s'exécute sur un système distribué en présence de nœuds rationnels.
Pour ce faire nous proposons *FullReview* un protocole qui se base sur la théorie des jeux pour motiver et forcer les nœuds rationnels à suivre les différentes étapes, non seulement au niveau de son propre protocole mais aussi au niveau de l'application qu'il surveille. En outre, *FullReview* utilise l'architecture classique d'un système responsable, qui associe à chaque nœud un ensemble de nœuds appélés moniteurs ou surveillants, et ayant un rôle de surveillance périodique du nœud en question.
Nous prouvons théoriquement que notre protocole est un équilibre de Nash, c'est-à-dire que les nœuds rationnels n'ont aucun intérêt à dévier du protocole.

Ce genre de protocole étant coûteux en terme d'échanges de messages, nous nous sommes intéressés à l'étude théorique des différentes techniques de gestion des moniteurs ou surveillants. L'objectif de cette étude est d'identifier les conditions sur les paramètres du protocole pour lesquelles une méthode de gestion convient mieux qu'une autre.

De plus nous évaluons notre protocole en l'appliquant à deux applications largement utilisées : *SplitStream*, un protocole efficace pour la multi-diffusion de flux vidéo et *Onion Routing*, le protocole de communication anonyme le plus utilisé. Les résultats montrent que *FullReview* détecte efficacement les comportements rationnels avec un

faible surcoût comparé au protocole *PeerReview* et passe à l'echelle comme ce dernier.

Mots-clés. Système distribué, tolérance aux fautes, notion de responsabilité, nœud rationnel, théorie des jeux.

Abstract

Accountability is becoming increasingly required in today's distributed systems. It allows not only to detect faults but also to build provable evidence about the misbehaving nodes in a distributed system. Rational nodes that aim at maximising their benefit without contributing their fair share to the system, are an example. In the literature, there exists two types of solutions that exploit accountability : specific solutions and generic solutions.
Specific solutions are related to a given type of distributed system and are built by taking into account the structure of the system and the running application. As for generic solutions, they are independent to the system.
In this thesis we consider the second type of solutions i.e., generic solutions. There exists two approaches in this class of solutions : hardware approach and software approach. Nowadays the only software and generic protocol that allows to enforce accountability in a distributed system is *PeerReview* protocol. This protocol is not based on any hardware configuration. However, it is not robust to rational behaviour in its own steps.

Our objective is to provide a generic software solution to enforce accountability on any underlying application that running on a distributed system in presence of rational nodes.
To reach this goal we propose *FullReview* a protocol that uses game theory to motivate and force rational participants to follow different steps, not only in its own protocol but also in the application that it monitors. Moreover *FullReview* uses the classical architecture of an accountable system. This architecture assigns to each node in the system, a set of nodes called monitors. Periodically each node is monitored by its set of monitors.
We theoretically prove that our protocol is a Nash equilibrium, i.e., nodes do not have any interest in deviating from it.

This kind of protocol being costly in terms of messages exchanged, we are interested to the theoretic study of different techniques of monitors management. The objective

of this study is to identify conditions on protocol parameters for which a method of management is more appropriate than another.

Furthermore, we practically evaluate *FullReview* by deploying it for enforcing accountability in two applications : (1) SplitStream, an efficient multicast protocol for live streaming, and (2) Onion routing, the most widely used anonymous communication protocol. Performance evaluation shows that *FullReview* effectively detects faults in presence of rational nodes while introducing a small overhead compared to PeerReview and scaling as *PeerReview*.

Keywords. Distributed system, fault tolerance, accountability, rational node, game theory.

Remerciements

Un tel travail n'aurait pu aboutir seul sans encadrement, ainsi mes remerciements vont tout d'abord à l'endroit de Vivien Quéma, professeur à Grenoble INP qui m'a encadré durant ces années de thèse. Je tiens à exprimer mes vifs remerciements à Sonia Ben Mokhtar, chargée de recherche au CNRS, qui m'a encadré depuis Lyon. Elle fut pour moi une encadrante attentive et disponible malgré son emploi du temps chargé.
J'ai beaucoup appris de ces deux encadrants très compétents et ce fut un réel plaisir de travailler avec eux.

Je tiens à remercier également les membres du jury notamment Didier Donsez, professeur à l'Université de Grenoble 1, de m'avoir fait l'honneur de présider ce jury ; David Bromberg professeur à l'Université de Rennes 1 et Gilles Muller, Directeur de Recherches à l'INRIA, d'avoir accepté d'être rapporteurs de cette thèse ; Sébastien Monnet maître de conférences à l'Université Pierre et Marie Curie, d'avoir accepté d'être examinateur de cette thèse.

J'aimerais également remercier tous les membres et anciens membres de l'équipe ERODS en particulier mes camarades thésards Jérémie, Vincent, Soguy, Ahmed, Brahim et Lucas.
Je ne pourrais terminer ces remerciements sans penser aux membres de l'équipe DRIM de l'INSA de Lyon dans laquelle j'ai passé un an. Je remercie en particulier Thomas et Tarek, mes collègues de bureau, pour nos moments de détente et nos discussions très enrichissantes.
Mes remerciements vont également à l'endroit de mes amis qui m'ont soutenu durant toutes ces années de thèse.

Enfin je tiens à remercier ma famille depuis le Mali pour leur soutien moral et leurs encouragements. En particulier je remercie mon père Gaoussou Diarra, ma mère Oumou Traoré, mes frères Abdoulaye Diarra et Mamadou Diarra qui m'ont toujours accompagné avec des conseils et n'ont ménagé aucun effort pour ma réussite.

Préface

Ce document est le fruit de trois années de recherches menées dans le Laboratoire d'Informatique de Grenoble au sein de l'équipe ERODS(Efficient RObuste Distributed Systems). L'objectif principal de ces travaux est l'obtention du grade de docteur 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 conduites sous la direction du Pr. Vivien Quéma (LIG ERODS/Grenoble INP) et Dr. Sonia Ben Mokhtar (LIRIS DRIM/INSA Lyon).

Cette thèse se situe dans le domaine de la tolérance aux fautes dans les systèmes distribués, en particulier, elle s'intéresse à la notion de responsabilité dans ces systèmes en présence des comportements dits rationnels qui sont sources de beaucoup de problèmes (dysfonctionnement, baisse de performance...) dans les systèmes distribués. Ainsi la notion de responsabilité s'avère être efficace pour faire face à ces problèmes.

Cette thèse a donné lieu à deux publications dans des conférences internationales de rang élevé et une soumission en cours dans un journal international.

1 FullReview : Practical Accountability in Presence of Selfish Nodes. (SRDS 2014)
2 RAC : a freerider-resilient, Scalable, Anonymous Communication Protocol. (ICDCS 2013)
3 FullReview : Practical Accountability in Presence of Selfish Nodes. (Elsevier)

Terminologie

Dans ce manuscript certains termes sont utilisés pour faciliter la compréhension. Nous présentons ces termes et leur signification dans le tableau I.

Terme	Signification
système responsable	système utilisant la notion de responsabilité
BAR	modèle Byzantin, Altruiste, Rationnel
send(...)	désigne l'envoi d'un message
fwd(...)	désigne le transfert d'un message
$Audit\_req$	envoi d'une requête d'audit
$Audit\_resp$	réponse à une requête d'audit
$Fwd\_outcome$	transfert d'une réponse à une requête d'audit
Challenge(...)	mise au défi d'un nœud
GetState(...)	demande de l'état d'un nœud
SendState(...)	envoi de l'état d'un nœud

TABLE I – Termes utilisés dans ce manuscript et leur signification.

Table des matières

Résumé **i**

Remerciements **v**

Préface **vii**

Terminologie **ix**

Table des matières **xi**

1 Introduction **1**
1.1 Contexte 1
1.2 Problématique 2
1.3 Contributions 3
1.4 Organisation du document 3

2 Tolérance aux fautes dans les systèmes distribués **5**
2.1 Définitions 5
2.2 Types de fautes 6
2.3 Les différentes approches 7
2.3.1 Détection 7
2.3.2 Tolérance 8
2.4 Conclusion 8

3 Rationalité dans les systèmes distribués **9**
3.1 Qu'est ce qu'un comportement rationnel ? 9
3.2 Cas d'un système P2P 10
3.3 Modèle BAR 11
3.3.1 Définition 11
3.3.2 Description 11
3.3.3 Limitations 13
3.4 Conclusion 14

4 Notion de responsabilité dans les systèmes distribués 15
4.1 Définition . . . 16
4.2 Construction d'un système responsable . . . 16
4.2.1 Architecture d'un système responsable . . . 16
4.2.2 Historique sécurisé . . . 17
4.3 État de l'art . . . 18
4.3.1 Solutions spécifiques . . . 18
4.3.2 Solutions génériques . . . 22
4.4 Conclusion . . . 26

5 PeerReview : Une solution logicielle générique assurant la notion de responsabilité dans un système distribué 29
5.1 Description . . . 30
5.1.1 Modèle de fautes . . . 30
5.1.2 Modèle de système . . . 30
5.1.3 Hypothèses . . . 30
5.1.4 Propriétés . . . 31
5.1.5 Sous-protocoles . . . 31
5.2 Déviations rationnelles dans *PeerReview* . . . 34
5.3 Impact des nœuds rationnels . . . 35
5.4 Conclusion . . . 37

6 FullReview : La notion de responsabilité en présence de comportements rationnels 39
6.1 Appliquer *PeerReview* à lui même . . . 40
6.2 Modèle du système . . . 40
6.3 Modèle de fautes . . . 41
6.4 Hypothèses . . . 42
6.5 Vue d'ensemble du protocole . . . 42
6.6 Description détaillée . . . 43
6.6.1 Protocoles de vérification d'historique . . . 44
6.6.2 Protocoles de détection de fautes . . . 48
6.7 Résistance aux comportements rationnels . . . 50
6.7.1 Protocole d'audit . . . 50
6.7.2 Protocole P augmenté . . . 51
6.7.3 Envoi des requêtes d'audit . . . 51
6.7.4 Traitement des requêtes d'audit . . . 52
6.7.5 Traitement des fautes d'omission . . . 53
6.8 Conclusion . . . 53

7 Gestion des surveillants 55
7.1 Gestion statique . . . 56
7.2 Gestion dynamique . . . 57

7.3 Vers un système sans surveillants 58
7.4 Évaluation 60
7.4.1 SplitStream 60
7.4.2 Onion routing 61
7.5 Conclusion 63

8 Évaluation de performance **67**
8.1 Applications 68
8.1.1 *SplitStream* 68
8.1.2 *Onion routing* 68
8.2 Configurations expérimentales 69
8.3 Performance en présence des nœuds rationnels 69
8.4 Performance dans le cas sans fautes 73
8.5 Passage à l'échelle de *FullReview* 78
8.6 Conclusion 78

9 Conclusion **81**
9.1 Objectif et contributions 81
9.2 Perspectives 83

Bibliographie **85**

Liste des figures **89**

Liste des tables **91**

Annexes **93**

A Pseudo-code des différents sous protocoles de *FullReview* **93**
A.1 Protocole de consistance 93
A.2 Protocole d'audit 94
A.3 Protocole de challenge réponse 96
A.4 Protocole d'omission de pannes 96

1

Introduction

Sommaire

1.1 Contexte 1
1.2 Problématique 2
1.3 Contributions 3
1.4 Organisation du document 3

Nous introduisons dans ce chapitre le contexte général dans lequel se situe ces travaux, la problématique et nos contributions.

1.1 Contexte

Depuis l'apparition des premiers ordinateurs jusque maintenant, l'informatique définie comme étant le traitement automatisé de l'information n'a cessé d'évoluer et est devenue presque incontournable dans tous les secteurs. Les masses d'informations ont également suivi cette évolution, ce qui a permis le passage jadis du traitement séquentiel au traitement distribué aujourd'hui, afin de bénéficier de la disponibilité des ressources et l'efficacité du traitement. C'est ainsi que les systèmes distribués ou répartis ont vu le jour pour répondre à ces critères (disponibilité, efficacité, etc). Des exemples de tels systèmes sont : les systèmes pair à pair (P2P) [1], les grilles de calcul [2], les service réseaux (DNS) [3], les systèmes de routage [4] etc.
Pour atteindre ces critères, ces systèmes mettent en interaction via un réseau de communication des milliers de machines autonomes souvent dispersées géographiquement et sont constitués de plusieurs domaines d'administration [5] comme illustré sur la figure 1.1. A cette échelle ils (les systèmes distribués) sont sujets à différents types de fautes d'ordre logicielles ou matérielles. Ces fautes peuvent provenir de plusieurs

sources telles que : crash de machines, bugs, mauvaises configurations, attaques malicieuses, utilisateurs malveillants modifiant le comportement des logiciels dans le but d'obtenir des bénéfices etc.
Dans ce contexte les deux questions fondamentales que les experts se posent sont les suivantes : (1) comment tolérer ou masquer ces fautes c'est à dire faire en sorte qu'un système fonctionne malgré la présence de ces fautes ? et (2) comment détecter ces fautes tout en s'assurant du bon fonctionnement du système ? Un nouveau domaine a fait son émergence pour répondre à ces questions : la tolérance aux fautes dans les systèmes distribués [6]. Ce domaine est constitué de deux volets, un premier volet qui s'occupe de la prévention des fautes (tolérer ou masquer des fautes) et un second volet traitant la détection des fautes.
Cette thèse se situe dans le second volet. En effet nous nous sommes intéressés à ce volet car il est peu étudié dans la littérature par rapport au premier et s'avère être moins coûteux du point de vue pratique pour certains types de fautes. Néanmoins il présente un problème principal qui est l'identification des nœuds responsables. Donc dans cette thèse, nous apportons une solution à ce problème basée sur la notion de responsabilité dans les systèmes distribués.

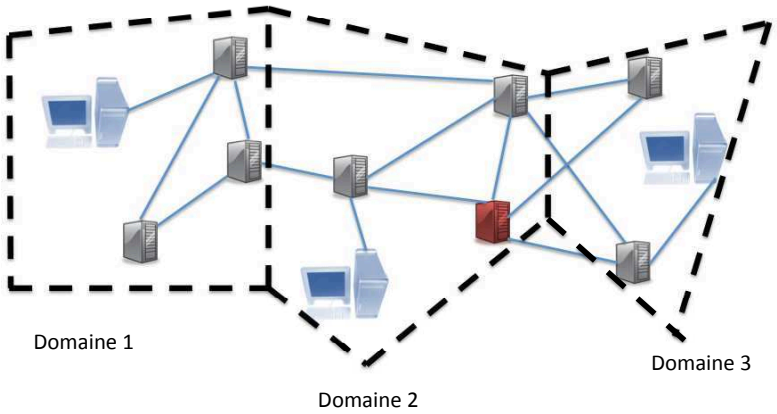


FIGURE 1.1 – Exemple d'un système distribué avec différents domaines d'administration

1.2 Problématique

En plus des fautes précédemment citées à savoir : crash de machines, bugs, mauvaises configurations, attaques malicieuses etc, les systèmes distribués sont sujets à des comportements dits rationnels, nous allons voir cette notion plus en détails dans le chapitre 3. Brièvement un nœud d'un système distribué se comporte de façon rationnelle dans un protocole s'exécutant sur un système distribué, lorsqu'il essaye de tirer profit du système pendant l'exécution du protocole sans y contribuer. Un exemple typique de

ce comportement est lorsque dans un système de partage de vidéo en direct, un nœud refuse de transmettre un morceau de vidéo à un autre nœud pour économiser de la bande passante. Ce comportement cause d'énormes dégâts dans un système distribué tels que la baisse de performance ou encore le dysfonctionnement du système.
Beaucoup de solutions [5, 7, 8, 9] ont été proposées dans le premier volet (prévention des fautes) pour tolérer les comportements rationnels et très peu dans le second volet (détection de fautes). Face à de tels comportements, comment détecter, identifier les nœuds responsables et par la suite convaincre les autres nœuds du système de leurs mauvais comportements ? Dans ce manuscrit nous allons apporter des éléments de réponses à cette question.

L'application des techniques de détection étant parfois très chère en terme d'échanges de messages, un autre problème auquel nous nous sommes intéressés est de savoir comment parvenir à une solution nécessitant un coût inférieur en terme d'échanges de messages.

1.3 Contributions

Les principales contributions de cette thèse sont les suivantes :

FullReview : Un nouveau protocole basé sur une solution logicielle générique sous-jacente renforçant la notion de responsabilité au niveau d'une application qui s'exécute sur un système distribué en présence de nœuds rationnels. FullReview utilise la théorie des jeux pour motiver et forcer les nœuds rationnels à suivre les différentes étapes du protocole. En plus nous prouvons théoriquement que notre protocole est un équilibre de Nash [10], c'est à dire que les nœuds rationnels n'ont aucun intérêt à dévier du protocole.

Gestion des moniteurs : Nous avons proposé une étude théorique sur les manières de déploiement des moniteurs d'un protocole assurant la notion de responsabilité au niveau d'une application s'exécutant sur un système distribué, afin d'aboutir à un coût raisonnable en terme d'échanges de messages. Pour cela nous nous sommes basés sur la manière d'assigner des surveillants à un nœud. En effet dans un système assurant la notion de responsabilité de façon générale, chaque nœud du système est associé à un groupe de nœuds appelés moniteurs ou surveillants qui le surveillent périodiquement. Nous verrons plus en détails ce mécanisme d'assignation dans la suite de ce manuscrit.

1.4 Organisation du document

Exceptés ce chapitre et celui de la conclusion, ce document est organisé en sept chapitres comme suit :

Le chapitre 2 - Tolérance aux fautes dans les systèmes distribués propose un état

de l'art restreint sur la notion de la tolérance aux fautes dans les systèmes distribués

Le chapitre 3 - Rationalité dans les systèmes distribués traite les comportements rationnels dans les systèmes distribués et présente le modèle BAR

Le chapitre 4 - Notion de responsabilité dans les systèmes distribués définit et décrit la notion de responsabilité dans les systèmes distribués et les solutions existantes dans la littérature, il propose également la technique de construction d'un système assurant cette notion

Le chapitre 5 - PeerReview décrit de façon détaillée le protocole *PeerReview*, la première solution logicielle dans la littérature garantissant la notion de responsabilité dans les systèmes distribués

Le chapitre 6 - FullReview propose *FullReview* notre solution à la notion de responsabilité en présence de comportements rationnels

Le chapitre 7 - Gestion des surveillants décrit la seconde contribution de cette thèse : la gestion des moniteurs ; l'objectif est d'avoir de bonnes performances en terme d'échanges de messages

Le chapitre 8 - Évaluation de performance porte sur l'évaluation de *FullReview* par rapport à *PeerReview*, tous les deux appliqués à deux applications largement utilisées

2

Tolérance aux fautes dans les systèmes distribués

Sommaire

2.1 Définitions 5
2.2 Types de fautes 6
2.3 Les différentes approches 7
2.3.1 Détection 7
2.3.2 Tolérance 8
2.4 Conclusion 8

La tolérance aux fautes est l'ensemble des techniques utilisées pour éviter les défaillances d'un système. Les systèmes informatiques répartis, en allant de l'Internet et en passant par les plateformes de calcul sont devenus indispensables et sont largement utilisés par beaucoup d'entreprises. Ces systèmes sont fréquemment exposés à des fautes (pannes, arrêts, mauvaise configuration etc) qui les empêchent de fonctionner correctement. Dans ce chapitre nous allons décrire les différents types de fautes et les approches utilisées pour les traiter en nous basant sur l'état de l'art sur la tolérance aux fautes, proposé dans [11].
Avant de passer à la description des types de fautes, nous allons d'abord définir certains termes relatifs aux systèmes distribués, afin de faciliter la compréhension du manuscrit.

2.1 Définitions

Un système distribué met en jeu plusieurs entités souvent dispersées géographiquement afin d'exécuter une tâche. Ainsi une entité est appelée `composant` ou `processus` ou `nœud`.
La tâche exécutée est appelée `application` ou `service`.

2.2 Types de fautes

Les différents types de fautes ont été largement étudiés dans la littérature [11, 12] et il existe entre autres :

Fautes par arrêt ou pannes franches Ces fautes conduisent à l'arrêt immédiat d'un composant du système qui fonctionnait correctement. Certains types de programmes erronés (une boucle à l'infini par exemple) sont souvent à l'origine de ces fautes.

Fautes d'omission ce sont des fautes qui surviennent à la suite de l'envoi ou de la réception d'un message par un composant du système mettant un temps assez long pour réagir. Par exemple le dysfonctionnement d'une carte réseau au niveau d'un composant peut aboutir à une faute d'omission.

Fautes dues au problème de réseau La congestion dans un réseau peut entrainer la perte de données et conduire à des faux résultats dans un calcul effectué par un système distribué.

Fautes malicieuses Ce sont des fautes commises intentionnellement par un ou plusieurs composants d'un système et qui forment parfois une coalition pour remonter un résultat erroné.

Fautes temporelles Ces fautes surviennent lorsque la réponse correcte à une requête émise par un composant du système n'est pas conforme aux spécifications temporelles du système. Par exemple une horloge trop rapide ou un délai de transmission trop long.

Fautes arbitraires ou Byzantines Ce sont des fautes dues à tout comportement qui s'écarte des spécifications d'un système de façon arbitraire. Par exemple une faute Byzantine peut être due à une erreur physique non détectée lors de la transmission d'un message. Ces types de fautes sont en général plus larges et englobent ceux cités ci-dessus.

Fautes rationnelles Ce sont des fautes dues à un comportement rationnel. Elles peuvent être classées dans la catégorie des fautes malicieuses. Par exemple une faute rationnelle survient lorsque dans une application de vidéo en direct, un nœud refuse de délivrer un morceau de vidéo. Ce refus se fait en se basant sur une fonction d'utilité connue. La fonction d'utilité s'obtient grâce à la consommation liée aux différentes ressources d'un système (bande passante, stockage, etc).
Dans ce manuscrit nous nous intéressons plus principalement à ces types de fautes. Comme annoncé dans l'introduction, notre objectif est de détecter ces comportements en utilisant la technique de responsabilité dans les systèmes distribués.

Après une vue d'ensemble sur les différents types de fautes, nous allons décrire les différentes approches utilisées pour les traiter.

2.3 Les différentes approches

Il est difficile voire impossible de concevoir des techniques de tolérance aux fautes dans les systèmes distribués sans faire des hypothèses car, dans le contexte de ces systèmes il y a deux familles : les systèmes synchrones [1], et les systèmes asynchrones. Dans la première famille, les hypothèses synchrones permettent la mise en œuvre facile des techniques de tolérance de fautes (par exemple la construction de détecteurs de pannes franches). Contrairement à la première famille, dans la deuxième famille il est très difficile voire impossible de faire des hypothèses sur le temps dans la mise en place des techniques de tolérance aux fautes. Il est donc nécessaire d'émettre d'autres hypothèses supplémentaires afin de rendre pratique la tolérance aux fautes. Ces hypothèses se font en général sur le réseau (fiabilité des canaux de communication à l'aide des moyens cryptographiques, délai de transmission des messages) ou sur des composants matériels.
Les techniques de tolérance aux fautes ne sont pas génériques, elles sont liées à des types de fautes. Autrement dit chaque type de fautes a en général ses techniques de tolérance aux fautes.

2.3.1 Détection

La détection a pour but de trouver la présence ou l'origine des fautes. Elle a l'avantage d'être plus efficace en terme de coût que la tolérance. Mais il n'est pas facile de la mettre en pratique pour certains types de fautes telles que les fautes Byzantines [13]. Les principales techniques utilisées dans le mécanisme de détection de fautes dans les systèmes distribués sont :

Vérification de résultats Cette technique est basée sur le vote majoritaire [14] ; l'idée est de comparer les différentes sorties des composants effectuant les mêmes traitements, afin de détecter les composants fautifs ;

Challenge-réponse Cette technique consiste à envoyer une requête à un nœud et attendre à une réponse pendant une durée déterminée ; elle permet surtout de détecter un nœud malicieux et distinguer un nœud lent (dû à un problème réseau) d'un nœud malicieux ; elle est utilisée dans [15].

Certification de résultats Cette technique probabiliste proposée par Varette et al. [16] permet de détecter des comportements malicieux en cas d'attaque massive d'un système distribué ; elle consiste à vérifier le résultat renvoyé par un système en se basant sur une post-condition et en introduisant une probabilité d'erreur quantifiée ;

Notion de responsabilité Cette technique permet de rendre chacun des nœuds d'un système responsable de ses actes et détecte ainsi les nœuds fautifs ; nous verrons plus en détails cette notion dans le chapitre 4.

1. Un système distribué synchrone est un système dans lequel on fait des hypothèses sur le temps.

2.3.2 Tolérance

Il est important de traiter ou de masquer ces fautes afin que le système puisse délivrer un service fiable malgré la présence des fautes d'où l'intérêt de la tolérance. Ici quand nous disons tolérance, nous faisons surtout allusion aux différentes approches de traitement de fautes dans les systèmes distribués. Ainsi nous avons :

Réplication de machines à états L'idée de cette approche est utiliser plusieurs copies du système sous la forme d'une machine à états [17] afin de masquer les fautes pour que le système puisse rester disponible ;

Reprise L'idée est d'enregistrer plusieurs états corrects du système afin de pouvoir revenir à un état correct en cas de problème ;

Expulsion de nœuds Elle est utilisée dans les environnements collaboratifs surtout dans le traitement des fautes rationnelles ; par exemple un nœud rationnel qui ne suit pas un protocole dans un tel environnement risque d'expulsion car, grâce aux techniques de la théorie des jeux un équilibre (équilibre de Nash [10]) est établit entre les nœuds rationnels du système ;

2.4 Conclusion

Pour conclure, dans ce chapitre nous avons abordé les principales techniques de tolérance de fautes dans les systèmes distribués. Ces techniques sont essentiellement constituées de deux grandes opérations : la détection et le traitement des fautes. Ces différentes approches sont indispensables pour la sûreté de fonctionnement d'un système distribué.
Dans le chapitre qui va suivre, nous allons nous intéresser aux fautes dues aux comportements dits rationnels. Ces comportements nécessitent une attention particulière car ils ont fait l'objet de plusieurs sujets de recherches dans la dernière décennie, ils ont également un lien avec cette thèse qui utilise la notion de responsabilité pour les traiter.

3

Rationalité dans les systèmes distribués

Sommaire

3.1 Qu'est ce qu'un comportement rationnel ? 9
3.2 Cas d'un système P2P 10
3.3 Modèle BAR 11
3.3.1 Définition 11
3.3.2 Description 11
3.3.3 Limitations 13
3.4 Conclusion 14

Nous avons vu dans les chapitres précédents qu'associer plusieurs milliers de nœuds dans le but de délivrer un service correct ne va pas sans conséquences surtout avec la présence de fautes. Nous avons aussi souligné que les comportements rationnels sont menaçants et causent d'énormes dégâts dans les systèmes distribués. Ainsi dans ce chapitre, nous allons abordé cette notion de comportement rationnel dans les systèmes distribués en y donnant une définition et en y présentant un état de l'art général. Ensuite nous ferons une brève description du modèle BAR qui nous servira pour la suite.

3.1 Qu'est ce qu'un comportement rationnel ?

Un nœud d'un système distribué se comporte de façon rationnelle lorsqu'il essaye de tirer profit du système sans y contribuer. Il maximise ainsi son bénéfice par rapport à une fonction d'utilité connue et qui est souvent fonction des ressources (réseau, cpu, stockage etc). Par exemple considérons une application de vidéo en direct qui s'exécute sur un système pair à pair (P2P) comme illustré sur la figure 3.1 à gauche. Périodiquement le nœud i sélectionne un ensemble de partenaires avec lesquels il

échange des morceaux de vidéos comme dans [18]. Intéressons nous aux échanges entre les deux nœuds i et j à droite de la figure 3.1. Le nœud i propose deux morceaux de vidéos u_1 et u_2 au nœud j qui à son tour demande seulement le morceau u_1. Ensuite il est servi par i avec u_1.
Lorsque le nœud i refuse de transmettre le morceau de vidéo u_1 à j afin de conserver sa bande passante, on dira que i s'est comporté de façon rationnelle. En outre si on fait l'hypothèse qu'il y a eu m messages échangés durant une période, la fonction d'utilité peut être définie comme $\alpha m + \beta$ où α et β sont des coefficients réels.

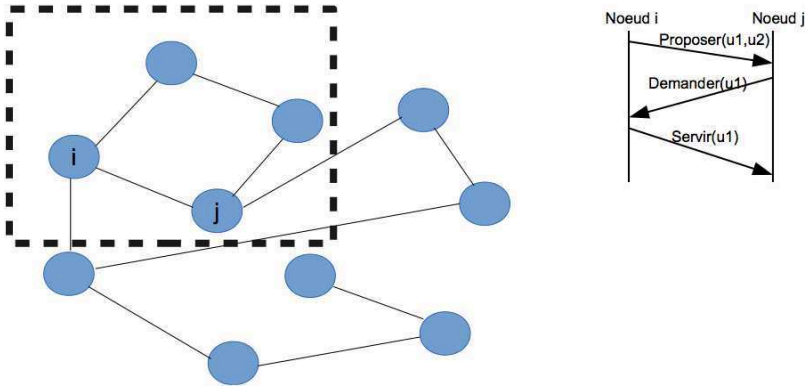


FIGURE 3.1 – Une application de vidéo en direct

3.2 Cas d'un système P2P

Avec l'essor des systèmes pair à pair, les comportements rationnels sont devenus de plus en plus fréquents et ont constitué une menace sérieuse dans les environnements collaboratifs comme signalé dans l'article [7]. C'est ainsi que des recherches dans le domaine des systèmes distribués se sont orientées vers la recherche de solutions à ces problèmes et ont abouti à de nouveaux protocoles [5, 8, 9, 18, 19, 20, 21, 22]. La plupart de ces solutions combinent des techniques de vérifications et la théorie des jeux pour inciter les nœuds dits rationnels à suivre un protocole. Ces solutions sont souvent spécifiques à un système donné. L'objectif de l'utilisation de la théorie des jeux est de parvenir à prouver que le protocole proposé est un équilibre de Nash c'est à dire qu'aucun nœud rationnel n'a d'intérêt à dévier du protocole.
Selon Shneidman et al [23], il existe différentes approches pour faire face aux comportements rationnels dans les environnements collaboratifs :

- ignorer les nœuds rationnels tout en espérant que le système va continuer à fonctionner car dans un environnement collaboratif comme un système P2P par exemple, il y a beaucoup de nœuds qui obéissent au protocole qui s'y exécute ;
- limiter l'effet qu'un nœud rationnel peut avoir sur l'exécution d'un système en utilisant un matériel spécifique [24] ou en faisant l'hypothèse qu'un nœud ne peut

pas modifier l'exécution d'un système (en général le système est construit dès le départ avec cette hypothèse) ;
- utiliser un algorithme distribué pour identifier et ignorer un nœud fautif [25] ; dans ce modèle un nœud rationnel qui dévie est considéré comme fautif ;
- construire un système de telle sorte qu'il puisse tolérer les comportements rationnels [26, 27] ;

Ces approches sont peu efficaces pour garantir des services fiables et stables dans les environnements collaboratifs. Elles ne proposent pas de mécanismes pour forcer ou motiver les nœuds rationnels à participer à un protocole. C'est ainsi que le modèle BAR que nous allons voir dans la section suivante a vu le jour.

3.3 Modèle BAR

La conception d'un système distribué robuste a été un grand défi à relever dans la dernière décennie. Dans ce contexte, un nouveau modèle appelé BAR a vu le jour.
Ainsi dans cette section, nous allons définir le modèle BAR, le décrire et ensuite présenter ses limitations.

3.3.1 Définition

BAR est l'acronyme de Byzantin, Altruiste, Rationnel. C'est un modèle de fautes qui a été introduit pour la première fois dans [5] pour construire un système robuste aux comportements rationnels.
Ainsi un protocole est dit BAR-résistant (c'est-à-dire qu'il résiste aux comportements rationnels) s'il tolère une quantité fixe de nœuds Byzantins et un nombre infini de nœuds rationnels. Les protocoles BAR-résistants combinent souvent la théorie des jeux afin de motiver les nœuds rationnels à suivre le protocole et la technique de responsabilisation afin de détecter les nœuds Byzantins en cas de déviation. Dans le passé, de multiples systèmes collaboratifs ont été conçus en considérant ce modèle, parmi lesquels nous avons les protocoles pour la dissémination résistante aux spams [8], des systèmes de fichiers distribués [5], des systèmes de diffusion de vidéo en direct [9, 18, 28], des systèmes de communication anonyme [20] et des systèmes de transfert de données entre entités [19].

3.3.2 Description

Comme indique son nom, le modèle BAR classe les nœuds d'un système collaboratif en trois catégories :

Byzantin : Les nœuds de cette catégorie dévient du système de façon arbitraire sans aucune raison connue ;

Altruiste : Les nœuds altruistes suivent correctement toutes les étapes d'un protocole auquel ils participent ;

Rationnel : Les nœuds rationnels ont pour objectif de maximiser leur bénéfice selon une fonction d'utilité connue(cf section 3.1) ; ils peuvent dévier d'un protocole auquel ils participent si seulement si, dévier augmente leur utilité ;

L'objectif du modèle BAR est de garantir certaines propriétés pour les nœuds altruistes et rationnels dans les protocoles de tolérance aux fautes. Deux classes de protocoles rentrent dans cet objectif :

Les protocoles qui incitent les rationnels à suivre les différentes étapes : Un protocole est de cette classe s'il garantit les propriétés de sûreté [1] et de vivacité [2] tout en incitant les nœuds rationnels à le suivre exactement [9, 20] ; il est nécessaire de définir une stratégie optimale pour les rationnels dans cette catégorie ;

Les protocoles qui tolèrent les comportements rationnels : Un protocole est de cette catégorie s'il garantit les propriétés de sûreté et de vivacité en présence de comportements rationnels ; dans cette classe un nœud rationnel exploite les optimisations locales non spécifiées dans le protocole sans mettre en cause les garanties globales [5] ;

La première classe est un sous ensemble de la deuxième classe comme illustré sur la figure 3.2. Le modèle BAR nécessite certaines hypothèses [5, 9] qui sont faites sur les

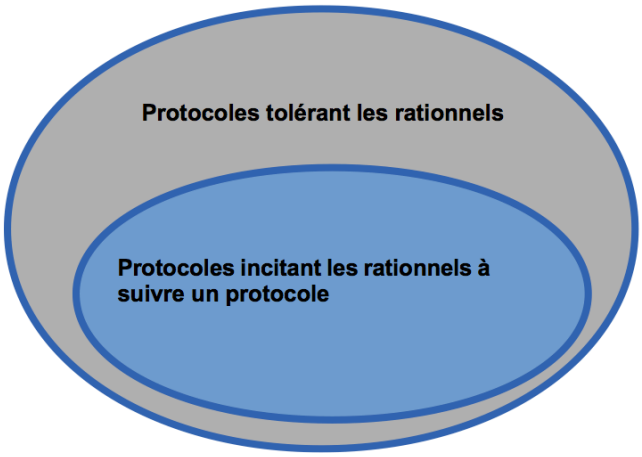


FIGURE 3.2 – Classes de protocoles dans le modèle BAR

nœuds rationnels. Parmi ces hypothèses nous avons :

- un nœud rationnel suit toujours la stratégie qui maximise son utilité ;
- un nœud rationnel cherche toujours à maximiser son bénéfice à long terme ;
- un nœud rationnel n'a aucun intérêt à recevoir un message qui n'est pas du protocole ;

1. la propriété de sûreté assure qu'un nœud commence et termine son exécution dans un état correct.
2. la propriété de vivacité assure qu'à terme un nœud termine son exécution, il ne rentrera pas dans une boucle infinie par exemple.

– un nœud rationnel est prudent quand il évalue l'impact d'un nœud Byzantin sur son bénéfice ;
– un nœud rationnel participe toujours à un protocole si celui ci est un équilibre de Nash ;
– un nœud rationnel ne peut pas former une coalition avec un autre ;

Le processus de construction d'un protocole utilisant le modèle BAR est composé des étapes suivantes :

1. définir la fonction d'utilité pour des nœuds rationnels dans le protocole en question ;
2. lister toutes les déviations des nœuds rationnels selon la fonction d'utilité définie ;
3. proposer des techniques qui vont motiver les nœuds rationnels à suivre le protocole pour chaque déviation identifiée ;
4. prouver que le protocole en question est un équilibre de Nash c'est à dire aucun nœud rationnel n'a intérêt à dévier du système ;

Le rêve de tout responsable de sécurité d'un système, est d'avoir un moyen automatique permettant de transformer un protocole donné en un protocole BAR-résistant. Deux solutions qui vont vers cette direction dans la littérature ont été proposées. La première solution est le protocole Nysiad [29] qui permet la transformation automatique d'un protocole donné en un protocole résistant aux fautes Byzantines. Nysiad atteint cet objectif en répliquant chaque nœud du système grâce à une variante de la réplication des machines à états (RSMs). Toutefois, le système résultant est vulnérable aux comportements rationnels. Contrairement à la première solution, la seconde est le protocole PeerReview [15] permettant de détecter automatiquement toute sorte de déviations observables aussi bien rationnelles que Byzantines qu'un nœud peut faire dans un protocole surveillé, nous verrons ce protocole plus en détails dans le chapitre 5.

Les deux premiers protocoles qui ont intégré le modèle BAR sont le protocole BAR-B [5] et le protocole BAR gossip [9].
BAR-B est un protocole collaboratif destiné à un service de sauvegarde. Ce protocole propose une architecture générique qui utilise le modèle BAR et qui a pour objectif de fournir un framework pour la réalisation des services collaboratifs.
BAR gossip est un protocole collaboratif destiné à la diffusion de vidéo en direct. Il traite les comportements rationnels dans les environnements dont la dissémination est épidémique [3].

3.3.3 Limitations

Malgré le fait que le modèle BAR rend possible la construction de systèmes tolérants les comportements rationnels, il présente une limitation majeure. En effet l'approche

3. Une dissémination est épidémique lorsque dans un système un nœud choisit périodiquement et de façon aléatoire un ensemble de nœuds à qui, il envoie l'information qu'il a reçue.

utilisée dans ce modèle est effectuée de façon manuelle par un expert système. Très complexe, cette approche est souvent source d'introduction de fautes dans les systèmes distribués. En outre, toute modification du système original nécessite la reconsidération du système tout en entier. Cela peut causer la création de nouvelles déviations rationnelles et Byzantines.

3.4 Conclusion

Dans ce chapitre nous avons abordé les comportements rationnels qui constituent une menace sérieuse quand à la sûreté de fonctionnement d'un système distribué. Plusieurs solutions allant de solutions matérielles à des solutions logicielles ont été proposées pour faire face à ces comportements. Parmi toutes ces solutions, celles considérant le modèle BAR se sont révélées efficaces, toujours d'actualité et présentent certaines limitations. Partant de ces constats, nous avons combiné ce modèle avec la technique de responsabilité pour concevoir un nouveau protocole de détection de comportements rationnels. Ce protocole constitue la contribution principale de cette thèse, et est décrit dans le chapitre 6.

Dans le chapitre qui va suivre nous allons introduire cette notion de responsabilité qui est une technique puissante pour faire face aux fautes dans les systèmes distribués et en particulier dans les environnements collaboratifs, et a été utilisée dans la conception de notre protocole.

4

Notion de responsabilité dans les systèmes distribués

Sommaire

4.1 Définition 16
4.2 Construction d'un système responsable 16
4.2.1 Architecture d'un système responsable 16
4.2.2 Historique sécurisé 17
4.3 État de l'art 18
4.3.1 Solutions spécifiques 18
4.3.2 Solutions génériques 22
4.4 Conclusion 26

Nous avons vu précédemment que les systèmes distribués peuvent être sujets à des fautes de types variés. Cependant, il est très difficile de connaitre dans un tel système les nœuds responsables de ces fautes. C'est ainsi que la notion de responsabilité a été introduite dans le domaine des systèmes informatiques répartis pour pallier à ce problème.

Bien avant son utilisation dans les systèmes distribués, la notion de responsabilité s'utilisait dans les systèmes bancaires (par exemple lorsqu'un client procède à une opération bancaire, il ne peut pas nier sa part de responsabilité en cas de problème).La première définition de cette notion est relative à l'éthique, en effet elle se réfère à la capacité de rendre une personne responsable de ses actes [30, 31, 32]. Dans les systèmes distribués, elle fait plutôt allusion à la technique consistant à tenir les nœuds responsables de leurs actions. Cette technique utilise les opérations effectuées par les nœuds d'un système pour atteindre son objectif.

Ce chapitre porte sur la notion de responsabilité dans les systèmes distribués, une technique efficace pour détecter les fautes et identifier les nœuds fautifs. La notion de

responsabilité se base sur les échanges entre les différents nœuds d'un système pour détecter les fautes et identifier les nœuds fautifs. Ces échanges sont enregistrés dans un historique sécurisé à l'aide de moyens cryptographiques.
Durant tout le long de ce chapitre, nous allons définir cette notion, ensuite nous décrirons les éléments nécessaires pour l'application de cette technique. Cette description sera suivie de la présentation des travaux qui ont été effectués dans ce domaine et d'une conclusion.

4.1 Définition

La notion de responsabilité dans les systèmes distribués est une technique permettant de détecter les fautes, identifier les nœuds responsables et convaincre les nœuds corrects à propos des nœuds fautifs.
Ainsi pour atteindre ces trois objectifs à savoir : détecter les fautes, identifier les nœuds responsables et convaincre les autres nœuds du système, cette technique nécessite l'utilisation d'un historique sécurisé dans lequel on enregistre toutes les actions relatives à un nœud. Nous verrons cela plus en détails dans la section 4.2.2.
Tout protocole utilisant cette notion garantit deux propriétés principales : la complétude et la précision. La complétude est la propriété qui assure que tout nœud fautif sera détecté ultérieurement tandis que la précision assure qu'aucun nœud correct ne sera jamais accusé de mauvais comportement par un autre nœud correct.

4.2 Construction d'un système responsable

La plupart des protocoles assurant la notion de responsabilité nécessitent l'utilisation d'un historique sécurisé et adoptent une certaine architecture. Dans cette section nous décrivons l'architecture classique d'un système responsable (un système assurant la notion de responsabilité) et l'historique sécurisé, deux éléments parfois indispensables dans la mise en place de la notion de responsabilité.

4.2.1 Architecture d'un système responsable

Selon l'architecture classique de la notion de responsabilité indiquée sur la figure 4.1, chaque nœud i d'un système distribué interagit avec un ensemble de nœuds appelés ses partenaires et qui apparaissent à droite de la figure. En plus de l'ensemble des partenaires, le nœud i a un ensemble de moniteurs ou surveillants qui vérifient périodiquement si i suit ou pas le protocole qui s'exécute sur le système. Cet ensemble est désigné par $m(i)$ et apparait en dessus de i sur la figure. De façon symétrique i surveille un ensemble de nœuds noté $m^{-1}(i)$ et qui apparait en dessous de i sur la figure. Pour effectuer toutes les actions de surveillance, chaque nœud maintient un historique sécurisé qui est non altérable et dans lequel on ne peut qu'ajouter des informations et écrire toutes les

interactions avec ses partenaires (les détails sur les historiques sécurisés sont donnés dans la section 4.2.2). Cet historique est périodiquement audité par les moniteurs de i ($m(i)$) afin de détecter les nœuds fautifs.

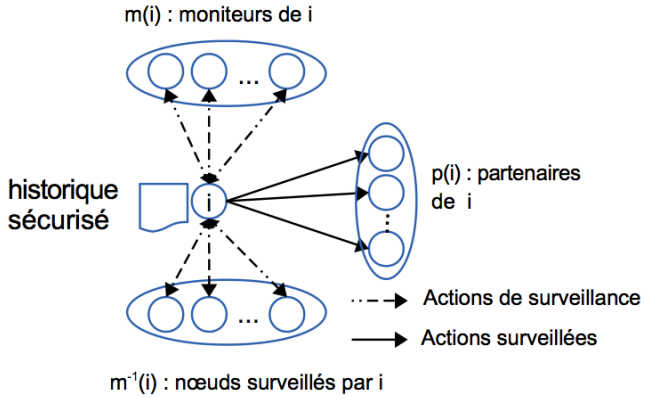


FIGURE 4.1 – Architecture simple d'un système responsable de ses actes.

4.2.2 Historique sécurisé

L'un des éléments essentiels dans la mise en place de la notion de responsabilité dans un système distribué est l'historique sécurisé. Son rôle principal est de stocker les traces des messages échangés entre un nœud et ses partenaires. Selon les exigences du système dans lequel on veut assurer la notion de responsabilité, les entrées d'un historique labellisées $e_0, ..., e_k$ peuvent contenir différentes informations parmi lesquelles l'identifiant du message envoyé par un nœud ou reçu de ses partenaires. Chaque entrée

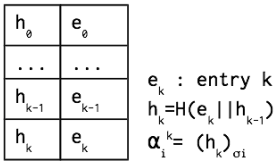


FIGURE 4.2 – Exemple d'un historique sécurisé.

e_k est associée à une valeur récursive h_k qui est définie comme étant le *hash* de e_k concaténé avec la valeur de h_{k-1} (où h_{-1} est une valeur fixée), et un authentificateur α_i^k, qui est un message contenant la valeur de h_k signé avec la clé privée [1] de i, c'est à dire $\alpha_i^k = (h_k)_{\sigma_i}$. Les authentificateurs permettent de vérifier qu'un nœud n'a pas modifié son historique. A titre d'exemple, considérons un nœud j parmi les surveillants

1. En général dans un système distribué chaque nœud possède une paire de clés : une clé publique et une clé privée

du nœud i. Si j obtient une paire d'authentificateurs α_i^0 et α_i^k correspondant aux entrées e_0 et e_k dans l'historique de i respectivement, il peut demander à i ses entrées $e_0, ..., e_k$ et recalculer $h_0, ..., h_k$. Si la valeur de h_k calculée diffère de celle que j a obtenu, alors le nœud j peut générer une preuve contre i pour modification de son historique. De plus, j peut convaincre tout autre nœud correct que l'historique de i a été altéré, en leur envoyant les authentificateurs signés α_i^0 et α_i^k avec les entrées envoyées par i.

Toutefois il est à noter que tous les systèmes distribués n'utilisent pas forcement l'architecture classique, certains ont des architectures propres à eux, nous verrons cela dans la section 4.3. Par contre ils utilisent tous l'historique sécurisé représenté sous une forme ou une autre.

4.3 État de l'art

Dans le contexte de la notion de responsabilité, deux sortes de solutions ont été proposées : les solutions spécifiques et les solutions génériques. Les solutions de la première catégorie sont relatives à un type de système distribué donné (elles se construisent en tenant compte de la structure et le type de service du système en question). Ces solutions sont limitées car elles ne sont pas applicables au delà des systèmes sur lesquels elles se basent. Ainsi celles de la seconde catégorie pallient à ce problème et sont indépendantes de la structure d'un système distribué donné.
Nous décrivons ces solutions dans la suite de ce chapitre.

4.3.1 Solutions spécifiques

Dans la littérature il existe des solutions spécifiques garantissant la notion de responsabilité dans les systèmes distribués. Parmi ces systèmes :

4.3.1.1 Dissent

Dissent est un protocole qui a été conçu par Corrigan-Gibbs et al [33] pour renforcer la notion de responsabilité dans un système de communication anonyme. Ce protocole oblige les nœuds à rendre compte de leurs actes. Il est composé de deux sous-protocoles : le premier sous-protocole repose sur un mécanisme de mélange et assure l'anonymat ; quant au second il permet l'échange des messages de tailles variables.
Ces deux sous-protocoles combinés rendent ainsi les nœuds responsables de leurs actes. En effet, les nœuds participants au protocole Dissent sont ordonnés et possèdent chacun deux paires de clés appelées respectivement paire de clés primaires et paire de clés secondaires. En plus chaque nœud du système maintient un historique sécurisé dans lequel sont enregistrés les entrées et les sorties. Le protocole fonctionne par ronde durant laquelle, chaque nœud chiffre le message qu'il veut envoyer. Ce chiffrement s'effectue en appliquant d'abord les clés secondaires publiques (dans l'ordre inverse des nœuds), ensuite les clés primaires publiques (dans l'ordre inverse des nœuds également).

Chaque nœud envoie ensuite son message chiffré au premier nœud du système.
Une fois tous les messages chiffrés reçus, le premier nœud enlève une première couche de chiffrement sur chaque message grâce à sa clé primaire privée, il mélange ensuite les messages obtenus et les envoie au second nœud. Le second nœud fait les mêmes opérations (déchiffrement et mélange) que le premier et envoie les résultats au troisième. Ce processus continue jusqu'à ce que les messages atteignent le dernier nœud et que toutes les couches de chiffrement primaire aient été enlevées.
Le dernier à son tour, diffuse l'ensemble des messages à tous les autres nœuds du système. A la réception des messages chaque nœud vérifie que son message est bien présent et n'a pas été altéré, en utilisant son historique sécurisé. A l'issue de cette vérification, il informe les autres nœuds s'il continue le protocole ou pas. Au cas où tous les nœuds sont d'accord pour continuer le protocole, alors chacun révèle aux autres sa clé privée secondaire afin de faciliter le déchiffrement des autres couches et obtenir par la suite les messages en clair. Dans le cas échéant c'est-à-dire un nœud A ne veut pas continuer le protocole suite à l'altération de son message, alors chaque nœud détruit la paire de clés secondaires de A et révèle sa paire de clés primaires. Ainsi chaque nœud i peut utiliser ses informations pour connaître le comportement de chaque autre nœud j en rejouant les différentes phases du protocole.
Et pour maintenir la notion de responsabilité dans le système durant tout le long du protocole, Dissent s'assure qu'aucun autre nœud n'expose un nœud correct, et qu'après une exécution, soit chaque nœud correct a reçu correctement les messages des autres nœuds corrects ou tous les nœuds corrects ont au moins exposé un nœud fautif.

4.3.1.2 CATS

Le protocole CATS(Certified Accountable Tamper-evident Storage service) [34] est un protocole basé sur une architecture client/serveur et destiné à garantir la notion de responsabilité dans un service de stockage distribué. CATS utilise les opérations rudimentaires d'un service de stockage classique (lecture et écriture). Les clients accèdent par lecture ou écriture aux objets stockés dans des répertoires partagés sur le serveur CATS. Pour tenir chacun des participants du protocole responsable de ses actes, CATS fournit des interfaces pour le challenge et l'audit. L'interface de challenge permet aux serveurs de vérifier que les écritures ont été bien faites et sont visibles par tous les autres clients, ce qui engage la responsabilité du client qui a effectué ces opérations. Les challenges obligent aussi le serveur à fournir une preuve cryptographique certifiant que ses actions concernant les résumés de l'état du système sont correctes et consistantes. En effet, périodiquement le serveur enregistre l'état du système dans une sorte d'historique qui est visible par tous les participants. L'interface d'audit permet à un auditeur [2] (un nœud qui fait l'audit) de vérifier l'intégrité des écritures en se basant sur un historique récent de l'état du système. Ainsi le serveur ne peut pas revenir sur ou modifier une opération d'écriture sans être détecté. Chaque participant a la possibilité de vérifier

2. Tout nœud peut être auditeur.

les résultats d'un audit sans un quelconque avis d'un auditeur. La figure 4.3 présente une vue d'ensemble du protocole CATS. Sur cette figure un client effectue différentes opérations (lecture, écriture etc) sur le serveur qui lui envoie des preuves d'exécution correctes des requêtes, des résultats et périodiquement un résumé de l'état du système sous la forme d'une signature. Ce résumé est aussi publié dans un historique. Le client peut ainsi comparer les différents états du système et vérifier les preuves.

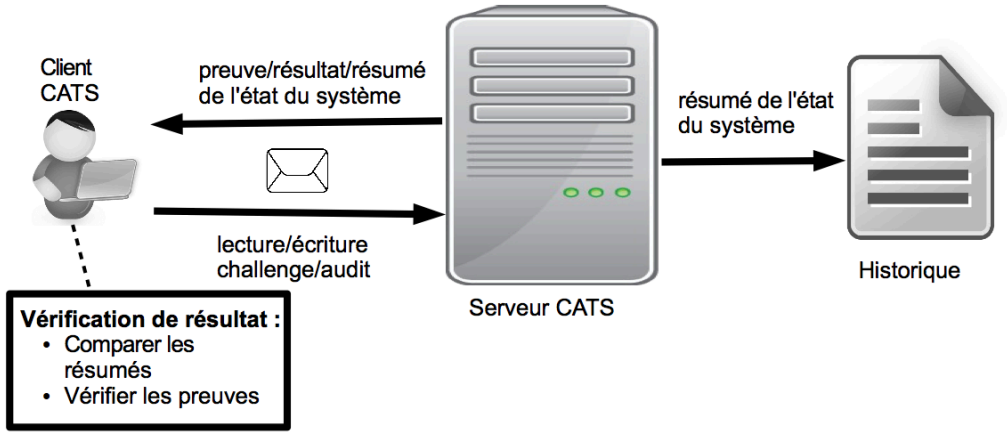


FIGURE 4.3 – Vue globale du protocole CATS

4.3.1.3 NetReview

NetReview est un protocole qui a été conçu par Haeberlen et al [35] pour renforcer la notion de responsabilité dans BGP (*Border Gateway Protocol*), un système de routage inter-domaine. BGP assure sa fonction de routage en faisant la coordination entre différentes entités appelées systèmes autonomes (en anglais autonomous systems) et distribuées. Chaque système autonome est constitué de plusieurs routeurs dont des routeurs *speaker*, qui servent à communiquer les différents systèmes autonomes (SA). Un système autonome est administré par un fournisseur d'accès à internet (FAI).
Chaque routeur *speaker* a un historique sécurisé à l'aide de moyens cryptographiques et dans lequel sont enregistrés tous les messages qui transitent entre les différents SA, comme illustré sur la figure 4.4 qui représente une vue d'ensemble du protocole NetReview. Ainsi pour rendre chacun des participants responsables de leurs actes, NetReview permet aux FAI de vérifier ces historiques selon un ensemble de règles, afin de détecter les nœuds fautifs. Si une faute est commise, NetReview garantit qu'elle sera au moins détectée par un auditeur avec une preuve vérifiable à l'appui, qui pourra être utilisée par la suite pour convaincre les autres participants.
L'enregistrement des messages se fait à l'aide des authentificateurs cryptographiques. Les authentificateurs sont des preuves qui prouvent qu'il y a eu des échanges entre les nœuds, et permettent également la détection d'un historique altéré.

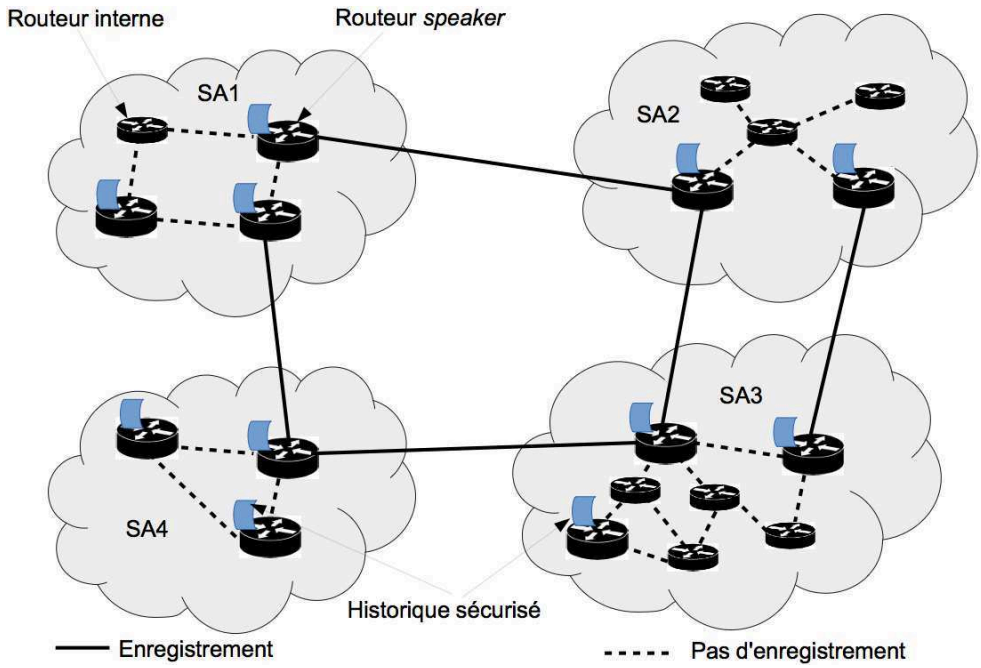


FIGURE 4.4 – Vue globale du protocole NetReview

4.3.1.4 *Repeat and Compare*

Repeat and Compare(RC) [36] est un protocole qui assure l'intégrité des données dans un système pair-à-pair de distribution de contenu (*Content Distribution Network*) en utilisant des techniques de la notion de responsabilité. Un CDN est constitué de plusieurs nœuds en réseau via l'Internet et qui coopèrent pour mettre à la disposition des utilisateurs du contenu multimédia volumineux. Les nœuds d'un CDN ont des rôles différents. Ainsi il y a des nœuds qui contiennent la source des données (serveurs d'origine), des nœuds sur lesquels sont stockées les données répliquées (les réplicas). RC atteint son objectif grâce à une attestation d'enregistrement et une ré-exécution partielle de façon répétée. En effet, comme illustre la figure 4.5, chaque réplica inclut une attestation d'enregistrement sécurisé à l'aide de moyens cryptographiques, dans sa réponse (étape 1 sur la figure), ensuite le client transfère l'attestation à un réplica choisi aléatoirement et qui joue le rôle de vérificateur (étape 2 sur la figure). Enfin, pour détecter les réplicas fautifs, le vérificateur répète l'exécution et vérifie si le résultat correspond à celui contenant dans l'enregistrement envoyé par le client (étape 3).

4.3.1.5 *CSAR*

CSAR(Cryptographically Strong Accountable Randomness) est une technique conçue par Backes et al [37] et qui permet d'assurer la notion de responsabilité dans les systèmes distribués utilisant des protocoles randomisés. Pour arriver à cette fin,

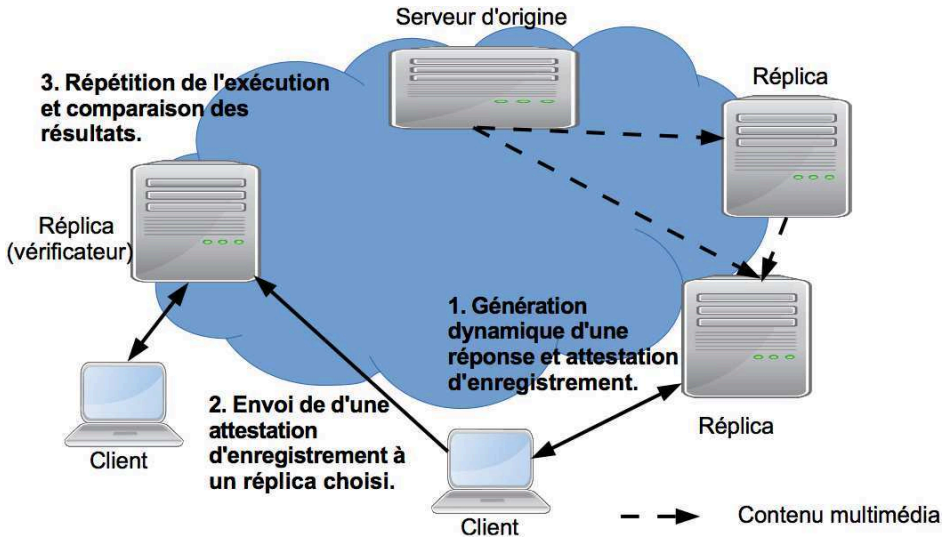


FIGURE 4.5 – Vue globale du protocole *Repeat and Compare*

CSAR génère une séquence pseudo-aléatoire et une preuve certifiant que les éléments de cette séquence jusqu'à un point donné sont correctement générés. Les futures valeurs générées dans la séquence restent imprévisibles. En plus CSAR permet à un auditeur externe au système de vérifier si un nœud est correct ou pas. Ce qui garantit sa propriété de forte "notion de responsabilité".

4.3.1.6 *AIP*

AIP(Accountable Internet Protocol) [38] utilise une hiérarchie d'auto-certification (*self-certifying hierarchy*) d'adresses afin d'assurer la notion de responsabilité au niveau de la couche réseau. Avec cette structure AIP peut détecter et prévenir des attaques dont certains systèmes distribués font l'objet. Ces attaques sont entre autres : l'usurpation d'adresse IP, le déni de service, le détournement d'adresse IP et le faux routage de paquets.

4.3.2 Solutions génériques

Les solutions génériques sont celles qui sont indépendantes de la structure des systèmes distribués auxquelles elles sont appliquées. Ainsi elles renforcent la notion de responsabilité au sein de beaucoup de systèmes. Elles comprennent deux familles de solutions : les solutions matérielles et les solutions logicielles.

4.3.2.1 Solutions matérielles

Pendant longtemps, les solutions connues pour mettre en place la notion de responsabilité dans les systèmes distribués étaient matérielles. Dans cette famille les protocoles existants dans la littérature sont :

TrInc [39] est un composant matériel qui permet de combattre les comportements malicieux dans un système distribué en rendant chaque participant responsable de ses actes. L'un des principaux objectifs de TrInc est d'éliminer la capacité des participants à équivoquer (interdire aux participants toute possibilité de fournir plusieurs résultats pour une même opération). Pour tirer parti de TrInc, chaque utilisateur doit intégrer une pièce matérielle appelée *trinket* à son ordinateur. Les messages envoyés et reçus passent par cette pièce. Prenons l'exemple d'un participant A qui veut envoyer un message m à un participant B. A doit inclure une attestation fournie par sa *trinket*. Cette attestation a deux rôles : (1) elle lie le message m à un compteur et (2) elle s'assure qu'aucun autre message même venant d'autres utilisateurs n'est lié à ce compteur. La *trinket* produit ces attestations en utilisant un compteur qui augmente de façon monotone avec une nouvelle attestation. De cette manière le participant A ne sera pas capable de lier un autre message m' au compteur utilisé par m sans être détecté. Cela rend A responsable de ses actions.

A2M mis en place par Chun et al [40], A2M(Attested Append-Only Memory) est un composant manipulant des historiques sécurisés et permettant ainsi à un protocole s'exécutant sur un système distribué de détecter les participants qui mentiraient, et cela pour assurer la notion de responsabilité. A2M contient un ensemble d'historiques ordonnés et ayant chacun un identifiant unique comme illustré sur la figure 4.6. L'entrée d'un historique est composée d'un identifiant (l'identifiant de l'historique), d'une valeur, d'un résumé récursif (contenant les résumés précédents) et est associée à un numéro de séquence variant entre deux bornes (une borne inférieure I et une borne supérieure S). Ainsi avec cette structure par exemple, un serveur ne peut pas répondre différemment aux clients qui envoient les mêmes requêtes sans être détecté.
En outre A2M fournit une interface constituée d'un ensemble de fonctions permettant de manipuler facilement les historiques (ajout, recherche, suppression etc).

Pasture est une librairie utilisant un composant matériel pour permettre un accès sécurisé à des données en mode hors ligne et assurer ainsi la notion de responsabilité au sein des utilisateurs. Conçue dans [41], Pasture garantit deux propriétés de sécurité : (1) un utilisateur ne peut pas nier le fait d'accéder à des données sans faire l'objet d'un audit, (2) un utilisateur ne peut jamais accéder à une donnée après avoir généré une preuve vérifiable de révocation concernant l'accès à cette donnée. Pasture a deux objectifs qui sont : permettre aux utilisateurs non fiables (*untrusted users*) de télécharger les données en ligne et sécuriser l'accès aux

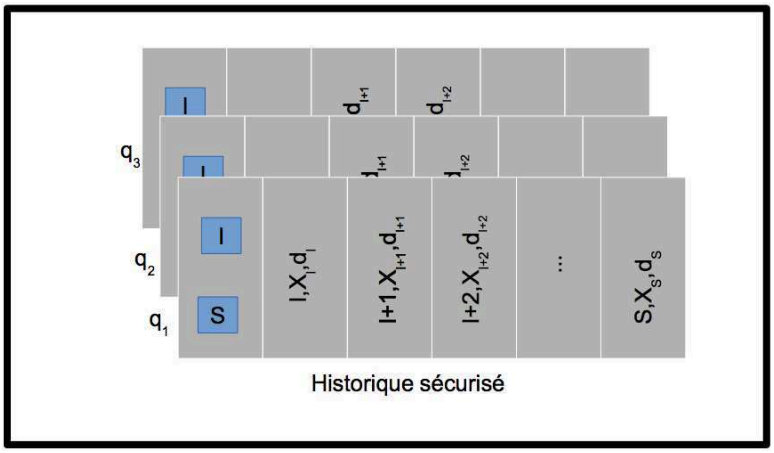


FIGURE 4.6 – Structure de A2M

données en mode hors ligne. Pour atteindre ces objectifs, Pasture utilise une puce TPM(Trusted Platform Module), un composant cryptographique matériel et le mode d'exécution sécurisée SEM (Secure Execution Mode). La puce TPM chiffre et déchiffre les messages, contient un registre de configuration de la plateforme PCR (Platform Control Register) qui permet de stocker les résumés des historiques des évènements. La figure 4.7 montre l'architecture de Pasture. Chaque nœud exécute une instance de Pasture identifiée par une paire unique de clés publique/privée qui est générée par la puce TPM correspondante. Tous les messages et les preuves générés par une instance de Pasture sont signés grâce à la clef privée. Les signatures sont vérifiées à l'aide de la clef publique.
Chaque instance de Pasture a un historique sécurisé contenant des décisions $D_1, D_2, ...$ concernant l'accès ou la révocation d'une clef. L'historique en entier est stocké sur un nœud non fiable et son résumé est stocké sur PCR au niveau de la puce TPM. L'application exécutant l'instance de Pasture utilise une interface pour créer et vérifier les clés de chiffrement durant le transfert des données, obtenir et révoquer l'accès aux clés de déchiffrement correspondant selon les décisions en mode hors ligne, et répondre à un audit. Ainsi avec cette architecture, Pasture rend un nœud responsable de ses actes.

4.3.2.2 Solutions logicielles

Les solutions logicielles ont été peu étudiées dans la littérature, et à notre connaissance PeerReview [15], AVM(*Accountable Virtual Machine*) [42] et le protocole proposé dans cette thèse sont les seuls assurant la notion de responsabilité dans les systèmes distribués.

PeeReview est le premier protocole logiciel apportant une réponse à la notion de responsabilité dans les systèmes. PeerReview est basé sur l'architecture classique

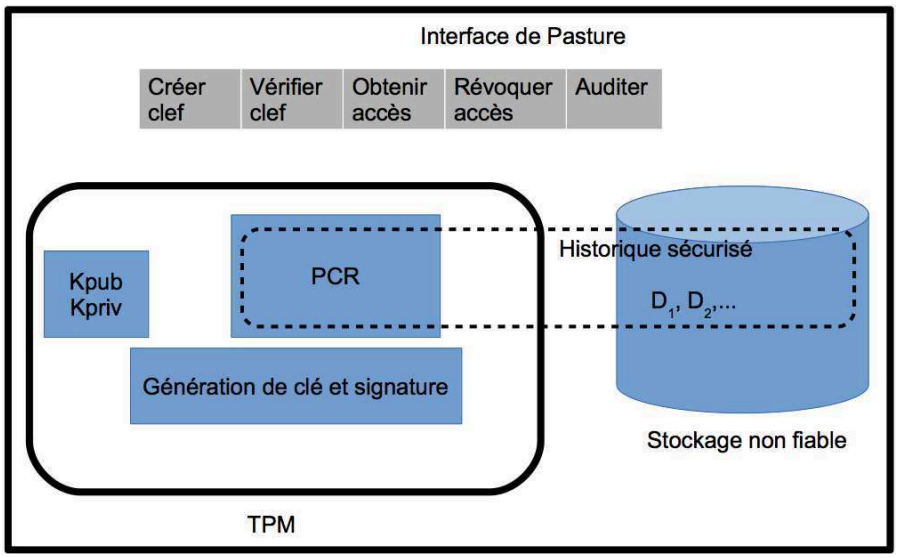


FIGURE 4.7 – Architecture de Pasture

(voir section 4.2.1) d'un système assurant la notion de responsabilité. En effet dans ce protocole chaque nœud a un historique sécurisé dans lequel il enregistre tous les échanges effectués avec les autres nœuds du système. En outre chaque nœud A du système est associé à un ensemble m de nœuds, qui auditent périodiquement l'historique de A afin de détecter les éventuelles déviations. Et si a un nœud a une preuve de dysfonctionnement contre un autre nœud, alors il peut convaincre les autres participants en rendant disponible cette preuve. Ces différents processus permettent ainsi de tenir chaque nœud responsable de ses actions. Nous verrons plus en détails ce protocole dans le chapitre 5.

AVM est une variante de *PeerReview* et a été conçu par les mêmes auteurs que ce dernier. Contrairement à ce dernier, il assure la notion de responsabilité dans les systèmes distribués en se basant sur des machines virtuelles. En effet il permet l'exécution d'une image binaire d'un logiciel sur des machines virtuelles, tout en permettant aux utilisateurs de surveiller le processus d'exécution, grâce à un historique sécurisé.
Pour mieux illustrer l'idée de ce protocole, considérons le scénario de la figure 4.8. Dans ce scénario, Alice utilise un logiciel L qui s'exécute sur une machine M contrôlée par Bob. Cependant, Alice ne connait pas la machine M, elle ne peut que s'y connecter via un réseau. L'objectif de AVM est de permettre à Alice de vérifier le comportement de M sans avoir confiance en Bob ou à un autre logiciel s'exécutant sur M. AVM atteint ainsi cet objectif, en permettant à Alice de posséder une implémentation de référence de M appelée M_R, qui exécute L. Ensuite, elle compare les sorties de M et de M_R pour savoir si M est correct ou pas.

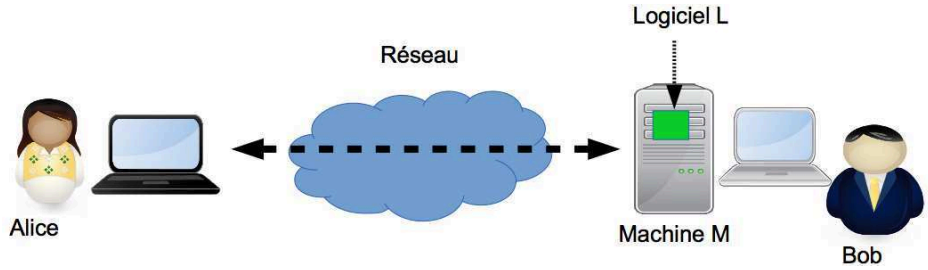


FIGURE 4.8 – Scénario d'illustration du protocole AVM

4.4 Conclusion

Pour résumer, dans ce chapitre nous avons abordé la notion de responsabilité dans un système distribué, en mettant l'accent sur le composant indispensable dans cette technique (l'historique sécurisé) et l'architecture classique d'un système responsable [3]. L'état de l'art fait dans ce chapitre part des solutions spécifiques aux solutions génériques. Les solutions génériques sont composées de deux familles : solutions matérielles et solutions logicielles. Dans la deuxième famille, il n'existe dans la littérature que deux protocoles à savoir PeerReview [15] et AVM [42]. Le protocole proposé dans ce manuscrit apportera une contribution à cette famille.
Le tableau 4.1 donne un aperçu des protocoles étudiés dans la littérature.

Protocole	SS	SG	Logiciel	Matériel	TSD
Dissent	•		•		communication anonyme
CATS	•		•		stockage
NetReview	•		•		routage
Repeat and Compare	•		•		CDN (distribution de contenu)
CSAR	•		•		système randomisé
AIP	•		•		protocole d'Internet
TrInc		•		•	
A2M		•		•	
Pasture		•		•	
PeerReview		•	•		
AVM		•	•		

TABLE 4.1 – Résumé des protocoles étudiés dans l'état de l'art. SS = Solution Spécifique, SG = Solution Générique, TSD = Type de Système Distribué auquel le protocole est appliqué.

Tous ces protocoles existants dans la littérature ont leur modèle de fautes c'est à dire les types et le nombre de fautes qu'ils sont capables de détecter ou de tolérer.

3. Système assurant la notion de responsabilité

Aucun d'entre eux ne traitent réellement les comportements dits rationnels ou s'ils les traitent c'est seulement au niveau de l'application qu'ils surveillent. Pour mettre cela en évidence, nous nous sommes intéressés au cas du protocole PeerReview qui fera l'objet du chapitre suivant. L'intérêt d'étudier en détails PeerReview n'est pas seulement pour la mise en évidence des comportements rationnels mais aussi parce que notre protocole y est basé.

5

PeerReview : Une solution logicielle générique assurant la notion de responsabilité dans un système distribué

Sommaire

5.1 Description 30
5.1.1 Modèle de fautes 30
5.1.2 Modèle de système 30
5.1.3 Hypothèses 30
5.1.4 Propriétés 31
5.1.5 Sous-protocoles 31
5.2 Déviations rationnelles dans *PeerReview* 34
5.3 Impact des nœuds rationnels 35
5.4 Conclusion 37

Comme annoncé dans le chapitre précédent, *PeerReview* [15] est le premier protocole basé sur une solution logicielle générique et qui tient chacun des nœuds d'un système distribué responsable de ses actes. Pour atteindre cet objectif, *PeerReview* utilise un historique sécurisé pour enregistrer les échanges entre les nœuds et une implantation de référence du protocole sous forme d'une machine à états déterministe. L'intérêt d'utiliser une implantation du protocole sous forme d' une machine à états est de pouvoir détecter les fautes durant la phase d'audit. En effet la machine à états aura comme entrée l'historique et sa sortie est "oui" s'il est correct et "non" dans le cas échéant.

Dans ce chapitre nous allons décrire en détails le protocole *PeerReview*. Nous nous intéresserons ensuite au cas des comportements rationnels et leur impact sur ce protocole.

5.1 Description

Cette section décrit en entier le protocole *PeerReview* en commençant par le modèle de fautes utilisé, ensuite le modèle de système supposé et les hypothèses faites par les auteurs, et en terminant par l'étude des sous protocoles constituant *PeerReview*. Ses propriétés y seront également abordées.

5.1.1 Modèle de fautes

Le protocole *PeerReview* étend l'état de l'art de la tolérance aux fautes Byzantines. Dans ce contexte, ses auteurs l'ont conçu de telle sorte qu'il puisse détecter une certaine proportion de fautes Byzantines. Les fautes Byzantines auxquelles il s'intéresse sont des fautes dites observables. Ce sont des fautes qui affectent causalement un nœud correct du système. Elles suscitent une attention particulière car dans un environnement collaboratif, un nœud fautif peut couvrir les traces d'un autre nœud fautif affectant par la suite un nœud correct. En reprenant l'exemple de la figure 3.1, page 10 (l'application de vidéo en direct), une faute observable apparait dans les cas suivants :

- le nœud i envoie un paquet erroné au nœud j ;
- le nœud i refuse d'envoyer le morceau $u1$ au nœud j ;
- le nœud i s'associe avec un autre nœud k pour tricher ;

5.1.2 Modèle de système

Chaque nœud i du système est modélisé comme un assemblage d'une machine à états S_i, d'un détecteur de fautes D_i et d'une application A_i. S_i et D_i communiquent ensemble et reprennent toutes les étapes de *PeerReview*. En fonction de son analyse, le détecteur indique trois statuts possibles que peut avoir un nœud : *exposé* (un nœud j a ce statut lorsque i a une preuve de dysfonctionnement contre j), *suspecté* (un nœud j a ce statut lorsque i attend pendant longtemps un message de sa part sans le recevoir) et *fiable* (un nœud j a ce statut lorsqu'il passe par tous les tests effectués). La figure 5.1 illustre ce modèle de système.

5.1.3 Hypothèses

Comme tout protocole s'exécutant sur un système distribué, *PeerReview* selon ses auteurs nécessite aussi des hypothèses. Ces hypothèses sont :

1. la machine à états S_i est déterministe c'est à dire à chaque entrée correspond une seule sortie unique ;
2. un message envoyé par un nœud correct à un autre est reçu par ce dernier à la longue, s'il est retransmis suffisamment souvent ;
3. les nœuds utilisent une fonction de hachage ayant les bonnes propriétés (sens unique, résistance aux collisions)

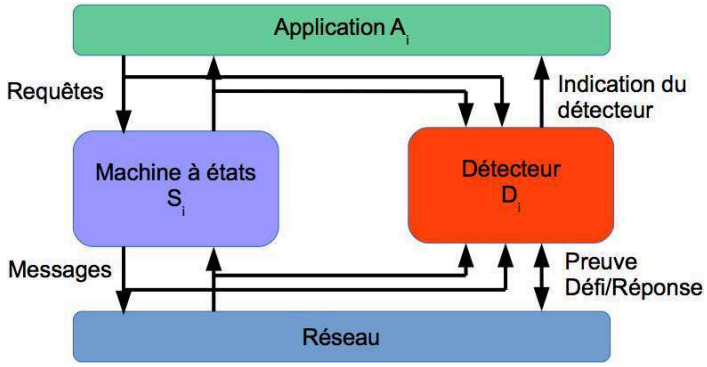


FIGURE 5.1 – Modèle de système du protocole *PeerReview*

4. chaque nœud a une paire de clés publique/privée liée à un identifiant du nœud ; les nœuds peuvent signer les messages et les nœuds fautifs ne peuvent pas forger la signature d'un nœud correct ;
5. chaque nœud a accès a l'implémentation de référence de tous les S_i ; cette implémentation peut faire une copie instantanée des états d'un nœud et ces états peuvent être initialisés à partir d'une copie donnée ;
6. pour chaque nœud i, il y a au moins un nœud correct appartenant à l'ensemble $i \cup m(i)$ ($m(i)$ est l'ensemble des surveillants de i) qui fait les études de vérification ; sans cette hypothèse le protocole *PeerReview* ne fonctionne pas correctement ;

Les hypothèses 1 à 3 sont communes à celles utilisées dans les techniques de réplication de machines à états [17] et à celles des protocoles de la tolérance aux fautes Byzantines [43].

5.1.4 Propriétés

Afin d'assurer au mieux la notion de responsabilité, *PeerReview* garantit comme la plupart des protocoles responsables, deux propriétés :

complétude elle s'assure que si un nœud devient fautif alors il est ultérieurement exposé par tous les nœuds corrects ;

précision cette propriété garantit qu'aucun nœud correct n'expose un autre nœud correct ;

5.1.5 Sous-protocoles

Considérons un système composé de N nœuds, chacun adoptant le modèle de système décrit dans la section 5.1.2. En se basant sur les hypothèses décrites précédemment et l'architecture classique d'un système responsable présentée dans le chapitre précédent, les nœuds s'engagent à être responsables de leurs actes en exécutant les protocoles suivants :

5.1.5.1 Engagement

Ce protocole assure que l'expéditeur (respectivement le destinataire) d'un message obtient une preuve vérifiable que le destinataire (respectivement l'expéditeur) a enregistré une transmission dans son historique sécurisé comme illustré sur la partie gauche de la figure 5.2. En effet, lorsqu'un nœud i envoie un message m à un nœud j, il ajoute une entrée e_k dans son historique et génère l'authentificateur correspondant $\alpha^k_{\sigma_i}$. Cet authentificateur est attaché au message m. A la réception des messages par j, ce dernier vérifie les authentificateurs, ajoute une nouvelle entrée correspondant à la réception des messages en question et génère l'authentificateur correspondant. Le nœud j envoie ensuite un message d'acquittement à i avec la preuve de l'enregistrement de m dans son historique.

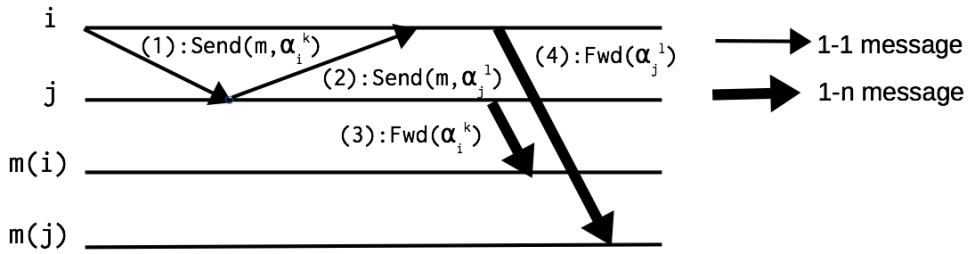


FIGURE 5.2 – Protocole d'engagement et de cohérence

5.1.5.2 Cohérence

Le protocole de cohérence permet de vérifier si chaque nœud maintient une unique version linéaire de l'historique et qui est cohérente avec tous les autentificateurs qu'il possède. Et si ce n'est pas le cas, le nœud ayant généré les authentificateurs est exposé par un surveillant correct. Ce protocole est illustré sur la partie droite de la figure 5.2. En effet, chaque nœud recevant un authentificateur (sur la figure i et j reçoivent tous les deux un authentificateur) d'un autre nœud doit le transférer aux surveillants de ce dernier. Dans la figure, le nœud i (respectivement j) transfère l'authentificateur $\alpha^l_{\sigma_i}$ (respectivement $\alpha^k_{\sigma_j}$) aux moniteurs du nœud j (respectivement moniteurs du nœud i). Cela permet aux moniteurs de collecter des preuves vérifiables d'envoi ou de réception de tous les messages du nœud qu'ils surveillent.

5.1.5.3 Audit

Périodiquement chaque surveillant met au défi les nœuds qu'il surveille, en leur demandant de lui renvoyer toutes les entrées comprises entre les numéros de séquence de deux authentificateurs qu'il choisit. Comme illustré sur la figure 5.3, chaque nœud de l'ensemble $m(i)$ envoie une requête d'audit au nœud i. Ainsi en utilisant les entrées envoyées par le nœud i, chaque surveillant extrait tous les authentificateurs que le nœud

i a reçu d'autres nœuds et qui apparaissent dans son historique. Ensuite il transfère ces authentificateurs aux surveillants des nœuds concernés. Par exemple sur la figure 5.3 les authentificateurs signés par le nœud j et ceux signés par le nœud x sont transférés respectivement aux surveillants du nœud j et du nœud x. Ainsi avec ces données, chaque surveillant s'assure que l'historique du nœud surveillé est issu d'une exécution correcte du protocole.

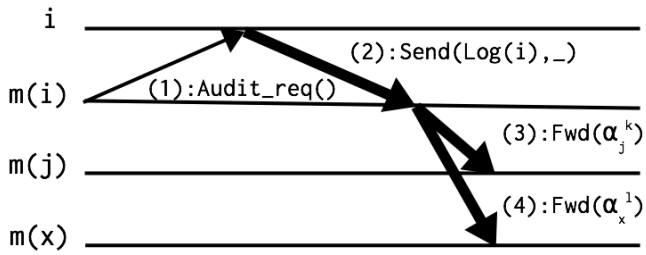


FIGURE 5.3 – Protocole d'audit

5.1.5.4 Challenge-Réponse

Dans un protocole s'exécutant sur un système distribué, il est parfois difficile de faire la différence entre un un problème de réseau et un nœud fautif qui ne répond pas à un message venant d'un nœud correct. Et cela, malgré les fortes hypothèses de synchronie faites. Ainsi le protocole Challenge-Réponse est utilisé dans *PeerReview* pour résoudre ce problème. Ce protocole fonctionne comme décrit sur la figure 5.4. En effet sur cette figure, si un nœud i attend durant une longue durée un message en provenance d'un autre nœud j, il le suspecte. Par conséquent, il crée un défi pour le nœud j et envoie ce défi aux surveillants de j. Ces surveillants auront pour tâche de faire suivre ce défi au nœud j. Si le nœud j ne répond pas, alors il est suspecté par ses surveillants. Et la suspicion persiste tant que j reste sans réponse.
PeerReview distingue deux types de challenge : le *challenge d'audit* et le *challenge d'envoi*. Le premier type est composé de deux authentificateurs (avec des numéros de séquence ordonnés). Après la vérification des signatures de ces authentificateurs, tout nœud correct a suffisamment de preuves pour se convaincre qu'un nœud i est fautif ou que l'historique de i contient, les entrées correspondant aux deux authentificateurs. Si i est correct, alors il répond au challenge en envoyant le morceau de son historique compris entre les deux numéros de séquence.
Le second type est composé d'un message simple m. A l'issue de l'envoi de m par un nœud i à un nœud j, j doit confirmer la réception sinon il sera suspecté.

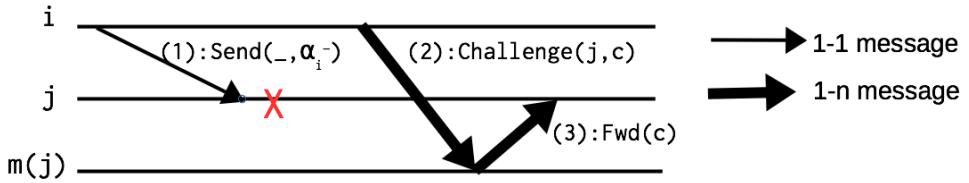


FIGURE 5.4 – Protocole de challenge/réponse

5.1.5.5 Transfert de preuve

Ce protocole a pour rôle de garantir que chaque nœud correct collecte finalement les mêmes éléments de preuve sur les autres nœuds du système. Plus précisément comme illustré dans la figure 5.5, le nœud i récupère périodiquement les défis collectés par les surveillants de chaque autre nœud j avec qui il interagit (par exemple ses partenaires directs). Il rejoue ensuite ces défis et obtient finalement le même résultat que les surveillants de j ont obtenu concernant j.

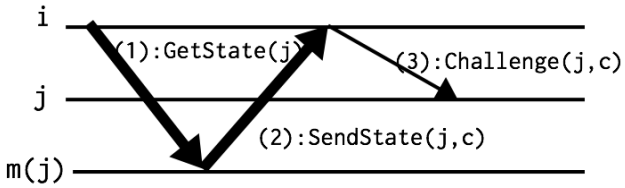


FIGURE 5.5 – Protocole de transfert de preuve

Le protocole *PeerReview* atteint ainsi son objectif à l'aide de ces sous-protocoles. Pour rappel, l'objectif principal de *PeerReview* est de garantir la notion de responsabilité avec les bonnes propriétés (complétude, précision) au sein de l'application qu'il surveille.
Cependant, il a un défaut majeur qui est dû à l'effet des comportements rationnels. En effet, *PeerReview* détecte les comportements rationnels au niveau de l'application qu'il surveille mais il est incapable de détecter ces comportements au sein de certains de ses sous protocoles durant son exécution. Pour mieux illustrer ce problème, dans les deux prochaines sections nous allons décrire les différentes déviations rationnelles dans *PeerReview*, et mesurer l'impact des rationnels sur la performance d'une application lorsqu'elle est surveillée par *PeerReview* en présence de comportements rationnels.

5.2 Déviations rationnelles dans *PeerReview*

Nous analysons les déviations rationnelles que les nœuds sont susceptibles de faire dans *PeerReview*. Pour cela nous prenons les sous protocoles cas par cas et décrivons les déviations possibles.

Engagement dans ce protocole, un nœud rationnel peut ne pas attacher certains authentificateurs aux messages qu'il envoie à ses partenaires durant l'exécution de l'application.

Cohérence durant l'exécution de ce protocole, un nœud rationnel recevant un authentificateur de la part d'un autre nœud j peut refuser de transférer cet authentificateur aux moniteurs du nœud j, dans le but de conserver sa bande passante (sur la figure 5.2, le nœud i peut ignorer l'étape 4).

Audit dans ce protocole, un surveillant rationnel peut parfois refuser d'envoyer des requêtes d'audit aux nœuds qu'il surveille afin de conserver sa bande passante ou ses ressources de calcul (par exemple sur la figure 5.3, un nœud de l'ensemble $m(i)$ peut ignorer l'étape 1). De plus, il pourrait envoyer des requêtes d'audit et affirmer qu'un nœud audité est correct sans effectuer toutes les étapes de vérifications. Enfin, pendant la phase d'audit, un moniteur rationnel peut refuser d'envoyer les authentificateurs extraits de l'historique du nœud qu'il surveille aux moniteurs des nœuds qui les ont générés (par exemple sur la figure 5.3, un nœud de l'ensemble $m(i)$ peut ignorer les étapes 3 et 4).

Challenge-Réponse dans ce protocole, un nœud rationnel peut éviter de mettre au défi des nœuds même s'il attend pendant une longue durée des messages de leur part (par exemple, il peut ignorer l'étape 2 de la figure 5.4). Les surveillants rationnels peuvent ignorer aussi le transfert d'un défi au nœud suspecté (par exemple, un nœud de l'ensemble $m(i)$ peut ignorer l'étape 3 de la figure 5.4).

Transfert de preuves durant ce protocole, un nœud rationnel peut refuser de récupérer des défis concernant d'autres nœuds ou d'exécuter les défis collectés tout en espérant que d'autres nœuds prendront soin d'expulser les nœuds fautifs dans le système (par exemple, le nœud i peut ignorer l'étape 1 et 3 sur la figure 5.5).

5.3 Impact des nœuds rationnels

Considérons un système dans lequel les nœuds peuvent être corrects, rationnels ou Byzantins. Comme introduit dans le chapitre 3, les nœuds corrects suivent le protocole, les nœuds Byzantins peuvent se comporter de façon arbitraire, et quant aux nœuds rationnels, ils essayent de maximiser leur bénéfice selon une fonction d'utilité connue. La conception du protocole *PeerReview* est basée sur une hypothèse très forte qui est l'existence d'au moins un nœud correct dans l'ensemble des surveillants qui exécute correctement toutes les phases de surveillance. Dans cette section, nous allons lever cette hypothèse et considérer que chaque surveillant peut se comporter de façon rationnelle s'il y voit un intérêt.
Notre objectif ici est de montrer sous ces conditions que les nœuds qui exécutent le protocole *PeerReview* peuvent ignorer certaines étapes de vérifications durant le protocole d'audit sans être détectés. De tels comportements peuvent impacter de façon dramatique les performances de l'application surveillée.

Ainsi pour évaluer l'impact des nœuds rationnels sur le protocole *PeerReview*, nous avons fait deux expériences. Dans la première expérience nous avons déployé sur cent nœuds l'application SplitStream [44], un protocole de diffusion efficace basé sur une structure sous forme d'arbres, surveillée par *PeerReview*. Dans la seconde expérience, nous avons déployé sur cent nœuds le protocole Onion routing [45], surveillé par *PeerReview*. Dans les deux expériences, nous avons utilisé les mêmes configurations expérimentales comme celles décrites dans le chapitre 8, et si un nœud rationnel remarque que ses surveillants sont rationnels (par exemple, ils ne demandent pas à auditer son historique), il se comporte aussi de façon rationnelle vis à vis des deux applications (SplitStream et Onion routing), en supprimant les messages qu'il reçoit ou qui ne lui sont pas destinés (messages à transférés).
Nous mesurons le pourcentage de messages perdus en fonction de la proportion de nœuds rationnels dans le système. Les résultats présentés par la figure 5.6 montrent qu'en présence de 30% de nœuds rationnels, les nœuds corrects exécutant le protocole SplitStream observent 54% de perte de messages. De façon similaire, dans l'application Onion routing les nœuds corrects constatent des pertes d'oignons allant jusqu'à 85% avec 30% de nœuds rationnels dans la configuration avec 5 relais.
Cette proportion de perte augmente et atteint 100% lorsque le nombre de relais augmente. Cela est dû au fait que la probabilité d'avoir des nœuds rationnels comme relais dans un chemin augmente proportionnellement avec le nombre de relais constituant ce chemin.

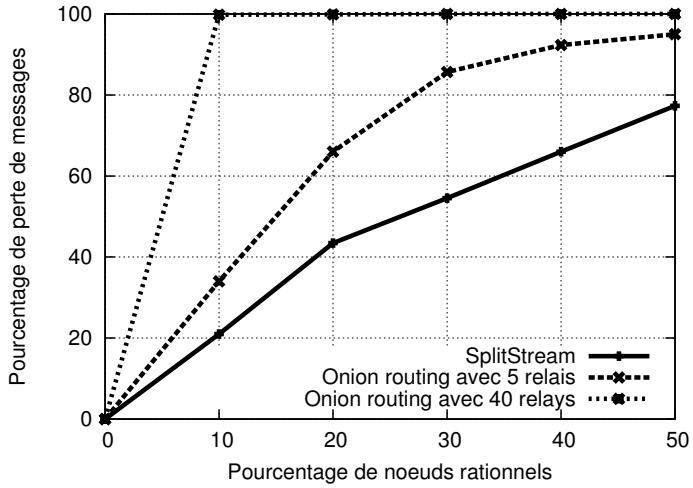


FIGURE 5.6 – Impact des nœuds rationnels sur les applications SplitStream et Onion routing surveillées par *PeerReview*.

5.4 Conclusion

Ce chapitre a porté sur la description détaillée du protocole *PeerReview*, le premier protocole utilisant une solution générique logicielle pour renforcer la responsabilité dans les systèmes distribués. Il utilise un historique sécurisé pour enregistrer les échanges entre les nœuds d'un système distribué, et est constitué de plusieurs sous-protocoles, chacun ayant un rôle spécifique. Ainsi, son protocole d'engagement s'assure que les enregistrements ont été bien effectués dans les historiques, son protocole de cohérence permet de vérifier si un historique n'a pas été altéré, son protocole d'audit a pour rôle de vérifier si un historique est cohérent avec une exécution correcte du protocole, et son protocole de transfert de preuve a pour rôle de récupérer les défis afin de permettre à un nœud de s'informer sur l'état d'un autre nœud du système.
Tandis que *PeerReview* détecte les fautes observables y compris les fautes rationnelles au sein de l'application qu'il surveille, il est vulnérable aux comportements rationnels au niveau de ses propres étapes comme décrit dans la section précédente. C'est dans ce contexte que nous avons conçu le protocole *FullReview* pour pallier à ce problème, et il constitue la contribution principale de cette thèse.
L'objet du prochain chapitre sera ce protocole et nous allons expliquer en détails comment il a été conçu, voir comment avec *FullReview* les nœuds rationnels seront forcer à participer aux différentes étapes de vérification.

6

FullReview : La notion de responsabilité en présence de comportements rationnels

Sommaire

6.1 Appliquer *PeerReview* à lui même 40
6.2 Modèle du système 40
6.3 Modèle de fautes 41
6.4 Hypothèses 42
6.5 Vue d'ensemble du protocole 42
6.6 Description détaillée 43
6.6.1 Protocoles de vérification d'historique 44
6.6.2 Protocoles de détection de fautes 48
6.7 Résistance aux comportements rationnels 50
6.7.1 Protocole d'audit 50
6.7.2 Protocole P augmenté 51
6.7.3 Envoi des requêtes d'audit 51
6.7.4 Traitement des requêtes d'audit 52
6.7.5 Traitement des fautes d'omission 53
6.8 Conclusion 53

Le traitement des comportements rationnels dans le domaine de la tolérance aux fautes a été au cœur de la recherche dans la dernière décennie. Cependant, parmi toutes les solutions logicielles existantes, aucune ne traite à proprement dit les comportements rationnels, ou si elle les traite c'est seulement au niveau de l'application surveillée. Comme vu dans le chapitre précédent *PeerReview* illustre bien l'impact de ces comportements sur la performance de l'application surveillée. La solution triviale à laquelle nous pourrons immédiatement penser pour résoudre ce problème est l'application de

PeerReview à lui même c'est à dire faire surveiller les étapes de vérification de *PeerReview* par une instance de *PeerReview*. Nous verrons dans ce chapitre que cette solution ne marche pas.
Ainsi pour résoudre ce problème de comportements rationnels, nous partons du protocole *PeerReview* et rajoutons une couche au dessus de ce dernier pour forcer les nœuds rationnels à participer aux étapes de surveillance.
Dans ce contexte, nous proposons *FullReview*, le premier protocole logiciel et générique qui renforce la notion de responsabilité et tolère les comportements rationnels non seulement au niveau de l'application surveillée mais aussi durant ses phases de vérification.
FullReview se base sur les techniques de la théorie des jeux pour forcer et motiver les nœuds rationnels à suivre les différentes étapes de son propre protocole et de l'application à laquelle il est appliqué. De plus, nous prouvons que ce protocole est un équilibre de Nash, c'est à dire que les nœuds rationnels n'ont aucun intérêt à dévier du protocole.
Ce chapitre décrit en détails *FullReview* en partant du modèle de système, du modèle de fautes, des hypothèses et en passant par ses sous protocoles.

6.1 Appliquer *PeerReview* à lui même

Dans cette section nous allons montrer qu'appliquer *PeerReview* à lui même pour faire face aux comportements rationnels est limitant.
Considérons un système distribué adoptant l'architecture classique d'un système responsable vu précédemment et sur lequel s'exécute une application surveillée par *PeerReview*. Soit i un nœud de ce système, z un nœud de $m(i)$ (l'ensemble des surveillants de i). Dans *PeerReview*, le nœud z audite périodiquement l'historique du nœud i. La question que nous nous posons est de savoir s'il est possible de vérifier que z a effectué toutes ses tâches de vérifications concernant i en utilisant une instance de *PeerReview*.
Pour répondre à cette question, considérons k un nœud appartenant à $m(z)$ (l'ensemble des surveillants de z), alors le nœud k va auditer l'historique de z et pour cela il a besoin de l'historique de i qui est en cours de vérification au niveau du nœud z. Et pour vérifier si le nœud k a effectué ses tâches de vérifications, un de ses surveillants aura besoin de l'historique du nœud qu'il a surveillé. Finalement, nous assistons à un processus récursif qui à la longue devient très coûteux et moins pratique. D'où l'intérêt d'un nouveau protocole plus pratique pour contrôler ses étapes de vérifications.

6.2 Modèle du système

Le système cible comme illustré sur la figure 6.1 est composé de deux protocoles : le protocole surveillé que nous désignons par P et le protocole qui surveille désigné par M. Les hypothèses sur P et M sont décrites à la section 6.4.

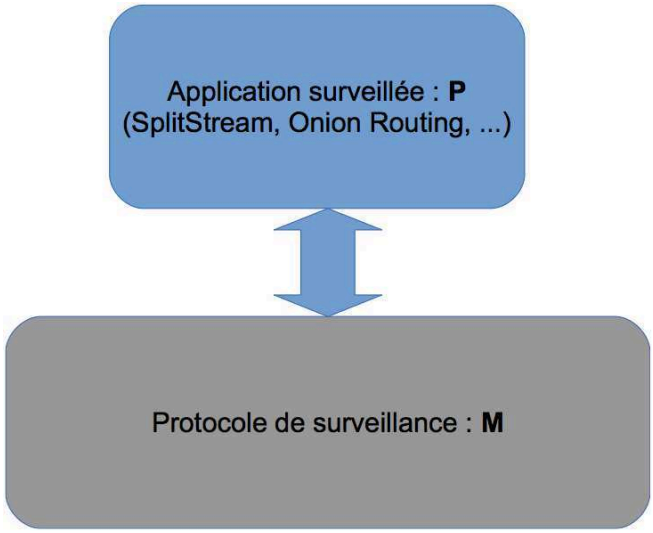


FIGURE 6.1 – Modèle de système de *FullReview*.

6.3 Modèle de fautes

Nous considérons dans le système une proportion fixe de nœuds Byzantins qui peuvent prendre des décisions arbitraires. Ils peuvent dévier du protocole P ou M pour n'importe quelle raison (par exemple, une panne, un bug, une menace). En outre, nous considérons un nombre illimité de nœuds rationnels (modèle BAR). Ces nœuds ont pour objectif de maximiser leur bénéfice selon une fonction d'utilité connue. Les nœuds rationnels dévient du protocole M s'ils y voient leur bénéfice dans cette déviation. Le bénéfice est calculé grâce à une fonction d'utilité définie comme suit :

1. à la communication : envoyer/recevoir aussi peu de messages de surveillance aux/d' autres nœuds que possible
2. aux calculs : faire le moins de calculs possibles relatifs à la surveillance d'autres nœuds

Par ailleurs, nous supposons que les nœuds sont averses au risque. Cela veut dire qu'avant de dévier, un nœud rationnel estime la probabilité d'être détecté dans le futur. Si cette probabilité est plus grande que zéro, il (un nœud rationnel) suit le protocole. Cette hypothèse est fréquemment utilisée dans les systèmes BAR-résistant [5]. Elle a particulièrement un sens dans les systèmes ayant la notion de responsabilité car la détection d'une déviation dans ces systèmes entraine l'éviction du nœud fautif du système. Toutefois, dans les systèmes où la pénalité est faible (par exemple une diminution de la valeur de réputation), il est plus commode de considérer différents modèles de comportements rationnels (risque affine par exemple). Cela n'est pas le cas de notre système.
Le modèle BAR suppose aussi que les nœuds rationnels joignent le système et y restent

pendant longtemps afin de chercher un bénéfice. De plus, les nœuds rationnels ne forment pas de coalition et supposent que les autres nœuds sont corrects.

6.4 Hypothèses

Comme dans *PeerReview*, nous supposons une identification cryptographique des nœuds. Plus précisément, chaque message envoyé sur le réseau est signé en utilisant la clé privée de l'expéditeur. Nous supposons que les primitives cryptographiques ne peuvent être falsifiées et que les fonctions de hachage sont résistantes à la collusion. En outre, nous supposons que les messages envoyés par un expéditeur à un destinataire donné sont toujours reçus s'ils sont retransmis infiniment souvent. Nous supposons que les nœuds ont une implémentation de référence de P sous forme de machines à états déterministes et qui peut être initialisée avec des points de contrôle (checkpoints) et dans laquelle nous pouvons injecter des entrées afin d'obtenir des sorties qui correspondent à celles qui sont enregistrées dans les historiques des nœuds.

6.5 Vue d'ensemble du protocole

Considérons un ensemble de N nœuds exécutant le protocole P défini comme étant un ensemble de machines à états déterministes. Dans *FullReview*, les nœuds sont structurés selon l'architecture classique de la notion de responsabilité décrite dans le chapitre 4. Chaque surveillant exécute le protocole de surveillance M aussi décrit comme un ensemble de machines à états déterministes. L'objectif du protocole *FullReview* est de forcer non seulement les nœuds rationnels à exécuter toutes les étapes des deux protocoles P et M mais aussi de détecter les nœuds Byzantins quand ils dévient du protocole P ou M. Pour atteindre cet objectif, chaque nœud i enregistre dans son historique sécurisé toutes ses interactions relatives aux deux protocoles P et M. Ensuite les surveillants de i (les nœuds dans $m(i)$) effectuent périodiquement un certain nombre de vérifications sur cet historique. Ces vérifications qui apparaissent sous forme de diagramme dans la figure 6.2, permettent à chaque surveillant d'obtenir une preuve quant à l'état du nœud i (correct ou suspect). Plus précisément, chaque nœud dans $m(i)$ commence par vérifier que i n'a pas modifié son historique (c'est à dire que le nœud n'a pas supprimé les entrées insérées précédemment). Nous appelons cette vérification qui apparait en premier sur le diagramme, la *vérification de cohérence* de l'historique. Nous expliquons comment cette vérification est effectuée dans la section 6.6.1.2. De plus, chaque nœud dans $m(i)$ vérifie que i garde un unique historique pour tous ses partenaires. Nous appelons cette vérification qui apparait en second sur le diagramme, la *vérification de consistance*. Ces deux vérifications en dessus du diagramme sont critiques dans la mise en place de la notion de responsabilité. En effet, si un nœud ajoute ou supprime des entrées dans son historique ou garde plusieurs versions d'un

historique, il pourrait dévier du protocole sans être détecté. Nous expliquons comment cette vérification est effectuée dans la section 6.6.1.3.

En outre, chaque nœud de $m(i)$ vérifie que les *motifs de communication* apparaissant dans l'historique de i sont cohérents avec les machines à états de M et P (troisième vérification sur le diagramme). Cette vérification s'assure que l'historique de i contient une séquence de messages qui reflètent un comportement correct. Par exemple, un historique correct devrait contenir des demandes périodiques de i à l'ensemble de ses surveillants, c'est à dire des nœuds dans $m(i)$. L'absence de tels messages périodiques reflète un comportement fautif. Nous expliquons comment ces vérifications sont effectuées dans le protocole *FullReview* dans la section 6.6.1.4.

Cependant, un historique qui exhibe une séquence correcte de messages n'est pas suffisant pour garantir un comportement correct. Ainsi, la dernière vérification qui est effectuée par les surveillants de i est d'évaluer si l'historique de i correspond à un historique issu d'une *exécution correcte* des protocoles P et M ou pas. Vérifier la conformité de l'historique de i avec celui issu d'une exécution correcte de P s'effectue comme dans le protocole *PeerReview*, c'est à dire en ré-exécutant le code du protocole P, en utilisant son implémentation de référence. En effet, les entrées présentes dans l'historique de i sont injectées dans l'implantation de référence de P, et les sorties obtenues sont ensuite comparées avec les sorties présentes dans l'historique. Les sorties qui ne correspondent à celles présentes dans l'historique constitueraient une preuve que i n'exécute pas correctement P.

Il n'est pas possible de faire la même vérification pour M. En effet, comme décrit dans la section 6.1, ré-exécuter le code de surveillance est une tâche récursive et nécessite que l'historique d'un nœud contienne l'historique de tous les autres nœuds qui sont en lien avec lui dans le graphe de surveillance (qui peut être tous les nœuds du système). Pour éviter ce problème, nous identifions toutes les opérations effectuées dans le protocole M et nous nous assurons que ces opérations sont effectuées par un ensemble de nœud en parallèle. Le résultat de chaque opération est ensuite collecté par les différents nœuds participants au protocole et ensuite envoyé aux surveillants des nœuds. Ces derniers comparent le résultat des calculs faits par les nœuds qu'ils surveillent avec celui obtenu avec d'autres nœuds. Comme les nœuds rationnels ne veulent pas être suspectés par les nœuds corrects, ils feront toujours les calculs correctement. Sur le diagramme de la figure 6.2, cette dernière vérification est effectuée avant la ré-exécution du code de P qui est très coûteuse. Les détails de comment le protocole *FullReview* vérifie que les nœuds ont correctement fait tous les calculs relatifs aux deux protocoles P et M sont décrits dans la section 6.6.1.5.

6.6 Description détaillée

Dans cette section, nous allons décrire en détails le protocole *FullReview* en présentant ses deux parties majeures à savoir : les protocoles de vérification d'historique (section 6.6.1) et les protocoles de détection de fautes (section 6.6.2). Finalement, nous

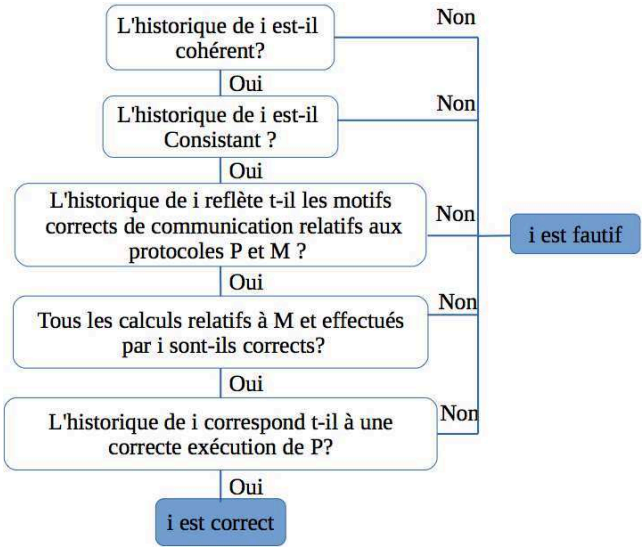


FIGURE 6.2 – Diagramme de décision des surveillants du protocole *FullReview*.

fournissons des informations prouvant que notre protocole est un équilibre de Nash (section 6.7).

6.6.1 Protocoles de vérification d'historique

Les protocoles de vérification d'historique comprennent : le protocole d'audit rationnel-résistant, la vérification de cohérence, la vérification de consistance, la vérification des motifs de communication et la vérification des calculs.

6.6.1.1 Le protocole d'audit rationnel-résistant du protocole FullReview

En utilisant l'historique sécurisé décrit plus haut, un nœud j surveillant le comportement d'un nœud i effectue un ensemble de vérifications pour évaluer si le nœud i est correct ou pas en suivant le diagramme de la figure 6.2. Toutefois, les surveillants rationnels peuvent être tentés de ne pas effectuer ces vérifications. Dans le but de forcer les surveillants à les effectuer, nous rendons les audits proactifs. En effet, nous divisons le temps en rondes et donnons la responsabilité à chaque nœud de demander périodiquement (par exemple à la fin de chaque ronde) à ses surveillants d'auditer son historique selon le diagramme de la figure 6.3 (le message Audit_req envoyé par i à ses surveillants $m(i)$). Ensuite chaque surveillant effectue les vérifications demandées et produit un certificat de conformité si le nœud i passe toutes ces vérifications. Dans le cas échéant, les surveillants de i constituent une preuve de dysfonctionnement de i, que tout nœud correct peut recalculer. Ce certificat est ensuite utilisé par i au début de la prochaine ronde dans l'objectif de communiquer avec ses partenaires. Sans ce certificat,

les partenaires de i refuseront d'interagir avec i. Notons que certains des surveillants de i peuvent ne pas répondre à une demande d'audit parce qu'il y a une panne ou un refus d'auditer l'historique de i. Nous décrivons comment traiter cette situation dans la section 6.6.2.3. Finalement, après avoir collecté les résultats d'audit produits par ses surveillants (Audit_resp message), i transfère le résultat agrégé aux surveillants de chacun de ses surveillants (Fwd_outcome message). Cette dernière étape est très importante pour les surveillants des surveillants de i (c'est à dire $m(m(i))$), car cela leur permet de vérifier si les nœuds qu'ils surveillent font correctement leur travail de surveillance ou pas. Des détails supplémentaires sur cette vérification sont donnés dans la section 6.6.1.5. Dans la suite nous décrivons en détail l'ensemble des vérifications faites par les surveillants de chaque nœud afin d'évaluer s'il est correct ou pas.

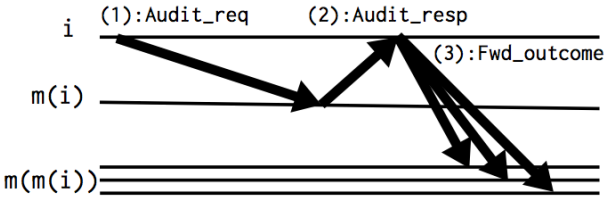


FIGURE 6.3 – Protocole d'audit de *FullReview*.

6.6.1.2 Vérification de cohérence

Elle permet de vérifier qu'un nœud n'a pas altéré son historique. Considérons un nœud j qui surveille un nœud i. Si j obtient une paire d'authentificateurs α_i^0 et α_i^k correspondant aux entrées e_0 et e_k respectivement dans l'historique de i, il peut demander à i ses entrées $e_0, ..., e_k$ et recalcule les valeurs $h_0, ..., h_k$ comme décrit dans la section 4.2.2. Si le h_k calculé est différent de celui que j a, il (le nœud j) peut accuser i de modifier son historique. De plus, j peut convaincre tout autre nœud correct du mauvais comportement de j en leur envoyant les authentificateurs signés α_i^0 et α_i^k avec les entrées venant de i. Pour effectuer ce type de vérification, chaque nœud doit enregistrer chaque message qu'il envoie par rapport aux deux protocoles P et M, et envoie les authentificateurs correspondants à ses partenaires. En outre, chaque nœud doit également transférer les authentificateurs reçus aux surveillants de ses partenaires. Cependant, les nœuds rationnels peuvent être tentés de ne pas suivre toutes ces étapes, c'est à dire éviter d'attacher les authentificateurs aux messages qu'ils envoient et/ou de transférer les authentificateurs reçus aux surveillants des partenaires. Nous montrons comment traiter ce problème dans la section 6.6.1.4.

6.6.1.3 Vérification de consistance

Un nœud i peut être tenté de maintenir plusieurs historiques corrects (par exemple, un historique avec chacun des nœuds avec qui il interagit). Pour détecter ce type de

dysfonctionnement, un surveillant j ayant l'ensemble des authentificateurs envoyés par i à d'autres nœuds vérifie que ces authentificateurs appartiennent à un même historique. Comme la vérification de cohérence, cette vérification nécessite aussi que les nœuds attachent les authentificateurs aux messages qu'ils envoient et transfèrent ceux (authentificateurs) reçus à leurs partenaires, et que les surveillants effectuent la vérification de consistance. Nous montrons comment encourager les nœuds rationnels à faire toutes ces étapes dans la section 6.6.1.4.

6.6.1.4 Vérification des motifs de communication

Dans cette partie du protocole, un surveillant d'un nœud i doit vérifier si l'historique de i reflète les motifs de communications corrects par rapport aux machines à états de P et M. Cependant, il n'est pas possible de considérer les machines à états de ces deux protocoles séparément car les étapes de M ont besoin d'être entrelacées avec des étapes de P dans certaines situations. Par exemple, comme vu dans les vérifications de cohérence et de consistance décrites ci-dessus, les nœuds doivent envoyer les authentificateurs avec les messages relatifs à P et transférer les authentificateurs reçus avec les messages relatifs à M. Pour atteindre cet objectif, la machine à états du protocole P est automatiquement augmentée d'un ensemble de transitions obligatoires comme indiqué sur la figure 6.4. Dans cette figure, et dans toutes les figures indiquant les automates dans ce chapitre, les transitions sont labellisées comme suit : (P|M :IN|OUT :message_type) où la première partie indique si le messages appartient au protocole P ou M ; la seconde partie indique respectivement si le message est reçu ou envoyé et la troisième partie est le type du message. Cette figure montre qu'à chaque instant si un nœud s'attend à un message de la part du protocole P, il devrait : (1) à la réception d'un message, transférer les authentificateurs inclus dans les messages aux surveillants des expéditeurs (transition labellisée (M :OUT :fwd_auth)) ; ou (2) accuser l'expéditeur si le message reçu ne contient pas un authentificateur en envoyant un message d'accusation aux surveillants de l'expéditeur (transition labellisée (M :OUT :accuse)) ; ou (3) suspecter ses partenaires si ces derniers ne lui envoient pas le message attendu (transition labellisée (M :IN :timeout)). Les transitions suivant cette dernière transition sont décrites dans la section 6.6.2.3.

Augmenter toutes les transitions de P relatives à la réception des messages comme montré sur la figure 6.4 force les nœuds rationnels à attacher les authentificateurs aux messages qu'ils envoient (autrement, les nœuds qui reçoivent ces messages peuvent les accuser). En plus, il force les nœuds rationnels à transférer les authentificateurs aux surveillants de leurs partenaires (autrement, leurs surveillants peuvent les accuser de se comporter rationnellement).

En plus de vérifier si l'historique du nœud surveillé est cohérent avec la machine à états de l'automate augmenté de P, les surveillants vérifient aussi si l'historique est cohérent avec les machines à états de M relatives au protocole d'audit (décrit plus haut dans cette section) et à la gestion des pannes d'omission. Les machines à états qui concernent le protocole d'audit apparaissent sur les figures 6.5 et 6.6, et celles

relatives à la gestion des pannes d'omission sont décrites dans la section suivante. Plus précisément, l'automate de la figure 6.5 montre les motifs de communication corrects d'un nœud i demandant à un de ses surveillants un audit (transition labellisée (M :OUT :audit_req)). Après avoir envoyé sa demande d'audit, le nœud i reçoit soit une réponse de ses surveillants contenant le résultat de l'audit (transition labellisée (M :IN :audit_resp)) ou ne reçoit pas de réponse (transition labellisée (M :IN :timeout)). Dans le premier cas, le nœud i transfère le résultat de l'audit aux surveillants de ses surveillants, ce qui permet à ces derniers de vérifier que le nœud qu'ils surveillent obtient le même résultat sur l'état de i que les autres surveillants de i. Dans le dernier cas, i considère que son surveillant est en panne et traite cette panne comme décrit dans la section suivante.

L'automate de la figure 6.6 montre les motifs corrects de communication d'un surveillant j qui reçoit une demande d'audit d'un nœud i qui est surveillé (la transition labellisé (M :IN :audit_req)). Après la réception de cette demande, le nœud j effectue l'audit de l'historique de i et envoie le résultat à i (la transition labellisée (M :OUT :audit_resp)).

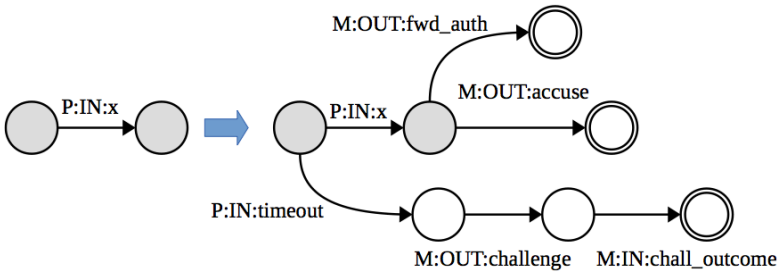


FIGURE 6.4 – Protocole P augmenté.

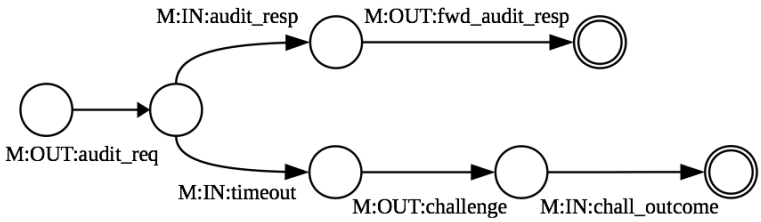


FIGURE 6.5 – Envoi des requêtes d'audit.

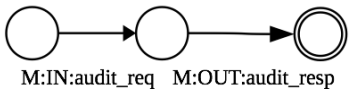


FIGURE 6.6 – Traitement des requêtes d'audit.

6.6.1.5 Vérification des calculs

Dans cette partie du protocole, chaque surveillant j dans l'ensemble des surveillants d'un nœud i vérifie que les calculs effectués par i, relatifs aux protocoles P et M sont corrects. Pour les calculs effectués par i et qui sont relatifs au protocole P, j utilise des points de vérification (copies de vérification) stockés dans l'historique de i et initialise l'implantation de référence qu'il a, avec le plus ancien point de vérification non utilisé. Ensuite, j rejoue toutes les entrées disponibles dans la portion de l'historique de i qu'il a audité et vérifie que les sorties produites par l'implantation de référence correspondent avec les sorties stockées dans l'historique. Si les sorties calculées ne correspondent à celles stockées, j accuse i de mauvais comportement. Que i passe cette vérification ou pas, j stocke le résultat de l'audit avec les authentificateurs correspondant à la portion de l'historique de i qui a été audité et envoie le résultat de l'audit à i comme décrit par le protocole d'audit (décrit plutôt dans cette section).

Contrairement aux calculs relatifs au protocole P, vérifier ceux relatifs au protocole de surveillance M ne peut pas être fait en ré-exécutant les étapes du protocole M. Nous avons démontré précédemment (section 6.1) que cela soulève un problème de récursion interminable.

Ainsi pour éviter un tel problème, nous utilisons des techniques de la théorie des jeux pour motiver et forcer les nœuds rationnels à effectuer correctement et à prendre part aux calculs relatifs au protocole M au lieu de les recalculer. Spécifiquement, comme décrit plutôt, après avoir reçu les résultats de l'audit envoyés par ses surveillants, un nœud agrège ces résultats et les transfère aux surveillants de ses surveillants. Ces nœuds reçoivent une information de type : (audited node ID, authenticators, monitor ID, outcome) pour chaque surveillant de i ayant prit part dans l'audit. Si la majorité des surveillants détecte un mauvais comportement dans l'historique de i et l'un d'entre eux, disons j ne détecte rien, alors j est accusé de mauvais comportement. Dans cette situation, j est rationnel s'il clame que i est correct sans effectuer aucune vérification ou Byzantin s'il répond arbitrairement. Comme les nœuds rationnels n'aiment pas être exclus du système, ils effectueront toujours correctement les calculs relatifs au protocole M. Plutôt, si la majorité des surveillants mais pas j considère que i est correct, alors j est considéré comme Byzantin car un nœud rationnel n'a aucun intérêt à accuser un nœud correct de mauvais comportement.

6.6.2 Protocoles de détection de fautes

FullReview gère trois types de fautes : les fautes observables, les fautes non observables et les pannes d'omission.

6.6.2.1 Détection de fautes observables

Comme vu dans le chapitre précédent les fautes observables sont celles qui affectent causalement un nœud correct d'un système. *FullReview* traite ces fautes en utilisant

les machines à états générées durant ses phases de vérifications. En effet, il rejoue l'exécution du système modélisé sous forme de machines à états et ayant comme entrées les historiques.

6.6.2.2 Détection de fautes non observables

Comme l'indique leur nom, ces fautes sont très difficiles à détecter. Un exemple de ce type de fautes est lorsqu'un surveillant x d'un nœud i d'un système revendique que l'historique de i est correct sans effectuer aucun calcul. Ainsi pour gérer ce comportement, *FullReview* se base sur la réplication des calculs (voir section 6.6.1.5). Spécifiquement, le nœud qui aura un résultat différent de la majorité des résultats renvoyés par les autres nœuds sera exposé.

6.6.2.3 Gestion des pannes d'omission

La gestion des pannes d'omission est faite dans *FullReview* comme décrit dans la figure 6.7. Plus précisément, si un nœud i attend un message d'un nœud j pour une longue durée, i suspecte j (après l'étape (1) sur la figure). Ainsi, i crée un défi (challenge) pour j et envoie ce défi aux surveillants de j (étape (2) sur la figure), qui le transfèrent à j (étape (3) sur la figure). Si j est toujours actif alors il répond au défi (étape (4)). Que le nœud j réponde ou non au défi, après un certain temps, les surveillants de j envoient un résultat à i pour résumer la situation (étape (5)).

Un nœud rationnel peut être tenté de ne pas suspecter un nœud même si il l'attend longtemps pour la réception d'un message, tout en supposant que les autres nœuds le feront. De façon similaire, un surveillant rationnel peut être tenté de ne pas transférer un défi envoyé par i à j tout en supposant que les autres surveillants vont le faire. Ces déviations ne sont pas possibles dans *FullReview* grâce à la vérification des motifs de communication effectuée par les surveillants sur les historiques des nœuds qu'ils surveillent. En effet les automates des figures 6.8 et 6.9 montrent les motifs de communication corrects qui devraient être présents dans l'historique d'un nœud quand, comme surveillant, il reçoit une plainte d'omission de panne de la part d'un de ses nœuds qu'il surveille et quand, comme nœud suspecté, il reçoit un défi de ses surveillants. L'historique d'un nœud rationnel devrait être conforme à ces automates autrement il est accusé par ses surveillants.

En plus, un nœud rationnel peut être tenté de suspecter un autre nœud au lieu d'effectuer une interaction coûteuse avec ce dernier. Pour éviter cette déviation, nous rendons le coût d'une suspicion plus cher que celui d'une interaction. Ensuite, pour éviter de saturer le système, nous adaptons ce coût à chacune des étapes des protocoles P et M. Par exemple, si envoyer un message m coûte xB en bande passante au nœud i, nous posons le coût de suspicion d'un nœud j (nœud avec qui i avait supposé qu'il a envoyé m) égal à $x + \delta B$. Comme cela, un nœud rationnel i préférera toujours envoyer m au lieu de suspecter j.

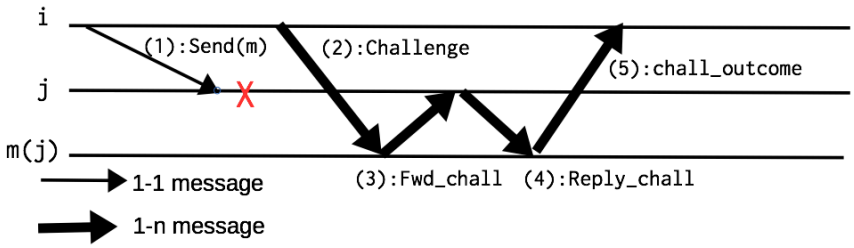


FIGURE 6.7 – Protocole de gestion de pannes d'omission de *FullReview*

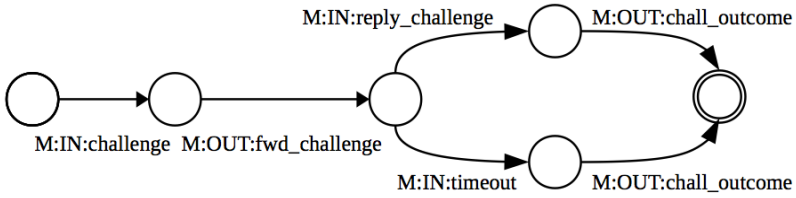


FIGURE 6.8 – Gestion des pannes d'omission.

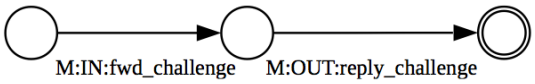


FIGURE 6.9 – Gestion des suspicions.

6.7 Résistance aux comportements rationnels

Dans cette section, nous analysons les déviations rationnelles possibles dans le protocole *FullReview*, et montrons comment motiver ou forcer les nœuds à suivre le protocole aux étapes où apparaissent ces déviations. Cela prouve par la suite que *FullReview* est un équilibre de Nash c'est à dire qu'aucun nœud rationnel n'a intérêt à dévier du protocole.
Dans *FullReview* les déviations rationnelles sont principalement au niveau du protocole d'audit, des automates générés lors des vérifications et dans le traitement des fautes d'omission. Ainsi pour chaque étape ou transition où il y a une déviation, nous allons la décrire et dire pourquoi les nœuds rationnels vont suivre le protocole plutôt que de dévier.

6.7.1 Protocole d'audit

Dans l'étape 2 du protocole d'audit *FullReview* (voir figure 6.3), après la réception d'une requête d'audit, chaque surveillant du nœud i effectue un certain nombre de vérifications nécessaires et produit par la suite un certificat de bon fonctionnement si le nœud i passe à tous ces tests.
Déviation rationnelle : Certains nœuds rationnels dans $m(i)$ (l'ensemble des surveillants de i) peuvent ne pas effectuer certaines vérifications.
Motivation : Après avoir reçu les réponses d'audit, le nœud i agrège ces réponses

et envoie ensuite le résultat final aux surveillants des surveillants de i. Ces derniers peuvent vérifier si un surveillant a bien effectué son travail d'audit ou pas. En cas de déviation, le surveillant fautif sera évicté du système et comme un nœud rationnel a peur de se faire expulser du système, il va donc faire correctement l'audit.

6.7.2 Protocole P augmenté

Cette partie concerne l'automate de la figure 6.4. Les transitions labellisées *P :IN :x*, *M :OUT :challenge* sont sujets chacune à une déviation rationnelle.
Déviation rationnelle : Dans la première transition, les nœuds rationnels peuvent refuser de transférer les authentificateurs qu'ils ont reçus, aux surveillants de leurs partenaires. Dans la seconde transition, un nœud rationnel peut ne pas suspecter un autre nœud même s'il attend durant une longue durée un message en provenance de ce nœud. Ainsi il suppose que le nœud est Byzantin et qu'un autre le suspectera et le mettra au défi.
Motivation : Pour la première transition, comme les nœuds rationnels suivent toujours le protocole d'engagement, ils supposeront que leurs partenaires sont corrects et que ces derniers auraient des preuves nécessaires, si jamais ils dévient. Quant à la seconde transition, un nœud rationnel n'a aucun intérêt à dévier car les traces apparaîtront dans son historique et il risquera une éviction de la part de ses surveillants, après l'exécution des protocoles de vérification d'historique (section 6.6.1).

6.7.3 Envoi des requêtes d'audit

A ce niveau, les déviations rationnelles correspondent à celles qui apparaissent sur l'automate représenté sur la figure 6.5 aux transitions $1, 2, 3, 5$ et 6.
Déviation rationnelle : Les déviations présentes ci-dessous correspondent respectivement à ces six transitions.

1. Un nœud rationnel peut ne pas demander à être audité par ses surveillants.
2. Un surveillant rationnel peut ne pas répondre à une demande d'audit de la part d'un nœud qu'il surveille.
3. Un nœud rationnel peut ne pas faire suivre le résultat de l'agrégation des réponses d'audit aux surveillants de ses surveillants.
4. Un nœud rationnel peut ne pas suspecter un surveillant qui n'a pas répondu à une requête d'audit.
5. Un nœud rationnel peut ne pas répondre à un défi lancé par un autre nœud.

Motivation : Les motivations ci-dessous incitent les nœuds rationnels à suivre respectivement ces transitions.

1. Si un nœud ne présente pas un certificat de conformité à la prochaine exécution du protocole, ses partenaires refuseront d'interagir avec lui. Cela pousse un nœud rationnel à toujours demander à être audité périodiquement par ses surveillants.

2. Si un nœud rationnel ne répond pas à une demande d'audit, il sera suspecté par le nœud qui a fait la demande. Comme le traitement d'une suspicion est plus coûteux qu'un envoi de réponse et peut conduire à une éviction du nœud en question, un nœud rationnel répondra toujours à une demande d'audit.
3. Grâce à la vérification des motifs de communication par les surveillants, la déviation faite à la transition 3 sera détectée. En effet, si l'historique d'un nœud contient la trace de la réception d'une réponse audit, et que cette trace n'est pas suivie par une autre trace indiquant qu'il y a eu un transfert de message (*forwarding*), alors il sera accusé de mauvais comportement (*misbehaviour*) et risque une éviction. Ainsi un nœud rationnel fait suivre toujours une réponse d'audit.
4. Après l'envoi d'une requête d'audit, si l'historique d'un nœud ne contient ni la trace de la réception d'une réponse, ni la trace de l'envoi d'un challenge aux surveillants du nœud qui n'a pas répondu, alors il sera accusé de mauvais comportement par ses surveillants au prochain audit. Comme un nœud rationnel ne veut pas être accusé, il mettra toujours au défi les nœuds qui ne répondent.
5. Un nœud rationnel qui ne répond pas à un défi risque de façon imminente une éviction. Comme un nœud rationnel ne veut être évicté du système, il répondra toujours à un défi.

6.7.4 Traitement des requêtes d'audit

Les déviations rationnelles et les motivations présentées dans cette section concernent l'automate illustré sur la figure 6.6 au niveau des transitions $1, 2$, et 3.
Déviation rationnelle : Les déviations suivantes correspondent respectivement à celles existant dans les trois transitions ci-dessus.

1. Un nœud rationnel peut ignorer la réception d'une requête d'audit.
2. Un nœud rationnel peut éviter d'effectuer des calculs durant la phase d'audit dans le but de sauvegarder son unité de calcul (*CPU*).
3. Un nœud rationnel peut refuser d'envoyer une réponse d'audit afin de sauvegarder sa bande passante.

Motivation : Les motivations correspondant à ces déviations respectives sont ci-dessous.

1. Si un nœud ignore la réception d'une requête d'audit, il est suspecté par le nœud audité. Comme un nœud rationnel ne veut pas être suspecté, il va toujours considérer une requête d'audit.
2. Si un nœud rationnel ment sur la réponse d'un audit, il risque une éviction si les nœuds corrects arrivent à une réponse différente (grâce à la réplication des calculs). Comme un nœud rationnel ne veut pas être exclu du système, il effectue correctement les vérifications comme prescrites dans le protocole.

3. Si un nœud rationnel esquive l'envoi d'une réponse à une requête d'audit, il sera suspecté par le nœud qui a fait la demande. Et comme un nœud rationnel ne veut être suspecté, il considérera toujours une requête d'audit.

6.7.5 Traitement des fautes d'omission

Dans cette section les déviations rationnelles et les motivations que nous allons présentées sont relatives à l'automate de la figure 6.8.
Déviation rationnelle : Les déviations présentes ci-dessous sont celles qui apparaissent respectivement aux transitions 1 et 2.

1. Un nœud rationnel peut ignorer un défi venant d'un de ses surveillants en refusant de le faire suivre.
2. Un nœud rationnel peut ne pas répondre à un défi.

Motivation : Les motivations ci-dessous vont pousser un nœud rationnel à exécuter respectivement les transitions citées ci-dessus.

1. Un nœud rationnel qui ignore une réception d'un défi risque une éviction, et pour éviter cela, il le fera suivre toujours.
2. La motivation précédente tient ici aussi.

6.8 Conclusion

Ce chapitre traite le problème de la mise en place de la notion de responsabilité dans les systèmes distribués en présence de comportements rationnels. Nous avons vu qu'appliquer *PeerReview* à lui même ne résout pas complètement ce problème et est très coûteux à mettre en place. C'est ainsi que nous avons proposé *FullReview* qui utilise différentes sortes de vérifications et des techniques de la théorie des jeux pour motiver ou forcer les nœuds rationnels à suivre toutes les étapes d'un protocole responsable (protocole assurant la notion de responsabilité dans un système distribué). *FullReview* utilise également l'architecture classique vue plus haut d'un système responsable.

Toutefois, proposer une solution à un tel problème ne suffit pas, il faut aussi voir si elle est pratique. Cela nous a amené à nous poser la question suivante : Existe-t-il une meilleure manière de configurer les surveillants d'un protocole responsable utilisant l'architecture classique, de telle sorte à avoir une bonne performance en terme d'échanges de messages ?
La réponse à cette question fera l'objet du chapitre prochain qui portera sur la gestion des surveillants dans un protocole responsable adoptant l'architecture classique.

7

Gestion des surveillants

Sommaire

7.1 Gestion statique 56
7.2 Gestion dynamique 57
7.3 Vers un système sans surveillants 58
7.4 Évaluation 60
7.4.1 SplitStream 60
7.4.2 Onion routing 61
7.5 Conclusion 63

Dans les protocoles *PeerReview* et *FullReview*, nous avons vu que la surveillance est indispensable pour la mise en pratique de la notion de responsabilité dans un système distribué. Cependant, il est nécessaire de s'intéresser à la façon dont sont déployés les moniteurs (surveillants) selon la structure de l'application surveillée, car ils exécutent la majorité des tâches dans chacun des deux protocole cités ci-dessus.
Ainsi l'objectif de ce chapitre est d'étudier les différents scénarios de déploiement des moniteurs afin d'identifier ceux donnant de bonnes performances en terme d'échanges de messages.

Les travaux effectués dans le cadre de cette thèse nous ont amené à distinguer trois manières de gérer les surveillants : la gestion statique, dynamique et sans surveillants. Nous expliquons chacune de ces trois méthodes grâce au protocole *PeerReview* (les mêmes résultats s'appliquent aussi à *FullReview* de façon similaire) qui adopte l'architecture d'un système responsable, en s'intéressant uniquement aux sous protocoles coûteux (consistance, audit et transfert de preuve). La description de ces méthodes de gestion sera suivie d'une évaluation en utilisant deux applications : SplitStream et Onion routing.

7.1 Gestion statique

C'est la gestion utilisée par défaut dans *PeerReview* et *FullReview*. Dans cette gestion statique, chaque nœud du système possède un nombre fixe de surveillants. Considérons ψ comme étant le nombre de surveillants, nous modélisons le nombre de messages échangés dans chacun des sous protocoles suivants :

Consistance : Dans ce protocole, si un nœud i reçoit un authentificateur de la part d'un autre nœud j, il le fait suivre aux surveillants de j, cela correspond à ψ(nombre de moniteurs par nœud) échanges. Ensuite, le transfert des authentificateurs extraits des surveillants du nœud i aux surveillants des nœuds qui les ont généré équivaut à ψ^2 échanges de message. Ces échanges sont illustrés sur la partie gauche de la figure 7.1.
Pour un système composé de N nœuds, si X est la moyenne du nombre d'authentificateurs générés (cette valeur dépend de l'application surveillée) et P la période d'audit, alors le nombre total de messages échangés durant ce protocole est donné par la formule $\underline{NXP(\psi^2 + \psi)}$.

Audit : L'audit périodique de l'historique de chaque nœud nécessite 2ψ échanges (2 échanges entre un nœud et chacun de ses surveillants) comme illustré sur la partie au milieu de la figure 7.1. Ainsi pour une période P d'audit, un nombre N de nœuds du système, le nombre total de messages échangés dans ce protocole vaut $\underline{2NP\psi}$. Dans *FullReview*, le nombre total de messages échangés durant ce protocole est estimé à $\underline{NP(2\psi + \psi^2)}$. En effet comme décrit dans la section 6.6.1.1 (page 44), après avoir collecté les réponses d'audit, un nœud i agrège ces réponses et envoie le résultat à chacun des moniteurs de ses moniteurs. Ce processus coûte ψ^2 (ψ est le nombre de moniteurs de i et chacun de ses moniteurs a aussi ψ surveillants) échanges de messages.

Transfert de preuve : Lorsqu'un nœud i veut connaitre le statut d'un autre nœud j, il contacte les surveillants du nœud j afin de s'informer. Cela équivaut à un échange de 2ψ messages. Ensuite avec 2 échanges de messages, le nœud i met au défi le nœud j. Ces échanges sont illustrés sur la partie droite de la figure 7.1.
Finalement le nombre total de messages échangés durant cette étape de *PeerReview* est égal à $\underline{NlP(2\psi + 2)}$ où l est le nombre de nœuds dont le statut est demandé par le nœud i.
Ce nombre d'échanges de messages diminue durant la même étape dans *FullReview*. Cela est dû au fait que le résultat de l'audit d'un nœud i est envoyé aux surveillants des surveillants de i. Donc un nœud quelconque n'a pas besoin de contacter les surveillants d'un autre pour s'informer du statut de ce dernier. Par conséquent le terme 2ψ disparaît dans la formule précédente. Nous obtenons $\underline{2NlP}$ comme étant le nombre de messages échangés durant cette étape de *FullReview*.

Le tableau 7.1 résume les échanges de messages ci-dessus. Pour résumer, la gestion dynamique des surveillants est modélisée en terme d'échanges de messages par les

formules suivantes :

- $\underline{NXP(\psi^2+\psi)+2NP\psi+NlP(2\psi+2)}$ dans *PeerReview*
- $\underline{NXP(\psi^2+\psi)+NP(2\psi+\psi^2)+2NlP}$ dans *FullReview*.

Ces formules s'obtiennent en faisant la somme des échanges de messages dans chacun des sous protocoles concernés dans chaque cas.

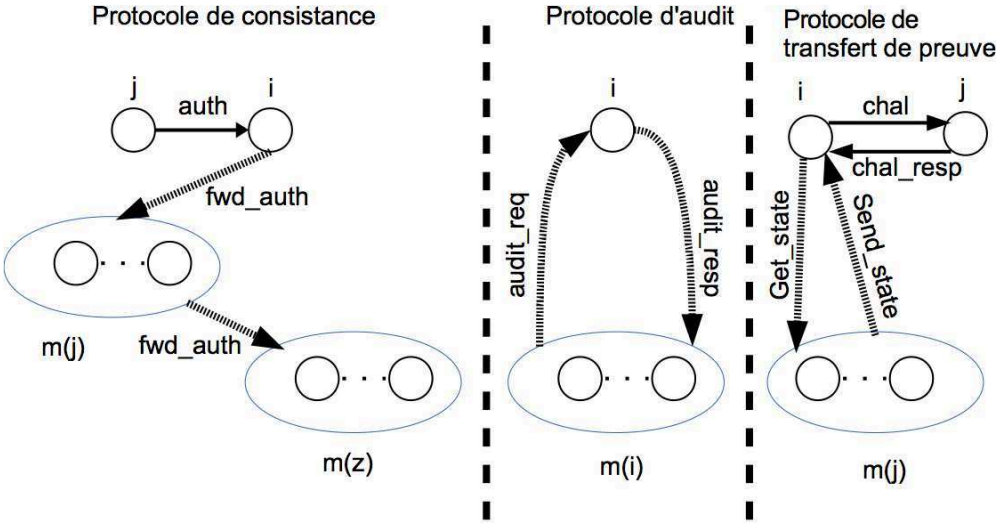


FIGURE 7.1 – Gestion statique : Scénario des échanges de messages.

	Sous protocole	Consistance	Audit	TP
Gestion statique	*PeerReview*	$NXP(\psi^2+\psi)$	$2NP\psi$	$NlP(2\psi+2)$
	FullReview	$NXP(\psi^2+\psi)$	$NP(2\psi+\psi^2)$	$2NlP$
Gestion dynamique	*PeerReview*	$XP\sum_{i=1}^{N} v_i$	$2P\sum_{i=1}^{N} v_i$	$2NlP$
	FullReview	$XP\sum_{i=1}^{N} v_i$	$\sum_{i=1}^{N} P(2v_i+v_i^2)$	$2NlP$
Gestion sans surveillants	*PeerReview*	$NXPR$	PN^2R	$2NlP$
	FullReview	$NXPR$	PN^2R	$2NlP$

TABLE 7.1 – Le nombre d'échanges de messages dans *PeerReview* comparé à celui dans *FullReview* selon les différentes méthodes de gestion. TP = Transfert de Preuve

7.2 Gestion dynamique

Dans la gestion dynamique, chaque nœud du système est associé à une vue (une sorte de liste) qui contient un certain nombre de nœuds. Périodiquement, chaque vue est mise à jour grâce à un protocole de *PeerSampling* [46]. Les nœuds dans la vue

jouent le rôle de surveillants. Pour faciliter les tâches exécutées par les surveillants, les authentificateurs et les défis générés sont diffusés dans chaque vue. En partant de cette description, nous allons modéliser les échanges dans chacun de ses sous-protocoles suivants :

Consistance : Périodiquement, lorsqu'un nœud i reçoit un authentificateur, il le diffuse dans sa vue comme illustré sur la partie gauche de la figure 7.2. Cette diffusion coûte v_i échanges de messages où v_i est la taille de la vue associée au nœud i.
En considérant un système constitué de N nœuds et X comme la moyenne du nombre d'authentificateurs générés par ce système, le nombre de messages échangés durant cette étape est estimé à $\underline{XP\sum_{i=1}^{N} v_i}$ où P est la période d'audit.

Audit : Dans la gestion dynamique, chaque nœud est périodiquement audité par les nœuds de sa vue. Il y a ainsi 2 échanges entre un nœud et chacun des éléments de sa vue comme indique la partie au milieu de la figure 7.2. Cela nécessite $2v_i$ échanges de messages (v_i étant la taille de la vue associée au nœud i).
Ainsi le nombre total de messages échangés dans cette phase par un système composé de N nœuds est estimé à $\underline{2P\sum_{i=1}^{N} v_i}$ dans *PeerReview* et $\underline{P(\sum_{i=1}^{N}(2v_i + v_i^2))}$ dans *FullReview* dû au transfert du résultat de l'audit aux surveillants des surveillants.

Transfert de preuve : Dans cette phase, avec la gestion dynamique, un nœud n'a pas besoin de s'informer du statut d'un autre grâce à la dissémination des authentificateurs dans les vues comme illustré sur la partie gauche de la figure 7.2. Cependant, il peut mettre au défi un autre nœud en ayant son statut et cela coûte deux échanges. Ainsi le nombre de messages échangés durant cette étape s'obtient par la formule $\underline{2NlP}$ (N étant le nombre de nœuds du système, l le nombre de nœuds dont le statut est demandé par un nœud i et P la période d'audit) aussi bien pour *PeerReview* que pour *FullReview*.

Finalement dans la gestion dynamique comme indique le tableau 7.1, le nombre total de messages échangés s'obtient avec les formules suivantes :

- $\underline{XP\sum_{i=1}^{N} v_i + 2P\sum_{i=1}^{N} v_i + 2NlP}$ pour le protocole *PeerReview*
- $\underline{XP\sum_{i=1}^{N} v_i + P(\sum_{i=1}^{N}(2v_i + v_i^2)) + 2NlP}$ pour le protocole *FullReview*.

7.3 Vers un système sans surveillants

L'idée de cette méthode est de pouvoir concevoir un système renforçant la notion de responsabilité sans l'assignation de surveillants aux nœuds. Pour atteindre cet objectif, nous utilisons une structure en multiples anneaux comme dans RAC [20] ou *Fireflies* [47] afin de diffuser les authentificateurs et les défis de manière fiable. Avec cette structure, chaque nœud du système a une position unique sur chaque anneau, cela implique que chaque nœud a un prédécesseur et un successeur sur chaque anneau. La diffusion est faite dans la liste des successeurs. Ce qui permet à chaque nœud du système

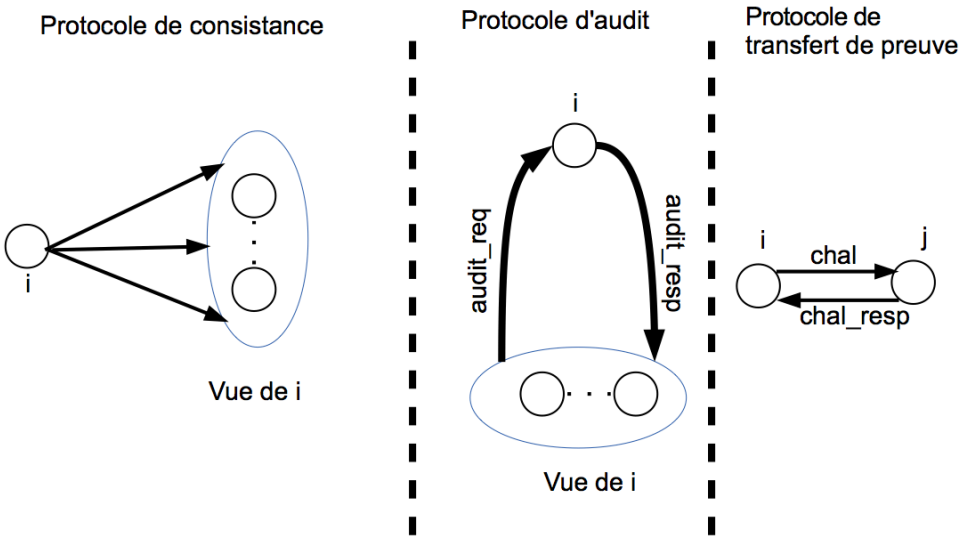


FIGURE 7.2 – Gestion dynamique : Scénario des échanges de messages.

s'il le désire, d'auditer un autre nœud (par exemple, les partenaires avec lesquels il a interagit). Nous modélisons le nombre d'échanges de messages dans les sous protocoles suivants :

Consistance : La diffusion d'un authentificateur par un nœud dans l'ensemble de ses successeurs, comme indique la partie à gauche de la figure 7.3 nécessite R échanges où R représente le nombre d'anneaux utilisés. Ainsi dans un système composé de N nœuds surveillés par *PeerReview* ou *FullReview*, et ayant généré X authentificateurs durant une période P, le nombre total de messages échangés équivaut à $\underline{NXPR}$.

Audit : Dans cette étape, périodiquement chaque nœud diffuse son historique dans le système grâce aux anneaux comme indiqué sur la partie au milieu de la figure 7.3. Avec cette structure (multiples anneaux) une diffusion dans le système coûte NR échanges de messages, et si périodiquement tous les nœuds diffusent dans le système ce coût devient $\underline{PN^2R}$, qu'il soit surveillé par *PeerReview* ou par *FullReview*.

Transfert de preuve : Comme les authentificateurs et les défis sont diffusés dans le système grâce à la structure en anneaux, un nœud peut mettre facilement mettre au défi un autre nœud si ce dernier est suspecté. Cette étape est illustrée sur la partie à droite de la figure 7.3. Les échanges de messages faits durant cette phase sont estimés à $\underline{2NlP}$ selon que le système soit surveillé par *PeerReview* ou *FullReview*.

En guise de conclusion pour cette section, le nombre de messages échangés dans la gestion sans moniteurs est modélisé par la formule $\underline{NXPR + PN^2R + 2NlP}$ comme indique le tableau 7.1 qu'il s'agisse de la surveillance du système avec *PeerReview* ou

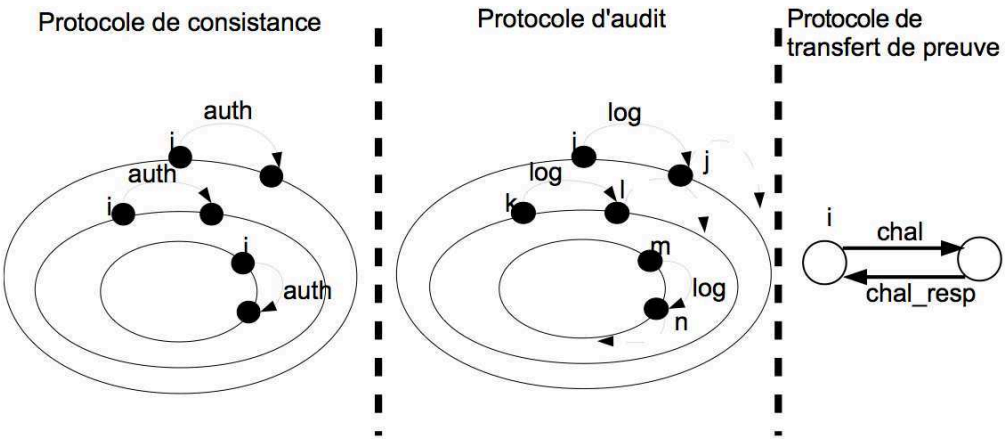


FIGURE 7.3 – Gestion sans surveillants : Scénario des échanges de messages.

avec *FullReview*.
Dans la section qui va suivre, nous évaluons chacune de ces méthodes en utilisant deux applications : *SplitStream* et *Onion routing*

7.4 Évaluation

Pour chacune des applications citées ci-dessus, nous évaluons les échanges de messages dans *PeerReview* lorsqu'elles sont surveillées par ce dernier, en utilisant chacune des trois méthodes de gestion.
Les expériences sont faites par simulation. Les valeurs de certains paramètres comme X (la moyenne des authentificateurs générés), l (le nombre de nœuds dont un nœud du système est intéressé par leur statut) dans la modélisation, dépendent de la structure de l'application et du nombre de nœuds total dans le système. Les valeurs de X sont obtenues grâce au simulateur de *PeerReview* développé par ses auteurs. En effet nous lançons ce simulateur durant 100 secondes, en variant le nombre de nœuds et en faisant surveiller l'application par *PeerReview*.
L'objectif de cette évaluation est de trouver les conditions dans lesquelles une méthode de gestion est mieux adaptée qu'une autre.

7.4.1 SplitStream

Nous mesurons le nombre de messages échangés dans *PeerReview* lorsqu'il surveille l'application SplitStream en fonction du nombre de nœuds du système, dans chaque méthode de gestion utilisant les formules obtenues ci-dessus. Dans chacune des trois méthodes un nœud s'intéresse aux statuts de 50% des nœuds du système. Le nombre de surveillants dans la gestion statique et le nombre d'anneaux dans la gestion sans surveillants valent tous 5.

La taille d'une vue dans la gestion dynamique est au moins $v_i = ln(N)$, celle-ci est la taille minimale requise pour une dissémination fiable [48]. La période d'audit est de 10 secondes. Selon les valeurs de certains paramètres, une méthode est mieux adaptée qu'une autre. Ces paramètres avec leurs valeurs figurent dans le tableau 7.2. Le résultat de la simulation du cas 1 (voir tableau 7.2) comme indique la figure 7.4a, montre que la gestion dynamique est pratiquement celle qui convient mieux. Cela est dû au fait que les vues ont la taille minimale ($ln(N)$), ce qui réduit le nombre d'échanges de messages. Cependant, en augmentant la taille des vues de telle sorte à avoir une dissémination encore plus fiable (cas 2 du tableau 7.2), la gestion sans surveillants est la plus adaptée comme le montre la figure 7.4b.
La gestion statique domine les autres lorsqu'on augmente à la fois la taille des vues et le nombre d'anneaux (cas 3 du tableau 7.2), ce résultat est illustré sur la figure 7.5.

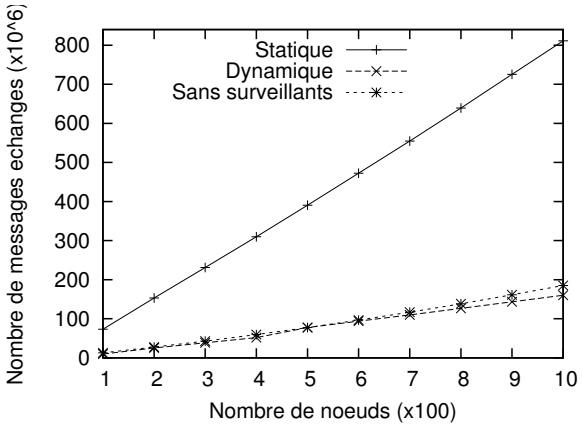
paramètres	l	v_i	ψ	R
cas 1	$50\%N$	$ln(N)$	5	5
cas 2	$50\%N$	$ln(N)+5$	5	5
cas 3	$50\%N$	$ln(N)+20$	3	9

TABLE 7.2 – SplitStream : Valeurs des paramètres

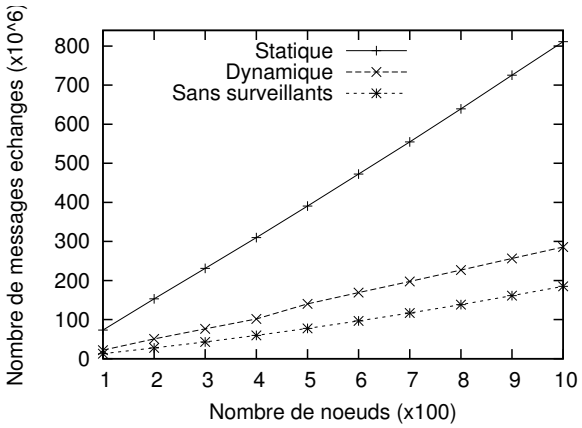
7.4.2 Onion routing

Onion routing est la seconde application que nous avons utilisée pour tester les trois méthodes de gestion.
Nous évaluons le nombre de messages échangés dans *PeerReview* lorsqu'il surveille cette application. Dans cette simulation, un nouveau paramètre apparait qui est le nombre de relais. Ce paramètre est indispensable dans le fonctionnement de l'application *Onion routing*. Nous le fixons à 40 [1]
Comme précédemment, les résultats montrent que selon les valeurs de certains paramètres (voir tableau 7.3), une méthode convient mieux qu'une autre. Ainsi les résultats obtenus avec le cas 1 (tableau 7.3) et illustrés sur la figure 7.6a, montrent que la gestion dynamique est la plus appropriée. Sur la figure 7.6b (illustration du cas 2), la gestion sans surveillants est plus adaptée que les autres. Par contre en augmentant le nombre d'anneaux et la taille des vues (cas 3), les résultats présentés sur la figure 7.7 montrent que la gestion statique est meilleure que les autres.

1. Étant donné que dans nos simulations nous faisons varier le nombre de nœuds de 100 à 1000, nous avons choisi ce nombre pour éviter un nombre de relais trop faible.



(a) Cas 1



(b) Cas 2

FIGURE 7.4 – Nombre de messages échangés dans *PeerReview* en fonction du nombre de nœuds lorsqu'il surveille SplitStream. Le cas 1 correspond à $l = 50\%N$, $v_i = ln(N)$, $\psi = 5$ et $R = 5$. Le cas 2 correspond à $l = 50\%N$, $v_i = ln(N) + 5$, $\psi = 5$ et $R = 5$.

paramètres	l	v_i	ψ	R
cas 1	$50\%N$	$ln(N)$	5	5
cas 2	$50\%N$	$ln(N) + 15$	5	5
cas 3	$50\%N$	$ln(N) + 25$	3	9

TABLE 7.3 – Onion routing : Valeurs des paramètres

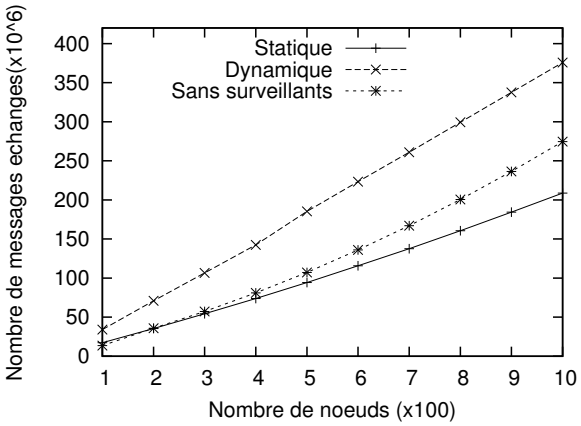
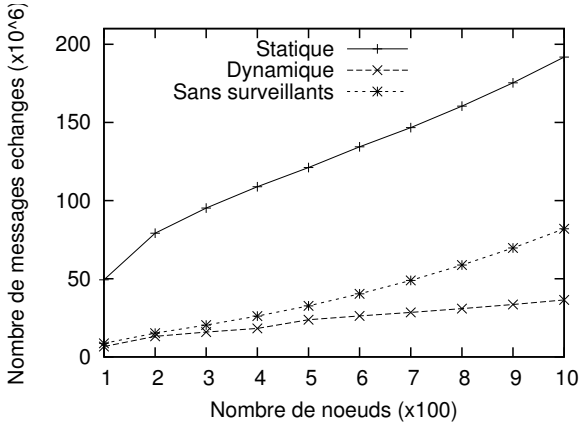


FIGURE 7.5 – Cas 3 : Nombre de messages échangés dans *PeerReview* en fonction du nombre de nœuds lorsqu'il surveille SplitStream. Ici $l = 50\%N$, $v_i = ln(N) + 20$, $\psi = 3$ et $R = 9$.

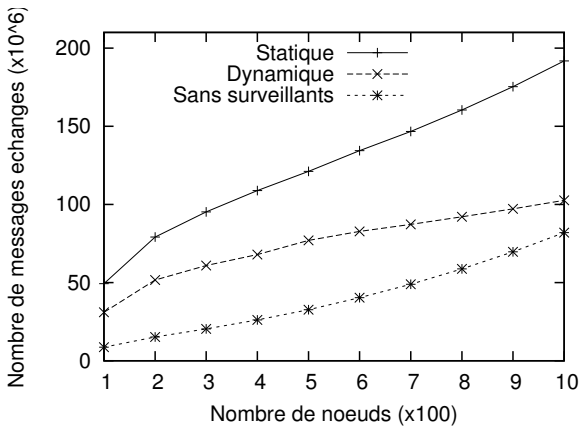
7.5 Conclusion

Ce chapitre propose des modèles pour le nombre de messages échangés durant les étapes exécutées par les surveillants dans un système responsable adoptant l'architecture classique, tout en décrivant les différentes manières de déploiement des surveillants.
Il existe trois méthodes pour gérer les surveillants : gestion statique, dynamique et sans surveillants. Nous avons montré que selon les valeurs de certains paramètres dans nos différents modèles, une méthode peut être mieux adaptée qu'une autre. Pour illustrer ces différents cas de figure, nous avons testé les modèles obtenus en utilisant comme protocole surveillant : *PeerReview* et comme applications surveillées : *SplitStream* et *Onion routing*.
Ce chapitre répond ainsi à la question posée tout au début, qui était de savoir s'il existe une meilleure manière de configurer les surveillants, afin d'obtenir de bonne performance en terme d'échanges de messages dans un système responsable.

Cependant, il reste une autre question qui est pour le moment sans réponse. Il s'agit de l'évaluation de notre protocole *FullReview* et sa comparaison avec *PeerReview*. Nous allons consacrer le chapitre qui va suivre à cette question.



(a) Cas 1



(b) Cas 2

FIGURE 7.6 – Nombre de messages échangés dans *PeerReview* en fonction du nombre de nœuds lorsqu'il surveille Onion routing. Le cas 1 correspond à $l = 50\%N$, $v_i = ln(N)$, $\psi = 5$ et $R = 5$. Le cas 2 correspond à $l = 50\%N$, $v_i = ln(N) + 15$, $\psi = 5$ et $R = 5$

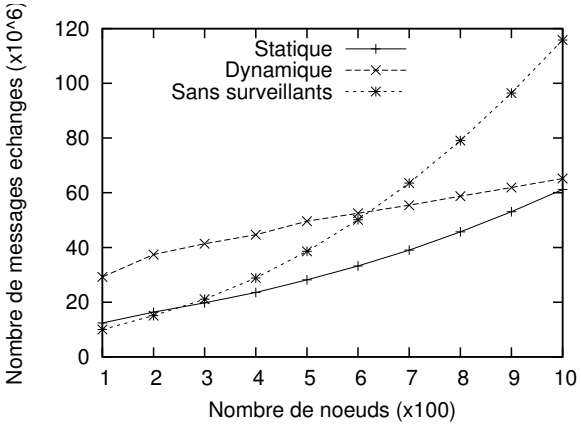


FIGURE 7.7 – Case 3 : Nombre de messages échangés dans *PeerReview* en fonction du nombre de nœuds lorsqu'il surveille SplitStream. Ici $l = 50\%N$, $v_i = ln(N) + 25$, $\psi = 3$ et $R = 9$.

8

Évaluation de performance

Sommaire

8.1 Applications 68

8.1.1 *SplitStream* 68

8.1.2 *Onion routing* 68

8.2 Configurations expérimentales 69

8.3 Performance en présence des nœuds rationnels 69

8.4 Performance dans le cas sans fautes 73

8.5 Passage à l'échelle de *FullReview* 78

8.6 Conclusion 78

Après la présentation de notre protocole *FullReview* et les différentes méthodes de gestion des surveillants, nous nous intéressons à la performance de *FullReview* par rapport à *PeerReview*. Pour cela nous utilisons deux applications distribuées : *SplitStream* et *Onion routing*. L'objectif est de parvenir à répondre aux questions suivantes :

- *FullReview* détecte-t-il des fautes en présence des comportements rationnels ?
- Quel est le surcoût lié à l'application de *FullReview* comparé à *PeerReview* ?
- *FullReview* passe-t-il à l'échelle ?

Nous commençons ce chapitre par introduire les deux applications et les configurations expérimentales dans la section 8.1 et 8.2, respectivement. Nous présentons ensuite la performance de *FullReview* en présence des nœuds rationnels (section 8.3) et dans le cas sans fautes (section 8.4). Finalement, nous étudions le passage à l'échelle du protocole *FullReview* dans la section 8.5.

8.1 Applications

Dans cette section nous allons décrire les deux applications qui seront utilisées dans l'évaluation. La première est une application de diffusion efficace et la seconde est un protocole de communication anonyme.

8.1.1 *SplitStream*

SplitStream [44] est un protocole qui organise les nœuds sous forme d'une structure en arbre où chaque nœud reçoit des messages de diffusion de son nœud parent et les transfère à ses nœuds enfants. La spécificité de *SplitStream* est d'équilibrer la charge du transfert entre les nœuds. Il atteint cet objectif en divisant le flux de diffusion en morceaux et utilise plusieurs arbres de diffusion pour distribuer chaque morceau. Pour nos expériences, le nœud source diffuse un flux de vidéo à 300kb/s, qui est la vitesse communément utilisée pour les applications de diffusion de vidéos. Chaque paquet émis par la source est envoyé à travers différents arbres de *multicast* où chaque nœud a deux nœuds fils.

Dans SplitStream, les nœuds rationnels dévient du protocole en refusant de transférer les paquets à leurs enfants. Comme résultat, ils peuvent obtenir le flux de vidéo tout en conservant la bande passante. Cependant, en présence des nœuds rationnels, les nœuds corrects peuvent subir la perte des paquets et par conséquent, ils reçoivent une version dégradée du flux de vidéo.

8.1.2 *Onion routing*

Onion routing [45] est un protocole conçu pour les communications anonymes. C'est le protocole utilisé dans le projet TOR [49], qui est couramment utilisé par un millier d'utilisateurs. Dans ce protocole, quand un nœud S veut envoyer un message à un nœud D, il choisit R autres nœuds, appelés relais et ayant pour rôle de transférer le message à la réception jusqu'à sa destination. Le nœud S chiffre successivement le message en utilisant la clé publique de chacun des relais, qui constitue l'*oignon*. Cet oignon est ensuite envoyé au premier relais. Chaque relais déchiffre une couche de l'oignon (c'est-à-dire enlève une couche de chiffrement) et le transfère au relais suivant jusqu'à sa destination finale. Pour les expériences avec *Onion routing*, chaque nœud envoie périodiquement un paquet au dernier nœud relais (nœud dont la clé publique a servi à construire la dernière couche de l'oignon). Le nombre de relais est un paramètre du protocole et dans toutes nos expériences, les messages ont une taille fixée à 10kB ; les messages de petite taille sont rembourrés avec des octets supplémentaires afin d'avoir les 10kB. Fixer la taille d'un message est habituellement utilisé dans le routage par oignon pour éviter qu'un attaquant suive un oignon dans le système en comparant la taille des messages transférés.

Dans ce protocole, un nœud rationnel peut choisir de ne pas transférer un oignon qui ne lui est pas destiné. Par conséquent, le destinataire ne recevra jamais le message

original. L'objectif de la conception d'une version rationnel-résistante de *Onion routing* est de s'assurer que les nœuds transféreront les oignons qu'ils reçoivent tout en fournissant des garanties d'anonymat.

8.2 Configurations expérimentales

Nous avons mesuré la performance de SplitStream et Onion routing dans deux configurations : (i) avec *PeerReview* et (ii) avec *FullReview*. Nos expériences ont été effectuées dans deux différentes configurations. Tout d'abord, nous avons effectué des expériences dans des conditions réelles en utilisant la grappe de calcul grid5000. Dans cette grappe nous avons utilisé 50 machines physiques avec 4 cœurs, 2.6GHz de fréquence, 4GB de RAM chacune et interconnectées avec un switch de 1Gb/s. Ces expériences ont été exécutées en déployant un nœud sur une machine physique et les courbes correspondant sont annotées avec [G5K] dans leurs labels. Pour compléter nos expériences, nous avons effectué des simulations en utilisant le simulateur de *PeerReview* développé par les auteurs [1] de *PeerReview*. Nous avons fait des simulations avec 1000 nœuds dans le but d'évaluer le passage à l'échelle de *FullReview*.
Les résultats de ces expériences sont annotées avec [SIM] dans leurs labels.

8.3 Performance en présence des nœuds rationnels

Dans cette section nous montrons que *FullReview* tolère les nœuds rationnels. Pour ce faire, nous effectuons trois expériences. Dans la première expérience, les nœuds rationnels suivent le modèle présenté dans le chapitre 6. En effet, ils dévient du protocole seulement s'ils y voient un intérêt ou s'il n'y a pas de risque d'être détectés.
Dans la seconde expérience, les rationnels se comportent comme dans la première expérience, seulement nous nous intéressons à l'application Onion routing, en faisant varier le nombre de relais.
Et dans la troisième expérience, nous considérons que les nœuds rationnels dévient s'ils y voient un intérêt sans considérer le risque d'exclusion. Cette dernière expérience montre que s'ils décident dévier, les nœuds rationnels sont rapidement détectés par leurs surveillants et sont exclus du système.
Pour toutes ces expériences les deux applications décrites plus haut sont surveillées chacune par *PeerReview* et *FullReview*. Le nombre de surveillants par nœud est fixé à 2 et la période d'audit est de 10s.

Les résultats de la première expérience sont présentés sur la figure 8.1a. Cette figure montre le pourcentage de messages reçus en fonction du pourcentage du nombre de nœuds rationnels. *SplitStream* et *FullReview* sont déployés sur 50 nœuds de la plateforme G5K (grid5000). Sur le même nombre de nœuds de la même plateforme nous déployons également *Onion routing* et *FullReview*, en fixant le nombre de relais

1. PeerReview code : http ://peerreview.mpi-sws.mpg.de/.

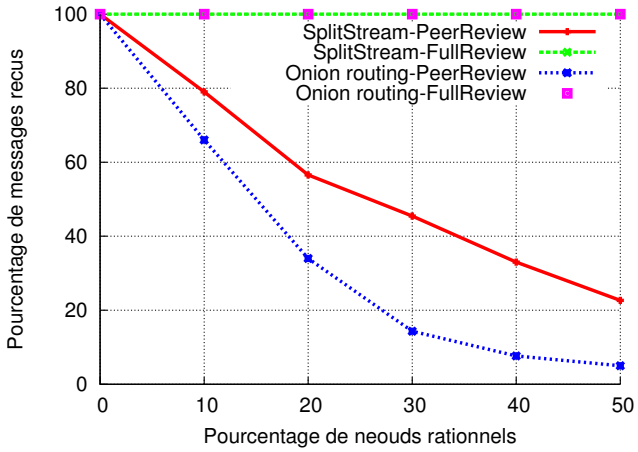
à 5, qui a été choisi aléatoirement. Les mêmes scénarios de déploiement ont été avec *PeerReview* avec chacune des deux applications.

Nous observons dans un premier temps sur cette figure qu'en utilisant *PeerReview*, SplitStream et Onion routing ne tolèrent pas les nœuds rationnels. En effet, en présence de seulement 10% de nœuds rationnels, seulement 79% et 66% de messages sont reçus dans les applications SplitStream et Onion routing, respectivement. Cela représente une perte de 21% et 34% de messages, respectivement, qui n'est pas acceptable.Ce pourcentage diminue quand la proportion de nœuds rationnels augmente et atteignant 23% dans SplitStream et 5% dans Onion routing, en présence de 50% de nœuds rationnels. Ainsi, en utilisant *FullReview*, nous constatons que tous les messages sont reçus dans les deux applications car les nœuds rationnels n'ont aucun intérêt à dévier du protocole.

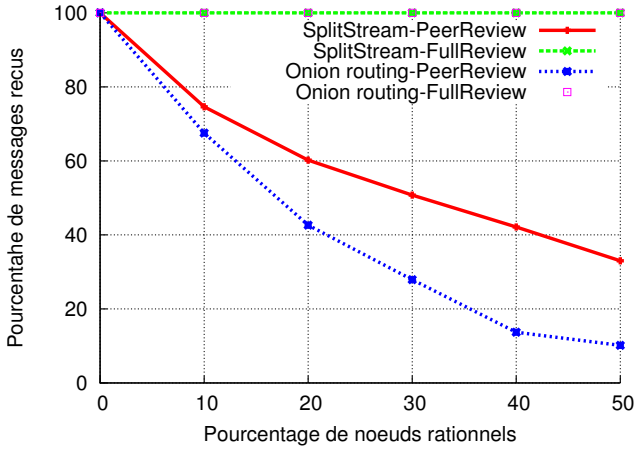
Pour consolider nos résultats des expériences faites sur une plateforme réelle, nous avons fait parallèlement les mêmes expériences par simulation. Ces résultats sont présentés sur la figure 8.1b. Nous arrivons à la même conclusion que précédemment.

Dans la seconde expérience, nous évaluons l'impact du nombre de relais dans *Onion routing* sur le pourcentage de messages reçus en présence des nœuds rationnels. Les résultats obtenus grâce à la plateforme G5K et par simulation, présentés sur la figure 8.2, montrent qu'augmenter le nombre de relais, conduit à de mauvais résultats pour *PeerReview*. En effet la probabilité de choisir un nœud rationnel comme relais dans Onion routing devient plus grande. Par exemple avec 10% de nœuds rationnels 67% de messages sont reçus lorsque nous utilisons 5 relais, ce pourcentage diminue jusqu'à 9% avec 40 relais (en simulations).
En outre, nous constatons que le pourcentage de messages reçus dans l'expérience sur G5K est faible. Par exemple, avec 10% de nœuds rationnels et 40 relais, seulement 1 message est reçu dans l'expérience toute entière. Toutefois, il est à noter que *Onion routing* surveillé par *FullReview* n'enregistre pas de perte de messages quelque soit le nombre de relais. Cela est dû au fait que les nœuds rationnels n'ont aucun intérêt à dévier du système.

Les résultats de la troisième expérience sont présentés dans la figure 8.3. Dans cette expérience, nous mesurons le pourcentage de messages reçus dans SplitStream avec *PeerReview* et *FullReview* où les nœuds rationnels commencent à dévier du protocole après 20s. Cette expérience a été lancée sur 50 nœuds en utilisant des simulations. Comme expliqué ci-dessus, dans cette expérience, les nœuds rationnels se comportent rationnellement sans mesurer le risque d'être détectés. En utilisant *PeerReview*, nous observons que les nœuds rationnels impactent le système sans être détectés, tandis qu'avec *FullReview* ils impactent le système seulement durant un laps de temps. Ce temps correspond à la fréquence d'audit, durant laquelle les nœuds rationnels sont détectés et évictés du système. Finalement, tous les messages sont reçus durant le reste de l'expérience. Remarquons que choisir une faible période d'audit, permet de détecter les nœuds rationnels rapidement, mais induit un surcoût supplémentaire, comme indiqué dans la section suivante.

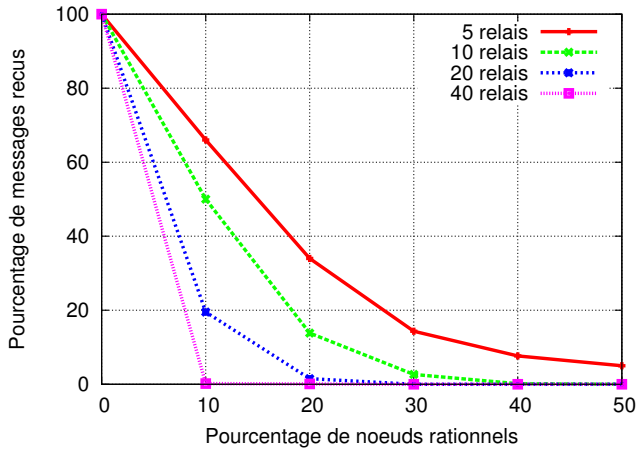


(a) [G5K]

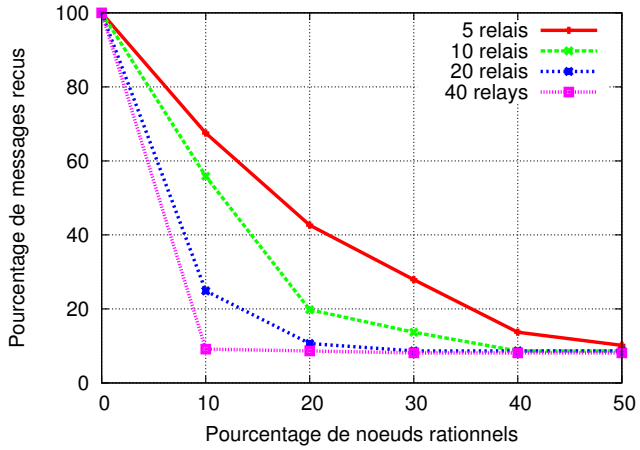


(b) [SIM]

FIGURE 8.1 – Pourcentage de messages reçus dans SplitStream et Onion routing en fonction du pourcentage de nœuds rationnels.



(a) [G5K]



(b) [SIM]

FIGURE 8.2 – Impact du nombre de relais sur le pourcentage de messages reçus dans Onion routing-*PeerReview*.

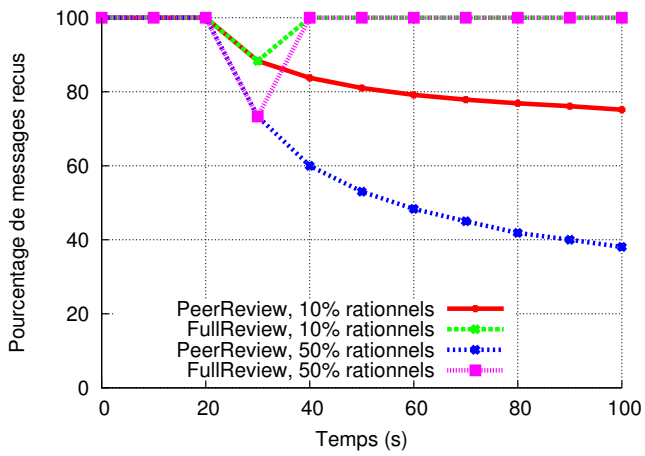


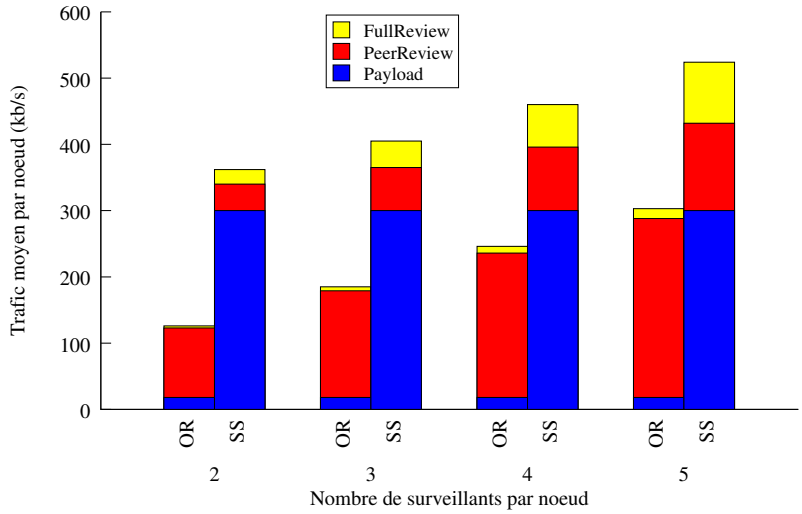
FIGURE 8.3 – [SIM] SplitStream Pourcentage de messages reçus durant une expérience dans laquelle entre 10% et 50% de nœuds commencent à agir rationnellement après 20s.

8.4 Performance dans le cas sans fautes

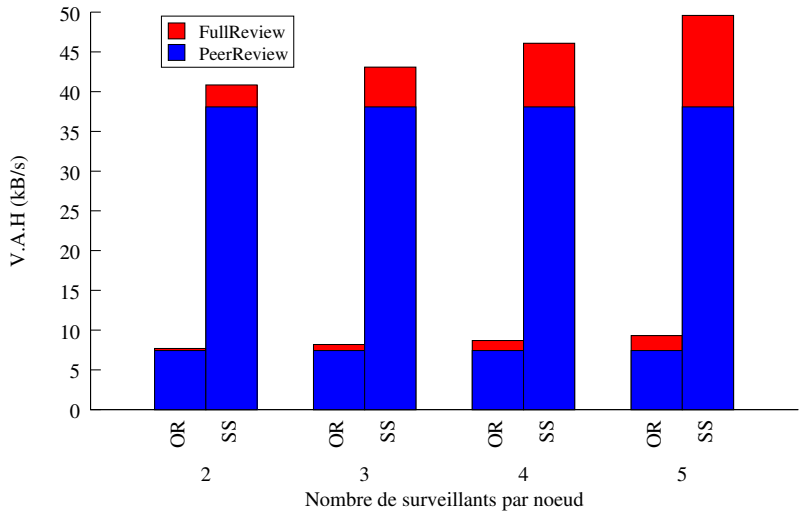
Dans cette section nous évaluons la performance et le surcoût de *FullReview*, comparé à *PeerReview*, dans le cas sans fautes. Pour ce faire, nous effectuons trois expériences. Nous lançons chacune de ces expériences en utilisant à la fois les simulations et la plateforme G5K. Nous remarquons que les résultats obtenus par simulations sont cohérents avec ceux obtenus avec G5K.

Dans les deux premières expériences, nous mesurons le trafic réseau et la vitesse à laquelle les historiques augmentent en fonction du nombre de surveillants, dans *PeerReview* et *FullReview* respectivement. Dans le cas de *Onion routing*, le chemin d'un oignon est composé de 5 relais. La figure 8.5 présente à la fois les résultats de SplitStream et *Onion routing* sur G5K, tandis que ceux de la figure 8.4 sont issus des simulations. Chaque valeur a été obtenue en exécutant le système durant 5 minutes avec 50 nœuds. Nous détaillons seulement les résultats provenant de la plateforme G5K. Cependant, nous avons des observations similaires en simulations.

Dans la figure 8.5a, chaque barre représente le trafic dû à la charge utile (*payload*) de l'application. Au dessus de cette charge utile, se trouve le trafic dû au protocole *PeerReview*, et en dessus de ce dernier apparait le surcoût induit par *FullReview* par rapport à *PeerReview*. Dans cette figure, nous observons que le trafic moyen par nœud augmente par rapport au nombre de surveillants à la fois dans *PeerReview* et *FullReview* dans les deux applications. Cela est dû aux nombreux messages qui sont échangés entre les nœuds et leurs surveillants. En plus, nous constatons que le surcoût lié à la notion de responsabilité dans l'application SplitStream est de 14% dans *PeerReview* avec deux surveillants et de 7% de coût supplémentaire dans *FullReview*. Ce surcoût augmente jusqu'à 45% pour *PeerReview* et 31% de coût supplémentaire pour *FullReview* lorsque

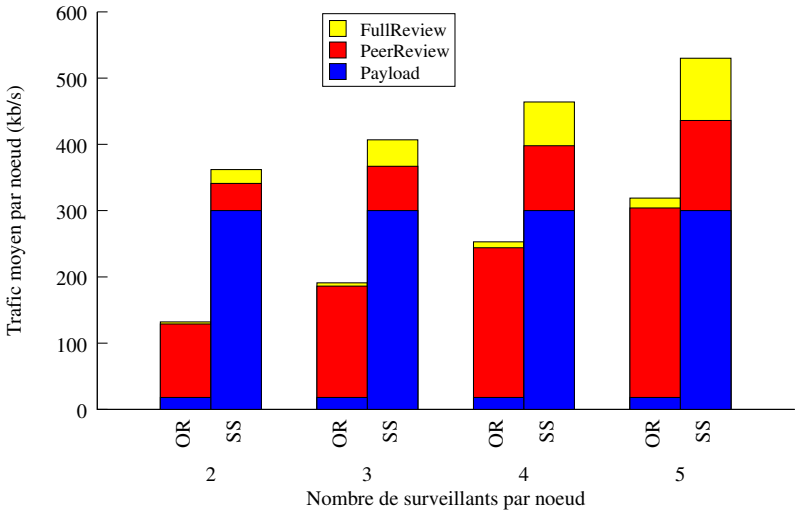


(a) trafic réseau.

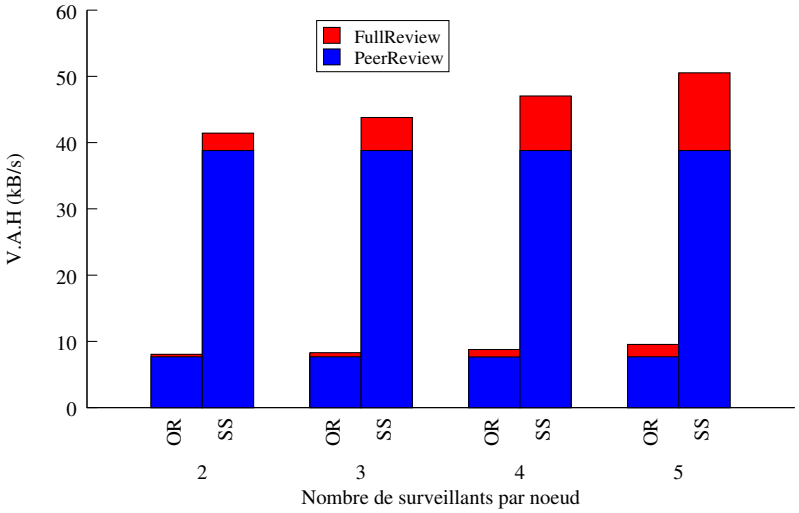


(b) Vitesse d'augmentation de l'historique.

FIGURE 8.4 – [SIM] Moyenne du trafic réseau et de la vitesse d'augmentation de l'historique (V.A.H) par nœud de *SplitStream*(SS) et *Onion routing*(OR) en fonction du nombre de surveillants.



(a) Trafic réseau.



(b) Vitesse d'augmentation de l'historique.

FIGURE 8.5 – [G5K] Moyenne du trafic réseau et de la vitesse d'augmentation de l'historique (V.A.H) par nœud de SplitStream (SS) et Onion routing (OR) en fonction du nombre de surveillants.

le nombre de surveillants vaut 5. Ces coûts sont beaucoup plus élevés si comparés à la charge utile de l'application Onion routing. Par exemple, renforcer la responsabilité dans Onion routing en utilisant *PeerReview* génère un trafic de 129kb/s par nœud tandis que l'application elle même génère une charge utile de 18kb/s seulement par nœud. Toutefois, en se plaçant dans ce contexte, ce résultat n'est pas mauvais, comme renforcer la responsabilité dans les protocoles de communication anonyme est une tâche relevant beaucoup de défis et pour laquelle les solutions existantes nécessitent une utilisation intensive des primitives de diffusion (comme dans RAC [20], Dissent [50]). En supposant de plus que les nœuds sont connectés avec des liens d'un Gigabit (dans le cas d'un LAN) ou même avec des liens de quelques Megabit (dans le cas d'un WAN), 129kb/s paraît un surcoût raisonnable.
La bonne nouvelle est que si un développeur accepte de payer le coût de la responsabilité en utilisant *PeerReview* dans un système avec une petite charge utile, utiliser un système responsable et résistant aux comportements rationnels, c'est à dire *FullReview*, lui coûterait un surplus de 3kb/s (c'est à dire 2% de trafic supplémentaire) avec deux surveillants et un surplus de 15kb/s (c'est à dire 5% de trafic supplémentaire) avec cinq surveillants. Notons que, globalement, renforcer la responsabilité en utilisant *PeerReview* est plus cher dans l'application Onion routing que dans l'application SplitStream car dans la première application les oignons en entier sont stockés dans l'historique tandis que dans la dernière application, au lieu de stocker les morceaux de vidéos reçus par les nœuds dans l'historique, nous stockons seulement leurs identifiants. En effet, stocker les oignons était le seul moyen que nous avons trouvé pour permettre aux surveillants de vérifier qu'un nœud a correctement déchiffré et transféré un oignon qu'il a reçu.

Dans la figure 8.5b, chaque barre représente la vitesse moyenne d'augmentation de l'historique des nœuds. De façon similaire à la figure précédente, le coût de *FullReview* apparaît comme un supplément au coût de *PeerReview*. Notons que les historiques ne croissent pas indéfiniment. En effet, comme dans *PeerReview*, les historiques sont tronqués après une certaine durée et les audits sont effectués seulement pour les nouvelles portions de l'historique. Bien évidemment, plus la période d'enregistrement choisie par le concepteur est longue, plus la probabilité d'empêcher des fautes est grande.

Les résultats apparaissant sur la figure montrent que la vitesse d'augmentation de l'historique de l'application *SplitStream* est plus grande que celle de l'application *Onion routing*, cela est dû au fait que l'application *SplitStream* génère plus de messages pour envoyer un flux de vidéo que Onion routing, et ainsi plusieurs interactions sont rajoutées à l'historique. En plus, nous observons que plus le nombre de surveillants par nœud augmente plus la vitesse d'augmentation de l'historique est grande. Dans l'application *Onion routing*, le surcoût en terme de la vitesse d'augmentation de l'historique est égal à 4.9% lorsqu'on utilise *FullReview* avec deux surveillants. Il augmente jusqu'à 24% lorsqu'on utilise cinq surveillants. Dans l'application *SplitStream*, ce surcoût est plus élevé et va de 6.8% à 30% lorsqu'on utilise respectivement deux et cinq surveillants. Encore, nous considérons ce surcoût comme raisonnable. En effet, dans le pire des cas de nos expériences (utiliser 5 surveillants dans l'application *SplitStream*), pour 24

heures d'enregistrement, les nœuds ont besoin de consacrer 4.4GB d'espace de stockage pour renforcer la notion de responsabilité en présence des nœuds rationnels, ce qui est raisonnable.

Dans la troisième expérience, nous mesurons l'impact de la période d'audit sur le surcoût de *FullReview* comparé à *PeerReview*. Nous faisons varier la période d'audit entre 1s et 30s. Le nombre de nœuds est de 50, avec 2 surveillants par nœud et 5 relais pour l'application Onion routing. Chaque expérience dure 5 minutes. Les résultats présentés dans la table 8.1 montrent que même avec une fréquence d'audit plus élevée (i.e chaque seconde), *FullReview* génère seulement 6.7% de plus de trafic et les historiques sont de 8.2% plus grands que dans *PeerReview* dans le pire des cas.

	Période d'audit	1s	5s	10s	30s
SS	Taille de l'historique	+7.4%	+6.8%	+6.7%	+6.4%
	Trafic réseau	+6.7%	+6.2%	+6.1%	+5.9%
OR	Taille de l'historique	+8.2%	+4.9%	+4.8%	+3.3%
	Trafic réseau	+2.9%	+2.6%	+2.3%	+1.9%

TABLE 8.1 – [G5K] Surcoût de *FullReview* comparé à *PeerReview*, pour les deux applications SplitStream (SS) et Onion routing (OR), avec une période d'audit allant de 1s à 30s.

Nous finissons ces expériences par une mesure analytique du coût des opérations effectuées par les surveillants dans *PeerReview* et *FullReview*, indépendamment de l'application qu'ils surveillent. Nous nous intéressons seulement aux sous protocoles les plus coûteux (protocole de cohérence et protocole d'audit). Ces coûts figurent dans la table 8.2. Cette table explique le pourquoi du surcoût lié à l'utilisation *FullReview*.

	Sous protocoles	Cohérence	Audit
Opérations cryptographiques	*PeerReview*	ψP	ψP
	FullReview	$2\psi P$	$2\psi P$
Nouvelles entrées dans l'historique	*PeerReview*	0	0
	FullReview	ψP	ψP

TABLE 8.2 – Surcoût lié à *FullReview* comparé à *PeerReview* pour un échange de message, indépendamment d'une application. La période d'audit est P et ψ représente le nombre de surveillants.

Pour résumer, *FullReview* rajoute un faible surcoût de plus que *PeerReview* en terme de trafic généré et de la taille de l'historique. Ce surcoût est principalement dû aux nouvelles entrées insérées par *FullReview* pour détecter les nœuds rationnels. De façon similaire à *PeerReview*, le coût de *FullReview* augmente avec le nombre de surveillants par nœud et avec la fréquence des audits. Globalement, en comptant sur l'augmentation des ressources (stockage et bande passante) à la disposition du grand public, le coût

pour renforcer la notion de responsabilité en présence des nœuds rationnels devient une option réaliste.

8.5 Passage à l'échelle de *FullReview*

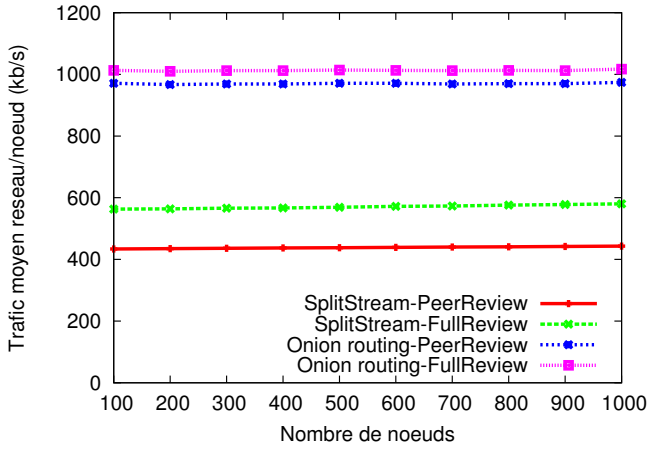
Dans cette section nous montrons que SplitStream-*FullReview* et Onion routing-*FullReview* passe à l'échelle jusqu'au moins 1000 nœuds.

La figure 8.6 présente le trafic réseau et la vitesse d'augmentation de l'historique de SplitStream et Onion routing, surveillée chacune par *PeerReview* et *FullReview*, en fonction du nombre de nœuds dans le système. Chaque valeur est mesurée grâce à une simulation qui dure 100s. En plus, le système est configuré avec 5 surveillants par nœud. Comme indiqué sur la figure 8.5, utiliser moins de surveillants fournit de meilleure performance. En plus, la période d'audit est fixée à 10s.
L'application *Onion routing* est configurée avec 40 relais et envoie des oignons à la vitesse de 16kb/s.

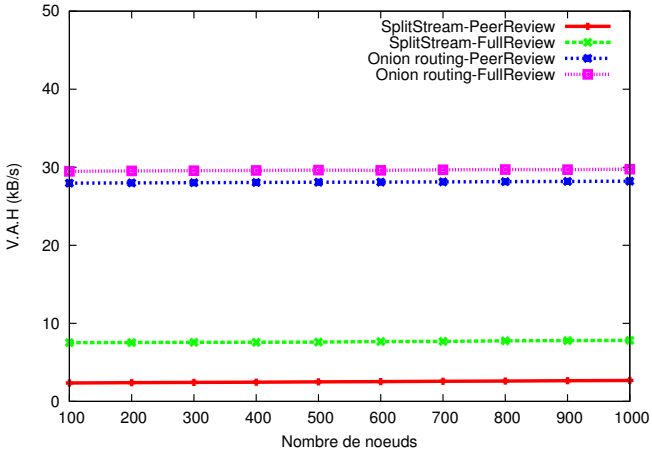
Sur cette figure nous pouvons tirer les conclusions suivantes. Tout d'abord, pour les deux applications *SplitStream* et *Onion routing*, le trafic réseau et la vitesse d'augmentation de l'historique de *FullReview* sont proportionnels (facteur constant) à ceux de *PeerReview*. Par exemple, avec *SplitStream*, la vitesse d'augmentation de l'historique (resp. le trafic réseau) de *FullReview* est égal à 1.4x (resp. 1.3x) à ceux de *PeerReview*. Cela est dû au fait que *FullReview* rajoute un nombre constant d'opérations à celles effectuées par *PeerReview*. Ensuite, nous pouvons observer que *FullReview* monte en charge jusqu'à 1000 nœuds, car le trafic réseau et la taille de l'historique restent équitablement stables malgré l'augmentation du nombre de nœuds. L'explication que nous pouvons donner à cette stabilité est le fait que chaque nœud interagit toujours avec le même nombre de nœuds en moyenne, quelque soit le nombre de nœuds dans le système (i.e. ses partenaires en fonction de l'application et du nombre de surveillants fixé).

8.6 Conclusion

Ce chapitre a porté sur l'évaluation du protocole *FullReview* comparativement à *PeerReview*. Pour rappel, cette évaluation avait trois objectifs : (1) montrer que *FullReview* détecte bien les comportements rationnels en présence de comportements rationnels, (2) évaluer le surcoût lié à l'application de *FullReview* et (3) montrer que *FullReview* passe à l'échelle.
Ainsi pour atteindre ces trois objectifs, nous avons évalué *FullReview* sur une grappe de machines physiques et en utilisant des simulations avec deux applications : *SplitStream*, un protocole de diffusion efficace, et *Onion routing* le protocole de communication anonyme le plus utilisé. A l'issue ce cette évaluation, nous pouvons tirer les conclusions suivantes :



(a) Trafic réseau.



(b) Taille de l'historique.

FIGURE 8.6 – [SIM] Moyenne du trafic réseau et de la vitesse d'augmentation de l'historique (V.A.H) de SplitStream et Onion routing en fonction du nombre de nœuds dans le système.

1. contrairement à *PeerReview*, *FullReview* tolère effectivement les comportements rationnels ;
2. *FullReview* a un surcoût faible comparé à *PeerReview* ;
3. *FullReview* passe à l'échelle et monte jusqu'à 1000 nœuds.

9

Conclusion

Sommaire

9.1 Objectif et contributions 81

9.2 Perspectives 83

Ce chapitre résume les différents chapitres abordés dans ce manuscrit, qui s'inscrit dans le cadre de la tolérance aux fautes dans les systèmes informatiques répartis. Pour cela nous allons d'abord rappeler l'objectif principal cette thèse et les contributions en faisant le lien avec les différents chapitres. Nous allons ensuite donner quelques pistes en guise de perspectives.

9.1 Objectif et contributions

Comme annoncé dans le premier chapitre (introduction), notre objectif principal était de concevoir un protocole générique sous-jacent qui détecte les comportements dits rationnels non seulement au niveau de ses propres étapes mais aussi niveau du système auquel il est appliqué, afin de contribuer à la tolérance aux fautes dans les systèmes distribués, en particulier la partie détection de fautes.

Pour atteindre cet objectif, nous avons divisé ce manuscrit en plusieurs chapitres qui sont plus ou moins liés. Ainsi :

Chapitre 2 Ce chapitre a porté sur la tolérance aux fautes dans les systèmes distribués de façon générale. Il décrivait les types de fautes et les approches utilisées dans la littérature pour les traiter ;

Chapitre 3 Ce chapitre s'intéressait particulièrement au cas des comportements rationnels. Pour rappel un nœud d'un système se comporte rationnellement lorsqu'il essaye de tirer parti du système dans y contribuer. Plusieurs solutions ont été

proposées dans la littérature pour résoudre ce problème et parmi toutes, le modèle BAR s'est révélée efficace pour le traiter. Cependant ce modèle présente des lacunes car il est manuel, et peut être source d'introduction de fautes.

Chapitre 4 La notion de responsabilité se référant à la capacité de détecter les fautes, identifier les nœuds fautifs et convaincre les autres nœuds d'un système, a été abordé dans ce chapitre. Elle s'inscrit dans le cadre de la détection des fautes. Cette technique combinée avec le modèle BAR a constitué le cœur de notre approche.

Chapitre 5 Ce chapitre a présenté *PeerReview*, la première solution logicielle générique au problème de la notion de responsabilité dans les systèmes informatiques répartis. Dans ce protocole, chaque participant a un historique sécurisé par les moyens cryptographiques, dans lequel il enregistre ses échanges avec les autres participants du système. Périodiquement, cet historique est audité par un sous ensemble de participants (les surveillants) afin de détecter les fautes. Cependant, ce protocole est vulnérable aux comportements rationnels et l'impact de ces comportements a été également étudié dans ce chapitre.

Chapitre 6 *FullReview*, la contribution principale de cette thèse a été présentée dans ce chapitre. Ce protocole combine le modèle BAR et la notion de responsabilité pour traiter les comportements rationnels non seulement au niveau de ses étapes mais aussi au niveau de l'application qu'il surveille. Pour cela, il part de *PeerReview* et rajoute une couche à ce dernier permettant de forcer ou de motiver les rationnels à participer aux différentes étapes.

Chapitre 7 Ce chapitre a porté sur les différentes méthodes de gestion des surveillants dans un protocole responsable. L'idée était d'étudier les différentes manières de déploiement des surveillants afin d'avoir moins d'échanges de messages. De cette étude, nous sommes parvenus à trois méthodes de gestion : la gestion statique, dynamique et sans surveillants. La gestion statique est celle dans laquelle chaque nœud possède un nombre fixe de surveillants. C'est la gestion par défaut utilisée dans *FullReview* et *PeerReview*. La gestion dynamique utilise la notion de vues. En effet dans cette gestion chaque nœud échange avec un ensemble de nœuds de taille variable, appelé vue. Chaque nœud dissémine ses authentificateurs dans sa vue. Enfin, la gestion sans surveillants utilise une structure de multiples anneaux pour organiser les nœuds du système. Ainsi chaque nœud a un successeur et un prédécesseur sur chaque anneau. Les authentificateurs sont disséminés dans la liste des successeurs.

Chapitre 8 Le protocole *FullReview* a été évalué dans ce chapitre comparativement à *PeerReview*. Cette évaluation a tiré les conclusions suivantes :

- *FullReview* détecte les comportements rationnels à tous les niveaux contrairement à *PeerReview*.
- *FullReview* induit un surcoût faible comparé à *PeerReview*.
- *FullReview* passe à l'échelle et peut tourner sur 1000 nœuds.

Le diagramme présenté sur la figure 9.1 montre comment nous sommes arrivés à nos contributions. En effet, nous sommes partis de la tolérance aux fautes (chapitre 2) en présentant les différents types de fautes, ensuite nous nous sommes intéressés particulièrement aux comportements rationnels (chapitre 3). La notion de responsabilité incluse dans le domaine de la tolérance aux fautes a été introduite (chapitre 4), afin de mieux comprendre le protocole *PeerReview* (chapitre 5). Nous avons combiné ensuite le modèle BAR vu au chapitre 3 et le protocole *PeerReview* pour aboutir à la première contribution au chapitre 6 (*FullReview*).
La seconde contribution (gestion des surveillants au chapitre 7) est obtenue grâce aux protocoles *PeerReview* et *FullReview*.

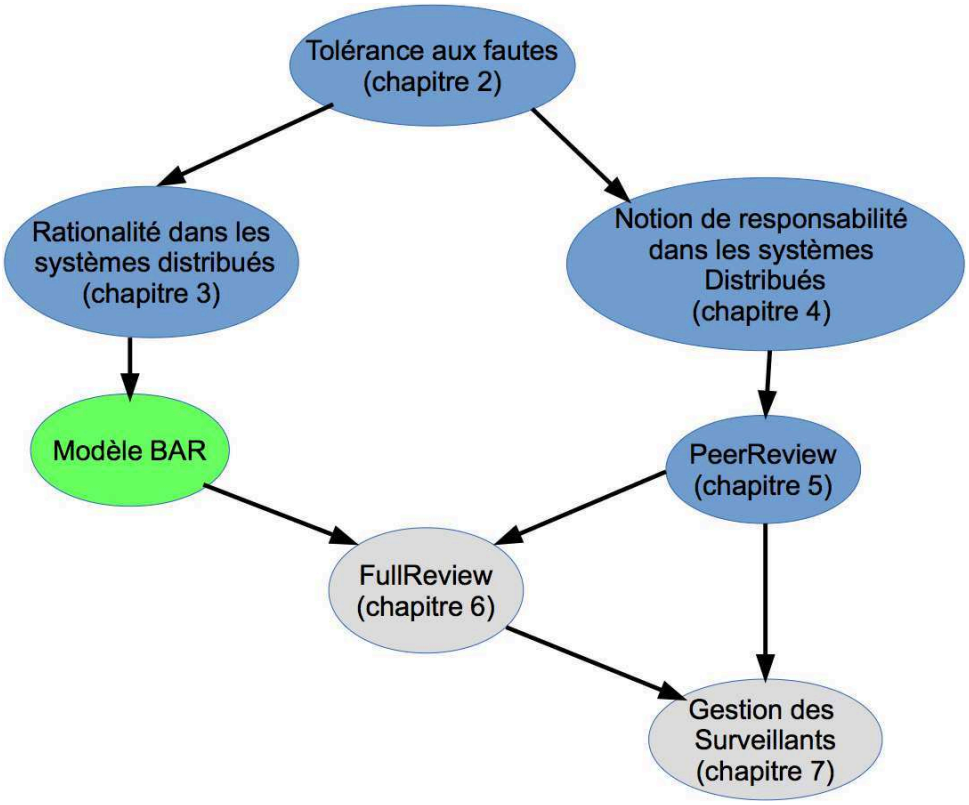


FIGURE 9.1 – Diagramme montrant les liens entre les différents chapitres.

9.2 Perspectives

Les comportements rationnels ne sont pas propres seulement aux systèmes distribués, il a été montré aussi qu'ils ont un grand impact dans les environnements mobiles [51, 52, 53]. Il existe plusieurs approches qui traitent plus ou moins ces com-

portements dans ces environnements. Elles sont souvent spécifiques à un type d'environnement mobile donné.
Cependant, aucune d'entres elles n'utilise la notion de responsabilité pour pallier à ce problème. Ainsi la suite de ces travaux pourraient être l'application de la technique de responsabilité dans les environnements mobiles tout en motivant les rationnels à participer aux différentes étapes du protocole.

Bibliographie

[1] R. Rodrigues and P. Druschel, "Peer-to-peer systems," *Commun. ACM*, vol. 53, pp. 72–82, 2010.

[2] B. Jacob, M. Brown, K. Fukui, and N. Travedi, "Introduction to grid computing," *IBM Redbooks*, December 2005.

[3] K. J. D. P. Mockapetris, "Development of the domain name system." in *SIGCOMM*, vol. 18, 1988, pp. 123–133.

[4] J. P. John, E. Katz-Bassett, A. Krishnamurthy, T. Anderson, and A. Venkataramani, "Consensus routing : The internet as a distributed systems." in *NSDI*, 2008.

[5] A. A. *et al.*, "Bar fault tolerance for cooperative services," in *Proceedings of SOSP*, 2005.

[6] F. Cristian, "Understanding fault-tolerant distributed systems," *COMMUNICATIONS OF THE ACM*, vol. 34, pp. 56–78, 1993.

[7] D. H. et al., "Free Riding on Gnutella Revisited : The Bell Tolls ?" *IEEE Distributed Systems Online*, 2005.

[8] S. Ben Mokhtar *et al.*, "Firespam : Spam resilient gossiping in the bar model," in *Proceedings of SRDS*, 2010.

[9] H. Li *et al.*, "Bar gossip," in *Proceedings of OSDI*, 2006.

[10] J. Nash, "Non-Cooperative Games," *Annals of Mathematics*, vol. 54, no. 2, 1951.

[11] J. Arlat, Y. Crouzet, Y. Deswarte, J.-C. Fabre, J.-C. Laprie, and D. Powell, *Tolérance aux fautes*. Les Editions Vuibert, 2006.

[12] B. R. A. Avizienis, JC. Laprie and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing." vol. 1. IEEE TDSC, January 2004, pp. 11–33.

[13] R. S. Leslie Lamport and M. Pease, "The byzantine generals problem." vol. 4. ACM TOPLAS, July 1982, pp. 382–401.

[14] B. W. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems*. Addison-Wesley Publishing, 1989.

[15] A. Haeberlen, P. Kouznetsov, and P. Druschel, "Peerreview : Practical accountability for distributed systems," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 175–188, 2007.

[16] J.-L. Roch and S. Varrette, "Probabilistic certification of divide and conquer algorithms on global computing platforms. application to fault-tolerant exact matrix-vector product." in *Parallel Symbolic Computation*. ACM, London, Ontario, Canada, July 2007.

[17] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach : a tutorial," vol. 22. ACM, December 1990, pp. 299–319.

[18] R. Guerraoui, K. Huguenin, A.-M. Kermarrec, M. Monod, and S. Prusty, "Lifting : lightweight freerider-tracking in gossip," in *Proceedings of the ACM/IFIP/USENIX 11th International Conference on Middleware*. Springer-Verlag, 2010, pp. 313–333.

[19] X. Vilaça, J. Leitao, M. Correia, and L. Rodrigues, "N-party bar transfer," in *Principles of Distributed Systems*. Springer, 2011, pp. 392–408.

[20] S. Ben Mokhtar, G. Berthou, A. Diarra, V. Quéma, and A. Shoker, "RAC : a freerider-resilient, scalable, anonymous communication protocol," in *Proceedings of the 33rd International Conference on Distributed Computing Systems (ICDCS)*, 2013.

[21] H. C. Li, A. Clement, M. Marchetti, M. Kapritsos, L. Robison, L. Alvisi, and M. Dahlin, "Flightpath : Obedience vs choice in cooperative services," in *In OSDI 2008*, 2008.

[22] S. Ben Mokhtar, J. Decouchant, and V. Quema, "Acting : Accurate freerider tracking in gossip." SRDS, October 2014, pp. 291–300.

[23] J. Shneidman and D. C. Parkes, "Rationality and self-interest in peer to peer newtworks." In Peer-to-peer systems II second international workshop, IPTPS, 2003, Berkeley, CA, USA.

[24] D. S. Adrian Perrig, Sean Smith and J. D. Tygar, "Sam : A flexible and secure auction architecture using trusted hardware." Submitted Manuscript, 1991.

[25] J. Feigenbaum and S. Shenker, "Distributed algorithmic mechanism design : Recent results and future directions," in *In Proceedings of the 6th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*. ACM Press, 2002, pp. 1–13.

[26] Y. Afek, Y. Ginzberg, S. Landau Feibish, and M. Sulamy, "Distributed computing building blocks for rational agents," in *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, ser. PODC '14. New York, NY, USA : ACM, 2014, pp. 406–415.

[27] G. Lena Cota, P.-L. Aublin, S. Ben Mokhtar, G. Gianini, E. Damiani, and L. Brunie, "A semi-automatic framework for the design of rational resilient collaborative systems," LIRIS laboratory, Tech. Rep., 2014.

[28] J. J.-D. Mol, J. A. Pouwelse, M. Meulpolder, D. H. Epema, and H. J. Sips, "Give-to-get : free-riding resilient video-on-demand in p2p systems," in *Electronic Imaging 2008*. International Society for Optics and Photonics, 2008, pp. 681 804–681 804.

[29] C. Ho, R. Van Renesse, M. Bickford, and D. Dolev, "Nysiad : Practical protocol transformation to tolerate byzantine failures." in *NSDI*, vol. 8, 2008, pp. 175–188.

[30] H. Nissenbaum, "Computing and accountability." vol. 37. ACM, January 1994, pp. 72–80.

[31] Y. Xiao, "Flow-net methodology for accountability in wireless networks." vol. 23. IEEE Network, September 2009, pp. 30–37.

[32] K. M. Yang Xiao and T. D., "Implementation and evaluation of accountability using flow-net in wireless networks." vol. 23. MILCOM, October 2010, pp. 7–12.

[33] H. C.-G. al., "Dissent : accountable anonymous group messagin erratum 2," yale, Tech. Rep., 2010.

[34] A. R. Yumerefendi and J. S. Chase, "Strong accountability for network storage," *ACM Transactions on Storage (TOS)*, vol. 3, no. 3, p. 11, 2007.

[35] A. Haeberlen, I. C. Avramopoulos, J. Rexford, and P. Druschel, "Netreview : Detecting when interdomain routing goes wrong." in *NSDI*, 2009, pp. 437–452.

[36] R. S. N. Michalakis and R. Grimm, "Ensuring content integrity for untrusted peer-to-peer content distribution networks." NSDI, 2007, pp. 11–11.

[37] A. H. M. Backes, P.D. Druschel and D. Unruh, "Csar : A practical and provable technique to make randomized systems accountable." NDSS, 2009.

[38] D. Anderson, H. Balakrishnan, N. Feamster, T. Koponen, D. Moon, and S. Shenker, "Csar : Accountable internet protocol (aip)." vol. 38. SIGCOMM, October 2008, pp. 339–350.

[39] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda, "Trinc : Small trusted hardware for large distributed systems." in *NSDI*, vol. 9, 2009, pp. 1–14.

[40] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz, "Attested append-only memory : Making adversaries stick to their word," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 189–204, 2007.

[41] R. Kotla, T. Rodeheffer, I. Roy, P. Stuedi, and B. Wester, "Pasture : secure offline data access using commodity trusted hardware," in *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*. USENIX Association, 2012, pp. 321–334.

[42] A. Haeberlen, P. Aditya, R. Rodrigues, and P. Druschel, "Accountable virtual machines." in *OSDI*, 2010, pp. 119–134.

[43] M. C. et al., "Practical byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems (TOCS)*, vol. 20, no. 4, 2002.

[44] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh, "Splitstream : high-bandwidth multicast in cooperative environments," in *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5. ACM, 2003, pp. 298–313.

[45] D. Goldschlag, M. Reed, and P. Syverson, "Onion routing," *Commun. ACM*, vol. 42, no. 2, 1999.

[46] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. Van Steen, "Gossip-based peer sampling," *ACM Transactions on Computer Systems (TOCS)*, vol. 25, no. 3, p. 8, 2007.

[47] H. Johansen *et al.*, "Fireflies : scalable support for intrusion-tolerant network overlays," in *Proceedings of EuroSys*, 2006.

[48] A. M. Kermarrec, L. Massoulie, and A. Ganesh, "Probabilistic reliable dissemination in large-scale systems," *IEEE TPDS*, vol. 14, no. 3, 2003.

[49] R. Dingledine *et al.*, "Tor : the second-generation onion router," in *Proceedings of USENIX Security Symposium*, 2004.

[50] H. Corrigan-Gibbs and B. Ford, "Dissent : accountable anonymous group messaging," in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 340–350.

[51] A. Mei and J. Stefa, "Give2get : Forwarding in social mobile wireless networks of selfish individuals." ICDCS, June 2010, pp. 488–497.

[52] B. Chen and M. Chan, "Mobicent : a credit-based incentive system for disruption tolerant network." INFOCOM, March 2010, pp. 1–9.

[53] S. Z. Qinghua Li and G. Cao, "Routing in socially selfish delay tolerant networks." INFOCOM, March 2010, pp. 1–9.

Table des figures

1.1 Exemple d'un système distribué avec différents domaines d'administration 2

3.1 Une application de vidéo en direct 10
3.2 Classes de protocoles dans le modèle BAR 12

4.1 Architecture simple d'un système responsable de ses actes. 17
4.2 Exemple d'un historique sécurisé. 17
4.3 Vue globale du protocole CATS 20
4.4 Vue globale du protocole NetReview 21
4.5 Vue globale du protocole *Repeat and Compare* 22
4.6 Structure de A2M 24
4.7 Architecture de Pasture 25
4.8 Scénario d'illustration du protocole AVM 26

5.1 Modèle de système du protocole *PeerReview* 31
5.2 Protocole d'engagement et de cohérence 32
5.3 Protocole d'audit 33
5.4 Protocole de challenge/réponse 34
5.5 Protocole de transfert de preuve 34
5.6 Impact des nœuds rationnels sur les applications SplitStream et Onion routing surveillées par *PeerReview*. 36

6.1 Modèle de système de *FullReview*. 41
6.2 Diagramme de décision des surveillants du protocole *FullReview*. . . 44
6.3 Protocole d'audit de *FullReview*. 45
6.4 Protocole P augmenté. 47
6.5 Envoi des requêtes d'audit. 47
6.6 Traitement des requêtes d'audit. 47
6.7 Protocole de gestion de pannes d'omission de *FullReview* 50
6.8 Gestion des pannes d'omission. 50
6.9 Gestion des suspicions. 50

7.1 Gestion statique : Scénario des échanges de messages. 57

7.2 Gestion dynamique : Scénario des échanges de messages. 59
7.3 Gestion sans surveillants : Scénario des échanges de messages. 60
7.4 Nombre de messages échangés dans *PeerReview* en fonction du nombre de nœuds lorsqu'il surveille SplitStream. Le cas 1 correspond à $l = 50\%N$, $v_i = ln(N)$, $\psi = 5$ et $R = 5$. Le cas 2 correspond à $l = 50\%N$, $v_i = ln(N) + 5$, $\psi = 5$ et $R = 5$. 62
7.5 Cas 3 : Nombre de messages échangés dans *PeerReview* en fonction du nombre de nœuds lorsqu'il surveille SplitStream. Ici $l = 50\%N$, $v_i = ln(N) + 20$, $\psi = 3$ et $R = 9$. 63
7.6 Nombre de messages échangés dans *PeerReview* en fonction du nombre de nœuds lorsqu'il surveille Onion routing. Le cas 1 correspond à $l = 50\%N$, $v_i = ln(N)$, $\psi = 5$ et $R = 5$. Le cas 2 correspond à $l = 50\%N$, $v_i = ln(N) + 15$, $\psi = 5$ et $R = 5$ 64
7.7 Case 3 : Nombre de messages échangés dans *PeerReview* en fonction du nombre de nœuds lorsqu'il surveille SplitStream. Ici $l = 50\%N$, $v_i = ln(N) + 25$, $\psi = 3$ et $R = 9$. 65

8.1 Pourcentage de messages reçus dans SplitStream et Onion routing en fonction du pourcentage de nœuds rationnels. 71
8.2 Impact du nombre de relais sur le pourcentage de messages reçus dans Onion routing-*PeerReview*. 72
8.3 [SIM] SplitStream Pourcentage de messages reçus durant une expérience dans laquelle entre 10% et 50% de nœuds commencent à agir rationnellement après 20s. 73
8.4 [SIM] Moyenne du trafic réseau et de la vitesse d'augmentation de l'historique (V.A.H) par nœud de *SplitStream*(SS) et *Onion routing*(OR) en fonction du nombre de surveillants. 74
8.5 [G5K] Moyenne du trafic réseau et de la vitesse d'augmentation de l'historique (V.A.H) par nœud de SplitStream (SS) et Onion routing (OR) en fonction du nombre de surveillants. 75
8.6 [SIM] Moyenne du trafic réseau et de la vitesse d'augmentation de l'historique (V.A.H) de SplitStream et Onion routing en fonction du nombre de nœuds dans le système. 79

9.1 Diagramme montrant les liens entre les différents chapitres. 83

Liste des tableaux

I Termes utilisés dans ce manuscript et leur signification. ix

4.1 Résumé des protocoles étudiés dans l'état de l'art. SS = Solution Spécifique, SG = Solution Générique, TSD = Type de Système Distribué auquel le protocole est appliqué. 26

7.1 Le nombre d'échanges de messages dans *PeerReview* comparé à celui dans *FullReview* selon les différentes méthodes de gestion. TP = Transfert de Preuve 57
7.2 SplitStream : Valeurs des paramètres 61
7.3 Onion routing : Valeurs des paramètres 62

8.1 [G5K] Surcoût de *FullReview* comparé à *PeerReview*, pour les deux applications SplitStream (SS) et Onion routing (OR), avec une période d'audit allant de 1s à 30s. 77
8.2 Surcoût lié à *FullReview* comparé à *PeerReview* pour un échange de message, indépendamment d'une application. La période d'audit est P et ψ représente le nombre de surveillants. 77

Pseudo-code des différents sous protocoles de *FullReview*

Cette annexe présente le pseudo-code des différents sous protocoles du protocole *FullReview*.

A.1 Protocole de consistance

```
receive(m){
      //Verify if message m is an authenticator
      if(m.type == AUTH){
            //Get the list of source node's monitors
            monitors_list = getMonitors(m.node);
            //Forward m to nodes in M set
            for(node in monitors_list){
                  send(m,node);
            }
      }
      //Verify if m is a log snippet
      if(m.type == LOG){
            //Extract authenticators from this log snippet
            auth_list = extract(m);
            //Forward each extracted authenticator to monitors
            for(auth in auth_list){
                  forwardToMonitors(auth);
            }
      }
}
```

A.2 Protocole d'audit

```
//Each node sends its log to its monitors since last audit
M = getMonitors(node_current);
for(node in M){
     send(log, node);
}
//upon reception
receive(m){
     if(m.type == LOG){
          /*Generate automatically an automata from
           P and M state machines*/
          aut = generate(P.state_machine,M.state_machine);
          /*Verify if patterns of communication are recognized by
            automata aut. Indeed communication patterns appearing
            in log must match with a correct execution of
            P and M protocols*/
          correct = false;
          for(comPat in m){
               if(isRecognized(comPat,aut)){
                    correct = true;
               }
               else{
                    correct = false;
                    break;
               }
          }
          //If correct is set to false, the audited node is suspect
          if(!correct){
               suspect(m.node);
          }

          //Send the outcome to the audited node
          outcome = correct;
          M = getMonitors(m.node);
          for(node in M){
               send(outcome,node);
          }
     }
    if(m.type == AUDIT_RESP){
       /*Aggregate outcomes and send the result
        to monitors of audited node's monitors.
        Each outcome is added into a list*/
        aggregated_list.add(m);

       //Get the list of monitors
        M = getMonitors(m.node);

       /*If the size of aggregated_list is equal
        to the number of monitors then the current
        node have received the outcome of each of
```

```
        its monitors. Then the list is sent*/
        if(aggregated_list.size == MONITORS_SIZE){
           for(node in M){
            //monitors of audited node's monitors
            MM_list.add(getMonitors(node));
           }
           send(aggregated_list, MM_list);
        }else{
             /*After a given amount of time, monitors
                 that do not send their outcome are suspected
               */
             if(timeout){
                 /*Get the list of nodes that
                  have sent their outcome*/
                 for(l in aggregated_list){
                   list_node.add(l.node);
                 }
                 for(node in M){
                   if(node not in list_node){
                      suspect(node);
                    }
                 }
               }
        }
     }
    if(m.type == OUTCOME){
       /*Verify if all elements in the list(aggregated_list) are
         equals. If not, elements that are different from
          the element having the higher number of occurrence,
          are suspected*/
      //Get the aggregated list
       a_list = m.content;
       /*Find the number of occurence of each element
        in a_list. This return a list of couple
        (elt,nb_occ)*/
       list_occ = findOccurrence(a_list);

       //Find the element having the higer number of occurence
       (elt,max) = maxOccurrence(list_occ);

       for((e,n) in list_occ){
          if(n!=max){
             supect(e.node);
          }
       }
     }
}
```

A.3 Protocole de challenge réponse

```
//Create a challenge for a suspected node
chal = createChall(suspected_node);

//Send the challenge to monitors of suspected node
M = getMoniors(suspected_node);
for(node in M){
   send(chal,node);
}
//Upon reception
receive(m){
  /*If m is a challenge then it's forwarded to
   nodes that current node monitors*/
  if(m.type == CHALL){
    monitored_nodes = getMonitoredNodes(m.node);
     for(node in monitored_nodes){
        send(m,node);
     }
  }
}
```

A.4 Protocole d'omission de pannes

```
/*This protocol is executed when a node
 waits for too long time a message from
 another node*/
if(timeout){
   //Challenge the waiting node
   challenge(waiting_node);
}

//upon reception
receive(m){
  /*If m is a challenge, an reply is sent
   to current node's monitors*/
   if(m.type == CHALL){
     M = getMonitors(current_node);
      rep = createReply();
      for(node in M){
         send(rep,node);
      }
   }
   /*If m is a outcome of a challenge, it's forwarded
    to node that creates the challenge initially*/
   if(m.type == CHALL_OUT){
     //Get node that creates the challenge
      node = getChallAuthor(m);
```

```
            send(node,m);
        }
    }
```