Contributions To Building Reliable Distributed Systems

Sonia Ben Mokhtar

▶ To cite this version:

Sonia Ben Mokhtar. Contributions To Building Reliable Distributed Systems. Performance [cs.PF]. INSA de Lyon, 2017. tel-01936371

HAL Id: tel-01936371
https://hal.science/tel-01936371

Submitted on 27 Nov 2018

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

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

UNIVERSITÉ DE LYON

École doctorale InfoMath de Lyon

Contributions To Building Reliable Distributed Systems

THÈSE

Soutenue le 21/12/2017 devant la commission d'examen

pour l'obtention d'une

Habilitation de l'Institut National des Sciences Appliquées de Lyon et l'Université Claude Bernard LYON I

(mention informatique)

par

Sonia BEN MOKHTAR

Composition du jury

Rapporteurs :	Pr. Rachid Guerraoui	École Polytechnique Fédérale de Lausanne
	Dr. Karama Kanoun	Laboratoire d'Analyse et d'Architecture des Systèmes
	Pr. Paulo Esteves-Verissimo	Université du Luxembourg
Examinateurs :	Pr. Gordon Blair	Lancaster University
	Pr. Lionel Brunie	INSA Lyon
	Pr. Pascal Felber	Université de Neuchâtel
	Pr. Mohand-Said Hacid	Université Claude Bernard Lyon 1
	Dr. Valérie Issarny	INRIA Sillicon Valley
	Dr. Gilles Muller	INRIA Paris
	Pr. Vivien Quéma	Grenoble INP

Laboratoire d'InfoRmatique en Image et Systèmes d'information (LIRIS)— UMR 5205

Mis en page avec la classe thesul.

à Lisie et à Fatzoh...

Table Of Contents

Table des figures **7**

Liste des tableaux **11**

Abstract

1
Introduction

1.1 Faults in today's distributed systems . . . 15
1.2 Contributions . . . 16
1.2.1 Robust Byzantine-fault tolerant state machine replication . . . 16
1.2.2 Robust Byzantine fault detection . . . 16
1.2.3 Privacy-preserving Byzantine fault detection . . . 17
1.2.4 Dealing with rational nodes in gossip-based spam filtering . . . 17
1.2.5 Dealing with colluding rational nodes . . . 18
1.2.6 Dealing with rational nodes in anonymous communication . . . 18
1.2.7 A framework for designing and injecting rational faults . . . 18
1.2.8 Automatically dealing with rational nodes . . . 18
1.3 Other contributions not included in this thesis . . . 19
1.3.1 Middleware for mobile systems . . . 19
1.3.2 Enforcing private Web search . . . 20
1.3.3 Enforcing location privacy . . . 21
1.4 Thesis Outline . . . 22

I Dealing with Arbitrary Faults **23**

2
***RBFT* : Redundant Byzantine Fault Tolerance**

2.1 Introduction . . . 25
2.2 System model . . . 26
2.3 Analysis of existing robust BFT protocols . . . 27
2.3.1 Prime . . . 27

2.3.2 Aardvark 28
2.3.3 Spinning 28
2.3.4 Summary 29
2.4 The *RBFT* protocol 30
2.4.1 Protocol overview 30
2.4.2 Detailed protocol steps 31
2.4.3 Monitoring mechanism 32
2.4.4 Protocol instance change mechanism 33
2.5 Implementation 33
2.6 Performance evaluation 34
2.6.1 Experimental settings 34
2.6.2 Fault-free case 35
2.6.3 *RBFT* under attack 36
2.7 Conclusion 38

3
***FullReview* : Practical Accountability in Presence of Selfish Nodes**

3.1 Introduction 41
3.2 Related works 42
3.3 Problem statement and system model 43
3.3.1 Problem statement 43
3.3.2 System model 44
3.4 *FullReview* protocol overview 45
3.5 *FullReview* detailed description 46
3.5.1 Accountability tools : Tamper Evident log 46
3.5.2 *FullReview* selfish-resilient audit protocol 47
3.5.3 Handling omission failures 50
3.5.4 Resilience to selfish nodes 50
3.6 Performance Evaluation 51
3.6.1 Applications 51
3.6.2 Experimental settings 51
3.6.3 Performance in presence of selfish nodes 52
3.6.4 Performance in the fault-free case 53
3.6.5 Scalability of *FullReview* 54
3.7 Conclusion 55

4
***PAG* : Private and Accountable Gossip**

4.1 Introduction 57
4.2 Accountable Forwarding and Privacy 58
4.2.1 Principles of gossip and selfishness 58
4.2.2 Accountability solutions 59

4.2.3 Privacy of users . . . 60
4.3 System Model and Objectives . . . 60
4.4 *PAG* in a Nutshell . . . 60
4.4.1 Enforcing accountability using a monitoring infrastructure . . . 61
4.4.2 Enforcing privacy using homomorphic hashes . . . 62
4.5 *PAG* Detailed Description . . . 63
4.5.1 Transmission of updates between monitored nodes . . . 63
4.5.2 Transmission of hashes to the monitoring infrastructure . . . 64
4.5.3 Homomorphic combination of hashes by monitors . . . 64
4.5.4 Application to a content-dissemination system. . . . 64
4.6 Privacy and Accountability Analysis . . . 65
4.6.1 Privacy Guarantees . . . 65
4.6.2 Accountability Analysis . . . 66
4.7 Performance Evaluation . . . 66
4.7.1 Methodology and Parameter Settings . . . 67
4.7.2 Comparisons with existing protocols . . . 67
4.7.3 Cryptographic costs . . . 69
4.7.4 Scalability . . . 69
4.7.5 Probabilistic study of the impact of coalitions on privacy . . . 69
4.8 Related Works . . . 70
4.9 Conclusion . . . 71

II Dealing with Rational Faults 73

5
***FireSpam* : Spam Resilient Gossiping in the BAR Model**

5.1 Introduction . . . 75
5.2 System Model . . . 76
5.3 FireSpam Design . . . 77
5.3.1 Ladder topology . . . 77
5.3.2 Ladder construction challenges . . . 78
5.3.3 *FireSpam* description . . . 79
5.4 FireSpam Robustness . . . 81
5.5 Performance Evaluation . . . 82
5.5.1 Correctness of forwarding views . . . 83
5.5.2 Reliability of good messages delivery . . . 83
5.5.3 Ratio of spam messages received . . . 83
5.5.4 Behavior under an eclipse attack . . . 84
5.5.5 Bandwidth consumption . . . 86
5.6 Related Work . . . 86
5.7 Conclusion . . . 87

6
***AcTing* : Accurate Freerider Tracking in Gossip**

6.1 Introduction 89
6.2 System Model 90
6.3 Protocol Overview 91
6.4 Protocol Details 93
6.4.1 Membership protocol 94
6.4.2 Partnership management 95
6.4.3 Audit protocol 96
6.4.4 Update exchanges 96
6.5 Risk versus gain analysis 97
6.6 Performance evaluation 99
6.6.1 Methodology and Parameter Setting 99
6.6.2 Impact of Colluders 100
6.6.3 Bandwidth consumption 102
6.6.4 Resilience to massive node departure 102
6.6.5 Scalability 104
6.7 Related Works 104
6.8 Conclusion 105

7
***RAC* : a Freerider-resilient, Scalable, Anonymous Communication Protocol**

7.1 Introduction 107
7.2 Definitions and related work 108
7.2.1 Definitions 108
7.2.2 Related work 109
7.3 The Case for a New Protocol 109
7.4 The *RAC* protocol 111
7.4.1 Key idea #1 : reducing the number of broadcasts 111
7.4.2 Key idea #2 : reducing the size of broadcast groups 112
7.4.3 Detailed description 112
7.5 Proofs 115
7.5.1 Anonymity proof 115
7.5.2 Freerider-resiliency proof 116
7.6 Evaluation 118
7.6.1 Simulation settings 118
7.6.2 Protocol configuration parameters 118
7.6.3 Throughput 119
7.6.4 Anonymity guarantees 119
7.7 Conclusion 121

8
Seine : a framework for designing and injecting rational faults

8.1 Introduction 123
8.2 Background 125
8.3 Domain Analysis 126
8.4 *SEINE* Overview 128
8.5 Modelling selfishness in *SEINE-L* 128
8.6 Injecting selfishness in PeerSim using *SEINE* 132
8.6.1 Library of Annotations 132
8.6.2 *SEINE-L* Compiler 133
8.6.3 Selfishness scenario generation 133
8.6.4 *SEINE* Implementation 134
8.7 Evaluation 135
8.7.1 Generality and expressiveness of *SEINE-L* 135
8.7.2 Accuracy of *SEINE-R* 135
8.7.3 Development effort 137
8.7.4 Simulation time 139
8.8 Related Work 140
8.9 Conclusion 141

9
RACOON **: A Framework for the Design of Accountable Selfish-Resilient Cooperative Systems**

9.1 Introduction 143
9.2 Background 145
9.3 *RACOON* Overview 146
9.4 *RACOON* Design Part 147
9.4.1 *RACOON* Specification Model 147
9.4.2 Generating Selfish Deviations 149
9.4.3 Game Mapping 149
9.5 *RACOON* Game-Based Simulation 152
9.6 Evaluation 153
9.6.1 *RACOON* Design Effort 153
9.6.2 Meeting System Objectives Using *RACOON* Simulations 154
9.6.3 Simulation Compared to Real System Deployment 154
9.6.4 *RACOON* Execution Time 156
9.6.5 *RACOON* Expressiveness 156
9.7 Related work 157
9.8 Conclusion 157

III Conclusion and Future Research Directions 159

10
Conclusion and Perspectives

10.1 Summary of contributions 161
10.2 Latest advances in building Byzantine tolerant systems 161
10.2.1 Making BFT protocols faster 162
10.2.2 Reducing the cost of BFT protocols 162
10.3 Latest advances in building rational resilient cooperative systems 162
10.4 Open research directions 163
10.4.1 Collaborative systems are not dead 163
10.4.2 The shift towards a Big Data world and the need of data privacy mechanisms 163
10.4.3 The shift towards a probabilistic world 164
10.5 Conclusion 165

Bibliographie 167

Table des figures

2.1 Prime throughput under attack relative to the throughput in the fault-free case. 28
2.2 Aardvark throughput under attack relative to the throughput in the fault-free case. 29
2.3 Spinning throughput under attack relative to the throughput in the fault-free case. 29
2.4 *RBFT* overview ($f = 1$). 30
2.5 *RBFT* protocol steps ($f = 1$). 31
2.6 Architecture of a single node ($f = 1$). The replicas (in circles) are processes that are independent from the different modules, implemented as threads (in rectangles). 33
2.7 Latency vs. throughput for various robust BFT protocols ($f = 1$). 34
2.8 *RBFT* throughput under *worst-attack-1* relative to the throughput in the fault-free case, for both a static and a dynamic load. 35
2.9 Throughput measured by the different nodes under worst attack 1 ($f = 1$, static workload, 4kB requests). 36
2.10 *RBFT* throughput under *worst-attack-2* relative to the throughput in the fault-free case, for both a static and a dynamic load. 37
2.11 Throughput measured by the different nodes under worst attack 2 ($f = 1$, static workload, 4kB requests). 37
2.12 Ordering latencies for the requests of two clients on the master protocol instance with an unfair primary, which starts to delay the request of one of the clients after 500 requests. . 38

3.1 Impact of selfish nodes in PeerReviewed SplitStream and Onion routing protocols. 44
3.2 Simple accountability architecture. 45
3.3 *FullReview* monitors decision diagram. 46
3.4 Example of a secure log. 47
3.5 *FullReview* audit protocol. 48
3.6 Augmenting the P protocol. 49
3.7 Sending audit requests. 49
3.8 Dealing with audit requests. 49
3.9 *FullReview* handling of omission failures 50
3.10 Dealing with omission failures. 50
3.11 Dealing with omission suspicions. 50
3.12 [G5K] Percentage of received messages in SplitStream and Onion routing as a function of the percentage of selfish nodes. 52
3.13 [SIM] SplitStream percentage of received messages during an experiment in which between 10% and 50% of nodes start to act selfishly after 20s. 53
3.14 [G5K] Average network traffic and log growing rate per node of SplitStream (SS) and Onion routing (OR) w.r.t. the number of monitors. 53
3.15 [SIM] Average network traffic and log growing rate of SplitStream and Onion routing w.r.t. the number of nodes in the system. 55

4.1 Forwarding of updates in a gossip-based system 59
4.2 Accountable gossip 59
4.3 Monitoring of nodes to ensure the forwarding of messages 61

4.4 Privacy preserving verification of a forwarding of a node B . . . 62
4.5 Exchange of updates between nodes . . . 63
4.6 Monitoring part of an interaction between two nodes . . . 65
4.7 Bandwidth consumption with a 300 kbps stream and 3 monitors . . . 67
4.8 Bandwidth consumption with 1000 nodes and a 300Kbps stream in function of the size of updates [sim] . . . 68
4.9 Scalability of *PAG* and *AcTing* with a 300Kbps content [sim] . . . 69
4.10 Resiliency against a global and active attacker . . . 70

5.1 *FireSpam* ladder topology. . . . 78
5.2 *FireSpam* roles. . . . 79
5.3 Forwarding views assigned by *FireSpam* as a function of the node filtering capability (5% of Byzantine nodes setting). . . . 83
5.4 Ratio of good messages delivery as a function of the number of hops. . . . 84
5.5 Ratio of spam messages that are received by nodes for a uniform (1st row), a power-law (2nd row), and an inverse power-law (3rd row) distribution of the spam filtering capabilities. 85
5.6 Ratio of spam messages that are received by nodes before the attack, during the attack, and after the attack (uniform distribution of the spam filtering capabilities). . . . 86

6.1 Overview of *AcTing*. . . . 92
6.2 Arrival of a new node. . . . 94
6.3 Handling of an omission failure. . . . 94
6.4 Establishment of new associations between nodes, which may imply audits. . . . 95
6.5 Update exchanges between nodes. . . . 97
6.6 Pseudocode of the algorithm used to estimate the number of times a colluding node avoids to send an update. . . . 98
6.7 Pseudocode of a part of the algorithm used to estimate the probability that a node received an update depending on the round number. . . . 98
6.8 Proportion of missed updates by correct nodes when a given proportion of the audience collude as a single group. . . . 101
6.9 Proportion of missed updates by correct nodes when 30% of the audience is rational, and collude in independent groups of equal sizes. . . . 101
6.10 Fault-free case : Cumulative distribution of average bandwidths. . . . 102
6.11 Nodes average bandwidth after a massive departure. . . . 103
6.12 Percentage of nodes that do not receive a viewable stream after a massive departure. . . . 103

7.1 Throughput as a function of the number of nodes for Dissent v1 and Dissent v2. . . . 110
7.2 Illustration of the *RAC* protocol. . . . 111
7.3 Throughput as a function of the number of nodes in the system for Dissent v1, Dissent v2, *RAC*-NoGroup and *RAC*-1000. . . . 119

8.1 Selfishness manifestations in gossip-based live streaming dissemination [Guerraoui et al., 2010a]. . . . 125
8.2 Feature diagram of a selfishness scenario. . . . 127
8.3 Overview of the *SEINE* framework. . . . 128
8.4 The outline of a *SEINE-L* specification. . . . 129
8.5 Comparison between the results published by Guerraoui et al. [Guerraoui et al., 2010a] and the results obtained with *SEINE*. . . . 136
8.6 Comparison between the results published by Ben Mokhtar et al. [Ben Mokhtar et al., 2014] and the results obtained with *SEINE*. . . . 137
8.7 Performance and contribution of BitTorrent and BitThief when downloading the same file, measured using *SEINE*. . . . 138

8.8 Number and distribution of Lines of Code (a) to specify the selfishness scenario into the faithful implementation of the use cases and (b) to modify such scenarios, with and without using *SEINE*. 138
8.9 Performance of BAR Gossip when varying (a) the number of colluding groups and (b) the fraction of resourceless mobile nodes. 140

9.1 Impact of the punishment values. 146
9.2 Impact of the audit period. 146
9.3 *RACOON* Overview. 147
9.4 The *R&R* protocol between nodes r_0 and R_1. 147
9.5 The Protocol Automaton of the *R&R* protocol. 149
9.6 The Protocol Automaton with selfish deviations. 151
9.7 The Protocol Game derived from the *R&R* protocol. 151
9.8 The sequence diagram of the $3P$ gossip protocol. 154
9.9 The Protocol Automaton of the $3P$ gossip protocol. 154
9.10 *RACOON* vs FullReview Configurations. 155
9.11 Simulation vs real deployment (logarithmic scale). 155
9.12 Onion Forwarding Protocol. 156

Liste des tableaux

2.1 Performance degradation of “robust” BFT protocols under attack. 26

3.1 [G5K] Overhead of *FullReview* compared to *PeerReview*, for both SplitStream (SS) and Onion routing (OR), with an audit period ranging from 1s to 30s. 54

4.1 Number of RSA signatures and homomorphic hashes per second in a system of 1000 nodes [sim] 68
4.2 Maximum video quality sustainable in function of the network links capacity, and the associated bandwidth consumption, in a system with 1000 nodes 68

6.1 Probability that a node receives an update in function of the number of rounds elapsed since its release by the source. 97
6.2 Overhead of colluders in *AcTing*. 100
6.3 Average bandwidth and memory usage of *AcTing* in function of the system size. 104

7.1 Anonymity guarantees of the various protocols in a system of 100.000 nodes. 120

8.1 The papers reviewed for the domain analysis, with information about their application domain and the selfishness investigated. 126
8.2 Lines of Code for expressing the selfishness scenarios of the papers considered in the domain analysis review. 135
8.3 Average execution time to evaluate a selfishness scenario using *SEINE* and the additional time it imposes. 140

9.1 Simulation and Real Deployment Parameters. 154
9.2 FullReview Configurations 155

Abstract

Thanks to the latest evolutions in hardware and networking technologies we live in a world where networked computing systems are everywhere ranging from small/medium daily objects (e.g., watches, smart phones, cars) to large infrastructures (e.g., cloud platforms and data centers). On top of these computing systems a plethora of software systems/applications are invading our daily lives. Because of their intrinsic distribution and the involvement of more and more parties with sometimes conflicting interests, these systems are becoming bigger and increasingly more complex and thus more subject to faults. In this manuscript we consider two types of faults : *Byzantine* faults and *rational* faults. Byzantine faults are the most generic type of faults caused by nodes (e.g., software or hardware components running in a physical machine participating in the system) that may behave arbitrarily (e.g., by crashing, being subject to a bug, being under the control of a malicious attacker). Rational faults are caused by nodes trying to maximize their own benefit without contributing their fair share to the system. Dealing with Byzantine and/or rational faults in large scale distributed systems has been and still is a very active field of research. This manuscript, which gathers a set of research works I conducted from 2010-2017 with my collaborators and PhD students, contributes to this field. It particularly presents the following contributions :

- RBFT : a robust Byzantine-fault tolerant state machine replication protocol ;
- FullReview : a robust Byzantine fault detection protocol ;
- PAG : a privacy-preserving Byzantine fault detection protocol ;
- FireSpam : a protocol for dealing with rational nodes in gossip-based spam filtering ;
- Acting : a live streaming protocol that deals with colluding rational nodes ;
- RAC : an anonymous communication protocol that deals with rational nodes ;
- Seine : a framework for designing and injecting rational faults and
- Racoon : a framework for automatically dealing with rational nodes.

I finally sketch few perspectives in this very challenging and exciting research domain.

Chapitre 1

Introduction

I obtained my PhD degree in 2007 from Université Pierre et Marie curie, on the topic of semantic middleware for pervasive computing under the supervision of Valérie Issarny and Nikolaos Georgantas. After that, I did a two years post-doc in the team of Licia Capra at University College London during which I worked on middleware for mobile social computing.
Since I joined the LIRIS laboratory in 2009 as a CNRS researcher, I have been working on dealing with faults in large scale distributed systems. My interest has particularly been focused on arbitrary faults (also called Byzantine faults) and on rational faults. *Byzantine faults* are the most generic class of faults. They are performed by nodes participating in a given system in which they may deviate from the original system design in an arbitrarily manner and at any point in time (e.g., by crashing, flooding other nodes with junk messages, behaving maliciously, etc.). On the other hand *rational faults* are faults performed by nodes which are willing to deviate from the system design only if they get a benefit in doing so (e.g., reducing the consumption of their resources, improving their perceived quality of service).
Dealing with Byzantine and rational faults have been the center of active research in the past decades. This thesis contributes to this field. Before presenting my contributions in Section 1.2, I will first present the context of today's distributed systems in Section 1.1. I will conclude this chapter by presenting other contributions not included in this thesis (Section 1.3) as well as the structure of this document (Section 1.4).

1.1 Faults in today's distributed systems

The last decades have foreseen major developments in hardware and hardware infrastructures. Desktop computers and laptops have evolved from single processor computers to multi-core and soon to be many-core computers containing 10s, 100s or even up to 1000s of independent processor cores. The decrease in the price of these powerful machines and the success of virtual environments has engendered the shift from localized grid/cluster computing to the emergence of massive cloud platforms enabling the monetized, ubiquitous access to shared computing resources. Major companies (e.g., Amazon, Microsoft, Oracle) have invested this field creating a shift to an economy of *-as-a-service. To give an idea of this shift, Amazon alone is estimated to have more than 2 million servers distributed in 44 data centres over the planet. This "Cloud Shift", which is massively followed by companies and institutions is expected to affect more than $1 Trillion in IT spending by 2020.

Besides the large becoming larger, we are also witnessing the small becoming smaller. Indeed, more and more of our daily objects are being equipped with a plethora of tiny sensors as well as powerful computing and networking capabilities. A famous study estimates that there will be more that 50 billion IoT-connected devices by 2020 engendering the "smart-* shift" (e.g., smart homes, smart offices, smart roads, smart cars, smart cities).

The evolution and the pervasiveness of computing and networking capabilities has driven the massive deployment of software systems, which are becoming more and more complex. Some of these systems require the large scale collaboration between the participating users. Popular examples of such systems

include crypto-currencies (e.g., bitcoin), live streaming systems (e.g., PPLive, BitTorrent Live), massively multiplayer online role-playing games (e.g., World of Warcraft, City of Heroes) or mobile crowdsensing applications (e.g., Waze). While some other systems are moving towards increasing autonomy from human actors such as self driving cars or surveillance drones.

While these evolutions have changed and will continue to change our societies and the way they interact with the technologies surrounding them, they have also opened the door to a wide variety of threats. Indeed, hardware infrastructures (e.g., from small connected things to data centres) may experience crashed components, corrupted hardware, transient electromagnetic interferences, delayed and corrupted messages. Furthermore, software systems running on this variety of hardware infrastructures may be subject to bugs, misconfigurations, malicious attackers, selfish users.

It is not that these issues are not known to the research community. However, what we are witnessing is that the hard issues are becoming harder to solve because of the criticality and complexity of today's hardware and software systems. In the context of this habilitation thesis, I will present a number of contributions that I worked on with my PhD students and collaborators in the period 2010-2017 and will try to step back and discuss how these research works are situated in the context described above. I will also try to sketch few perspectives in this very challenging and exciting research domain.

1.2 Contributions

1.2.1 Robust Byzantine-fault tolerant state machine replication

Byzantine fault tolerant state machine replication (BFT) [Lamport et al., 1982], [Castro and Liskov, 1999] is an established field of research in which researchers design protocols enabling the replication of a given system that can be represented as a set of deterministic state machines in presence of a limited fraction of Byzantine nodes. In particular, the major role of a BFT protocol is to order client requests in such a way that each replica executes these requests in the same order, despite a fraction of nodes acting in a Byzantine manner. In this context, the main stream contributions have focused on designing BFT protocols that perform this ordering as fast as possible thus enabling the replication of legacy systems without penalizing their performance (e.g., Zyzzyva [Kotla et al., 2009], Aliph [Guerraoui et al., 2010b], Eve [Kaptritsos et al., 2012]). An under looked question at that time was the performance (e.g., the throughput) of these protocols under Byzantine attacks. Specifically, it has been shown in [Clement et al., 2009b], that the performance of well established BFT protocols drop drastically (the throughput can drop to 0 in some cases) if Byzantine nodes perform Byzantine attacks (e.g., delaying the ordering of requests, flooding other replicas). Few protocols, that we will call robust BFT protocols in this thesis have thus been devised to alleviate this problem, among which Prime [Amir et al., 2008], Spinning [Veronese et al., 2009] and Aardvark [Clement et al., 2009b]. In this work, we stared by analysing the weaknesses of these three protocols by building Byzantine attacks that degraded their performance experimentally. We then, designed a novel robust BFT protocol, i.e., RBFT [Aublin et al., 2013] that has performance guarantees under Byzantine attacks. This work, which will be presented in Chapter 2, has been carried out in the context of the PhD thesis of Pierre-Louis Aublin and has been published in the proceedings of the 33rd IEEE International Conference on Distributed Computing Systems (IEEE ICDCS'13).

1.2.2 Robust Byzantine fault detection

Another way to deal with Byzantine faults is Byzantine fault detection. Instead of replicating a legacy system as done when using BFT protocols, the latter is monitored in order to detect if the behaviour of the nodes participating in the system diverges from an expected correct behaviour. Accountability, which refers to the ability to detect and expose node faults, is among the most effective solutions for enforcing fault detection in large scale distributed systems. In the last decade various solutions have been proposed to enforce accountability for specific applications (e.g., anonymous communication [Corrigan-Gibbs and Ford, 2010], online games [Yahyavi et al., 2013], network storage [Yumerefendi and Chase, 2007], randomised systems [Backes et al., 2009], inter domain routing [Haeberlen et al., 2009], virtualised systems [Andreas*et al.*,]). While these solutions offer strong accountability guarantees, their usability is

limited to the specific application domain for which they have been devised. Hence, generic solutions that are not tailored to a specific application have been proposed, some of which rely on trusted hardware (e.g., Trinc [Levin et al., 2009], A2M [Chun et al., 2007], Pasture [Kotla et al., 2012]) while others are generic software solutions (e.g., PeerReview [Haeberlen et al., 2007a], AVMs [Andreas*et al.*,]). Our work targets this second category of systems as they do not require users (worldwide) to acquire specific hardware. These systems generally rely on secure logs where nodes keep track of their interactions with other nodes. Each log is then periodically audited by a set of other nodes assigned by the system, i.e. the node's monitors. During an audit, the monitors verify that the monitored node did not tamper with its log and that the latter corresponds to a correct execution of the legacy system. Our first contribution in this work is to demonstrate experimentally that the monitors in these systems are not themselves subject to accountability measures, which makes the overall system vulnerable if all the monitors of a given node deviate from the monitoring procedure. We then present *FullReview* [Diarra et al., 2014] a robust accountability system, in which monitors have to stick to the monitoring protocol. This work has been carried out in the context of the PhD thesis of Amadou Diarra and has been published in the proceedings of the 33rd IEEE Symposium on Reliable Distributed Systems (IEEE SRDS'14).

1.2.3 Privacy-preserving Byzantine fault detection

As discussed above, accountability mechanisms (e.g., PeerReview, AVMs, FullReview), which require that nodes log their interactions with others and periodically inspect each others' logs are effective solutions to deter faults. However, these solutions require that nodes disclose the content of their logs to third parties, which may leak sensitive information about them. Building on a monitoring infrastructure and on homomorphic cryptographic procedures, we propose in this work *PAG* [Decouchant et al., 2016], the first accountable and privacy-preserving accountability protocol targeted to gossip-based message dissemination. This work has been carried out in the context of the PhD thesis of Jérémie Decouchant and has been published in the 36th International Conference on Distributed Computing Systems (IEEE ICDCS'16).

1.2.4 Dealing with rational nodes in gossip-based spam filtering

This work is the first of a suite of rational resilient protocols presented in this habilitation thesis. It is situated in the context of gossip protocols [Demers et al., 1987], which are an efficient and reliable way to disseminate information in decentralized distributed systems. To disseminate messages using a gossip protocol, each node in the systems periodically disseminates the recently received messages to a set of randomly selected nodes. Thanks to the randomness they rely on, to their decentralisation and to the message redundancy they engender, these protocols, have been proven to have very desirable properties including their robustness to node failures and their scalability to millions of nodes. These protocols have nevertheless a drawback : they are unable to limit the dissemination of spam messages. Indeed, messages are redundantly disseminated in the network and it is enough that a small subset of nodes forward spam messages to have them received by a majority of nodes. In this work we present *FireSpam*, a gossiping protocol that is able to limit spam dissemination. *FireSpam* organizes nodes in a ladder topology, where nodes highly capable to filter spam are at the top of the ladder, whereas nodes with a low spam filtering capability are at the bottom of the ladder. Messages are disseminated from the bottom of the ladder to its top. The ladder does thus act as a progressive spam filter. In order to make it usable in practice, we designed *FireSpam* in the BAR (Byzantine, Altruistic, Rational) failure model. This model, introduced by Alvisi et al. in [Aiyer et al., 2005] considers the presence of a limited proportion of Byzantine nodes while the remaining non-Byzantine nodes can behave either altruistically (by sticking to the original protocol specification) or rationally if they have interest to do so. Besides resisting Byzantine nodes, we demonstrate in this work that our proposed protocol is a Nash equilibrium [Nash, 1951], which means that rational nodes have no interest in deviating from it. This work has been carried out in the context of the PhD thesis of Alessio Pace and has been published in the proceedings of the 29th IEEE Symposium on Reliable Distributed Systems (IEEE SRDS'10).

1.2.5 Dealing with colluding rational nodes

This work is also situated in the context of gossip-based content dissemination protocols. Instead of focusing on spam dissemination (as in *FireSpam*) we have focused on building a content dissemination protocol that is resilient to rational nodes. Indeed, various literature studies (e.g., [Guerraoui et al., 2010a], [Li *et al.*,]) have shown that these protocols suffer from rational nodes, i.e., nodes that aim at downloading the content without contributing their fair share to the system. While the problem of rational nodes that act individually has been well addressed in the literature, *colluding* rational nodes was still an open issue. Indeed, LiFTinG [Guerraoui et al., 2010a], the only existing gossip protocol addressing this issue, yields a high ratio of false positive accusations of correct nodes. In this work, we propose AcTinG, a protocol that prevents rational collusions in gossip-based content dissemination protocols, while guaranteeing zero false positive accusations. This work has been carried out in the context of the PhD thesis of Jérémie Decouchant and has been published in the proceedings of the 33rd IEEE Symposium on Reliable Distributed Systems (IEEE SRDS'14).

1.2.6 Dealing with rational nodes in anonymous communication

This work is situated in the context of anonymous communication protocols. We showed in this work that onion routing [Goldschlag et al., 1999] the widely used anonymous communication protocol is subject to rational faults. Recent protocols have thus been proposed to deal with this issue (e.g., Dissent [Corrigan-Gibbs and Ford, 2010], [Wolinsky et al., 2012]). However, these protocols do not scale to large systems, and some of them further assume the existence of trusted servers. In this chapter, we present *RAC*, the first anonymous communication protocol that tolerates rational faults and that scales to large systems. Scalability comes from the fact that the complexity of *RAC* in terms of the number of message exchanges is independent from the number of nodes in the system. Another important aspect of *RAC* is that it does not rely on any trusted third party. This work has been carried out in the context of the PhD thesis of Gauthier Berthou and has been published in the proceedings of the 33rd IEEE International Conference on Distributed Computing Systems (IEEE ICDCS'13).

1.2.7 A framework for designing and injecting rational faults

The works presented in the previous chapters (i.e., *FireSpam*, *AcTing*, *RAC*) show various application domains on which rational behaviours may occur and present protocols to deal with these behaviours. While the proposed solutions are effective, they hardly generalise to other application domains. In the aim of finding generic solutions to deal with rational behaviours, we started the work presented in this chapter. In particular, we have been interested in studying the impact of rational behaviours on a given system's performance. Indeed, finding countermeasures to rational behaviours and testing their effectiveness requires quantifying their impact using a set of performance metrics that are of interest to the target system designer. Current techniques for understanding the impact of selfish behaviours remain manual and time-consuming [Krishnan et al., 2004]. To overcome these difficulties, we present in this chapter *SEINE*, a simulation framework for rapid modelling and evaluation of selfish behaviours in a given cooperative system. *SEINE* relies on a domain-specific language (*SEINE-L*) for specifying selfishness scenarios, and provides semi-automatic support for their implementation and study in the state-of-the-art simulator PeerSim [Jelasity et al.,]. This work has been carried out in the context of the PhD thesis of Guido Lena Cota and has been published in the proceedings of the 47th IEEE/IFIP International Conference on Dependable Systems and Networks (IEEE DSN'17).

1.2.8 Automatically dealing with rational nodes

In addition to providing a framework for assessing the impact of rational behaviours on a system's performance, we have investigated novel mechanisms for automatically injecting and configuring countermeasures in collaborative systems. To this end we present in this chapter *RACOON*, a framework that integrates accountability and reputation mechanisms into a given protocol and automatically configure these mechanisms to meet a set of performance and resilience objectives set by the system designer.

The automatic tuning of accountability and reputation mechanisms is done by combining Game Theory that is used to reason about the behaviour of rational nodes and simulations to estimate the impact of rational actions on the performance of the overall system. We illustrate the benefits of using *RACOON* by designing two cooperative systems : a P2P live streaming and an anonymous communication system. This work has been carried out in the context of the PhD thesis of Guido Lena Cota and has been published in the proceedings of 34th IEEE Symposium on Reliable Distributed Systems (IEEE SRDS'15). An extension of this work, which is not included in this manuscript on which we investigated the use of evolutionary game theory has also been published in IEEE Transactions on Dependable and Secure Computing (IEEE TDSC 2017).

1.3 Other contributions not included in this thesis

In addition to the works presented in this thesis, I have investigated three major research topics : building middleware solutions for mobile systems, enforcing private Web search and enforcing location privacy. These three topics are briefly summarized in the following three sections.

1.3.1 Middleware for mobile systems

My work in this research topic is inherited from the background I acquired during my PhD work, which I did under the supervision of Valérie Issarny and Nikolaos Georgantas at INRIA Roquencourt (2004-2007) in addition to the post-doctoral work I carried out in collaboration with Licia Capra at University College London. This work includes two major sub-topics : routing in delay tolerant networks and enforcing trust and reputation in mobile crowd sensing systems. My contributions in these two topics are described below.

Message routing in delay tolerant networks. Delay Tolerant Networks (DTNs) [Fall, 2003], also called opportunistic networks are a type of mobile networks constructed by the (intermittent) connection of co-located mobile devices. Contrary to Mobile Ad-hoc NETworks (MANETs) [Royer and Toh, 1999], in DTNs a complete routing path between two nodes that wish to communicate cannot be guaranteed. Due to this constraint, the applications developed for these networks are necessarily geo-localized with no critical time constrains (e.g., advert dissemination, recommendation of points of interest, asynchronous communication). In this context, one of the major challenges is to develop routing algorithms that are reliable (i.e., with little message loss) and efficient in terms of delivery time and battery consumption. My contributions in this topic are chronologically listed in the table below :

PhD Student	Year	Description	Reference
Afra Mashhadi (UCL)	2009	In this work we proposed a content dissemination protocol which leverages human mobility and human social links to improve content dissemination in DTNs.	[Mashhadi et al., 2009]
Jingwei Miao (INSA)	2011	In this work we proposed a DTN routing protocol that exploits individual node mobility properties (e.g., nodes centrality, nodes regularity) to efficiently route messages.	[Miao et al., 2011]
Jingwei Miao (INSA)	2012	In this work we presented a survey of works that aim at preventing selfish behaviors in DTNs.	[Miao et al., 2012]
Afra Mashhadi (UCL)	2012	In this work we presented a content dissemination protocol that faily balances the load between nodes participating in a DTN network.	[Mashhadi et al., 2012]
Jingwei Miao (INSA)	2013	In this work we presented an experimental study of the impact of selfissness in DTN routing protocols.	[Miao et al., 2013]
Jingwei Miao (INSA)	2013	In this work we presented a privacy preserving routing protocol for DTNs.	[Hasan et al., 2013], [Miao et al., 2016]
Jingwei Miao (INSA)	2015	In this work we presented a delay and cost balancing protocol for message routing in DTNs.	[Miao et al., 2015]

Enforcing trust and reputation in mobile participatory sensing. Participatory sensing is an emerging paradigm in which citizens everywhere voluntarily use their computational devices to capture and share sensed data from their surrounding environments in order to monitor and analyse some phenomenon (e.g., weather, road traffic, pollution, etc.). Interest in participatory sensing systems has risen since a large mobile sensor network can now be opportunistically constructed with much less cost and effort than it was the case a decade ago. However, relying on citizens who share their contributions raises many challenges. For instance, participants can disrupt the system by contributing corrupted, fabricated, or erroneous data. Consequently, one of the main challenges in this topic is the ability to evaluate the veracity and accuracy of participants contributions in order to build robust and reliable participatory sensing systems. My contributions in this topic are summarized in the table below :

PhD Student	Year	Description	Reference
Hayam Mousa (INSA)	2015	In this work we presented a survey of trust management and reputation systems in mobile participatory sensing applications.	[Mousa et al., 2015]
Hayam Mousa (INSA)	2017	In this work we presented a reputation system for mobile participatory sensing systems that is resilient against colluding and malicious adversaries.	[Mousa et al., 2017b]
Hayam Mousa (INSA)	2017	In this work we presented a privacy-preserving and reputation-ware mobile participatory sensing system	[Mousa et al., 2017a]

1.3.2 Enforcing private Web search

Web Search engines have become an indispensable online service enabling users to retrieve relevant content from the ever increasing amount of data populating the Internet. However, the accuracy of their recommended documents comes from the exploitation of user personal data (i.e., user past queries). Hence, using search engines raises serious privacy issues as the latter gather large amounts of data, which may contain sensitive information such as user location, interests, health issues, sexual, political or religious

preferences. Protecting users privacy while enabling them to use search engines is thus an important problem that has attracted interest in the research community in the past decade. I started working on this problem in the context of the EEXCESS European project (2013-2016). I have in particular co-supervised the PhD thesis of Albin Petit in collaboration with Antoine Boutet and Thomas Cerqueus two post-doctoral candidates funded on the same project. My contributions in this context are summarized in the table below :

PhD Student/Post-doc	Year	Title	Reference
Albin Petit (INSA-Uni. Passau)	2015	In this work we proposed a novel private Web search mechanism called PEAS, which is based on two assumed non-colluding servers that form a privacy proxy in addition to the obfuscation of user search queries.	[Petit et al., 2015]
Albin Petit (INSA-Uni. Passau)	2016	In this work we proposed SimAttack a user-reidentification attack as a novel tool for experimentally assessing the effectiveness of private Web search mechanisms.	[Petit et al., 2016]
Antoine Boutet (INSA)	2017	In this work we proposed X-Search a secure proxy that ebables clients to privately use search engines and that relies on Intel SGX enclaves.	[Ben Mokhtar et al., 2017]

1.3.3 Enforcing location privacy

The protection of location data is another important problem I have worked on in the last couple of years. Location data is nowadays gathered by a plethora of mobile applications (e.g., for finding points of interest around the user, playing geo-located games, navigating to a destination). However, this data can reveal sensitive information about the users (e.g., home and work places, health status, sexual orientation, political or religious preferences). The main challenge in this context is to propose location privacy protection mechanisms (LPPMs) that protect user mobility data while preserving data utility, i.e., the quality of the resulting data after the application of LPPMs. The work I carried on in this topic involve three PhD students : Vincent Primault (who will defend in 2018) has worked on enforcing location privacy with time distortion anonymisation ; Sophie Cerf follows a control-theoretic approach to configuring location privacy protection mechanisms so as to meet a set of privacy/utility objectives ; and Mohamed Maouche who follows an approach to protecting location data through an abstraction of user mobility as a heat-map of its past trajectories. My major contributions in topic are summarized in the table below :

PhD Student/Post-doc	Year	Description	Reference
Vincent Primault (INSA)		In this work we proposed Promesse, a novel LPPM that preserves user privacy by erasing their points of interests through the smoothing of their speed over their mobility traces.	[Primault et al., 2015]
Vincent Primault (INSA)		In this work we proposed ALP, a tool that enables finding a privacy/utility tradeoff for a given LPPM using a greedy approach.	[Primault et al., 2016]
Sophie Cerf (Uni. Grenoble)	2017	In this work we proposed PULP, a control-theoretic approach for finding a privacy/utility tradeoff for a given LPPM. Compared to ALP, PULP performs the LPPM configuration in O(1).	[Cerf et al., 2017]
Mohamed Maouche (INSA)	2017	In this work we proposed AP-Attack, a user re-identification attack that is based on the representation of user mobility as a heatmap.	[Maouche et al., 2017]

1.4 Thesis Outline

This manuscript contains only my works related to enforcing dependability in distributed systems. The manuscript is structured in three parts. The first part of this manuscript is related to dealing with arbitrary faults. It contains three chapters : robust Byzantine fault tolerant state machine replication (Chapter 2) ; robust Byzantine fault detection (Chapter 3) and privacy-preserving Byzantine fault detection (Chapter 4). Then, the second part focuses on my works related to dealing with rational behaviours. It contains five chapters : dealing with rational nodes in gossip-based spam filtering (Chapter 5) : dealing with colluding rational nodes (Chapter 6) ; dealing with rational nodes in anonymous communication protocols (Chapter 7) ; a framework for designing and injecting rational faults (Chapter 8) and a tool for automatically dealing with rational faults (Chapter 9). The last part concludes this manuscript with a presentation of the latest advances in the topics addressed in this manuscript and a discussion of open research directions (Chapter 10).

Please note that the chapters correspond to independent research articles with very little modifications. There might thus be some redundancy in definitions and related works sections.

Première partie

Dealing with Arbitrary Faults

Chapitre 2

RBFT : Redundant Byzantine Fault Tolerance

Byzantine Fault Tolerant state machine replication (BFT) protocols are replication protocols that tolerate arbitrary faults of a fraction of the replicas. Although significant efforts have been recently made, existing BFT protocols do not provide acceptable performance when faults occur. As we show in this chapter, this comes from the fact that all existing BFT protocols targeting high throughput use a special replica, called the primary, which indicates to other replicas the order in which requests should be processed. This primary can be *smartly* malicious and degrade the performance of the system without being detected by correct replicas. In this chapter, we propose a new approach, called *RBFT* for Redundant-BFT : we execute multiple instances of the same BFT protocol, each with a primary replica executing on a different machine. All the instances order the requests, but only the requests ordered by one of the instances, called the master instance, are actually executed. The performance of the different instances is closely monitored, in order to check that the master instance provides adequate performance. If that is not the case, the primary replica of the master instance is considered malicious and replaced. We implemented *RBFT* and compared its performance to that of other existing robust protocols. Our evaluation shows that *RBFT* achieves similar performance as the most robust protocols when there is no failure and that, under faults, its maximum performance degradation is about 3%, whereas it is at least equal to 78% for existing protocols. This work has been carried out in the context of the PhD thesis of Pierre-Louis Aublin and has been published in the proceedings of the 33rd IEEE International Conference on Distributed Computing Systems (IEEE ICDCS'13).

2.1 Introduction

Byzantine Fault Tolerant (BFT) state machine replication is an efficient and effective approach to deal with arbitrary software and hardware faults [Cowling et al., 2006, Abd-El-Malek et al., 2005, Wood et al., 2011, Garcia et al., 2011, Clement et al., 2009a]. The wide range of research carried out in the field of BFT in the last decade primarily focused on building *fast* BFT protocols, i.e., protocols that are designed to provide the best possible performance in the common case (i.e. in the absence of faults) [Castro and Liskov, 1999, Kotla et al., 2009, Guerraoui et al., 2010b, Kaptritsos et al., 2012]. More recently, interest has been given to *robustness*, i.e., building BFT protocols that achieve good performance when faults occur. Three protocols have been proposed to address this issue that are, Prime [Amir et al., 2008], Aardvark [Clement et al., 2009b], and Spinning [Veronese et al., 2009]. Unfortunately, as shown in Table 2.1 (details are provided in Section 8.2), these protocols are not effectively robust : the maximum performance degradation they can suffer when some faults occur is at least 78%, which is not acceptable.

The reason why the above mentioned BFT protocols are not robust is that they rely on a dedicated replica, called *primary*, to order requests. Even if there exists several mechanisms to detect and recover from a malicious primary, the primary can be *smartly* malicious. Despite efforts from other replicas to control that it behaves correctly, it can slow the performance down to the detection threshold, without

	Prime	Aardvark	Spinning
Maximum throughput *degradation*	78%	87%	99%

TABLE 2.1 – Performance degradation of “robust” BFT protocols under attack.

being caught. To design a really robust BFT protocol, a legitimate idea that comes to mind is to avoid using a primary. One such protocol has been proposed by Boran and Schiper [Borran and Schiper, 2009]. This protocol has a theoretical interest, but it has no practical interest. Indeed, the price to pay to avoid using a primary is that, before ordering every request, replicas need to be sure that they received a message from all other correct replicas. As replicas do not know which replicas are correct, they need to wait for a timeout (that is increased if it is not long enough). This yields very poor performance and this explains why this protocol has never been implemented. A number of other protocols have been devised to enforce intrusion tolerance (e.g., [Sousa et al., 2010]). These protocols rely on what is called *proactive recovery*, in which nodes are periodically rejuvenated (e.g., their cryptographic keys are changed and/or a clean version of their operating system is loaded). If performed sufficiently often, node rejuvenation makes it difficult for an attacker to corrupt enough nodes to harm the system. These solutions are complementary to the robustness mechanisms studied in this chapter.

In this chapter, we propose *RBFT* (*Redundant Byzantine Fault Tolerance*), a new approach to designing robust BFT protocols. In *RBFT*, multiple instances of a BFT protocol are executed in parallel. Each instance has a primary replica. The various primary replicas are all executed on different machines. While all protocol instances order requests, only one instance (called the *master instance*) effectively executes them. Other instances (called *backup instances*) order requests in order to compare the throughput they achieve to that achieved by the master instance. If the master instance is slower, the primary of the master instance is considered malicious and the replicas elect a new primary, at each protocol instance. Note that *RBFT* is intended for open loop systems (such as e.g., Zookeeper [Hunt et al., 2010] or Boxwood [MacCormick et al., 2004] asynchronous API), i.e., systems where a client may send multiple requests in parallel without waiting the reception of replies of anterior requests. Indeed, in a closed loop system, the rate of incoming requests would be conditioned by the rate of the master instance. Said differently, backup instances would never be faster than the master instance. RBFT further implements a fairness mechanism between clients by monitoring the latency of requests, which assures that client requests are fairly processed.

We implemented *RBFT* and compared its performance to that achieved by Prime, Aardvark, and Spinning. Our evaluation on a cluster of machines shows that *RBFT* achieves comparable performance in the fault-free case to the most robust protocols, and that it only suffers a 3% performance degradation under failures.

The rest of the chapter is organized as follows. We first present the system model in Section 6.2. We then present an analysis of state-of-the-art robust BFT protocols in Section 8.2. In Section 9.4 we present the design and principles of Redundant Byzantine Fault Tolerance, and we present in Section 2.5 an instantiation of it : the *RBFT* protocol. In Section 9.6 we present our experimental evaluation of *RBFT*. Finally, we conclude the chapter in Section 9.8.

2.2 System model

The system is composed of N nodes. We assume the Byzantine failure model, in which any finite number of faulty clients can behave arbitrarily and at most $f = \lfloor \frac{N-1}{3} \rfloor$ nodes are faulty, which is the theoretical lower bound [Lamport, 2004]. We consider the physical machine as the smallest faulty-component : if a single process is compromised, then we consider that the whole machine is compromised. Faulty nodes and clients can collude to compromise the replicated service. Nevertheless, they cannot break cryptographic techniques (e.g., signatures, message authentication codes (MACs), collision-resistant hashing). Furthermore, we assume an asynchronous network where *synchronous intervals*, during which messages are delivered within an unknown bounded delay, occur infinitely often. Finally, we denote a message m signed by node i's public key by $\langle m \rangle_{\sigma_i}$, a message m authenticated by a node i with a MAC

for a node j by $\langle m \rangle_{\mu_{i,j}}$, and a message m authenticated by a node i with a MAC authenticator, i.e., an array containing one MAC per node, by $\langle m \rangle_{\vec{\mu}_i}$. Our system model is in line with the assumptions of other papers in the field, e.g., [Castro and Liskov, 1999].

We address in this chapter the problem of robust Byzantine Fault Tolerant state machine replication in open-loop systems, i.e., systems in which clients do not need to wait for the reply of a request before sending new requests [Schroeder et al., 2006]. In an open loop system, even if the malicious primary of the master instance delays requests, correct primaries of the backup instances will still be able to order new requests coming from clients and thus to detect the misbehaving primary. This is not the case in closed-loop systems. We will consider the robustness of BFT protocols intended for closed-loop systems in our future work.

2.3 Analysis of existing robust BFT protocols

We present in this section an analysis of existing robust BFT protocols i.e., Prime [Amir et al., 2008], Spinning [Veronese et al., 2009] and Aardvark [Clement et al., 2009b]. These three protocols are the only protocols designed to target the robustness problem. Indeed, an analysis of other famous BFT protocols, e.g., PBFT [Castro and Liskov, 1999], QU [Abd-El-Malek et al., 2005], HQ [Cowling et al., 2006] and Zyzzyva [Kotla et al., 2009], performed in [Clement et al., 2009b], has shown that these protocols suffer from a robustness issue. Specifically, although they are build to eventually recover from attacks, the throughput of all of them drops to zero during a possibly long time interval corresponding to the duration of the attack, which is not acceptable for their clients. In all these protocols, the system is composed of $N = 3f + 1$ replicas, among which one has the role for proposing sequence number to requests, i.e., the primary.

2.3.1 Prime

In Prime [Amir et al., 2008], clients send their requests to any replica in the system. Replicas periodically exchange the requests they receive from clients. As such, they are aware of current requests to order and start expecting ordering messages from the primary which should contain them. Furthermore, whether there are requests to order or not, the primary must periodically send (possibly empty) ordering messages. This allows non-primary replicas to expect ordering messages with a given frequency. In order to improve the accuracy of the expected frequency at which a primary should send messages, replicas monitor the network performance. Specifically, replicas periodically measure the round-trip time between each pair of them. This measure allows them to compute the maximum delay that should separate the sending of two ordering messages performed by a correct primary. This delay is computed as a function of three parameters : the round-trip time between replicas, the time needed to execute a batch of requests, and a constant that accounts for the variability of the network latency, which is set by the developer. If the primary becomes slower than what is expected by the replicas, then it is replaced.

The Prime protocol is not robust for the following reason. If the monitoring is inaccurate, the delay expected for a primary to send ordering messages can be too long, which gives the opportunity for a malicious primary to delay ordering messages. We performed the following experiment (in order to increase the round-trip time) : a malicious primary colludes with a single faulty client. The latter sends a request that is heavier to process than other requests (1ms vs 0.1ms in our experiments). This increases the monitored round-trip time, and this gives the opportunity for the malicious primary to delay requests issued by correct clients. Figure 2.1 presents the throughput under attack relative to the throughput in the fault-free case, in percentage, as a function of the requests size, for both a static and a dynamic load (details on the two workloads are given in Section 9.6). We observe that the primary is able to degrade the system throughput down to 22% of the performance in the fault-free case. In other words, the throughput under attack drops by up to 78%.

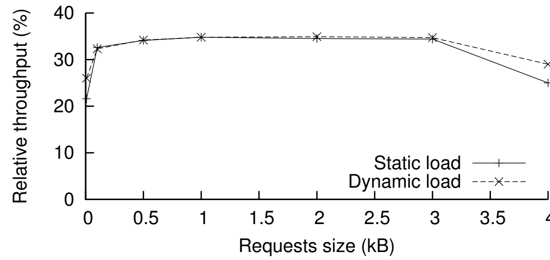


FIGURE 2.1 – Prime throughput under attack relative to the throughput in the fault-free case.

2.3.2 Aardvark

Aardvark [Clement et al., 2009b] is a BFT protocol based on PBFT [Castro and Liskov, 1999], the practical BFT protocol presented by Castro and Liskov. An important principle in the robustness of Aardvark is the presence of regular changes of the primary replica. Each time the primary is changed, a new configuration, called *view*, is started. The authors of Aardvark argue that regularly changing the primary allows limiting the throughput degradation a malicious primary may cause. This regular primary changes are performed as follows. A primary replica is required to achieve at the beginning of a view a throughput at least equal to 90% of the maximum throughput achieved by the primary replicas of the last N views (where N is the number of replicas). After an initial grace period of 5 seconds where the required throughput is stable, the non-primary replicas periodically raise this required throughput by a factor of 0.01, until the primary replica fails to provide it. At that point, a primary change occurs and a new replica becomes the primary. In addition to expecting a minimal throughput from the primary, the replicas monitor the frequency at which the primary sends ordering messages. Specifically, replicas start a timer, called heartbeat timer, after the reception of each ordering message from the primary. If this timer expires before the primary sends another ordering message, a primary change is voted by replicas. In addition to regular view changes and heartbeat timers, Aardvark implements a number of robustness mechanisms to deal with malicious clients and replicas. For instance, it uses separate Network Interface Controllers (NICs) for clients and replicas. This avoids client traffic to slow down the replica-to-replica communication. Further, this enables the isolation of replicas that would flood the network with unfaithful messages.

As long as the system is saturated, the amount of damage a faulty primary can do on the system is limited, as the throughput expected by replicas is close to the maximal throughput clients can sustain. We performed an experiment in which the primary tries to delay requests as much as it can, under a static load. Results, depicted in Figure 2.2, show that the throughput provided by the system under attack is at least 76% of the throughput observed in the fault-free case. When the load is dynamic, however, the performance degradation can potentially be much higher. Indeed, when the load is low, the expected throughput computed by replicas is also low. If the load suddenly increases, a malicious primary can benefit from the low expectations computed by replicas to delay requests. We performed different experiments under the dynamic load described in Section 9.6, for different message sizes. Results, depicted in Figure 2.2, show that because of a malicious primary under a dynamic load, the throughput of the system can drop down to 13% of the throughput that would have been provided if the primary would have been detected (the maximum throughput degradation being 87%).

2.3.3 Spinning

Spinning [Veronese et al., 2009] is a BFT protocol also based on PBFT [Castro and Liskov, 1999]. Similarly to Aardvark, Spinning performs regular primary changes. The particularity of Spinning is that these primary changes are automatically performed after the primary has ordered a single batch of requests. In this protocol, requests are sent by clients to all replicas. As soon as a non-primary replica receives a request, it starts a timer and waits for a request ordering message from the primary containing this request. If the timer expires, after a duration $S_{timeout}$, then the current primary is blacklisted (i.e.,

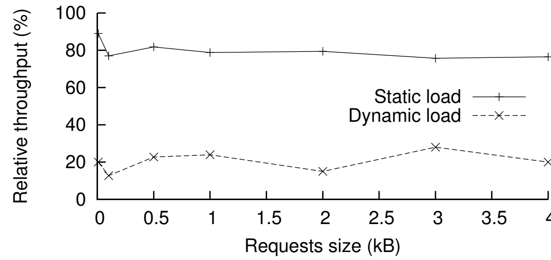


FIGURE 2.2 – Aardvark throughput under attack relative to the throughput in the fault-free case.

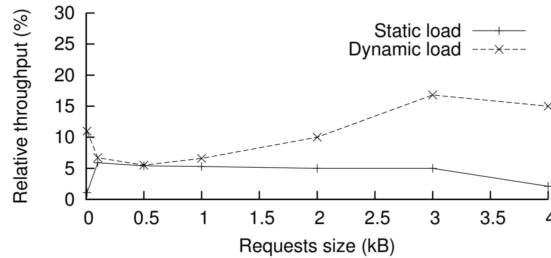


FIGURE 2.3 – Spinning throughput under attack relative to the throughput in the fault-free case.

it will no longer become a primary in the future [1]), another replica becomes the primary, and $S_{timeout}$ is doubled. As soon as a request has been successfully ordered, the primary is automatically changed (i.e., there is no message exchange between replicas) and the value of $S_{timeout}$ is reset to its initial value. Note that the value of $S_{timeout}$ is a system parameter statically defined ; it does not depend on live monitoring of the system.

The Spinning protocol is not robust for the following reason. A malicious primary can delay the request ordering messages by a little less than $S_{timeout}$. It will drastically reduce the throughput, without being detected. As a result, it can continue to delay future ordering messages, the next time it becomes the primary. We ran several experiments where the malicious primary was delaying the sending of the ordering messages by 40ms (which is the value used by the authors of Spinning in [Veronese et al., 2009]), for both under a static and a dynamic workload. Results, depicted in Figure 2.3, show that the throughput drops dramatically down to 1% and 4.5% of the fault-free throughput, under the static and dynamic workloads, respectively. This throughput degradation of up to 99% is clearly not acceptable.

2.3.4 Summary

In this section we have seen that so-called robust BFT protocols, i.e., Prime, Aardvark and Spinning, are not effectively robust. A primary node can be *smartly* malicious and cause huge performance degradation without being caught. Prime is robust as long as the network meets a certain level of synchrony. If the variance of the network is too high, then a malicious primary can heavily impact the performance of the system. Aardvark is robust as long as the load is static. If Aardvark is fed up with a dynamic load (for instance, a load corresponding to connections to a website, which may contain many spikes), then it is not guaranteeing good performance anymore. Spinning is robust only $2f + 1$ requests over $3f + 1$, when the current primary is a correct replica. However, Spinning has an important weakness every time a malicious primary is in place : the malicious primary can delay the sending of the ordering messages up to the maximal allowed time.

1. If f replicas are already blacklisted, then the oldest one is removed from the blacklist, to ensure the liveness of the system.

2.4 The *RBFT* protocol

We have seen in the previous section that existing BFT protocols that claim to be robust are not actually robust. In this section, we present *RBFT*, a new BFT protocol stems from the concepts of Redundant Byzantine Fault Tolerance : multiples instances of a BFT protocol are executed simultaneously, each one with a primary replica running on a different machine. We start by an overview of *RBFT*. We then detail the various steps followed by replicas. Finally, we detail two key mechanisms that are used in *RBFT* : the monitoring mechanism and the protocol instance change mechanism.

2.4.1 Protocol overview

As for the other robust BFT protocols, *RBFT* requires $3f+1$ nodes (i.e., $3f+1$ physical machines). Each node runs $f+1$ protocol instances of a BFT protocol in parallel (see Figure 2.4). As we theoretically show in the companion technical report [Aublin et al., 2013], $f+1$ protocol instances is necessary and sufficient to detect a faulty primary and ensure the robustness of the protocol. This means that each of the N nodes in the system runs locally one replica for each protocol instance. Note that the different instances order the requests following a 3-phase commit protocol similar to PBFT [Castro and Liskov, 1999]. Primary replicas of the various instances are placed on nodes in such a way that, at any time, there is at most one primary replica per node. One of the $f+1$ protocol instances is called the *master instance*, while the others are called the *backup instances*. All instances order client requests, but only the requests ordered by the master instance are executed by the nodes. Backup instances only order requests in order to be able to monitor the master instance. For that purpose, each node runs a monitoring module that computes the throughput of the $f+1$ protocol instances. If $2f+1$ nodes observe that the ratio between the performance of the master instance and the best backup instance is lower than a given threshold, then the primary of the master instance is considered to be malicious, and a new one is elected. Intuitively, this means that a majority of correct nodes agree on the fact that a protocol instance change is necessary (the full correctness proof of *RBFT* can be found in the companion technical report [Aublin et al., 2013]). An alternative could be to change the master instance to the instance which provides the highest throughput. This would require a mechanism to synchronize the state of the different instances when switching, similar to the switching mechanism of Abstract [Guerraoui et al., 2010b]. We will explore this design in our future work.

In *RBFT*, the high level goal is the same as for the other robust BFT protocols we have studied previously : replicas monitor the throughput of the primary and trigger the recovery mechanism when the primary is slow. The approach we use is radically different. It is not possible for replicas to guess what the throughput of a *non*-malicious primary would be. Therefore, the key idea is to leverage multicore architectures to run multiple instances of the same protocol in parallel. Nodes have to compare the throughput achieved by the different instances to know whether a protocol instance change is required or not. We are confident to say (given the theoretical [Aublin et al., 2013] and experimental analysis) that this approach allows us to build an effectively robust BFT protocol.

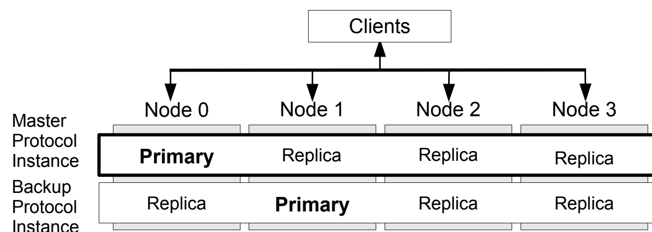


FIGURE 2.4 – *RBFT* overview ($f = 1$).

For *RBFT* to correctly work it is important that the $f+1$ instances receive the same client requests. To that purpose when a node receives a client request, it does not give it directly to the $f+1$ replicas it is hosting. Rather, it forwards the request to all other nodes. When a node has received $2f+1$ copies of a

client request (possibly including its own copy), it knows that every correct node will eventually receive the request (because the request has been sent to at least one correct node). Consequently, it gives the request to the $f+1$ replicas it hosts.

Finally, note that each protocol instance implements a full-fledged BFT protocol, very similar to the Aardvark protocol described in the previous section. There is nevertheless a significant difference : a protocol instance does not proceed to a view change by its own. Indeed, the view changes in *RBFT* are controlled by the monitoring mechanism and apply on every protocol instance at the same time.

2.4.2 Detailed protocol steps

RBFT protocol steps are described hereafter and depicted in Figure 2.5. The numbering of the steps is the same as the one used in the figure.

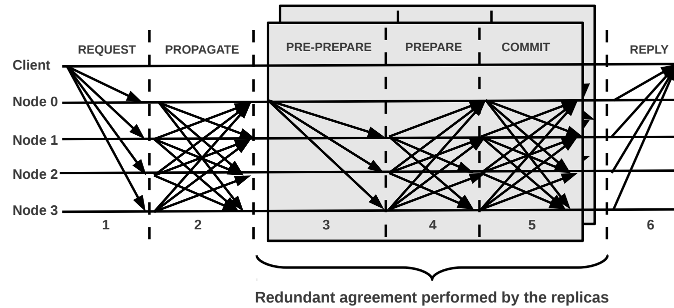


FIGURE 2.5 – *RBFT* protocol steps ($f = 1$).

1. The client sends a request to all the nodes. A client c sends a REQUEST message $\langle\langle \text{REQUEST}, o, rid, c\rangle_{\sigma_c}, c\rangle_{\vec{\mu}_c}$ to all the nodes (Step 1 in the figure). This message contains the requested operation o, a request identifier rid, and the client id c. It is signed with c's private key, and then authenticated with a MAC authenticator for all nodes. On reception of a REQUEST message, a node i verifies the MAC authenticator. If the MAC is valid, it verifies the signature of the request. If the signature is invalid, then the client is blacklisted : further requests will not be processed [2]. If the request has already been executed, i resends the reply to the client. Otherwise, it moves to the following step. We use signatures as the request needs to be forwarded by nodes to each other and using a MAC authenticator alone would fail in guaranteeing non-repudiation, as detailed in [Clement et al., 2009b]. Similarly, using signatures alone would open the possibility for malicious clients to overload nodes by sending unfaithful requests with wrong signatures, that are more costly to verify than MACs.

2. The correct nodes propagate the request to all the nodes. Once the request has been verified, the node sends a $\langle \text{PROPAGATE}, \langle \text{REQUEST}, o, s, c\rangle_{\sigma_c}, i\rangle_{\vec{\mu}_i}$ message to all nodes. This step ensures that every correct node will eventually receive the request as long as the request has been sent to at least one correct node. On reception of a PROPAGATE message coming from node j, node i first verifies the MAC authenticator. If the MAC is valid, and it is the first time i receives this request, i verifies the signature of the request. If the signature is valid, i sends a PROPAGATE message to the other nodes. When a node receives $f+1$ PROPAGATE messages for a given request, the request is ready to be given to the replicas of the $f+1$ protocol instances running locally, for ordering. As proved in the technical report [Aublin et al., 2013], only $f+1$ PROPAGATE messages are sufficient to guarantee that if malicious primaries can order a given request, all correct primaries will eventually be able to order the same request. Note that the replicas do not order the whole request but only its identifiers (i.e., the client id, request id and digest). Not only the whole request is not necessary for the ordering phase, but it also improves the performance as there is less data to process.

2. Deviations performed by clients or nodes on the security protocols, e.g., a client that would continuously initiate the key exchange protocol, are considered out of the scope of the work and will be studied as future work

3. 4. and 5. The replicas of each protocol instance execute a three phase commit protocol to order the request. When the primary replica p of a protocol instance receives a request, it sends a PRE-PREPARE message $\langle \text{PRE-PREPARE}, v, n, c,$
$rid, d\rangle_{\vec{\mu}_p}$ authenticated with a MAC authenticator for every replica of its protocol instance (Step 3 in the figure). A replica that is not the primary of its protocol instance stores the message and expects a corresponding PRE-PREPARE message. When a replica receives a PRE-PREPARE message from the primary of its protocol instance, it verifies the validity of the MAC. It then replies to the PRE-PREPARE message by sending a PREPARE message to all other replicas, only if the node it is running on already received $f+1$ copies of the request. Without this verification, a malicious primary may collude with faulty clients that would send correct requests only to him, in order to boost the performance of the protocol instance of the malicious primary at the expense of the other protocol instances. Following the reception of $2f$ matching PREPARE messages from distinct replicas of the same protocol instance that are consistent with a PRE-PREPARE message, a replica r sends a commit message $\langle \text{COMMIT}, v, n, d, r\rangle_{\vec{\mu}_r}$ that is authenticated with a MAC authenticator (Step 5 in the figure). After the reception of $2f+1$ matching COMMIT messages from distinct replicas of the same protocol instance, a replica gives back the ordered request to the node it is running on.

6. The nodes execute the request and send a reply message to the client. Each time a node receives an ordered request from a replica of the master instance, the request operation is executed. After the operation has been executed, the node i sends a REPLY message $\langle \text{REPLY}, u, i\rangle_{\mu_{i,c}}$ to client c that is authenticated with a MAC, where u is the result of the request execution (Step 6 in the figure). When the client c receives $f+1$ valid and matching $\langle \text{REPLY}, u, i\rangle_{\mu_{i,c}}$ from different nodes i, it accepts u as the result of the execution of the request.

2.4.3 Monitoring mechanism

RBFT implements a monitoring mechanism to detect whether the master protocol instance is faulty or not. This monitoring mechanism works as follows. Each node keeps a counter $nbreqs_i$ for each protocol instance i, which corresponds to the number of requests that have been ordered by the replica of the corresponding instance (i.e. for which $2f+1$ COMMIT messages have been collected). Periodically, the node uses these counters to compute the throughput of each protocol instance replica and then resets the counters. The throughput values are compared as follows. If the ratio between the throughput of the master instance t_{master} and the average throughput of the backup instances t_{backup} is lower than a given threshold Δ, then the primary of the master protocol instance is suspected to be malicious, and the node initiates a protocol instance change, as detailed in the next section. The value of Δ depends on the ratio between the throughput observed in the fault-free case and the throughput observed under attack. Note that the state of the different protocol instances is not synchronized : they can diverge and order requests in different orders without compromising the safety nor the liveness of the protocol.

In addition to the monitoring of the throughput, the monitoring mechanism also tracks the time needed by the replicas to order the requests. This mechanism ensures that the primary of the master protocol instance is fair towards all the clients. Specifically, each node measures the individual latency of each request, say lat_{req} and the average latency for each client, say lat_c, for each replica running on the same node. A configuration parameter, Λ, defines the maximal acceptable latency for any given request. Another parameter, Ω, defines the maximal acceptable difference between the average latency of a client on the different protocol instances. These two parameters depend on the workload and on the experimental settings. When the node sends a request to the various replicas running locally for ordering, it records the current time. Then, when it receives the corresponding ordered request, it computes its latency lat_{req} and the average latency for all the requests of this client lat_c. If this request has been ordered by the replica of the master instance and if lat_{req} is greater than Λ, or the difference between lat_c and the average latency for this client on the other protocol instances is greater than Ω, then the node starts a protocol instance change. Note that we define the value of the different parameters, Δ, Λ and Ω both theoretically and experimentally, as described in the companion technical report [Aublin et al., 2013] : their value depends on the cost of the cryptographic operations and on the network conditions.

2.4.4 Protocol instance change mechanism

In this section, we describe the protocol instance change mechanism that is used to replace the faulty primary at the master protocol instance. Because there is only at most one primary per node in *RBFT*, this also implies to replace all the primaries on all the protocol instances.

Each node i keeps a counter cpi_i, which uniquely identifies a protocol instance change message. When a node i detects too much difference between the performance of the master instance and the performance of the backup instances (as detailed in the previous section), it sends an $\langle \text{INSTANCE_CHANGE}, cpi_i, i\rangle_{\vec{\mu}_i}$ message authenticated with a MAC authenticator to all the other nodes.

When a node j receives an INSTANCE_CHANGE message from node i, it verifies the MAC and handles it as follows. If $cpi_i < cpi_j$, then this message was intended for a previous INSTANCE_CHANGE and is discarded. On the contrary, if $cpi_i \geq cpi_j$, then the node checks if it should also send an INSTANCE_CHANGE message. It does so only if it also observes too much difference between the performance of the replicas. Upon the reception of $2f + 1$ valid and matching INSTANCE_CHANGE messages, the node increments cpi and initiates a view change on every protocol instance that runs locally. As a result, each protocol instance elects a new primary and the malicious replica of the master instance is no longer the primary.

2.5 Implementation

We have implemented *RBFT* in C++ using the Aardvark code base. Similarly to Aardvark, *RBFT* adopts separate Network Interface Controllers (NICs). Not only this allows to avoid client traffic to slow down node-to-node communication, but it also protects against a flooding attack performed by faulty nodes. In this situation, *RBFT* closes the NIC of the faulty node for a given time period, which gives time to the faulty node to restart or get repaired without penalizing the performance of the whole system. Moreover, as our implementation of Aardvark, *RBFT* uses TCP. We use it because it eases the development task : on the contrary of UDP, TCP provides a loss-less, FIFO communication channel. Previous works (e.g., Zyzzyva [Kotla et al., 2009] or PBFT [Castro and Liskov, 1999]) have shown that the bottleneck in BFT protocols is actually cryptography, not network usage. Interestingly, the BFT protocol that currently provides the highest throughput uses TCP [Guerraoui et al., 2010b]. This protocol, called Chain, implies the smallest number of cryptographic operations at the bottleneck replicas, hence its very good performance. For comparison we also have implemented a UDP version of *RBFT*. We show, in Section 2.6.2, that this implementation provides the same performance.

Figure 2.6 depicts the architecture of a single node in *RBFT* in the case when one fault is tolerated (i.e. $f = 1$). We observe that two replicas belonging to two different protocol instances are hosted by the node : replica 0 from protocol instance 0 (noted $p_{0,0}$ in the figure) and replica 0 from protocol instance 1 (noted $p_{0,1}$ in the figure). We also observe that the node uses $3f + 1 = 4$ NICs, as Aardvark : 1 NIC for the communication with the clients, and 1 NIC per other node.

As depicted in the figure, a number of modules run on each node. We describe the behavior of these modules below.

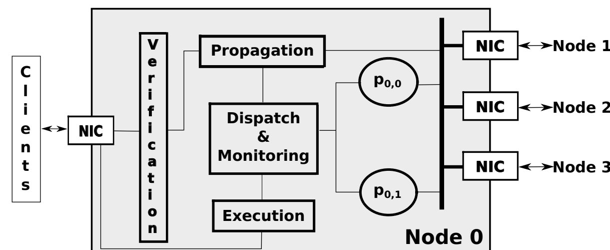
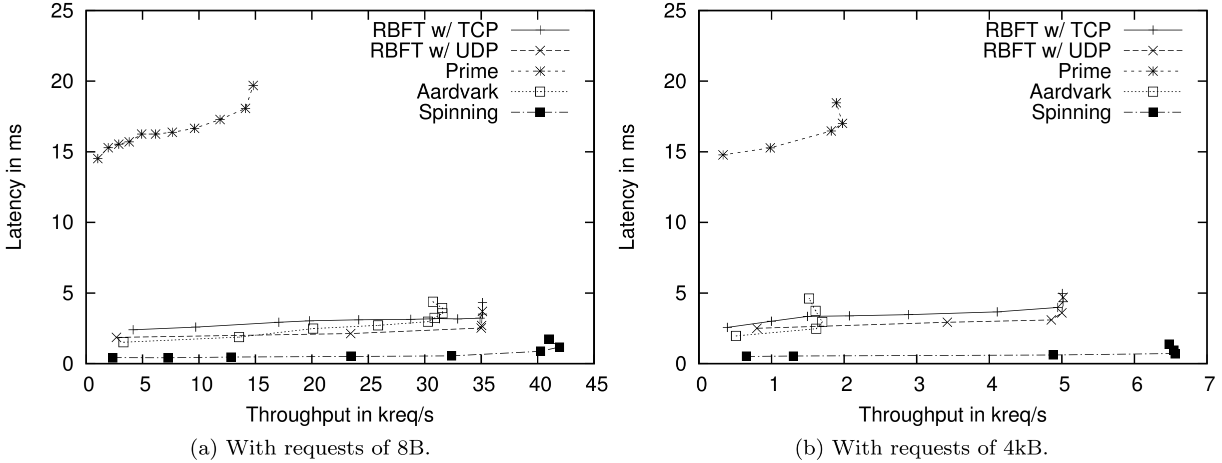


FIGURE 2.6 – Architecture of a single node ($f = 1$). The replicas (in circles) are processes that are independent from the different modules, implemented as threads (in rectangles).



(a) With requests of 8B.

(b) With requests of 4kB.

FIGURE 2.7 – Latency vs. throughput for various robust BFT protocols ($f = 1$).

When a request is received from a client, it is verified by the *Verification* module. If the request is correct then the *Propagation* module disseminates it to all the other nodes through the node-to-node NICs (by balancing the load) and waits for similar messages from them. Once it has received $f + 1$ such messages, the *Propagation* module sends the request to the *Dispatch & Monitoring* module, which logs the request and gives it to the replicas of the protocol instances running locally (i.e., $p_{0,0}$ and $p_{0,1}$ in the figure). The replicas interact with other replicas of the same protocol instance (running on the other nodes) via separate NICs, to order the request. Then, each replica gives back the ordered request to the *Dispatch & Monitoring* module. The latter sends ordered requests coming from the master protocol instance to the *Execution* module, which executes them and replies to the client.

The *Verification*, *Propagation*, *Dispatch & Monitoring* and *Execution* modules are implemented as separate threads. Moreover, the replicas are implemented as independent processes. The various threads and processes are deployed on distinct cores (our machines have 8 cores). Leveraging multicore machines improves the performance of the system, as the different protocol instances can really execute concurrently.

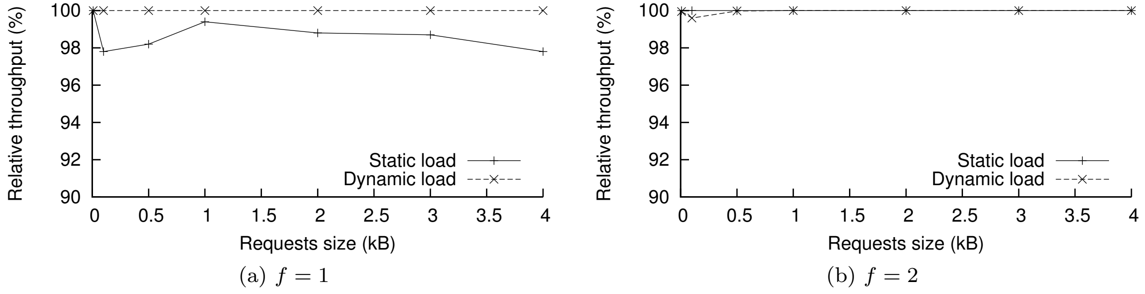
2.6 Performance evaluation

In this section we present a performance analysis of *RBFT*. After describing the hardware and software settings we use, we start by comparing the performance of *RBFT* against the state-of-the-art robust BFT protocols described in Section 8.2 in the fault-free case. Finally, we present a performance analysis of *RBFT* under attack.

Our evaluation makes the following points. We first show that *RBFT* has comparable performance to state-of-the art protocols in the fault-free case. Second, we show that under the two worst possible attacks, where faulty clients collude with faulty replicas, the throughput degradation caused to *RBFT* is limited to 3%.

2.6.1 Experimental settings

We evaluate the performance of Prime, Aardvark, Spinning and *RBFT* on a cluster composed of eight Dell PowerEdge T610 and two Dell Precision WorkStation T7400. The T610 host two quad-core Intel Xeon E5620 processors clocked at 2.40GHz with 16GB of RAM and ten network interfaces. The T7400 host two quad-core Intel Xeon E5410 processors clocked at 2.33GHz with 8GB of RAM and five network interfaces. All these machines run a Linux kernel version 2.6.32 and are interconnected via a Gigabit switch. We run experiments with up to two Byzantine faults, i.e., $f \leq 2$. The nodes are always launched on the T610 machines. The remaining machines are used to run the clients. Unless specified, we consider



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

FIGURE 2.8 – *RBFT* throughput under *worst-attack-1* relative to the throughput in the fault-free case, for both a static and a dynamic load.

the TCP implementation of *RBFT* configured with $f = 1$.

We run our experiments under two different workloads : a static load, where the system is saturated and the clients send their requests at a constant rate, and a dynamic workload, where the incoming throughput varies. We present the workload used for requests of 8B. Similar workloads have been used for the other requests sizes with possibly fewer clients as the peak throughput has been reached with fewer clients. The experiment starts with a single client. We then progressively increase the number of clients up to 10. Then we simulate a load spike, with 50 clients. At last, the number of clients progressively decreases, until there is only one client issuing requests. When the load is dynamic, we consider the average throughput observed on the whole experiment. A similar workload has already been used in [Guerraoui et al., 2010b].

Finally, the clients send their requests in an open-loop as defined in [Schroeder et al., 2006], i.e., they do not wait for the reply of a request before sending a new one.

2.6.2 Fault-free case

In this section we present the fault-free performance of *RBFT*, Prime, Aardvark and Spinning. We also show the performance of *RBFT* when the communication protocol used between the nodes is UDP instead of TCP. We run the experiments under a static load, with requests of either 8B or 4kB. Figures 2.7a and 2.7b present the latency achieved by the different protocols as a function of the throughput, for requests of 8B and 4kB, respectively.

Fist of all, we observe that Spinning provides the highest peak throughput and the lowest latency. Specifically, with requests of 8B, its throughput is 20% higher than the throughput of *RBFT* and Aardvark, and 183% higher than the throughput of Prime. Similarly, with requests of 4kB the throughput of Spinning is 30% higher than the throughput of *RBFT*. Its good performance is mostly due to the fact that it relies only on MACs, while the other protocols (*RBFT*, Prime and Aardvark) use signatures in addition to MACs. Although signatures are an order of magnitude more costly than MACs, they are necessary to prevent certain attacks [Clement et al., 2009b]. Moreover, Spinning uses UDP multicast for both the communication between the replicas and the communication between the clients and the replicas. These two properties allow Spinning to provide a very low latency.

The second observation we make is that the performance of *RBFT* is higher than the performance of Aardvark. For requests of 8B, the peak throughput of *RBFT* is 35 kreq/s, while the peak throughput of Aardvark is 31.6 kreq/s. Similarly, for requests of 4kB, the peak throughput of *RBFT* is 5 kreq/s, while the peak throughput of Aardvark is 1.7 kreq/s. This might seem surprising as the BFT protocol that is run by each protocol instance in *RBFT* mimics Aardvark and uses the same code base. The reason why *RBFT* is more efficient is that it does not perform regular view changes (remember that view changes are replaced by protocol instance changes that occur only when there are faults). To confirm this explanation, we disabled the view changes in Aardvark and we obtained the same performance as *RBFT* for small requests. For bigger requests, the better performance of *RBFT* is due to the fact that the protocol instances only order requests identifiers instead of the whole request. When they order the

whole requests, the peak throughput drops to 1.8 kreq/s for requests of 4kB.

Third, we observe that Prime provides the worst performance, especially in terms of latency. Indeed, its latency is an order of magnitude higher than the latency of the other protocols for both request sizes. This high latency is due to the fact that the Prime protocol solely relies on signatures, which are known to be slower than MACs. Moreover, it is due to the fact that in Prime, the primary does not send request ordering messages following the flow of arrival of requests, but periodically.

Finally, we observe that the UDP and TCP implementations of *RBFT* exhibit the same peak throughput. The only difference is that the UDP implementation provides a latency 22% lower (resp. 18%) than the TCP implementation, for requests of 8B (resp. 4kB). This increase in latency is due to the mechanisms TCP uses to enforce robustness (acknowledgements, flow control, etc.). Note that the choice of the communication mechanism has no impact on the performance of *RBFT* under attack, as we found similar results while using TCP or UDP.

2.6.3 *RBFT* under attack

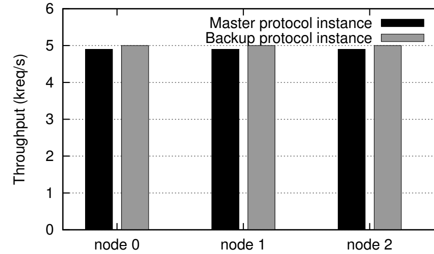


FIGURE 2.9 – Throughput measured by the different nodes under worst attack 1 ($f = 1$, static workload, 4kB requests).

In this section, we first present the two worst attacks that can be done to *RBFT*. These attacks show the maximal damage an attacker can make on the protocol in two different situations : (1) when the primary of the master instance is correct (*worst-attack-1*), and (2) when the primary of the master instance is malicious (*worst-attack-2*). We consider these attacks as worst attacks in their respective context, because in each case, all the f malicious nodes and any number of malicious clients collude to reduce the system performance. We run the experiments in two configurations : $f = 1$ and $f = 2$. We show that the maximal throughput degradation is 3%. Finally, we show that the monitoring of the latency prevents a faulty primary at the master instance not to be fair with a subset of the clients when constructing an ordering message.

Worst-attack-1

In this attack, there are f faulty nodes and all clients are faulty. The primary of the master protocol instance is correct (i.e. it runs on a correct node). The goal of the attack is to decrease as much as possible the performance of the master instance, without inducing a protocol instance change. A faulty node can cause the following damages : first, it can flood other nodes ; second, the replicas it hosts are also faulty and they can take actions to reduce the throughput of the protocol. Let p be the node on which the primary of the master protocol instance runs. The attack we are performing is the following : (i) the clients (that are all faulty) send requests that can be verified by all nodes, but the one on which the primary of the master protocol instance runs ; (ii) the f faulty nodes flood p with invalid PROPAGATE messages of the maximal size ; (iii) the faulty replicas of the master protocol instance flood the correct ones with invalid messages of the maximal size ; (iv) the faulty replicas of the master protocol instance do not take part in the protocol.

We run this experiment with a request size ranging from 8B to 4kB. Figure 2.8 presents the throughput under attack relative to the throughput in the fault-free case, under both the static and dynamic load

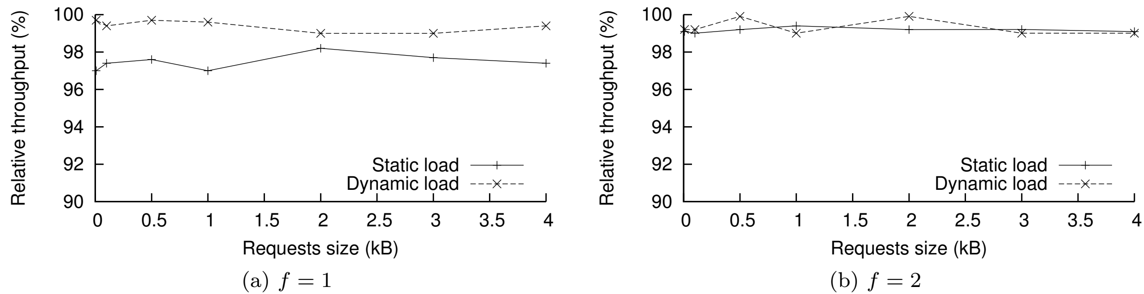


FIGURE 2.10 – *RBFT* throughput under *worst-attack-2* relative to the throughput in the fault-free case, for both a static and a dynamic load.

described earlier. We observe that *RBFT* is robust : the throughput loss is below 2.2%, under a static load and that it is null with the dynamic load, when $f = 1$. When at most two faults are tolerated, i.e., $f = 2$, we observe that the throughput loss is even lower : at most 0.4%.

In order to illustrate the behavior of *RBFT*, we registered the throughput measures performed by the monitoring mechanism of the different nodes. Results are depicted in Figure 2.9 for the static workload, with a request size of 4kB, and when at most one fault is tolerated (we observed similar results in other configurations). This figure first shows that each node measures the same throughput. Moreover, it shows that the throughput measured for the master protocol instance is very close to the throughput measured for the backup protocol instance (2% difference). Note that we do not represent the values reported by the monitoring module of node 3 as this is the faulty node in this experiment (it can thus report arbitrary values).

Worst-attack-2

In this attack, there are f faulty nodes and all clients are faulty. The primary of the master protocol instance is faulty (i.e. it runs on a faulty node). Faulty clients and nodes aim at decreasing the performance of the backup protocol instances in order to give a margin for the faulty primary of the master protocol instance to delay requests without being detected. Towards this purpose, the faulty nodes collude with the faulty clients, and take the following actions : (i) the (faulty) clients send invalid requests to the correct nodes ; (ii) the f faulty nodes flood the correct nodes with invalid messages of the maximal size and do not participate in the PROPAGATE phase ; (iii) the replicas of the backup protocol instances executing on the f faulty nodes flood the correct ones with invalid messages of the maximal size, and do not take part in the protocol.

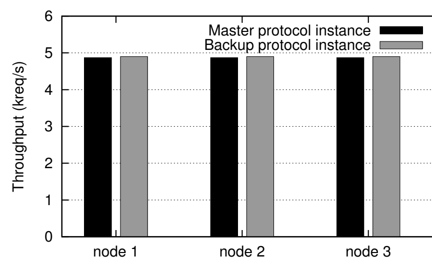


FIGURE 2.11 – Throughput measured by the different nodes under worst attack 2 ($f = 1$, static workload, 4kB requests).

We run this experiment with a request size ranging from 8B to 4kB. Figure 2.10 presents the throughput under attack relative to the throughput in the fault-free case, for both the static and dynamic load described earlier. The malicious primary of the master instance decreases its throughput by delaying the

requests, down to the limit value such that the throughput ratio observed at the correct nodes is be greater or equal than Δ. Indeed, a lower ratio observed at the correct nodes implies a protocol instance change. We again observe that *RBFT* is robust : the maximum throughput loss is below 3% when $f = 1$. It is even below when $f = 2$: less than 1%.

As for the first worst attack, we present in Figure 2.11 the throughput measures that are performed by the monitoring mechanism running on the different nodes. Similarly to the previous attack, we observe that all nodes measure the same throughput. Moreover, we observe that the throughput measured for the master protocol instance is almost similar to the one measured for the backup instance. This explains why *RBFT* is robust.

Unfair primary

In this attack, the primary of the master instance is not fair and proposes an ordering on the requests of a given client less frequently than for the other clients. We show that the malicious primary cannot increase the latency of the requests of a single client beyond a small limit. We run an experiment with $f = 1$, 2 clients and requests of 4kB. The maximal acceptable latency, Λ, is 1.5ms. We also set a high value for Ω, the maximal acceptable difference between the average latency of a client on the different protocol instances. Consequently, we easily observe that the primary of the master instance cannot increase the latency of a client beyond Λ.

Figure 2.12 presents the latency of each request for both clients. At the beginning and for 500 requests, the malicious primary, of the master protocol instance, acts normally. The average latency is 0.8ms. Then it proposes an ordering for the requests of the first client less frequently, so that the average latency observed for this client is 1.3ms, during 500 more requests. At request 1000, it increases even more the latency observed for this client. This single request has a latency of 1.6ms. As it is higher than the maximal acceptable latency, the different nodes vote a Protocol Instance Change. As a result, the malicious primary of the master instance is evicted and replaced by a correct replica. This correct replica is fair and provides the same latency for the requests of both clients. Note that in this experiment the throughput monitoring mechanism did not triggered a protocol instance change because the malicious primary of the master instance provided a similar throughput than the throughput of the backup protocol instances.

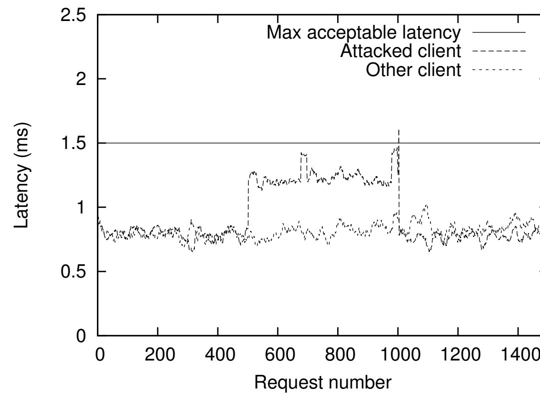


FIGURE 2.12 – Ordering latencies for the requests of two clients on the master protocol instance with an unfair primary, which starts to delay the request of one of the clients after 500 requests.

2.7 Conclusion

In this chapter we have demonstrated that the state-of-the-art robust BFT protocols are not effectively robust. We have shown that a malicious primary replica can drastically degrade performance. To tackle the robustness problem, we have proposed a new approach : *RBFT*, for Redundant Byzantine Fault Tolerance.

The key idea in *RBFT* is to run several instances of a BFT protocol in parallel, and to monitor their performance in order to detect a malicious primary. We have shown that the performance of *RBFT* in the fault-free case is equivalent to the performance of the state-of-the-art robust BFT protocols. Moreover, we have shown that *RBFT* is more robust than existing protocols as it bounds the throughput degradation caused by a malicious primary to 3%, even in drastic conditions where an unlimited number of malicious clients collude with the f malicious nodes to harm the system. Not only we found a small performance loss for the case $f = 1$, but this loss is even smaller for $f = 2$. In our future work, we plan to study how *RBFT* can be extended to deal with closed-loop systems, i.e. systems in which clients wait for replies to their previous requests before issuing new requests.

Chapitre 3

FullReview : Practical Accountability in Presence of Selfish Nodes

Accountability is becoming increasingly required in today's distributed systems. Indeed, accountability allows not only to detect faults but also to build provable evidence about the misbehaving participants of a distributed system. There exists a number of solutions to enforce accountability in distributed systems, among which PeerReview is the only solution that is not specific to a given application and that does not rely on any special hardware. However, this protocol is not resilient to selfish nodes, i.e. nodes that aim at maximising their benefit without contributing their fair share to the system. Our objective in this chapter is to provide a *software solution* to enforce *accountability* on any underlying application in presence of *selfish nodes*. To tackle this problem, we propose the *FullReview* protocol. *FullReview* relies on game theory by embedding incentives that force nodes to stick to the protocol. We theoretically prove that our protocol is a Nash equilibrium, i.e. that nodes do not have any interest in deviating from it. Furthermore, we practically evaluate *FullReview* by deploying it for enforcing accountability in two applications : (1) SplitStream, an efficient multicast protocol, and (2) Onion routing, the most widely used anonymous communication protocol. Performance evaluation shows that *FullReview* effectively detects faults in presence of selfish nodes while incurring a small overhead compared to PeerReview and scaling as PeerReview. This work has been carried out in the context of the PhD thesis of Amadou Diarra and has been published in the proceedings of the 33rd IEEE Symposium on Reliable Distributed Systems (IEEE SRDS'14).

3.1 Introduction

Distributed systems have always been the scene of various software and hardware failures. These failures can have diverse sources such as the crash of machines, bugs, misconfigurations, as well as malicious attacks and users that deliberately tamper with their software to gain some benefit. These failures are especially difficult to deal with when the distributed system spans over multiple administrative domains (also referred to as MAD distributed systems) [Aiyer et al., 2005]. Examples of such systems include peer-to-peer systems, computer grids, network services (e.g., DNS), federated information systems and inter-domain routing.

Accountability, which refers to the ability to detect and expose node faults, is a promising paradigm to deal with these types of failures. In the last decade various solutions have been proposed to enforce accountability for specific applications (e.g., anonymous communication [Corrigan-Gibbs and Ford, 2010], online games [Yahyavi et al., 2013], network storage [Yumerefendi and Chase, 2007], randomised systems [Backes et al., 2009], inter domain routing [Haeberlen et al., 2009], virtualised systems [Andreas*et al.*,]). While these solutions offer strong accountability guarantees, their usability is limited to the specific application domain for which they have been devised. Hence, generic solutions that are not tailored to a specific application have been proposed, some of which rely on trusted hardware (e.g., Trinc [Levin et al., 2009], A2M [Chun et al., 2007], Pasture [Kotla et al., 2012]) while others are generic software solutions.

Our work targets this second category of systems as they do not require users (worldwide) to acquire specific hardware. To the best of our knowledge, PeerReview [Haeberlen et al., 2007a] is the only protocol that falls into this category of systems. In this protocol, nodes log their interactions with other nodes in a *secure log*. This log is then periodically audited by a set of other nodes assigned by the system, i.e. the node's monitors. During their audit, the monitors verify that the monitored node did not tamper with its log and that the latter corresponds to a correct execution of the monitored protocol. An attractive result of PeerReview in addition to its wide applicability is that it provides two theoretical guarantees : completeness and accuracy. Informally, completeness refers to the ability to detect (eventually) all the observable faults, while accuracy refers to the ability to never accuse correct nodes of misbehaviour.

PeerReview works under the Byzantine failure model, i.e. a model where a majority of nodes are correct and where a fixed (known) proportion of nodes in the system can behave arbitrarily. While dealing with Byzantine nodes is important, it has been demonstrated that in open collaborative environments selfish nodes, also called free riders, constitute a real threat [et al., 2000, Krishnan et al., 2004, Feldman *et al.*, 2006, Cunha *et al.*, 2013]. Selfish nodes are nodes that tamper with their software (or download a tampered software developed by others) in order to benefit from the system without contributing their fair share to it.

In PeerReview, nodes are not encouraged to participate to the monitoring of other nodes, which makes it vulnerable to selfish nodes. Specifically, in presence of a proportion of selfish nodes, some nodes in the system can be unsupervised if all their monitors behave selfishly. As a result, these nodes can harm the system without being detected, breaking the completeness property of PeerReview. To measure the impact of this threat in practice, we deployed PeerReview for enforcing accountability in the following two protocols : SplitStream [Castro et al., 2003], an efficient multicast protocol and Onion routing [Goldschlag et al., 1999], the most used anonymous communication protocol. Experiments show that in presence of 30% of selfish nodes, 54% and 85% of messages are lost using the first and the second protocols, respectively.

In this chapter, we embrace the challenge of designing a selfish-resilient protocol for enforcing accountability in distributed systems and present the *FullReview* protocol. The objective of *FullReview* is to force selfish nodes to participate in the monitoring of other nodes while they are executing a given protocol. To reach this objective, the first idea that one may have is to make monitors themselves accountable for their actions by applying PeerReview. We show in this chapter that this is not possible because using PeerReview to monitor itself would require that each node's log contains the log of all the other nodes in the system, which is not scalable.

To overcome this problem, *FullReview* relies on a game theoretic approach to force selfish nodes to stick to the monitoring protocol. Specifically, *FullReview* is a complete redesign of the PeerReview protocol, in which we have embedded incentives in such a way that it is not in the interest of any node to deviate from the protocol, i.e. we prove that *FullReview* is a Nash equilibrium [Nash, 1951] [3].

We implemented *FullReview* and used it to monitor the two protocols SplitStream and Onion routing. Performance evaluation performed on a cluster of 50 machines shows that *FullReview* is resilient to selfish nodes and that it incurs a reasonable overhead compared to PeerReview. Complementary simulations show that *FullReview* scales up to 1000 nodes.

The remaining of this chapter is structured as follows. First, we present the related works in Section 3.2. Then, we show the impact of selfish nodes in PeerReview and present our system model in Section 3.3. Further, we present an overview of *FullReview* and its detailed description in sections 9.3 and 3.5, respectively. Finally, we present the performance evaluation of *FullReview* in Section 3.6 and concluding remarks in Section 9.8.

3.2 Related works

Building robust distributed systems has been at the heart of many research efforts in the last decade. In this context, a new model called the Byzantine, Altruistic, Rational (BAR) model has been proposed [Aiyer et al., 2005]. This model considers three types of nodes : *Byzantine* nodes are nodes

3. Due to the lack of space, this proof is available in the companion technical report : https ://-sites.google.com/site/soniabm/

that can deviate arbitrarily from the protocol ; *rational* nodes are nodes that deviate from the protocol if the performed deviation allows them to increase their own benefit according to a known utility function ; *altruistic* nodes are nodes that always stick to the protocol. In this context, a protocol is said to be BAR-resilient if it tolerates a fixed amount of Byzantine nodes and an unlimited proportion of rational nodes. BAR-resilient protocols often combine game theory by adding incentives that encourage rational nodes to stick to the protocol and accountability techniques that expose Byzantine nodes in case of deviation. In the last years, various collaborative systems have been designed according to this model including protocols for spam resilient content dissemination [Mokhtar et al., 2010], distributed file systems [Aiyer et al., 2005], video live streaming [Mol et al., 2008, Li *et al.*, , Guerraoui et al., 2010a], anonymous communication [Ben Mokhtar et al., 2013] and N-party data transfer [Vilaça et al., 2011]. The process by which a new BAR-resilient protocol is designed usually involves the following steps : (1) define the utility function of rational nodes in the considered protocol ; (2) list all the possible rational deviations according to the defined utility function ; (3) for each identified deviation, propose incentives for rational nodes such that any deviation would engender a loss in the utility perceived by the deviating node and mechanisms that would catch the considered Byzantine deviation ; (4) prove that the proposed protocol is a Nash equilibrium. The major limitation of this approach is that it has to be performed manually by a system expert, which is complex and possibly error prone. Furthermore, any modification in the original system requires to rethink the system as a whole, as the latter may introduce new rational or Byzantine deviations. Rational nodes in the BAR-model correspond to selfish nodes in our work.

A grail that security managers may dream of having is a way of automatically transforming a given protocol into a BAR-resilient protocol. Two solutions that go towards this direction have been proposed in the literature. First, Nysiad [Ho et al., 2008] allows the automatic transformation of a given protocol to a Byzantine resilient system. Nysiad reaches this objective by replicating each node using a variant of replicated state machines (RSMs). However, the resulting system does not deal with rational nodes. Contrarily to Nysiad, PeerReview [Haeberlen et al., 2007a] allows to automatically detect all sorts of observable deviations, including both selfish and Byzantine deviations, that a node would perform in a given monitored protocol. PeerReview reaches this objective by using tamper evident logs and assigning monitors to nodes, which periodically assess the correctness of a node by comparing its log with a correct execution of the protocol obtained using a reference implementation. However, while PeerReview allows to deter faults in the underlying protocol to which it is applied, it does not detect deviations performed by nodes on its own protocol steps.

Our objective in this chapter is to design the first generic protocol that deals with both selfish and Byzantine nodes on any underlying protocol.

3.3 Problem statement and system model

We present in this section an evidence that the PeerReview protocol fails to enforce accountability in presence of selfish nodes in Section 3.3.1. We then present our system model in Section 3.3.2.

3.3.1 Problem statement

Let us consider a system where nodes can be correct, selfish or Byzantine. As introduced in the previous section, correct nodes follow the protocol, Byzantine nodes can behave arbitrarily and selfish nodes aim at maximizing their benefit with respect to a known utility function. The PeerReview protocol has been designed under the assumption that every node is monitored by a set of monitors and that each monitor set contains at least one correct node that executes all the monitoring steps. In this work, we remove this assumption and consider that any node in the system can behave selfishly if it has an interest in doing so. We show that nodes executing PeerReview can skip some steps of the monitoring protocol without being detected and that such behaviour can have a dramatic impact on the performance of the monitored protocol. We provide in the companion technical report [4] a complete analysis of all the protocol steps of PeerReview and list all the selfish deviations that they are subject to. Due to the lack of space, we present here our practical results only. Specifically, to assess the impact of selfish nodes in

4. The technical report is available on my web page : https ://sites.google.com/site/soniabm/

PeerReview, we performed the following two experiments. In the first experiment, we deployed on one hundred nodes the SplitStream protocol [Castro et al., 2003], an efficient tree based multicast protocol, monitored by PeerReview. In the second experiment, we deployed one hundred nodes running the Onion routing protocol [Goldschlag et al., 1999] monitored by PeerReview. In both cases, we used the same experimental settings as the ones described in Section 3.6. In both experiments, if a selfish node notices that its monitors are selfish (e.g., because they never ask to audit its log), it also behaves selfishly with respect to the SplitStream and Onion routing protocols by dropping messages it receives and that are not intended to him.

We measure the percentage of lost messages with respect to the proportion of selfish nodes in the system. Results, depicted in Figure 3.1, show that in presence of up to 30% of selfish nodes, correct nodes running the SplitStream protocol observe 54% of message loss. Similarly, in the Onion routing application, correct nodes experience a loss in their onions that can reach 85% with 30% of selfish nodes in a configuration with five relays. This proportion increases and reaches 100% when the number of relays increases. This is due to the fact that the probability of having a selfish relay in a path increases proportionally with the number of relays constituting this path.

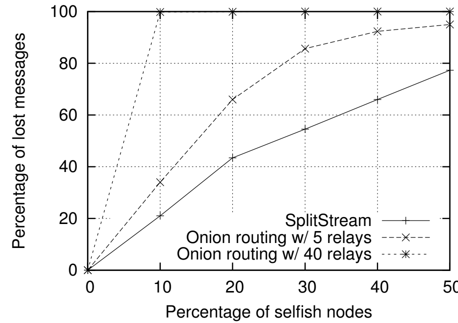


FIGURE 3.1 – Impact of selfish nodes in PeerReviewed SplitStream and Onion routing protocols.

The question we raise in this chapter is thus how to enforce accountability in any underlying protocol in presence of selfish nodes ? We answer this question in the remaining of the paper.

3.3.2 System model

Our target system is composed of two protocols : the monitored protocol to which we will refer as P and the monitoring protocol to which we will refer as M.

Fault model. We consider a fixed proportion of *Byzantine* nodes that can take arbitrary decisions. They can deviate from either P or M protocols for any reason (e.g., a failure, a bug, a threat). Furthermore, we consider any number of *selfish* nodes. These nodes aim at maximising their benefit according to a known utility function. Selfish nodes will deviate from M if they gain some benefit in doing so. Specifically, this benefit can be represented along the following axes :

1. (*Communication*) Sending/receiving as little as possible monitoring messages to/from other nodes.
2. (*Computation*) Performing as little as possible monitoring-related computations for other nodes.

Moreover, we assume that selfish nodes are *risk averse.* This means that before performing any deviation, a selfish node estimates the probability to be detected in the future. If this probability is greater than zero, a selfish node sticks to the protocol. This assumption is commonly used in BAR systems [Aiyer et al., 2005]. This assumption makes particularly sense in accountable systems because the detection of a deviation in these systems directly leads to the eviction of the faulty node from the system. Instead, in systems where the penalty is weaker, e.g., a decrease in a reputation value, it appears more appropriate to consider different selfishness models (e.g., risk affine). This is not the case of our system.

The BAR model also supposes that selfish nodes join and remain in the system for a long time and seek a long-term benefit. Moreover, selfish nodes do not collude and assume that other nodes are correct.

System assumptions. As in PeerReview, we assume a cryptographic identification of nodes. Specifically, each message sent in the network is signed using the sender's cryptographic key. We assume that

cryptographic primitives can not be forged and that hash functions are collusion resistant. Moreover, we assume that messages sent by a sender to a given receiver are always received if retransmitted infinitely often. We assume that nodes have a deterministic reference implementation of P that can be initialised with checkpoints and to which we can inject inputs in order to get the corresponding outputs.

3.4 *FullReview* protocol overview

Let us consider a set of N nodes executing a protocol P defined as a set of deterministic state machines. In *FullReview*, nodes take part in a classical accountability architecture as depicted in Figure 6.1. Specifically, each node i in our system interacts with a set of nodes referred to as i's partners and appearing on its right side in the figure. In addition to its set of partners, node i is assigned a set of monitors that periodically verify whether i sticks to the specification of the protocol P or not. This set of nodes is referred to as $m(i)$ and appears above i in the figure. Symmetrically, i monitors a set of nodes : the set of nodes referred to as $m^{-1}(i)$ and appearing below i in the figure. To perform this monitoring, each node maintains a *secure log* that is tamper evident and append only, in which it writes all its interactions with its partners (details on secure logs are given in Section 3.5.1). This log is periodically audited by i's monitors. Each monitor runs a monitoring protocol M also described as a set of deterministic state machines.

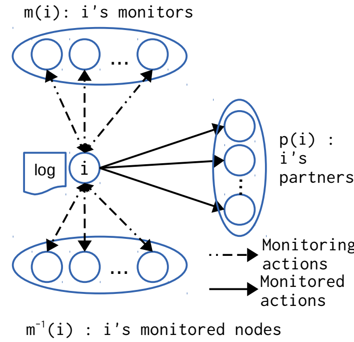


FIGURE 3.2 – Simple accountability architecture.

The objective of *FullReview* is to force selfish nodes to execute all the steps of both protocols P and M and to detect when Byzantine nodes deviate from either protocols P or M. To reach this objective, each node i logs in its secure log all its actions related to both protocols P and M. Then, i's monitors, i.e. nodes in the set $m(i)$, periodically perform a set of verifications on this log. These verifications, which are depicted in the diagram of Figure 3.3, allow each monitor to reach evidence about the correctness of i. Specifically, each node in $m(i)$ starts by verifying that i did not tamper with its log (e.g., that the node did not delete previously inserted entries). We call this verification, which appears on the top of the diagram, log *coherence check*. We explain how this verification is performed in Section 3.5.2.0.

Further, each node in $m(i)$ verifies that i holds a unique log for all its partners. We call this verification, which appears second in the diagram, log *consistency check*. The above two verifications are critical for the accountability system to be effective. Indeed, if a node manages to add/delete log entries or to have multiple versions of a log, it could deviate from the protocol without being detected. We explain how this verification is performed in Section 3.5.2.0.

Moreover, each node in $m(i)$ verifies that the *communication patterns* appearing in i's log are coherent with M and P's state machines (third verification in the diagram). This verification ensures that i's log contains a sequencing of messages that reflect a correct behaviour. For instance, a correct log should contain periodic requests from i to the set of nodes it monitors, i.e. the nodes in $m^{-1}(i)$. The absence of such periodic messages reflects a faulty behaviour. We explain how these verifications are performed in *FullReview* in Section 3.5.2.0.

However, a log that exhibits a correct sequencing of messages is not sufficient to guarantee a correct behaviour. Hence, the last verification that is performed by i's monitors is to assess whether i's log corresponds to a *correct execution* of the protocols P and M or not. Verifying the conformance of i's log

with a correct execution of P is performed as in the PeerReview protocol, i.e. by re-executing the code of the protocol P using a reference implementation. Specifically, the inputs present in i's log are injected in the reference implementation of P and the produced outputs are compared with the outputs present in i's log. Mismatching outputs would constitute an evidence that i did not correctly execute P.

Doing the same verification for the protocol M is not possible. Indeed, as further discussed in Section 3.5.2.0, re-executing the monitoring code is a recursive task and requires that a node's log contains the log of all the other nodes that are linked to him in the monitoring graph (which may possibly be all the nodes in the system). To avoid such an overkill, we identify all the computations performed in the protocol M and ensure that these computations are performed by a set of nodes in parallel. The outcome of each computation is then collected from the various participating nodes and sent to the nodes' monitors. The latter compare the outcome of the computation performed by their monitored node with respect to what other nodes have computed. As selfish nodes do not want to be exposed by correct nodes, they will always perform the computation correctly. In the diagram of Figure 3.3, this last verification is performed before the re-execution of P's code because the latter is more costly. Details of how *FullReview* verifies that nodes correctly executed the computations appearing in both protocols M and P are described in Sections 3.5.2.0.

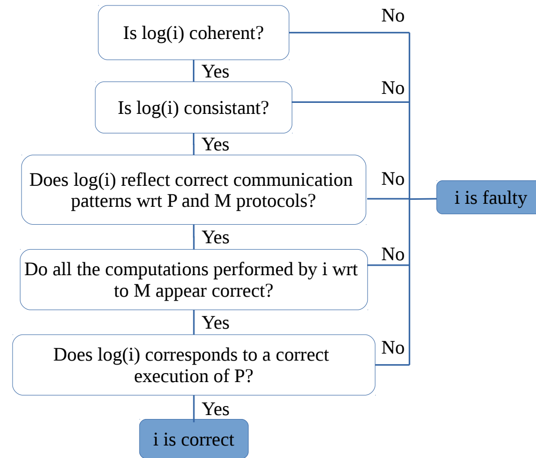


FIGURE 3.3 – *FullReview* monitors decision diagram.

3.5 *FullReview* detailed description

We start this section by introducing secure logs, a central component for enforcing accountability (Section 3.5.1). We then present the two major parts of our protocol, i.e. the audit protocol (Section 6.4.3) and the omission failure protocol (Section 3.5.3). Finally, we give some information about how we carried out the Nash equilibrium proof for our protocol (Section 3.5.4).

3.5.1 Accountability tools : Tamper Evident log

Secure logs are often used to enforce accountability in distributed systems. A secure log is generally used to store the messages exchanged by a node with its partners. According to the requirements of the accountable system, log entries labelled $e_0, ..., e_k$ can contain various information among which an identifier of the logged message, whether the message was sent or received by the node as well as its parameters.

To each log entry e_k corresponds a recursive value h_k, computed as a hash of e_k concatenated with the value of h_{k-1} (where h_{-1} is a fixed value), and an authenticator α_i^k, which is a message containing the value of h_k signed with i's private key, i.e. $\alpha_i^k = (h_k)_{\sigma_i}$. Authenticators allow verifying that a node log has not been tampered with. For instance, consider a node j among node's i monitors. If j gets a pair of authenticators α_i^0 and α_i^k corresponding to the entries e_0 and e_k of i's log respectively, it can ask

h_0	e_0
...	...
h_{k-1}	e_{k-1}
h_k	e_k

e_k : entry k
$h_k = H(e_k || h_{k-1})$
$\alpha_i^k = (h_k)_{\sigma i}$

FIGURE 3.4 – Example of a secure log.

i of its log entries $e_0, ..., e_k$ and recompute $h_0, ..., h_k$. If the computed h_k differs from the one held by j, the latter can accuse i of tampering with its log. Further, j can convince any other correct node of the misbehaviour of i by sending to it the signed authenticators α_i^0 and α_i^k along with the log entries sent by i.

3.5.2 *FullReview* selfish-resilient audit protocol

Using the secure log described above, a node j monitoring the behaviour of a node i performs a set of verifications to assess the correctness of i following the diagram of Figure 3.3. However, selfish monitors might be tempted not to perform these verifications. In order to force monitors to perform them, we make audits proactive. Specifically, we divide time in rounds and give the responsibility for each node to periodically (e.g., at the end of each round) ask its monitors to audit its log following the diagram depicted in Figure 3.5 (the Audit_req message sent from i to its monitors $m(i)$). Then, each monitor performs the required verifications and produces a certificate of correctness if the node passes all of them. In the opposite case, i's monitors send a proof of misbehaviour to i including the evidence of i's misbehaviour, which any correct node can recompute. This certificate is then used by i at the beginning of the following round in order to communicate with its partners. Without such a certificate, i's partners will refuse to interact with i. Note that some of i's monitors might be unresponsive (either because of a failure or to avoid auditing i's log). We describe how we deal with this situation in Section 3.5.3. Finally, after collecting the outcome of the audit produced by its monitors (Audit_resp message), i forwards the aggregated outcome to the monitors of each of its monitor (Fwd_outcome message). This last step is useful for the monitors of i's monitors (i.e. $m(m(i))$) in order to verify whether the nodes they monitor correctly performed their monitoring tasks or not. Further details on this verification are given in Section 3.5.2.0.

In the following we describe in detail the set of verifications performed by the monitors of each node to assess its correctness.

Log coherence check

Allows verifying that a node's log has not been tampered with. Consider a node j that monitors a node i. If j gets a pair of authenticators α_i^0 and α_i^k corresponding to the entries e_0 and e_k of i's log respectively, it can ask i for its log entries $e_0, ..., e_k$ and recompute $h_0, ..., h_k$. If the computed h_k differs from the one held by j, the latter can accuse i of tampering with its log. Further, j can convince any other correct node of the misbehaviour of i by sending to it the signed authenticators α_i^0 and α_i^k along with the log entries sent by i. To perform this type of verification each node shall log each message it sends as part of the protocols P and M and send the corresponding authenticator to its partner. Furthermore, each node shall forward the received authenticators to its partners' monitors. However, selfish nodes might be tempted not to follow these steps, i.e. avoid attaching authenticators with messages they send and/or avoid forwarding received authenticators to the partner's monitors. We show how we deal with this issue in Section 3.5.2.0.

Log consistency check

Node i might be tempted to maintain many correct logs (e.g., one log for each node with whom it interacts). To detect this type of misbehaviour, a monitor j that holds a set of authenticators sent by i to other nodes verifies that these authenticators belong to the same log. Similarly to the log coherence check, this verification requires that nodes attach authenticators to all messages they sent and forward

received authenticators to their partners' monitors, and that monitors perform the consistency check. We show how we encourage selfish nodes to perform these steps in Section 3.5.2.0.

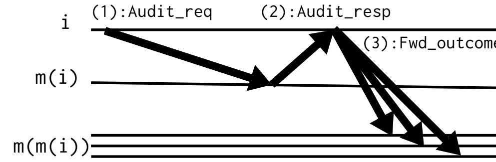


FIGURE 3.5 – *FullReview* audit protocol.

Verifying communication patterns

In this part of the protocol, a node in the monitor set of a node i is responsible for assessing whether the log of i reflects correct communication patterns with respect to the state machines of P and M. However, it is not possible to consider the state machines of these two protocols separately as in some situations steps of M need to be interleaved with steps of P. For instance, as seen in the log coherence and consistency checks described above, nodes need to send authenticators along with messages related to P and need to forward authenticators received along with messages related to P. To reach this objective, the state machine of the protocol P is automatically augmented with a set of mandatory transitions as depicted in Figure 3.6. In this figure, and in all the figures depicting automata in the paper, transitions are labelled as follows : (P|M :IN|OUT :message_type) where the first part refers to whether the message belongs to the protocol P or M ; the second part indicates respectively whether the message is received or sent and the third part is the message type. This figure shows that each time a node is expecting a message as part of the protocol P, it should : (1) upon receiving the message, forward the included authenticator to the sender's monitors (transition labelled (M :OUT :fwd_auth)) ; or (2) accuse the sender if the message did not contain an authenticator by sending an accusation message to the sender's monitors (transition labelled (M :OUT :accuse)) ; or (3) suspect its partner if the latter did not send the expected message (transition labelled (M :IN :timeout)). The transitions following this latter transition are further described in Section 3.5.3.

Augmenting all the transitions of P related to the reception of messages as shown in Figure 3.6 forces selfish nodes to attach authenticators to the messages they send (otherwise, nodes that receive theses messages might accuse them). Furthermore, it forces selfish nodes to forward the received authenticators to their partner's monitors (otherwise, their monitors might accuse them of behaving selfishly).

In addition to verifying that a monitored node's log is coherent with the state machine of the P augmented automaton, monitors verify that the log is coherent with M state machines related to the audit protocol (described earlier in this section) and with M state machines related to the handling of omission failures. The former state machines are depicted in Figures 3.7 and 3.8 while the latter are described in the following section. Specifically, the automaton of Figure 3.7 shows the correct communication patterns of a node i asking one of its monitors for an audit (transition labelled (M :OUT :audit_req)). After sending his audit request, node i either receives a response from its monitor containing the outcome of the audit (transition labelled (M :IN :audit_resp)) or does not receive a reply (the transition labelled (M :IN :timeout)). In the former case, node i forwards the outcome of the audit to the monitors of all of its monitors, which allows them to verify that their monitored node reached the same outcome about the correctness of i as the other monitors of i. In the latter case, i considers that its monitor has failed and handles this failure as described in the following section.

The automaton of Figure 3.8 shows the correct communication patterns of a monitor j that receives an audit request from a node i that it is monitoring (the transition labelled (M :IN :audit_req)). After the reception of this request, node j performs the audit of i's log and sends back the outcome to i (the transition labelled (M :OUT :audit_resp)).

Verifying computations

In this part of the protocol, each monitor j in the monitor set of a node i verifies that the computations performed by i as part of the protocols P and M are correct. For the computations performed by i and

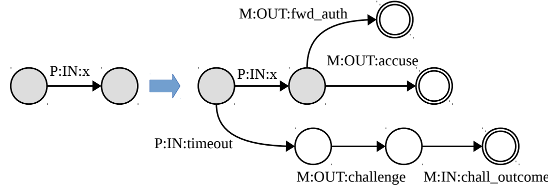


FIGURE 3.6 – Augmenting the P protocol.

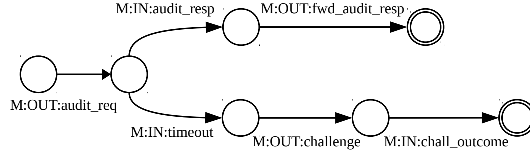


FIGURE 3.7 – Sending audit requests.

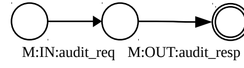


FIGURE 3.8 – Dealing with audit requests.

that are related to P, j use checkpoints stored in i's log and initializes the reference implementation it has with the oldest non-verified checkpoint. Further, j replays all the inputs available in the portion of i's log it is auditing and verifies that the outputs produced by the reference implementation match with the outputs stored in the log. If the computed outputs do not match with the logged ones, j accuses i of misbehaviour. Whether i passes this verification or not, j stores the outcome of the audit along with the authenticators corresponding to the portion of the log of i that it has audited and sends the outcome of the audit to i as prescribed by the audit protocol (described earlier in this section).

Contrarily to the computations related to the protocol P, verifying those related to the monitoring protocol M can not be done by re-executing the steps of the protocol M. To intuitively understand why, let us consider the following example, where node i monitors node $i-1$ (among other nodes) and is monitored by node $i+1$ (among other nodes). At a given execution time, the monitor of node $i+1$, say node $i+2$ would like to audit node $i+1$'s log to verify that it is correctly performing its monitoring actions regarding the behaviour of i. To do so, node $i+2$ needs to get access to node i's log, which is available in $i+1$'s log. Hence, to check whether $i+1$ has correctly done his monitoring actions, it needs to verify whether $i+1$ correctly audited i's by replaying the audit verifications itself. However, to verify whether i is effectively correct, $i+2$ must verify whether i correctly executed its monitoring steps with respect to $i-1$. To do this last verification, $i+2$ must verify whether the outcome of i's audit over $i-1$'s log is correct and is thus obliged to audit itself $i-1$'s log. This process clearly leads each node to recursively obtain and audit the logs of all the other nodes that it is connected to in the monitoring graph, which is not practical.

To avoid such an overkill, we use incentives to force selfish nodes to correctly perform the computations taking part of the protocol M instead of recomputing them. Specifically, as described earlier, after receiving the outcomes of the audit sent by its monitors, a node aggregates these results and forwards them to the monitors of its monitors. These nodes receive an information of the type : (audited node ID, authenticators, monitor ID, outcome) for each of i's monitors that took part in the audit. If a majority of monitors detects a misbehaviour in i's log and one of them, say node j, did not, then j is accused of misbehaviour. In this situation, j is selfish if it claimed that i is correct without performing the verification or Byzantine if it replied arbitrarily. As selfish nodes do not want to be excluded from the system, they always perform the computations related to M correctly. Instead, if a majority of monitors but j considers that i is correct, j is considered Byzantine, as a selfish node do not have any interest in accusing a correct node of misbehaviour.

3.5.3 Handling omission failures

The handling of omission failures is done in *FullReview* as depicted in Figure 3.9. Specifically, if a node i waits for a given message from a given node j for too long, i suspects j (after step (1) in the figure). To do so, i creates a challenge for j and sends this challenge to j's monitors (step (2) in the figure), who forward the challenge to j (step (3) in the figure). If j is still alive in the system then it replies to the challenge (step (4)). Whether j replied or not to the challenge, after a given amount of time j's monitors send an outcome of the challenge to i summarizing the situation (step (5)).

A selfish node may be tempted not to suspect a node even if it has waited for too long to receive a message assuming that other nodes will take care of that. Similarly, a selfish monitor might be tempted not to forward a challenge send by i to j assuming that the other monitors will do so. These two deviations are not possible in *FullReview* because of the verification of communication patterns performed by monitors on their monitored node's log. Specifically, the automata of Figures 3.10 and 3.11 show the correct communication patterns that should be present in the log of a node when, as a monitor, it receives an omission failure complaint about one of its monitored nodes and when, as a suspected node, it receives a challenge from its monitor. The log of a selfish node should be conform to these automata, otherwise it is accused by its monitors.

In addition, a selfish node might be tempted to suspect a node instead of performing a costly interaction with him. To avoid this deviation, we make the cost of suspecting a node higher than the cost of interacting with him. To avoid to overload the system, we adapt this cost to each step of the protocols P and M. For instance, if sending a message m costs xB of bandwidth to node i, we make the cost of suspecting a node j to whom i was supposed to send m equal to $x + \delta B$. As such, a selfish node i will always prefer to send m instead of suspecting j.

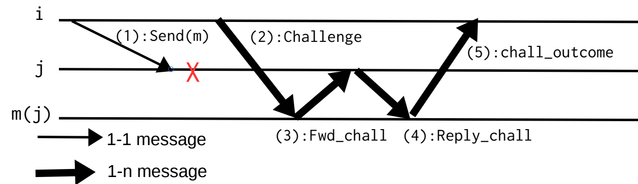


FIGURE 3.9 – *FullReview* handling of omission failures

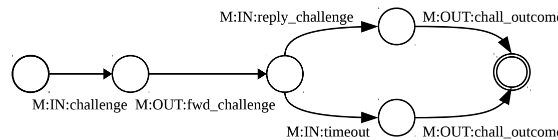


FIGURE 3.10 – Dealing with omission failures.

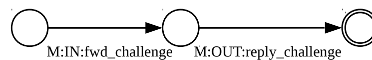


FIGURE 3.11 – Dealing with omission suspicions.

3.5.4 Resilience to selfish nodes

We carried out a detailed analysis of all the protocol steps of *FullReview*. For each of these steps we listed all the selfish deviations and the corresponding incentives that prove that selfish nodes do not have any interest in performing them. As a result we prove that *FullReview* is a Nash equilibrium. Due to the lack of space, this analysis is presented in the technical report available on my web page.

3.6 Performance Evaluation

In this section we evaluate the performance of *PeerReview* and *FullReview* with two distributed applications : SplitStream and Onion routing. We start by introducing the two applications and our experimental settings in Section 3.6.1 and 3.6.2, respectively. We then present the performance of *FullReview* in presence of selfish nodes (Section 3.6.3) and in the fault-free case (Section 3.6.4). Finally, we assess the scalability of *FullReview* (Section 3.6.5).

Overall, our evaluation draws the following conclusions. First, we show using real experiments that *FullReview* can effectively detect faults in presence of selfish nodes. Second, *FullReview* adds a small overhead compared to *PeerReview* both in terms of traffic generated and storage. Finally, using complementary simulations, we show that *FullReview* is scalable up to at least 1000 nodes.

3.6.1 Applications

Accountable Efficient Multicast

SplitStream [Castro et al., 2003] is a protocol that organises nodes in a tree structure where each node receives multicast messages from its parent node and forwards them to its child nodes. The specificity of SplitStream is that it aims at balancing the forwarding load between nodes. It reaches this objective by splitting the multicast stream into stripes and using different multicast trees to distribute each stripe. For our experiments, the source node generated a video stream of 300kb/s, which is a common rate for video-streaming applications. Each packet emitted by the source was sent through a different multicast tree where each node had two children.

In SplitStream, selfish nodes deviate by not forwarding updates to their child nodes. As a result they can get the video stream while saving bandwidth. However, in presence of selfish nodes, correct nodes may experience frame loss and consequently receive a degraded version of the video stream.

Accountable Anonymous Communication

Onion routing [Goldschlag et al., 1999] is a protocol designed for anonymous communications. It is the protocol used in the TOR project [Dingledine et al., 2004], which is widely used by thousands of users daily. In this protocol, when a node S wants to send a message to a node D, it chooses R other nodes, called relays, that will forward the message up to its destination. Node S encrypts successively the message using the public key of each of these relays, which constitutes the *onion* and then sends it to the first relay. Each relay decrypts one layer of the onion (i.e. removes one layer of encryption) and forwards it to the next one until it reaches its final destination. For Onion routing experiments, each node periodically emitted a packet to a randomly chosen node through a parametric number of relays. In all our experiments, messages have a fixed size of 10kB ; smaller messages are padded with additional bytes in order to meet this requirement. Fixing message size is usually done in onion routing as it avoids an attacker to follow the progression of an onion in the system by comparing the size of forwarded messages.

In this protocol, a selfish node can choose not to forward an onion that is not intended to him. As a result, the destination will never receive the original message. The objective with designing a selfish-resilient version of Onion routing is to ensure that nodes will forward the onions they receive while providing anonymity guarantees.

3.6.2 Experimental settings

We have measured the performance of SplitStream and Onion routing in two configurations : (i) with *PeerReview* and (ii) with *FullReview*. Our experiments have been performed in two different settings. First, we performed experiments in real conditions using the public Grid'5000 cluster. In this cluster we used 50 quad-core physical machines clocked at 2.6GHz with 4GB of RAM that are interconnected via a Gigabit switch. These experiments have been run by deploying one logical node per physical machine and corresponding curves are annotated with [G5K] in their labels. To complement our experiments, we performed simulations using the *PeerReview* simulator that has been developed by *PeerReview* authors [5].

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

We performed simulations with up to 1000 nodes, in order to assess the scalability of *FullReview*. Results of these experiments are annotated with [SIM] in their labels.

3.6.3 Performance in presence of selfish nodes

In this section we show that *FullReview* tolerates selfish nodes. To this end, we perform two experiments. In the first experiment, selfish nodes follow the model presented in Section 3.3.2. Specifically, they deviate only if they have an interest to do so and if there is no risk to be caught. Instead, in the second experiment, we consider that selfish nodes deviate if they have an interest to do so without considering the risk of exclusion. This latter experiment shows that if they decide to do so, selfish nodes are quickly detected by their monitors and excluded from the system.

For both experiments we used the two applications monitored by *PeerReview* and *FullReview*. The number of monitors per node is fixed to 2 and the audit period is set to 10s.

The results of the first experiment are presented in Figure 3.12. This figure shows the percentage of received messages as a function of the percentage of selfish nodes. SplitStream and *FullReview* are deployed with 50 nodes on G5K. We evaluate *FullReview* with different number of relays (5, 10, 20 and 40), that are chosen at random. However, due to lack of space, we present the results with 5 relays only. Increasing the number of relays leads to worst results for *PeerReview* as the probability to choose a selfish node in an Onion routing path becomes higher. We first observe in this figure that, using *PeerReview*, SplitStream and Onion routing do not tolerate selfish nodes. Indeed, in presence of only 10% of selfish nodes, only 79% and 66% of messages are received in the SplitStream and Onion routing applications, respectively. This represents a loss of 21% and 34% messages, respectively, which is not acceptable. This percentage decreases when the proportion of selfish nodes increases, reaching 23% in SplitStream and 5% in Onion routing, in presence of 50% of selfish nodes. Instead, using *FullReview*, we observe that all messages are received in both applications as selfish nodes have no interest in deviating.

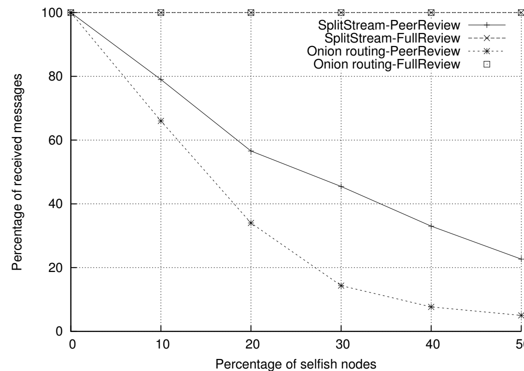


FIGURE 3.12 – [G5K] Percentage of received messages in SplitStream and Onion routing as a function of the percentage of selfish nodes.

The results of the second experiment are presented in Figure 3.13. In this experiment, we measure the percentage of received messages in SplitStream with *PeerReview* and *FullReview* where selfish nodes start to deviate from the protocol after 20s. This experiment has been launched with 50 nodes using simulations. As explained above, in this experiment, selfish nodes behave selfishly without reasoning on the risk of being detected. Using *PeerReview*, we observe that selfish nodes impact the system as soon as they behave selfishly, without ever being detected. Using *FullReview*, we observe that selfish nodes impact the system during a small time frame, corresponding to the audit frequency, after which they are detected and evicted from the system. As a result, all the messages are received for the rest of the experiment. Note that choosing a smaller audit period allows the system to detect selfish nodes more rapidly, but at the expense of some additional overhead, as we show in the next section.

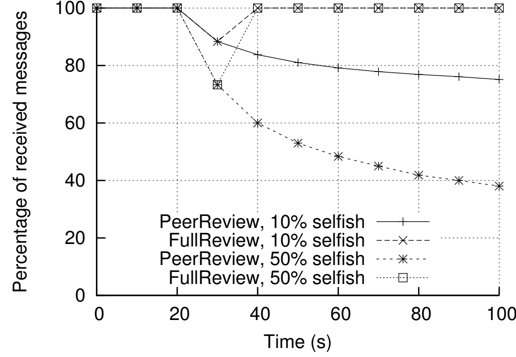


Figure 3.13 – [SIM] SplitStream percentage of received messages during an experiment in which between 10% and 50% of nodes start to act selfishly after 20s.

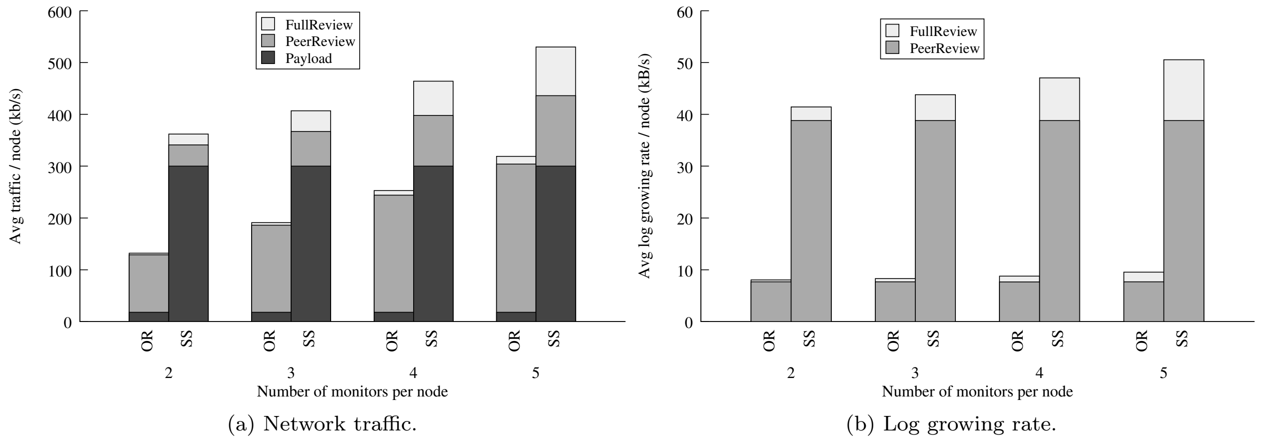


Figure 3.14 – [G5K] Average network traffic and log growing rate per node of SplitStream (SS) and Onion routing (OR) w.r.t. the number of monitors.

3.6.4 Performance in the fault-free case

In this section we assess the performance and overhead of *FullReview*, compared to *PeerReview*, in the fault-free case. To this end, we perform three experiments. We launch each of the experiments of this section both using simulations and G5K, but we show the results on G5K only. The results using simulations are consistent and can be found in the companion technical report available on my web page.

In the first two experiments, we measure the network traffic and the rate at which the logs grow w.r.t. the number of monitors, in *PeerReview* and *FullReview* respectively. In the case of Onion routing, an onion path was composed of 5 relays. Figure 3.14 presents the results for both SplitStream and Onion routing. Each value has been obtained by running the system with 50 nodes during 5 minutes.

In the left figure, each bar represents the traffic due to the payload of the application. On top of this payload is the traffic due to *PeerReview*, on top of which is the overhead of *FullReview* in addition to the one of *PeerReview*. In this figure, we observe that the average traffic per node increases w.r.t. to the number of monitors for both *PeerReview* and *FullReview* in the two applications. This is due to all the messages that need to be exchanged between nodes and their monitors. Further, we observe that the overhead due to accountability in the SplitStream application has an overall cost of 14% in *PeerReview* with two monitors and an extra cost of 7% in *FullReview*. This overhead grows up to 45% for *PeerReview* and an additional 31% for *FullReview* when 5 monitors are used. These costs are much higher if compared to the payload of the Onion routing application. For instance, enforcing accountability in Onion routing using *PeerReview* generates a traffic of 129kb/s per node while the application itself generates a payload of only 18kb/s per node. However, put into context this result is not bad, as enforcing accountability in anonymous communication protocols is a very challenging task for which existing solutions often require the heavy use of broadcast primitives (e.g., RAC [Ben Mokhtar et al., 2013], Dissent [Corrigan-Gibbs and

Ford, 2010]). Further, assuming that nodes are connected using Gigabit links (in the case of a LAN) or even using few Megabit links (in the case of a WAN), 129kb/s seems a reasonable overhead. The good news is that if the developer accepts to pay the cost of accountability using *PeerReview* in a system with a small payload, using a selfish resilient accountability system, i.e. *FullReview*, would cost him an extra 3kb/s (i.e. 2% more traffic) with two monitors and an extra 15kb/s (i.e. 5% more traffic) with five monitors. Note that, overall, enforcing accountability using *PeerReview* is more expensive in the Onion routing application than in the SplitStream application because in the former application the full onions are stored in the log while in the latter instead of storing the video chunks received by nodes in the log, we store only their identifier. Indeed, storing onions was the only way we found to enable monitors to verify that a node has correctly decrypted and forwarded an onion it received.

In the right figure, each bar represents the average growing rate of the log of nodes. Similarly to the previous figure, the cost of *FullReview* is shown as a delta in addition to the cost of *PeerReview*. Note that logs do not grow forever. Indeed, as in *PeerReview*, logs are truncated after a given amount of time and audits are performed only for the new parts of the log. Obviously, the longer the logging period chosen by the designer, the higher the probability to deter faults.

Results depicted in this figure show that the log growing rate of the SplitStream application is higher than log growing rate of the Onion routing application, which is due to the fact that the SplitStream application generates more messages to send the video stream than Onion routing, and thus more interactions are added to the log. Further we observe that the higher the number of monitors per node the higher the log growing rate. On the Onion routing application, the overhead in terms of log growing rate is equal to 4.9% when using *FullReview* with two monitors and increases up to 24% when using five monitors. On the SplitStream application, this overhead is higher as it spans from 6.8% to 30% when using respectively two and five monitors. Yet, we consider this overhead as reasonable. Indeed, in the worst of our experiments (i.e, in the SplitStream application using five monitors), for 24 hours logging, nodes need to devote 4.4GB of storage for enforcing accountability in presence of selfish nodes, which is reasonable.

In the third experiment, we measure the impact of the audit period on the overhead of *FullReview* compared to *PeerReview*. The audit period was ranging from 1s to 30s. We set the number of nodes to 50, with 2 monitors per node and 5 relays for the Onion routing application. Each experiment last 5 minutes. Results, presented in Table 3.1, show that even with a high frequency of audit (i.e. every second), *FullReview* generates only 6.7% more traffic and logs are 8.2% larger than *PeerReview* in the worst case.

	Audit period	1s	5s	10s	30s
SSSS	Log size	+7.4%	+6.8%	+6.7%	+6.4%
	Network traffic	+6.7%	+6.2%	+6.1%	+5.9%
OROR	Log size	+8.2%	+4.9%	+4.8%	+3.3%
	Network traffic	+2.9%	+2.6%	+2.3%	+1.9%

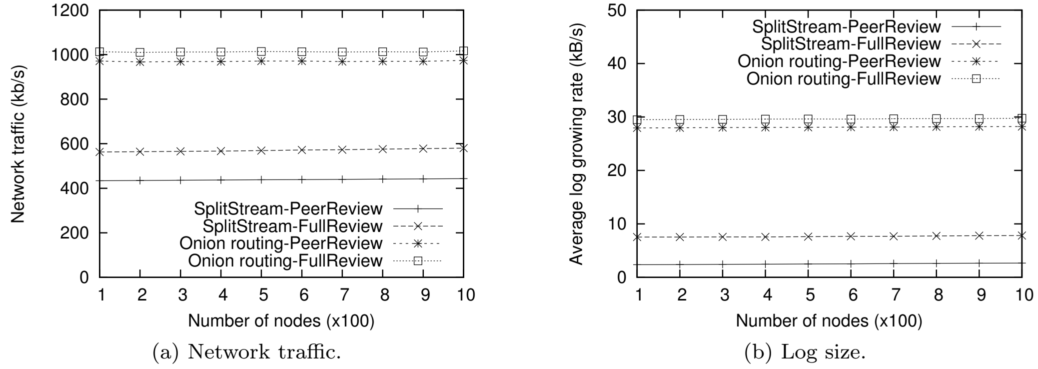
Table 3.1 – [G5K] Overhead of *FullReview* compared to *PeerReview*, for both SplitStream (SS) and Onion routing (OR), with an audit period ranging from 1s to 30s.

To summarize, *FullReview* adds a small overhead to *PeerReview* in terms of generated traffic and log size. This overhead is mainly due to the new log entries inserted by *FullReview* to detect selfish nodes. Similarly to *PeerReview*, the cost of *FullReview* increases with the number of monitors per node and with the frequency of the audits. Overall, accounting for the increasing resources (storage and network bandwidth) at the disposal of a large public (Terabytes of storage and Megabits of network bandwidth), the cost of enforcing accountability in presence of selfish nodes becomes a realistic option.

3.6.5 Scalability of *FullReview*

In this section we show that SplitStream-*FullReview* and Onion routing-*FullReview* scale up to at least 1000 nodes.

Figure 3.15 presents the network traffic and the log growing rate of SplitStream and Onion routing, for both *PeerReview* and *FullReview*, as a function of the number of nodes in the system. Each value



(a) Network traffic.

(b) Log size.

FIGURE 3.15 – [SIM] Average network traffic and log growing rate of SplitStream and Onion routing w.r.t. the number of nodes in the system.

has been measured via a simulation that lasts 100s. Moreover, the system has been configured with 5 monitors per nodes. As one could expect from the results of Figure 3.14, using less monitors provides better performance. In addition, the audit period was set to 10s. Finally, Onion routing was configured with 40 relays and was sending onions at a rate of 16kb/s.

From this figure we can draw the following conclusions. First of all, for both SplitStream and Onion routing, the network traffic and log growing rate of *FullReview* is within a constant factor of *PeerReview*. For instance, with SplitStream, the log growing rate (resp. network traffic) of *FullReview* is equal to 1.4x (resp. 1.3x) the one of *PeerReview*. This is due to the fact that *FullReview* adds a constant number of operations on the ones performed by *PeerReview*. Second, we can observe that *FullReview* scales up to 1000 nodes, as the network traffic and log size remain fairly stable despite the increase of the number of nodes. The reason is that each node always interacts with the same number of nodes on average, whatever the overall number of nodes in the system (i.e. its partners w.r.t. to the application and a fixed number of monitors).

3.7 Conclusion

this chapter addresses the problem of accountable distributed systems in presence of selfish nodes. We have shown that *PeerReview*, the only software generic solution to enforce accountability, does not tolerate selfish nodes. To tackle this problem we propose the *FullReview* protocol. This protocol uses game theory techniques by embedding incentives that force selfish nodes to stick to the protocol. We have evaluated *FullReview* on a cluster of physical machines and using simulation with two applications : SplitStream, an efficient multicast protocol, and Onion routing, the most widely used anonymous communication protocol. Our evaluation makes the following points. First, contrarily to *PeerReview*, *FullReview* effectively tolerates selfish nodes. Second, *FullReview* has a low additional overhead compared to *PeerReview*. Finally, *FullReview* scales up to 1000 nodes.

Chapitre 4

PAG : Private and Accountable Gossip

A large variety of content sharing applications rely, at least partially, on gossip-based dissemination protocols. However, these protocols are subject to various types of faults, among which selfish behaviours performed by nodes that benefit from the system without contributing their fair share to it. Accountability mechanisms (e.g., PeerReview, AVMs, FullReview), which require that nodes log their interactions with others and periodically inspect each others' logs are effective solutions to deter faults. However, these solutions require that nodes disclose the content of their logs, which may leak sensitive information about them. Building on a monitoring infrastructure and on homomorphic cryptographic procedures, we propose in this chapter *PAG*, the first accountable and partially privacy-preserving gossip protocol. We assess *PAG* theoretically using the ProVerif cryptographic protocol verifier and evaluate it experimentally using both a real deployment on a cluster of 48 machines and simulations. The performance evaluation of *PAG*, performed using a video live streaming application, shows that it is compatible with the visualisation of live video content on commodity Internet connections. Furthermore, *PAG*'s bandwidth consumption inherits the desirable scalability properties of gossip when the number of users in the system grows. This work has been carried out in the context of the PhD thesis of Jérémie Decouchant and has been published in the 36th International Conference on Distributed Computing Systems (IEEE ICDCS'16).

4.1 Introduction

Peer-to-peer content sharing systems account for a large amount of traffic in today's Internet [Basher et al., 2008]. Examples of such systems that are widely used by thousands of users everyday include P2P content delivery networks (e.g., BitTorrent, eMule), on demand audio and video streaming (e.g., PopcornTime, Velocix [vel,]) and P2P TV (e.g., PPLive [Vu et al., 2010], LiveSky [Yin et al., 2009]). These systems often rely (totally or partially) on gossip-based message dissemination schemes [Bonald *et al.*, ,Kermarrec et al., 2003] where nodes periodically exchange data chunks with randomly chosen nodes.

Gossip-based systems are cost effective, scalable up to millions of users and highly resilient to churn. However, they rely on the willingness of the users to share their resources (e.g., upload bandwidth, CPU cycles, memory) with each other. Consequently, they are vulnerable to selfish behaviours performed by users that modify their software or use a tampered version of it (e.g., BitThief, BitTyrant).

Recent studies performed on real peer-to-peer systems (e.g., [Eidenbenz*et al.*, 2011, Cunha *et al.*, 2013]) have shown that clients behaving selfishly get better performance than compliant ones (e.g., in a live streaming system, they would download the video stream faster while saving upload bandwidth). Moreover, these studies show that above a given proportion of selfish clients, the compliant clients observe a major degradation in the quality of the video stream they obtain. Dealing with selfish behaviours has thus been the centre of active research in the last decade.

In this context, accountability protocols, which have recently been proposed for fault detection in distributed systems (e.g., [Haeberlen et al., 2007a, Andreas*et al.*, , Diarra et al., 2014]) are promising candidates for effectively dealing with selfish behaviours. In an accountable system, nodes register their interactions with each other in secure logs. These logs are periodically inspected by a set of nodes in the

system acting as monitors. In case of fault detection, the monitors generate a proof of misbehaviour and the misbehaving nodes get punished. Thanks to their effectiveness, these protocols have been recently used to build selfish-resilient content dissemination systems (e.g., [Ben Mokhtar et al., 2014, Aditya et al., 2012]).

However, these solutions require nodes to share their log with each other, which may leak sensitive information about them. For instance, in a video streaming application a curious monitor inspecting a node's log may learn about the user's *interests*, which may disclose information such as her sex, age, sexual, political or religious preferences. Furthermore, monitors can infer further information by analysing *the interest graphs* made of links between nodes sharing similar interests.

In this context, a challenging objective is to build protocols that enforce accountability while preserving their users' privacy. However, this objective may seem contradictory as there is a clear trade-off between privacy and accountability : the deeper the verifications that can be performed regarding the behaviour of a node, the more faults can be deterred but the more information need to be collected.

Few protocols have been recently proposed to address both accountability and privacy issues in distributed systems. Among them Dissent [Corrigan-Gibbs and Ford, 2010] and RAC [Ben Mokhtar et al., 2013] allow nodes to exchange messages anonymously while forcing them to stick to the original protocol specification. However, these systems heavily rely on all-to-all communications, which makes their performance unsuitable for live content dissemination systems.

We propose in this chapter *PAG*, an accountable and privacy-preserving gossip protocol, practical for live content sharing applications, where messages exchanged between any two nodes are kept private. *PAG* is decentralised and relies on nodes acting as monitors to enforce the correct dissemination of content updates. Specifically, through an homomorphic encryption of the disseminated messages, monitors enforce accountability without getting access to the content of the exchanged messages, thus protecting users' *interests*. Finally, through the modification of cryptographic keys at every hop of the dissemination process, monitors can not follow the progress of a given content update in the dissemination graph, thus preventing monitors from building *interest graphs*.

We assess *PAG* both theoretically and experimentally regarding accountability, privacy and performance. Regarding *accountability*, our analysis shows that *PAG* is a Nash equilibrium [Nash, 1951], which means that selfish nodes have no interest in deviating from the protocol. Further, regarding *privacy*, we prove the resilience of *PAG* against a global and active opponent using the ProVerif cryptographic protocol verifier [Blanchet, 2001]. Finally, regarding *performance*, we show through the implementation of a video live streaming application instantiated using 432 clients deployed in a cluster of 48 machines that : (1) *PAG* is practical in terms of bandwidth and CPU costs for streaming live video content compared to anonymous communication protocols like RAC and (2) its cryptographic overhead is reasonable for modern architectures. Complementary simulations show that *PAG* scales logarithmically in terms of bandwidth consumption.

The rest of the paper is structured as follows. Section 8.2 provides some background about accountability and privacy in gossip. Section 4.3 presents our system model and our objectives. Section 4.4 presents the building blocks of *PAG*, which consist in a monitoring infrastructure and homomorphic messages hashing. Section 9.4 details the message exchanges of nodes running *PAG*. Section 4.6 presents the privacy guarantees and an accountability analysis of *PAG*. Section 9.6 presents a detailed performance evaluation. Section 6.7 reviews the related works. Section 9.8 concludes this chapter.

4.2 Accountable Forwarding and Privacy

In this section, we present the principles of the gossip paradigm and introduce selfish deviations that nodes may execute (part 4.2.1). We further present accountability solutions (part 4.2.2) and explain how accountability solutions threaten the privacy of users (part 4.2.3).

4.2.1 Principles of gossip and selfishness

Peer-to-peer gossip protocols aim at reliably distributing a content (e.g., a video stream, membership updates) to a set of interested nodes. To do so, gossip protocols handle two tasks. First, they handle the

neighbourhood of nodes by providing them with a selection of partners with which they can interact. This is commonly achieved by relying on a full membership protocol (e.g., [Johansen et al., 2006, Li *et al.*,]), or on a distributed random peer sampling protocol (e.g., [Ganesh et al., 2001, Jelasity et al., 2007]). Second, gossip protocols handle message exchanges enabling a content to be disseminated to all the participating nodes with a high probability [Kermarrec et al., 2003]. As membership management does not handle content, we focus on bringing accountability and privacy to the second task. Using the gossip principle, message exchanges are organized in *rounds* (whose duration is called the *gossip period*). A special node that holds the content to disseminate (also called the *source*), generates and periodically sends chunks of this content (also called *updates*), to a set of nodes chosen uniformly at random. Then, periodically, each node taking part in the dissemination is in charge of sharing the updates it receives with f other randomly selected nodes (f is also called the dissemination *fanout*).

Figure 4.1 illustrates the gossip-based dissemination of updates, from the point of view of a node X depicted in the centre of the figure. Specifically, node X has a set of f_p predecessors $\{P_1, \ldots, P_{f_p}\}$ and a set of f_s successors $\{S_1, \ldots, S_{f_s}\}$ that have been picked uniformly at random from the nodes participating in the system. In this example, during round R, node X receives a set of data chunks from its predecessors (i.e., $\{u_1\}$ from P_1, ..., $\{u_{f_p}\}$ from P_{f_p} in the figure) and has to forward the received chunks in the following round R+1 to all its successors (i.e., $\{u_1, \ldots, u_{f_p}\}$ to $S_1, \ldots, S_{f_s}$ in the figure).

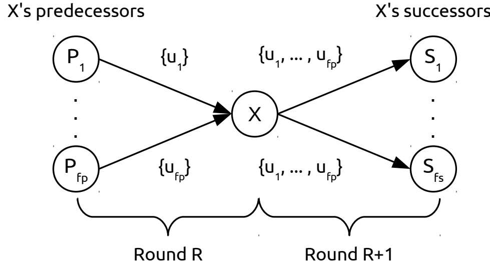


FIGURE 4.1 – Forwarding of updates in a gossip-based system

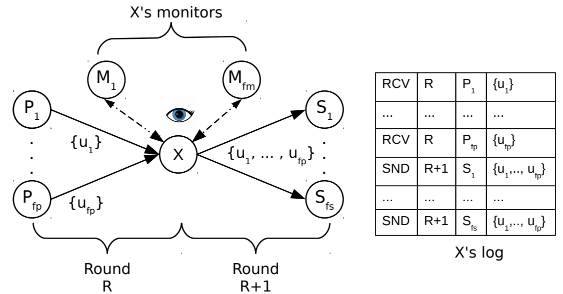


FIGURE 4.2 – Accountable gossip

However, as presented in various studies (e.g., [et al., 2000, Krishnan et al., 2004]), gossip-based dissemination suffers from nodes behaving selfishly. Selfish behaviour takes place when nodes tamper with their software or use tampered software in order to maximise their benefit (e.g., receiving the disseminated content as fast as possible) while minimising their contribution to the system (e.g., saving bandwidth or computational resources).

4.2.2 Accountability solutions

Accountability mechanisms (e.g., PeerReview [Haeberlen et al., 2007a], FullReview [Diarra et al., 2014], AVMs [Andreas*et al.*,]) are effective solutions to deter faults in distributed systems. These mechanisms have already been used as incentives for forcing selfish nodes to participate in gossip-based content sharing protocols (e.g., [Ben Mokhtar et al., 2014]). Figure 4.2 shows an accountable gossip protocol in which a node X logs its interactions with its predecessors and successors in a secure log (depicted in the right part of the figure). For example, the first line of this log specifies that node X received $\{u_1\}$ from node P_1 during round R. Secure logs can either rely on cryptography techniques (e.g., recursive hash functions in [Haeberlen et al., 2007a, Andreas*et al.*,]) or on secure hardware (as in [Levin et al., 2009]) to make them tamper evident and append only. In these systems, each node X is further assigned a set of monitors (depicted above X in the figure) that periodically audit its log in order to assess whether the logged entries correspond to a correct execution of the gossip protocol. For instance, in the figure each monitor can check that node X has forwarded all the updates it received during round R (i.e., $\{u_1, ..., u_{fp}\}$) to all its successors (i.e., $S_1, ..., S_{fs}$) during round R+1.

4.2.3 Privacy of users

A major drawback of accountability mechanisms is that nodes must share their interaction logs with their monitors. In gossip-based applications such as content sharing or live video streaming applications, this allows monitors to learn about nodes *interests* and thus possibly infer sensitive information about them. Indeed, various studies (e.g., [Zheleva and Getoor, 2009, Hu et al., 2011]) have shown that the consumed media can disclose information about individuals (e.g., gender, sexual, religious or political preferences). Further to learning nodes interests, this allows to learn the *interest graphs* between nodes sharing similar interests, thus possibly inferring private information about them (e.g., learning that a given user is a lesbian because it is interested in similar contents as a person known to be a lesbian).

4.3 System Model and Objectives

In this section, we present the assumptions we make in the rest of this chapter and the objectives of *PAG*.

Communications and cryptographic assumptions. As classically made in gossip-based protocols (e.g., in [Li *et al.*,] and [Li et al., 2008]) we structure time using rounds. Nodes are roughly synchronized, which allows them to check each others' periodical exchanges based on the specification of the exchange protocol. Nodes are uniquely identified with an integer identifier, for example deterministically computed using their IP addresses, and cannot generate multiple identities. Further, we assume that nodes can generate prime numbers, and have access to secure asymmetric key encryptions and signatures. We will denote $p_k(X)$ the public key of a node X, $\{m\}_X$ the encryption of a message m by node X, and $\langle m \rangle_X$ a message m along with its signature by node X.

Gossip sessions and monitoring infrastructure. We assume that several gossip sessions disseminating different contents can hold simultaneously in the system. Each content is generated and signed by its source. Updates are propagated along with their signature so that they can be verified by the nodes upon reception, which prevents data tampering. Nodes interested in a content have to obtain the public key of its source using an external service. We assume that a membership protocol (e.g., Fireflies [Johansen et al., 2006]) provides nodes with a set of successors and monitors that can be identified, for a given round, by each node in the system (as it has been done in [Li *et al.*, , Li et al., 2008, Ben Mokhtar et al., 2014]).

Nodes and adversary models. We consider several types of nodes. *Correct nodes* strictly follow the protocols. In particular, the source of each session is assumed to be correct. *Selfish nodes* are self-interested, and deviate from a protocol in any way that would improve their benefit (e.g., reduced bandwidth consumption or CPU overhead). We consider a *global and active opponent*, which is the strongest model of attacker. A global opponent can monitor and record the traffic on network links. Active means that it can control some nodes in the system and make them share information or deviate from the protocol (if possible) in order to reduce the privacy of other nodes. The only limitation of the global and active opponent is that it is not able to invert encryptions.

Objectives. To protect a gossip-based system from selfish deviations, we aim at enforcing the two following properties :

R_1 *Obligation to receive :* At a given communication round, a node must receive the updates sent by its predecessors that it never received.

R_2 *Obligation to forward :* A node must forward the updates it received at a given communication round R to all its successors during round R+1.

In addition, we aim at enforcing the following privacy property, which prevents an attacker from building interest graphs :

P_1 *Unlinkability between updates and nodes :* Suppose that node A sends an update u to node B. Other nodes than A and B should not be able to link A or B with u.

4.4 *PAG* in a Nutshell

In this section, we present two mechanisms composing *PAG* that when combined provide both accountability and privacy to dissemination protocols. We first present how nodes monitor each other to enforce

accountability in part 4.4.1. Then, we introduce the intuition of the cryptographic procedures that preserve privacy in part 4.4.2.

4.4.1 Enforcing accountability using a monitoring infrastructure

We use a log-less monitoring infrastructure, because maintaining the consistency of secure logs is costly in terms of exchanges and computations. In addition, logs, which are public, reveal too much information about nodes. The first key idea of this infrastructure is that the monitors of a node A learn which updates A receives from declarations of the node (which are then verified by its predecessors' monitors), and check that A's successors receive these updates (from their monitors). Second, message transmissions between monitors allow them to maintain the same information about the updates a node receives and has to retransmit. Finally, using classical techniques we handle omission failures.

In the following, let M(A), respectively M(B), be the set of monitors of node A, respectively node B. Figure 4.3 illustrates the situation where a node A that received an update u, forwards it to node B (message 1.) during round R. Upon reception of u, node B sends an acknowledgement to its monitors (message 3.) and to node A (message 2.). If node A does not receive an Ack, it emits an accusation against node B, which consists in sending to nodes in M(B) the update u, and making them forward it to node B and ask for an acknowledgement. If nodes in M(B) receive an Ack from node B, they transmit it to the nodes in M(A) using the $Confirm(\langle Ack(u, A)\rangle_B)$ message (message 4.), otherwise they send a Nack message to nodes in M(A). Using this message, nodes in M(A) can check that node A (i) contacted all its successors, and (ii) forwarded the right update. In the meanwhile, nodes in M(B) have learnt that node B received the update u, or that node B is unresponsive.

We now briefly explain why the best interest of nodes is to transmit each message represented in Figure 4.3. We assume that node A received update u, and that its monitors are informed of this reception. In addition, at least one node in each monitoring set is assumed to be correct. First, node A will correctly send u to node B, because sending accusations is more costly than sending a single update. Second, nodes in M(A) expect to receive either a $Confirm(\langle Ack(u, A, B)\rangle_B)$ message (which proves that node B received u), or a Nack message (if node B did not send a signed Ack to A). If nodes in M(A) receive none of these messages, they have to determine if node A did not send update u, or if node B did not send the acknowledgement to its monitors. To reach this goal, they ask node A for the acknowledgement that node B should have sent. If node A cannot exhibit this acknowledgement it is considered guilty because it did not accuse node B, otherwise node B is considered guilty.

To summarise, this monitoring infrastructure forces nodes to interact with their successors as well as to receive and forward updates. However, monitors are aware of all the transmitted messages. Our next goal is to hide this information.

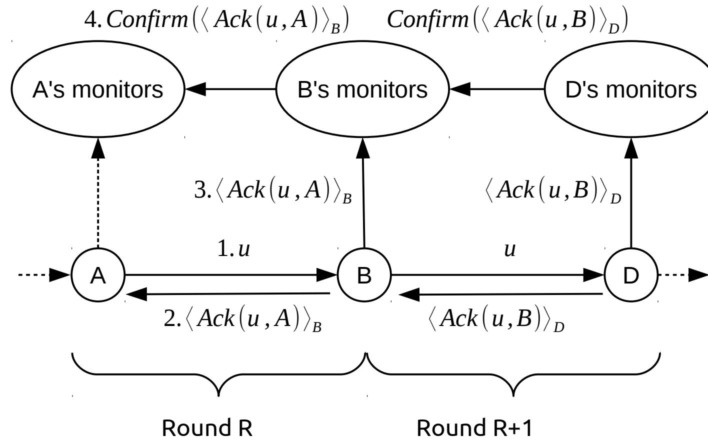


FIGURE 4.3 – Monitoring of nodes to ensure the forwarding of messages

4.4.2 Enforcing privacy using homomorphic hashes

A straightforward solution that one may think of to enforce privacy is to encrypt updates. However, doing so is not sufficient against a global and active opponent, because nodes which participate in the protocol would know the correspondence between a content and its encryption, and the monitors of a node would know which encrypted updates were received. To deal with this issue, we rely on homomorphic procedures that allow monitors to check that nodes forward updates and prevents them from learning which updates are exchanged.

Specifically, we use a hash function, noted H, based on an unpadded RSA encryption, that exploits two of its multiplicative properties. Its public key consists of a modulus M and an exponent p, then the hash of an update u is given by $H(u)_{(p,M)} = u^p \mod M$. Let u_1 and u_2 be two updates, and p_1 and p_2 be two exponents. The following homomorphic properties can be established :

$$H(u_1)_{(p,M)} \cdot H(u_2)_{(p,M)} = H(u_1 \cdot u_2)_{(p,M)} \quad H\left(H(u)_{(p_1,M)}\right)_{(p_2,M)} = H(u)_{(p_1 \cdot p_2,M)}$$

Any hash function verifying these two properties could be used to check the dissemination of messages. We will use a modulo size of 512 bits, as recommended in [Enisa, 2014]. Nodes cannot decrypt the hashed updates, as the value of the modulus M is smaller than the size of updates.

Figure 4.4 illustrates the intuition of the application of this hash to check retransmissions. In this figure, nodes A and F are two predecessors of node B, and node D is a successor of node B. We represent only two of the j predecessors of node B for the sake of simplicity, even though having 3 predecessors is a minimum to ensure privacy. We only focus on the reception and forwarding of the messages that node B receives. The same steps would also apply to the nodes A, F and D to secure each forwarding step.

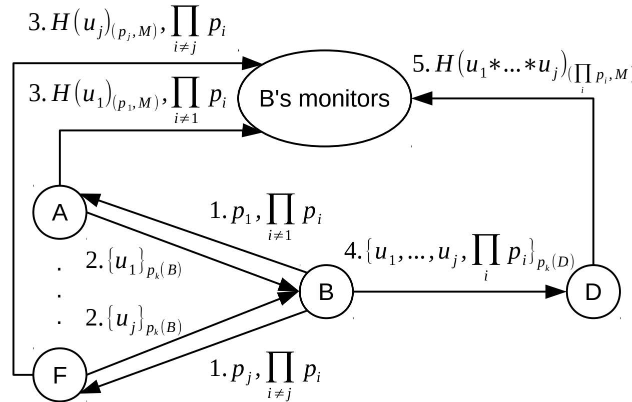


FIGURE 4.4 – Privacy preserving verification of a forwarding of a node B

Nodes A and F send updates u_1 and u_j to node B respectively. First, nodes A and F ask node B to send them a prime number. Node B chooses two prime numbers p_1 and p_j and respectively sends $(p_1, \prod_{i \neq 1} p_i)$ and $(p_j, \prod_{i \neq j} p_i)$ (messages 1.) to nodes A and F respectively. Then, nodes A and F send their two updates to node B encrypted with its public key (messages 2.). Nodes A and F declare (messages 3.) to the monitors of node B, that they sent updates to node B whose hashes are respectively equal to $H(u_1)_{(p_1,M)}$ and $H(u_j)_{(p_j,M)}$. Node B then forwards the updates u_1 and u_j to node D, and joins the product $\prod_i p_i$ (message 4.). Node D acknowledges the reception of u_1, ..., u_j using the hash value $H(u_1 * ... * u_j)_{(\prod_i p_i, M)}$ to the monitors of node B that verify that the following equation is verified :

$$\left(H(u_1)_{(p_1,M)}\right)^{\prod_{i \neq 1} p_i} * ... * \left(H(u_j)_{(p_j,M)}\right)^{\prod_{i \neq j} p_i} \mod M$$
$$= H(u_1 * ... * u_j)_{(\prod_i p_i, M)}$$

This short example shows that the monitors of a node are able to check that it forwards the updates it receives without learning the actual content. To break this privacy, an attacker would have to learn the prime numbers a node has chosen. With this information it would decrypt the exchanges a node had with its predecessors, or successors. In practice, predecessors and monitors of a node receive the product of prime numbers, and are not able to factorise it efficiently, as it is a notoriously known hard problem.

4.5 *PAG* Detailed Description

The monitoring infrastructure and the homomorphic hashes must be carefully combined so that selfish nodes can not deviate from the protocol without being detected. In this section, we detail the *PAG* protocol. We finally explain how the forwarding mechanism is adapted to build a gossip-based content dissemination protocol.

4.5.1 Transmission of updates between monitored nodes

Figure 4.5 presents the exchanges that occur when node A forwards a set S_A of updates to node B, which owns the set S_B of updates, during round number R. First, node A asks a prime number to node B that it will use to hash the product of the updates in S_A (message 1.). Node B generates one prime number p_i for each of its predecessors. We note $K(R, B) = \prod_i p_i$ the product of the prime numbers that node B chooses during round R to receive updates.

In message 2., node B replies with the primary key p_j in a message signed using its private key, and then encrypted using node A's public key. It also joins the homomorphic hashes $H(u_{i \in S_B})_{(p_j, M)}$ of the messages in S_B using p_j. Upon reception of this message, node A can check if the updates in S_A are not in S_B, and thus avoid to send them, as node B already owns them.

In message 3., node A serves in a message signed using its private key, and then encrypted using node B's public key, the updates in $S_A \setminus S_B$, that node B does not have, and K(R-1,A). The value of K(R-1,A) is the product of the prime numbers node A used to receive the updates in S_A from its predecessors during round R-1. Node B has to use this value to acknowledge the reception of updates using the hash function, in order for the monitors of node A to check its forwarding.

In message 4., node A sends to node B a signed attestation that declares the value of the hash of the product of the messages in S_A using p_j. This message will later be transmitted to the monitors of node B, which will then check the forwarding of node B based on its value.

In message 5., which is signed, node B acknowledges the reception of the messages in S_A using the hash of their product with $K(R - 1, A)$. If necessary, node A can later use this message as a proof that it did forward the right set of messages to node B during round R.

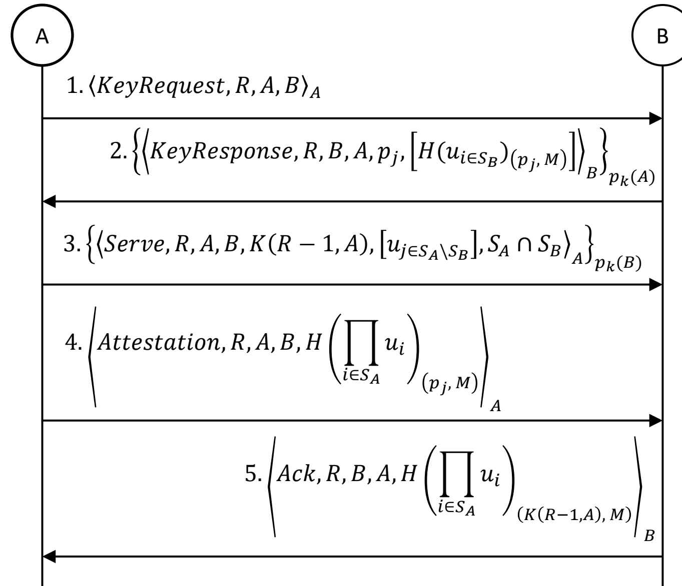


FIGURE 4.5 – Exchange of updates between nodes

4.5.2 Transmission of hashes to the monitoring infrastructure

Monitors check that the node they monitor (i) contacts all its successors, (ii) forwards all the messages it received at round R during round R+1. The first verification consists in checking the reception of messages from the monitors of each successor, which correspond to forwarded updates. For the second verification, monitors have to compute the homomorphic hash of the product of all the messages that the node receives during a given round, and check that its successors during the following round acknowledge this hash.

In practice, at each round, the monitors of a node expect to receive messages from it, and from the monitors of its successors. Figure 4.6 illustrates the mechanisms that allow the monitors to perform these tasks after node B has received the set S_A of updates from node A. In this figure, the monitors of node B are nodes A, D and G, and node B sends two messages to only one of its own monitors, to prevent monitors from receiving all the products of the prime numbers.

Message 6. is a copy of the acknowledgement that node B sent to node A in (message 5. of Figure 4.5), and message 7., which is signed, contains the attestation that node A sent in message 4. of Figure 4.5, and the product of the prime numbers that node B used to receive messages from its other predecessors during round R.

The monitor that receives these two messages, here node D, from node B computes the value
$\left(H\left(\prod_{i\in S_A} u_i\right)_{(p_j,M)}\right)^{\prod_{k\neq j} p_k} \mod M =$
$H\left(\prod_{i\in S_A} u_i\right)_{(\prod_k p_k,M)} = H\left(\prod_{i\in S_A} u_i\right)_{(K(R,B),M)}$
and broadcasts it to the other monitors of node B, along with message 6. To check that monitors correctly compute and forward the hashes of updates, nodes can compute this value and send it to their monitors. Monitors are then able to check each other's correctness.

4.5.3 Homomorphic combination of hashes by monitors

During a round, each monitor of node B computes the product of all the hash values forwarded by the other monitors of node B. At the end of the round, the monitors of node B know the homomorphic hash of the updates that node B received, computed using the product of the prime numbers that node B has chosen. This hash must then be acknowledged by the successors of node B during the following round.

Suppose that node B receives the set of messages S_A from node A, and the set of messages S_F from node F during a given round. Let $\prod_j p_j$ be the product of the prime numbers that node B used to receive these messages. The monitors of node B obtain the hash of the union of S_A and S_F applying the formula

$$H\left(S_A \cup S_F\right)_{(\prod_j p_j,M)} = H\left(S_A\right)_{(\prod_j p_j,M)} \times H\left(S_F\right)_{(\prod_j p_j,M)}$$

To allow the monitors of node A to obtain the right part of this formula, the monitors of node B have to forward them the acknowledgement (message 9.). The monitors of node A can then verify that node B received the correct set of messages from node A.

4.5.4 Application to a content-dissemination system.

In this section we present the important details or optimisations that allow the protocol to become practical when applied to a gossip-based dissemination protocol.

Buffermap transmissions. A node sends to its predecessors the hashes of a proportion of the messages it owns, in order to avoid multiple receptions. Determining how many hashes to send is dependent on the applications, and more particularly on the sizes of updates and of hashes. In our scenario, updates were bigger than their hashes, and the best results in terms of bandwidth consumptions were obtained when the updates of the last 4 rounds were hashed and transmitted.

Multiple receptions. While *PAG* avoids some multiple receptions of a same update, they can still occur when a node simultaneously receive updates from different predecessors. However, to limit the bandwidth consumption of nodes it is necessary to make them forward these updates only once. To do so, when a node sends an update it also joins to it an integer which describes the number of times it was received by the sending node during the previous round. This enables the receiving node to accurately

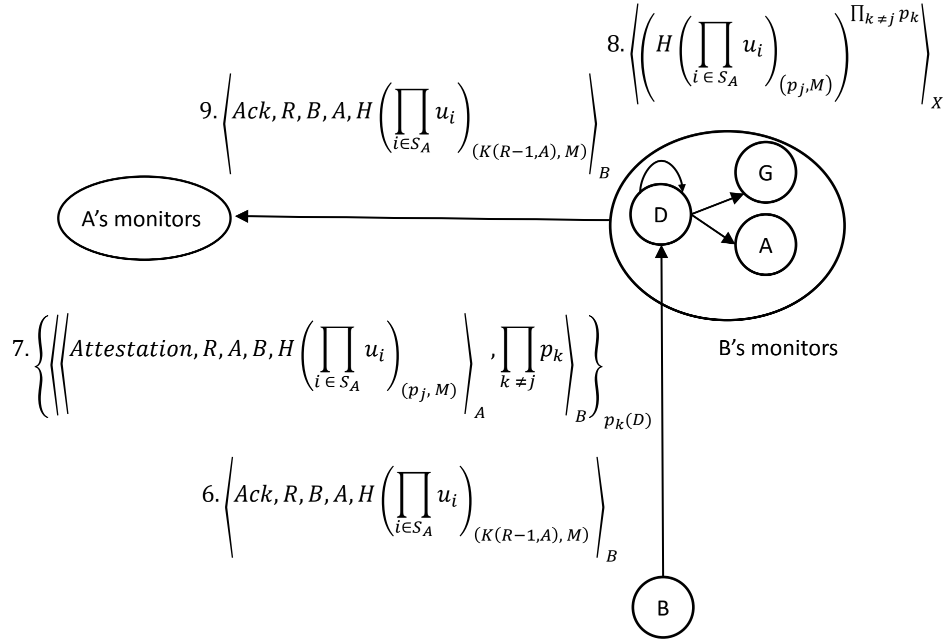


FIGURE 4.6 – Monitoring part of an interaction between two nodes

compute the hash of the set of received updates, and the monitors to match the hashes of received updates with the ones of forwarded messages.

Expiration of updates. In the context of live-streaming, updates have an expiration date after which nodes should not continue to forward them. Determining this expiration delay is up to the system designer. To allow updates to stop being propagated, when a node forwards them to another node, it separates the updates in two lists : the first one contains updates that will expire in the next round, and that should not be forwarded, while the other one contains updates that must be forwarded. A small modification of the messages and monitoring exchanges allow updates to expire. The monitors of a node acknowledge the reception of the first list and check the propagation of the second list.

4.6 Privacy and Accountability Analysis

In this section, we present the results of the security analysis we made using the cryptographic protocol verifier ProVerif (part 4.6.1). This proof shows that property P_1 holds against a global and active attacker if it controls less than f nodes, where f is the number of successors per node. We then briefly explain why the implementation of the forwarding mechanism provides accountability (part 4.6.2).

4.6.1 Privacy Guarantees

ProVerif [Blanchet, 2001] is an automatic cryptographic protocol verifier that uses Horn clauses to detect possible attacks. Using ProVerif, we modelled the cryptographic mechanisms of PAG[6]. This model shows that there is no attack on the privacy property P_1 that involves less than f nodes, where f is the number of predecessors, successors and monitors per node.

We consider the representative situation where a node B, assumed to be correct, receives updates from three predecessors A_1, A_2 and A_3, and has to forward them to one of its successors C. For each node, we instantiated a set of monitors. The case where f=3 is the simplest where the protocol can be proved secure. Increasing the value of f reinforces the security of the protocol, as the necessary number of colluding nodes sharing information in order to break the privacy also increases. The aim of an attacker is to obtain the value of a prime number that node B chooses in order to obtain the detail of the exchange

6. The code is available at https ://github.com/jdecouchant/PAG

between node B and this node. For attacks to be feasible, we assume that the attacker has access to the list of updates that node B may have received from its predecessor. In order to find the updates that B received, the attacker would have to hash any possible combination of updates using the prime number and see if it is equal to the observation. This attack is not really practical because the number of subsets of a set of size N is equal to 2^N.

We modelled several attack scenarios to assess the privacy property P_1. These scenarios can be grouped under two cases :

— **Case (1).** The attacker listens all communications on the network, and tries to break the privacy of exchanges between nodes A_1 and B. The attacker can replay, or inject messages in the network.

— **Case (2).** In addition to the assumptions of case 1., we consider that at most (f-1) nodes among the monitors or predecessors of a node are part of a coalition. This case can be instantiated with several configurations (e.g., (f-2) monitors and 1 predecessor, (f-3) monitors and 2 predecessors, etc.) that were all tested in our configuration.

In case (1), ProVerif proves that no attack exists on the cryptographic procedures of *PAG*. The experiments in case (2) confirm that no attacks exist if the opponent controls less than f nodes. An attack is possible if f nodes collude among the monitors or predecessors of a node, and ProVerif found it. In this case, the opponent is able to obtain the prime numbers that B generated and thus learn the updates node B received.

4.6.2 Accountability Analysis

We present in this section a sketch of the incentives that enforce properties R_1 and R_2.

Let us consider the exchanges depicted in Figure 4.5. In the following, we briefly explain the incentives that force a selfish node, say node A, to follow the steps depicted in this figure. Remember that nodes register the messages they send or receive, and can use them to prove their correctness or that another node deviated.

Node A computes the set of updates that its successor does not have (which enforces R_1) and send them, along with the identifiers of the updates its successor already have (message 3). If a node does not send the right set of updates to its successors then the verification its monitors run will fail. Eventually, as its successors receive signed messages that they can exhibit, it will be proved guilty. The attestation (message 4) that node A sends can be verified by node B, thus a selfish node will correctly compute its value. In return, the acknowledgement (message 5) that node B sends can also be verified by node A. This acknowledgement forces node B to inform its monitors about the updates it received from node A (messages 6 and 7 of figure 4.6). Finally, if the verifications of node A pass then it means that it forwarded the right set of updates to the right nodes. After having received these updates, node B is engaged towards its own monitors to continue the forwarding of updates, which enforces property R_2.

4.7 Performance Evaluation

In this section, we present the performance evaluation of *PAG*. We start by introducing our methodology and the values of the protocol's parameters (part 4.7.1). We then evaluate the overhead of *PAG* in terms of bandwidth consumption using both simulations and real code deployments compared to state-of-the art competitors while varying the size of the content being disseminated (parts 4.7.2). Further, we evaluate the cryptographic costs of *PAG* (part 4.7.3) as well as its scalability with respect to the number of users (part 6.6.5). We finally evaluate the proportion of exchanges that an active and global attacker may discover if it controls more than f nodes in the system (part 4.7.5).

Overall, our evaluation shows that *PAG* is more costly than existing accountable gossip protocols which do not preserve privacy. Yet, contrary to accountable anonymous communication protocols, its performance is compatible with streaming live video content on commodity Internet connections. Furthermore, *PAG*'s cryptographic overhead can be handled by modern architectures and thanks to its inherited gossip properties, its bandwidth overhead scales logarithmically with the number of nodes in the system. Finally, *PAG* improves the resilience to active and global opponents compared to state of the art protocols.

4.7.1 Methodology and Parameter Settings

To assess the performance of *PAG*, we implemented it in Java and used it as a video live streaming application. When it is not specified, *PAG* is configured with the same numbers of successors and monitors per node (e.g., 3 when the system contains 1000 nodes). In this context, a source node diffuses a video stream at a fixed rate and sends each update to random successors. Updates are then disseminated using *PAG* or one of the protocols we compare *PAG* with. Among these protocols are an accountable gossip protocol, and two anonymous communications protocols. ***AcTing*** [Ben Mokhtar et al., 2014] is an accountable gossip protocol that does not preserve the privacy of nodes as nodes maintain a secure log, and audit each other. **RAC** [Ben Mokhtar et al., 2013] is an anonymous communication system that forces nodes to relay the messages that other nodes send. Using RAC, a source could send a content to all nodes anonymously while enforcing accountability. We do not study Dissent [Corrigan-Gibbs and Ford, 2010] at it was shown to be even more costly than RAC in [Ben Mokhtar et al., 2013].

Real deployment settings. We deployed *PAG* on 48 machines of the Grid5000 cluster, interconnected using a 1Gb/s network, and using 9 instances per machine, thus totalling 432 nodes. Each machine contains a 4-cores Intel Xeon L5420 processor clocked at 2.5Ghz with 32GB of RAM. A source groups packets in windows of 40 packets. The duration of one round is set to one second, and updates of 938B are released 10 seconds before being consumed by the nodes' media player. Signatures are generated using RSA-2048. The sizes of the generated prime numbers is set to 512 bits. The modulus used in the homomorphic hashes is 512 bits long.

Simulations settings. We used the OMNeT++ [omn, 2013] simulator, and ran a C++ implementation of *PAG* using the same parameters value we used in our deployment. We also computed the scalability of the protocol when the number of nodes was too high to be simulated.

4.7.2 Comparisons with existing protocols

We first compare the bandwidth consumption of *PAG* to the one of *AcTing* [Ben Mokhtar et al., 2014]. Figure 4.7 presents the cumulative distribution functions of the bandwidth consumptions of nodes during a 300Kbps streaming session. In average, nodes running *AcTing* consume 460 Kbps, while they consume 1050Kbps using *PAG*. This additional cost comes from the forwarding policy : nodes must receive, at

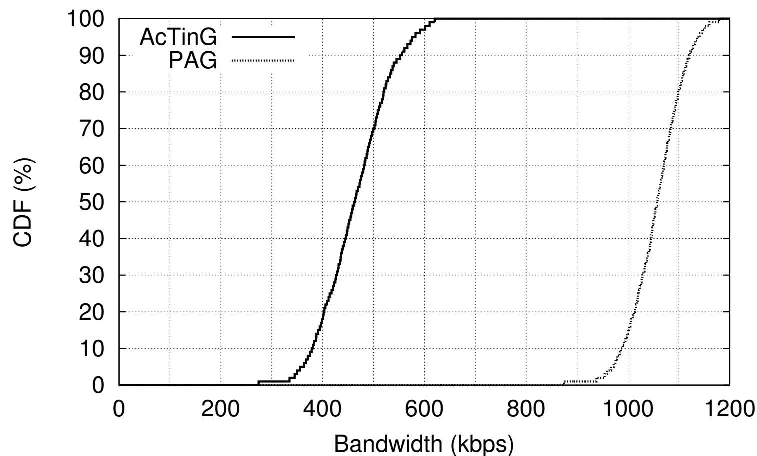


FIGURE 4.7 – Bandwidth consumption with a 300 kbps stream and 3 monitors

least through hashes, the updates of their predecessors. Hence, a given node may have to forward several times a given update to its successors. *AcTing* is less costly because nodes can refuse updates, and it is then controlled using their log during audits. Increasing the number of monitors does not significantly increase the bandwidth cost of the protocol, because the messages transmitted between and to monitors are small, and allows a better resilience to collective deviations between nodes.

Relying on anonymous communication systems to run a gossip protocol would enforce privacy. However, these protocols are costly and can not scale like *PAG* and *AcTing*. We thus designed a second set of experiments. The first two lines of Table 4.1 present the video qualities we considered and the associated payload size. Table 4.2 details the results we obtained with 1000 nodes. For each network capacity, ranging from 1.5Mbps to 10Gbps, we study the maximum video quality that each protocol can provide, and the amount of bandwidth that is used. For example, with 1.5Mbps network links *AcTing* provides a 480p video using 1.4Mbps. *PAG* is more costly than *AcTing* which is also accountable but not privacy preserving. Using 10Mbps network links, *PAG* provides at most a 480p video, consuming 6.9Mbps of bandwidth. In comparison, *AcTing* is able to send a 1080p video using 6Mbps of bandwidth.

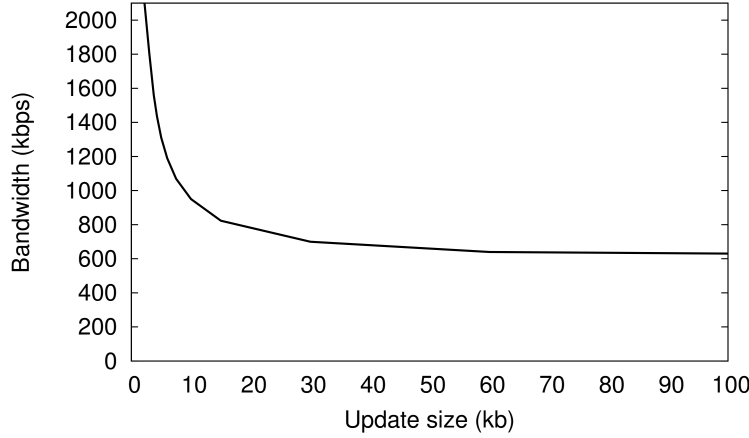


Figure 4.8 – Bandwidth consumption with 1000 nodes and a 300Kbps stream in function of the size of updates [sim]

However, using anonymous communication systems would be much more costly. The maximum payload that RAC is able to provide using 10Gbps network links is equal to 63kpbs, which is far from the minimum of 300Kbps that a basic streaming session would require.

Video quality	144p	240p	360p	480p	720p	1080p
Payload size (Kbps)	80	300	750	1000	2500	4500
RSA signatures	33	33	33	33	33	33
Hashes	133	475	1170	1560	3934	7200

Table 4.1 – Number of RSA signatures and homomorphic hashes per second in a system of 1000 nodes [sim]

	Prv.	Acc.	**1.5Mbps ADSL Lite**	**10Mbps Ethernet**	**100Mbps Fast Ether.**	**1 Gbps GB Ether.**	**10Gbps 10 GB Ether.**
PAG	✓	✓	660 Kbps	6.9 Mbps	31 Mbps	31 Mbps	31 Mbps
AcTing	×	✓	1.4 Mbps	6 Mbps	6 Mbps	6 Mbps	6 Mbps
RAC	✓	✓	∅	∅	∅	∅	∅

Table 4.2 – Maximum video quality sustainable in function of the network links capacity, and the associated bandwidth consumption, in a system with 1000 nodes

Impact of updates size. Although we used 938B updates in the previous experiments, as was done in [Li *et al.*, , Guerraoui et al., 2010a], Figure 4.8 shows that using bigger updates can further decrease

the bandwidth consumption of *PAG*. This is due to the fact that more content can be represented under each hash. For example, nodes propagating 10Kb updates needed to perform 370 homomorphic hashes per second, while propagating 100Kb updates decreased this number to 52 hashes per second. In addition, using larger updates also enables the CPU overhead to decrease, as hashes are computed modulo M.

4.7.3 Cryptographic costs

PAG relies on cryptographic mechanisms, which dominate its CPU cost. To evaluate this cost, we measured the number of generated RSA encryptions and homomorphic hashes per second rather than the CPU load, which depends on the hardware used. We measured these numbers depending on the video quality, and depicted the results in Table 4.1. The number of RSA signatures is always equal to 33, as it depends on the number of messages generated by the protocol, while the number of homomorphic hashes performed depends on the video quality, and more precisely on the number of 938B updates. Using openssl, we measured that each core of the machines we used is able to perform 4800 hashes per second with a 512-bits modulus. Thus using a single core to compute homomorphic hashes is enough to obtain a video quality up to 720p using a 512 bits modulus, which would generate 3924 hashes per second. Using more cores would provide enough cryptographic power to support better video qualities. In addition, using a 256 bits modulus can also be considered secure enough in many situations, and it would significantly reduce the bandwidth overhead of the protocol. Overall, we believe that *PAG* can be used by a wide range of commodity hardware.

4.7.4 Scalability

In this experiment we increase the number of nodes in the system and measure their bandwidth consumption. The bandwidth scalability of *PAG* comes from its gossip nature, as in a system of N nodes, each user has $log(N)$ successors. Figure 4.9 presents the bandwidth consumption of *AcTing* and *PAG* depending on the system size when a 300Kbps video stream is disseminated. With a million nodes *PAG* consumes 2.5Mbps, while *AcTing* needs 840Kbps. In these conditions, *PAG* is able to provide nodes with the full 300Kbps stream, while consuming less bandwidth than anonymous communication systems, which are too costly to be used.

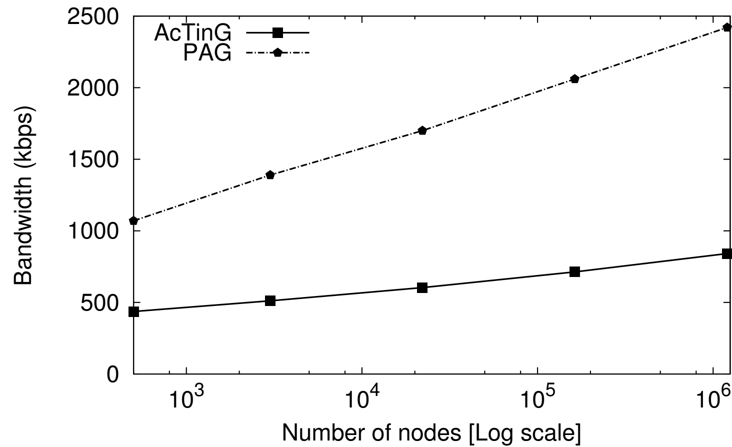


FIGURE 4.9 – Scalability of *PAG* and *AcTing* with a 300Kbps content [sim]

4.7.5 Probabilistic study of the impact of coalitions on privacy

As proved using ProVerif, a coalition of at least f nodes can obtain the prime numbers used in some nodes' interactions. We now evaluate the privacy leakage performed by a global and active attacker that would control more than f nodes in *PAG*. We also perform this attack on an existing state of the art protocol, i.e., *AcTing* [Ben Mokhtar et al., 2014]. In AcTinG, monitors probabilistically audits nodes' secure logs,

which contain the detail of past interactions. Differently, in *PAG*, it is possible to discover the details of the interactions of a node if all its predecessors except at most two and at least one of the monitors of this node collude. This essentially means that collecting the prime numbers a node used, and observing all its encrypted interactions is enough to understand them. The success of attacks is evaluated in terms of probabilities as nodes are randomly affected predecessors, successors, and monitors. We evaluate the probability that an exchange between two nodes is discovered in function of the size of the coalition.

In Figure 4.10, we evaluate the proportion of exchanges that an attacker controlling a variable proportion of the membership can discover. The lowest possible proportion in this case is represented in plain black, and expresses the probability that at least one of the two nodes that interact is corrupted. As this figure shows, increasing the number of monitors, and the number of nodes in the system, makes the privacy guarantees of *PAG* close to ideal, while all interactions are discovered when an attacker controls 10% of nodes in *AcTing*.

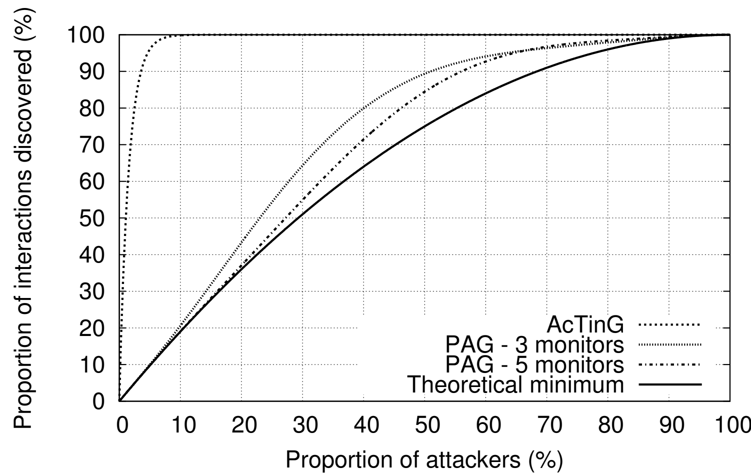


FIGURE 4.10 – Resiliency against a global and active attacker

4.8 Related Works

In this section we describe how existing solutions relate to the problem we solve in this chapter.

Selfish resilient gossip protocols. BAR Gossip [Li *et al.*,] and FlightPath [Li et al., 2008] are streaming protocols that handle both selfish and Byzantine deviations through deterministic interactions. However, peers are not forced to initiate exchanges. Differently, LiFTinG [Guerraoui et al., 2010a] uses audits to control that nodes forward the updates they receive. However, it relies on statistical properties that may produce up to 12% of false-positives, and false-negatives, and reveal the detail of the interactions of nodes. **Anonymous communications.** The first anonymous communication protocols, DC-Net [Waidner and Pfitzmann, 1989] and Onion Routing [Goldschlag et al., 1999], focused on enabling the strongest possible anonymity level for the former, and on providing practical performance for the latter. However, they take the participation of nodes for granted. More recently, Dissent [Corrigan-Gibbs and Ford, 2010] and RAC [Ben Mokhtar et al., 2013] force nodes to execute their role of relay. However, using these systems can be seen as a performance overkill. For example, for each message sent anonymously, Dissent [Corrigan-Gibbs and Ford, 2010] uses trusted nodes which receive anonymous communication requests and runs a protocol involving all-to-all communications.

Zero-Knowledge proofs. Classical zero-knowledge proofs can not always be applied because they are designed to verify functions that has a fixed number of inputs, whereas in many distributed systems both the size and the number of a node's "inputs" (the messages it has received from other nodes) are not known. In particular, in gossip, the quantity of messages a node receives during a time interval is not predictable.

Accountability approaches. Software-based accountability approaches, which include PeerReview [Haeberlen et al., 2007a], and A2M [Chun et al., 2007], make nodes maintain a secure log, and audit each

other. Audits transfer logs, and reveal detailed information about nodes. Similarly, hardware-based accountability approaches [Chun et al., 2007, Levin et al., 2009] rely on a trusted hardware to maintain secure logs. For example, PeerReview associates each node with a set of monitors, which verify it using the secure log. In our context, these applications would allow an attacker to track the propagation of an update among a membership, even if its encrypted.

Privacy-preserving accountability. In [Papadimitriou et al., 2013], nodes maintain a Merkle Hash Tree in which the leaves represent all the possible states of a node, and regularly publish its root hash value. Nodes that interacted with a given node are then able to collectively, but anonymously, check its state. This approach has been applied to a BGP routing system [Zhao et al., 2012]. However elegant, we believe that this approach cannot be applied to gossip protocols, as it is not possible to concisely represent all the possible states of a node.

Virtual currency. In [Belenkiy et al., 2007], a virtual currency approach is shown to provide accountability without compromising privacy in a peer-to-peer system. However, contrary to *PAG*, which is fully-decentralised, this solution requires two trusted entities that have access to nodes' information : i) a bank, which maintain an account for each user and knows about all transactions in the system ; and ii) the arbiter, which ensures the fair exchange of e-cash for data.

4.9 Conclusion

A number of gossip-based content dissemination protocols tolerating selfish behaviours have been proposed in the past. However, they do not preserve the privacy of users. On the other side of the spectrum, accountable anonymous communication protocols are too costly to be used to disseminate live multimedia content. In this chapter, we have presented the first protocol that enforces accountability through a monitoring infrastructure and still preserves the privacy of users thanks to homomorphic cryptographic procedures. Performance evaluation of *PAG* combining both a real deployment and simulations has demonstrated that its bandwidth consumption is compatible with streaming live content and that the partial privacy of nodes is close to optimal, even in presence of a global and active attacker. We have also shown that the reasonable cryptographic overhead of *PAG* makes it accessible to modern architectures, and that it exhibits very desirable scalability properties with a logarithmic growth of bandwidth consumption, comparable to standard gossip-based protocols. The privacy of nodes could be further enhanced if even the direct neighbors of nodes could not determine the media content they are interested in. Doing so in a peer-to-peer system is challenging, and future works include the design of a dissemination protocol that would improve on the obfuscation approach, which hide the interests of nodes by making them receive several contents at the same time.

Deuxième partie

Dealing with Rational Faults

Chapitre 5

FireSpam : Spam Resilient Gossiping in the BAR Model

Gossip protocols are an efficient and reliable way to disseminate information. These protocols have nevertheless a drawback : they are unable to limit the dissemination of spam messages. Indeed, messages are redundantly disseminated in the network and it is enough that a small subset of nodes forward spam messages to have them received by a majority of nodes.

In this chapter, we present *FireSpam*, a gossiping protocol that is able to limit spam dissemination. *FireSpam* organizes nodes in a ladder topology, where nodes highly capable to filter spam are at the top of the ladder, whereas nodes with a low spam filtering capability are at the bottom of the ladder. Messages are disseminated from the bottom of the ladder to its top. The ladder does thus act as a progressive spam filter.

In order to make it usable in practice, we designed *FireSpam* in the BAR model. This model takes into account selfish and malicious behaviors. We evaluate *FireSpam* using simulations. We show that it drastically limits the dissemination of spam messages, while still ensuring reliable dissemination of good messages. This work has been carried out in the context of the PhD thesis of Alessio Pace and has been published in the proceedings of the 29th IEEE Symposium on Reliable Distributed Systems (IEEE SRDS'10).

5.1 Introduction

P2P systems are now commonplace. Popular examples of P2P systems comprise collaboration platforms (e.g., instant messaging systems such as ICQ, Jabber, Skype, or Internet forums), content distribution systems (e.g., Napster, Kazaa, Freenet), etc. In this chapter, we focus on communication and collaboration platforms such as Internet forums. In such platforms, an appealing way for disseminating content produced by users is to use gossip-based broadcasting protocols [Eugster et al., 2001, Ganesh et al., 2002, Kermarrec and Ganesh, 2006, Eugster and Guerraoui, 2001, Lin and Marzullo, 1999, van Renesse and Birman, 2002]. Roughly speaking, gossip-based dissemination protocols work as follows : each node forwards all the messages it receives to a randomly chosen subset of nodes. The advantages of gossip-based dissemination protocols are that they are simple to design, and yet allow fast and reliable dissemination of messages to large networks of nodes.

Nevertheless, as it has been observed in [Gavidia et al., 2007], gossip-based disseminations protocols are ideal vectors to disseminate spam messages. By spam message we refer to a message having unappropriate (i.e. junk) content, and we do not consider spam a duplicate legitimate message due to the redundancy of the gossiping protocol itself. Indeed, simple strategies consisting in having each node locally filter the messages it detects as spam do not work. The reason is that gossip-based protocols are highly redundant and random, i.e. each node receives messages multiple times from different nodes. Consequently, as we show in Section 9.6, it is enough that a small subset of nodes do not filter a spam message to have it received by most nodes in the network.

In this chapter, we present *FireSpam*, the first gossiping protocol that is able to limit spam dissemination in the presence of rational and Byzantine nodes. In *FireSpam*, nodes are organized in a ladder topology according to their capability to filter spam : nodes having a high (resp. low) filtering capability are located at the top (resp. bottom) of the ladder. Messages are disseminated from the bottom to the top of the ladder, which acts as a progressive spam filter. The rational behind this topology is that nodes that actively filter spam (i.e., those with a high filtering capability) progressively climb the ladder and will eventually be less overwhelmed by spam messages. We actually show in Section 9.6 that the ladder drastically limits the dissemination of spam messages.

While organizing nodes in a ladder topology can easily be achieved if all nodes in the system behave correctly, it appears challenging to build such a topology in the presence of nodes acting selfishly or maliciously. Nodes with such behaviors are common in P2P systems and they do thus need to be taken into account when designing *FireSpam* for it to be usable in practice. Consequently, we have designed *FireSpam* considering the BAR model [Aiyer et al., 2005]. This model states that there are three kinds of nodes : *altruistic* nodes that strictly follow the protocol, *Byzantine* nodes that can behave arbitrarily, and *rational* nodes that are willing to deviate from the protocol if there is a gain in doing so.

In order to tolerate rational nodes, *FireSpam* encompasses a set of incentive-compatible mechanisms that make the protocol a strict Nash-equilibrium [Nash, 1951]. More precisely, the incentive mechanisms used in *FireSpam* ensure that it is in a rational node's best interest to always follow the protocol. Moreover, *FireSpam* encompasses a set of mechanisms guaranteeing that Byzantine nodes are detected and evicted from the system. These mechanisms assume a known upper bound on the number of Byzantine nodes in the system.

We have assessed the robustness of *FireSpam* both theoretically and practically. From a theoretical point of view, we prove that *FireSpam* is a strict Nash-equilibrium. From a practical point of view, we have assessed *FireSpam* through an extensive simulation study that shows that (i) it reliably delivers good messages, (ii) it drastically limits the dissemination of spam messages, and (iii) it cannot be harmed by a set of Byzantine nodes colluding to break the ladder topology.

The remaining of the paper is structured as follows : we present in Section 6.2 the system model. Follows a description of the design of *FireSpam* in Section 9.4. We analyze the robustness of *FireSpam* in Section 5.4 and present the performance evaluation in Section 9.6. Finally, we analyze related research efforts in Section 9.7 and conclude the paper in Section 9.8.

5.2 System Model

In this section, we present the system model, which decomposes into a message model, a fault model as well as a set of system assumptions.

Messages. Nodes can generate two types of messages : *good* messages and *spam* messages. *Good* messages are of interest to all the nodes and must be reliably disseminated in the network. Instead, *spam* messages should be filtered during the dissemination process in order to reach as few nodes as possible.

Node filtering capability. Each node in FireSpam has a *Spam Filtering Capability* (also called pollution awareness in [Lee et al.,]) that expresses the ability of a node to detect *spam* messages. We assume that messages assessed by nodes to be *good* or *spam* effectively fall into that category with a very high probability, i.e., nodes do very few (i.e, less than 5%) false positive or false negative message classification (i.e., *spam* messages classified as *good* and *good* messages classified as *spam*, respectively).

Fault model. We consider the BAR model, in which nodes can be Byzantine, altruistic or rational. *Altruistic* nodes follow the protocol exactly. They can only fail by crashing. On the other side of the spectrum, *Byzantine* nodes can take arbitrary decisions (e.g., dropping *good* messages). They can deviate from the protocol for any reason (e.g., a failure, a bug, a threat) and can collude with each other. Finally, *rational* nodes aim at maximising their benefit according to a known utility function. We suppose that rational nodes join and remain in the system for a long time and seek a long-term benefit. Moreover, rational nodes do not collude and assume that other nodes are altruistic. A rational node can deviate from

the protocol if the generated utility increases accordingly (e.g., they can decide not to forward messages for saving bandwidth). In this chapter the benefit is proportional to the amount of *good* messages received and is conversely proportional to the amount of *spam* messages received as well as to the amount of bandwidth consumed by receiving/forwarding messages from/to other nodes (respectively). Specifically, this benefit can be represented along the following axes :

1. (G) Receiving as much as possible (possibly, all) of the *good* messages disseminated in the system,
2. (S) Receiving as little as possible (possibly, none) of the *spam* messages disseminated in the system,
3. (F) Forwarding as little as possible (possibly, none) of the received *good* messages.

We can informally define the overall benefit as :

$$B = \alpha G + \beta S + \gamma F \tag{5.1}$$

where $\alpha \gg \beta \gg \gamma$, intuitively meaning that nodes do not want to trade-off the reliable reception of *good* messages to receive less spam or consume less bandwidth. Further, for a node it is of more benefit receiving less spam, rather than saving some bandwidth.

System assumptions. We assume a cryptographic identification of nodes as it is assumed in many practical gossip-based content dissemination protocols (e.g., [Malkhi et al., 2001]). Each message sent in the network is therefore signed using the sender's cryptographic key. Furthermore, we assume that non-byzantine nodes maintain clocks synchronised within δ seconds and communicate over reliable links. Moreover, we assume that messages sent by a sender to a given receiver are always received within a bounded time.

5.3 FireSpam Design

In this section, we present *FireSpam*, a spam resilient gossiping protocol. As explained in the introduction, *FireSpam* organizes nodes in a ladder topology. Moreover, it uses a set of mechanisms to tolerate both rational and Byzantine behaviors. We start the section by a description of the ladder topology underlying *FireSpam*. We then describe the challenges raised by the construction of the ladder. Finally, we detail the *FireSpam* protocol.

5.3.1 Ladder topology

In *FireSpam*, nodes are organized according to their filtering capability in a ladder topology. Nodes at the bottom of the ladder have the lowest filtering capability, whereas nodes at the top of ladder have the highest filtering capability. Messages produced by a node p are first sent to a set of nodes located at the bottom of the ladder (called the *Publication View* of p). When a node q receives a message, it decides whether to filter it (when it considers it as a *spam* message) or to forward it to a set of nodes that are surrounding it in the ladder (called the *Forwarding View* of q). Good messages will thus eventually reach all nodes in the ladder, whereas spam messages will be progressively filtered by nodes along the ladder. The rational behind this ladder topology is that nodes that actively filter spam (i.e., have a high filtering capability), will progressively climb the ladder and be located at the top of it. Consequently, the latter will receive less spam than nodes with a lower filtering capability. Nodes are thus rewarded to filter spam.

Figure 5.1 shows an example of message dissemination in *FireSpam*. In this figure, a node p generates a spam message m. Node p sends m to its publication view (i.e. the three nodes at the bottom of the ladder). Nodes that are able to filter m are represented with double circles in the figure. Note that as nodes are organised in the ladder according to their filtering capability, it is more likely to find nodes able to filter m at the middle and top of the ladder than at the bottom of it. Nodes that do not filter m (i.e. nodes with a single circle), forward it to nodes in their forwarding view. On the depicted example, we observe that m is eventually filtered as it progressively reaches nodes that are able to filter m.

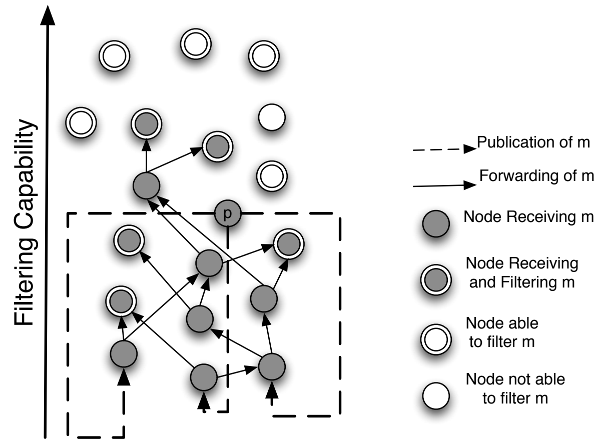


FIGURE 5.1 – *FireSpam* ladder topology.

5.3.2 Ladder construction challenges

While organizing nodes in a ladder topology can easily be achieved if all nodes in the system behave correctly, it appears challenging to build such a topology in the presence of rational and Byzantine nodes. In particular, for *FireSpam* to function in the presence of rational and Byzantine nodes we need to insure :

- **Reliable dissemination :** *good* messages must be forwarded by all nodes. Consequently, rational and Byzantine nodes should be discouraged or punished when dropping *good* messages.
- **Correct node evaluation :** in order for the ladder to have a correct topology, it is crucial that the filtering capability of nodes be regularly and correctly assessed. For instance, rational and Byzantine nodes should not have ways to forge filtering capability assessment.
- **Correct view assignment :** the publication and forwarding views of nodes must be correctly assigned (i.e., according to their filtering capability). For instance, rational and Byzantine nodes should not be able to forge view assignments in order to take a larger benefit from the protocol (e.g., choosing as neighbors nodes with a high filtering capability).

In order to deal with the above challenges, *FireSpam* decomposes in a set of mechanisms enabling the reliable construction of the overlay and the reliable dissemination of messages.

In order to enable the reliable dissemination of *good* messages, nodes classify messages in three categories. The *good* and *spam* categories are used when the node can assess with evidence that the message is a *good* or a *spam* message, respectively. Instead, the *undetermined* category is used when the node is not able to assess whether the message is a spam or a good message. This classification allows detecting nodes that intentionally filter good messages. Indeed, a node is expected to send all messages that it classifies in the *good* and *undetermined* categories. Consequently, if a node does not receive a message m it classified as *good* from a node p it is in the forwarding view of, it will conclude that p intentionally filtered m. As we will see, in that case, it will take actions to punish p.

Furthermore, to enable correct node evaluation and correct view assignment, a node is deterministically assigned a set of *node monitors*. The reason why *FireSpam* relies on node monitors for node evaluation and view assignment is the following : a node can obviously not self-assess its filtering capability. Otherwise, rational nodes could simply claim they have the highest possible filtering capability. An intuitive idea would be to rely on nodes that are in the forwarding view of the node to be assessed. Indeed, these nodes receive all messages sent by p and seem thus to be good candidates to evaluate its capability to filter spam. Nevertheless, these nodes could act rationally by forging the evaluation of nodes with high filtering capabilities in order to keep them close by in the ladder and benefit from their filtering capability. It is thus necessary to rely on third party node monitors and to define incentive mechanisms guaranteeing that these node monitors will behave correctly. Node monitors of a node n are actually responsible for the following tasks :

— Assessing the filtering capability of n.
— Assigning the publication and forwarding views of n.
— Detecting any misbehaviour of n and possibly blacklisting and evicting it from the system.
— Detecting any misbehaviour of other node monitors of n and possibly evicting them.

Node monitors for every node in the system are dynamically allocated using the Fireflies overlay network [Johansen et al., 2015]. Fireflies relies on a multiple ring topology and allows associating to each node a set of nodes (node monitors in the case of *FireSpam*) such that a majority of them are non-Byzantine. Note that node monitors also use the broadcasting service provided by Fireflies. This latter ensures that messages are reliably delivered despite the presence of Byzantine nodes, and thus within a bounded time [7].

5.3.3 *FireSpam* description

FireSpam is decomposed in a set of steps that are depicted in Fig. 5.2. We describe them below. The incentive mechanisms that are used to encourage rational nodes to follow the protocol are described in Section 5.4.

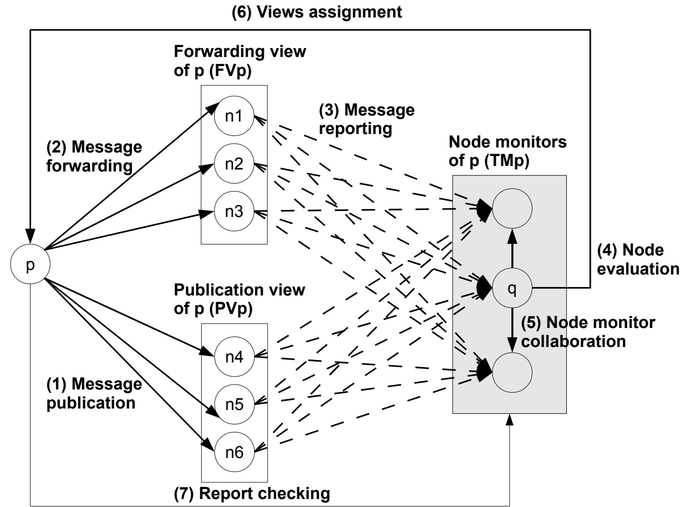


FIGURE 5.2 – *FireSpam* roles.

(1) Message publication. To publish a message m, node p sends it to all the nodes in its publication view, i.e., nodes n_4, n_5, and n_6.

(2) Message forwarding. When node p receives a message m, it forwards it to all nodes in its forwarding view, i.e., nodes n_1, n_2, and n_3.

(3) Message reporting. When nodes receive a message from p, they send a *report* to p's node monitors.

(4) Node evaluation. Node monitors use the reports sent by nodes in the forwarding view of p to assess its filtering capability. This is made possible by the fact that all messages are broadcast in the entire network. Consequently, it is possible for a node monitor to compare the filtering capability of two nodes by simply comparing the number of messages they forwarded during the same time interval : a node p forwards less (respectively, more) messages than a node q if it has a higher (respectively, lower) filtering

7. Timeouts can be dynamically set based on the monitoring of the current message delivery latency across the gossip-based platform.

capability than q. Recall that we make the assumption that a node only generates 5% false positive when assessing whether messages are spam or good. If a node was evaluating other nodes throughout the lifetime of a system, it could thus theoretically achieve a 5% precision on the evaluation of other nodes. In practice, however, nodes are evaluated on finite time intervals. To obtain a reasonable precision, we evaluate nodes on time intervals during which a large number of messages are broadcast (i.e. 1000 messages in the evaluation presented in Section 9.6).

In order to compute the number of messages actually forwarded by a node p, node monitors rely on *majorityReport*. These reports are sent by p's node monitors whenever they receive a majority of *reports* stating that p correctly forwarded a message m. Note that *majorityReports* are sent to all node monitors in the network in order to inform them that p has actually forwarded message m to a majority of nodes in its forwarding view.

On reception of the first *majorityReport* from a node monitor of node p, a node monitor updates the evaluation of p by incrementing by one the number of messages forwarded by p. Note that *reports* are also used by node monitors to detect nodes that do not forward *good* messages. Specifically, node monitors of a node p check that p correctly forwarded the messages that it considers *good*. If that is not the case, it considers that p has not followed the protocol.

(5) Node monitor collaboration. In order to insure that node monitors follow the protocol, each node monitor of node p is responsible for controlling whether other node monitors are behaving correctly or not. Specifically, each time a node monitor sends a message to other node monitors, it expects to receive a "similar" message from other node monitors (i.e. if it sends a *majorityReport* for the peer p, it expects the other monitors to send a *majorityReport* for p as well, possibly made with *report*s produced by different neighbors of p). The reason is that they follow a deterministic protocol that produces the same outputs provided they receive the same inputs. If a node monitor does not receive an expected message from one of the other node monitors, it collaborates with the other node monitors of p in order to evict it from the network (if a majority of node monitors are willing to do it).

(6) View assignment. Nodes belonging to the set of node monitors of node p periodically computes a forwarding view and a publication view for p. They then send the computed views to the other node monitors of p, which check their correctness. If a node monitor q does not send a view or sends a wrong view, the other node monitors can possibly evict q (if a majority of node monitors are willing to do it).

The computation of views is performed as follows : every node monitor knows the set of nodes in the system and their filtering capability (thanks to the *majorityReports* it receives). Views are then assigned using the same mechanism than the one used in Fireflies to assign predecessors to nodes [Johansen et al., 2015]. This mechanism allows deterministically associating to each node sets of $2t+1$ nodes, out of which there is a probability $\mathcal{P}$ that at most t nodes be Byzantine. The larger the set of nodes from which the $2t+1$ nodes are chosen, the higher the probability $\mathcal{P}$. In the remaining of this paper, we call *candidate set*, the set of nodes from which the $2t+1$ nodes are chosen.

Additionally, we impose that the forwarding and publication views of any node be disjoint from its node monitors set. This condition is necessary to ensure that node monitors have no incentive in not adhering to the protocol.

(7) Report checking. As rational nodes assume that other nodes are altruistic, a rational node may decide to save bandwidth by omitting to send *reports*. In order to avoid this behavior, nodes sending messages periodically query their node monitors to check that nodes in their forwarding view correctly reported about the messages they sent. Node monitors having received *reports* send them back to senders. Note that these replies have *fixed size* in order to encourage nodes to send the correct information (indeed there is no incentive in sending wrong information as this will consume the exact same amount of bandwidth).

5.4 FireSpam Robustness

Assessing the robustness of FireSpam against both Byzantine and rational behaviours has two requirements. First, the protocol properties (i.e., reliable dissemination, correct node evaluation and correct view assignment) need to be guaranteed even in the presence of a bounded number Byzantine nodes, provided that non-Byzantine nodes follow the protocol. Second, we need to demonstrate that it is in rational nodes' best interest to follow the protocol.

As shown in Section 9.4, *FireSpam* design choices guarantee its properties in the presence of a bounded amount of Byzantine nodes. Indeed, forwarding and publication views, as well as node monitors are assigned in a way that guarantees that they comprise a majority of non-Byzantine nodes.

In order to show that rational nodes follow the protocol, we prove in a game theoretic framework along the lines of works like [Aiyer et al., 2005], [Li *et al.*,] that *FireSpam* is a strict Nash-equilibrium [Nash, 1951]. Towards this purpose, we need to demonstrate that each part of the protocol (presented in Section 9.4) is a Nash equilibrium. Due to the lack of space, we briefly present in this section the incentive mechanisms provided in *FireSpam* to discourage nodes from deviating from the protocol. The complete proof can be found online [8].

(1) Incentives for message publication. A rational node r publishes a message m by sending it to all nodes in its publication view. In fact, it is in r's interest to have its message m reliably delivered into the system.

(2) Incentives for message forwarding. Upon the reception of a message m, not detected as *spam*, a rational node r always forwards m to all the nodes in its forwarding view FV_r. In fact, r forwards m otherwise it risks :

- To be blacklisted by a node p in FV_r to which r did not forward m. Specifically, if p receives a message from another node than r and classifies it as *good*, it will detect r's misbehaviour and blacklist it.
- To be blacklisted and suggested for eviction by a node p among its monitors M_r. Specifically, if p receives m and classifies it as *good*, it will expect the reception of a majority of *reports* from nodes in r's forwarding view. If p does not receive such reports, it will blacklist r and suggests to evict it. A majority of monitors suggesting for eviction of r will then cause r removal from the system.

(3) Incentives for message reporting. Upon reception of a message m from a node p, a rational node r will always report to all the monitors of p about the reception of m. In fact, p periodically checks on its node monitors and for each message it has forwarded, that all the nodes in its forwarding view sent a *report*. The node monitors reply to the sender with a *fixed message size* that contains the signed evidence of the reception of those *reports*. If r did not send a report, p will detect it and will consequently blacklist r.

(4) Incentives for correct node evaluation. At the heart of correct node evaluation is the majority report dissemination performed by monitoring nodes. Once a rational monitor r has collected a majority of reports about a monitored node p for a given message m, r will disseminate this information in the network. In fact, all p's node monitors (including r) have an equivalent majority report. Thus, as the dissemination in the monitor overlay is reliable [9], the other monitors can observe whether r has indeed disseminated the report or not. In this latter case, r will be blacklisted.

(5) Incentives for monitor collaboration. A rational monitor r of a node p always sends correct messages to other monitors of the same node. In fact, each time a node monitor sends a message to other node monitors, it expects to receive the same message from other node monitors (e.g., majority reports). Thus, If a node monitor does not receive an expected message from r, it collaborates with the other node monitors of p in order to evict r from the network.

8. FireSpam technical report : https ://sites.google.com/site/soniabm/
9. The reliability of message dissemination in the monitors overlay mesh is provided by Fireflies [Johansen et al., 2015].

(6) Incentives for correct view assignment. Similarly to the correct evaluation of nodes, biased view assignment can be very easily detected by node monitors as this process is deterministic. Hence, a node monitor that computes and disseminates a wrong forwarding or publication view risks eviction by other node monitors of the same node.

(7) Incentives for report checking and aswering to report checks. A rational node r periodically checks on its node monitors the existence of *reports* sent by nodes in r's forwarding view. If r does not check for the reception of those reports, it's evaluation may be biased (e.g., by Byzantine nodes) and it further risks blacklisting. Hence, report checking allows a node to insure that its monitors are aware if its forwarding activity and discourage receivers from not sending reports.

Furthermore, a rational monitor r, once queried by a monitored node p about reports sent by one of its neighbors, always reply and with the correct answer. In fact, if r does not reply, p will blacklist it. Then, the incentive to provide the correct answer is twofold. First, the response must be of a given fixed size, otherwise r will be blacklisted by p. Second, as the monitor (i.e., r) and the monitored node (i.e., p) can not be in each other's forwarding view, then r has no interest in not providing the correct answer to p.

5.5 Performance Evaluation

We have performed a simulation-based evaluation of *FireSpam*. To that purpose, we developed a C++ event-based simulator à la PeerSim [Jelasity et al.,]. We simulated a system composed of 1,000 nodes for three different filtering capability distributions : uniform, power-law, inverse of power-law. Moreover, we vary the ratio of Byzantine nodes that are in the system and that *FireSpam* is configured to tolerate : 1%, 3%, 5%. All other nodes are rational. In our simulations, Byzantine nodes do not forward good messages, do not filter spam messages, send randomly generated reports, and take random decisions when they act as monitors. In all experiments, the candidate sets used to select forwarding views have for size 5 times the number of Byzantine nodes in the system. Consequently, this guarantees that in a system with at most 5% Byzantine nodes, selecting 13 nodes in each forwarding view will ensure with probability 99.29% that a majority of them are non-Byzantine. Finally, the number of monitors per node is set to 5, which guarantees with probability at least 99.88% that each node has a majority of non-Byzantine nodes in its view when at most 5% of the nodes are Byzantine.

We compare *FireSpam* against lpbcast [Eugster et al., 2001], a popular gossip-based protocol that we extended to take into account spam filtering capabilities of nodes. Roughly speaking, in lpbcast, each node forwards the messages it receives to a set of nodes that continuously changes and that represents a random sample of the network. It has been proved that this set must have a size of $log(N) + c$ (where N is the size of the network and c is a constant) to ensure reliable delivery of messages [Kermarrec et al., 2003]. In our experiment, this size is set to 15. We slightly modified the behavior of nodes so that they do not forward the messages they detect as spam.

We first assess the correctness of the forwarding views that *FireSpam* assigns to nodes. Our experiment shows that each node has in its forwarding view a set of nodes among those having the closest filtering capabilities. We then show that both *FireSpam* and lpbcast ensure reliable dissemination of good messages. Follows an evaluation of the average ratio of spams received by each node as a function of the distribution of filtering capabilities and of the ratio of Byzantine nodes in the system. This evaluation shows that in all cases, *FireSpam* drastically reduces the ratio of spam messages received by the nodes that are in the upper part of the ladder. It also shows that the higher the filtering capability of a node, the lower the spam ratio it receives. We then assess the behavior of *FireSpam* under an eclipse attack, i.e. when a set of Byzantine nodes collude to break the ladder topology. This experiment shows that under an eclipse attack, *FireSpam* successfully maintains the ladder topology, ensures reliable dissemination of good messages, and keeps filtering spam messages in an efficient way. Finally, we assess the cost of *FireSpam* in terms of bandwidth consumption.

5.5.1 Correctness of forwarding views

Fig. 5.3 depicts the forwarding view assignment made by *FireSpam* in a system with 5% of Byzantine nodes. We observed very similar results for the three spam filtering distributions. We thus only report results for the uniform distribution. The X and Y axises both represent nodes, ordered by filtering capability. The forwarding view of a node X comprises the set of nodes for which a point is plotted. As explained in the previous section, each forwarding view comprises 15 nodes that are selected in a set that comprises 5 times the number of Byzantine nodes in the system, i.e. $5 \times 50 = 250$ nodes. We observe that *FireSpam* correctly assigns views : every node has a forwarding view that comprises 15 nodes that are chosen from the adequate candidate set. For instance, the node located at $x = 500$ has in its forwarding view a set of nodes located between $y_{min} = 375$ and $y_{max} = 625$.

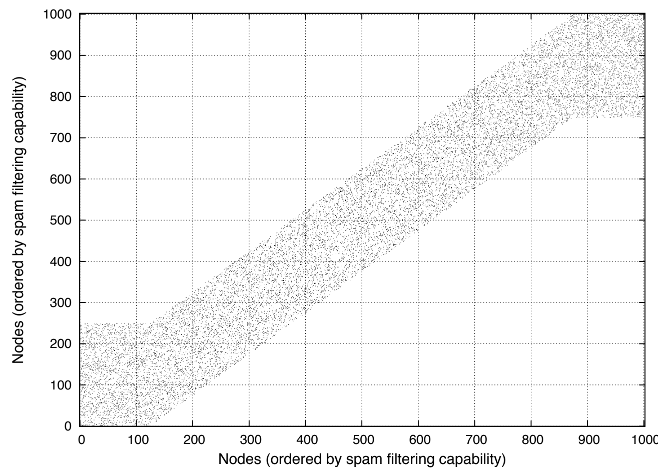


FIGURE 5.3 – Forwarding views assigned by *FireSpam* as a function of the node filtering capability (5% of Byzantine nodes setting).

5.5.2 Reliability of good messages delivery

Fig. 5.4 depicts the ratio of good messages delivery as a function of the number of hops for lpbcast and for *FireSpam* configured to tolerate 1%, 3%, and 5% of Byzantine nodes, respectively. The observed results were very similar for the three spam filtering distributions (as a consequence of the fact that they all lead to similar ladders, as noted in the previous section). We thus only plot results for the uniform distribution. We observe that both protocols achieve 100% reliability in delivery. Nevertheless, we observe that the average latency (expressed in the number of hops) to reach all nodes is higher for *FireSpam* than for lpbcast. This comes from the usage of the ladder, in which messages are progressively disseminated from the bottom to the top. Moreover, we observe that the latency decreases with the number of Byzantine nodes that are tolerated by *FireSpam*. This comes from the fact that this decreases the size of the candidate set from which forwarding views are chosen. As a consequence, the height of the ladder increases, and thus the time it takes to disseminate messages.

5.5.3 Ratio of spam messages received

Fig. 5.5 depicts the ratio of spam messages that are received by every node. The X axis represents nodes, ordered by filtering capability. Nodes on the left (resp. right) of the axis are thus located at the bottom (resp. top) of the ladder. We plot results for lpbcast and *FireSpam* for the three spam filtering capability distributions. Regarding *FireSpam*, we plot results obtained when the protocol is configured to tolerate 1%, 3% and 5% of Byzantine nodes, respectively.

We can first observe that, when using lpbcast, the ratio of spam messages that are received is the same for all nodes. This comes from the fact that each node has the same probability to communicate

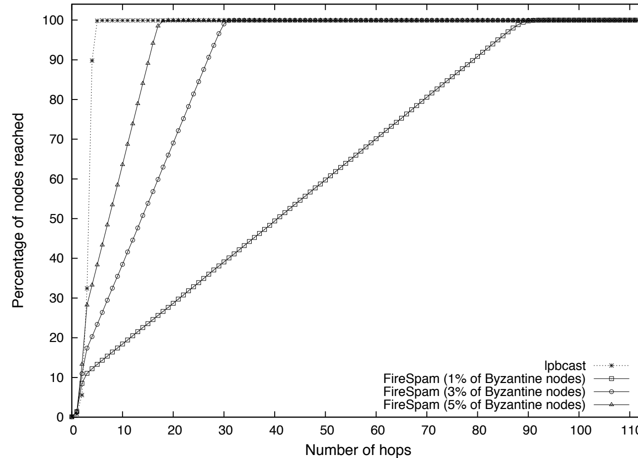


FIGURE 5.4 – Ratio of good messages delivery as a function of the number of hops.

with all nodes in the network. Thus, the benefit of spam filtering is uniformly spread among nodes. Not surprisingly, we also observe that the ratio of spam messages that are received is much lower when using the inverse power-law distribution (38%) than when using other distributions (around 75%). This comes from the fact that in the former case, many nodes have a very high filtering capability.

The second observation we can make is that using *FireSpam*, nodes receive a ratio of spam messages that is conversely proportional to their spam filtering capability. We do thus see that nodes have a clear incentive to filter spam messages, as this will consequently decrease the ratio of spam messages they receive. For instance, in all three distributions, the very best nodes receive at least 5 times less spam messages than the worst nodes.

We also observe that nodes that are at the bottom of the ladder receive more spam than using lpbcast, but also much more spam than nodes that follow them in the ladder. This is explained by the fact that these nodes receive all messages that are generated in the system because they are in the publication views of nodes. Once they have received messages, the filtering process starts and nodes located higher in the ladder receive fewer spam. Finally, we can also observe that in all three distributions, increasing the number of Byzantine nodes that can be tolerated decreases the overall spam filtering efficiency. This is explained by the fact that the candidate sets used to generate forwarding views become bigger, which results in a faster dissemination from the bottom to the top of the ladder topology, thus reducing the probability for spam messages to be filtered while being disseminated using the ladder.

5.5.4 Behavior under an eclipse attack

In this section, we study the behavior of *FireSpam* when 5% of the nodes are Byzantine and collude with the aim of harming the ladder. We consider a uniform distribution of the spam filtering capabilities. In order to harm the ladder, Byzantine nodes behave as follows : during a long enough period of time, they all behave similarly, filtrating and forwarding the same messages. Consequently, they end up at the same location in the ladder (in the middle in the presented experiment). Once they have all reached the middle of the ladder, they all start the attack simultaneously. During the attack, they do not forward good messages, and they do not filter spam messages. Fig. 5.6 depicts the ratio of spam messages that are actually received by every node in the system. Nodes are depicted in the X axis, ordered by spam filtering capability. We plot three different lines : before the attack (T0), during the attack (T1), and after the attack (T2).

Before the attack (line T0), the ratio of spam messages that are received is the same as the one depicted in Fig. 5.5 for the uniform distribution and 5% of Byzantine nodes. During the attack (line T1), we observe that the ratio of spam messages that are received by nodes in the upper part of the ladder (i.e. on the right of the X axis) slightly increases. This is due to the fact that the 50 colluding nodes no longer filter spam messages. These colluding nodes are located between $x = 475$ and $x = 525$, hence the

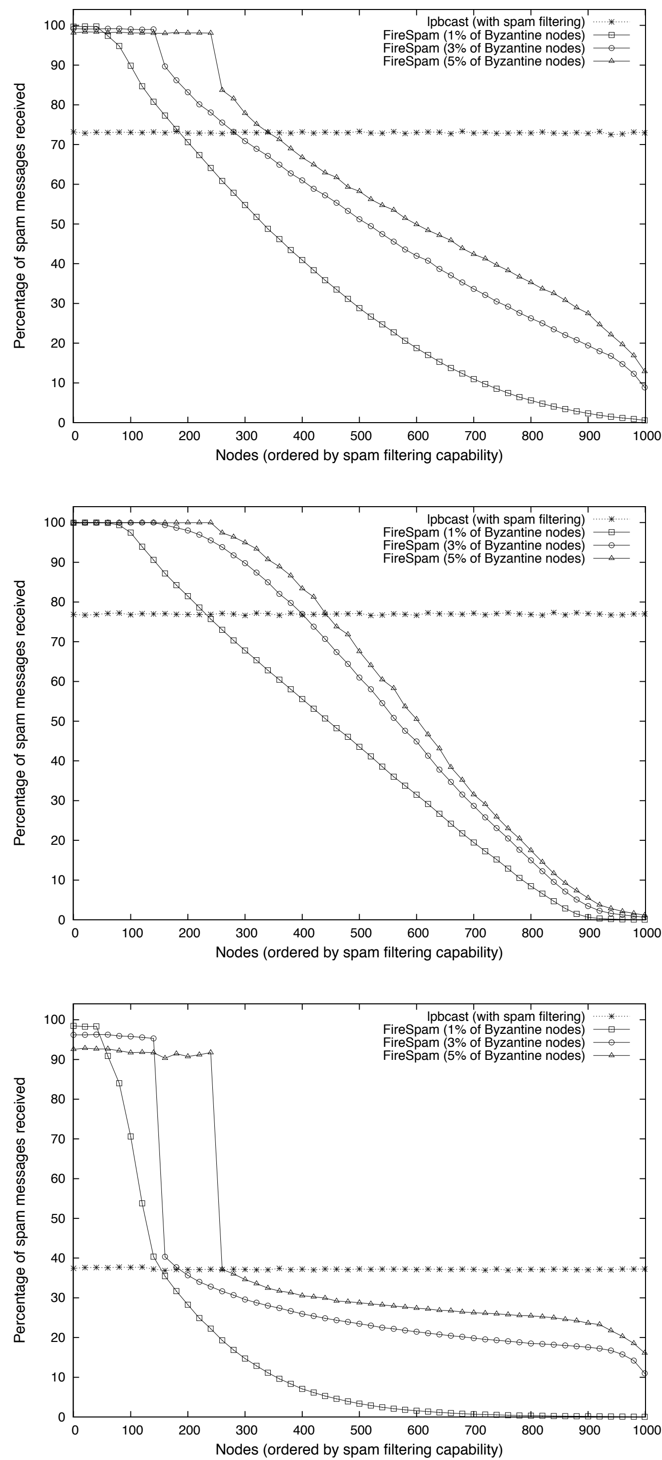


FIGURE 5.5 – Ratio of spam messages that are received by nodes for a uniform (1st row), a power-law (2nd row), and an inverse power-law (3rd row) distribution of the spam filtering capabilities.

increase of the ratio of spam received by nodes located above the colluding nodes in the ladder (i.e. nodes located on the right of x=525). Finally, we observe that some time after the attack (line T2), colluding nodes have been evicted from the network. The ratio of spam messages received by nodes in the upper part of the ladder is thus very close to the one that was observed before the attack (line T0). The slight difference comes from the fact that before the attack, colluding nodes were participating in the filtering of spam messages.

Moreover, we assessed the reliability of good messages delivery before, during and after the attack. For lack of space reasons, we do not include plots. Our results shown that *FireSpam* was able to reliably deliver good messages to all nodes during all the experiment.

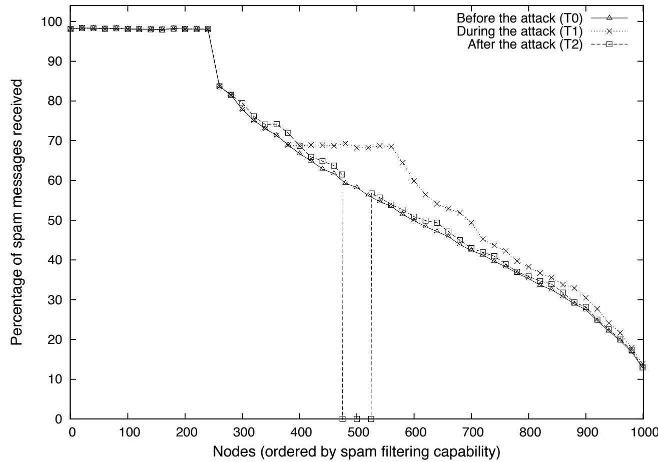


FIGURE 5.6 – Ratio of spam messages that are received by nodes before the attack, during the attack, and after the attack (uniform distribution of the spam filtering capabilities).

5.5.5 Bandwidth consumption

We have evaluated the bandwidth consumption of *FireSpam*. More precisely, we assessed the extra bandwidth consumption it incurs with respect to lpbcast for a system with 1000 nodes, when it is configured to tolerate 5% of Byzantine nodes. In that experiment, nodes in the system generate messages every 1 minute on average. We observed that *FireSpam* incurs a cost of $1,5kB/s$. In the incurred overhead, the ratio of control messages with respect to useful messages it is in the order of 6.885. In other words about 15% of the generated traffic in *FireSpam* is due to message dissemination while 85% of the traffic is dedicated to fault tolerance. Note that our design choices in *FireSpam* favour robustness against rational and byzantine behaviours, while performance has been a secondary concern. A number of optimisations can be investigated to reduce the amount of control messages generated by *FireSpam*.

5.6 Related Work

The spam filtering capability upon which *FireSpam* is based can be related to the "pollution awareness" of users as found in [Lee et al.,]. Albeit in a slightly different context (P2P file sharing instead of information dissemination), the authors study how P2P users most of the time help unintentionally spreading a polluted file [10] in the network. They observe in fact that a large fraction of users are insensitive to pollution (they do not check a downloaded file) and that even when they are, a signification portion of them fail in detecting that a file is polluted. They propose then a P2P user model where the pollution awareness of users expresses a user's probability to detect a polluted file.

Our work also relates to existing works in the following areas : gossip-based dissemination protocols,

10. A file which does not correspond to its description.

protocols for the BAR model, spam filtering mechanisms, and reputation systems.

Gossip-based dissemination protocols. Several gossip-based dissemination protocols have been designed. These protocols either use flat topologies, e.g. [Eugster et al., 2001], or hierarchical ones [Ganesh et al., 2002, Kermarrec and Ganesh, 2006, Eugster and Guerraoui, 2001, Lin and Marzullo, 1999, van Renesse and Birman, 2002]. Nevertheless, their design protocol does not address the possible presence of spam. In [Gavidia et al., 2007], the authors propose to limit spam in gossip protocols by adding security measures (integrity checking). Hovever, this simple approach does not reward in the mid-long term nodes who have behaved better than others by filtering more spam messages. Also, it does not provide incentives to prevent rational behavior.

Note also that our work has some similarities with slicing protocols, e.g. [Gramoli et al., 2009], which aims at dynamically organizing nodes into slices. It also share some similarities with clustering protocols, e.g. [Jelasity and Babaoglu, 2005, Voulgaris and van Steen, 2005] that cluster nodes according to self-announced attribute values. Nevertheless, existing slicing and clustering protocols do not consider Byzantine or rational behaviors.

Protocols for the BAR model. Several protocols have been designed for the BAR model. The closest protocols are BAR Gossip [Li *et al.*,] and FlightPath [Li et al., 2008] that are both gossip-based dissemination protocols customized for video streaming. Similarly to *FireSpam*, BAR Gossip is proved to be a strict Nash-equilibria. Our proof of the robustness of *FireSpam* is actually inspired by the proof of the robustness of BAR Gossip. Note also that *FireSpam* uses some mechanisms and credible threats that were used in BAR gossip, e.g. deterministic choice of dissemination targets, balanced cost in queries, unilateral denial of service mechanisms (e.g. blacklisting). The main distinction contribution with respect to these works is our ladder topology, and that the technique used to enforce this topology leverages on the necessity of having monitors which are themselves nodes partecipating in the system.

Spam filtering mechanisms. Email systems usually implement some black listing mechanisms [Golbeck and Hendler, 2004], which have a similar goal than the eviction mechanism defined in *FireSpam*. Also, while *FireSpam* currently relies on users for deciding which messages should be filtered, it could be extended to rely on machine-based learning techniques [Meyer and Whateley, 2004] in order to automate the detection of spam messages at each node.

Reputation systems. Several reputation systems have been designed that, as it has been observed in [Jusang et al., 2007], differ in the following ways : i) assumptions and usage of social links between nodes, ii) reputation model used to rank nodes, and iii) data management scheme to store reputations. To the best of our knowledge, no reputation system has been designed taking into account rational and Byzantine behaviors.

Concerning social links, *FireSpam* does not make any usage of social links, unlike protocols Ostra [Mislove et al., 2008] and KarmaNET [Spear et al., 2009]. Concerning reputation models, it is important to note that many different reputation models have been proposed. For example, there are "transitivity flow" models [Kamvar et al., 2003, Member-Zhou and Fellow-Hwang, 2007, Xiong and Liu, 2004], fuzzy logic models [Song et al., 2005, Aringhieri et al., 2006], statistical bayesian models [Wang and Vassileva, 2003, Buchegger and Le Boudec, 2004], or more generic data aggregation models [Grolimund et al., 2006]. The ranking model in *FireSpam* falls in this latter category : nodes are ordered according to the number of messages they forwarded. Finally, concerning data management schemes to store reputations, *FireSpam* share similarities with existing protocols that use decentralized data management schemes, e.g. [Gerard et al., , Kamvar et al., 2003]. The main contribution of *FireSpam* is that it defines incentives to actually guarantee that relevant information will be stored and that adequate usage will be made of the stored information (e.g. correct assignment of forwarding views).

5.7 Conclusion

Gossiping protocols are known to allow reliable and fast delivery of information in large-scale networks. Currently existing protocols cannot limit the amount of spam messages generated by nodes in the system. In this chapter, we present *FireSpam*, the first gossiping protocol able to limit spam dissemination.

FireSpam organizes nodes in a ladder topology : nodes with low spam filtering capability are at the bottom of the ladder, whereas nodes with a high filtering capability are at the top. *FireSpam* has been designed in the BAR model : it can thus tolerate an unbounded number of rational nodes, and a bounded number of Byzantine nodes. We have extensively evaluated it using simulations. Our results show that it allows to drastically limit the dissemination of spam, without hurting the reliability of good message delivery.

Chapitre 6

AcTing : Accurate Freerider Tracking in Gossip

Gossip-based content dissemination protocols are a scalable and cheap alternative to centralised content sharing systems. However, it is well known that these protocols suffer from rational nodes, i.e., nodes that aim at downloading the content without contributing their fair share to the system. While the problem of rational nodes that act individually has been well addressed in the literature, *colluding* rational nodes is still an open issue. Indeed, LiFTinG, the only existing gossip protocol addressing this issue, yields a high ratio of false positive accusations of correct nodes. In this chapter, we propose *AcTing*, a protocol that prevents rational collusions in gossip-based content dissemination protocols, while guaranteeing zero false positive accusations. We assess the performance of AcTinG on a testbed comprising 400 nodes running on 100 physical machines, and compare its behaviour in the presence of colluders against two state-of-the-art protocols : BAR Gossip that is the most robust protocol handling *non*-colluding rational nodes, and LiFTinG, the only existing gossip protocol that handles colluding nodes. The performance evaluation shows that *AcTing* is able to deliver all messages despite the presence of colluders, whereas both LiFTinG and BAR Gossip suffer heavy message loss. It also shows that *AcTing* is resilient to massive churn. Finally, using simulations involving up to a million nodes, we show that *AcTing* exhibits similar scalability properties as standard gossip-based dissemination protocols. This work has been carried out in the context of the PhD thesis of Jérémie Decouchant and has been published in the proceedings of the 33rd IEEE Symposium on Reliable Distributed Systems (IEEE SRDS'14).

6.1 Introduction

It is well known that content sharing applications account for a large proportion of traffic over the Internet. The most popular of these applications include collaborative downloading (e.g., BitTorrent) and peer-to-peer live streaming (e.g., P2PLive). Relying on the P2P paradigm offers robustness to failures, scalability up to hundreds of thousands of nodes, and adaptability. Indeed, P2P systems can handle massive node arrival/departure and are highly resilient to churn. From the point of view of content providers, relying on a P2P system allows shifting cost (e.g., bandwidth) to clients, and avoids the need for maintaining dedicated servers.

A major problem that face large scale P2P systems deployed on the public domain is the existence of rational nodes, i.e., nodes that aim at receiving content without contributing their fair share, by forwarding it to others. Existing studies have shown that the presence of even a small portion of rational nodes significantly degrades the system performance [et al., 2000, Krishnan et al., 2004, Feldman *et al.*, 2006, Cunha *et al.*, 2013]. This is why a number of protocols have been devised in the last decade to deal with the problem of rational nodes in collaborative systems, (e.g., rational resilient live streaming [Mol et al., 2008, Li *et al.*, , Guerraoui et al., 2010a], spam filtering content dissemination [Mokhtar et al., 2010] and N-party transfer [Vilaça et al., 2011]). All these protocols provide incentives that encourage/force rational nodes to participate in the system. However, apart from the protocol presented in [Guerraoui

et al., 2010a], all the existing solutions work under the assumption that rational nodes do not collude. This problem though has been demonstrated to be a reality in existing file sharing applications [Lian et al., 2007]. Handling colluding nodes is a difficult problem provided that colluders generally perform unobservable actions from the point of view of the collaborative protocol [Eidenbenz*et al.*, 2011], which makes their deviations difficult to deter. For example, a group of colluders could be a group of nodes that collaborate to exchange content between each other "off the record" (e.g., using the silent broadcast protocol described in [Eidenbenz*et al.*, 2011]). Such colluders do not share with nodes not belonging to the group the content they receive off the record, thus harming the protocol.

To the best of our knowledge, the only gossip-based content dissemination protocol trying to prevent collusions is the LiFTinG protocol [Guerraoui et al., 2010a]. In this protocol, nodes log their interactions with other nodes and perform distributed audits of each others logs. In order to be cost effective, this protocol relies on cryptography-free procedures and statistical analysis of these logs. For instance, a node is suspected of colluding with another node if the frequency of its interactions with the latter is greater than an expected average. Unfortunately, as analysed by the authors themselves, due to their statistical nature and to message losses, the mechanisms implemented in LiFTinG do not allow to catch all rational collusions (false negatives), and may even lead to wrong exclusions of correct nodes (false positives). Experiments that we performed, and that are described in Section 6.6.2, confirm this result and further show that, for instance, when 30% of nodes collude (either in a large group or in small collusion groups), correct nodes observe 25% of message losses.

The challenge we embrace in this chapter is the design of a rational-resilient content dissemination protocol that prevents collusions to occur and that does not wrongfully exclude correct nodes. An observation one can start with is : a colluding behaviour can be considered as a Byzantine behaviour [Lamport et al., 1982]. A legitimate question is thus to know whether it is possible to rely on existing techniques for Byzantine fault tolerance and Byzantine fault detection, such as Nysiad [Ho et al., 2008], PeerReview [Haeberlen et al., 2007a], Accountable Virtual Machines [Andreas*et al.*,], Trinc [Levin et al., 2009], or A2M [Chun et al., 2007] ? The answer is No. The reason is that these generic solutions for Byzantine fault tolerance and detection either assume a limited proportion of faulty nodes, or the existence of trusted nodes or hardware. Instead, we assume in this chapter that all nodes can be rational, and we do not rely on any trusted entity, whether software or hardware.

In this chapter, we present *AcTing* a content dissemination protocol that tolerates an unlimited number of (possibly colluding) rational nodes, while guaranteeing that no correct node is ever expelled, and that all rational deviations are eventually detected. To reach this objective, we adopt a different approach than the one used in the LiFTinG protocol : rather than trying to detect collusions a-posteriori, we built *AcTing* in such a way that it is *not* in the interest of nodes to collude. We analyse each step of the protocol and describe the incentives that force rational (possibly colluding) nodes to stick to the protocol. We perform a performance evaluation of *AcTing*. Its performance is compared with two protocols : BAR Gossip, the state-of-the-art gossip protocol that is able to handle *non*-colluding rational nodes and LiFTinG, the state-of-the-art gossip protocol that is able to handle colluding nodes. We implement a streaming application that we deploy on top of the three protocols. We deploy 400 nodes on one hundred physical machines and show that *AcTing* is able to deliver the entire stream despite the presence of colluders, whereas LiFTinG and BAR Gossip, both suffer heavy message losses. We also show that *AcTing* is resilient to churn, and using complementary simulations involving up to a million nodes, that it is scalable : it yields a logarithmic growth of memory and bandwidth consumption, comparable to standard gossip based protocols [Patrick *et al.*, 2004].

The rest of the paper is structured as follows. Section 6.2 describes our system model. Section 9.3 introduces the core ideas of *AcTing*. Section 6.4 provides a detailed presentation of *AcTing*. Section 6.5 discusses its resilience to (colluding) rational nodes. Section 9.6 presents a detailed performance evaluation. Section 6.7 reviews the related works. Section 9.8 concludes the paper.

6.2 System Model

We consider a system with N nodes, which are uniquely identified, e.g., using a hash value of their IP address. We assume that nodes can join and leave the system (gently or by crashing) at any time.

We consider two classes of nodes : *correct* nodes and *rational* nodes. Correct nodes follow the protocol. Rational nodes are defined as in [Li *et al.*,] extended with the notion of collusion : they aim at getting the content (i.e., missing the lowest possible number of updates) at the lowest possible overhead in terms of bandwidth consumption. This means that rational nodes would deviate in any sort from the protocol, possibly by *colluding* with each other, as long as the deviation saves their resources while not impacting the quality of the content they are getting.

Specifically, the benefit of colluding rational nodes can be represented along the following axes :

1. (*Stream Quality*) Receiving as much as possible (possibly, all) stream updates,
2. (*Communication*) Sending as little as possible (possibly, none) stream updates or protocol messages to nodes not belonging to their coalition,
3. (*Computation*) Performing as little as possible computations for other nodes.

Colluding rational nodes would typically exchange updates off the record, and, in order to save bandwidth, would not share the updates they obtained secretly with nodes outside their group. It is important to note that rational nodes are *risk averse*, i.e., they never deviate from the protocol if there is any risk of being evicted from the system. This assumption is commonly used in BAR systems [Aiyer et al., 2005]. Furthermore, this assumption is particularly relevant in our context as we use accountability techniques to deter faults and accuse nodes (as described in the following section). In this context, when a fault is detected, a proof of misbehaviour is produced, which can convince any correct node in the system of the necessity of evicting the misbehaving node. As eviction corresponds to an infinite penalty, no benefit is worth taking such risk. We also suppose that rational nodes join and remain in the system for a long time and seek a long-term benefit.

We refer to the source as the node that is disseminating a given content. We assume that each content is disseminated from a single source at a time but our principles can be easily applied to systems where the content is disseminated from multiple sources at the same time. We assume that all nodes but the source may be rational, or experience failures, and may organise themselves in colluding groups of arbitrary sizes. Classical fault-tolerance techniques (e.g., [Bressoud *et al.*, 1996]) can relax the assumption that the source does not fail.

We assume that the network allows every pair of nodes to exchange messages, and that they are eventually received if retransmitted sufficiently often. We also assume that hash functions are collision resistant and that cryptographic primitives cannot be forged. We assume that nodes are provided a pair of asymmetric keys, and denote a message m signed by a node i using its private key as $(m)_{\sigma(i)}$.

As in [Li *et al.*,] and [Li et al., 2008], we assume that nodes maintain clocks synchronised within δ seconds, and we structure time as a sequence of rounds in which nodes exchange updates. We assume that nodes have a secure log that is used to check their correctness through its analysis. A secure log is a log that is tamper evident and append only. Many systems recently defined variants of secure logs among which [Haeberlen et al., 2007a, Andreas*et al.*, , Levin et al., 2009, Chun et al., 2007]. We build on the secure log presented in [Haeberlen et al., 2007a].

6.3 Protocol Overview

We present *AcTing*, a gossip-based dissemination protocol that guarantees the following two properties : (i) a correct node is never expelled, and (ii) a rational node that deviates from the protocol in a way that impacts the performance of correct nodes is eventually suspected by all correct nodes. In the remainder of this section, we describe the principles of *AcTing* that allow us to guarantee the above two properties. Protocol details are then presented in Section 6.4.

Figure 6.1 shows an overview of our protocol. In this figure, the source node s, which is the node from which the dissemination originates, cuts the content into chunks that we call *updates*. It then periodically disseminates these updates to a set of nodes (arrows 1 in the figure). To join this content dissemination session, a new node (p_n in the figure) needs to know a node that is already part of it, as described in Section 6.4.1 (arrow 2 in the figure). In the middle of the figure, a node p_x, which characterises any node in the system except the source, has a set of nodes that it has selected as *partners* (depicted on its right side in the figure). Further, p_x has a set of nodes that selected it as a partner (depicted on its left side), to

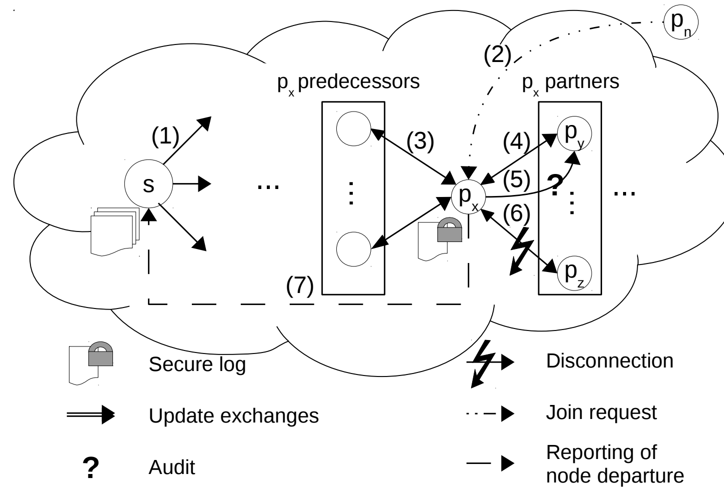


FIGURE 6.1 – Overview of *AcTing*.

which we refer as p_x's *predecessors*. Periodically, p_x has to share with its partners (arrow 4 in the figure) and with its predecessors (arrow 3 in the figure) the updates it received. In order to maximise the quality of the content it receives, p_x may be tempted to (1) act rationally by receiving updates and not sharing them with its partners, or predecessors, and (2) collude with other nodes in the system (not necessarily its partners or predecessors) to get updates off the record without sharing them with anyone else. To avoid these temptations, the core idea underlying *AcTing* is to make nodes accountable for their actions. Specifically, each node in *AcTing* logs in a *secure log* its interactions with other nodes in the system, including the identifiers of the updates it received. Because any node can verify the information in the log of a node it is interacting with, the latter will be obliged to send to its partners the updates it has, and to receive the updates it is missing. Consequently, no node will have an interest in behaving rationally or forming collusions. Indeed, assume that node p_x colludes with another node to receive an update u off the record. Node p_x will not be able to record update u in its log (because the exchange was unofficial; we explain later how it is done). The good news for node p_x is that it does not have to forward u to other nodes because u does not appear in its log. The problem is that the next time a correct node having u in its log will interact with node p_x, it will send update u to p_x. Consequently, p_x will eventually have to forward u, and thus will have wasted its bandwidth, because it will have received u twice (off the record and from a correct node).

This core idea raises several questions and challenges that we answer in the remainder of this section. **"What if p_x chooses only colluders as partners with which it will interact with in the near future ?"**. This way, p_x could accept updates and arrange with its future partners so as they do not audit its log, or so they do not send it updates it already received unofficially. Our protocol deals with this issue by forcing nodes to (periodically) establish random, yet *deterministically verifiable* partnerships as presented in Section 6.4.2. Specifically, each time a node p_x has to change its partners, it computes their identifier using a pseudo random generation function seeded with a deterministically computed seed. As such, nodes that will audit its log will be able to verify the legitimacy of the partners that it has selected.
"What if a node, p_x, maintains many (correct) logs ?". For instance, p_x could have a log in which an update u appears, which it will show to nodes who already have u (to avoid sending it to them), and another log in which the same update does not appear, which will be presented to nodes that do not have u (to avoid having to send it to them). This problem is known as *equivocation*, i.e., the ability to make conflicting statements to different participants [Levin et al., 2009]. We deal with this issue by forcing nodes to audit their partners' logs at the beginning of each new partnership (arrow 5 in the figure). This audit verifies the consistency of the log of a node as a whole as presented in Section 6.4.3.
"Isn't this periodic exchange of logs a performance overkill ?". It is not necessary to audit the logs of nodes each time two nodes exchange updates. Indeed, we build on the assumption that colluders, and rational nodes in general, are risk averse. Hence, it is enough to ensure that for each step of the protocol, a deviation has a high probability to be detected in the near future, in order to make sure that

rational nodes will not deviate. Consequently, instead of performing audits each time nodes communicate, audits are triggered in a *random yet verifiable* manner. Indeed, audits (from the point of view of audited nodes) must not be predictable, because rational nodes would seize an opportunity to deviate undetected if they could predict them. Yet they must be verifiable (from the point of view of nodes performing them), because rational nodes have to be forced to trigger this procedure. To reach this objective, a node that starts a new partnership with a node performs a deterministic computation that results in a boolean telling it whether it should audit its partner or not.
"What if rational nodes decide not to answer to correct nodes to avoid trading updates, or being audited ?". There are many reasons why a rational node may be tempted not to answer to a request from a correct node. This could, for instance, preserve it from sending its log and being audited as a result (arrow 6 in the figure). This type of misbehaviour is known as *omission failures*. We deal with this problem using a mechanism where unresponsive nodes are eventually suspected by all correct nodes, which stop interacting with them (as described in Section 6.4.1). As it is not in the interest of rational nodes to be isolated in the system, a rational node in *AcTing* will answer all correct node requests. To avoid correct nodes to be expelled from the system because one of their message has been lost or delayed, we allow suspicions to be released, e.g., if the missing message eventually arrives. Similarly, rational nodes may be tempted to wrongly suspect correct nodes of omission failure, by claiming that they did not send a given message to them, as it is the only reason why a node can skip mandatory interactions. We avoid this deviation by overcharging the sending of suspicion messages in such a way that it is more costly to suspect a node of omission failure than to effectively interact with it. As such, nodes would suspect other nodes of omission failures only if they are really missing a given message. Instead, if a node effectively left the system (assume node p_z in the figure), its predecessors (among which, node p_x in the figure) contact p_z's partners to collect evidence about the effective unresponsiveness of p_z (as described in Section 6.4.1). Then, p_x sends this evidence to the source node (arrow 7 in the figure), which eventually updates the membership list, and will also inform its partners during future exchanges.
Summarising, our protocol builds on accountability techniques, and on a set of mechanisms to provide incentives to rational, possibly colluding, nodes to stick to the protocol. Specifically, to avoid nodes from selecting their partners, our protocol relies on *random yet verifiable partnerships*. To be efficient it relies on *random yet verifiable audits*. To discourage rational nodes from being falsely unresponsive, our protocol handles *omission failures*. Finally, to discourage nodes from wrongly suspecting their partners our protocol *associates an extra cost with suspicion messages*.

6.4 Protocol Details

We have presented the principles of *AcTing* in the previous section. In this section, we detail the steps of the protocol.
In a nutshell, *AcTing* divides time in rounds. At each round the source disseminates new updates, which come to expiration after *RTE* rounds, to a small set of randomly chosen nodes. To get updates, each node initiates and maintains partnerships with other nodes with whom it exchanges updates at each round. The partners are selected using a pseudo-random number generator function, i.e., PRNG, seeded deterministically (e.g., with the node's public key concatenated with the round number). At the beginning of a round, each node contacts all of its partners in order to propose updates to them and to request updates from them. Every *Period* rounds, each node updates its set of partners. Each time a node starts a new partnership with a node, the two nodes audit each others' log with a given probability. The membership is managed in a distributed manner by nodes who periodically inform the source of the arrival and the departure of nodes. Yet, it is the responsibility of the source to disseminate an updated list of alive nodes every *Epoch* rounds.
The remainder of this section describes the sub protocols constituting *AcTing* in detail, as follows. First, we present the membership protocol (Section 6.4.1), which allows dealing with new nodes joining the system, nodes leaving it and unresponsive nodes. Then, we present the partnership management (Section 6.4.2), the audit (Section 6.4.3) and the update exchange protocols (Section 6.4.4), which allow handling the partnerships between nodes, auditing their logs and exchanging updates between partners, respectively.

6.4.1 Membership protocol

The membership protocol handles the arrival and the departure of nodes as well as the management of the membership list. Our membership protocol is fully distributed, rational resilient, and handles massive nodes arrival and departure.

Node arrival

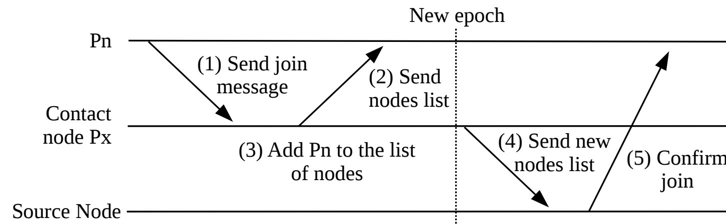


Figure 6.2 – Arrival of a new node.

The arrival of a new node follows the sequence of messages depicted in Figure 6.2. In this figure, we assume that node p_n, which would like to join a given content dissemination session, has installed the *AcTing* software. This means that p_n has an empty secure log with the related security primitives. We also assume that p_n knows an entry point in the system, say p_x, which we call the *contact node* of p_n. To join a content dissemination session, p_n sends a join request to p_x (step (1) in the diagram). The latter replies with the list of active nodes of the current epoch (step (2) in the diagram). Using this list, p_n computes its list of new partners using the PRNG function as described in Section 6.4.2 and contacts them to start receiving the content. At the beginning of a new round, each node, including node p_x informs the source of the arrival of new members that have contacted it (step (4)). Using theses messages, the source confirms to the new members their integration in the system and updates the membership list (step (5)).

Node departure and omission failures

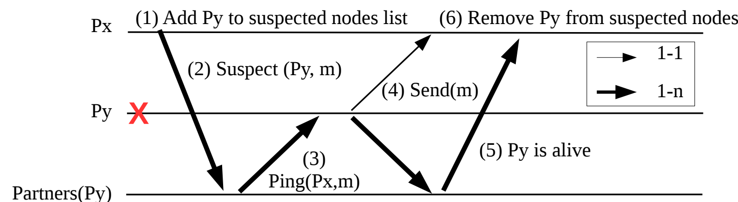


Figure 6.3 – Handling of an omission failure.

If a node p_x is expecting a message from one of its partners p_y for too long [11], it suspects p_y of omission failure as depicted in the diagram of Figure 6.3. Specifically, p_x adds p_y in its local list of suspected nodes (step (1) in the figure) and sends a suspicion message to the other partners of p_y (step (2)). This message includes the type of message that p_x is expecting from p_y. Then, each of p_y's partners pings p_y (step (3)). If p_y is alive, it replies to both its partners and p_x with the missing message (step (4)). After a given time slot, each of p_y's partners replies to p_x with a signed message certifying whether p_y responded to the ping message or not (step (5)). Using this message, p_x either removes p_y from its list of suspected nodes if p_y replied (step (6)) or sends an eviction message to the source including the messages received from p_y's partners.
To be sure that a rational node will never suspect a correct node, in order to avoid initiating or accepting an interaction, we make the cost of sending a suspicion message higher than the cost of a normal interaction. Hence, unless it is a real suspicion, a node will never suspect another node.

11. Delays for node suspicion are configured in an implementation dependent manner

Membership list update

Periodically, nodes that served as contact nodes for others send their list of new nodes to the source. Furthermore, nodes that hold an evidence of the departure of one partner send it to the source. The latter updates the membership list and sends it, at the beginning of each epoch, to the nodes along with the content. In order to fasten the removal of dead nodes from the membership list, an optimisation consists in letting the source disseminate the list of dead nodes at the beginning of each round along with the stream, instead of waiting the following epoch. As soon as a node receives these incremental updates from the source, it removes the corresponding nodes from its list of alive nodes, which avoids selecting them when new partnerships have to be established before the new epoch. In order to preserve nodes from the massive arrival of new nodes, which may consume their bandwidth, we adopt the optimisation defined in [Li et al., 2008], which allows splitting the load between the older nodes and the new ones. Specifically, this optimisation prevents new nodes from establishing too many partnerships with older nodes.

6.4.2 Partnership management

Each node p_x maintains partnerships with f other nodes, which are selected with the PRNG function seeded with a deterministically computed seed (e.g., p_x's public key concatenated with the round number) among the non-suspected nodes of the last membership list. This process is depicted in the diagram of Figure 6.4. If a selected node is not responding, node p_x has to propagate a suspicion, and once it is confirmed, p_x is allowed to find a new partner. Every *Period* rounds, a node p_x breaks the f partnerships it initiated, without informing its partners which know when the partnerships are supposed to end. A node having an identifier id will change its partnerships during round r if $(id + r) \mod Period = 0$. To initiate a new partnership with a node p_y, node p_x sends an association request to p_y (step (2) in the diagram).

At the beginning of a partnership, a node p_x may trigger an in-depth audit of its new partner p_y (step (4) in the diagram), by contacting the partners p_y had in the RTE previous rounds, and asking them to return their own log of the last RTE rounds including the current round (step (5) in the diagram). To reduce the cost of the protocol, nodes perform these audits in a random manner, i.e., each time they are in a position to perform an audit, they flip a coin and decide whether they should audit their partner or not. Nevertheless, to avoid that rational nodes hide behind this randomness to avoid auditing their partners, we make this randomness verifiable. Towards this purpose, we use the secure log *authenticators*, which are signed messages computed from the node's log as detailed in Section 6.4.3. These values are unpredictable as they depend on the current state of a node's log. Specifically, each time a node p_x is in a position to perform an audit of a new partner p_y, it computes the hash of its public key concatenated with the public key of p_y and the round number. The value of this hash modulo 100 gives a number that p_x uses to decide whether it should audit its new partner. For instance, if the probability of auditing a node fixed by the protocol is 30%, p_x audits p_y if the result of the modulo function is between 0 and 29. Node p_x further logs the authenticators it used to compute the value of this boolean, in order to justify, in future audits, the reason why it performed or did not perform the audit of p_y. If the audit must take place, p_x contacts p_y's partners, and asks for their logs.

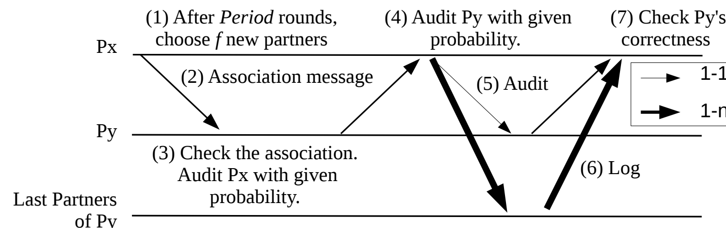


FIGURE 6.4 – Establishment of new associations between nodes, which may imply audits.

6.4.3 Audit protocol

In our protocol, the secure log is used to keep track of the communication a node had with other nodes in the system. Specifically, each entry in the log of a node A corresponds to a message sent (resp. received) by A to (resp. from) another node B. A log entry e_i is of the form $e_i = (seqno_i, h_i, c_i)$ where $seqno_i$ is a monotonically increasing sequence number, h_i is a hash value linked with the previous entries in the log and c_i is a type-specific content, which may include the message sent (resp. received) by A as well as other information such as authenticators (as defined below). The value of h_i is computed as follows : $h_i = H(h_{i-1}||seqno_i||H(c_i))$, where $h_0 = 0$, H is a hash function and $||$ stands for concatenation.

Each time a log entry e_i is added to the log of a node A, an authenticator α_i is generated. This authenticator, which is a signed message $\alpha_i = (seqno_i, h_i)_{\sigma(A)}$, states that A has a log entry e_i with a corresponding hash h_i. By sending the authenticator α_i to a node B, A commits to having logged the entry e_i and to the content of its log before e_i. Any node that receives α_i can use it to inspect e_i and all the entries preceding e_i in the log of A. Upon reception of a log, any node is able to recompute the hash values it contains, according to the log entries, and thus to check their validity. In addition, a log entry for a received message must include a matching authenticator, implying that a node cannot invent an entry for a message it did not receive. These two properties make the secure logs tamper-evident and append only.

As described in the partnership management protocol, when node p_x must audit node p_y, it asks p_y's partners to return their logs. Upon reception of these logs, node p_x verifies :
(i) the consistency of the logs, by recomputing the recursive hash values associated to log entries,
(ii) the presence of the exchanges p_y was supposed to initiate,
(iii) that p_y declared the updates it was supposed to receive from the source, if p_y was supposed to interact with the source,
(iv) that the exchanges correspond to a correct execution of the protocol, i.e., that p_y proposed to all its partners all the updates that appear in its log, that p_y requested from its partners all the updates it was missing, that p_y served to its partner all the updates they were requesting and that p_y logged all the identifiers of the updates it received,
(v) that p_y suspected all its partners that did not follow a given step of the protocol as prescribed by the omission failure protocol,
(vi) that p_y audited all the partners it was supposed to audit.
As any other node, the source also maintains partnerships and regularly changes its partners, i.e., the nodes it serves. The source follows the partnership management and the updates exchange protocols, except that it does not send any log and it is not audited by nodes [12]. This forces the nodes to log the identifiers of the updates they received from the source, as they are deterministically chosen among the epoch membership list, which is known by all nodes. Hence, any node can check that the received updates were correctly declared. As the serving rate of the source is constant, the identifier of the updates that are released at each round are also known.

6.4.4 Update exchanges

At the beginning of each round and for the duration of their partnership, two partners p_x and p_y exchange updates as depicted in Figure 6.5. Specifically, node p_x (resp. p_y) starts the exchange by generating a proposition message containing the identifiers of all the updates that appear in its log and that did not expire yet. Node p_x (resp. p_y) logs this proposition message in its log and generates the corresponding authenticator. Then, p_x (resp. p_y) sends the proposition message along with the corresponding authenticator to p_y. Upon reception of the proposition message, which it logs, node p_y (resp. p_x) selects the updates it is missing and replies to p_x (resp. p_y) with an update request. The update request is logged at the two parties. Finally, p_x (resp. p_y) serves the missing updates, and logs the serve message. Each partner then terminates the exchange by logging the identifiers of the updates it received, in its log. The nodes will then propagate the received updates during the following rounds, because we cannot ensure that nodes will immediately share them.

12. We recall that the source is assumed to be a correct node.

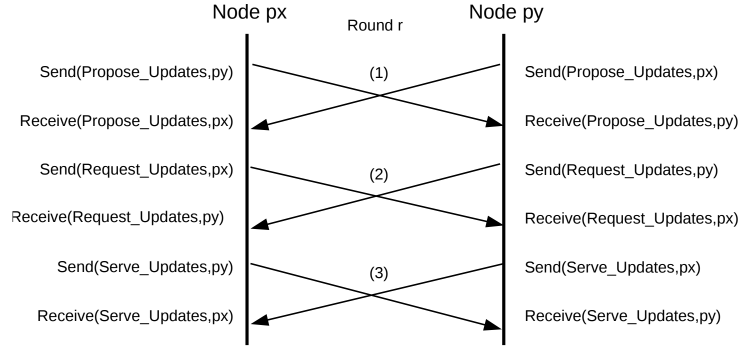


FIGURE 6.5 – Update exchanges between nodes.

6.5 Risk versus gain analysis

We first evaluate the risk that two colluding partners take by deviating, for example when interacting as prescribed by the protocol, but without logging the updates they exchange. Specifically, consider two partners p_x and p_y that decide to collude. Assume p_x holds update u. To help p_y saving bandwidth in future rounds, p_y sends a proposition message to p_x that does not contain u, but logs that it has proposed u. As such, the logs of p_x and p_y appear correct if audited separately as p_x can not be blamed of not requesting u (as the official proposition sent by p_y does not contain u) and p_y can not be blamed of not proposing u as it appears in his log that he has done so. We define the risk as the probability that such a deviation is deterred by an audit.

Let us compute this risk. If any of the two colluding nodes is audited during the time where the exchange is contained in their logs, they will be discovered. Let us consider a system of N nodes, where C nodes are part of a single colluding group. A node's log contains the entries of the last RTE rounds. A participating node initiates f partnerships with other nodes, which are changed after $Period$ rounds. Let P_{audit} the probability that a node audits each of its new partners. When establishing a new partnership, a rational node is not audited if its new partner is colluding with it (which happens with probability $\frac{C}{N}$), or if the new partner is correct but the protocol prescribes not to perform the audit. On average, each of the two nodes interacts with $\frac{2\times f\times RTE}{Period}$ partners during the time the deviation is visible. Finally, we obtain that the risk a deviation is detected is equal to : $\left(1-\left(\frac{C}{N}+\left(1-\frac{C}{N}\right)\times(1-P_{audit})\right)^{\frac{2.f.RTE}{Period}}\right)^2$

Let us now compute the gain of performing the above deviation. To do this we need to compute the number of interactions that a rational node may have with correct nodes that do not hold the update u after receiving it from its colluding partner, i.e., correct nodes to which the rational node would have had to send u if it has received it officially from its colluding partner. To do so, we use the algorithm of Figure 6.6. The principle of this algorithm is that during each of the RTE rounds that follow the round at which the deviation occurred, $2*f$ interactions happen. Each of these interactions, has a probability $\frac{C}{N}$ to involve another colluding node. When it is not the case, this other node owns the missing update with a probability that depends on the number of rounds elapsed since its release by the source.

In Figure 6.7, we present a part of the algorithm that we used to determine the probability that a node receives an update r rounds after it is released by the source. Using the values of the parameters we used in our evaluations, we obtained the probabilities that are contained in table 6.1.

Round	1	2	3	4	5
Probability (%)	0.12	0.48	1.9	7.3	25.6
Round	6	7	8	9	10
Probability (%)	65.5	87.4	95.4	98.3	99.4

TABLE 6.1 – Probability that a node receives an update in function of the number of rounds elapsed since its release by the source.

```
saved_sends_nb = 0;
for round_id in 1..RTE do
   for association_id in 1..2*f do
      if random() > C/N then
         if random() < probability[round_id] then
            return  return saved_sends_nb;
         else
            saved_sends_nb = saved_sends_nb + 1;
         end if
      end if
   end for
end for
return  saved_sends_nb;
```

FIGURE 6.6 – Pseudocode of the algorithm used to estimate the number of times a colluding node avoids to send an update.

```
infected_nb = f;
non_infected_nb = N;
for round in 1..RTE do
   new_nodes = ∅;
   for node in 1..infected_nb*f do
      node_id = random(1, N);
      if node_id > infected_nb and not new_nodes.contains(node_id) then
         new_nodes.add(node_id)
      end if
   end for
   Print(infected_nb)
end for
```

FIGURE 6.7 – Pseudocode of a part of the algorithm used to estimate the probability that a node received an update depending on the round number.

Using the average of the outputs of this algorithm, we can compute the proportion of interactions in which an update will not be sent by rational nodes. To obtain the long term gain, we multiply this proportion by the probability that a rational node has to meet an accomplice to be able to execute this deviation, which is $\frac{C}{N}$.

Computing the risk, and the gain, with the values of the parameters used in the protocol and further described in the following section, we obtain that the risk two colluding nodes take is equal to 60%, and the long term gain of the associated deviation is equal to 3%. Thus, rational nodes are exposed with a high risk each time they execute the deviation, and can only hope for a very small benefit. As a result, we conclude that rational nodes will not collude with their partners to exchange updates off the record. As said above, a complete analysis of the incentives provided by the protocol can be found in a technical report available online [13]. Note that nodes can still collude silently with nodes that are not their partners. Yet, it they do so, they are still obliged to execute the protocol correctly, i.e., request updates they do not officially hold and propose updates they officially hold to correct nodes. Hence, their collusion will not have any impact on the quality of the stream perceived by correct nodes.

6.6 Performance evaluation

In this section, we present the performance evaluation of the *AcTing* protocol. We start by introducing our methodology (Section 6.6.1). Then, we compare the impact of colluders on *AcTing*, BAR Gossip, and LiFTinG (Section 6.6.2). We choose BAR Gossip as it is the most robust rational resilient content dissemination protocol that has been proposed so far and LiFTinG as it is the only state-of-the-art content dissemination protocol that handles colluders. We then assess the bandwidth consumption of *AcTing* (Section 6.6.3), its performance in the case of massive node departure (Section 6.6.4) and its scalability in terms of memory and bandwidth consumption using simulations involving up to a million nodes (Section 6.6.5).

Overall, our evaluation draws the following conclusions : In a real deployment involving 400 nodes and in presence of colluders, correct nodes using *AcTing* do not experience any degradation in the quality of the content they receive while those using BAR Gossip and LiFTinG experience heavy message loss in presence of colluders independently from their organisation (whether in small or larger groups). On the other hand, we show that nodes that decide to collude in *AcTing* experience a heavy overhead, which discourages them from staying in the coalition. Moreover, we show that *AcTing* bandwidth consumption is reasonable and that *AcTing* is resilient to massive node departure. Finally, we show that *AcTing* is scalable as simulations involving up to a million nodes exhibit that both the bandwidth and memory consumptions of *AcTing* follow a logarithmic growth in the number of nodes. However, we acknowledge that the source may become a bottleneck as the number of nodes increase, as it periodically receive notifications. Solving this issue is classically done by using a tracker, i.e., a centralised server that handles membership, as in the FlightPath protocol [Li et al., 2008], which could easily be integrated in our system. The tracker could even be replicated using classical fault-tolerance techniques (e.g., [Bressoud *et al.*, 1996]).

6.6.1 Methodology and Parameter Setting

To assess the performance of *AcTing*, BAR Gossip and LiFTinG, we used them to implement three video live streaming applications. In these applications, a source node, selected randomly, diffuses a video stream at a rate of 300 kbps, during 5 minutes, and proposes each update to 5 random nodes. Updates are then disseminated using either *AcTing*, BAR Gossip or LiFTinG, respectively. In order to provide a fair comparison, we implemented the three streaming applications in Java using the same code base. We deployed the three applications in 400 nodes running in one hundred physical machines of the Grid5000 cluster, interconnected with a 1Gb/s network that we limited to 1Mb/s. Each machine is composed of an Intel Xeon L5420 processor clocked at 2.5GHz with 32GB of RAM. In the three applications, to provide

13. AcTing technical report : https ://sites.google.com/site/soniabm/

further tolerance to message loss (combined with retransmissions), the source groups packets in windows of 40 packets, including 4 FEC [14] coded packets.

The duration of one round is set to one second, and updates are released 10 seconds before being consumed by the nodes media player. Note that nodes dynamically adapt the number of their partners according to the size of the membership list : each node establishes $\left\lceil \frac{\ln(N)}{2} \right\rceil$ partnerships that it maintains for a duration of five rounds. For instance, in the fault free case, with $N = 400$, each node has 3 partners. At the beginning of each partnership, nodes performed audits with a probability of 5%, which, as we show in Section 6.5, allows the system to detect deviations with a probability of 60% when up to 10% of the audience colludes in a single group. The cryptographic primitives consisted in a 1024-bit RSA signature and a SHA-1 hash.

6.6.2 Impact of Colluders

In this section, we experimentally study the impact of colluders on the BAR Gossip, LiFTinG, and *AcTing* protocols. We implemented colluders from the code base of correct nodes in each protocol as follows. Colluders exchange unofficially among each other all the stream updates they received from correct nodes. Furthermore, colluders execute all the possible undetectable rational deviations that exist in the underlying protocol. For instance, in BAR Gossip, colluders never take part of the optimistic push protocol, which allows nodes to altruistically push updates to other nodes. Similarly, in LiFTinG, colluders do not audit the logs of other nodes and do not reply to messages sent by other nodes asking them to assess the behaviour of their previous partners unless the considered partner is among the group. As a result, correct nodes will be blamed by their correct auditors. In this situation the system administrator has two choices : (1) adjust the detection threshold to avoid false positives (by decreasing its value), which opens the doors to colluders for freeriding or (2) adjust the detection threshold to detect colluders (by increasing its value), which results in very high values of false positive accusations. In this experiment, we considered the first situation. A complementary experiment showed that in the second situation, adjusting the threshold to exclude 20% of colluders incurred the exclusion of 43% of correct nodes in the system. Finally, in AcTinG, colluders do not forward updates they received unofficially to their correct partners unless they also received them officially.

We varied the number of colluders, as well as the size of colluding groups. We measure the percentage of missed updates observed by correct nodes in presence of a proportion of colluders. We first studied the case in which all colluders belong to the same group. Results are depicted in Figure 6.8. The X axis presents the proportion of nodes that collude, while the Y axis presents the percentage of missed updates experienced by correct nodes in presence of colluders. We notice that correct nodes miss up to 98% of updates with BAR Gossip and 72% of updates with LiFTinG, whereas they do not miss any update with *AcTing*.

We then studied the impact of spreading colluders in multiple independent groups. More specifically, we made several experiments in which we distributed 30% of all the nodes in colluding groups of identical size. We depict the results in Figure 6.9. The X axis presents the size of colluding groups, while the Y axis presents the percentage of missed updates observed by correct nodes. We observe that spreading colluders in different groups has the same impact on the quality of the content downloaded by correct nodes.

Group size	2	4	8	10	50
Overhead (%)	34.35	51.53	60.12	61.84	67.33

TABLE 6.2 – Overhead of colluders in *AcTing*.

The reason why correct nodes do not observe missed updates when using *AcTing* is that we designed *AcTing* in such a way that colluders will eventually receive all the updates officially from their correct partners and will thus be obliged to forward them officially to their correct partners. Hence, engaging in a colluding group only yields an extra overhead due to the unofficial dissemination of updates among the

14. FEC stands for Forward Error Correction.

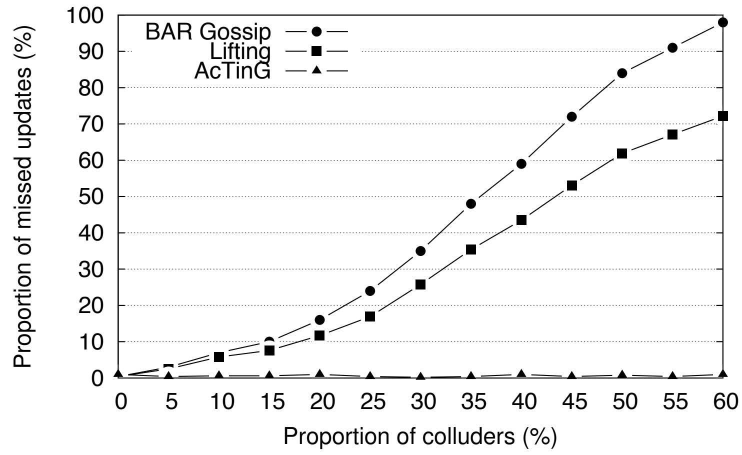


FIGURE 6.8 – Proportion of missed updates by correct nodes when a given proportion of the audience collude as a single group.

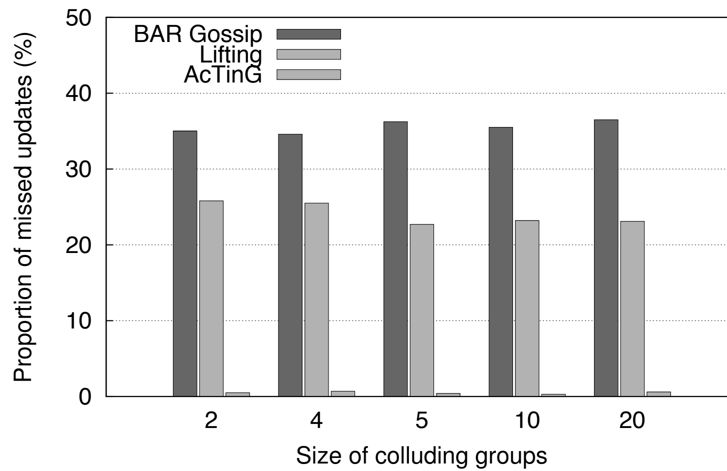


FIGURE 6.9 – Proportion of missed updates by correct nodes when 30% of the audience is rational, and collude in independent groups of equal sizes.

group. We have measured this overhead and results are depicted in Table 6.2. From this table we observe that the overhead due to collusion is at least of 34% of the size of the stream (case of a group containing only two colluders). In addition, as seen in section 6.5, in a scenario where 10% of nodes collude, and where audits are performed 5% of the time, each deviation will be detected with a probability of 60%. Moreover, exchanging updates without declaring them will provide at most a gain equal to 3%. Consequently, nodes in *AcTing* have no interest in colluding as they would not observe any increase in the quality of the stream they get, take a very high risk of being evicted, experience very low benefit, while suffering a useless waste of bandwidth.

6.6.3 Bandwidth consumption

To assess the overhead of *AcTing*, we plot in Figure 6.10 the cumulative distribution of the average bandwidth consumption of nodes. Recall that *AcTing* is used to broadcast a 300kbps stream. Figure 6.10 shows that *AcTing* induces a reasonable overhead (that is mostly due to the transmission of logs). We also measured the memory consumption of *AcTing*, which is due to the storage of secure logs and authenticators. Our measures have showed that a node consumes 3MB of memory for each partnership, in the worst case.

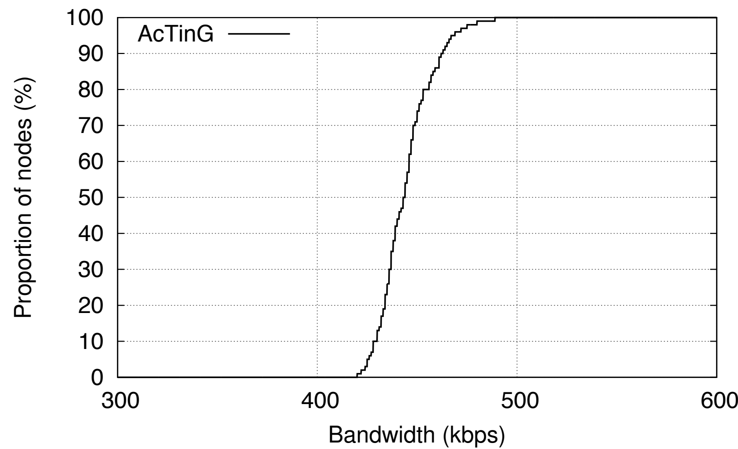


FIGURE 6.10 – Fault-free case : Cumulative distribution of average bandwidths.

6.6.4 Resilience to massive node departure

In the case of a massive node departure, the remaining nodes need to quickly replace their left partners with alive nodes in order not to miss updates. In this experiment, we measure the bandwidth consumption and the percentage of missed updates when 60% and 70% of nodes suddenly leave the streaming session. Results are depicted in Figures 6.11 and 6.12 respectively. Specifically, we observe in Figure 6.11 that the massive node departure, which happens 500 seconds after the beginning of the experiment, immediately causes a decrease in the average bandwidth consumed by the remaining nodes, as they stop exchanging messages with their left partners. This decrease (62% and 75% in the case of the departure of 60% and 70% of nodes, respectively) is followed by an increase (of up to 18% and 27% in the former two cases), which corresponds to the messages exchanged by nodes to establish new partnerships (including a given proportion of audits). Finally, we observe that 30 seconds later, the average bandwidth consumption stabilises around 430 kbs (13% less than the original value), which is due to the decrease of the necessary number of partners per node.

We also compute the percentage of nodes that do not receive a viewable stream [15]. We observe in Figure 6.12 that only 2,5% nodes do not receive a viewable stream during the first second when 60%

15. The stream is not viewable when more than 5% of the streaming windows cannot be displayed because of missed updates [Li *et al.*,]

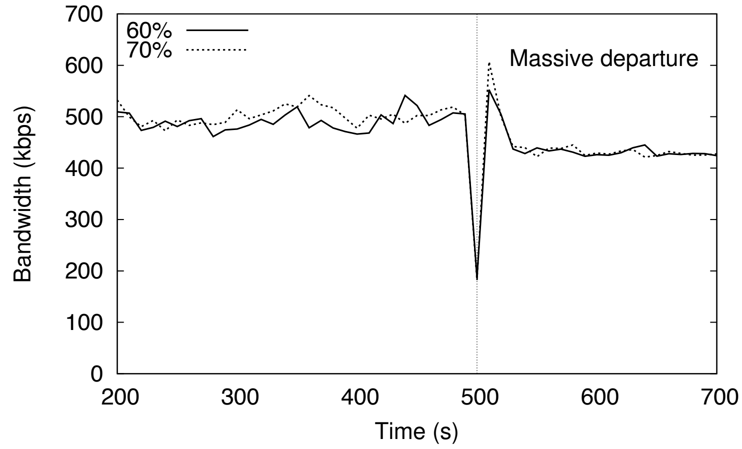


FIGURE 6.11 – Nodes average bandwidth after a massive departure.

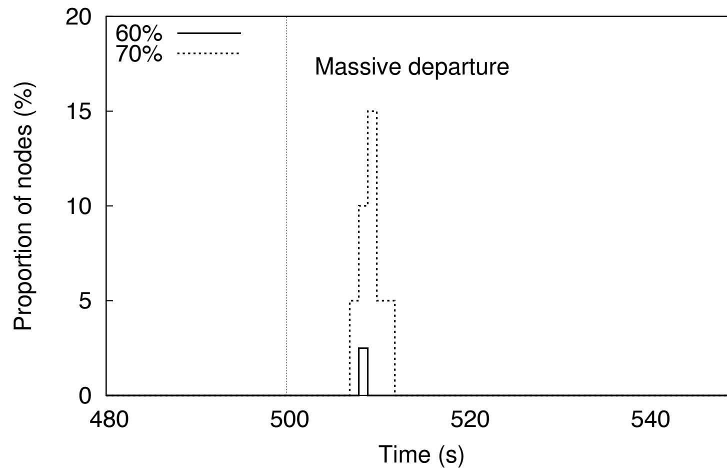


FIGURE 6.12 – Percentage of nodes that do not receive a viewable stream after a massive departure.

nodes leave the system, and between 5% and 15% nodes do not receive a viewable during at most five seconds when 70% nodes leave the system.

6.6.5 Scalability

We performed simulations to evaluate the bandwidth, and the memory consumption, of *AcTing* when the number of nodes increases in the system.

Results, depicted in Table 6.3, show that both the bandwidth consumption and the memory consumption of *AcTing* grow logarithmically with respect to the number of nodes in the system. Indeed, these values depend linearly on the number of partners a node has, which grows logarithmically with the system size.

System size	Bandwidth consumption (Kbps)	Memory usage (Mb)
100	380.0	6.4
500	436.6	9.5
3,000	511.1	12.7
22,000	603.4	15.9
160,000	713.5	19.1
1,200,000	841.4	22.3

TABLE 6.3 – Average bandwidth and memory usage of *AcTing* in function of the system size.

6.7 Related Works

In this section, we focus on peer-to-peer content dissemination protocols that handle rational nodes. These protocols can be classified into two categories, according to the way file chunks (called *updates* in the following) are exchanged between nodes. The first category of protocols is composed of *symmetric* protocols. These protocols force nodes to collaborate, as the number of updates they get from a node is proportional to the number of updates they have to offer (this principle is often referred to as tit-for-tat). BAR Gossip [Li *et al.*,] and FlightPath [Li et al., 2008] are symmetric protocols relying on game theory. Both provide incentives to ensure that rational nodes respectively have no, or a limited, interest in deviating from the protocol. In terms of robustness to rational nodes, the BAR Gossip protocol exhibits stronger properties than the FlightPath protocol. Indeed, nodes in FlightPath are assumed to deviate only if the benefit they get is higher than a threshold, which is not the case in BAR Gossip. While the authors of these two protocols point out the problem of colluding rational nodes in [Li *et al.*,], none of them address it.

The second category of protocols is composed of *asymmetric* protocols. These protocols require nodes to altruistically push update identifiers to other nodes, which subsequently pull updates of interest. A first protocol in that category is the one presented in [Haridasan et al., 2008]. This protocol aims at adapting the contribution of nodes to the systems, according to their available resources. This protocol assumes the existence of trusted auditors that run in dedicated external nodes and does not deal with colluders. A second protocol in that category is LiFTinG [Guerraoui et al., 2010a]. To the best of our knowledge, LiFTinG is the only existing peer-to-peer content dissemination protocol that tackles the problem of colluding rational nodes. Specifically, LiFTinG sporadically verifies the distribution of the interactions a given node performed with other nodes in the system. Nodes that collude with other nodes break the uniform distribution of partner selection, which may result in their detection. In order to be cost effective, LiFTinG only performs sporadic audits, and relies on non-secure logs that can contain wrong information, be incomplete, be tampered with and, as a consequence, be inconsistent the ones with respect to the others. As a result, LiFTinG suffers from two major limitations : correct nodes can be wrongly evicted from the system (false positives), and a proportion of colluding rational nodes can harm the system without being detected (false negatives).

6.8 Conclusion

A number of gossip-based content dissemination protocols tolerating rational behaviours have been proposed. A limitation of these protocols is that they do not handle rational nodes that collude, i.e. that act as a group in order to improve their benefit. The only exception is the LiFTinG protocol that performs sporadic checks on insecure logs to try to detect colluding nodes. We have shown in this chapter that neither LiFTinG nor BAR Gossip, the most robust rational resilient content dissemination protocol, are effectively resilient to colluders. In this chapter, we have presented *AcTing*, the first content dissemination protocol that tolerates rational nodes acting both individually and in collusions, and that guarantees zero false positive accusations. Performance evaluation combining both a real deployment and simulations has demonstrated that nodes running AcTinG are able to deliver the entire content despite the presence of colluders. We have also shown that AcTinG is resilient to churn, and exhibits very desirable scalability properties with a logarithmic growth of memory and bandwidth consumption, comparable to standard gossip based protocols. Our future work includes the study of the applicability of the AcTinG principles to other types of collaborative applications for the accurate detection of rational (possibly colluding) nodes.

Chapitre 7

RAC : a Freerider-resilient, Scalable, Anonymous Communication Protocol

Enabling anonymous communication over the Internet is crucial. The first protocols that have been devised for anonymous communication are subject to freeriding. Recent protocols have thus been proposed to deal with this issue. However, these protocols do not scale to large systems, and some of them further assume the existence of trusted servers. In this chapter, we present *RAC*, the first anonymous communication protocol that tolerates freeriders and that scales to large systems. Scalability comes from the fact that the complexity of *RAC* in terms of the number of message exchanges is independent from the number of nodes in the system. Another important aspect of *RAC* is that it does not rely on any trusted third party. We theoretically prove, using game theory, that our protocol is a Nash equilibrium, i.e, that freeriders have no interest in deviating from the protocol. Further, we experimentally evaluate *RAC* using simulations. Our evaluation shows that, whatever the size of the system (up to 100.000 nodes), the nodes participating in the system observe the same throughput. This work has been carried out in the context of the PhD thesis of Gauthier Berthou and has been published in the proceedings of the 33rd IEEE International Conference on Distributed Computing Systems (IEEE ICDCS'13).

7.1 Introduction

Anonymous communication protocols are important for they allow the dissemination of sensitive content over the Internet. The first protocols that have been proposed in the literature to enable anonymous communication are DC-Net [Chaum, 1988] and onion routing [Goldschlag et al., 1999]. These two protocols have focused on enabling the strongest possible anonymity for the former, and on providing practical performance for the latter.

An issue shared by the two above protocols, as well as by other protocols devised using the same principles (e.g., [Reiter and Rubin, 1998, Goel *et al.*, 2003]) is that they take the participation of nodes for granted. This assumption is unrealistic in collaborative systems, which are well-known to be perfect playgrounds for freeriders. A freerider is a node that benefits from the system, while trying to minimize its contribution to it, in order to save resources. Research has thus moved towards the design of freerider-resilient anonymous communication protocols.

The first protocol that has been devised to deal with freeriders is called Dissent v1 [Corrigan-Gibbs and Ford, 2010]. This protocol forces nodes to participate in the protocol. Specifically, for each message sent anonymously, Dissent v1 forces each node in the system to send messages to all the other nodes. As such, it is easy to detect whether a node contributed its fair share to the system. Unfortunately, this approach yields very poor performance. Indeed, this protocol becomes unpractical for systems involving as little as 40 or 50 nodes.

A second protocol, called Dissent v2 [Wolinsky et al., 2012], has been very recently proposed with the aim to improve the performance achieved by Dissent v1. The key idea of this protocol is to rely on a set of trusted nodes to avoid involving all the nodes in the system for every anonymous communication. In

this protocol, each trusted node receives anonymous communication requests from untrusted nodes and runs a protocol involving all-to-all communications between trusted nodes. This considerably reduces the number of messages exchanged for every anonymous communication, which allows Dissent v2 to exhibit better performance than Dissent v1. However, as we show in Section 7.3, this protocol is still not scalable : trusted nodes are involved in all communications and the throughput achieved by the protocol drastically decreases when the number of nodes in the system increases, until it reaches zero (for a system with 100.000 nodes). Besides, this protocol relies on trusted nodes, which is something that people sending anonymous data usually prefer to avoid.

Our contribution in this article is RAC, a freerider-resilient anonymous communication protocol that scales to large systems. Regarding the resilience to freeriders, we theoretically prove, using game theory, that our protocol is a Nash equilibrium, i.e, that freeriders have no interest in deviating from the protocol. Regarding scalability, both the number of broadcast messages and the size of the groups in which the broadcast messages need to be sent, are *independent* from the number of nodes in the system. Specifically, in our protocol, these parameters are exclusively dependent on constants that are associated with the degree of anonymity our system guarantees. In other words, RAC is scalable and it exhibits a clear tradeoff between anonymity and performance. Finally, RAC does not assume any trusted server.

We experimentally evaluate RAC using simulations. Our evaluation shows that RAC achieves a very good level of anonymity. Moreover it shows that, contrary to Dissent v1 and Dissent v2 that exhibit a drop of throughput when the number of nodes grows, the throughput of RAC remains constant when increasing the size of the system.

The remaining of the paper is structured as follows. We first introduce some definitions and analyze the related work in Section 8.2. We then discuss the need of a new protocol in Section 7.3, before presenting our protocol in Section 7.4. We further present the proofs of RAC freerider resiliency, as well as the proofs of RAC anonymity guarantees in Section 7.5. We finally present the experimental evaluation of RAC in Section 7.6, and our concluding remarks in Section 9.8.

7.2 Definitions and related work

We start this section by the definition of anonymity properties guaranteed by anonymous communication protocols. We then review existing anonymous communication protocols.

7.2.1 Definitions

We rely in this chapter on the definitions introduced by Pfitzmann and Hansen [Pfitzmann and Hansen, 2008]. We consider a system composed of nodes that communicate with each other via communication channels. In such a system, a node is acting *anonymously* if it is impossible for an observer to distinguish this node from the other nodes present in the system. Specifically, researchers distinguish the following three anonymity properties. The first one is *sender anonymity.* This property holds if it is not possible to identify the sender of any given message. The second property is *receiver anonymity.* This property holds if it is not possible to identify the destination of any given message. The third property is *unlinkability.* This property holds if an observer is not able to identify a pair of nodes as communicating with each other.

The goal of an anonymous communication protocol is to guarantee one or more (preferably all) of these anonymity properties in presence of an opponent, i.e., a malicious entity trying to break these properties. All existing protocols, as well as the protocol we present in this chapter, consider the strongest possible opponent, called the *global* and *active* opponent. *Global* means that the opponent can monitor and record the traffic on all the network links. *Active* means that the opponent can control some nodes in the system and make them deviate from the protocol in order to reduce the anonymity of other nodes. The higher the number of nodes that the opponent must control to break a protocol, the stronger the anonymity guaranteed by this protocol. The only limitation of the global and active opponent is that it is not able to invert encryption.

7.2.2 Related work

The two pioneering protocols for anonymous communication are the DC-Net [Chaum, 1988] and Onion routing [Goldschlag et al., 1999] protocols.

The DC-Net protocol exhibits strong anonymity guarantees. Specifically, it is not possible for an opponent to break anonymity without controlling all the nodes executing the protocol. DC-Net reaches this objective by relying on the principle of secret-sharing. Specifically, nodes in DC-Net proceed in rounds. During one round, only one node is allowed to send a message. If two nodes send a message during the same round, there is a *collision* and none of the messages is correctly delivered. To avoid collisions, there exist mechanisms for reserving sending slots [Waidner and Pfitzmann, 1989, Golle and Juels, 2004]. To guarantee anonymity, DC-Net organizes nodes in a structured network and requires nodes to forward encrypted messages received from their neighbors. Nodes then rely on a XOR-based mechanism (that they apply on messages they receive from their different neighbors) to decrypt messages. The major limitation of DC-Net is that it yields a considerable overhead. Indeed, at every round, every pair of nodes in the system needs to exchange messages. This is why other protocols that aim at reducing the cost of DC-Net have been devised. The Herbivore protocol [Goel *et al.*, 2003] is one such protocol. In this protocol, nodes are organized in groups, which limits the exchange of messages inside the groups. Despite this optimization, this protocol is considered as unusable in practice as soon as the system grows to more than (approximately) 50 nodes, as analyzed in [Edman and Yener, 2009].

The onion routing protocol guarantees a lower degree of anonymity than DC-Net. It is nevertheless considered secure enough to be widely used in practice [Dingledine et al., 2004]. Furthermore, it exhibits way better performance. This protocol works as follows. A node wanting to send a message randomly selects a list of nodes (called relays) and encrypts the message into multiple layers of encryption : one layer for every relay. The resulting encrypted message is called an onion. The node then sends the onion to the first relay that deciphers the first layer. This layer contains the address of the second relay and an inner onion, encrypted with the key of the second relay. The first relay sends the inner onion to the second relay. This process is repeated until the message reaches its destination. Note that because of its onion structure, a forwarded message changes at each relay. Various variants of this protocol have been proposed in the literature, such as crowds [Reiter and Rubin, 1998], cashmere [Zhuang et al., 2005], Tarzan [Freedman and Morris, 2002], and TOR [Dingledine et al., 2004]. The three first protocols aim at better tolerating churn. The last paper describes a popular implementation of the onion routing protocol.

The major limitation of these protocols is that they assume that nodes participating in the system are *altruistic*, i.e., they follow the protocol. However, anonymous communication protocols are subject to nodes that freeride in order to benefit from the system, without contributing their fair share to it. This explains why a recent attention has been given to the design of freerider-resilient protocols, in which all nodes are obliged to participate, otherwise risking eviction.

The first protocol that follows this direction is the Dissent v1 protocol [Corrigan-Gibbs and Ford, 2010]. This protocol relies on the principles of DC-Net to which it adds a double encryption system. This system allows stopping the execution of a round whenever a node detects the misbehavior of another node (including freeriding). It then allows exposing the misbehaving node without breaking anonymity. While this protocol is resilient to freeriding, it suffers from the same performance limitations as DC-Net, as further analyzed in Section 7.3. A second protocol, i.e., Dissent v2 [Wolinsky et al., 2012], has very recently been proposed to address this performance issue. The key idea behind this protocol is to run Dissent v1 on a small number of trusted servers. These servers are then used by a large number of nodes (called clients in [Wolinsky et al., 2012]) to enforce the anonymity of their communications. To reach this objective, nodes need to trust the servers they use to exchange messages, which is a strong assumption. Performance wise, as analyzed in the following section, even though Dissent v2 exhibits better performance that Dissent v1, it still suffers from a scalability issue.

7.3 The Case for a New Protocol

In this section, we motivate the need for a new anonymous communication protocol. We show, using simulations, that the two existing freerider-resilient anonymous communication protocols achieve poor

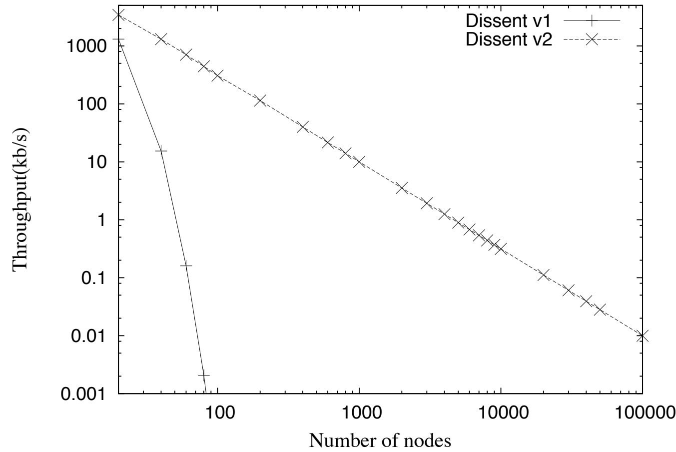


FIGURE 7.1 – Throughput as a function of the number of nodes for Dissent v1 and Dissent v2.

performance in large-scale systems. We simulate (using the Omnet++ simulator), a network of nodes connected via 1Gb/s links [16]. In our simulation, each node chooses a random node to which it sends anonymous messages of a fixed size equal to 10kB, at the highest possible throughput it can sustain. We run this experiment several times with an increasing number of nodes, and we measure the average throughput at which nodes receive anonymous messages.

We configure Dissent v2 with the optimal number of trusted servers for each network size. Specifically, increasing the number of trusted servers allows reducing the number of messages processed by each trusted server. However, it also increases the number of broadcast messages exchanged between trusted servers, and the size of these broadcasts. For each different number of nodes in the system, there is thus an optimal number of servers to be used, which maximizes the throughput of the protocol. Moreover, in order to balance the load, we equally distribute the number of nodes between trusted servers.

The throughput as a function of the number of nodes in the system for both Dissent v1 and Dissent v2 is depicted in Figure 7.1. From these results, we can first observe that the throughput of nodes in Dissent v1 drops down to (almost) zero when the number of nodes is higher than 50. The reason is that for each anonymous communication, every node needs to send a message to all the other nodes in the system. This requires the sending of what is equivalent to N broadcast messages involving all the nodes, where N is the number of nodes in the system. Let us note the cost of Dissent v1 as $N * Bcast(N)$ [17], which refers to the fact that for each anonymous communication, N broadcast messages are sent in a group of N nodes [18]. It becomes thus obvious why Dissent v1 exhibits such poor performance when the number of nodes in the system increases. Indeed, both the number of broadcast messages sent in the network, and the size of the broadcast groups depends on the system size.

We can also observe from Figure 7.1 that the throughput reached by nodes in Dissent v2 is higher than the throughput reached by nodes in Dissent v1, which was the key objective of this protocol. Nevertheless, the throughput of Dissent v2 decreases when the number of nodes in the system increases. The reason of this decrease is that, for each anonymous communication, Dissent v2 requires the sending of S broadcasts (where S is the number of trusted servers) in a group of size S servers, plus one broadcast in a group of size $\frac{N}{S}$. Hence, the cost of Dissent v2 is equal to $Bcast(\frac{N}{S}) + S * Bcast(S)$, which is also dependent on N.

We conclude from that study that existing freerider-resilient anonymous communication protocols are not scalable. This motivates the development of a new protocol that scales, i.e. a protocol that has a cost independent from the number of nodes in the system.

16. This ideal network configuration allows us to measure the maximum throughput each protocol can reach.

17. In the remaining of the paper, we adopt the following notation : we write "the protocol P has a cost of $x * Bcast(y)$" to refer to the fact that for each anonymous communication done using P, x broadcast messages are sent in a group of y nodes.

18. For the sake of simplicity we assume that all broadcast messages have approximately the same size.

7.4 The *RAC* protocol

In this section, we present *RAC*, a freerider-resilient, anonymous communication protocol that scales to large systems. Scalability stems from two key ideas : (1) the reduction of the number of messages broadcast in the system, and (2) the reduction of the size of the broadcast groups. We first explain how we achieve these two key ideas. We then provide a detailed description of the protocol.

7.4.1 Key idea #1 : reducing the number of broadcasts

In order to be more efficient than Dissent v1 and Dissent v2, we decided to start the design of our protocol from the principle of the onion routing protocol. Indeed, this is the most efficient available protocol existing today. As described earlier, in onion routing, the sender of a message selects a subset of nodes to act as relays. Although the anonymity of onion routing, which is a function of the number of relays, is lower than the one of protocols such as DC-Net, it is strong enough to be widely used in practice. The problem with existing onion-based protocols is that freeriders have no interest in acting as relays ; they will thus drop the messages they are supposed to relay whenever they can (i.e. when it does not endanger their anonymity). It is thus necessary to find a way to monitor the behavior of relays. The only node that knows all the relays on the path of an onion and that can identify all the layers of the onion is the sender of the message. The question is : *how to make the sender monitor the relays without disclosing its identity ?* Indeed, if a relay knows that a node is using it as a relay, it can break the sender anonymity. Our solution is to have senders and relays broadcast the messages they send or relay. Broadcasting messages allows senders to receive the messages broadcast by relays and to check that relays forward correctly, without disclosing their identity (every node receives the messages).

The broadcast exchanges must be reliable, despite the presence of freeriders and opponent nodes. In order to force nodes to forward messages, the broadcast protocol we propose relies on a structured network similar to the one used in the Fireflies group membership protocol [Johansen et al., 2006]. Specifically, nodes are placed on several virtual rings using a hash function. On each ring, a node has a predecessor node and a successor node. The broadcasting protocol works as follows : each time a node receives a message from one of its predecessors, it forwards it to all its successors. A node thus expects to receive each given message from all its other predecessors. If a node does not receive a given message from one of its predecessors, it considers the latter as a freerider. This forces nodes to forward all the messages they receive. Note that a node also verifies that its predecessors are broadcasting at a constant rate, because this is required to ensure anonymity, as explained in the detailed protocol description.

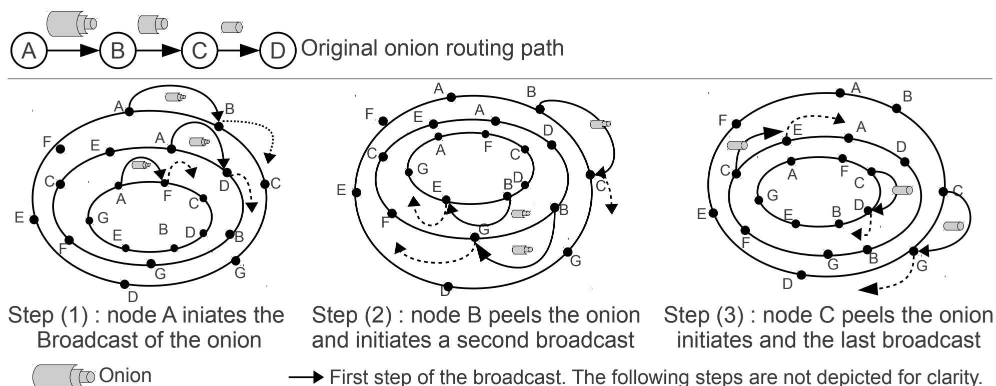


FIGURE 7.2 – Illustration of the *RAC* protocol.

Figure 7.2 illustrates the dissemination of a message in *RAC*. A node A that wants to send a message to a node D builds an onion (containing two relays B and C in that case). The node then broadcasts the onion using the multiple ring structure. Upon the reception of the onion, each node first forwards the onion to its successors. Each node then tries to decipher the onion. If a node manages to decipher the

onion and it obtains a new onion, this means that the node is a relay and it will also broadcast the new onion (this is the case of node B and then of node C). If a *node* obtains a clear message, this means that it is the destination (this is the case of node D). If a node does not manage to decipher the onion, it does not do anything else than forwarding the onion to its successors.

The resulting protocol has a cost of : $L * R * Bcast(N)$, where $L << N$ is the number of relays employed in the onion path, and $R << N$ is the number of rings used in the broadcast protocol. In the experiments presented in Section 7.5, we use $L = 5$ and $R = 7$. As we show, these values are enough to ensure a high level of anonymity : receiver anonymity and unlinkability are optimal, and an opponent only has a probability of $9.9 * 10^{-7}$ to break the sender anonymity.

With the protocol presented in this section, we have reached the first objective : making the number of messages that are broadcast in the system independent of N. In the next section, we explain how we reach our second objective : making the size of groups in which messages are broadcast independent of N.

7.4.2 Key idea #2 : reducing the size of broadcast groups

In order to make the size of groups in which messages are broadcast independent of N, we propose to cluster nodes into groups of approximately the same size, say G [19]. The identifier of the group to which a node should belong is computed deterministically by the node when it first joins the system (e.g., using a hash of its public key modulo the number of groups). If two nodes that belong to the same group want to communicate they use the protocol we saw in the previous section, inside the group they belong to (of size G) instead of running it in the whole system. The cost is thus reduced to $L * R * Bcast(G)$. The remaining question is : *How do we enable the communication between two nodes that do not belong to the same group ?* A straightforward solution would be to run the protocol presented in the previous section in a supergroup composed of the union of the two groups to which these nodes belong. The resulting cost would thus be $L * R * Bcast(2 * G)$. We adopt in our protocol a more optimized solution described below.

The sender sends the anonymous message by running the protocol in his group, as if the destination was part of the same group. It nevertheless sets a marker in the innermost onion for the last relay, to inform him about the group Id of the destination node. This allows sending most broadcasts inside a group of size G. Upon receiving the innermost onion, the last relay needs to perform two actions. First, it needs to forward the message to the destination node, which belongs to another group. Second, it needs to inform the sender that it effectively forwarded the message. To perform these two actions at once, the last relay broadcasts the message in a super group constituted of the union of the two groups, i.e., its group and the group of the destination. This super group is what we call a *channel* in the remaining of the paper. As such, the overall cost of the protocol is equal to $(L - 1) * R * Bcast(G)$ for the first part of the protocol, which executes inside the sender's group plus $R * Bcast(2G)$ for the broadcast that the last relay executes inside the channel. As $Bcast(2G) = 2 * Bcast(G)$, the overall cost of our protocol is equal to $(L + 1) * R * Bcast(G)$, which is lower than the original $L * R * Bcast(2 * G)$ because for the commonly chosen values of L, $L + 1 < 2 * L$.

7.4.3 Detailed description

In this section, we provide a detailed description of the protocol.

Joining the system. Each node in the system has an identifier (ID), a view containing the list of the nodes present in the system, and two private/public key pairs. The first pair of keys is linked to the node ID. We call these keys the ID keys. The second pair of keys is used to encrypt messages for their destination (using the key of the destination). This pair of keys cannot be linked to the node ID. We call these keys the pseudonym keys. The way nodes learn about pseudonym keys is application-dependent. For instance, in an anonymous publish-subscribe system, nodes would subscribe to a given topic using their public pseudonym key.

19. Note that using smaller groups (size G instead of N) gives an opponent that captures a message more information about the possible senders and receivers of this message. Specifically, the opponent knows that the sender and the receiver of the given message is one among the G nodes of the group, instead of one among the N nodes of the system. This is nevertheless not an issue if G is big enough. In our experiments, we use $G = 1000$.

To join the system, a node n sends a JOIN request to a node x that is already part of the system. This JOIN request contains the ID public key of n, denoted by K, and ID, the identifier of n, which determines the group that n will join. To compute this identifier, we use a solution inspired by the Herbivore system [Goel *et al.*, 2003]. Specifically, let f and g be two one-way functions. The new-coming node has to generate random vectors until it finds a vector $y \neq K$ such that the least significant mk bits of $f(K)$ are equal to those of $f(y)$. The value $g(K, y)$ gives n the value of its ID. An opponent node may want to circumvent the system to join a particular group. The solution we use is robust to this attack, because it is difficult for a node to obtain the values of K and y that are necessary to join a given group, provided that the functions g and f are one-way functions.

Once x has received the JOIN request from n, it computes which group n should join (for example, the group containing the node with the nearest ID). Then, x anonymously broadcasts the JOIN request to this group. Upon receiving this request, all nodes of the group verify that the ID of n is correct. If the ID is not correct, the request is ignored ; otherwise, the nodes add n to their view and compute the positions of n in the various rings used to broadcast messages. This is done as in the Fireflies protocol [Johansen et al., 2006] : the position of a node on the ith ring is determined by the hash of the couple (ID, i). The number of rings to create depends on the size of the system, as well as of the percentage of opponent nodes that is assumed in the system. More precisely, the successor set of each node (which is, for a given node, a set comprising the successor of this node on the various rings) should contain a majority of non-opponent nodes, and this majority should be large-enough to ensure reliable dissemination of broadcast messages [20]. Concretely, in a system comprising 1000 nodes and assuming that 10% of the nodes are opponent nodes, it is enough to use 7 rings to ensure that the successor sets will contain less than 3 opponent nodes with a probability $0,999$.

After a period T, which corresponds to the maximum time necessary for a message to reach all nodes in the group, x sends a READY message to n, signaling that all nodes have been informed of its arrival in the system. When n receives the READY message, it sends a message to its followers and predecessor to indicate them that they can use it as their new follower or predecessor. Other nodes than n's predecessors and followers need to wait a period equal to $2T$ from receiving n JOIN request before using n as a relay (to make sure that n completed the join procedure). Once n has joined a group, the nodes of this group broadcast the JOIN messages in the channels they belong to. This way, all nodes in the system are informed of n's arrival and n can find its position in each channel.

Managing groups. To guarantee a lower bound on anonymity and an upper bound on the cost of the protocol, groups must have a size bigger than s_{min}, and smaller than s_{max}, which are system parameters. When the size of a group becomes lower than s_{min}, nodes belonging to the group broadcast a message indicating that the group should be dissolved. They then rejoin the system in order to be assigned to another group. When nodes join the system and a group becomes larger than s_{max}, nodes of this group broadcast a message indicating that the group should be split in two. Nodes then compute which of the two new groups they should join. Specifically, nodes with the lower IDs go in the first group, and nodes with the higher IDs go in the second group. Finally, each node computes its position in the various rings of the group and connect to its new followers and predecessors.

Sending a message. To send a message, a node ciphers the message with the public pseudonym key of the destination node. It then randomly chooses L public ID keys of nodes of its group and ciphers the message in successive layers (called onions). Each layer contains a flag that allows a node to know whether it successfully deciphered it. If the message destination is in another group, the innermost layer also contains a marker indicating the channel to which it should be broadcast. Once the L layers have been created, the sender pads the message to reach a defined size, and initiates a broadcast by sending the message to its direct successors on the different rings of its group. Messages are padded because it makes it impossible for opponent nodes to use the size of network packets to track the path followed by a given message. Note that, similarly as in the onion routing protocol, to preserve the anonymity of the sender, nodes must send or forward messages at a constant rate, defined by the system. If a node does not have messages to send or forward, it must create fake messages (called *noise* messages) and send them in

20. To ensure reliable dissemination in a system with N nodes, each node should have at least $log(N) + c$ non-opponent nodes in its successor set, where c is a constant [Kermarrec et al., 2003].

the system, following the above-described procedure.

Receiving a message. Upon reception of a message, a node checks if it has already received the message. If that is not the case, the node forwards the message to its direct successors on the different rings of the group or channel in which the message is broadcast. The node then checks whether it can decipher the message using its private ID key. In case of success (the node is able to read the flag put by the sending node), this means that the node should act as a relay for this message. Consequently, it checks if there is a marker indicating that the message should be broadcast in a channel. It then pads the deciphered message and sends it to its direct successors on the different rings of the appropriate group or channel. Otherwise, the node checks whether the message is intended to it by trying to decipher it using its private pseudonym key. If that is the case, it delivers the message. Note that only innermost layers are broadcast in the channels, so as when they receive a message on a channel, nodes only have to check if they can decipher it with their private pseudonym key.

Checking the misbehavior of nodes. In order to discourage freeriding, nodes check that (1) the relays they use to send their own messages correctly forward messages, (2) the nodes that directly precede them in the different rings of channels and group correctly forward messages (i.e. once and only once), and (3) the nodes that directly precede them in the different rings of their group send messages at a constant rate. Every time a node discovers a misbehavior, it locally blacklists the corresponding node. Each node maintains several blacklists : a blacklist per channel for suspected predecessors, a blacklist for their group for suspected predecessors, and a blacklist for suspected relays. We explain later how these blacklists are used to evict nodes from the system.

The first check is performed as follows. When a node sends one of its own messages, it keeps a copy of the various layers of the message, together with the public ID keys they have been ciphered with. It then expects to receive the messages corresponding to the different layers before the expiration of a timer (recall that all messages are broadcast in the system). The first relay, if any, that does not correctly decipher and forward the message, is suspected and added by the sender to the relays blacklist and it is not used anymore by this node.

The second check is performed as follows. Provided that messages are broadcast, for each message, a node expects to receive a copy from each of its direct predecessors in the channel or group in which the message is broadcast. If a predecessor does not send a copy of a given message within a bounded time [21], of if it sends a given message twice [22], it will be suspected by its successors, who will add it to the appropriate predecessors blacklist.

The third check is performed as follows. Each node checks that it receives messages from its direct predecessors on the different rings of its group at a constant rate. Whenever a node detects a misbehaving predecessor, it adds it to the predecessors blacklist.

Evicting nodes. As described above, nodes maintain two kinds of blacklists : a blacklist for suspected relays, and several blacklists for suspected predecessors. Nodes disseminate these blacklists as follows. The relays blacklist is disseminated periodically to the node's group. Because it can disclose information about the identity of a message sender, this list has to be disseminated anonymously. To avoid an attack where malicious nodes send more than one blacklist at each round, we use the shuffle protocol of Dissent v1 which allows permuting a set of fixed-length messages and broadcasting the set to all members with cryptographically strong anonymity. The predecessors blacklists are disseminated as clear messages in the channels or group to which they correspond.

A node A removes a node B from its view as soon as it collects evidence that : (1) B belongs to the predecessors blacklist of $(t+1)$ of B's followers in one channel or group, with t the maximum number of opponent followers that node B can have (as defined by Fireflies [Johansen et al., 2006]), or (2) B belongs to the relays blacklist of $(f+1)$ nodes of B's group, with f the maximum number of opponent nodes in a group. Further, if B is one of A's predecessors or followers, it replaces it with a new predecessor or successor deterministically computed from the view updated after the eviction of B.

When a node is evicted from a group or a channel, the nodes of its group must broadcast messages to all the channels it belonged to. This message informs the nodes of the channels that the node was

21. Our implementation uses TCP, which ensures reliable delivery between pairs of nodes.
22. The node could be performing a replay attack [Pries et al., 2008].

evicted. Nodes that fail at sending this message are suspected by the nodes of their group. Nodes in the channels wait to received $(f+1)$ eviction notifications to take this eviction into account.

7.5 Proofs

In this section, we first prove that *RAC* ensures anonymity. Then we prove that *RAC* is freerider-resilient.

7.5.1 Anonymity proof

We prove that *RAC* ensures sender anonymity, receiver anonymity, and unlinkability. We use the following notations : the network size is denoted by N, the groups size is denoted by G, the fraction of opponent nodes is denoted by f, and the number of relays used to disseminate each message is denoted by L. We explained in section 7.4.2 that dividing nodes into groups reduce nodes anonymity from one among N to one among G. We now show how this anonymity of one among G is protected. We first assume that opponent nodes are passive, which means that they follow the protocol and can only gather information by monitoring the network. We then consider the case of active opponents nodes that are willing to deviate from the protocol in order to break anonymity.

Passive opponent nodes

Sender anonymity As nodes in *RAC* broadcast messages at a constant rate, it is not possible to know if they are forwarding messages, sending messages or sending noise. Thus, an opponent node receiving a message can only discover the sender of this message when it colludes with all the relays of that message. This is quite hard to achieve. In fact, the probability for a correct node to build a path only containing opponent nodes is equal to $\prod_{i=0}^{L} \frac{X-i}{G-i}$, with X the number of opponent nodes in its group. Thus, an opponent that tries to break sender anonymity should control a number X of nodes as large as possible in the targeted node's group. But, as nodes are randomly spread among the groups, the probability that the opponent control X nodes in a given group is equal to $\prod_{i=0}^{X-1} \frac{fN-i}{N-i}$. As a result, the probability that the opponent break the sender anonymity of a given node is equal to $\max_X(\prod_{i=0}^{L} \frac{X-i}{G-i-1} * \prod_{i=0}^{X-1} \frac{fN-i}{N-i})$. With $N = 100.000$, $G = 1000$, $f = 5\%$, and $L = 5$ this probability is equal to $5.7 * 10^{-25}$, which is extremely low.

Receiver anonymity When a node sends a message m to a destination node n_D, it ciphers m using n_D's public pseudonym key, and broadcasts it using a set of relays. Note first that, as explained in Section 7.4, the public pseudonym key of n_D cannot be linked to n_D. The only node that is able to decipher the message m is node n_D, using its private pseudonym key. Nodes cannot distinguish n_D from other nodes (i.e. they have no way to know that it was able to decipher the message). For an external observer, n_D behaves like every other nodes in the system : it forwards the message m once and only once to all its direct successors in the different rings. Consequently, no node is able to detect that n_D is the destination of message m. This is optimal : to break receiver anonymity the opponent must control all the nodes of the group but one.

Unlinkability As we have seen before, *RAC* ensures optimal receiver anonymity in the groups. Consequently, it is impossible for a node to determine the destination of a given message m within a group. It is thus impossible for an external observer to know whether two nodes are communicating or not. At best, an opponent can know that two nodes are communicating using a channel. But it cannot know which nodes are communicating. Hence, the protocol ensures unlinkability.

Active opponent nodes

Sender anonymity An active opponent can try to break sender anonymity : (1) by trying to force nodes to build relay paths only containing opponent nodes, or (2) by trying to evict some nodes from

the system. Evicting nodes can be used to reduce the number of non-malicious nodes in the system, or to render the system prone to intersection attacks [Raymond, 2001] by comparing sent messages before and after the eviction of some nodes.
Case 1 : an opponent node acting as relay can drop the messages it is supposed to broadcast. This forces the sending node to build new paths and, thus, increases the probability that the node build a path that is totally composed of opponent nodes. Nevertheless, this is not an easy task since, if an opponent node drops a message, it will be blacklisted by the sender that will not use it as a relay anymore. Consequently, provided there is a fraction f of opponent nodes, if they coordinate their actions, and if each time a new path is created, it contains an opponent node, then they will be able to force at most fN new paths to be created. This makes the probability of building a complete path of opponent nodes very low. For instance, in a system with $N = 100.000$, $G = 1000$, $f = 5\%$ and $L = 5$, the probability that opponent nodes manage to force one single node to build a path only composed of opponent nodes is at most $2.8 * 10^{-23}$. This is extremely low.

Case 2 : an opponent node can try to break sender anonymity by evicting nodes from the system (e.g. to allow intersection attacks [Raymond, 2001] to be run). In RAC, a node n is evicted if : (1) $fG + 1$ nodes notify their group that the node n misbehaves, or (2) a majority of the direct successors of node n on the different rings notify their group or channel that node n misbehaves. Let us consider the first case. There are fG opponent nodes in the network. Consequently, opponent nodes alone cannot force the eviction of node n by sending a wrong notification to the group. They need to behave in such a way that at least one non-opponent node will also send a notification regarding node n. The only thing they can do is to try to stop the dissemination of some messages, so that node n will not be reached and will possibly be blacklisted (e.g. if n was a relay for that message). Nevertheless, as shown in [Johansen et al., 2006], it is possible to increase the reliability of the broadcast by increasing the number of rings. It is thus possible to use a number of rings guaranteeing that opponent nodes will have an arbitrarily low probability to succeed in evicting node n by making a non-opponent node send a notification against it. Let us now consider the second case. As explained in Section 7.4, the higher the number of rings, the lower the probability that node n will have a majority of opponent nodes in its set of direct successors. Consequently, it is possible to use a number of rings guaranteeing that opponent nodes will have an arbitrarily low probability to succeed in evicting node n due to a majority of opponent nodes in its set of direct successors. For exemple with $f = 5\%$, a number of rings equal to 7 guarantees that each node has a probability lower than $6.0 * 10^{-6}$ to have a majority of opponent nodes in its set of direct successors.

Receiver anonymity Opponent nodes can try to break receiver anonymity using the same attacks as the ones described in the proof of the "sender anonymity" property. As we have seen, these attacks do not succeed. Consequently, receiver anonymity is not impacted by the presence of active opponent nodes.

Unlinkability The proof is similar to the one made for passive opponent nodes.

7.5.2 Freerider-resiliency proof

In this section, we prove that RAC tolerates freeriders, i.e. that freeriders do not have any interest in deviating from the protocol. To prove this, we prove that RAC provides a *Nash equilibrium* [Nash, 1951]. In a Nash equilibrium, no node has an incentive to unilaterally deviate from the equilibrium, assuming every other node follows the protocol. Note that the proof presented in this chapter makes similar assumptions as those presented in previous works related to freeriders, e.g., [Aiyer et al., 2005,Li *et al.*, ,Mokhtar et al., 2010].

Assumptions on freeriders

To reason about freeriders, it is necessary to formalize their behavior. In the case of RAC, the benefit of a node n depends of the following parameters :
- (A) reducing the risks of compromising its anonymity.
- (T) succeeding in sending its own messages.

- ◦ (R) receiving messages that are intended to it.
- ◦ (F) forwarding as few messages as possible.
- ◦ (C) ciphering as few messages as possible.
- ◦ (D) deciphering as few messages as possible.

We can define the overall benefit of a node n as $B = \alpha A + \beta T + \gamma R + \delta F + \omega C + \phi D$, where $\alpha \approx \beta \approx \gamma \gg \delta \approx \omega \approx \phi$. Intuitively, this means that nodes do not want to trade-off their anonymity and the reliable transmission of their own messages against a lower bandwidth and CPU consumption. Moreover, freeriders are assumed not to collude ; freeriders expect opponent nodes to try to decrease their benefit as much as possible ; freeriders expect other nodes to follow the protocol.

Nash equilibrium proof

Theorem 1 *The RAC protocol provides a Nash equilibrium.*

We prove the above theorem following the approach adopted by related work [Aiyer et al., 2005, Li *et al.*,] : we decompose the theorem into a set of lemmas representing the protocol steps, and we explain why it is in the best interest of a freerider to follow the protocol at each given step. The proof of Theorem 1 directly follows from the proofs of the different lemmas.

Lemma 1 *A freerider always sends messages to all its direct successors in the different rings of a channel or group.*

A freerider knows that up to half of its direct successors but one can be opponent nodes and can thus send wrong notifications to the group or channel. Consequently, a freerider knows that it risks eviction if at least one correct successor sends a notification to the group or channel. Consequently, a freerider sends messages to all its direct successors in the different rings.

Lemma 2 *A freerider always correctly forwards the messages it acts as a relay for.*

A freerider n knows that the sender of a message will notify the group if it does not receive the message that node n was supposed to broadcast as a relay. Provided that there are fG opponent nodes in the group that can send wrong notifications to the group, node n knows that it is enough to be suspected by one non-opponent node to be evicted from the system. Consequently, node n correctly forwards the messages it acts as a relay for.

Lemma 3 *A freerider always checks that its predecessors are sending every message once and only once.*

A direct predecessor that does not send a given message can be an opponent node trying to run an *(N-1)-attack* [Serjantov et al., 2003] on the node. As a result, freeriders always checks if all direct predecessors send them all messages they are supposed to. A direct predecessor that sends a given message twice can be an opponent node trying to run a *replay attack* [Pries et al., 2008] on the node. As a result, freeriders always checks that their direct predecessors do not send the same message twice.

Lemma 4 *A freerider sends the list of nodes it suspects during the periodic anonymous blacklist broadcasting.*

As shown in [Corrigan-Gibbs and Ford, 2010], the anonymous blacklist broadcasting protocol we rely on is accountable. Consequently, a freerider participates in it to avoid eviction by the network. Moreover, messages sent in this protocol have a fixed-size. Consequently, a freerider does not gain bandwidth or CPU by sending a message containing wrong information. A freerider thus sends the exact list of nodes that it suspects.

Lemma 5 *A freerider broadcasts the join requests it receives from nodes that want to enter in the system.*

A freerider has interest in helping nodes to enter the system because that increases its anonymity. In fact, the more nodes in the system, the more difficult for an opponent to control a significant of the network. Moreover, helping nodes to join the system ensures that opponent nodes are not the only ones to control who can join the system. If only opponent nodes control who can join the system, they may only accept opponent nodes, and this way, increase their chances of breaking anonymity. Finally, freeriders have interest in helping nodes they want to communicate with to join the system.

Lemma 6 *A freerider always sends new messages (possibly noise) at the rate required by the protocol.*

A freerider n knows that if it omits to send new messages (possibly noise) at the rate defined by the protocol to some of its direct successors, the latter will accuse it. Moreover, node n knows that up to half of its direct successors but one can be opponent nodes and can thus produce wrong accusations. Consequently, node n knows that it risks being evicted if at least one correct successor sent a notification to the group. Consequently, node n sends messages at the required rate.

Lemma 7 *A freerider always checks if its predecessors send new messages at the rate defined by the protocol.*

A freerider notifies the group or channel if one of its direct predecessors sends a message at a higher rate than that defined by the protocol. Indeed, such a behavior increases its bandwidth and CPU consumption. Moreover, a freerider notifies the group if one of its direct predecessors sends messages at a lower rate than required as this can be the sign of an opponent running an attack.

7.6 Evaluation

In this section, we evaluate both the performance and the anonymity of *RAC*. We evaluate the former using simulations and the latter using the formulas presented in Section 7.5. We use simulations in order to be able to evaluate the performance of our protocol in configurations comprising up to 100.000 nodes. The objective of this evaluation is to answer the following two questions :

- What is the throughput achieved by *RAC*, Dissent v1, and Dissent v2 ?
- What are the anonymity guarantees ensured by *RAC*, Dissent v1, and Dissent v2 ?

We start by a presentation of the simulation settings and the configuration parameters used for the various protocols. We then reply to the two questions mentioned above.

7.6.1 Simulation settings

We performed simulations using Omnet++ [omn, 2013], a discrete event simulator written in C++. Using Omnet++, we simulate a network of nodes interconnected by a router. Nodes are connected to the router using 1Gb/s links. We use this ideal network configuration as it allows evaluating the maximum throughput that each protocol can achieve. We plan to evaluate the complexity of *RAC* and its performance in a real setting as part of our future work.

7.6.2 Protocol configuration parameters

In all the evaluations, we consider two configurations for *RAC* : a configuration without group (i.e. all nodes actually belong to a single group), and a configuration with groups comprising 1000 nodes each. We refer to these two configurations as *RAC*-NoGroup and *RAC*-1000, respectively. In both configurations, we used seven rings for the broadcast protocol, i.e., $R = 7$ and five relays for the onion paths, i.e., $L = 5$.

For each system size, we configure Dissent v2 with the optimal number of trusted servers (as explained in Section 7.3).

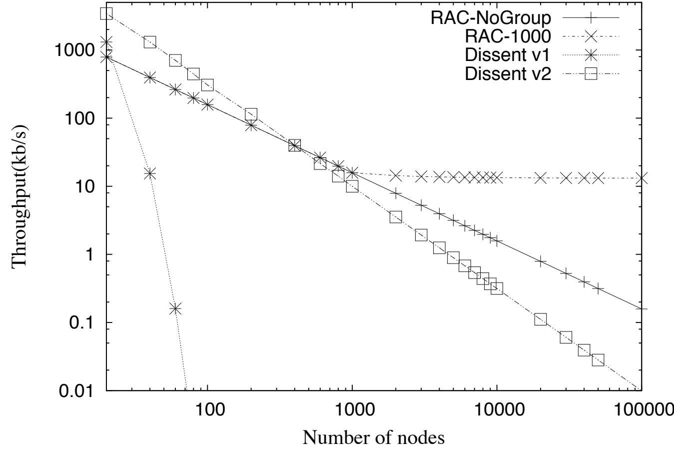


FIGURE 7.3 – Throughput as a function of the number of nodes in the system for Dissent v1, Dissent v2, RAC-NoGroup and RAC-1000.

7.6.3 Throughput

In order to assess the throughput of the various protocols, we run the following experiment. We consider a system comprising N nodes. Each node randomly selects a destination node and sends anonymous messages to this node at the maximum throughput it can sustain. Messages have a fixed size of 10kB. We measure, for each protocol, the average throughput at which the N nodes receive anonymous messages. Figure 7.3 shows the throughput measured for RAC-NoGroup, RAC-1000, Dissent v1, and Dissent v2 as a function of N. Note that in this situation, with an onion path length of 5, the throughput provided by onion routing is 200Mb/s.

We first observe that both RAC configurations achieve a better throughput than the two versions of Dissent when there are more than 1000 nodes in the system. For instance, when the system contains 100.000 nodes, the throughput of RAC-NoGroup (resp. RAC-1000) is 15 times (resp. 1300 times) higher than that of Dissent v2. With this system size, Dissent v1 is so slow that we were not even able to observe a single message delivery.

We also observe that RAC-1000 scales : in systems comprising more than 1000 nodes, its throughput is not impacted by the size of the system. This was expected. Indeed, adding nodes in the system does not increase the number of broadcasts required for each message that is anonymously sent, nor the size of the broadcast groups that are used.

Finally, we observe that when N is smaller than 1000, both RAC configurations achieve the same throughput. This comes from the fact that when N is smaller than 1000, RAC-1000 only uses one group, and does thus execute the exact same protocol as RAC-NoGroup.

To summarize, unlike other protocols, RAC-1000 scales : performance do not decrease when increasing the system size.

7.6.4 Anonymity guarantees

We compare in this section the anonymity guaranteed by the two configurations of RAC against Dissent v1, Dissent v2 and the onion routing protocol in the case of a passive opponent. Results are depicted in Table 7.1. The numbers depicted in this table result from the instantiation of the formulas characterizing the anonymity guarantees of each protocol.

The first line of the table represents the size of the set to which the sender or the receiver of a given message belong. The higher the size, the better. This value should not be too small as it can disclose a lot of information regarding the identities of the sender and receiver of a message. For instance, if this value is equal to 10, this means that the probability that a given node be the sender or the receiver of a given message is $\frac{1}{10}$. From the table, we can observe that this value is the highest possible for all protocols but

System with 100.000 nodesSystem with 100.000 nodes			Dissent v1	Dissent v2	Onion	*RAC*-NoGroup	*RAC*-1000
Anonymity : one amongAnonymity : one among			100.000	100.000	100.000	100.000	1000
Probability to break a given node anonymity of type T when controlling P% of the nodes	% of opponent nodes (P)	Anonymity type (T)					
	90%	Sender	0	0	0.53	0.53	$7.1 * 10^{-11}$
		Receiver	0	0	0.53	0	$1.1 * 10^{-46}$
		Unlinkability	0	0	0.53	0	$1.1 * 10^{-46}$
	50%	Sender	0	0	$1.5 * 10^{-2}$	$1.5 * 10^{-2}$	$1.8 * 10^{-16}$
		Receiver	0	0	$1.5 * 10^{-2}$	0	$1.2 * 10^{-303}$
		Unlinkability	0	0	$1.5 * 10^{-2}$	0	$1.2 * 10^{-303}$
	10%	Sender	0	0	$9.9 * 10^{-7}$	$9.9 * 10^{-7}$	$7.3 * 10^{-22}$
		Receiver	0	0	$9.9 * 10^{-7}$	0	$5.8 * 10^{-1020}$
		Unlinkbility	0	0	$9.9 * 10^{-7}$	0	$5.8 * 10^{-1020}$

TABLE 7.1 – Anonymity guarantees of the various protocols in a system of 100.000 nodes.

RAC-1000, for which it is equal to 1000, i.e., the size of the groups. We believe that 1000 is a big enough value in most contexts. This value can be increased if required by RAC users (using the s_{min} parameter presented in the detailed protocol description).

The table is then structured in three subparts, corresponding to the proportion of nodes controlled by the opponent (P in the table). We consider three values of P : 10%, 50%, and 90%. For each value of P, we compute the probability that the opponent be able to disclose the identity of the sender, the receiver or to link the sender and the receiver of a given message. Results show that these probabilities are equal to zero for both Dissent v1 and Dissent v2. Indeed, in these protocols, the opponent must control all the nodes in the system (or all the trusted servers in Dissent v2) to break anonymity. We also observe that in both configurations of RAC, these probabilities, although not (always) null, are extremely low, making RAC an extremely robust protocol.

We also observe that, counter-intuitively, the probability that an opponent break anonymity in RAC-1000 is lower than in RAC-NoGroup. This is due to the fact that in RAC-1000, a node cannot choose the group to which it belongs. Consequently, an opponent needs to control almost all the nodes in the system for having enough nodes in the same group in order to break anonymity within that group.

Finally, we observe that RAC-1000 provides stronger anonymity guarantees than onion routing in all cases. This is due to two reasons. First, RAC-1000 has a better sender anonymity than onion routing due to the fact that it uses groups (see explanation above for RAC-NoGroup vs RAC-1000). Second, to break the receiver anonymity in onion routing, the opponent must only control the relays of the onion path. Instead, in our protocol, the opponent must control all the nodes of the destination group, which is less likely to happen. Better unlinkability follows from better receiver anonymity.

7.7 Conclusion

Two protocols for anonymous communication in the presence of freeriders have been recently proposed. Unfortunately, they do not scale : their performance decrease when increasing the number of nodes in the system. In this chapter, we present RAC, a freerider-resilient anonymous communication protocol that scales, while providing strong anonymity guarantees.

Chapitre 8

Seine : a framework for designing and injecting rational faults

Selfishness is one of the key problems that confronts developers of cooperative distributed systems (e.g., file-sharing networks, voluntary computing). It has the potential to severely degrade system performance and to lead to instability and failures. Current techniques for understanding the impact of selfish behaviours and designing effective countermeasures remain manual and time-consuming, requiring multi-domain expertise. To overcome these difficulties, we propose SEINE, a simulation framework for rapid modelling and evaluation of selfish behaviours in a given cooperative system. SEINE relies on a domain-specific language (SEINE-L) for specifying selfishness scenarios, and provides semi-automatic support for their implementation and study in a state-of-the-art simulator. We show in this paper that (1) SEINE-L is expressive enough to specify fifteen selfishness scenarios taken from the literature, (2) SEINE is accurate in predicting the impact of selfishness compared to real experiments, and (3) SEINE substantially reduces the development effort compared to traditional manual approaches. This work has been carried out in the context of the PhD thesis of Guido Lena Cota and has been published in the proceedings of the 47th IEEE/IFIP International Conference on Dependable Systems and Networks (IEEE DSN'17).

8.1 Introduction

Selfish behaviours are an inherent and critical problem of cooperative distributed systems such as peer-to-peer (P2P), grid and volunteer computing, and self-organising networks. These behaviours are performed by participants that benefit from the system without contributing their fair share to it [Hughes et al., 2005, Handurukande et al., 2006, Cunha *et al.*, 2013]. An example of selfish behaviour in a P2P live streaming system is to download a given video file without sharing it with other nodes in order to save local bandwidth. It has been measured that such behaviours may have severe impacts on the system performance, causing a significant degradation of its reliability and efficiency [Guerraoui et al., 2010a, Li *et al.*, , Cota et al., 2015]. For instance, in the above-mentioned live streaming system, if 25% of nodes free ride by not sharing the received video chunks, then half of the remaining nodes receive a degraded stream [Guerraoui et al., 2010a].

In this context, understanding the impact that selfish behaviours have on the system performance is crucial to the design of effective selfishness countermeasures. However, this can be done only by modelling and injecting selfish behaviours into the system under consideration, which is a non-trivial task. Indeed, to carry on this task, the system designer has to first analyse the functional specification of the considered system and identify those steps (e.g., functions) for which selfish nodes may behave in a non-cooperative way. Then, on each of the identified steps, the designer has to decide what are the possible selfish behaviours that are meaningful in the context of her application and implement the corresponding behaviours. Finally, the designer has to invest considerable effort in experiments to assess the impact of the introduced behaviours on the performance of the system.

For the purpose of these experiments, a designer can rely on frameworks for developing and evaluating

real distributed systems [Killian et al., 2007, Leonini et al., 2009] or simulations of such systems [Montresor and Jelasity, 2009, Basu et al., 2013]. However, existing frameworks do not provide any specific support for modelling and injecting selfishness, which has to be done manually. In practice, the designer hard codes both the control and logic of selfish behaviours into the parts of the system implementation that are affected by that behaviours. This activity typically results in generating variant implementations of the same system (i.e., one for each node behaviour), or in creating a single implementation which incorporates all the possible behaviours as well as the algorithmic functionalities for their control (e.g., variables, if-clauses). The increased complexity and redundancy of the source codes reduce their readability, maintainability, and evolution. Finally, once a system implementation is available, the designer usually proceeds with an extensive experimental campaign to quantify the harm caused by various selfishness manifestations of different proportions of selfish behaviours. To the best of our knowledge, such a domain-specific evaluation has to be conducted manually by the system designer, which is tedious and time-consuming.

In order to help system designers in this task, we propose *SEINE*, a framework for modelling various types of selfish behaviours in a given system and automatically understanding their impact on the system performance through simulations. *SEINE* relies on a *Domain-Specific Language* (DSL), called *SEINE-L*, for describing the behaviour of selfish nodes, along with an annotation library to associate such specification with a system implementation for the state-of-the-art simulator PeerSim [Montresor and Jelasity, 2009]. To design *SEINE-L*, we have conducted a domain analysis on state-of-the-art papers related to building selfish-resilient systems. *SEINE-L* provides a unified semantics for defining *selfishness scenarios*, which allow describing capabilities, interests and behaviours of different types of nodes participating in the system. The *SEINE* framework provides a compiler and the run-time system supporting the automatic and systematic evaluation of different selfishness scenarios in the PeerSim simulation framework. Simulations return a set of statistics on the behaviour of the system when in the presence of the specified type of selfish nodes.

The use of the *SEINE* framework supports a clear separation of selfishness concerns from the main logic of a cooperative distributed system. This separation improves overall maintainability, reuse, and reproducibility of both the system implementation and experiments. Particularly, the *SEINE-L* specification allows to describe and easily compare the same experimental conditions in different versions of the same cooperative system.

Overall, the present work makes the following contributions :

- We present the design of *SEINE-L* by conducting an analysis of 15 state-of-the-art articles related to the subject. We evaluate the expressiveness of the language by showing that *SEINE-L* can capture the semantics of the 38 selfish behaviours described in the papers analysed.
- We assess the impact of a selfishness scenario in a PeerSim simulation. Through the evaluation in three complete use cases, namely, a gossip-based dissemination protocol, a live streaming protocol (i.e., BAR Gossip [Li *et al.*,]) and a file sharing system (i.e., BitTorrent [Cohen, 2003, Locher *et al.*, 2006]), we show that the simulations enabled by *SEINE* are accurate with respect to real measurements performed on these systems.
- We show that *SEINE* facilitates a substantial reduction of the effort required to model, code, and evaluate selfish behaviours in a given system. First, we evaluate the effort quantitatively, showing that the number of lines of code required for assessing different selfishness scenarios in the three use cases using *SEINE* is almost an order of magnitude lower than the one required by their manual implementation. Then, we present a qualitative evaluation discussing the ease of use of *SEINE* in the rapid development ant test of different selfishness scenarios.

The remainder of the chapter is organised as follows. Section 8.2 introduces background information on selfishness in cooperative systems, and Section 8.3 presents an exhaustive analysis of the literature on the subject. Section 9.3 provides an overview of *SEINE*, followed by a detailed description of its components : the DSL for modelling a selfishness scenario (Section 8.5) and the support tools for injecting it into a PeerSim simulation (Section 8.6). Section 9.6 presents a performance evaluation of *SEINE*. Section 9.7 reviews the related work. Finally, the chapter concludes in Section 9.8.

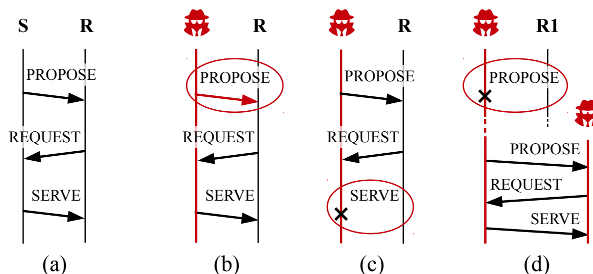


FIGURE 8.1 – Selfishness manifestations in gossip-based live streaming dissemination [Guerraoui et al., 2010a].

8.2 Background

A cooperative system is a complex distributed system that relies on voluntary resource contributions from its participants to perform the system function. File sharing systems (e.g., Gnutella [Hughes et al., 2005], eDonkey [Handurukande et al., 2006], BitTorrent [Cohen, 2003]) are the most widespread and well-known examples of cooperative systems. Other examples include cooperative distributed computing [Kwok et al., 2007, Anderson, 2004], self-organizing wireless networks [Yoo and Agrawal, 2006, Mei and Stefa, 2012, Miranda and Rodrigues, 2003], anonymous communication protocols [Ngan et al., 2010], and many others [Cox and Noble, 2003, Gramaglia et al., 2012, Ben Mokhtar et al., 2014]. Most cooperative systems are characterised by untrusted autonomous individuals with their own objectives — that are not necessarily aligned with the system's objectives — and full control over the device that they use to interact with the system [Mitchell and Teague, 2003]. In this context, a selfish node is an autonomous, strategic and self-interested individual that cooperates with other nodes only if such behaviour increases its local benefits. In practice, a selfish node may choose to stop behaving cooperatively if one or more of the following conditions occur : (i) the cost of contributing resources to other nodes outweighs the benefits received from the system ; (ii) the system does not impose punishments for selfish behaviours, or the punishment is not fast, certain and painful enough to be credible ; (iii) there are economic or social reasons for cooperating only with a restricted group of nodes ; (iv) the node suffers from persistent resource shortages, due for example to hardware or software limitations of the device that hosts the node (e.g., battery-powered devices).

Selfish behaviours have been observed in many cooperative systems and may have multiple manifestations [Hughes et al., 2005, Cunha *et al.*, 2013, Ngan et al., 2010, Yoo and Agrawal, 2006]. For instance, let us consider the gossip-based live streaming system described by Guerraoui et al. [Guerraoui et al., 2010a], consisting of a source node that disseminates video chunks to a set of nodes over a P2P network. Fig. 8.1(a) represents the simplified interaction protocol between nodes participating in this system : a node S periodically sends a `PROPOSE` message containing the video chunks it has received to a set of randomly chosen partners (R), and asks them to reply with a `REQUEST` message that indicates the chunks they are missing. Finally, S delivers the requested chunks with a `SERVE` message.

If S is selfish, it may decide to save its bandwidth consumption by reducing the number of chunks R would request. To this end, S can provide false information to R, proposing fewer chunks than what are currently available (see Fig. 8.1(b)). Another strategy for the selfish node S to reduce its bandwidth consumption is to not serve all the requested chunks, but only a subset of them. In the extreme, S serves no chunks (see Fig. 8.1(c)). Guerraoui et al. demonstrated experimentally that if 25% of nodes make deviations like those in (b-c), then the fraction of cooperative nodes that are not able to view a clear stream reaches up to 50%. Lastly, Guerraoui et al. considered the possibility of collusion among nodes. For example, in Fig. 8.1(d) S does not start the dissemination protocol with nodes outside its colluding group (R1 in the figure), so as to dedicate more bandwidth to exchanging data with its colluders. Experiments have shown that a colluder can decrease its contribution up to 15% without suffering any performance degradation [Guerraoui et al., 2010a].

The examples above describe various manifestations of selfishness in a particular system and the

		Selfish deviation type [a]				*Selfish deviation type* [a]
Reference	**Domain**	D	F	M	C	O
Ben Mokhtar et al. [Ben Mokhtar et al., 2014]	Data Distribution	✓	×	×	✓	×
Ben Mokhtar et al. [Mokhtar et al., 2010]	Data Distribution	✓	✓	×	×	×
Guerraoui et al. [Guerraoui et al., 2010a]	Data Distribution	×	✓	✓	✓	×
Hughes et al. [Hughes et al., 2005]	Data Distribution	✓	×	×	×	×
Li et al. [Li *et al.*,]	Data Distribution	×	✓	✓	✓	×
Lian et al. [Lian et al., 2007]	Data Distribution	×	×	×	✓	×
Locher et al. [Locher *et al.*, 2006]	Data Distribution	✓	×	✓	×	✓
Piatek et al. [Piatek et al., 2010]	Data Distribution	×	✓	×	✓	×
Sirivianos et al. [Sirivianos et al., 2007]	Data Distribution	✓	×	✓	×	×
Anderson et al. [Anderson, 2004]	Computing	×	✓	×	✓	×
Kwok et al. [Kwok et al., 2007]	Computing	×	✓	✓	×	×
Cox and Noble [Cox and Noble, 2003]	Backup & Storage	×	✓	×	×	×
Gramaglia et al. [Gramaglia et al., 2012]	Backup & Storage	✓	×	×	×	×
Mei and Stefa [Mei and Stefa, 2012]	Networking	✓	✓	×	×	×
Ngan et al. [Ngan et al., 2010]	Anonym. Comm.	✓	✓	×	×	×

[a] D : defection , F : free ride, M : misreport, C : collusion, O : other types.

TABLE 8.1 – The papers reviewed for the domain analysis, with information about their application domain and the selfishness investigated.

impact they have on its performance. To help a designer assess the impact of selfishness on any cooperative system, we start by presenting, in the following section, a unified model for selfish behaviours resulting from an extensive analysis of state-of-the-art works on this topic.

8.3 Domain Analysis

To gather domain knowledge on the problem of selfishness in cooperative systems, we performed a systematic analysis of the problem domain. Our goal is to identify possible commonalities in the motivations and executions of such behaviours, so as to build a domain-specific terminology and semantics for their representation and understanding.

Given the vast body of literature on the subject, we selected as inputs of the domain analysis 15 state-of-the-art papers that are of particular interest to the research community and that provide detailed descriptions of concrete selfish behaviours. Table 8.1 lists the selected papers and reports some of their relevant aspects, namely, the application domain of the target system (e.g., data distribution, distributed computing, networking) and other information to characterise the selfishness manifestation therein investigated. The output of our analysis is a model for the specification of *selfishness scenarios* in cooperative systems, whose formal representation is given by the feature diagram in Fig. 8.2.[23] A selfishness scenario consists of a non-empty set of *node models*, which describe interests and capabilities of types of nodes, and a set of *selfishness models*, which describe selfish behaviours. We present each of these components below.

Node model

The participants of a cooperative system constitute a heterogeneous population both in their personal interests and capabilities. A node model describes a type of participants in the system and specifies their interests and capabilities in terms of *resources*. A resource is a physical or logical commodity that increases the personal utility of nodes that possess it. A physical resource represent a node's capacity, such as *bandwidth*, *CPU* power, *storage* space, or *energy*. A logical resource can be a high-level and application-specific *service* offered by the cooperative system (e.g., file-sharing, message routing), or the *incentive* created by a cooperation enforcement mechanism (e.g., money, level of trust). The *capability* of

23. The feature diagram in the figure is a cardinality-based extension of the *FODA* notation [Czarnecki et al., 2004].

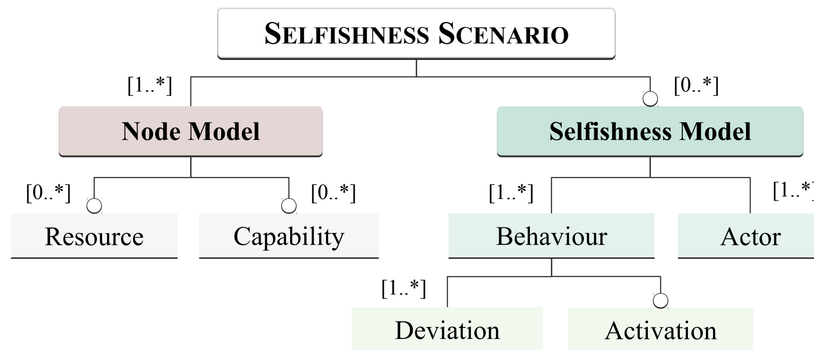


FIGURE 8.2 – Feature diagram of a selfishness scenario.

a node defines a constraint over a resource. For example, mobile nodes usually have lower communication and computation capabilities than desktop nodes, which can be expressed as a stricter constraint on the bandwidth and CPU resource, respectively.

Selfishness model

A node can be the *actor* of one or more selfish *behaviours*. In a selfishness model, a behaviour is described as the implementation of a non-empty set of *deviations* from the intended execution of protocols in the cooperative system. We define a *deviation point* as the step of a system protocol in which a deviation may take place. The wide range of motivations behind selfish behaviours, as well as the application-specific nature of their implementation, generates a tremendous number of possible deviations for any given cooperative behaviour. Nevertheless, based on our review of the available literature, we could identify four recurring types of deviations, named defection, free-riding, misreporting, and collusion. As shown in the last columns of Table 8.1, these types match almost all the selfish behaviours analysed in our review. The only exception is the rarest-first policy for requesting file pieces, which is very specific to the implementation of BitTorrent [Lian et al., 2007].

A *defection* is an intentional omission in the execution of a system protocol. A selfish node performs a defection to stop the protocol execution, so as to prevent requesters from consuming or even asking for its resources. *Free-riding* is a reduction in the amount of resources contributed by a node without stopping the protocol execution. The literature on cooperative systems offers other definitions of free riding, such as the complete lack of contribution [Lian et al., 2007, Ngan et al., 2010, Sirivianos et al., 2007], or downloading more data than what is uploaded [Hughes et al., 2005]. Our definition is more general because it applies to any resource, and it is more precise because it can be clearly distinguished from deviations that achieve a similar result by stopping the system protocols. A *misreport* consists in the communication of false or inaccurate information, to avoid contribution or gain better access to resources. Finally, a *collusion* is the coordinated execution of a selfish behaviour by a group of nodes that act together to increase their benefits. Collusions are more difficult to detect than individual deviations [Guerraoui et al., 2010a, Ben Mokhtar et al., 2014], because colluders can reciprocally hide their misbehaviours. Examples of misreporting and collusion have been discussed in the previous section and shown in Fig. 8.1(b) and (d).

In the selfishness model, the *activation* rule describes the conditions that motivate a node to start behaving selfishly. Examples of activation rules are exceeding a threshold amount of resource consumption or the delivery of a service (e.g., a file download). Another typical situation consists in providing false information to a monitoring mechanism to cover up previous deviations. Thus, a selfish behaviour may be the activator of other selfish behaviours.

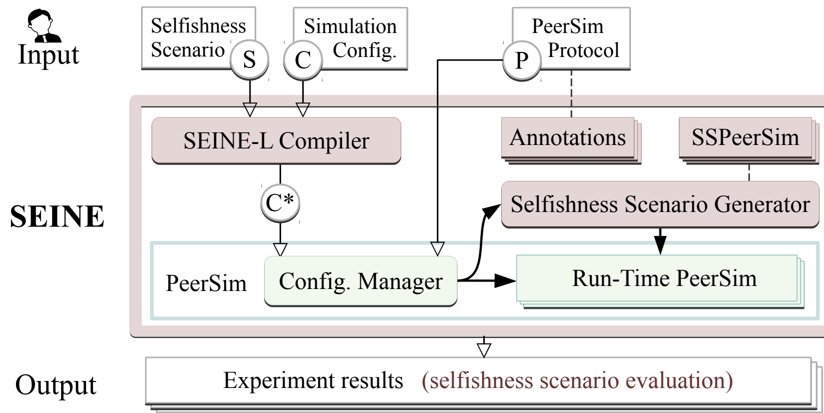


FIGURE 8.3 – Overview of the *SEINE* framework.

8.4 *SEINE* Overview

The *SEINE* framework aims to help cooperative system designers to evaluate the impact of selfish behaviours on the system performance. The framework builds on the lessons learnt from the domain analysis presented in Section 8.3, providing designers with modelling and simulation tools to describe and experiment with selfishness scenarios in a given system. *SEINE* relies on the PeerSim open-source simulator for large-scale distributed systems [Montresor and Jelasity, 2009]. The results of the simulation experiments are the output of *SEINE*.

Fig. 8.3 provides an overview of the *SEINE* framework. To begin, the system designer (hereafter "Designer", for brevity) produces the input files required by the framework, namely, a selfishness scenario (S in the figure) specified using the *SEINE-L* DSL, a configuration file to set up the simulation (C), and a Java implementation of the protocols underlying the system (P). The clear separation between selfishness and implementation concerns facilitates maintenance and reuse of the S and P artefacts. To associate the DSL declarations to the affected protocol implementation components (e.g., classes, variables, methods), the Designer decorates such components using a library of *Annotations* provided by the *SEINE* framework.

Upon creating all the input files, the Designer uses *SEINE* to study the behaviour of the system defined by C and P when faced with the selfishness scenario described in S. First, the *SEINE-L Compiler* generates the configuration file C^*, which extends C with instructions for injecting selfish behaviours into P as well as for monitoring the system performance. Second, *SEINE* calls the *Configuration Manager* included in the PeerSim library to build the experiment at run-time via reading C^* and instantiating the specified simulation components. These components are Java classes that implement (i) the nodes that compose the network, (ii) the set of protocols hosted by each node (including P), (iii) the observers that monitor or modify the behaviour of the simulated system, and (iv) the *Selfishness Scenario Generator* that injects the selfishness scenario into the simulation run-time. In particular, the Selfishness Scenario Generator uses Aspect-Oriented Programming techniques [Filman et al., 2004] and relies on a library of Java classes (*SSPeerSim*, in Fig. 8.3) to interact with the PeerSim simulator. Finally, the *SEINE* framework presents the results of the simulation as a collection of statistics describing the behaviour of the simulated cooperative system for the given selfishness scenario.

8.5 Modelling selfishness in *SEINE-L*

SEINE-L provides a clear and concise description of the capabilities, interests and behaviours of different classes of nodes participating in the protocols of a cooperative system. The semantics of the DSL builds on the domain analysis presented in Section 8.3, while its syntax is based on Java property files, i.e., collections of pairs associating a property name to a property value.

Fig. 8.4 illustrates the outline of a *SEINE-L* program. The entry point is the keyword `seine` followed by a dot and the name of the selfishness scenario. Then, the DSL provides five top-level language constructs : *resource*s of interest, *indicator*s of the system state, *node* models, *selfishness* models, and *observers* to monitor the system behaviour. The declarations of the first four constructs have the format `keyword.[construct_name]`, whereas the observers are defined inside a block of statements in curly braces.

```
seine.[selfishness_scenario_name] {
    resource.[r1_name]
    ...
    indicator.[i1_name]
    ...
    node.[n1_name] { ... }
    ...
    selfishness.[s1_name] { ... }
    ...
    observers { ... }
}
```

FIGURE 8.4 – The outline of a *SEINE-L* specification.

We illustrate the usage of the *SEINE-L* constructs by describing in detail the specification of a selfishness scenario for the live streaming system presented in Section 8.2 and originally described by Guerraoui et al. [Guerraoui et al., 2010a]. Listing 8.1 shows the *SEINE-L* specification of this scenario, called *LSS*, which we refer to in the remainder of this section. In *LSS*, a node can be either mobile or desktop, depending on its device type (lines 6-11). The scenario shows that mobile nodes have severely constrained resources, and, particularly, their upload bandwidth capacity is half of the desktops' capacity (line 9). The selfish behaviours declared in the *LSS* example are based on those illustrated in Fig. 8.1, which aim to reduce the bandwidth dedicated to other nodes (lines 14-24 in Listing 8.1) or to non-colluders (lines 25-30).

Comments

Comments begin with `#` and continue to the end of the line, as illustrated in lines 1, 3, and 12.

Resources

The keyword `resource` introduces the declaration of a physical or logical resource. The declared resources must be associated with a value, which can function as an indicator of its current state. Line 3 of the *LSS* scenario declares the *bwCapacity* resource, which refers to the upload bandwidth capacity of nodes. A resource declaration can also provide instructions to initialize its values. In our example, all nodes are initialised with the same upload capacity (mode `uniform`) of 1000 kbps. *SEINE-L* allows other two initialisation modes : `random` and `linear` (i.e., a linearly increasing distribution of values within a specified range).

Indicators

An `indicator` declaration specifies a quantifiable attribute of either the system or a node type that depends upon its current state. For instance, the *batteryLeft* indicator in line 4 of Listing 8.1 can be used to guard the battery level of mobile nodes. Indicators cannot be initialised within a *SEINE-L* program.

Node model

Each node model is declared using the `node` keyword followed by a name and a block of properties delimited by curly braces. The *LSS* scenario declares mobile and desktop nodes, respectively, in lines 6-10 and line 11. The properties that can characterise a node model are listed below :

```
1  # Three-phase gossip-based live streaming
2  seine.LSS {
3    resource.bwCapacity uniform(1000) # kbps
4    indicator.batteryLeft

6    node.mobile {
7      fraction 0.3
8      selfish 0.5
9      capability bwCapacity(500)
10   }
11   node.desktop { selfish 0.1 }
12   node.exclude 0 # the source of the streaming

14   selfishness.smMobile {
15     actor mobile(0.8)
16     behaviour.bhvAggressive {
17       activation batteryLeft < 30
18       freeriding { degree 0.8 on send_SERVE }
19     }
20     behaviour.bhvWeak {
21       freeriding { degree 0.3 on send_SERVE }
22       misreport { degree 0.3 on send_PROPOSE }
23     }
24   }
25   selfishness.smColluders {
26     actor desktop mobile(0.2)
27     behaviour.bhvCollusive {
28       collusion.probability 0.15
29     }
30   }
31   observers {
32     period 100
33     name package.path.LSSObserver
34   }
35 }
```

Listing 8.1 – A *SEINE-L* specification of a selfishness scenario for the live streaming system described in [Guerraoui et al., 2010a].

- `fraction` is the proportion of nodes in the system that hold this node model. If omitted, the fraction is set evenly by the preprocessor so that all node fractions sum up to 1. For instance, *desktop* nodes in the considered scenario will have the fraction set to 0.7.
- `selfish` is the fraction of selfish nodes within this node model. The default value is 1, i.e., all nodes holding this model are selfish.
- `capability` is a list of constraints over the values of the declared resources. Line 10 of the *LSS* scenario, for example, halves the upload bandwidth capacity of mobile peers, to 500 kbps. The DSL syntax prevents the definition of capabilities on resources that are not specified in the program.

There might be reasons to exclude a given set of nodes from the scope of the selfishness scenario, for instance because they represent special devices or trusted parties. In *SEINE-L*, this can be achieved using the `node.exclude` keywords followed by the identifiers (i.e., integers) of the nodes to exclude. As an example, line 12 of Listing 8.1 excludes the first node from the *LSS* scenario, because it represents the streaming source, which is assumed to be always cooperative.

Selfishness model

The `selfishness` declaration specifies the selfish behaviours adopted by a certain configuration of nodes. Such a configuration is expressed by the `actor` keyword followed by a list of terms, each defining the fraction of nodes of a given model to associate with the selfishness under specification. In Listing 8.1, the selfish behaviours of the *LSS* scenario described above are grouped into two selfishness declarations, namely, *smMobile* and *smColluders*. The actors of the *smMobile* model are defined in line 15 as 80% of the selfish population of the *mobile* nodes. In practice, given that 15% of nodes in the live streaming system were described in lines 7-8 as selfish mobile nodes (i.e., 50% of 30% of the overall population), the percentage of nodes that adopt the *smMobile* selfishness model is 12%. Notice that in the `actor` declaration in line 26 in Listing 8.1, the fraction of *desktop* nodes is not specified.

```
actor desktop mobile(0.2)
```

In this case, the default value is 1, which means that all selfish *desktop* nodes are actors of the *smColluders* model.

Each selfish behaviour that constitutes a selfishness model is described by a `behaviour` declaration. A behaviour is a list of selfish deviations from the correct execution of a system protocol ; such deviations are strategically interrelated and triggered by the same activation rule, which is introduced in *SEINE-L* by the `activation` keyword. An activation rule defines a condition (e.g., greater-than-or-equal-to) over the current value of a resource or indicator declared in the selfishness scenario. The *LSS* specification, for example, indicates in line 17 that every mobile node with a *smMobile* selfishness model switches to a more aggressive behaviour to reduce bandwidth consumption when it is running out of battery (i.e., the battery level drops below 30%). In contrast, if no activation rule is specified, then the selfish behaviour is always triggered. This is the case of the *bhvWeak* (lines 20-23) and *bhvCollusive* (lines 27-29) behaviours. Support for the specification of logical expressions to combine multiple activation rules is left to future work.

A selfish behaviour specifies a non-empty set of deviations from the correct execution of certain steps (deviation points) of the system protocols. The *SEINE-L* syntax allows to declare five types of deviations, based on the classification developed from the domain analysis. Each deviation type is introduced by its own keyword, namely, `defection`, `freeriding` (`free-riding` is also accepted), `misreport`, `collusion`, and `other`, if none of the previous types applies. The execution of a deviation can be further characterised by the following additional properties of the deviation declaration :

- `probability` indicates the probability to deviate if the activation rule of the corresponding behaviour is met. The default value is 1.
- `on` constrains the possible deviation points of a deviation. For instance, in the free-riding declaration of the *bhvAggressive* behaviour (line 18), the `on` property ties the execution of this deviation to the deviation point named *send_SERVE* (see Fig. 8.1(c)). Multiple deviation points can be listed as illustrated below, separated by whitespace.

```
on send_PROPOSE send_REQUEST send_SERVE
```

SEINE-L also allows specifying the steps of a system protocol execution in which the deviation *cannot* take place, by preceding the name of a deviation point with an exclamation mark. For instance, the code fragment below specifies a free-riding deviation that affects all deviation points except the one named *send_SERVE*.

```
freeriding { on !send_SERVE }
```

- `degree` is a real value between 0 and 1 that specifies the intensity of free riding and misreport deviations (default value 1). In particular, the degree quantifies the reduction in the amount of resources contributed by a node in the case of a free-riding deviation, and the reduction in the reliability of the information provided in the case of a misreport deviation.

Different deviations of the same type may affect the same deviation points. For instance, in the *LSS* scenario, the *smMobile* model includes two behaviours that specify a free-riding deviation on the *send_SERVE* deviation point. Moreover, if the value of the *batteryLeft* indicator is below 30%, then both behaviours are activated. These conflicts are resolved by triggering the first deviation in order of appearance in the program. The development of more sophisticated conflict resolution strategies is another area of future study.

To conclude, *SEINE-L* also allows a compact declaration for deviations with only one property, that is, the declaration used in line 28 of Listing 8.1.

Observers

The language constructs presented so far focus on the description of a selfishness scenario. In addition, *SEINE-L* provides a means to set-up monitoring components for assessing the performance of a cooperative system under that scenario. This can be done using the `observers` declaration. In practice, an observer is a Java object that collects statistics on the system performance during its simulation with PeerSim (see Section 8.6 for more details). The `observers` declaration specifies the full class `name`s of each observer object to enable, as well as the `period` between two monitoring events in terms of simulated seconds (the default value is 100). For instance, the *LSS* scenario sets up the periodic execution of the *LSSObserver* object every 100 simulated seconds. This can also be specified using the compact form below.

```
observers.name package.path.LSSObserver
```

8.6 Injecting selfishness in PeerSim using *SEINE*

The *SEINE* framework comprises the PeerSim simulator [Montresor and Jelasity, 2009], a library of annotations for linking the *SEINE-L* specification of a selfishness scenario to the source code of PeerSim protocols, a compiler for *SEINE-L*, and a generator of simulation components for modelling, executing and monitoring a selfishness scenario in PeerSim. In the remainder of this section, we present each of the novel tools developed for *SEINE*.

8.6.1 Library of Annotations

Annotations are the means to link a selfishness scenario for a given system to the concrete implementation of that system, i.e., a set of PeerSim protocols. More precisely, the Designer can associate declarations of a *SEINE-L* specification to the affected program elements (e.g., classes, fields, methods) by decorating an element definition with annotations. This operation requires small and simple modifications of the original code. The *SEINE* framework provides eight types of annotation. `@Seine` decorates the declaration of each class implementing a PeerSim protocol to associate with the *SEINE-L* specification. In other words, it specifies which protocols are affected by the selfishness scenario. `@Resource` and `@Indicator` declare a field modelling a resource or indicator in *SEINE-L*, respectively.

The remaining annotation types allow indicating deviation points in PeerSim protocols. Concretely, a deviation point is the Java method that implements the part of the protocol behaviour in which one or more deviations may take place. These annotations are named after their deviation type (i.e.,

```
1  @Seine
2  public class LSS {
3    ...
4    @Misreport @Collusion(ref_arg = 1)
5    public void send_PROPOSE
6        (List cnksId, LiveStreaming req) { ... }
7    ...
8    @Freeriding
9    public void send_SERVE
10       (List cnks, LiveStreaming req) { ... }
11 }
```

Listing 8.2 – A fragment of the PeerSim protocol implementing the system to associate with the *LSS* scenario in Listing 8.1.

`@Defection`, `@Freeriding`, `@Misreport`, `@Collusion`, and `@OtherDeviation`) and target method declarations. As an example, let Listing 8.2 be a fragment of the PeerSim implementation of the live streaming system to experiment with the *LSS* selfishness scenario presented in Section 8.5. The annotations in lines 4 and 8 indicate the default deviation points for any declarations of the corresponding deviation type provided in *LSS*. For instance, the `collusion` deviation in line 28 in Listing 8.1 is implicitly associated with any method that has been annotated with `@Collusion`, such as `send_PROPOSE` in Listing 8.2.

The annotations to indicate deviation points can have different attributes, depending on the deviation type that it represents. For instance, the `@Collusion` annotation in Listing 8.2 (line 4) specifies the attribute `ref_arg`, which indicates what argument of the *send_PROPOSE* method is the reference to the protocol run by a potential colluder. This is the first argument by default. The same attribute is also supported by the `@Freeriding` and `@Misreport` annotation types, identifying the argument of the deviation point that may be affected by the deviation. For instance, the free riding annotation in line 8 of Listing 8.2 can modify the value of the list of chunks `cnks` to deliver to the requester `req`. For reasons of space, an exhaustive overview of all annotation attributes is beyond the scope of this paper.

8.6.2 *SEINE-L* Compiler

As shown in Fig. 8.3, the *SEINE-L* compiler performs a source-to-source transformation of the *SEINE-L* specification (*S*, in the figure) into the PeerSim configuration file format.

The *SEINE-L* compiler performs statically various consistency checks on the selfishness scenario specification. Due to the declarative nature of the DSL, it is possible to verify the consistency of a specification with respect to the following properties : no omission (i.e., each referenced construct must be declared), no double declaration, correctness of the node model distribution (i.e., the proportions of the declared node models must sum to 1), and of the selfishness model distribution (i.e., for each node model, the proportions of selfish nodes adopting a selfishness model must sum to 1). If any of these properties is not fulfilled, then the compiler reports the error and stops.

In addition to the detection of errors in the *SEINE-L* specification, the *SEINE-L* compiler can also verify whether there are inconsistencies in the association between the DSL program and the annotated PeerSim protocol. More precisely, it verifies that (i) the protocol class is decorated with the `@Seine` annotation, (ii) for each resource and indicator in the specification there exists a class field with the same name that has been properly annotated, and (iii) for each deviation point explicitly defined in a deviation declaration (using the `on` property) there exists a method declaration with the same name that has been consistently annotated.

8.6.3 Selfishness scenario generation

The configuration file generated by the *SEINE-L* compiler enables the *Configuration Manager* of the PeerSim simulator to initialise the native simulation objects (e.g., nodes, protocols, monitors) as well as the selfishness scenario objects (e.g., resources, node models, deviations). Specifically, each language construct of the *SEINE-L* syntax is implemented in a Java class in the *SSPeerSim* Java library, which is included in the *SEINE* framework.

At run time, the Configuration Manager gives instructions to the *Selfishness Scenario Generator* to properly instantiate the classes in *SSPeerSim* so as to generate the objects that support the simulation of the selfishness scenario. Also, the Selfishness Scenario Generator uses Aspect-Oriented Programming (AOP) [Filman et al., 2004] techniques to modify the execution of the PeerSim protocol components that have been annotated by the Designer, in such a way as to inject selfish behaviours and mode model capabilities. For coherence with the *SEINE* and PeerSim frameworks, both written in Java, we chose AspectJ [Kiczales et al., 2001] as the aspect-oriented language. In AspectJ, cross-cutting behaviours are described in class-like modules, called *aspects*. An aspect includes *advice* constructs for describing code to be inserted at given locations (*joinpoints*) of a standard Java program. Such locations are specified by *pointcut* constructs. An advice can insert the code *before* or *after* such locations, or it can replace existing code (*around* advice). The Selfishness Scenario Generator includes the aspects listed below.

- `SeineProtocolAspect` can extend PeerSim protocol classes by adding fields for storing selfishness-related information (e.g., the name of the node model implemented, the selfish behaviours that can be performed) as well as methods for behaving accordingly to the selfishness scenario provided. The pointcut of this aspect intercepts all the classes decorated with the `@Seine` annotation.
- `ResourceIndicatorAspect` replaces getters/setters of the fields decorated with `@Resource` and `@Indicator` annotation types with a new implementation that (i) checks the fulfilment of each activation condition that may trigger a selfish behaviour, and, only for resources, (ii) constrains the values to the range specified by a capability condition.
- `SelfishInjectionAspect` specifies the advice that replaces the correct implementation of an annotated deviation point with that of a selfish deviation. More precisely, first it checks whether the deviation can take place, by verifying that the node executing the method can perform a deviation of that type and that the deviation is currently activated. If these conditions are verified, then the deviation implemented in the advice can be executed ; otherwise, the execution proceeds according to the reference implementation.

The code snippet in Listing 8.3 illustrates the integration of deviation code by the `SelfishInjectionAspect` into the *send_SERVE* method. According to the *LSS* selfishness scenario presented in Listing 8.1, this method is a deviation point only for selfish mobile nodes that adopt the *smMobile* selfishness model, i.e., 12% of the overall system population (see Section 8.5).

```
1  @Freeriding
2  public void send_SERVE( ... ) {
3    /** SelfishInjectionAspect advice */
4    boolean can_deviate = /* Verification */ ;
5    if(can_deviate) {
6      /* Execution of the deviation code */
7    }
8    /** End of SelfishInjectionAspect advice */

10   /* reference method implementation */ ...
11 }
```

Listing 8.3 – A code fragment representing code injection into the *send_SERVE* method.

Another type of simulation component instantiated by the Selfishness Scenario Generator is the set of observers that monitor and gather statistics on the system performance. The *SEINE* framework aids the Designer in creating application domain-specific observers, by providing in the *SSPeerSim* library an abstract class that defines the methods that need to be implemented. The execution of the Designer's observers is coordinated by a configurable controller that is automatically operated by the Selfishness Scenario Generator. Particularly, the set-up of the controller is specified by the `observers` declaration of the *SEINE-L* program (see Section 8.5).

8.6.4 *SEINE* Implementation

All tools and components in *SEINE* are written in Java. The entire implementation consists of almost 4000 lines of code, not including third-party components (i.e., the PeerSim simulator) and automatically generated code (i.e., the *SEINE-L* parser, built using ANTLR [Parr, 2013]). We developed the *SEINE* framework in a modular and loosely coupled manner, which promotes extensibility and reuse of its core

components. For example, the interaction with the PeerSim simulator is implemented in a separate and independent module (the *SSPeerSim* library), which is less than 25% of the entire source code.

8.7 Evaluation

In this section, we demonstrate the benefits of using *SEINE* to describe selfish behaviours and evaluate their impact on cooperative systems. We start by assessing the generality and expressiveness of the *SEINE-L* language by outlining some of our experiences in describing selfishness scenarios with our DSL. Then, we evaluate the accuracy of the *SEINE* output, developing and testing three use cases selected from our literature review, namely, a gossip-based live streaming protocol, a selfish-resilient media streaming protocol, and a selfish client for the BitTorrent protocol. Also, we assess the effort required by a Designer to implement and test the use cases. Finally, we show that *SEINE* imposes a small time overhead on the normal execution of the PeerSim simulator.

The *SEINE* framework will be made available, fully and freely, upon acceptance of the paper. To facilitate the reproducibility of our results, the configuration files related to the experiments reported in this section will also be available for download on the project website.

8.7.1 Generality and expressiveness of *SEINE-L*

We have used *SEINE-L* to express all of the selfishness scenarios described in the studies reviewed for the domain analysis (see Section 8.3). Many of these works present various strategies to save bandwidth in data distribution applications, such as Gnutella [Hughes et al., 2005], BitTorrent [Lian et al., 2007], and PPLive [Piatek et al., 2010]. Other works investigate selfishness in different domains, like the paper of Kwok et al. [Kwok et al., 2007] that studies Grid computing systems, and specifically the impact of task dispatching policies within a Grid site that allocate resources only to local tasks. Overall, the number and variety of the cooperative systems considered, as well as the different degrees of complexity of the selfishness scenarios therein described, demonstrate the general applicability and the expressive power of *SEINE-L*.

Table 8.2 shows that *SEINE-L* files are concise : the selfishness scenarios specified are between 14 and 38 Lines of Code (LoC), with an average of 25 LoC.

Reference	LoC	Reference	LoC
Ben Mokhtar et al. [Ben Mokhtar et al., 2014]	29	Sirivianos et al. [Sirivianos et al., 2007]	24
Ben Mokhtar et al. [Mokhtar et al., 2010]	34	Anderson et al. [Anderson, 2004]	17
Kwok et al. [Kwok et al., 2007]	24	Cox and Noble [Cox and Noble, 2003]	14
Guerraoui et al. [Guerraoui et al., 2010a]	25	Gramaglia et al. [Gramaglia et al., 2012]	35
Hughes et al. [Hughes et al., 2005]	19	Mei and Stefa [Mei and Stefa, 2012]	28
Li et al. [Li *et al.*,]	38	Ngan et al. [Ngan et al., 2010]	30
Lian et al. [Lian et al., 2007]	17	Piatek et al. [Piatek et al., 2010]	18
Locher et al. [Locher *et al.*, 2006]	23		

Table 8.2 – Lines of Code for expressing the selfishness scenarios of the papers considered in the domain analysis review.

8.7.2 Accuracy of *SEINE-R*

To validate the accuracy of *SEINE*, we compared the results produced by our framework with those published in three use cases selected from the literature review. We discuss each use case separately below.

Live Streaming

We consider the gossip-based streaming system presented by Guerraoui et al. [Guerraoui et al., 2010a] and already described in Section 8.5. Despite its simplicity, this system is realistic enough to serve as

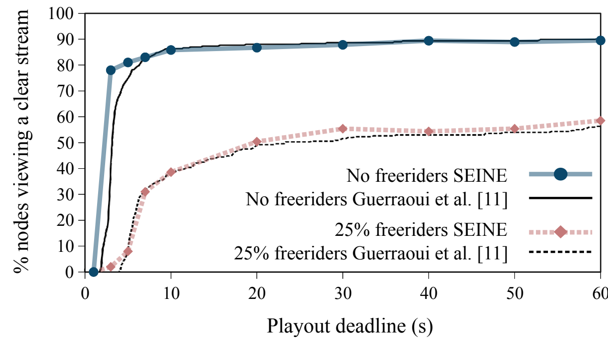


FIGURE 8.5 – Comparison between the results published by Guerraoui et al. [Guerraoui et al., 2010a] and the results obtained with *SEINE*.

a representative example of a practical live streaming application. Guerraoui et al. deployed the system over PlanetLab, [24] in which a source node streams video chunks with a bit rate of 674kbps to 300 nodes having upload bandwidth limited to 1000kbps. They tested one scenario with only cooperative nodes and another scenario with a quarter of the nodes being selfish, following the selfishness model described in Section 8.5. To assess the system performance in both scenarios, Guerraoui et al. considered the fraction of cooperative nodes perceiving a clear stream (i.e., viewing at least 99% of the streamed chunks) when varying the playout deadline from 0 to 60 seconds.

We developed the gossip-based live streaming system as a PeerSim protocol and we used *SEINE* to describe and simulate the same selfishness scenario as well as the same experiment setting as investigated by Guerraoui et al. [Guerraoui et al., 2010a]. We ran ten simulations for each set of parameters, obtaining a fairly low standard deviation (0.02 on average), and we used the mean value to compare with the reference results. Fig. 8.5 shows the high level of accuracy of *SEINE-R*, with an almost perfect match of the curves representing the scenario with selfish nodes and a satisfactory correspondence also in the selfish-free scenario. [25] Note that in the latter scenario our results show high accuracy when the playout deadline is above 7 seconds, whereas, below this time limit, *SEINE* simulations indicated better results. This is mainly due to the lower, real-world reliability of the PlanetLab network compared with the perfect but simulated reliability of the PeerSim network.

BAR Gossip

Proposed by Li et al. [Li *et al.*,], BAR Gossip is a P2P live streaming system designed to tolerate both Byzantine and selfish peers. To this end, BAR Gossip includes mechanisms to enforce cooperation, namely, verifiable partner selection and data exchange mechanisms that make non-cooperative behaviours detectable and punishable. We select this use case to show that *SEINE* can also be used as a tool for testing performance and robustness of selfish-resilient protocols. For instance, BAR Gossip has been proven vulnerable to colluding nodes [Ben Mokhtar et al., 2014], which exchange video chunks off-the-track among each other, thereby decreasing the system efficiency and particularly the streaming experience of non-colluding nodes.

We assessed the accuracy of *SEINE* in reproducing the BAR Gossip selfishness scenario that was presented and experimentally studied by Ben Mokhtar et al. [Ben Mokhtar et al., 2014]. That study deployed 400 nodes in the Grid'5000 testbed, [26] each node running either a compliant version of BAR Gossip or a collusion-enabled implementation. We developed BAR Gossip [Li *et al.*,] in PeerSim and set up its protocols using the configuration reported by Ben Mokhtar et al. [Ben Mokhtar et al., 2014]. Then, we simulated the system when varying the proportion of colluding nodes, from 0 to 50, and we measured the fraction of missed updates by cooperative nodes. Again, we ran ten simulations for each setting, obtaining an average standard deviation below 0.01. As clearly depicted in Fig. 8.6, the accuracy of the

24. PlanetLab : `https://www.planet-lab.org/`
25. Our results are plotted over a copy of the figure published in [Guerraoui et al., 2010a].
26. Grid'5000 : `https://www.grid5000.fr/`

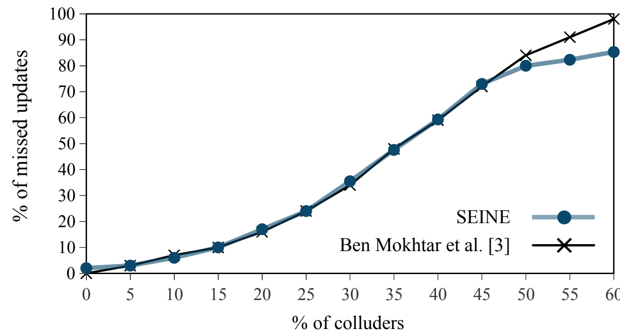


FIGURE 8.6 – Comparison between the results published by Ben Mokhtar et al. [Ben Mokhtar et al., 2014] and the results obtained with *SEINE*.

results output by *SEINE-R* with respect to the ones provided by the authors of the reported study is very high (0.996 Pearson correlation score). The gap between the results when the proportion of colluders is above 50% is due to some missing parameters in the description of the experiment setting, especially the maximum of chunks that can be exchanged during the optimistic push protocol.

BitThief

Locher et al. [Locher *et al.*, 2006] developed and released software to download files in the BitTorrent protocol without uploading any data. Specifically, they implemented and openly distributed a selfish client, *BitThief*, capable of attaining fast downloads without contributing. The purpose of this use case is twofold. First, it shows the accuracy of *SEINE* on empirical experiments performed on real-world applications. Second, it proves the simplicity of using *SEINE* when a PeerSim implementation of the system is already available. Specifically, we used the BitTorrent code published on the PeerSim project website. [27]

We used *SEINE* to reproduce the real world experiment described by Locher et al. [Locher *et al.*, 2006], which consists in monitoring the download times for one torrent in a BitTorrent network with 5% BitThief clients. BitThief exploits several features of the BitTorrent protocol by means of a set of selfish deviations from the default client implementation. For example, a BitThief client can open up to 500 connections with other peers (the default value is 80), to increase its probability of receiving useful file pieces. In their experiment, Locher et al. showed that such deviations allow BitThief clients to download the file with performance comparable to (if not better than) the default clients, while not contributing any file pieces to other peers. The same result have been obtained in our experiment using *SEINE*, as can been observed in Fig.s 8.7(a)-(b).

8.7.3 Development effort

To show the benefits of using *SEINE* in terms of design and development complexity, we discuss the effort required to describe, implement and maintain selfish behaviours in our use cases. Using *SEINE*, the specification of the selfishness scenario to test is clearly separated from its actual integration into the cooperative system code. Such separation of concerns facilitates description and maintenance of node models and selfish behaviours, allowing the Designer to focus only on the *SEINE-L* program and on a few annotations of the system code. On the contrary, without using *SEINE*, the selfishness scenario must be hard coded into the PeerSim Java programs and configuration files.

As can be noted from Fig. 8.8(a), implementing a selfishness scenario using *SEINE* is not only easier but also extremely concise regarding Lines of Code. The bars in the figure provide a graphical representation of the volume and distribution of code to modify with respect to the faithful implementation of the cooperative system (the white part of the bar). The figure also reports the exact number of LoC to add

27. `http://peersim.sourceforge.net/code/bittorrent.tar.gz`.

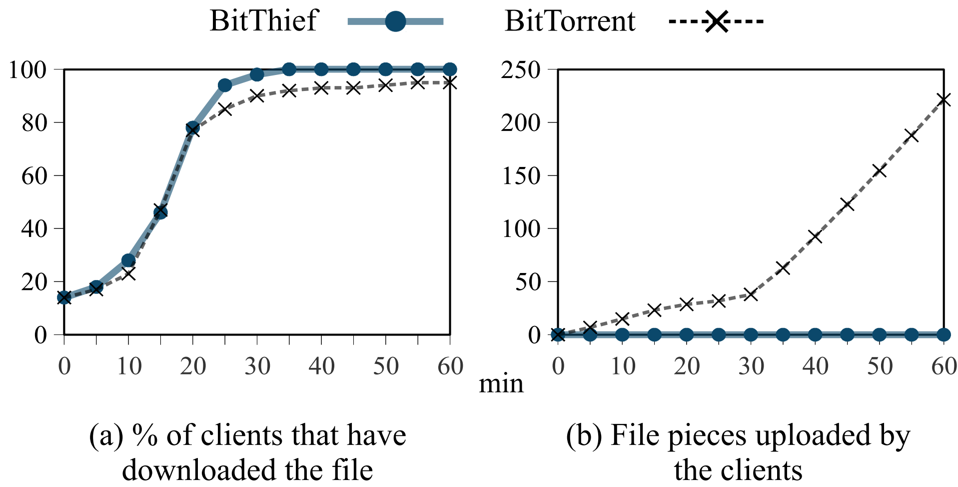


FIGURE 8.7 – Performance and contribution of BitTorrent and BitThief when downloading the same file, measured using *SEINE*.

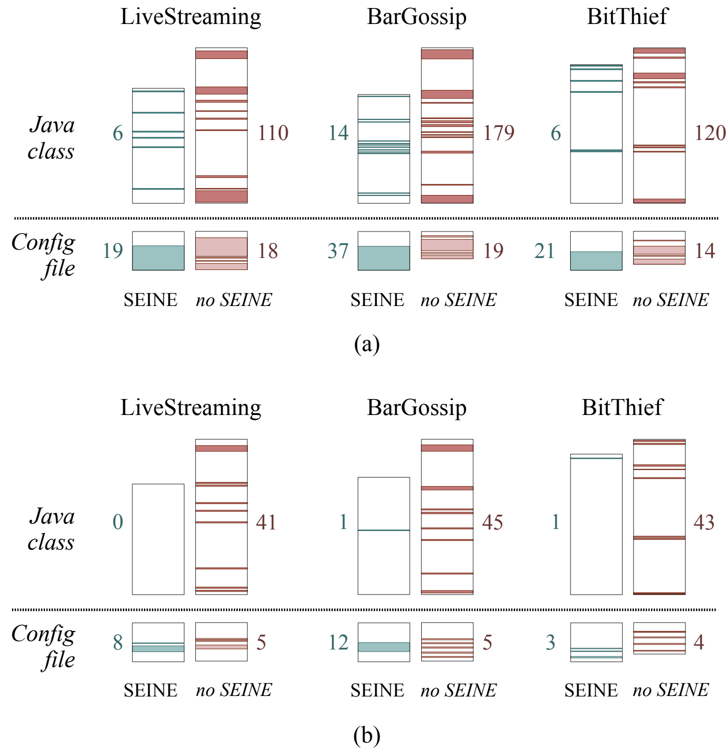


FIGURE 8.8 – Number and distribution of Lines of Code (a) to specify the selfishness scenario into the faithful implementation of the use cases and (b) to modify such scenarios, with and without using *SEINE*.

for each use case. We can derive two observations from these results. First, implementing a selfishness scenario using *SEINE* requires four to five times less code to write than without using our framework. Second, when not using *SEINE*, such implementation (i.e., parameters, variables and methods) is scattered across the code of the PeerSim Java program and configuration file ; on the other hand, the annotation library included in *SEINE-R* requires the Designer only to annotate existing fields and methods of the PeerSim program, and to write a *SEINE-L* program directly into a configuration file.

To illustrate the gain in flexibility and maintainability of testing selfishness scenarios using *SEINE*, we propose simple modifications to the scenarios of our use cases and we discuss the effort required to adapt the input files.

- *Live Streaming* : we duplicate a selfishness model specifying a different activation policy and deviation parameters (i.e., a higher degree of free riding and misreporting).
- *BAR Gossip* : we remove a selfishness model.
- *BitThief* : we remove a resource and we add a probability of execution to all deviations.

Fig. 8.8(b) illustrates the number of Lines of Code to modify (i.e., add, remove, or edit) in each use case to implement the modifications listed above. Using *SEINE*, modifying the Java class requires modification of at most 1 line, which corresponds to inserting or dropping an annotation. Furthermore, when updating the configuration file, the Designer operates on coherent and consecutive blocks, such as the selfishness model block. In contrast, as can be observed in Fig. 8.8(b), implementing the modifications to the selfishness scenarios when not using *SEINE* leads to more LoC to modify, which are scattered across the sources.

To demonstrate how *SEINE* facilitates fast development and testing of different selfishness scenarios, we present two test cases for the BAR Gossip cooperative system. In the first test case, we start from the selfishness scenario described in [Ben Mokhtar et al., 2014] and discussed in Section 8.7.2, and we evaluate the impact of the size and number of colluding groups. More precisely, we fix the fraction of selfish nodes in the system to 20%, and we evaluate the quality of stream perceived by selfish and cooperative nodes when varying from one big colluding group to 10 smaller groups of equal size. Results depicted in Fig. 8.9(a) show that the percentage of updates missed by selfish nodes increases as they form colluding groups of smaller size. This is due to the lower probability for colluders to meet, given the random nature of the underlying gossip protocol for chunk dissemination, which cannot be cheated in BAR Gossip by design. On the contrary, the absence of significant changes in the performance for cooperative nodes indicates that they are not affected by how colluders organise themselves into groups. The system Designer can implement this test case using *SEINE* without modifying a single line of code in the PeerSim implementation of BAR Gossip, but only operating on the *SEINE-L* code. Specifically, the Designer first duplicates the selfishness model describing the collusive behaviour as many times as the number of colluding groups she wants to create. Then, the Designer modifies the fractions of the `actor` declarations, in such a way that they sum to one.

As a second test case, we investigate the impact of mobile nodes with lower bandwidth capabilities on the performance of BAR Gossip. Similarly to the previous test case, this scenario modification does not change the system implementation but only the *SEINE-L* description of the selfishness scenario. In particular, the Designer has to create a new node model block (i.e., *mobile*) which limits the bandwidth capacity with respect to the original node model (i.e., *desktop*). For example, the bandwidth capacity of desktop nodes is 1000 kbps, whereas it is capped to 300 kbps for mobile nodes. This modification to the scenario corresponds to adding 4 lines to the *SEINE-L* program and changing a few numeric values (e.g., refactoring the fraction of desktop nodes). Fig. 8.9(b) reports the results of this test case, showing that the percentage of updates that are missed by cooperative nodes decreases as the fraction of mobile nodes increases. This result can be explained by the lower contribution that mobile nodes make to the chunk dissemination protocols, due to their limited resource capabilities.

To conclude, test cases demonstrate how the development and testing of cooperative systems greatly benefited from the *SEINE* functionalities. Evaluating a new selfishness scenario in a complex system like BAR Gossip only took less than an hour, including the simulation time.

8.7.4 Simulation time

In this section, we evaluate the extra execution time imposed by *SEINE* on the regular PeerSim performance. To this end, we defined 10 different selfishness scenarios for each use case described in Section 8.7.2, and we ran 40 simulations for each scenario. The results, summarised in Table 8.3, show that *SEINE* imposes an average extra execution time of 5% (standard deviation 0.03), ranging from the 1.6% extra time achieved by the BAR Gossip use case to the 7.8% extra time of BitThief. Such a short duration increase — 11 seconds out of 154 seconds, at most — appears to be reasonable in light of the benefits provided by the *SEINE* framework.

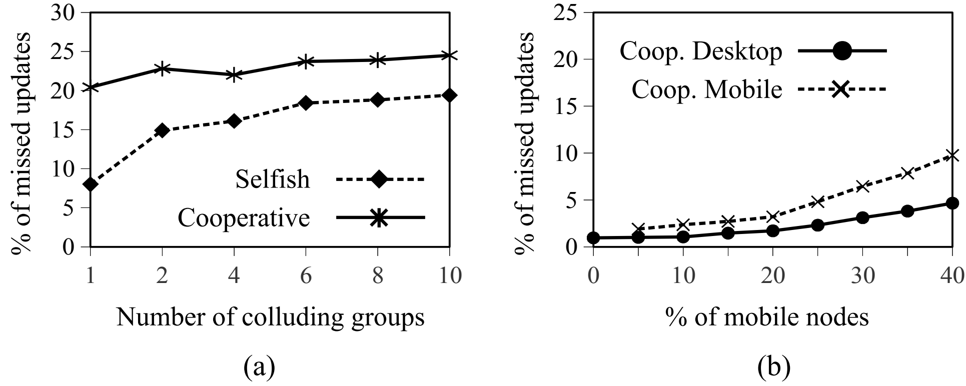


FIGURE 8.9 – Performance of BAR Gossip when varying (a) the number of colluding groups and (b) the fraction of resourceless mobile nodes.

Use Case	**Execution time** (ms)	**Extra time** (ms)
Live Streaming	86,900	4,791
BAR Gossip	15,088	238
BitThief	154,604	11,176

TABLE 8.3 – Average execution time to evaluate a selfishness scenario using *SEINE* and the additional time it imposes.

8.8 Related Work

The ways selfishness has been evaluated in the literature of cooperative systems can be broadly divided into analytical and experimental approaches. Analytic analysis, especially game theory [Myerson, 2013], provides mathematical tools to reason about selfishness and cooperation in competitive situations like those underlying a cooperative system [Li *et al.*, ,Ben Mokhtar et al., 2014,Mokhtar et al., 2010,Gramaglia et al., 2012]. However, applying formal approaches to study real systems tends to be complex [Mahajan et al., 2004, Rahman et al., 2011]. Particularly, game theory approaches require manually creating a mathematical model of the system (the game), including the alternative strategies available to the system participants (the players) and their preferences over the possible outcomes of the system. Then, game-theoretic arguments have to be formulated to assess what strategy is the most likely to be played by the players. In addition to being complex and time-consuming, carrying out this process is also prone to modelling errors, due to assumptions and simplifications to make the model tractable [Rahman et al., 2011]. Finally, and most relevant to our work, game theory can be helpful in understanding the decision-making process of system participants, but not to estimate the impact of their decisions on the system.

Adopting an experimental approach, like in *SEINE*, can be an appropriate solution to overcome the shortcomings of analytical modelling. Concretely, an experiment consists in implementing a selfish behaviour in a real or simulated instance of the system [Gustedt et al., 2009]. Testbeds such as Grid'5000 and PlanetLab provide the physical infrastructure to perform experiments with real distributed applications on real networks, in a configurable and monitorable manner. On the other hand, like many other authors [Ngan et al., 2010,Mei and Stefa, 2012,Mokhtar et al., 2010,Li *et al.*, ,Gramaglia et al., 2012,Kwok et al., 2007], we consider simulations a more practical tool for conducting comprehensive evaluation campaigns on large-scale systems such as cooperative systems. The use of simulations allows for a perfect control over the experimental conditions, high reproducibility, faster execution time, and the possibility of simulating millions of nodes on a single host [Basu et al., 2013]. These features come at the price of a high level of abstraction, which can introduce some bias in the experimental results [Gustedt et al., 2009]. Relying on a well-established and extensively tested simulator — e.g., PeerSim [Montresor and Jelasity, 2009], used by *SEINE* — provides more certainty on the accuracy of the results.

Although a number of the existing frameworks suitable for the experimental evaluation of cooperative

systems have the ability to script and inject events into the system, we find that almost none of them explicitly support the generation and assessment of selfish behaviours. Indeed, in most cases, the support for scripted events allows to specify system dynamics (e.g., churn management [Montresor and Jelasity, 2009, Leonini et al., 2009]) or to inject simple fault events into specific system components [Gustedt et al., 2009, Basu et al., 2013].

SEINE is also related to a significant body of work in the area of languages and tools for building and testing dependable distributed systems. In particular, practical frameworks such as Mace [Killian et al., 2007], Splay [Leonini et al., 2009], and MOLStream [Friedman et al., 2014] provide language support that enables developers to work on the different concerns that comprise a distributed system in isolation, thereby simplifying the overall process. Although performance and dependability concerns are also taken into account by these frameworks (e.g., faults handling support, performance and correctness analysis), there is no explicit guidance for addressing selfishness-related issues. To the best of our knowledge, the only exception is RACOON [Cota et al., 2015], a framework for designing and configuring selfishness countermeasures for P2P systems. RACOON includes a custom built simulation environment that allows assessing the selfish-resilience of the designed system against a fixed set of three simple deviations. On the contrary, *SEINE* supports a considerably more expressive power and customisation of selfish behaviours, allowing for covering most of the state-of-the-art selfishness manifestations, including the small subset supported by RACOON. Furthermore, *SEINE* supports the automatic injection of the specified behaviours into a PeerSim implementation of the system, while in RACOON such behaviours need to be manually implemented for a custom built simulator [Cota et al., 2015].

8.9 Conclusion

In this chapter, we presented *SEINE*, a semi-automatic framework for fast modelling and evaluation of selfish behaviours in cooperative distributed systems. At the heart of *SEINE* is the *selfishness scenario* model that we built through a systematic domain analysis on the subject. Based on this model, we developed an expressive domain-specific language (*SEINE-L*) and run-time support for the specification, implementation, and study of selfish behaviours in the state-of-the-art simulator PeerSim. We illustrated the generality of *SEINE-L* by describing the 15 selfishness scenarios extracted from the domain analysis. Then, we showed the accuracy and ease of use of *SEINE* in evaluating the impact of selfish behaviours in three use cases selected from the literature. The SEINE framework will be made freely available upon acceptance of the paper.

Our future work includes the extension of the framework to offer new selfish deviations (e.g., computation or communication delays) and activation policies (e.g., game-theoretic strategies), as well as to support experiments on different simulators or real testbeds. We will also study how to design a convenient language workbench for *SEINE-L* to further ease the specification, reuse and evolution of selfishness scenarios.

Chapitre 9

RACOON : A Framework for the Design of Accountable Selfish-Resilient Cooperative Systems

A challenge in designing a cooperative distributed system is to ensure that the system is able to tolerate selfish nodes that strategically deviate from their specification whenever doing so is convenient. In this chapter, we propose *RACOON*, a framework for the design of cooperative systems that are resilient to selfish behaviours. While most existing solutions target specific systems or types of selfishness, *RACOON* proposes a generic and semi-automatic approach that achieves robust and reusable results. Also, *RACOON* supports the system designer in the performance-oriented tuning of the system, by proposing a novel approach that combines Game Theory and simulations. We illustrate the benefits of using *RACOON* by designing two cooperative systems : a P2P live streaming and an anonymous communication system. In simulations and a real deployment of the two applications on a testbed comprising 100 nodes, the systems designed using *RACOON* achieve both resilience to selfish nodes and high performance. This work has been carried out in the context of the PhD thesis of Guido Lena Cota and has been published in the proceedings of 34th IEEE Symposium on Reliable Distributed Systems (IEEE SRDS'15). An extension of this work, which is not included in this manuscript on which we investigated the use of evolutionary game theory has also been published in IEEE Transactions on Dependable and Secure Computing (IEEE TDSC 2017).

9.1 Introduction

Today, cooperative systems, such as peer-to-peer (P2P) instant messaging and voice over IP (e.g., Skype), P2P file sharing (e.g., BitTorrent, eDonkey), and P2P live streaming (e.g., P2PTV, PPLIVE, Popcorn Time), are among those generating the most Internet traffic [Cho et al., 2006,Karagiannis *et al.*, 2004,Plissonneau *et al.*, 2005]. The success of these systems mainly resides in their high scalability and robustness to failures, without requiring costly dedicated servers. Common to all cooperative systems, is the assumption that nodes are willing to share their (communication and computational) resources with others. However, in practice [Cunha *et al.*, 2013,Feldman *et al.*, 2006,Hughes et al., 2005], real systems suffer from selfish nodes that decide whether to cooperate depending on the behaviour that increases their utility (e.g., receiving video data, without sharing data with others, in order to save bandwidth).

A number of solutions have been proposed to deal with selfishness in cooperative systems [Aiyer et al., 2005,Ben Mokhtar et al., 2013,Mokhtar et al., 2010,Guerraoui et al., 2010a,Li *et al.*, ,Li et al., 2008]. Most of these solutions rely on Game Theory [Osborne *et al.*, 1994] for modelling and reasoning about selfish behaviours. The classical process for designing selfish-resilient cooperative systems first requires that the system designer exhaustively list the set of deviations that can be performed by selfish nodes. Then, the system designer has to carefully provide an incentive/countermeasure for each deviation he has identified.

Finally, the system designer has to prove that the resulting system is a *Nash Equilibrium* [Osborne *et al.*, 1994]. This proof guarantees that the best strategy for selfish nodes, with respect to a known utility function, is to conform to the system specification. However, carrying out this process is complex, time consuming, and error prone, especially for system designers that are not experts in Game Theory.

An alternative solution to the above *static* approach, is to employ recent accountability mechanisms (e.g., FullReview [Diarra et al., 2014], Lifting [Guerraoui et al., 2010a], and PeerReview [Haeberlen et al., 2007b]), with the goal of *dynamically* forcing nodes to be responsible for their actions. Noteworthy among these mechanisms is FullReview, which appears to be the only one specifically designed for dealing with selfish nodes. In an accountable system, each node maintains a *secure log* to record its interactions with other nodes. Each node is further associated with a set of *monitor* nodes, which periodically check whether the log entries correspond to a correct execution of the underlying protocol. If any deviation is detected, then the monitors build a proof of misbehaviour that can be verified by any correct node, and a *punishment* is inflicted on the misbehaving one.

While making nodes accountable for their actions may constitute an effective incentive for selfish nodes to be cooperative, configuring accountability mechanisms for building a selfish-resilient cooperative system is a challenging task. Indeed, the configuration of accountability mechanisms requires that the system designer set a number of parameters (e.g., number of monitors, frequency of audits, severity of punishment) that directly affect the system performance. In the literature [Diarra et al., 2014, Andreas*et al.*, , Haeberlen et al., 2007b], no indication is provided for the setting of these parameters. Indeed, the calibration of accountability mechanisms should satisfy conflicting requirements : on the one hand, it should provide the desired resilience to selfish behaviours without wrongly penalizing correct nodes, and on the other hand, it should impose minimal overhead. Resolving these constraints requires the manual planning of an extensive number of experiments/simulations, and measuring the impact of each parameter on the system performance. Moreover, this requires the ability to inject and to automatically reason about selfish deviations, which is not possible using existing simulators (e.g., PeerSim [28], NS-3 [29]).

In this chapter, we propose *RACOON*, a design and simulation framework for building selfish-resilient cooperative systems. Specifically, *RACOON* is composed of two parts. The *design* part first allows the system designer to describe the communication protocols of a desired cooperative system, along with a set of performance and selfish-resilience objectives. Then, this part integrates the FullReview accountability mechanisms for detecting faults as well as a reputation system for handling rewards and punishments. Finally, the *design* part automatically extends the provided specifications of the communication protocols with selfish deviations, and transforms the extended specifications into a set of games (one per protocol). The game models provide a mathematical framework to reason about the behaviour of selfish nodes, and are used as input to the second part of *RACOON*, i.e., the *game-based simulation* part. Using a set of simulations automatically carried out in this part, *RACOON* outputs a configuration file for calibrating both the FullReview accountability mechanisms and the reputation system, which allows the resulting system to meet the objectives specified by the system designer.

We demonstrate the benefits of using *RACOON* by designing two selfish-resilient cooperative systems : a P2P live streaming system and an anonymous communication system based on the onion routing protocol [Goldschlag et al., 1999]. Simulations, as well as complementary performance evaluations involving 100 clients on a cluster of real machines, show that the live streaming system configuration chosen by *RACOON* allows correct nodes to visualize a stream of good quality in the presence of selfish nodes. Further, this configuration allows to meet a set of performance requirements set by the designer, including limiting the bandwidth overhead under a fixed threshold. Finally, we show that the process of designing and simulating the anonymous communication protocol reuses more than 42% of the code developed for the live streaming protocol.

In summary, the *RACOON* framework has the following contributions :

- *RACOON* is the first tool that automatically generates selfish deviations and performs simulations involving game theory to reason on the behaviour of selfish nodes ;
- *RACOON* is able to automatically find a configuration of an accountability system that meets a set of performance and selfish-resilience objectives set by a system designer on a given cooperative

28. PeerSim : http ://peersim.sourceforge.net/
29. NS-3 : https ://www.nsnam.org/

system ;
- *RACOON* simulations are accurate compared to the performance of the corresponding real system ;
- *RACOON* specification and simulation code is highly reusable.

The rest of the paper is organized as follows. Section 9.2 presents background on accountability mechanisms and their configuration. Section 9.3 presents an overview of *RACOON*, followed by a detailed description of its two essential components : the design part (Section 9.4), and the game-based simulation part (Section 9.5). Section 9.6 presents a performance evaluation of *RACOON*. Finally, Section 9.7 presents related work, and the paper concludes in Section 9.8.

9.2 Background

Recent solutions for enforcing accountability in distributed systems (e.g., [Diarra et al., 2014,Guerraoui et al., 2010a,Andreas*et al.*, ,Haeberlen et al., 2007b]) rely on secure logging and monitoring mechanisms. We focus on FullReview [Diarra et al., 2014], which enables accountability in the presence of selfish nodes. FullReview applies to a set of N nodes executing a set of protocols P, defined as deterministic state machines. As part of P, each node i interacts with a set of nodes referred to as i's *partners*. When deploying FullReview [Diarra et al., 2014], each node i maintains a *secure log* that is tamper-evident and append-only, in which i records all its interactions with its partners. Further, each node i is assigned a set of *monitor* nodes that periodically verify whether i sticks to the specification of P. If any deviation is detected, i's monitors inflict a *punishment* on i, which could vary from the eviction of i to the reduction of its reputation value, if the system is coupled with a reputation management system.

While enforcing accountability can constitute an effective incentive for selfish nodes to behave cooperatively, configuring the accountability mechanisms in order to discourage selfish behaviours, without excessively degrading performance, is a challenging task. Among the parameters to set are :
- The **number of monitors** associated to each node. More monitors implies more computation and communication overhead ;
- The **audit period** : the period between two log audits initiated by a node's monitors ;
- The **probability of audit** : the probability that a monitor audits its monitored nodes at the end of each audit period ;
- The **reward/punishment function** : the way in which nodes are rewarded (resp. punished) in the case of a successful (resp. unsuccessful) audit.

The last parameter is crucial as setting weak punishments may increase the number of selfish deviations while setting strong punishments may lead to the wrongful eviction of a correct node if the network is not reliable (e.g., in a mobile environment), or if the node gets suddenly disconnected (e.g., characterized by churn in P2P systems).

To highlight the trade-offs that the system designer has to take into account when setting up these parameters, we performed an experiment, involving a gossip-based live streaming protocol monitored by FullReview. In this protocol, a source node disseminates a set of video chunks to a subset of nodes over an unreliable network. Periodically each node sends the video chunks it received to a set of randomly chosen partners, and asks them for any video chunks it is missing. In our experiment, we assume that the system designer aims at designing a selfish-resilient live streaming protocol in which : (1) correct nodes do not experience more than 3% jitter despite the presence of up to 50% selfish nodes ; (2) correct nodes are not wrongfully expelled from the system even if the network suffers from up to 5% of message loss and (3) the average bandwidth consumption per node including both the video stream (which already consumes 600Kbps) and the accountability mechanisms does not exceed 1Mbps. To reach this objective, we start with the FullReview default configuration (i.e., the audit period being 10 seconds, and the probability of audit being 1) and vary the degree of punishments inflicted on nodes by the accountability mechanisms.

Fig. 9.1 shows the percentage of correct nodes wrongly evicted by FullReview and the percentage of selfish deviations observed in the system for various values (in our experiment less than 10% selfish deviations for the selfish nodes translates into an experienced jitter lower than 3%). The results are as expected, showing a clear increase in the percentage of correct nodes wrongly evicted from the system. Nevertheless, the punishment values 1 and 1.5 satisfy the first two requirements set by the designer. We thus go further and measure the communication overhead incurred in the system for 1.5, which

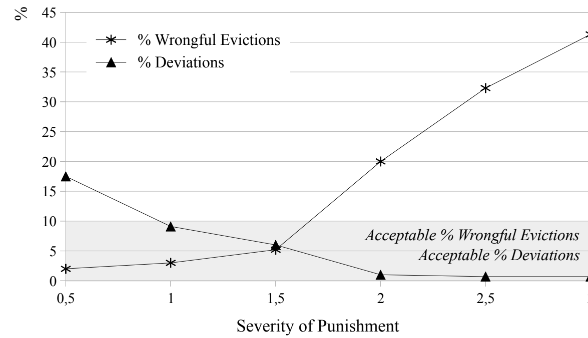


FIGURE 9.1 – Impact of the punishment values.

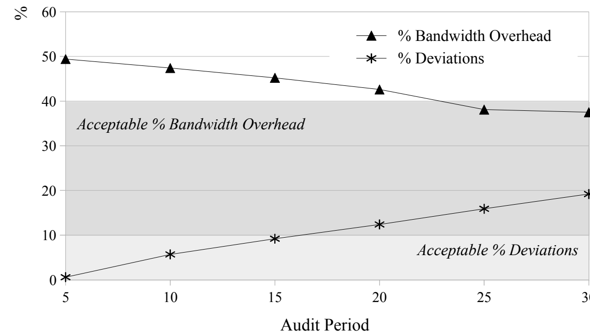


FIGURE 9.2 – Impact of the audit period.

has the lowest % of deviations, while varying the FullReview audit period, as this parameter highly impacts the communication overhead. The results, depicted in Figure 9.2, show that, on the one hand, increasing the audit period decreases the overhead, because logs are requested and audits are made less often by monitors ; on the other hand, the longer the audit period, the slower faults are deterred, thereby increasing the percentage of selfish deviations. However, none of the tested values achieves a bandwidth consumption that meets the third requirement set by the designer. The designer has thus to continue the manual calibration process by testing other pairs of parameter values and running further experiments.

From this experiment, it is clear that manually calibrating accountability mechanisms in order to reach both selfish-resilience and performance objectives is a challenging task. We show in the following sections how *RACOON* helps the system designer automatically reach these objectives.

9.3 *RACOON* Overview

To help a system designer build selfish-resilient cooperative systems, we propose the design and simulation framework *RACOON*. As depicted in Fig. 9.3, *RACOON* is composed of two main parts : the *design part* and the *game-based simulation part*. We give an overview of these parts here, and then provide more detail in Sections 9.4 and 9.5, respectively.

The input of the design part is a system specification provided by the system designer. This specification, referred to as P in Fig. 9.3, contains a set of deterministic state machines, describing communication protocols composing the cooperative system under consideration as well as a set of performance and selfish-resilience objectives. Then, *RACOON* integrates into the provided specification accountability and reputation mechanisms configured with a default configuration (Step (1) in the figure). These mechanisms are used to detect selfish deviations and to reward/punish nodes in case of positive/negative audits, respectively. *RACOON* then automatically extends the state machines with new transitions that represent selfish deviations, according to a selfishness model provided by *RACOON* (Step (2) in the figure). Finally, *RACOON* transforms the extended state machines into games (Step (3) in the figure).

The games produced by the first part of *RACOON*, are used as an input of the second part, i.e., the

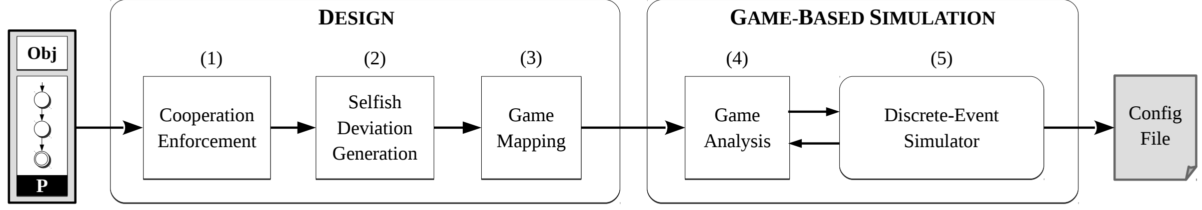


FIGURE 9.3 – *RACOON* Overview.

game-based simulator. The objective of this part is to produce a configuration file for both the accountability and reputation mechanisms that satisfies the performance and selfishness-resilience objectives set by the designer. This is done using game-theory driven simulations that are automatically carried out by *RACOON*. Specifically, each time an action of a selfish node needs to be simulated (Step (5) of the figure), *RACOON* refers to the game analysis (Step (4) in the figure) to identify the best strategy to adopt from the point of view of the selfish node (i.e., whether the latter should stick to a given protocol step or deviate from it). Once *RACOON* has found a set of configuration values for both the accountability and reputation mechanisms that meet the objectives set by the designer, the latter can proceed with the implementation of his system. This is done by including API calls to the accountability mechanisms employed and configuring them as prescribed by *RACOON*.

9.4 *RACOON* Design Part

In this section, we present the design part of *RACOON*. We first present the specification model, then the process by which *RACOON* generates selfish deviations, and finally the transformation of the specification model into a set of games.

9.4.1 *RACOON* Specification Model

RACOON includes a specification model to assist the system designer in correctly describing all the information required by the framework. This model allows describing the *functional specification* of the cooperative system, the *list of objectives* that the system must fulfill, and the *initial configuration* of the cooperation enforcement mechanisms. To illustrate our model and in the rest of this chapter, we use the communication protocol *R&R* (*Request & Response*) shown in Fig. 9.4. In this protocol, a node r_0 sends a request message g_0 to a group of nodes collectively named R_1 (the capital letter denotes a set of unique nodes), and upon receiving g_0, each node in R_1 replies with a response message g_1.

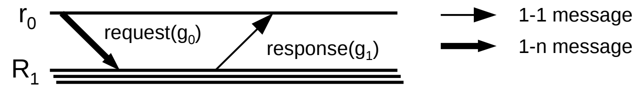


FIGURE 9.4 – The *R&R* protocol between nodes r_0 and R_1.

Functional Specification

The functional specification P of a cooperative system is provided by means of communication protocols : a set of rules of interaction that define what actions (i.e., methods) each node can take at each step. *RACOON* describes each communication protocol in P as a deterministic finite state machine, called a *Protocol Automaton*. A Protocol Automaton PA is a tuple $\langle R, S, s_0, F, T, Meth, G, C, V \rangle$, where :

- $R \neq \emptyset$, is the set of *Roles* that a node may undertake in the protocol. A role can represent either a node or a group of unique nodes. For example, in the protocol *R&R*, there are two roles : r_0 and R_1, where R_1 corresponds to the set of recipients of the message g_0. The cardinality of a role denotes the number of nodes that it represents. More formally, a role $r \in R$ is a tuple $\langle id, cardinality \rangle$;

- $S \neq \emptyset$, is the set of *States* that the system goes through when implementing the protocol. Some special states are : the start state s_0, and the non-empty set of final states $F \subseteq S$, in which the protocol terminates.
- $T \neq \emptyset$, is the set of state *Transitions.* A transition $t \in T$ is a tuple $\langle id, state1, state2, method\rangle$, where $state1$ and $state2 \in S$ are the source and target states, and $method \in Meth$ is the method that triggers t ;
- $Meth \neq \emptyset$, is the set of *Methods.* There are two types of methods : a communication method represents the delivery of a message from one role to another ; a computation method refers to local computations. For instance, in Fig. 9.4, *request* is a communication method that sends a message g_0 to R_1. Formally, a method $m \in Meth$ is a tuple $\langle id, invokerRole, message\rangle$, where $message$ is defined only for communication methods ;
- G is the finite set of *Messages* conveyed by communication methods. A message $g \in G$ is a tuple $\langle id, recipientRole, content\rangle$, where *content* is the content carried by the message ;
- C is the set of *Contents* delivered by the messages. A content $c \in C$ is either a single data-unit (e.g., a binary file), or a collection (e.g., a list of integers). The specification model defines c as the tuple $\langle id, ctype, size, collection\rangle$, where *ctype* is the data type,[30] $size$ is the memory size of a single data-unit in c (given in bytes), and *collection* is a boolean value (i.e., true if c is a set of data-units) ;
- V is the set of content *Constraints.* A constraint prescribes a relationship that has to be fulfilled by two contents. A constraint $v \in V$ is a tuple $\langle id, c1, c2, vtype\rangle$, where the two contents $c1$ and $c2 \in C$ are subject to the relationship defined in *vtype*. Specifically, *vtype* can identify either an ordering relation (e.g., $=, <, >$) or a set operation (e.g., subset, equal).

A Protocol Automaton can be represented by means of a state diagram. Fig. 9.5 shows the state diagram of the *R&R* protocol. The label on a transition provides information about the method that triggers the transition, and about the message that might be sent. For example, the label between states s_1 and s_2, indicates that role R_1 invokes the communication method *response*, which conveys the message g_1 to role r_0.

System Objectives

The specification model includes the list of selfish-resilience and performance objectives that the target system must satisfy. Each objective defines a threshold value for a system metric that can be measured in the *RACOON* simulator (e.g., number of messages sent/received, number of audits performed). *RACOON* natively supports the definition of three kinds of objectives :

- **Deviation rate** : the frequency of the deviations performed by selfish nodes ;
- **Bandwidth overhead** : the bandwidth consumed by the cooperative enforcement mechanisms M ;
- **Wrongful eviction rate** : the percentage of correct nodes wrongly evicted, due to false-positive accusations.

Cooperation Enforcement Mechanisms

RACOON automatically integrates a set of accountability and reputation mechanisms into the system specification. Specifically, in addition to the P protocols described by the designer, *RACOON* applies to each node the FullReview accountability protocols, and the reputation system used by *RACOON*. We refer to the accountability and reputation protocols as the M protocol. FullReview contains a set of verification protocols including an audit protocol and an evidence-transfer protocol. These protocols are periodically executed by nodes to monitor their partners and exchange information about their partners' status. The reputation system, on the other hand, only contains computational methods that are triggered by monitors to update the reputation value of a node after an audit. The state machines corresponding to both kinds of protocols are described in detail in the companion technical report[31].

30. Defined by the XML Schema type system.
31. Racoon Technican Report : https ://sites.google.com/site/soniabm/

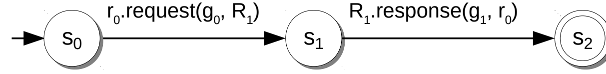


FIGURE 9.5 – The Protocol Automaton of the *R&R* protocol.

As introduced in Section 9.2, FullReview and the reputation system have to be configured with respect to the audit period, the probability of audit, the number of monitors, and the reward/punishment function, respectively. At this stage of the process, these parameters are given their default value.

9.4.2 Generating Selfish Deviations

The utility that a node obtains from participating in a cooperative system is given by : (1) the benefits, i.e., the positive value obtained by consuming resources ; and (2) the costs, i.e., the negative value obtained due to the cost of sharing resources. The utility function is a mathematical model that evaluates the choices facing a node in terms of these quantities.

A selfish node decides whether to stick to the protocol specification or to deviate from it, depending on which option maximizes the utility function. In the context of cooperation, a selfish node can increase its utility by reducing the cost of sharing resources. In this chapter, we consider only deviations that aim at saving bandwidth consumption, leaving the investigation of other types of selfishness (e.g., computational or information-related) for future work.

In a communication protocol, the bandwidth consumption depends on the number and size of the messages that are exchanged between nodes. *RACOON* automatically generates three types of communication-related deviations : (1) *timeout* deviation : the node does not perform the prescribed method within the time limit ; (2) *subset* deviation : the node sends only a subset of the correct message content ; and (3) *multicast* deviation : the node only sends a message to a subset of the legitimate recipients.

Alg. 1 shows the pseudo-code for generating the deviations listed above. The algorithm, called *CDG*, takes a PA as input, and extends it with new elements (states, transitions, roles, etc.) representing deviations.

Hereafter, we describe the pseudo-code in more detail.

Timeout Deviations. For each non-final state $s \in S$, the algorithm generates a timeout deviation by calling the procedure *GenTimeoutDev* (line 3 in Alg. 1). This method creates a new final state s' and a new empty transition connecting s with s'.

Subset Deviations. For each outgoing transition $t \in OT$ of every non-final state, such that t is triggered by a communication method, the algorithm checks whether the content c is a collection of data-units (line 7). If so, line 8 calls the procedure *GenSubsetDev*, which creates new elements to represent the deviation. In particular, the procedure creates the new content c' (line 19), which has the same data type and size as c, but has a single data-unit.

Multicast Deviations. For each outgoing transition $t \in OT$ of every non-final state, such that t is triggered by a communication method, the algorithm checks whether the recipient of the message sent during t has a cardinality larger than 1 (line 12 in Alg. 1). If so, line 10 calls the procedure *GenMulticastDev* to create the role r' (line 27) with a smaller cardinality than the correct one (i.e., cardinality 1).

Fig. 9.6 shows the result of executing the *CDG* algorithm on the Protocol Automaton PA of Fig. 9.5. In the correct execution of PA, in the initial state s_0, the role r_0 sends a message (g_0) to R_1. However, if r_0 is played by a selfish node, he may also (from top to bottom in Fig. 9.6) : timeout the protocol, send a message with a smaller payload (g_0'), or send g_0 to a subset of recipients (R_1'').

9.4.3 Game Mapping

The Protocol Automaton extended in the previous step describes possible behaviours of selfish nodes, but gives no indication of the likelihood of any particular behaviour. *RACOON* uses Game Theory [Osborne *et al.*, 1994] to address this issue. Specifically, the framework provides an automatic tool to translate the PA into a game, referred to as the *Protocol Game PG*. This game provides the necessary

Alg. 1: Pseudo-code for the *CDG* algorithm, which generates communication-related deviations.

```
Data: A Protocol Automaton PA.

Algorithm CDG(PA)
1   foreach non-final state s do
2       r := s.activeRole
3       CreateTimeoutDev(s)
4       OT := outgoing transitions of s
5       foreach transition t ∈ OT s.t. t.method.message ≠ null do
6           c := t.method.message.content
7           if c.collection then
8               CreateSubsetDev(t, c)
9           if t.state2.activeRole.cardinality > 1 then
10              CreateMulticastDev(t)

Procedure GenTimeoutDev(s)
11  s' := ⟨new_sId, s.activeRole, s.audit⟩
12  m' := ⟨"timeout", s.activeRole, 0⟩
13  t' := ⟨new_tId, s, s', m'⟩
14  add s', m', and t' to PA

Procedure GenSubsetDev(t,c)
15  s' := ⟨new_sId, t.state2.activeRole, t.state2.audit⟩
16  c' := ⟨new_cId, c.ctype, c.size, false⟩
17  updateConstraints(c')
18  m := t.method; g := m.message
19  g' := ⟨new_gId, g.recipientRole, c'⟩'
20  m' := ⟨new_mId, m.invokerRole, g'⟩
21  t' := ⟨new_tId, t.state1, s', m'⟩
22  add s', c', g', m', and t' to PA
23  copyOutgoingTransitions(s',t.state1)

Procedure GenMulticastDev(t)
24  r' := ⟨new_rId, 1⟩
25  s' := ⟨new_sId, u', t.state2.audit⟩
26  m := t.method; g := m.message
27  g' := ⟨new_gId, u', g.content⟩'
28  m' := ⟨new_mId, m.invokerRole, g'⟩
29  t' := ⟨new_tId, t.state1, s', m'⟩
30  add r', s', g', m', and t' to PA
31  copyOutgoingTransitions(s',t.state1)
```

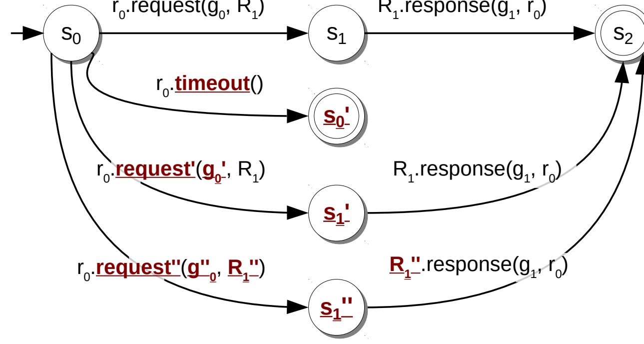


FIGURE 9.6 – The Protocol Automaton with selfish deviations.

structure to support the next step of *RACOON*, i.e., the game-based simulations. The process by which this game is generated is described below.

Game type and Players. The Protocol Automaton PA is modelled as a non-cooperative sequential game among self-interested players [Osborne *et al.*, 1994]. Each player is assigned to exactly one role $r \in R$. The game is non-cooperative because each player competes against the others for maximizing his own utility. At the same time, the game is sequential because players have a specific order of actions to follow, as specified by PA.

A sequential game is usually represented as a tree, also called *extensive form representation* [Osborne *et al.*, 1994]. A node in PG is derived from a state in PA, and is labelled with the player who has to move. Each leaf in PG translates to a final state in PA, while each edge in PG corresponds to a transition in PA. Edges in PG are labelled with *actions*, which correspond to methods m defined in PA. The sequence of actions that a player p_i might choose in a play constitutes his *strategy*. A *strategy profile* is a vector specifying a strategy for every player. Fig. 9.7 shows the extensive form representation of the game derived from the *R&R* protocol.

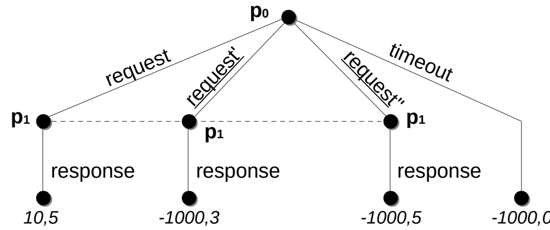


FIGURE 9.7 – The Protocol Game derived from the *R&R* protocol.

Utility function and payoffs. The utility function of a player assigns a *payoff* to every possible game outcome (i.e., the result obtained in correspondence to a strategy profile). The utility function implemented by *RACOON* has two terms : the cost k of sharing resources, and the incentives provided by the cooperation enforcement mechanisms.[32] Because in this chapter we only consider communication-related selfishness, k is calculated as the bandwidth necessary to implement a certain player's strategy. Furthermore, we calculate incentives as a function ρ of the reputation variation ΔR determined by an audit. Note that ΔR also depends on the current reputation of the node, for the definition of the reputation update function.

Formally, we write the utility r_i for player p_i when implementing his strategy σ_i as follows :

$$u_i(\sigma_i) = k(\sigma_i) + \rho(\Delta R)$$

The definition of $\rho(\Delta R)$ takes into account the following considerations. The selfish nature of players encourages them to deviate from the protocol as much as possible (to save resources), as long as this

32. We assume that all players obtain the same benefit from playing the game.

can be accomplished without causing their eviction from the system. Therefore, if a player's reputation increases at a point in time ($\Delta R > 0$), then he is more likely to deviate in the future, as he will be further from the eviction threshold than before. In this case, the function ρ assumes a positive value, which corresponds to a payoff increment. Likewise, on the contrary, ρ assumes a negative value if a player's reputation decreases ($\Delta R < 0$). A particular case is if the reputation value goes below the eviction threshold, in which case the player is evicted from the system. This negative event by itself corresponds to a cost, conventionally set at a very high nominal value.

In Fig. 9.7, the pairs of numbers (one for each player) below each leaf are the payoffs, which express the utility of playing the strategy that terminates in in that leaf. In the following, we briefly discuss how some of the payoff values in Fig. 9.7 are obtained. Let us consider the following setting of the *R&R* protocol. The role r_0 has to send a request message g_0 to role R_1, with R_1 having cardinality 5. The request message conveys a content c_0 : a list of $n = 10$ boolean values, each of size 1 byte. Thus, for instance, the communication cost k for player p_0 when playing the strategy σ_1 (i.e., sending the correct request message) is :

$$k(\sigma_1) = R_1.cardinality \times (c_0.size \times n) = 50$$

Let us assume that the execution of M returns : $\Delta R = 1$ when p_0 chooses the correct strategy σ_1, and $\Delta R = -1$ otherwise (i.e., selfish deviations). Fig. 9.7 illustrates a possible payoff structure that can be obtained from the above setting, in which $\rho(1)$ determines a payoff increment of 60, and $\rho(-1)$ determines a payoff decrement of -1000.

9.5 *RACOON* Game-Based Simulation

We now describe how *RACOON* computes a configuration for the accountability and reputation mechanisms that ensures the enforcement of the system objectives set by the system designer. This is realized in two interleaving phases : simulation and game analysis.

The simulation phase explores the space of the parameters presented in Section 9.2, searching for a setting that reaches the objectives defined by the system designer. The automatic approach adopted by *RACOON* is to simulate the target system in different regions of the parameter space, using game-theoretic analysis to drive the behaviour of selfish nodes. The exploration ends when a configuration satisfying the system designer's objectives is found.

The *RACOON* framework includes a discrete-event simulator for P2P overlay networks, which uses as input the specification P of the cooperative system (extended with selfish deviations), and the objective requirements to achieve. This simulator supports a cycle-based simulation model. Specifically, at each cycle, every node executes P in turn. Correct nodes will never deviate from the correct implementation of P, while the behaviour of selfish nodes can change from one cycle to another according to the action that maximizes their utility.

To simulate the behaviour of selfish nodes, the *RACOON* simulator (*R-sim*) interacts with the game-theoretic tool (*GT-tool*) included in the framework. At each simulation cycle, and for each Protocol Automaton $PA \in P$, the interaction between *R-sim* and *GT-tool* proceeds as follows : (1) *GT-tool* translates PA into a Protocol Game PG, and configures it with the current reputation of the interacting nodes (provided by *R-sim*) ; (2) *GT-tool* conducts a game analysis to identify the best strategy of each node ; (3) *GT-tool* returns the obtained strategies to *R-sim*, so that they can be executed in the simulation.

The game-theoretic analysis performed at step (2) determines the possible steady states of PG, which are the equilibrium points. *RACOON* uses the Sequential Equilibrium (SE) solution [Osborne *et al.*, 1994], a refinement of the Nash Equilibrium for sequential game with imperfect information. To find the SE of PG, *GT-tool* uses Gambit,[33] an open-source library of tools for solving non-cooperative games. Specifically, Gambit implements the algorithm by Koller, Megiddo and von Stengel [Koller et al., 1994], using linear programming. In the Protocol Game of Fig. 9.7, the SE found by Gambit is the strategy profile *(request, response)*, which indicates that the expected behaviour of players p_0 and p_1 is to execute the methods with *request* and *response*, respectively. The *RACOON* simulator uses this information and

33. Gambit : http ://sourceforge.net/projects/gambit/

simulates the players' behaviour accordingly. For example, if at a given turn the equilibrium strategy of a selfish node is to perform a timeout deviation, then the simulator will skip any execution of the communication protocol of that node in that turn.

A single simulation allows verifying the selfish-resilience and performance guarantees offered by a given configuration of the accountability and reputation mechanisms. To find a configuration that meets all the objectives set by the designer, *RACOON* explores the space of configuration parameters using a greedy algorithm optimized with a set of heuristics. For instance, if when simulating a given configuration the overhead is already above the threshold fixed by the designer, *RACOON* will not increase the number of monitors, the probability of audit or the audit period in the next configuration to be explored as this would further increase the overhead. We show in the following section that thanks to our heuristics, *RACOON* manages to converge in a reasonable time (18 minutes on average). We plan to investigate more optimized exploration algorithms (e.g., simulated annealing) in future work.

The outcome of the simulator is a configuration of the cooperation enforcement mechanisms that reaches all the objectives of the system designer. If no configuration is found (which may happen if the specified objectives are contradictory), the simulator asks the designer to relax some of his objectives. Once a configuration is found, the system designer proceeds with the implementation of the cooperative system, in which he integrates the FullReview API calls configured using the configuration file provided by *RACOON*. In our future work, we further aim at automatizing the implementation step, by extending *RACOON* with a new module for the generation of executable code. There already exists good solutions for generating code (e.g., MACE [Killian et al., 2007]), which we plan to integrate in the future version of our framework.

9.6 Evaluation

In this section, we demonstrate the benefits of using the *RACOON* framework to design selfish-resilient cooperative systems. First, we assess the *design effort* required by the system designer to specify the P2P live streaming protocol of Section 9.2, along with a set of objectives he wants to achieve. Second, we assess the *effectiveness* of *RACOON* by comparing the quality of a configuration it finds with a set of FullReview configurations. Third, we assess the *accuracy* of the simulations performed by *RACOON* compared to a real implementation of the accountable live streaming system. Further, we evaluate the *performance* of *RACOON* by measuring the average time necessary to find satisfactory configurations in 30 different use cases. Finally, we show the degree of *re-usability* of the *RACOON* specification and simulation code by evaluating the effort required by the system designer to specify and simulate an anonymous communication protocol starting from the live streaming protocol.

9.6.1 *RACOON* Design Effort

We show in this section, the effort necessary for the system designer to describe a live streaming system using *RACOON*. The functional specification of this system, briefly introduced in Section 9.2, defines the three-phase ($3P$) gossip-based protocol studied in [Guerraoui et al., 2010a] and depicted in Fig. 9.8. This protocol involves two roles : the supplier S proposes the set of chunks it has received to a set of consumers C, which in turn request any chunks they need. The protocol ends when S sends to C the requested chunks. This protocol is executed periodically by the set of nodes participating in a live streaming session. Each chunk is associated with an expiration time (i.e., the *play-out delay*). A node can only playback the chunks that have not yet expired. Fig. 9.9 illustrates the Protocol Automaton of the $3P$ gossip protocol. The full *RACOON* specification of this protocol is found in the companion technical report [34].

In our experiment, we assume that the system designer wants to set the objectives introduced in Section 9.2 :

Obj.1 A deviation rate lower than 10% ;

Obj.2 A bandwidth overhead lower than 40% ;

34. Racoon Technical Report : https ://sites.google.com/site/soniabm/

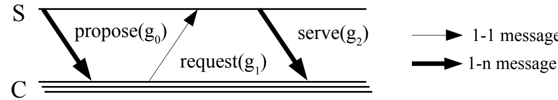


FIGURE 9.8 – The sequence diagram of the $3P$ gossip protocol.

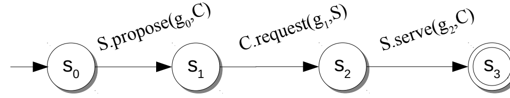


FIGURE 9.9 – The Protocol Automaton of the $3P$ gossip protocol.

Obj.3 A wrongful eviction rate lower than 10%.

Overall, the XML specification of this protocol contains 48 lines. In addition to writing this specification, the system designer has to develop a new module for the *RACOON* simulator, which implements the P2P live streaming system model. The implementation classes of this module contain (500 lines of code (LOC). This includes the implementation of the $3P$ gossip protocol and custom monitors to measure live-streaming metrics (e.g., the number of chunks transmitted/received). The overall simulation code of this application further uses 3200 LOC provided by *RACOON* libraries (e.g., the simulation engine and the exploration algorithm, FullReview, and the reputation system).

9.6.2 Meeting System Objectives Using *RACOON* Simulations

Given the above specification and the corresponding simulation code, *RACOON* explores the space of possible configurations by running a set of simulations to find a configuration that satisfies the objectives set by the designer. To carry out these simulations, we configured the live streaming system using the parameters depicted in the second column of Table 9.1.

The configuration proposed by *RACOON* is depicted in the last column of Table 9.2. We compare the performance of this configuration with the five FullReview configurations depicted in the same table. We selected these configurations by varying two parameters : the audit period and the severity of the punishment. The first four configurations correspond to four combinations of low and high values of these parameters (referred to as L and H, respectively in the configuration names). These values are the lowest and highest values tested in the experiments of Section 9.2, respectively. Besides these combinations, we selected the best configuration found in Section 9.2 (labelled M-M in Table 9.2) as it already satisfies the first two requirements set by the designer.

Fig. 9.10 shows the simulation results of the six configurations. The *RACOON* configuration is the only one that fulfills all the design objectives, which are depicted as horizontal dotted lines in the figure. Furthermore, this configuration provides up to 33% fewer deviations, 42% fewer wrongful evictions and 17% lower overhead than the others.

9.6.3 Simulation Compared to Real System Deployment

To demonstrate the accuracy of *RACOON* simulations, we implemented a prototype of the live streaming protocol described above. We configured the accountability and reputation mechanisms using the parameters depicted in the last column of Table 9.2. Then, we deployed the prototype on a cluster of real

Parameter	Simulation	Experiment
Network size (nodes)	1000	100
Broadcast bandwidth (Kbps)	600	600
Partner set size (nodes)	7	7
Play-out delay (rounds)	6	6
Bandwidth capacity (Kbps)	1000	1000

TABLE 9.1 – Simulation and Real Deployment Parameters.

	L-L	L-H	H-L	H-H	M-M	*RACOON*
Audit Period	5	5	30	30	15	5
Punishment	0.5	3.0	0.5	3.0	1.5	1.0
Prob. of Audit	1.0	1.0	1.0	1.0	1.0	0.5

TABLE 9.2 – FullReview Configurations

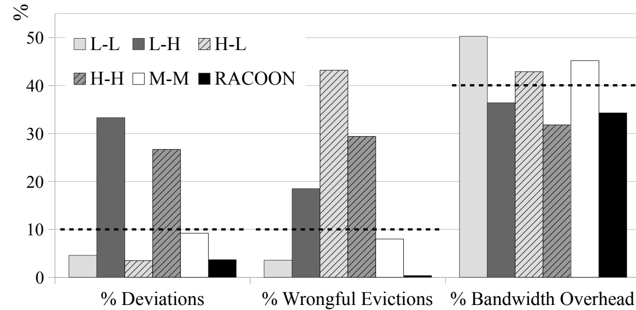


FIGURE 9.10 – *RACOON* vs FullReview Configurations.

machines.[35] Specifically, we run 100 clients on 10 eight-core physical machines. Each machine is clocked at 2.5GHz with 32GB of RAM, and is interconnected with the others via a Gigabit switch. We performed our experiment in this environment in order to measure the impact of selfish nodes on the observed jitter without the risk of fluctuating networking conditions. Otherwise, it would be difficult to assess whether a deteriorated stream quality comes from selfish nodes or from the transient network.

The third column of Table 9.1 describes our experimental settings. Note that the only difference with the simulation settings is the number of nodes in the network. In this experiment, we measure the jitter experienced by correct nodes as a function of the fraction of selfish nodes in the system.

Fig. 9.11 presents the results of our evaluation. This figure contains a curve showing the impact of selfish nodes on traditional Gossip (i.e., without any accountability mechanisms) as well as the two curves for the system designed using *RACOON*. The "SIM - *RACOON*" curve is obtained using *RACOON* simulations, whereas the "G5K - *RACOON*" curve is obtained using the real deployment. From this figure, we observe that without accountability mechanisms, correct nodes experience 10% jitter with only 10% of nodes behaving selfishly, which prevents them from watching the video stream. Further this figure shows that the simulated curve and the real one, overlap up to the inclusion of 50% of selfish nodes in the system. Above this value the curves still exhibit a comparable shape. Finally, this figure shows that despite 90% of nodes becoming selfish, the configuration found by *RACOON* allows correct nodes to watch the video stream with a jitter lower than 3%, which reflects the effectiveness of *RACOON*.

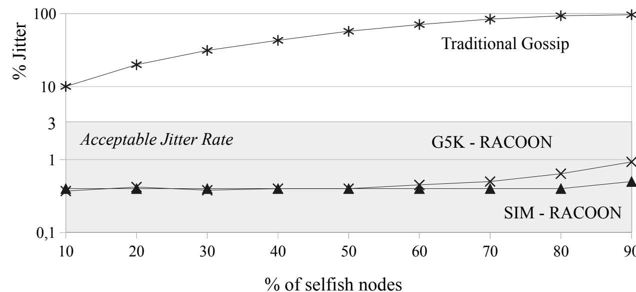


FIGURE 9.11 – Simulation vs real deployment (logarithmic scale).

35. Grid'5000 : http ://www.grid5000.fr

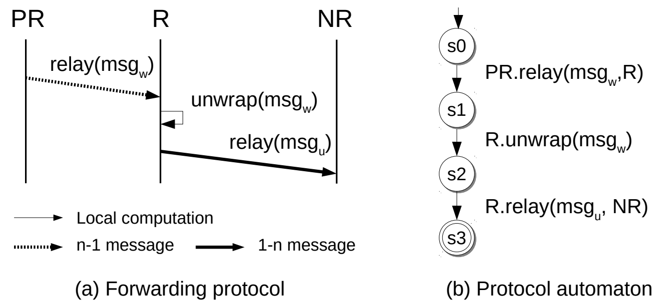


FIGURE 9.12 – Onion Forwarding Protocol.

9.6.4 *RACOON* Execution Time

To evaluate the time necessary for *RACOON* to find a satisfactory configuration, we performed the following experiment. First, we defined a set of 30 different scenarios in the live streaming application. Each scenario is a unique combination of the following elements : system objectives, simulation settings (e.g., number of nodes, bandwidth capacity, play-out delay), percentage of selfish nodes in the system, and message loss rate. Second, we measured the number of configurations that *RACOON* explores in each case before finding a satisfactory solution. Finally, we average these numbers over the total number of scenarios that have been considered. The results show that, for each scenario, an average of 26 configurations are explored by *RACOON* before finding a satisfactory one. Considering that each configuration corresponds to one executed simulation, and that each simulation lasts approximately 42 seconds,[36] the exploration algorithm takes on average about 18 minutes to complete. This duration appears to be reasonable as all the activities performed by *RACOON* are done offline at design time.

9.6.5 *RACOON* Expressiveness

We conclude this section by illustrating the generality of our framework. To this end, we use *RACOON* to design an anonymous communication protocol based on the Onion Routing protocol [Goldschlag et al., 1999]. In Onion Routing, when a source node wants to send a message to a destination node, it builds a circuit of *relay* nodes. The source node changes the circuit periodically, and relays can support many circuits simultaneously. To achieve anonymity, the source uses the public key of each relay along the circuit to successively encrypt the message, which constitutes an *onion*. Fig.9.12a illustrates the protocol enabling the forwarding of onions. In this protocol, each relay R : (i) receives onion messages from their predecessor in the circuit (PR) ; (ii) decrypts the external layer of each onion ; (iii) forwards the resulting onions to their respective successor NR.

To design a selfish-resilient onion forwarding protocol using *RACOON*, the system designer follows the same steps we have seen to design the live streaming system. First, he provides the *RACOON* specification of the system. Due to space limitations, we provide the full specification in the companion technical report[37]. This specification describes three roles, which are the relay R, and its previous (PR) and next (NR) hops. Note that PR and NR have cardinality $n \geq 1$, because they represent a set of relay nodes. The Protocol Automaton of the forwarding protocol can be modelled as in Fig. 9.12b. The protocol has a communication method and a computation method. The communication method *relay* sends messages that carry the single onion to forward. The second method, called *unwrap*, is a decryption operation (i.e., local computation). A selfish relay R that aims at saving bandwidth, strategically drops onions that are not intended for it.

Designing this system using *RACOON* only required writing 44 lines of XML specification and 350 LOC for the implementation of the simulation module. Notice that the system designer can reuse more than 42% of the implementation code written for the P2P live streaming module, and only requires to implement the methods of the Protocol Automaton depicted in Fig. 9.12. The overall simulation code

36. Average value over 1000 simulations, run on a 2.8 GHz machine with 8 GB of RAM.
37. Racoon Technical Report : https ://sites.google.com/site/soniabm/

further uses the same 3200 LOC used in the P2P live streaming protocol and provided by *RACOON* libraries. Simulations of this use case realized using *RACOON* can be found in the companion technical report [38].

9.7 Related work

Game Theoretic approaches

Much work on the potential of Game Theory as tool for system designers has been carried out in the context of content-disseminating applications [Li *et al.*, ,Ben Mokhtar et al., 2014,Li et al., 2008], wireless networking [Cagalj et al., 2005,Srivastava et al., 2005], exchange protocols [Buttyán and Hubaux, 2001], and anonymity and privacy mechanisms [Ben Mokhtar et al., 2013,Freudiger *et al.*, 2009]. The objective of these approaches is to make cooperation the best choice for all nodes, i.e. a Nash Equilibrium. Although promising results have been achieved, most of the reported solutions are tailored to a specific system or are too difficult to adapt to a changing environment. A notable example is the BAR Model of Aiyer et al. [Aiyer et al., 2005], which is a general architecture for building cooperative systems that are robust to selfish and Byzantine nodes. However, the design of a BAR-tolerant protocol is a complex task that has to be performed manually [Aiyer et al., 2005, Mokhtar et al., 2010, Li *et al.*,]. Moreover, this approach suffers from poor maintainability and reusability : every change in the system parameters requires a full revision of the solution.

Non-Game theoretic approaches

Incentive-based mechanisms discourage selfish behaviors by making cooperation more attractive for selfish nodes. For example, in credit-based systems [Buttyán and Hubaux, 2003, Zhong *et al.*, 2003], cooperating nodes gain credits, which are the virtual currency to spend for using the system. These approaches require a trusted central authority, which may not be consistent with the system under design (e.g., in ad hoc mobile networks). On the contrary, the *RACOON* framework adopts a generic distributed solution, i.e., the cooperation enforcement mechanisms are deployed on each node. Another popular form of incentive mechanism is reputation [Marti and Garcia-Molina, 2006], which is, in the context of our work, a measure of a node's cooperation. However, a selfish node can cheat the system by misreporting reputation information. In the reputation system included in *RACOON* such cheating become detectable and punishable, because the reputation value depends on the (provable) auditing results of an accountability mechanism.

Accountability approaches such as [Diarra et al., 2014, Guerraoui et al., 2010a, Andreas*et al.*, , Haeberlen et al., 2007b] provide another strategy for dealing with selfish nodes. An accountability system discourages deviations by exposing a misbehaving node to the risk of punishment. Despite the high generality and the proven effectiveness of these approaches, their application incurs a non-negligible cost to the system (e.g., message overhead, intensive use of cryptography). Moreover, they often rely on strong assumptions (e.g., the presence of a quorum of correct nodes, no message loss [Diarra et al., 2014,Haeberlen et al., 2007b]). *RACOON* proposes a semi-automatic approach to configure an accountability mechanism (i.e., FullReview [Diarra et al., 2014]) in such a way as to achieve good performance and selfish-resilience.

Finally, several frameworks [Urbán et al., 2001] and Domain-Specific Languages [Biely et al., 2013, Killian et al., 2007] have been proposed to ease the task of designing and maintaining secure distributed system. Although these solutions yield good results in terms of system performance and designer effort, none of them address the specific threat of selfish deviations in cooperative systems.

9.8 Conclusion

In this chapter we presented *RACOON*, a novel framework for designing and configuring distributed cooperative systems that are resilient to selfish nodes. *RACOON* relies on accountability and reputation mechanisms to enforce cooperation among selfish nodes. Using a combination of simulation and Game

38. Racoon Technical Report : https ://sites.google.com/site/soniabm/

Theory, RACOON automatically configures these mechanisms in a way that meets a set of selfish-resilience and performance objectives specified by the system designer. We illustrated the benefits of using *RACOON* by designing a P2P live streaming system and an anonymous communication system. The evaluation of the P2P live streaming system performed using both simulations and a real deployment shows that this system achieves selfish-resilience and high performance. Furthermore, we showed that the *RACOON* specification and simulation code are highly reusable.

Our future work include the integration of a domain specific language (e.g., MACE [Killian et al., 2007]) into *RACOON* to automatically generate executable code. This could be done by defining transformation rules between the *RACOON* specification model and the MACE language.

Troisième partie

Conclusion and Future Research Directions

Chapitre 10

Conclusion and Perspectives

In the context of this thesis we presented contributions that increase the robustness of distributed systems against Byzantine and rational behaviours. We will present in this chapter a summary of these contributions (Section 10.1) the latest advances in this topic (Section 10.2 and 10.3) before discussing future research directions (Section 10.4).

10.1 Summary of contributions

We presented in this manuscript a set of contributions for building Byzantine resilient and rational-resilient systems. Specifically, we first presented RBFT, a robust Byzantine-fault tolerant state machine replication protocol. The main feature of RBFT is that it offers performance guarantees under Byzantine attacks (i.e., when all Byzantine nodes in addition to Byzantine clients collude to attack the system). We then presented FullReview, an accountability system that enables the detection of Byzantine nodes using a collaborative monitoring system in presence of Byzantine and selfish monitors. We further presented PAG, a privacy-preserving accountability protocol targeted to gossip-based message dissemination. Follows a set of contributions related to dealing with selfishness in distributed systems. In this context, we first presented FireSpam a protocol for dealing with rational nodes in gossip-based spam filtering. We then presented two protocols one that addresses the problem of colluding rational nodes (i.e., AcTing) and another one that aims at preventing selfishness when nodes are anonymous (i.e., RAC). Finally, the two last contributions of this manuscript are generic tools for injecting selfishness behaviours in distributed systems (i.e., Seine) and for plugging and tuning selfishness countermeasures in distributed systems (i.e., Racoon).

There are a number of short term perspectives that we could work on to extend or improve the works proposed in this manuscript. Among these perspectives we are currently working on countermeasures to a security attack performed on the secure logs used in the PAG and AcTing protocols [Amrit et al., 2017]. The fixes have been found and are currently being tested before publication. Another interesting perspective would be to integrate Seine and Racoon to enable the end-to-end design and testing of selfish-resilient systems. Seine could also be extended with the support of real system deployments instead of (or in addition to) simulations.
Other long term perspectives are further discussed later in this chapter.

10.2 Latest advances in building Byzantine tolerant systems

There have been many efforts to build and improve Byzantine fault tolerant systems (and in particular state machine replication protocols) in the last decade. We describe here the two major trends that aim at increasing the adoptability of such systems that have been often criticized due to their cost. The first direction (described in Section 10.2.1) is towards improving the performance of BFT systems (e.g., in terms of latency and throughput). The second direction (described in Section 10.2.2) is towards reducing the necessary number of replicas.

10.2.1 Making BFT protocols faster

The emergence and wide propagation of multicore machines has led to a number of research works that exploit machine parallelism to improve the performance of BFT protocols. Examples of these protocols include Eve [Kapritsos et al., 2012], in which replicas start by speculatively executing groups of requests concurrently and then verify that they can reach agreement on the output produced by a correct replica ; if they can not, they would roll back and execute the requests sequentially.

Another protocol, which share some similarities with our proposed RBFT protocol is COP [Behl et al., 2015] in which consensus instances are executed in parallel. The parallelism enabled by multi-core machines is thus heavily exploited, which allows COP to reach unprecedented throughput.

Complementary solutions to make BFT protocols more efficient are the ones that adapt to the underlying workload. Changing BFT protocols to adapt to the underlying workload has first been proposed in [Guerraoui et al., 2010b]. Another solution presented in [Bahsoun et al., 2015] uses machine learning algorithms to detect the changes in working conditions (e.g., workloads) and choose the most appropriate BFT protocol accordingly.

All these offorts towards having faster BFT protocols increase the chances that production systems will eventually integrate them. Recently many Blockchain initiatives (e.g., HyperLedger fabric [39]) are investigating the utilisation of existing fast BFT protocols instead of the Nakamoto Consensus protocol based on proof-of-work.

10.2.2 Reducing the cost of BFT protocols

Classical BFT protocols have to rely on $3f + 1$ replicas to ensure that the replicated service remains operational even if up to f of these replicas behave arbitrarily. In order to encourage the wide adoption of BFT protocols, researchers have targeted the reduction of the number of required replicas. Besides reducing the number of physical machines, these solutions allow the reduction of the computation and communication cost of service replication. To reach this objective, researchers generally rely on secure hardware (e.g., A2M, Trinc) to prevent or detect equivocation : that is the fact that a malicious replica sends two conflicting statements to two distinct sets of correct nodes. Using a trusted subsystem allowed researchers to move from the required $3f + 1$ nodes to tolerate f Byzantine nodes down to $2f + 1$ nodes [Chun et al., 2007, Veronese et al., 2013]. A different solution based on an FPGA-based trusted subsystem for the authentication of protocol messages has also been proposed in [Distler et al., 2016]. More recently, a solution based on Intel SGX enclaves has been proposed [Behl et al., 2017].

Other approaches to build cheap BFT protocols than the ones that use trusted hardware include solutions that modify the system and network assumptions. An example of such protocol is the XFT protocol [Liu et al., 2016], which works with a weaker yet (expected to be) more realistic adversarial model in which the adversary does not take control of the faulty replicas and of the network at the same time.

With the expected wide propagation of secure enclaves in commodity hardware and even in smartphones as shown in the latest iPhone which embeds a secure enclave chip, BFT protocols that rely on trusted subsystems may be eventually adopted in practice.

10.3 Latest advances in building rational resilient cooperative systems

In addition to the application domains considered in this manuscript, rational/selfish behaviours have been also studied in the context of mobile systems. There are a number of mobile applications that have been considered in this context. These include edge-assisted video streaming [Vilaça et al., 2017], routing in ad-hoc networks and delay-tolerant networks [Jedari et al., 2017] and crowd sensing applications [Li et al., 2016]. One of the challenges in these systems is to deal with the lack of resources of the involved thin devices. Hence, due to this constraint, selfish behaviours are more likely to occur as mobile users may be tempted to installing software that would save their battery lifetime. In this context, a challenge

39. HyperLedger Fabric : https ://hyperledger-fabric.readthedocs.io

for countermeasure developpers is that the latter need to be resource effective in order to be usable in practice.

Techniques used to deal with rational nodes in mobile systems either rely on classical game theory on which strong incentives are implemented to enforce cooperation between nodes or on evolutionary game theory that enable incremental incentives that evolve according to the dynamic behaviour of nodes in the system [Vilaça et al., 2017].

With the envisioned advent of IoT applications, which part of them will involve mobile things relying on mobile communication protocols, it is likely that the above studies will serve as a starting point for dealing with node selfishness. Novel issues may also arise as further discussed in the following section.

10.4 Open research directions

10.4.1 Collaborative systems are not dead

Collaborative (Peer-to-Peer) systems are back. After decades of research in this topic at the heart of which fundamental contributions have been proposed such as Gossip protocols and distributed hash tables (DHTs) ; after a number of successful applications including (legal/illegal) planet scale file sharing (eMule, BitTorrent) and anonymous communication protocols (TOR) ; collaborative systems are back with a pletora of new challenges.

Among the collaborative applications that are already in place are crypto-currencies and their underlying blockchain technology. The success of these applications and the possible money to be earned from participating in them has attracted many cyber-criminals around the planet. A number of the carried attacks can be categorized as selfish attacks where users deviate from the original protocol to illegitimately earn financial benefits. Preliminary research works have been done for studying selfishness in Bitcoin including the works described in [Catalini and Gans, 2016], [Zhang and Preneel, 2017] [Kwon et al., 2017] [Göbel et al., 2016] [Sapirshtein et al., 2016].

Other envisioned collaborative application domains include collaborative IoT and self driving cars. In collaborative IoT [Behmann and Wu, 2015], researchers imagine a world of connected things that would locally collect, share and process users and contextual data to offer them added value services. Some of these works include the usage of computational resources available in the vicinity of connected objects (i.e., edge and fog computing).

In self driving cars [Blevis, 2017], cars equipped with thousands of sensors as well as powerful computing and networking capabilities, interact with the environment (e.g., road, other cars) to safely drive users to their desired destination. The first prototypes of self driving cars are already driving people in the city of Pittsburgh (USA) [40].

These application domains definitely open new challenges for dealing with selfish behaviours. Indeed, assumptions often made in game-theory based solutions (e.g., the fact that utility functions of rational nodes are known in advance, the fact that rational nodes are risk averse) do not always stand in real life. In the context of the envisioned applications, selfish behaviours may have disastrous consequences (e.g., a selfish self-driving car may cause accidents). It is thus important to design selfishness countermeasures that are suitable to these type of environments. In addition to selfishness prevention through the deployment of appropriate incentive schemes, it is important to augment the underlying systems with selfishness detection mechanisms (i.e., appropriate accountability mechanisms).

10.4.2 The shift towards a Big Data world and the need of data privacy mechanisms

Nowadays, a wide variety of online services (e.g., web search engines, location-based services, recommender systems) are being used by billions of users on a daily basis. Key to the success of these services is the personalisation of their results, that is returning to each user those results that are closer to her interests. For instance, given a web search query sent by two different users, search engines generally rank differently the search results to best fit each user preferences [Langville and Meyer, 2011]. The latter

40. Uber Self Driving Cars : https ://www.uber.com/cities/pittsburgh/self-driving-ubers/

preferences are generally computed by relying on user profiles that are learnt from past user queries. However, according to the underlying application, user profiles may contain sensitive information about end users. For instance, in the context of location-based services, user profiles contain user mobility data from which it is easy to infer information such as a user's home and workplace or even her sexual, religious or political preferences if she regularly visits gay bars, worship places or the head quarter of a political party [Gambs et al., 2010]. However, user profiles, which are widely exploited by online service providers due to their inherent business model based on online advertising, might severely threaten user privacy if they end up into the hands of untrusted services. Recent events have shown that the latter risk of data privacy leakage is becoming a reality due to the massive use of cloud services by online service providers and by end users. Indeed, a 2014 study from Gartner found that 94% of organizations either already are or plan to store their consumer data in the cloud. Another study by the same organization envision that the "Cloud Shift" by 2020 will affect more than 1 Trillion dollars in IT spending[41]. On the same line, cloud providers are experiencing an exponential growth of their storage capabilities[42]. Due to their success and to the unprecedented value that can be extracted from the data they store (e.g., financial information, health information, trade secrets, intellectual property) cloud providers are becoming the target of devastating attacks[43]. Examples of such attacks that took place in 2016 include hospital ransomware, millions of Dropbox account details leaked, other millions of snapchat accounts compromised to cite a few. In this context, it becomes urgent to devise mechanisms that allow users to securely access online services without fearing that their data will be leaked out from the cloud platforms where it is being stored and processed. For addressing this challenge, the research community in the past years has been very active in devising mechanisms for accessing online services in a privacy-preserving way (e.g., [Guha et al., 2011, Petit et al., 2015, Guha et al., 2012, Aïmeur et al., 2008]) or for designing novel secure online services (e.g., [Chor et al., 1995]). However, while the former make rather far reaching trust assumptions, the latter rely on heavy cryptographic techniques. In practice, existing solution can hardly be transposed to reality as the first group of solutions incurs the risk that the assumptions gets broken whilst the second group has typically high resource demands and severely degraded service performance. In this context, one of the open challenges is to devise practical protocols for enabling users to use online services in a secure and privacy-preserving way (i.e., reaching at the same time satisfactory security/privacy and performance properties). One of the possible research direction to reach this objective is exploring disruptive hardware technologies such as Intel SGX and ARM trust zone to develop new efficient and privacy preserving protocols. There have been many success stories for using these technologies in the last couple of years [Pires et al., 2016, Schuster et al., 2015, Zhu et al., 2016, Tang and Cai, , Ben Mokhtar et al., 2017]. There also have been some attacks (mostly using side channel data) against these system that need to be addressed but the research community is already investigating solutions to these attacks [Shih et al., 2017].

A third dimension to consider is related to the quality of the service obtained after applying data protection mechanisms. Indeed, a number of privacy-preservation mechanisms add noise to user data (e.g., protection mechanisms that operate on location data may add spatial or temporal noise to the latter [Andrés et al., 2013, Primault et al., 2015]). As a result an online location-based service (e.g., a service for finding restaurants in the vicinity of the user or a GPS navigation service) may have its offered service altered if not disabled at all. It is thus important to consider the resulting data utility in addition to performance and privacy guarantees. We have engaged in this context a number of research works particularly focused on location data [Cerf et al., 2017, Primault et al., 2016]. This works could be extended to other types of data (e.g., sensory data in the context of IoT applications).

10.4.3 The shift towards a probabilistic world

AI and probabilistic algorithms are everywhere. In the last couple of years we have witnessed a move towards a probabilistic world (as opposed to a deterministic world). Examples include the success story of Bitcoin with its planet scale probabilistic proof-of-work (PoW) consensus as well as an increasing num-

41. Gartner prevision on IT spending in 2020 : http ://www.gartner.com/newsroom/id/3384720
42. Cisco Global Cloud Index : Forecast and Methodology, 2015-2020 : http ://www.cisco.com/c/dam/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.pdf
43. Biggest attacks of 2016 : http ://www.darkreading.com/cloud/biggest-attacks-of-2016-(so-far)

ber of applications embedding machine learning algorithms. Classical systems where machine learning algorithms are already deployed include spam filtering, malware detection or biometric recognition. More recently with the resurrection of deep learning algorithms, mostly due to the progress of GPU technologies and to the availability of massive learning datasets novel machine-learning applications have emerged [44]. In this context, recent AI based success stories include self-driving public transportation systems, satellite data classification systems, and even systems that help blind and visually disabled people "see". The question that raises to fault-tolerance experts is : are these probabilistic algorithms trustworthy ? said differently, could Byzantine nodes change the behaviour of probabilistic algorithms and cause system failures ? Recent works have shown that the answer to this last question is YES. Indeed, several attacks have shown that by altering their input data, the output of machine learning algorithms can be manipulated [Biggio et al., 2014b, Biggio et al., 2014b, Biggio et al., 2014a]. It is thus important to develop novel mechanisms for enforcing fault-tolerance/fault detection in these systems. An interesting research field to explore in this context is what is called adversarial machine learning [Huang et al., 2011, Barth et al., 2010, Li and Vorobeychik, 2014], where two machine learning algorithms are confronted : while the first one plays the role of a defence system the second one plays the role of the attacker. The result is that the two networks self train each other so as effective protection mechanisms can be found.

10.5 Conclusion

As discussed throughout this manuscript, computing systems are becoming more and more complex (powerful and pervasive computing), more open (pervasive networking) and hence more subject to various types of failures. In addition to this, there is money to be earned from large scale distributed systems (e.g., cloud systems storing sensitive data, bitcoin), which creates harmful vocations (cyber-criminality). The recent attacks on bitcoin, IoT based DoS attacks and the recent ransomware attacks that had major consequences on many businesses and institutions are a snapshot of what is expected to happen in the future. Studies have indeed shown that cyber-criminality has been ever increasing in the last couple of years and is not expected to stop in a near future. In this hostile world a question that can be raised is : what guarantees can we enforce beyond the classical ones offered by existing research algorithms and their implementations (e.g., safety and liveness), which hold only under specific system and network assumptions ? Said differently : are existing systems secure and dependable enough today and will they be secure and dependable enough tomorrow ?

In this very challenging context, a lot needs to be done to provide more efficient, dependable, secure and privacy preserving systems. This will likely occupy my time and the time of a significant part of the research community in the incoming years.

44. AI success stories : https ://www.nvidia.com/en-us/deep-learning-ai/customer-stories/

Bibliographie

[vel,] Velocix. `http://www.velocix.com`. Accessed : 2015-07-26.

[omn, 2013] (2013). The omnet++ simulation environment. `http://www.omnetpp.org/`.

[Abd-El-Malek et al., 2005] Abd-El-Malek, M., Ganger, G. R., Goodson, G. R., Reiter, M. K., and Wylie, J. J. (2005). Fault-scalable Byzantine fault-tolerant services. In *ACM Symposium on Operating Systems Principles* (SOSP'05). ACM.

[Aditya et al., 2012] Aditya, P., Zhao, M., Lin, Y., Haeberlen, A., Druschel, P., Maggs, B., and Wishon, B. (2012). Reliable client accounting for p2p-infrastructure hybrids. *USENIX Symposium on Networked Systems Design and Implementation*, (NSDI'12).

[Aïmeur et al., 2008] Aïmeur, E., Brassard, G., Fernandez, J. M., and Onana, F. S. M. (2008). Alambic : a privacy-preserving recommender system for electronic commerce. *International Journal of Information Security*, 7(5) :307–334.

[Aiyer et al., 2005] Aiyer, A. S., Alvisi, L., Clement, A., Dahlin, M., Martin, J.-P., and Porth, C. (2005). BAR fault tolerance for cooperative services. In *ACM Symposium on Operating Systems Principles*, (SOSP'05).

[Amir et al., 2008] Amir, Y., Coan, B., Kirsch, J., and Lane, J. (2008). Byzantine replication under attack. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, (DSN'08), pages 105–114.

[Amrit et al., 2017] Amrit, K., Lauradoux, C., and Lafourcade, P. (2017). Duck Attack on Accountable Distributed Systems. In *14th EAI International Conference on Mobile and Ubiquitous Systems : Computing, Networking and Services*, Melbourne, Australia.

[Anderson, 2004] Anderson, D. P. (2004). BOINC : A system for public-resource computing and storage. In *Proceedings of IEEE/ACM International Workshop on Grid Computing*.

[Andreas*et al.*,] Andreas*et al.*, H. Accountable virtual machines. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, (OSDI'10).

[Andrés et al., 2013] Andrés, M. E., Bordenabe, N. E., Chatzikokolakis, K., and Palamidessi, C. (2013). Geo-indistinguishability : Differential privacy for location-based systems. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, (CCS'13), pages 901–914. ACM.

[Aringhieri et al., 2006] Aringhieri, R., Damiani, E., Di Vimercati, S., Paraboschi, S., and Samarati, P. (2006). Fuzzy techniques for trust and reputation management in anonymous peer-to-peer systems. *Journal of the American Society for Information Science and Technology*, 57(4) :528–537.

[Aublin et al., 2013] Aublin, P.-L., Mokhtar, S. B., Pace, A., and Quéma, V. (2013). RBFT : Redundant Byzantine Faulte Tolerance. In *33rd International Conference on Distributed Computing Systems*, (ICDCS'13).

[Backes et al., 2009] Backes, M., Druschel, P., Haeberlen, A., and Unruh, D. (2009). A practical and provable technique to make randomized systems accountable. In *Networked and Distributed Systems Security Symposium*, (NDSS'09).

[Bahsoun et al., 2015] Bahsoun, J.-P., Guerraoui, R., and Shoker, A. (2015). Making bft protocols really adaptive. In *IEEE International Parallel and Distributed Processing Symposium*, (IPDPS'15).

[Barth et al., 2010] Barth, A., Rubinstein, B. I., Sundararajan, M., Mitchell, J. C., Song, D., and Bartlett, P. L. (2010). A learning-based approach to reactive security. In *Financial Cryptography*, pages 192–206. Springer.

[Basher et al., 2008] Basher, N., Mahanti, A., Mahanti, A., Williamson, C., and Arlitt, M. (2008). A comparative analysis of web and peer-to-peer traffic. In *the international World Wide Web Symposium*, (WWW'08).

[Basu et al., 2013] Basu, A., Fleming, S., Stanier, J., Naicken, S., Wakeman, I., and Gurbani, V. K. (2013). The state of Peer-to-Peer network simulators. *ACM Computing Surveys (CSUR)*, 45(4).

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

[Behl et al., 2017] Behl, J., Distler, T., and Kapitza, R. (2017). Hybrids on steroids : Sgx-based high performance bft. In *Proceedings of the Twelfth European Conference on Computer Systems* (Eurosys'17), pages 222–237. ACM.

[Behmann and Wu, 2015] Behmann, F. and Wu, K. (2015). *Collaborative internet of things (C-IoT) : For future smart connected life and business*. John Wiley & Sons.

[Belenkiy et al., 2007] Belenkiy, M., Chase, M., Erway, C. C., Jannotti, J., Küpçü, A., Lysyanskaya, A., and Rachlin, E. (2007). Making p2p accountable without losing privacy. In *ACM workshop on Privacy in electronic society*.

[Ben Mokhtar et al., 2013] Ben Mokhtar, S., Berthou, G., Diarra, A., Quḿa, V., and Shoker, A. (2013). RAC : a freerider-resilient, scalable, anonymous communication protocol. In *International Conference on Distributed Computing Systems (ICDCS'13)*.

[Ben Mokhtar et al., 2017] Ben Mokhtar, S., Boutet, A., Felber, P., Pasin, M., Pires, R., and Schiavoni, V. (2017). X-search : Revisiting private web search using intel sgx. In *Proceedings of the ACM/IFIP/USENIX Middleware conference 2017*.

[Ben Mokhtar et al., 2014] Ben Mokhtar, S., Decouchant, J., and Quéma, V. (2014). AcTinG : Accurate freerider tracking in gossip. In *In Proceedings of the 33rd Symposium of Reliable Distributed Systems (SRDS'14)*. IEEE.

[Biely et al., 2013] Biely, M., Delgado, P., Milosevic, Z., and Schiper, A. (2013). Distal : A framework for implementing fault-tolerant distributed algorithms. In *IEEE conference on Dependable Systems and Networks, (DSN'13)*.

[Biggio et al., 2014a] Biggio, B., Fumera, G., and Roli, F. (2014a). Pattern recognition systems under attack : Design issues and research challenges. *International Journal of Pattern Recognition and Artificial Intelligence*, 28(07) :14.

[Biggio et al., 2014b] Biggio, B., Fumera, G., and Roli, F. (2014b). Security evaluation of pattern classifiers under attack. *IEEE transactions on knowledge and data engineering*, 26(4) :984–996.

[Blanchet, 2001] Blanchet, B. (2001). An efficient cryptographic protocol verifier based on prolog rules. In *Proceedings of the 14th IEEE workshop on Computer Security Foundations (CSFW'01)*.

[Blevis, 2017] Blevis, E. (2017). Selfish-driving car. *Interactions*, 24(2) :88.

[Bonald *et al.*,] Bonald *et al.*, T. Epidemic live streaming : optimal performance trade-offs. In *Proceedings of SIGMETRICS'08*.

[Borran and Schiper, 2009] Borran, F. and Schiper, A. (2009). Brief announcement : a leader-free byzantine consensus algorithm. In *the International Symposium on DIStributed Computing (DISC'09)*.

[Bressoud *et al.*, 1996] Bressoud *et al.*, T. (1996). Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, 14(1) :80–107.

[Buchegger and Le Boudec, 2004] Buchegger, S. and Le Boudec, J. (2004). A robust reputation system for mobile ad-hoc networks. *Proceedings of Workshop on the Economics of Networked Systems (P2PEcon'04)*.

[Buttyán and Hubaux, 2001] Buttyán, L. and Hubaux, J.-P. (2001). Rational exchange-a formal model based on game theory. In *Electronic Commerce*. Springer.

[Buttyán and Hubaux, 2003] Buttyán, L. and Hubaux, J.-P. (2003). Stimulating cooperation in self-organizing mobile ad hoc networks. *Mobile Networks and Applications*, 8(5).

[Cagalj et al., 2005] Cagalj, M., Ganeriwal, S., Aad, I., and Hubaux, J.-P. (2005). On selfish behavior in CSMA/CA networks. In *INFOCOM*, volume 4.

[Castro et al., 2003] Castro, M., Druschel, P., Kermarrec, A.-M., Nandi, A., Rowstron, A., and Singh, A. (2003). Splitstream : high-bandwidth multicast in cooperative environments. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 298–313. ACM.

[Castro and Liskov, 1999] Castro, M. and Liskov, B. (1999). Practical byzantine fault tolerance. In *the USENIX Symposium on Operating Systems Design and Implementation (OSDI'99).*

[Catalini and Gans, 2016] Catalini, C. and Gans, J. S. (2016). Some simple economics of the blockchain. Technical report, National Bureau of Economic Research.

[Cerf et al., 2017] Cerf, S., Primault, V., Boutet, A., Ben Mokhtar, S., Birke, R., Chen, L., Bouchenak, S., Marchand, N., and Robu, B. (2017). Achieving privacy and utility trade-off in mobility databases with pulp. In *Proceedings of the 36th IEEE Symposium on Reliable Distributed Systems (IEEE SRDS'17)).*

[Chaum, 1988] Chaum, D. (1988). The dining cryptographers problem : Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1(1).

[Cho et al., 2006] Cho, K., Fukuda, K., Esaki, H., and Kato, A. (2006). The impact and implications of the growth in residential user-to-user traffic. In *ACM SIGCOMM 2006.*

[Chor et al., 1995] Chor, B., Goldreich, O., Kushilevitz, E., and Sudan, M. (1995). Private information retrieval. In *the 36th Annual Symposium on Foundations of Computer Science, 1995*, pages 41–50. IEEE.

[Chun et al., 2007] Chun, B.-G., Maniatis, P., Shenker, S., and Kubiatowicz, J. (2007). Attested append-only memory : Making adversaries stick to their word. *ACM SIGOPS Operating Systems Review*, 41(6) :189–204.

[Clement et al., 2009a] Clement, A., Kapritsos, M., Lee, S., Wang, Y., Alvisi, L., Dahlin, M., and Riche, T. (2009a). Upright cluster services. In *Proceedings of the ACM Symposium on Operating Systems Principles* (SOSP'09), pages 277–290, New York, NY, USA. ACM.

[Clement et al., 2009b] Clement, A., Wong, E., Alvisi, L., Dahlin, M., and Marchetti, M. (2009b). Making byzantine fault tolerant systems tolerate byzantine faults. In *NSDI*, pages 153–168, Berkeley, CA, USA. USENIX Association.

[Cohen, 2003] Cohen, B. (2003). Incentives build robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer systems.*

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

[Cota et al., 2015] Cota, G. L., Mokhtar, S. B., Lawall, J., Muller, G., Gianini, G., Damiani, E., and Brunie, L. (2015). A framework for the design configuration of accountable selfish-resilient peer-to-peer systems. In *IEEE 34th Symposium on Reliable Distributed Systems (SRDS), 2015*, pages 276–285. IEEE.

[Cowling et al., 2006] Cowling, J., Myers, D., Liskov, B., Rodrigues, R., and Shrira, L. (2006). HQ replication : a hybrid quorum protocol for Byzantine fault tolerance. In *the USENIX Symposium on Operating Systems Design and Implementation (OSDI'06).*

[Cox and Noble, 2003] Cox, L. P. and Noble, B. D. (2003). Samsara : Honor among thieves in Peer-to-Peer storage. In *Proceedings of the ACM Symposium on Operating Systems Principles*, (SOSP'03).

[Cunha *et al.*, 2013] Cunha *et al.*, I. (2013). Can peer-to-peer live streaming systems coexist with free riders ? In *IEEE 13th International Conference on Peer-to-Peer Computing,.*

[Czarnecki et al., 2004] Czarnecki, K., Helsen, S., and Eisenecker, U. (2004). Staged configuration using feature models. In *International Conference on Software Product Lines*. Springer.

[Decouchant et al., 2016] Decouchant, J., Ben Mokhtar, S., Petit, A., and Quéma, V. (2016). Pag : Private and accountable gossip. *Proccedings of the 36th IEEE International Conference on Distributed Computing Systems (ICDCS).*

[Demers et al., 1987] Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., and Terry, D. (1987). Epidemic algorithms for replicated database maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, pages 1–12, New York, NY, USA. ACM.

[Diarra et al., 2014] Diarra, A., Mokhtar, S. B., Aublin, P.-L., and Quéma, V. (2014). Fullreview : Practical accountability in presence of selfish nodes. In *Proceedings of the 2014 IEEE 33rd International Symposium on Reliable Distributed Systems*, SRDS '14, pages 271–280, Washington, DC, USA. IEEE Computer Society.

[Dingledine et al., 2004] Dingledine, R., Mathewson, N., and Syverson, P. (2004). Tor : the second-generation onion router. In *Proceedings of USENIX Security Symposium.*

[Distler et al., 2016] Distler, T., Cachin, C., and Kapitza, R. (2016). Resource-efficient byzantine fault tolerance. *IEEE Transactions on Computers*, 65(9) :2807–2819.

[Edman and Yener, 2009] Edman, M. and Yener, B. (2009). On anonymity in an electronic society : A survey of anonymous communication systems. *ACM Computing Surveys*, 42(1).

[Eidenbenz*et al.*, 2011] Eidenbenz*et al.*, R. (2011). Hidden communication in p2p networks steganographic handshake and broadcast. In *Proceedings IEEE of INFOCOM, 2011*, pages 954–962. IEEE.

[Enisa, 2014] Enisa (2014). Algorithms, key size and parameters report. Technical Report.

[et al., 2000] Eytan A. and Bernardo A. H. (2000). Free riding on gnutella. *First Monday Online Journal*, 5(10).

[Eugster and Guerraoui, 2001] Eugster, P. T. and Guerraoui, R. (2001). Hierarchical Probabilistic Multicast. Technical report, EPFL LPD.

[Eugster et al., 2001] Eugster, P. T., Guerraoui, R., Handurukande, S. B., Kouznetsov, P., and Kermarrec, A.-M. (2001). Lightweight probabilistic broadcast. In *DSN '01 : Proceedings of the 2001 International Conference on Dependable Systems and Networks (formerly : FTCS)*, pages 443–452, Washington, DC, USA. IEEE Computer Society.

[Fall, 2003] Fall, K. (2003). A delay-tolerant network architecture for challenged internets. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 27–34. ACM.

[Feldman *et al.*, 2006] Feldman *et al.*, M. (2006). Free-riding and whitewashing in peer-to-peer systems. *Selected Areas in Communications, IEEE Journal on*, 24(5).

[Filman et al., 2004] Filman, R., Elrad, T., Clarke, S., and Akşit, M. (2004). *Aspect-Oriented software development.* Addison-Wesley Professional.

[Freedman and Morris, 2002] Freedman, M. J. and Morris, R. (2002). Tarzan : a peer-to-peer anonymizing network layer. In *Proceedings of the CCS.*

[Freudiger *et al.*, 2009] Freudiger *et al.*, J. (2009). On non-cooperative location privacy : a game-theoretic analysis. In *Proceedings of CCS*. ACM.

[Friedman et al., 2014] Friedman, R., Libov, A., and Vigfusson, Y. (2014). MOLStream : A modular rapid development and evaluation framework for live P2P streaming. In *Proceedings of IEEE ICDCS'14.*

[Gambs et al., 2010] Gambs, S., Killijian, M.-O., and del Prado Cortez, M. N. (2010). Show me how you move and i will tell you who you are. In *Proceedings of the 3rd ACM SIGSPATIAL International Workshop on Security and Privacy in GIS and LBS*, pages 34–41. ACM.

[Ganesh et al., 2001] Ganesh, A. J., Kermarrec, A.-M., and Massoulié, L. (2001). Scamp : Peer-to-peer lightweight membership service for large-scale group communication. In *Networked Group Communication.*

[Ganesh et al., 2002] Ganesh, A. J., Kermarrec, A.-M., and Massoulié, L. (2002). Hiscamp : self-organizing hierarchical membership protocol. In *EW10 : Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 133–139, New York, NY, USA. ACM.

[Garcia et al., 2011] Garcia, R., Rodrigues, R., and Preguiça, N. (2011). Efficient middleware for byzantine fault tolerant database replication. In *Proceedings of EuroSys'11*.

[Gavidia et al., 2007] Gavidia, D., Jesi, G., Gamage, C., and van Steen, M. (2007). Canning spam in wireless gossip networks. In *Fourth Annual Conference on Wireless on Demand Network Systems and Services, 2007. WONS'07*.

[Gerard et al.,] Gerard, J., Cai, H., and Wang, J. Alliatrust : A trustable reputation management scheme for unstructured P2P systems. *Advances in Grid and Pervasive Computing*, pages 115–125.

[Göbel et al., 2016] Göbel, J., Keeler, H. P., Krzesinski, A. E., and Taylor, P. G. (2016). Bitcoin blockchain dynamics : The selfish-mine strategy in the presence of propagation delay. *Performance Evaluation*, 104 :23–41.

[Goel *et al.*, 2003] Goel *et al.*, S. (2003). Herbivore : A scalable and efficient protocol for anonymous communication. tech rep. Technical report, Cornell University.

[Golbeck and Hendler, 2004] Golbeck, J. and Hendler, J. (2004). Reputation network analysis for email filtering. In *Proceedings of the First Conference on Email and Anti-Spam*, volume 44, pages 54–58. Citeseer.

[Goldschlag et al., 1999] Goldschlag, D., Reed, M., and Syverson, P. (1999). Onion routing. *Communications of the ACM*, 42(2) :39–41.

[Golle and Juels, 2004] Golle, P. and Juels, A. (2004). Dining cryptographers revisited. In *Proceedings of EUROCRYPT*.

[Gramaglia et al., 2012] Gramaglia, M., Urueña, M., and Martinez-Yelmo, I. (2012). Off-line incentive mechanism for long-term P2P backup storage. *Computer Communications*.

[Gramoli et al., 2009] Gramoli, V., Vigfusson, Y., Birman, K., Kermarrec, A.-M., and van Renesse, R. (2009). Slicing distributed systems. *IEEE Transactions on Computers*, 58(11) :1444–1455.

[Grolimund et al., 2006] Grolimund, D., Meisser, L., Schmid, S., and Wattenhofer, R. (2006). Havelaar : A robust and efficient reputation system for active peer-to-peer systems. In *1st Workshop on the Economics of Networked Systems (NetEcon), University of Michigan, Ann Arbor, Michigan, USA*.

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

[Guerraoui et al., 2010b] Guerraoui, R., Knežević, N., Quéma, V., and Vukolić, M. (2010b). The next 700 bft protocols. In *Proceedings of EuroSys'10*, pages 363–376.

[Guha et al., 2011] Guha, S., Cheng, B., and Francis, P. (2011). Privad : Practical privacy in online advertising. In *USENIX conference on Networked systems design and implementation*, pages 169–182.

[Guha et al., 2012] Guha, S., Jain, M., and Padmanabhan, V. N. (2012). Koi : A location-privacy platform for smartphone apps. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 14–14. USENIX Association.

[Gustedt et al., 2009] Gustedt, J., Jeannot, E., and Quinson, M. (2009). Experimental validation in large-scale systems : a survey of methodologies. *Parallel Processing Letters*.

[Haeberlen et al., 2009] Haeberlen, A., Avramopoulos, I. C., Rexford, J., and Druschel, P. (2009). Netreview : Detecting when interdomain routing goes wrong. In *NSDI*, pages 437–452.

[Haeberlen et al., 2007a] Haeberlen, A., Kouznetsov, P., and Druschel, P. (2007a). Peerreview : Practical accountability for distributed systems. *ACM SIGOPS Operating Systems Review*, 41(6) :175–188.

[Haeberlen et al., 2007b] Haeberlen, A., Kouznetsov, P., and Druschel, P. (2007b). PeerReview : Practical accountability for distributed systems. In *Symposium on Operating Systems Principles* (SOSP'07).

[Handurukande et al., 2006] Handurukande, S. B., Kermarrec, A.-M., Le Fessant, F., Massoulié, L., and Patarin, S. (2006). Peer sharing behaviour in the eDonkey network, and implications for the design of server-less file sharing systems. In *Proceedings of the ACM SIGOPS EuroSys conference*.

[Haridasan et al., 2008] Haridasan, M., Jansch-Porto, I., and Van Renesse, R. (2008). Enforcing fairness in a live-streaming system. In *Electronic Imaging 2008*.

[Hasan et al., 2013] Hasan, O., Miao, J., Ben Mokhtar, S., and Brunie, L. (2013). A privacy preserving prediction-based routing protocol for mobile delay tolerant networks. In *Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on*, pages 546–553. IEEE.

[Ho et al., 2008] Ho, C., Van Renesse, R., Bickford, M., and Dolev, D. (2008). Nysiad : Practical protocol transformation to tolerate byzantine failures. In *NSDI*, volume 8, pages 175–188.

[Hu et al., 2011] Hu, H., Ahn, G.-J., and Jorgensen, J. (2011). Detecting and resolving privacy conflicts for collaborative data sharing in online social networks. In *ACSAC*.

[Huang et al., 2011] Huang, L., Joseph, A. D., Nelson, B., Rubinstein, B. I., and Tygar, J. (2011). Adversarial machine learning. In *Proceedings of the 4th ACM workshop on Security and artificial intelligence*, pages 43–58. ACM.

[Hughes et al., 2005] Hughes, D., Coulson, G., and Walkerdine, J. (2005). Free riding on Gnutella revisited : the bell tolls ? *IEEE Distributed Systems Online*.

[Hunt et al., 2010] Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. (2010). Zookeeper : wait-free coordination for internet-scale systems. USENIX ATC.

[Jedari et al., 2017] Jedari, B., Liu, L., Qiu, T., Rahim, A., and Xia, F. (2017). A game-theoretic incentive scheme for social-aware routing in selfish mobile social networks. *Future Generation Computer Systems*, 70 :178–190.

[Jelasity and Babaoglu, 2005] Jelasity, M. and Babaoglu, O. (2005). T-Man : Gossip-based overlay topology management. In *Proceedings of Engineering Self-Organising Applications (ESOA'05)*.

[Jelasity et al.,] Jelasity, M., Montresor, A., Jesi, G. P., and Voulgaris, S. The Peersim simulator. `http://peersim.sf.net`.

[Jelasity et al., 2007] Jelasity, M., Voulgaris, S., Guerraoui, R., Kermarrec, A.-M., and Van Steen, M. (2007). Gossip-based peer sampling. *ACM Transactions on Computer Systems (TOCS)*, 25(3) :8.

[Johansen et al., 2006] Johansen, H., Allavena, A., and Van Renesse, R. (2006). Fireflies : scalable support for intrusion-tolerant network overlays. In *ACM SIGOPS OSR*.

[Johansen et al., 2015] Johansen, H. D., Renesse, R. V., Vigfusson, Y., and Johansen, D. (2015). Fireflies : A secure and scalable membership and gossip service. *ACM Transactions on Computer Systems*

[Jusang et al., 2007] Jusang, A., Ismail, R., and Boyd, C. (2007). A survey of trust and reputation systems for online service provision. *Decision Support Systems*, 43(2) :618–644.

[Kamvar et al., 2003] Kamvar, S. D., Schlosser, M. T., and Garcia-Molina, H. (2003). The eigentrust algorithm for reputation management in p2p networks. In *Proceedings of the 12th international conference on World Wide Web*. ACM.

[Kapritsos et al., 2012] Kapritsos, M., Wang, Y., Quema, V., Clement, A., Alvisi, L., Dahlin, M., et al. (2012). All about eve : Execute-verify replication for multi-core servers. In *OSDI*, volume 12, pages 237–250.

[Kaptritsos et al., 2012] Kaptritsos, M., Wang, Y., Quema, V., Clement, A., Alvisi, L., and Dahlin, M. (2012). Eve : Execute-verify replication for multi-core servers. In *OSDI 2012*.

[Karagiannis *et al.*, 2004] Karagiannis *et al.*, T. (2004). Is P2P dying or just hiding ? [P2P traffic measurement]. In *GLOBECOM*, volume 3.

[Kermarrec and Ganesh, 2006] Kermarrec, A.-M. and Ganesh, A. J. (2006). Efficient and adaptive epidemic-style protocols for reliable and scalable multicast. *IEEE Transactions Parallel and Distributed Systems*, 17(7) :593–605.

[Kermarrec et al., 2003] Kermarrec, A.-M., Massoulié, L., and Ganesh, A. J. (2003). Probabilistic reliable dissemination in large-scale systems. *IEEE Transactions on Parallel and Distributed Systems*, 14(3) :248–258.

[Kiczales et al., 2001] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An overview of AspectJ. In *European Conf. on Object-Oriented Programming*. Springer.

[Killian et al., 2007] Killian, C. E., Anderson, J. W., Braud, R., Jhala, R., and Vahdat, A. M. (2007). Mace : language support for building distributed systems. *SIGPLAN Conference on Programming Language Design and Implementation*.

[Koller et al., 1994] Koller, D., Megiddo, N., and von Stengel, B. (1994). Fast algorithms for finding randomized strategies in game trees. In *STOC*.

[Kotla et al., 2009] Kotla, R., Alvisi, L., Dahlin, M., Clement, A., and Wong, E. (2009). Zyzzyva : Speculative byzantine fault tolerance. *ACM Trans. Comput. Syst.*, 27(4) :1–39.

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

[Krishnan et al., 2004] Krishnan, R., Smith, M. D., Tang, Z., and Telang, R. (2004). The impact of free-riding on peer-to-peer networks. In *System Sciences, 2004. Proceedings of the 37th Annual Hawaii International Conference on*, pages 10–pp. IEEE.

[Kwok et al., 2007] Kwok, Y.-K., Hwang, K., and Song, S. (2007). Selfish grids : Game-theoretic modeling and NAS/PSA benchmark evaluation. *IEEE TPDS*.

[Kwon et al., 2017] Kwon, Y., Kim, D., Son, Y., Vasserman, E., and Kim, Y. (2017). Be selfish and avoid dilemmas : Fork after withholding (faw) attacks on bitcoin. *arXiv preprint arXiv :1708.09790*.

[Lamport, 2004] Lamport, L. (2004). Lower bounds for asynchronous consensus.

[Lamport et al., 1982] Lamport, L., Shostak, R. E., and Pease, M. C. (1982). The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*

[Langville and Meyer, 2011] Langville, A. N. and Meyer, C. D. (2011). *Google's PageRank and beyond : The science of search engine rankings*. Princeton University Press.

[Lee et al.,] Lee, U., Choi, M., Cho, J., Sanadidi, M. Y., and Gerla, M. Understanding pollution dynamics in p2p file sharing. In *In Proceedings of the 5th International Workshop on Peer-toPeer Systems (IPTPS'06*.

[Leonini et al., 2009] Leonini, L., Rivière, É., and Felber, P. (2009). SPLAY : Distributed systems evaluation made simple (or how to turn ideas into live systems in a breeze). In *Proc. of NSDI*. USENIX Association.

[Levin et al., 2009] Levin, D., Douceur, J. R., Lorch, J. R., and Moscibroda, T. (2009). Trinc : Small trusted hardware for large distributed systems. In *NSDI*, volume 9, pages 1–14.

[Li and Vorobeychik, 2014] Li, B. and Vorobeychik, Y. (2014). Feature cross-substitution in adversarial classification. In *Advances in neural information processing systems*, pages 2087–2095.

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

[Li et al., 2016] Li, Q., Yang, P., Tang, S., Zhang, M., and Fan, X. (2016). Equilibrium is priceless : selfish task allocation for mobile crowdsourcing network. *EURASIP Journal on Wireless Communications and Networking*, 2016(1) :166.

[Li *et al.*,] Li *et al.*, H. C. Bar gossip. In *Proceedings of OSDI'06*.

[Lian et al., 2007] Lian, Q., Zhang, Z., Yang, M., Zhao, B. Y., Dai, Y., and Li, X. (2007). An empirical study of collusion behavior in the maze p2p file-sharing system. In *27th International Conference on Distributed Computing Systems, 2007. ICDCS'07.*, pages 56–56. IEEE.

[Lin and Marzullo, 1999] Lin, M. J. and Marzullo, K. (1999). Directional gossip : Gossip in a wide area network. In *European Dependable Computing Conference*.

[Liu et al., 2016] Liu, S., Viotti, P., Cachin, C., Quéma, V., and Vukolic, M. (2016). Xft : Practical fault tolerance beyond crashes. In *OSDI*, pages 485–500.

[Locher *et al.*, 2006] Locher *et al.*, T. (2006). Free riding in bittorrent is cheap. In *Proc. Workshop on Hot Topics in Networks (HotNets)*, pages 85–90. Citeseer.

[MacCormick et al., 2004] MacCormick, J., Murphy, N., Najork, M., Thekkath, C. A., and Zhou, L. (2004). Boxwood : abstractions as the foundation for storage infrastructure. OSDI, pages 8–8.

[Mahajan et al., 2004] Mahajan, R., Rodrig, M., Wetherall, D., and Zahorjan, J. (2004). Experiences applying game theory to system design. In *Proc. of the ACM SIGCOMM workshop on Practice and Theory of Incentives in Networked Systems.*

[Malkhi et al., 2001] Malkhi, D., Reiter, M. K., Rodeh, O., and Sella, Y. (2001). Efficient update diffusion in byzantine environments. In *SRDS*, pages 90–98.

[Maouche et al., 2017] Maouche, M., Ben Mokhtar, S., and Bouchenak, S. (2017). Ap-attack : A novel re-identification attack on mobility datasets. In *Proceedings of the 14th International Conference on Mobile and Ubiquitous Systems : Computing, Networking and Service (Mobiquitous'17).*

[Marti and Garcia-Molina, 2006] Marti, S. and Garcia-Molina, H. (2006). Taxonomy of trust : Categorizing p2p reputation systems. *Computer Networks*, 50(4).

[Mashhadi et al., 2009] Mashhadi, A. J., Ben Mokhtar, S., and Capra, L. (2009). Habit : Leveraging human mobility and social network for efficient content dissemination in delay tolerant networks. In *World of Wireless, Mobile and Multimedia Networks & Workshops, 2009. WoWMoM 2009. IEEE International Symposium on a*, pages 1–6. IEEE.

[Mashhadi et al., 2012] Mashhadi, A. J., Mokhtar, S. B., and Capra, L. (2012). Fair content dissemination in participatory DTNs. *Ad Hoc Networks*, 10(8) :1633–1645.

[Mei and Stefa, 2012] Mei, A. and Stefa, J. (2012). Give2Get : Forwarding in social mobile wireless networks of selfish individuals. *IEEE TDSC.*

[Member-Zhou and Fellow-Hwang, 2007] Member-Zhou, R. and Fellow-Hwang, K. (2007). Powertrust : A robust and scalable reputation system for trusted peer-to-peer computing. *IEEE Trans. Parallel Distrib. Syst.*, 18(4) :460–473.

[Meyer and Whateley, 2004] Meyer, T. and Whateley, B. (2004). SpamBayes : Effective open-source, Bayesian based, email classification system. In *Proceedings of the First Conference on Email and Anti-Spam (CEAS)*, volume 98.

[Miao et al., 2011] Miao, J., Hasan, O., Mokhtar, S. B., and Brunie, L. (2011). An adaptive routing algorithm for mobile delay tolerant networks. In *14th International Symposium on Wireless Personal Multimedia Communications, WPMC 2011, Brest, France, October 3-7, 2011*, pages 1–5.

[Miao et al., 2015] Miao, J., Hasan, O., Mokhtar, S. B., Brunie, L., and Gianini, G. (2015). A delay and cost balancing protocol for message routing in mobile delay tolerant networks. *Ad Hoc Networks*, 25 :430–443.

[Miao et al., 2016] Miao, J., Hasan, O., Mokhtar, S. B., Brunie, L., and Hasan, A. (2016). 4pr : Privacy preserving routing in mobile delay tolerant networks. *Computer Networks*, 111 :17–28.

[Miao et al., 2012] Miao, J., Hasan, O., Mokhtar, S. B., Brunie, L., and Yim, K. (2012). An analysis of strategies for preventing selfish behavior in mobile delay tolerant networks. In *Sixth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing, IMIS 2012, Palermo, Italy, July 4-6, 2012*, pages 208–215.

[Miao et al., 2013] Miao, J., Hasan, O., Mokhtar, S. B., Brunie, L., and Yim, K. (2013). An investigation on the unwillingness of nodes to participate in mobile delay tolerant network routing. *International Journal of Information Management*, 33(2) :252–262.

[Miranda and Rodrigues, 2003] Miranda, H. and Rodrigues, L. (2003). Friends and foes : Preventing selfishness in open mobile ad hoc networks. In *Proc. of ICDCS Workshop on Mobile Distributed Computing.* IEEE.

[Mislove et al., 2008] Mislove, A., Post, A., Druschel, P., and Gummadi, P. K. (2008). Ostra : Leveraging trust to thwart unwanted communication. In *NSDI.*

[Mitchell and Teague, 2003] Mitchell, J. C. and Teague, V. (2003). Autonomous nodes and distributed mechanisms. In *Software Security—Theories and Systems*. Springer.

[Mokhtar et al., 2010] Mokhtar, S. B., Pace, A., and Quema, V. (2010). Firespam : Spam resilient gossiping in the bar model. In *29th IEEE Symposium on Reliable Distributed Systems, 2010*, pages 225–234. IEEE.

[Mol et al., 2008] Mol, J. J.-D., Pouwelse, J. A., Meulpolder, M., Epema, D. H., and Sips, H. J. (2008). Give-to-get : free-riding resilient video-on-demand in p2p systems. In *Electronic Imaging 2008*, pages 681804–681804. International Society for Optics and Photonics.

[Montresor and Jelasity, 2009] Montresor, A. and Jelasity, M. (2009). PeerSim : A scalable P2P simulator. In *IEEE Int. Conf. on Peer-to-Peer Computing*. IEEE.

[Mousa et al., 2017a] Mousa, H., Ben Mokhtar, S., Omar Hasan, L. B., Younes, O., and Hadhoud, M. (2017a). Privasense : Privacy-preserving and reputation-aware mobile participatory sensing. In *Proceedings of the 14th International Conference on Mobile and Ubiquitous Systems : Computing, Networking and Service (Mobiquitous'17)*.

[Mousa et al., 2017b] Mousa, H., Mokhtar, S. B., Hasan, O., Brunie, L., Youness, O. S., and Mohiy, H. (2017.b). A reputation system resilient against colluding and malicious adversaries in mobile participatory sensing applications. In *The 14th Annual IEEE Consumer Communications & Networking Conference (CCNC 2017)*.

[Mousa et al., 2015] Mousa, H., Mokhtar, S. B., Hasan, O., Younes, O., Hadhoud, M., and Brunie, L. (2015). Trust management and reputation systems in mobile participatory sensing applications : A survey. *Computer Networks*, 90 :49–73.

[Myerson, 2013] Myerson, R. B. (2013). *Game Theory*. Harvard university press.

[Nash, 1951] Nash, J. (1951). Non-Cooperative Games. *Annals of Mathematics*, 54(2).

[Ngan et al., 2010] Ngan, T.-W., Dingledine, R., and Wallach, D. S. (2010). Building incentives into Tor. In *Int. Conf. on Financial Cryptography and Data Security*. Springer.

[Osborne *et al.*, 1994] Osborne *et al.*, M. J. (1994). *A course in game theory*. MIT press.

[Papadimitriou et al., 2013] Papadimitriou, A., Zhao, M., and Haeberlen, A. (2013). Towards privacy-preserving fault detection. In *Hot Topics in Dependable Systems*.

[Parr, 2013] Parr, T. (2013). *The definitive ANTLR 4 reference*. Pragmatic Bookshelf.

[Patrick *et al.*, 2004] Patrick *et al.*, E. (2004). Epidemic information dissemination in distributed systems. *IEEE Computer*, 37(5).

[Petit et al., 2015] Petit, A., Cerqueus, T., Ben Mokhtar, S., Brunie, L., and Kosch, H. (2015). Peas : Private, efficient and accurate web search. In *14th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, Helsinki, Finland.

[Petit et al., 2016] Petit, A., Cerqueus, T., Boutet, A., Mokhtar, S. B., Coquil, D., Brunie, L., and Kosch, H. (2016). Simattack : private web search under fire. *Journal of Internet Services and Applications*, 7(1) :2.

[Pfitzmann and Hansen, 2008] Pfitzmann, A. and Hansen, M. (2008). Anonymity, unlinkability, undetectability, unobservability, pseudonymity, and identity management – a consolidated proposal for terminology.

[Piatek et al., 2010] Piatek, M., Krishnamurthy, A., Venkataramani, A., Yang, Y. R., Zhang, D., and Jaffe, A. (2010). Contracts : Practical contribution incentives for P2P live streaming. In *Proc. of NSDI*. USENIX Association.

[Pires et al., 2016] Pires, R., Pasin, M., Felber, P., and Fetzer, C. (2016). Secure content-based routing using intel software guard extensions. In *Proceedings of the 17th International Middleware Conference*, page 10. ACM.

[Plissonneau *et al.*, 2005] Plissonneau *et al.*, L. (2005). Analysis of peer-to-peer traffic on adsl.

[Pries et al., 2008] Pries, R., Yu, W., Fu, X., and Zhao, W. (2008). A new replay attack against anonymous communication networks. In *Proceedings of ICC*.

[Primault et al., 2016] Primault, V., Boutet, A., Mokhtar, S. B., and Brunie, L. (2016). Adaptive location privacy with alp. In *Reliable Distributed Systems (SRDS), 2016 IEEE 35th Symposium on*, pages 269–278. IEEE.

[Primault et al., 2015] Primault, V., Mokhtar, S. B., Lauradoux, C., and Brunie, L. (2015). Time distortion anonymization for the publication of mobility data with high utility. In *Trustcom/BigData-SE/ISPA, 2015 IEEE*, volume 1, pages 539–546. IEEE.

[Rahman et al., 2011] Rahman, R., Vinkó, T., Hales, D., Pouwelse, J., and Sips, H. (2011). Design space analysis for modeling incentives in distributed systems. In *ACM SIGCOMM Computer Communication Review*.

[Raymond, 2001] Raymond, J.-F. (2001). Traffic analysis : Protocols, attacks, design issues, and open problems. In *Designing Privacy Enhancing Technologies*, volume 2009. Springer.

[Reiter and Rubin, 1998] Reiter, M. K. and Rubin, A. D. (1998). Crowds : anonymity for web transactions. *ACM Trans. Inf. Syst. Secur.*, 1(1).

[Royer and Toh, 1999] Royer, E. M. and Toh, C.-K. (1999). A review of current routing protocols for ad hoc mobile wireless networks. *IEEE personal communications*, 6(2) :46–55.

[Sapirshtein et al., 2016] Sapirshtein, A., Sompolinsky, Y., and Zohar, A. (2016). Optimal selfish mining strategies in bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 515–532. Springer.

[Schroeder et al., 2006] Schroeder, B., Wierman, A., and Harchol-Balter, M. (2006). Open versus closed : a cautionary tale. NSDI, pages 18–18, Berkeley, CA, USA. USENIX Association.

[Schuster et al., 2015] Schuster, F., Costa, M., Fournet, C., Gkantsidis, C., Peinado, M., Mainar-Ruiz, G., and Russinovich, M. (2015). Vc3 : Trustworthy data analytics in the cloud using sgx. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 38–54. IEEE.

[Serjantov et al., 2003] Serjantov, A., Dingledine, R., and Syverson, P. (2003). From a trickle to a flood : Active attacks on several mix types. In Petitcolas, F., editor, *Information Hiding*, volume 2578 of *Lecture Notes in Computer Science*. Springer.

[Shih et al., 2017] Shih, M.-W., Lee, S., Kim, T., and Peinado, M. (2017). T-sgx : Eradicating controlled-channel attacks against enclave programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA.*

[Sirivianos et al., 2007] Sirivianos, M., Park, J. H., Yang, X., and Jarecki, S. (2007). Dandelion : Cooperative content distribution with robust incentives. In *USENIX Annual Technical Conference.*

[Song et al., 2005] Song, S., Hwang, K., Zhou, R., and Kwok, Y. (2005). Trusted P2P transactions with fuzzy reputation aggregation. *IEEE Internet Computing*, 9(6) :24–34.

[Sousa et al., 2010] Sousa, P., Bessani, A. N., Correia, M., Neves, N. F., and Verissimo, P. (2010). Highly available intrusion-tolerant services with proactive-reactive recovery. *IEEE Trans. Parallel Distrib. Syst.*, 21(4) :452–465.

[Spear et al., 2009] Spear, M., Lu, X., Matloff, N., and Wu, S. F. (2009). Karmanet : Leveraging trust ed social path to create judicious forwarders. Technical report, UCDAVIS.

[Srivastava et al., 2005] Srivastava, V., Neel, J. O., MacKenzie, A. B., Menon, R., DaSilva, L. A., Hicks, J. E., Reed, J. H., and Gilles, R. P. (2005). Using game theory to analyze wireless ad hoc networks. *IEEE Communications Surveys and Tutorials*, 7(1-4).

[Tang and Cai,] Tang, C. and Cai, C. Verifiable mobile online social network privacy-preserving location sharing scheme. *Concurrency and Computation : Practice and Experience.*

[Urbán et al., 2001] Urbán, P., Défago, X., and Schiper, A. (2001). Neko : A single environment to simulate and prototype distributed algorithms. In *ICOIN.*

[van Renesse and Birman, 2002] van Renesse, R. and Birman, K. (2002). Scalable management and data mining using astrolabe.

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

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

[Vilaça et al., 2011] Vilaça, X., Leitao, J., Correia, M., and Rodrigues, L. (2011). N-party bar transfer. In *Principles of Distributed Systems*, pages 392–408. Springer.

[Vilaça et al., 2017] Vilaça, X., Rodrigues, L., Silva, J., and Miranda, H. (2017). Fastrank : Practical lightweight tolerance to rational behaviour in edge assisted streaming. In *Proceedings of the 18th International Conference on Distributed Computing and Networking*, page 21. ACM.

[Voulgaris and van Steen, 2005] Voulgaris, S. and van Steen, M. (2005). Epidemic-style management of semantic overlays for content-based searching.

[Vu et al., 2010] Vu, L., Gupta, I., Nahrstedt, K., and Liang, J. (2010). Understanding overlay characteristics of a large-scale peer-to-peer iptv system. *TOMCCAP*.

[Waidner and Pfitzmann, 1989] Waidner, M. and Pfitzmann, B. (1989). The dining cryptographers in the disco : Unconditional sender and recipient untraceability with computationally secure serviceability. In *Proceedings of EUROCRYPT*.

[Wang and Vassileva, 2003] Wang, Y. and Vassileva, J. (2003). Trust and reputation model in peer-to-peer networks. In *Proceedings of the 3rd International Conference on Peer-to-Peer Computing*, page 150. Citeseer.

[Wolinsky et al., 2012] Wolinsky, D. I., Corrigan-Gibbs, H., and Ford, B. (2012). Dissent in numbers : Making strong anonymity scale. In *Proceedings of OSDI*.

[Wood et al., 2011] Wood, T., Singh, R., Venkataramani, A., Shenoy, P., and Cecchet, E. (2011). Zz and the art of practical bft execution. EuroSys.

[Xiong and Liu, 2004] Xiong, L. and Liu, L. (2004). Peertrust : Supporting reputation-based trust for peer-to-peer electronic communities. *IEEE Trans. Knowl. Data Eng.*

[Yahyavi et al., 2013] Yahyavi, A., Huguenin, K., Gascon-Samson, J., Kienzle, J., Kemme, B., et al. (2013). Watchmen : Scalable cheat-resistant support for distributed multi-player online games. In *Proceedings of the 33rd International Conference on Distributed Computing Systems (ICDCS)*, pages 1–10.

[Yin et al., 2009] Yin, H., Liu, X., Zhan, T., Sekar, V., Qiu, F., Lin, C., Zhang, H., and Li, B. (2009). Design and deployment of a hybrid cdn-p2p system for live video streaming : experiences with livesky. In *ACM Multimedia*.

[Yoo and Agrawal, 2006] Yoo, Y. and Agrawal, D. P. (2006). Why does it pay to be selfish in a MANET ? *IEEE Wireless Communications*.

[Yumerefendi and Chase, 2007] Yumerefendi, A. R. and Chase, J. S. (2007). Strong accountability for network storage. *ACM Transactions on Storage (TOS)*, 3(3) :11.

[Zhang and Preneel, 2017] Zhang, R. and Preneel, B. (2017). Publish or perish : A backward-compatible defense against selfish mining in bitcoin. In *Cryptographers' Track at the RSA Conference*, pages 277–292. Springer.

[Zhao et al., 2012] Zhao, M., Zhou, W., Gurney, A. J., Haeberlen, A., Sherr, M., and Loo, B. T. (2012). Private and verifiable interdomain routing decisions. In *SIGCOMM*.

[Zheleva and Getoor, 2009] Zheleva, E. and Getoor, L. (2009). To join or not to join : the illusion of privacy in social networks with mixed public and private user profiles. In *WWW*.

[Zhong *et al.*, 2003] Zhong *et al.*, S. (2003). Sprite : A simple, cheat-proof, credit-based system for mobile ad-hoc networks. In *INFOCOM*. IEEE.

[Zhu et al., 2016] Zhu, H., Lu, R., Huang, C., Chen, L., and Li, H. (2016). An efficient privacy-preserving location-based services query scheme in outsourced cloud. *IEEE Transactions on Vehicular Technology*, 65(9) :7729–7739.

[Zhuang et al., 2005] Zhuang, L., Zhou, F., Zhao, B. Y., and Rowstron, A. (2005). Cashmere : resilient anonymous routing. In *Proceedings of NSDI*.