HAL
open science

Collusions and Privacy in Rational-Resilient Gossip

Jérémie Decouchant

To cite this version:

Jérémie Decouchant. Collusions and Privacy in Rational-Resilient Gossip. Cryptography and Security [cs.CR]. Université Grenoble Alpes, 2015. English. NNT : 2015GREAM034 . tel-01259242

HAL Id: tel-01259242

https://theses.hal.science/tel-01259242v1

Submitted on 20 Jan 2016

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

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

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Jérémie Decouchant

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

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

Collusions and Privacy in Rational-Resilient Gossip

Thèse soutenue publiquement le **9 novembre 2015**,
devant le jury composé de :

Prof. Noël de Palma

Université de Grenoble, Président

Dr Laurent Réveillère

Enseirb-Matmeca, Rapporteur

Prof. Sébastien Tixeuil

Université Pierre et Marie Curie, Rapporteur

Dr Étienne Rivière

Université de Neuchâtel, Examinateur

Prof. Vivien Quéma

INP Grenoble, Directeur de thèse

Dr Sonia Ben Mokhtar

CNRS LIRIS, Co-Encadrante de thèse

Abstract

Gossip-based content dissemination protocols are a scalable and cheap alternative to centralized 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. In addition, previous rational-resilient gossip-based solutions require nodes to log their interactions with others, and disclose the content of their logs, which may disclose sensitive information. Nowadays, a consensus exists on the necessity of reinforcing the control of users on their personal information. Nonetheless, to the best of our knowledge no privacy-preserving rational-resilient gossip-based content dissemination system exists.

The contributions of this thesis are twofold.
First, we present AcTinG, a protocol that prevents rational collusions in gossip-based content dissemination protocols, while guaranteeing zero false positive accusations. AcTing makes nodes maintain secure logs and mutually check each others' correctness thanks to verifiable but non predictable audits. As a consequence of its design, it is shown to be a Nash-equilibrium. A performance evaluation shows that AcTinG is able to deliver all messages despite the presence of colluders, and exhibits similar scalability properties as standard gossip-based dissemination protocols.

Second, we describe PAG, the first accountable and privacy-preserving gossip protocol. PAG builds on a monitoring infrastructure, and homomorphic cryptographic procedures to provide privacy to nodes while making sure that nodes forward the content they receive. The theoretical evaluation of PAG shows that breaking the privacy of interactions is difficult, even in presence of a global and active opponent. We assess this protocol both in terms of privacy and performance using a deployment performed on a cluster of machines, simulations involving up to a million of nodes, and theoretical proofs. The bandwidth overhead is much lower than existing anonymous communication protocols, while still being practical in terms of CPU usage.

Keywords. Gossip, selfish nodes, collusions, accountability, privacy, homomorphic encryption.

Résumé

Les protocoles de dissémination de contenus randomisés sont une alternative bon marché et pouvant monter en charge aux systèmes centralisés. Cependant, il est bien connu que ces protocoles souffrent en présence de comportements individualistes, i.e., de participants qui cherchent à recevoir un contenu sans contribuer en retour à sa propagation. Alors que le problème des participants égoïstes a été bien étudié dans la littérature, les coalitions de participants égoïstes ont été laissés de côté. De plus, les manières actuelles permettant de limiter ou tolérer ces comportements exigent des noeuds qu'ils enregistrent leurs interactions, et rendent public leur contenu, ce qui peut dévoiler des informations gênantes. De nos jours, il y a consensus autour du besoin de renforcer les possibilités de contrôle des usagers de systèmes informatiques sur leurs données personnelles. Cependant, en l'état de nos connaissances, il n'existe pas de protocole qui évite de divulguer des informations personnelles sur les utilisateurs tout en limitant l'impact des comportements individualistes.

Cette thèse apporte deux contributions. Tout d'abord, nous présentons AcTinG, un protocole qui empêche les coalitions de noeuds individualistes dans les systèmes pair-à-pair de dissémination de contenus, tout en garantissant une absence de faux-positifs dans le processus de détection de fautes. Les utilisateurs de AcTinG enregistrent leurs interactions dans des enregistrements sécurisés, et se vérifient les uns les autres grâce à une procédure d'audit non prédictible, mais vérifiable *a posteriori*. Ce protocole est un équilibre de Nash par construction. Une évaluation de performance montre qu'AcTinG est capable de fournir les messages à tous les noeuds malgré la présence de coalitions, et présente des propriétés de passage à l'échelle similaires aux protocoles classiques de dissémination aléatoire.

Ensuite, nous décrivons PAG, le premier protocole qui évite de dévoiler des informations sur les usagers tout en les contrôlant afin d'éviter les comportements égoïstes. PAG se base sur une architecture de surveillance, formée par les participants, ainsi que des procédures de chiffrement homomorphiques. L'évaluation théorique de ce protocole montre qu'obtenir le détail des interactions des noeuds est difficile, même en cas d'attaques collectives. Nous évaluons ce protocole en terme de protection de l'intimité des interactions et en terme de performance en utilisant un déploiement effectué sur un cluster de machines, ainsi que des simulations qui impliquent jusqu'à un million de participants, et enfin en utilisant des preuves théoriques. Ce protocole a un surcoût en

bande-passante inférieur aux protocoles de communications anonymes existants, et est raisonnable en terme de coût cryptographique.

Mots-clés. Gossip, utilisateurs égoïstes, coalitions, responsabilité, protection vie privée, chiffrement homomorphique.

Preface

This thesis presents the research conducted in the Erods team of the LIG (Laboratoire d'Informatique of Grenoble) to pursue the Ph.D. in the specialty "Informatics" from the Doctoral School "Mathématiques, Sciences et Technologies de l'Information, Informatique" of the University of Grenoble. The research activities have been carried out under the supervision of Prof. Vivien Quéma (LIG/Grenoble INP) and Dr. Sonia Ben Mokhtar (LIRIS/CNRS).

This thesis focuses on two problems faced by content dissemination protocols in peer-to-peer networks: (i) the presence of selfish nodes, following either individual or collective strategies, and (ii) the protection of the privacy of users which is endangered by the mechanisms traditionally used to detect selfish nodes.

While the second part of this thesis is currently under submission, novel contributions to tackle problem (i) have been published in an international conference:

- **AcTinG: Accurate Freerider Tracking in Gossip**. Sonia Ben Mokhtar, Jérémie Decouchant, Vivien Quéma. *In Proceedings of the International Symposium on Reliable Distributed Systems (SRDS), Nara, Japan, October 2014* [5]

During these three years, I also had the occasion to do research on a problem, concerning the management of memory on multicore architectures. We will not detail this thematic in this document. This work led to a publication in an international conference:

- **Large Pages May Be Harmful on NUMA Systems**. Fabien Gaud, Baptiste Lepers, Jérémie Decouchant, Justin R. Funston, Alexandra Fedorova, Vivien Quéma *In Proceedings of the USENIX Annual Technical Conference 2014, Philadelphia, USA, June 2014*

Acknowledgments

Je tiens à adresser mes sincères remerciements à :

M. Vivien Quéma, mon directeur de thèse, pour avoir guidé mes travaux, et m'avoir donné la liberté d'explorer différents thèmes de recherche.

Mme Sonia Ben Mokhtar, ma co-directrice de thèse, pour ses conseils, et le travail de rédaction qu'elle a fourni sur nos publications. J'ai particulièrement apprécié ses retours sur les problèmes pointus que j'ai rencontré. Ses retours ont très souvent permis d'ébaucher une nouvelle solution.

M. Noël de Palma, pour m'avoir fait l'honneur de présider le jury de cette thèse, et pour l'intérêt qu'il a bien voulu y porter.

M. Laurent Réveillère, et M. Sébastien Tixeuil, pour le temps et les efforts qu'ils ont consacrés à la révision de ce document et pour l'honneur qu'ils m'ont fait en participant au jury d'évaluation de ma thèse.

Baptiste Lepers, Pierre-Louis Aublin, Gautier Berthou, et Amadou Diarra, avec qui j'ai partagé mon bureau durant mes deux premières années de thèse, pour avoir partagé leurs connaissances avec moi, et pour les bons moments passés ensemble.

Les membres de l'équipe Erods pour la bonne humeur qu'ils ont su maintenir dans l'équipe, et pour nos relations amicales.

Mon papa, Dominique, ainsi que sa compagne, Sonia, pour m'avoir suivi et soutenu pendant ma thèse, et pour la révision patiente et détaillée de ce document. Je remercie aussi bien sûr ma maman, Véronique, ma soeur Delphine et Florian, mon frère Raphaël et Audrey pour le soutien qu'ils m'ont témoigné et pour leur présence.

La famille de Leslie qui m'a rendu tant de services, et depuis bien avant la période de ma thèse. Pour ne citer que les plus récents et anecdotiques, j'ai apprécié les colis, les petits toasts, les orangeades et les retouches de vêtements avec Michel et Yvonne, l'hospitalité en période de canicule et les vacances avec Valérie, Yvon et Fiona. Merci à tous pour votre gentillesse en toutes circonstances.

Finalement, j'aimerais remercier Leslie, qui m'a toujours soutenu pendant ces trois années. Je lui suis très reconnaissant d'avoir été présente à mes côtés, et d'avoir rendu ma vie plus douce.

Terminology

Term used	Concept	Alternative terms
User	The human who uses an application.	
Node	The virtual representation of a user in a P2P system.	
Information dissemination	The process of serving every node in an audience with a given information. In a P2P context nodes forward to each other this information.	Content dissemination
Gossip protocol	A P2P information dissemination protocol that relies on random exchanges between nodes.	
Selfish node	A peer that tries to minimize its participation (e.g., upload bandwidth, CPU usage) in a protocol while maximizing its benefit (e.g., delay to receive a content).	Rational node, freerider, opportunistic node
Byzantine node	A node deviating from the protocol in an arbitrary or malicious way.	Malicious node
Selfish-resilient protocol	An information dissemination protocol that can efficiently serve a content to an audience of nodes in presence of a given proportion of selfish nodes.	Rational-resilient protocol
Collusion-resilient protocol	An information dissemination protocol that can efficiently serve a content to an audience in presence of selfish nodes that collude to improve their benefit, or avoid being detected.	
Source	In a content-dissemination system, the node that generates the content to be disseminated.	Broadcaster
Update	A chunk of content that a source releases, and that peers want to receive.	Chunk, Message
Membership	The set of nodes that participates in a protocol at a given moment.	
Session	Dissemination of one or several contents among a membership.	
Churn	Nodes arrival and departure, possibly completely desynchronized. Maintaining the membership in presence of churn is challenging.	
Round	Interval of time between two successive emissions of updates by the source during which nodes have to complete their exchanges.	Gossip period

Table I – Terminology used in this document.

Contents

	Page
Abstract	**i**
Preface	**v**
Acknowledgments	**vii**
Terminology	**ix**
Contents	**xi**
List of figures	**xv**
List of tables	**xvii**
Introduction	**1**
Scientific context	1
Gossip, coalitions of freeriders and privacy	3
Objectives of this research work	4
Research location	5
Organization of this document	5
CHAPTER 1 Gossip with colluding rational nodes	**7**
1.1. Probabilistic dissemination	8
1.1.1. Structured and unstructured overlays	8
1.1.2. Principles of gossip	9
1.1.3. Probabilistic guarantees	9
1.2. Selfish behaviors	11
1.2.1. BAR model	11
1.2.2. Utility functions	12
1.2.3. Byzantine fault tolerance or Nash equilibrium	12
1.3. Impact of selfish nodes	13
1.3.1. Individual deviations	13
1.3.2. Collusions	14

1.3.3. Privacy and selfish nodes 14

Part I Rational collusions in gossip-based dissemination systems 15

CHAPTER 2 Gossip in presence of rational nodes 17

2.1. Rational resilient dissemination protocols 18
2.1.1. Nash equilibriums 18
2.1.2. Audits and statistical approaches 23
2.2. Accountability techniques 30
2.2.1. Software-only accountability 30
2.2.2. Hardware-assisted accountability 33
2.3. BAR-transformation protocols 37
2.4. Summary 39
2.4.1. Requirements 40
2.4.2. Drawbacks of existing solutions 41
2.4.3. Conclusion 43

CHAPTER 3 AcTinG: accurate freerider detection in gossip 45

3.1. Introduction to $AcTinG$ 46
3.1.1. Principal ideas 46
3.1.2. System model 47
3.1.3. Protocol overview 48
3.2. $AcTinG$ 50
3.2.1. Protocol details 51
3.2.2. Membership protocol 51
3.2.3. Partnership management 53
3.2.4. Audit protocol 54
3.2.5. Update exchanges 56
3.3. Proofs 56
3.3.1. Risk versus gain analysis 56
3.3.2. Resilience to (colluding) rational nodes 58
3.4. Evaluation 64
3.4.1. Methodology and parameters setting 65
3.4.2. Impact of colluders 66
3.4.3. Bandwidth consumption 68
3.4.4. Resilience to massive node departure 68
3.4.5. Scalability 69
3.5. Conclusion 71

Part II Privacy and rational resiliency in gossip-based dissemination systems 73

CHAPTER 4 Privacy in rational-resilient gossip 75

4.1. Principles of gossip and selfish behaviours 76

4.1.1. Selfish behaviours 77
4.1.2. Requirements against selfish behaviours 79
4.1.3. Accountability solutions 79
4.1.4. Privacy requirements 79
4.2. Rational-resilient gossip protocols 80
4.2.1. Rational resiliency by design 80
4.2.2. Audit-based approaches 81
4.2.3. Virtual currency approach. 82
4.3. Anonymous communication protocols 83
4.3.1. Altruistic relaying 83
4.3.2. Rational resilient relaying 85
4.4. Accountable and privacy preserving approaches 86
4.4.1. Zero-knowledge proofs 87
4.4.2. Collaborative verification protocols 87
4.5. Preserving privacy in other contexts 88
4.5.1. Peer-to-peer protocols 88
4.6. Summary 90
4.6.1. Requirements 91
4.6.2. Summary of existing solutions 91
4.6.3. Conclusion 93

CHAPTER 5 PAG: Private and accountable gossip 95

5.1. System model 97
5.1.1. Communications and cryptographic assumptions 97
5.1.2. Gossip sessions 97
5.1.3. Nodes behaviors 98
5.1.4. Adversary model 98
5.2. PAG Overview 98
5.2.1. Global membership and monitoring 98
5.2.2. Exchanges of updates using gossip 99
5.2.3. Enforcing accountability using monitoring 100
5.2.4. Enforcing privacy using homomorphic encryptions 101
5.3. Design of PAG 103
5.3.1. Forwarding updates 103
5.3.2. Encrypting a set of updates 104
5.3.3. Combining all encryptions 106
5.3.4. Practical implementation details. 106
5.4. Security, privacy and accountability 108
5.4.1. $k - PAG$: clustering sessions 108
5.4.2. Enforcing $\mathbf{P_3}$ under global and active attacks 109
5.4.3. Accountability against selfish deviations 110
5.5. Performance evaluation 113
5.5.1. Methodology and Parameter Setting 113
5.5.2. Probabilistic study of the impact of coalitions 114
5.5.3. Comparison with an accountable gossip protocol and impact of the number of contents 115
5.5.4. Comparison with anonymous communication systems 116

5.5.5. Impact of updates size 119
5.5.6. Cryptographic costs 119
5.5.7. Scalability 120
5.6. Conclusion 120

Conclusion **123**

References **127**

Appendix **133**
.1. ProVerif code of PAG 133
.2. Probabilistic evaluation of PAG resiliency to collusions 138

List of Figures

1. Distributed systems architecture paradigms 2

1.1. Example of gossip with 7 nodes 10
1.2. Example of gossip with 7 nodes in presence of a rational node 13

2.1. Basic trade illustration of BAR Gossip (*Based on* [22]). 21
2.2. Global auditing architecture (*Based on* [24]). 25
2.3. Example of the architecture of Fireflies (*Based on* [25]). 26
2.4. Cross-checking protocol in LiFTinG (*Based on* [26]). 29
2.5. Illustration of a secure log (*Based on* [29]). 31

3.1. Overview of $AcTinG$. 48
3.2. Arrival of a new node. 51
3.3. Handling of an omission failure. 52
3.4. Handling of a node departure. 53
3.5. Establishment of new associations between nodes, which may imply audits. 54
3.6. Update exchanges between nodes. 56
3.7. Pseudocode of the program that is used to estimate the number of time a colluding node avoids to send an update. 58
3.8. Proportion of missed updates by correct nodes when a given proportion of the audience collude as a single group. 67
3.9. Proportion of missed updates by correct nodes when 30% of the audience is rational, and collude in independent groups of equal sizes. . . 67
3.10. Fault-free case: Cumulative distribution of average bandwidths. . . . 68
3.11. Nodes average bandwidth after a massive departure. 69
3.12. Percentage of nodes that do not receive a viewable stream after a massive departure. 70

4.1. Forwarding of updates in a gossip-based system 77
4.2. Examples of selfish deviations 78
4.3. Accountable gossip 80
4.4. Sending of an onion using two relays. 85

5.1. Membership using FireFlies [25] 99

5.2. Exchanges of updates between nodes that have different interests . . . 99
5.3. Monitoring of nodes to ensure the forwarding of updates 101
5.4. Privacy preserving verification of a forwarding of a node B 102
5.5. Propagation of messages inside a session 105
5.6. Monitoring part of an interaction between two nodes 107
5.7. $k - PAG$ illustration with k=3 108
5.8. Privacy presence of a global and active attacker controlling a varying proportion of the membership. 115
5.9. Bandwidth consumption of $k - PAG$ with several 300kbps contents per cluster and 3 monitors per node [sim] 116
5.10. Average bandwidth consumption of nodes running PAG with a 300Kbps payload. 117
5.11. Bandwidth consumption with 1000 nodes and a 300Kbps stream in function of the size of updates [sim] 119
5.12. Scalability of PAG compared to $AcTinG$ with a 300kbps content [sim] 121

List of Tables

I. Terminology used in this document. x

I. Possible attacks on the LiFTinG protocol (*Based on* [26]). 28
II. Values and source of blames emitted during direct verifications in LiFTinG (*Based on* [26]). 29
III. TrInc: Global state of a trinket 36
IV. TrInc: Per-counter state. 36

I. Overhead of colluders in $AcTinG$. 68
II. Average bandwidth and memory usage of $AcTinG$ in function of the system size. 70

I. Detailed summary of existing approaches. 92

I. Maximum video quality sustainable in function of the network links capacity, and the associated bandwidth consumption, in a system with 1000 nodes 118
II. Number of RSA-1024 signatures and homomorphic encryptions per second in a system of 1000 nodes [sim] 120

Introduction

Contents

Scientific context 1
Gossip, coalitions of freeriders and privacy 3
Objectives of this research work 4
Research location 5
Organization of this document 5

In this section, we first present the scientific context of this document, and more specifically, we focus on the problem of disseminating information inside a P2P network in presence of selfish nodes. We develop the motivation of our research in the two following sections. We first analyze the negative impact that selfish nodes have on the dissemination of information in a P2P system. Then, we detail how selfish-resilient protocols may allow users to learn information about each others, and explain for which reasons they may be reluctant to use such protocols. After providing a description of these two problems, we detail the objectives of this research work. The first contribution consists in the design of a gossip protocol that is resilient to individual, and collective, selfish behaviors. The second one is a protocol that limits the information that is revealed about users, while deterring selfish behaviors. Finally, we give a brief description of the contents of this document.

Scientific context

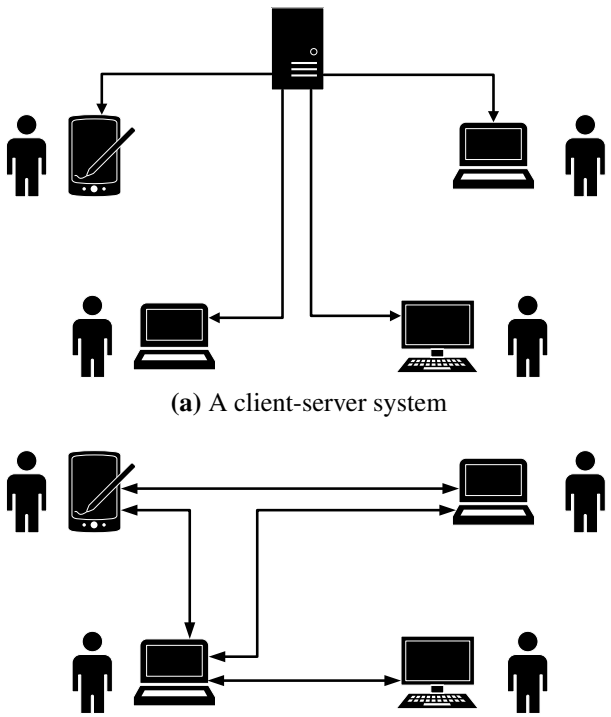
The Internet allowed the development of numerous applications which use it to connect physically distant machines in order to provide elaborated services. In a general way, in these distributed applications some network nodes provide a service while others consume it. Several paradigms exist to design applications over a network, including the client-server and the peer-to-peer approaches, which are illustrated in Figure 1.

The traditional client-server approach is built around a server, which processes the requests of one or more clients. Using this paradigm, the service quality can suffer from the nature of the Internet and the habits of the users. Thus, the server can become a bottleneck if it receives too many requests simultaneously, or if the network links are

overloaded. In addition, the server constitutes a single point of failure of the application, and could crash or even become Byzantine. Due to the cost of high performance servers, applying this paradigm is not accessible to everyone. In Figure 1, a single server answers the requests of 4 clients, which use different devices (e.g., a laptop, a smartphone, a personal computer).

Differently, in the peer-to-peer (P2P) model, the users of the application, called peers, play both the roles of clients and servers, sharing their resources (CPU cycles, memory, network bandwidth) to the application, and benefiting in return from it. Using a P2P architecture instead of a client-server approach can be interesting for several reasons. First, there is no single point of failure, or bottleneck, and peers can join or leave the system at any moment. Then, scaling the system up to several thousands of nodes is easier and cheaper, and it is possible to design an application whose membership is able to absorb quick and massive joining/departure of nodes without significantly degrading the quality experience of nodes. In Figure 1, all nodes collaborate with two other nodes, both in emission and in reception, to benefit from the application.

It is worth mentioning that hybrid approaches (e.g., used in [6, 7]) exist, where some servers help a P2P system with some tasks. For example, in NetSession [7] peers are coordinated by a dedicated infrastructure to exchange content. These hybrid approaches seek to obtain the best of the client-server and peer-to-peer approaches.



(a) A client-server system

(b) An example of peer-to-peer system

Figure 1 – Distributed systems architecture paradigms

Historically, file sharing applications were the firsts to adopt the P2P paradigm. Nowadays, these applications are mainly represented by BitTorrent. In a file sharing application each user is interested in obtaining a copy of at least one file. Each file is splitted in fragments, and nodes have to collaborate to obtain all the fragments, and reconstitute the file. Users will receive fragments from those who have it, and send it to the others. In this case, each peer has to dedicate some of its upload bandwidth to send fragments to other nodes, even if it does not benefit from doing it. Other types of applications that relied on the P2P approach include phone applications (Skype [79]), instant messaging (Jabber [80], ICQ [81]), live streaming (SopCast [82], Veetle [83]), and even social networking (Diaspora [84]), in which client sites are peer nodes.

In this document, we are focusing on the problem of information dissemination in P2P networks, which consists in efficiently providing a given content to a set of nodes. Indeed, this is a central service in various applications, for example used to update the list of peers in an applications, or share a multimedia content.

A simple yet efficient way to disseminate content in a network of peers is to use random exchanges between nodes. More precisely, each user has to choose randomly a given number of other users to which it will send fragments of the content it already owns. Then, each of these users that received these fragments will have to forward further this content to other peers. After a given number of hops each member of the audience will have received all the fragments with high probability. This paradigm, called gossip, has the advantage of being simple, of ensuring probabilistic guarantees on the dissemination success, and of easily tolerating the arrival, or departure, of nodes.

Gossip, coalitions of freeriders and privacy

As we previously said, users in a P2P system are expected to share their resources and benefit from the system in return. It has been shown that some users tend to be selfish, as they try to avoid contributing to the system, for various reasons, while trying to maximize their benefit. For example, in DSL networks, nodes have more bandwidth available in download than in upload, and peers may be tempted to participate to a gossip session even if they are not able to contribute properly to it. Another possible reason is that users may want to minimize their participation in a protocol to hide it from their Internet Service Provider (ISP).

In addition, selfish nodes can collude to collectively decrease their contribution to the system, or to increase more their benefit. For example, nodes located inside a same local area network could be more interested in exchanging chunks between them, because the latencies of their exchanges are lower and their throughput higher. These exchanges, which are not random, could degrade the performance of the global gossip system. For example, if all exchanges between nodes have to be balanced (i.e., a user must send the same number of fragments that it is receiving), then these nodes will receive updates from their coalition more frequently, and collaborate less with other nodes. Identifying collective rational deviations in state-of-the-art protocols, and designing a gossip-based

protocol protected from rational coalitions constitute the first contribution of this thesis.

The impact of individual or collective selfish nodes is particularly acute in the case of live streaming applications. In these applications, a particular node produces content, for example a video, creates the fragments, and immediately starts spreading the fragments in the audience of users, which have to share them. Regularly, some fragments are supposed to have been received by all nodes and are then delivered to the media application of nodes. These updates will not be exchanged in upcoming exchanges. Thus, if some nodes have not received the updates due to selfish behaviors, they will suffer from a degradation of their experience quality.

In order to protect a protocol from selfish nodes, which do not participate as they are supposed to, system designers usually try to detect all kind of faults and evict the guilty nodes, or try to design a system in which the best interest of nodes is to participate correctly. As we will see later in this document, in the first case, the details of the interactions of nodes can be learned by other participating nodes, while in the second case all rational behaviors cannot be avoided. There is a tension between the idea of verifying the behavior of nodes and the idea of protecting their privacy. Nowadays, Internet users are aware that their private information is worth something. For example, two billions dollars are used each year in the United States to buy personal data [85]. As a consequence, users may want to have more control on their data, and may avoid to use protocols that do not offer them this possibility. Conciliating both privacy and resistance to selfish nodes is the second problem we tackle in this document.

Objectives of this research work

We highlighted two problems that P2P content sharing applications face. First, we have shown that selfish peers threaten the good dissemination of content, either by deviating individually, or by elaborating collaborative strategies to game the system. Thus content dissemination systems have to detect selfish behaviors and evict the guilty nodes, or to provide incentives for nodes to stick to the protocol. We defined, prototyped and evaluated $AcTinG$, a gossip-based protocol that deters and detects collusions of selfish nodes. The performance of $AcTinG$ was evaluated on a testbed comprising 400 nodes running on 100 physical machines, and compared its behavior in the presence of collusions to two state-of-the-art protocols. The performance evaluation shows that $AcTinG$ is able to deliver all messages. It also shows that $AcTinG$ is resilient to massive churn. Finally, using simulations, we showed that $AcTinG$ exhibits similar scalability properties as standard gossip-based dissemination protocols.

Then, we argued that detection methods typically leak information about users, either to a central entity, or to other users. Nowadays, people are becoming reluctant to use systems that collect information about them. Designing a gossip protocol that limits selfish behaviors and protect the users' privacy is an interesting challenge. The second objective of this work is to define and prototype such a gossip protocol. We evaluated this protocol, named PAG, using deployments on a cluster of 48 machines, simulations involving up to 10000 nodes, and a cryptographic protocol verifier. The performance

evaluation of PAG, performed using a video live streaming application, shows that PAG is compatible with the visualization of live content on commodity Internet connections. Furthermore, PAG's bandwidth consumption scales logarithmically with the number of users thanks to the inherited properties of gossip.

Research location

This research has been made in the Erods team of the LIG (Laboratory of Informatics at Grenoble). This team is interested in the construction and the management of Cloud systems, but also in distributed and multicore systems.

Organization of this document

This thesis is organized into 5 Chapters, and 2 Parts:

Chapter 1 shortly reviews the fundamental concepts behind gossip-based content disseminations, presents the problems caused by selfish nodes and how protocols generally deal with them. We also explain that selfish nodes may collude and thus perturb the dissemination of information, and that traditional mechanisms designed to deal with selfish nodes may endanger the privacy of users.

Rational collusions in gossip-based dissemination systems

Part I treats the problem of collusions of selfish nodes in gossip, and is made of Chapters 2 and 3.

Gossip in presence of rational nodes

Chapter 2 presents an overview of the works related to rational-resilient gossip-based dissemination. We detail gossip protocols that were specially designed to detect or deter selfish deviations, accountability approaches and general Byzantine-tolerant approaches. We identify the requirements of a gossip protocol accounting for collective selfish behaviors, and present a summary of whether existing approaches match them.

AcTinG: accurate freerider detection in gossip

Chapter 3 proposes a novel gossip-based protocol, named $AcTinG$, designed specially to tolerate collective selfish deviations. This chapter provides an overview and details of our prototype of $AcTinG$. Nodes running this protocol are shown to have all interest in sticking to it, and if they do not, would not harm the dissemination of updates and eventually be detected. $AcTinG$ is evaluated using deployments on a cluster of 40 machines, simulations involving up to one million nodes, and scalability metrics.

Privacy and rational resiliency in gossip-based dissemination systems

Part II treats the problem of privacy in selfish-resilient gossip, and is made of Chapters 4 and 5.

Privacy in rational-resilient gossip
Chapter 4 studies the problem of protecting the privacy of nodes while tolerating selfish nodes in gossip. First, we formalize the requirements to tolerate both selfish behaviors and protect the privacy of users in gossip. Then, we explain that existing selfish-resilient gossip protocols may leak information about users. Moreover, we assert that some anonymous communication protocols are both accountable and privacy-preserving but suffer from poor performance. In the same way, we introduce some solutions that combine both privacy and accountability, but we claim that they cannot be applied to gossip.

PAG: Private and accountable gossip
Chapter 5 proposes and evaluates a gossip-based protocol, named PAG, that forces nodes to participate actively in the dissemination of a content, while limiting the information they can learn the one about the other. First, we detail our assumptions about nodes and about the system. Then, we introduce the intuition behind PAG that relies on homomorphic encryptions. We also explain how this intuition is implemented in practice, using message exchanges. In addition, we detail the practical details that make PAG practical. Finally, the security of PAG is assessed using a cryptographic protocol verifier, simulations and a prototype of PAG is used to evaluate its performance on a deployment on a cluster of 48 machines.

Lastly, we conclude this document and expand on the possible future works. Extensions include designing a more lightweight privacy-preserving dissemination algorithm and handling collusions in PAG. Another challenge is to exclude faulty nodes while preserving the privacy of correct nodes.

Chapter 1

Gossip with colluding rational nodes

Contents

1.1. Probabilistic dissemination 8

1.1.1. Structured and unstructured overlays 8

1.1.2. Principles of gossip 9

1.1.3. Probabilistic guarantees 9

1.2. Selfish behaviors 11

1.2.1. BAR model 11

1.2.2. Utility functions 12

1.2.3. Byzantine fault tolerance or Nash equilibrium 12

1.3. Impact of selfish nodes 13

1.3.1. Individual deviations 13

1.3.2. Collusions 14

1.3.3. Privacy and selfish nodes 14

In this chapter, we introduce the fundamental concepts we will use throughout this document. This chapter has been thought as an extension of the introduction in the sense that it motivates the problems that are studied in this thesis. As such, the following chapters are more detailed, and develop notions that are quickly presented here.

First, in Section 1.1 we describe the theoretical foundations of gossip. Then, in Section 1.2 we present selfish behaviors which form a subset of the possible deviations that nodes could follow, and explain how these particular deviations are traditionally modeled. Finally, we describe in Section 1.3 the impact of individual or collective selfish deviations on the dissemination of updates, but also more surprisingly on the privacy of users.

SECTION 1.1

Probabilistic dissemination

In this section, we introduce the various existing ways to disseminate content in a peer-to-peer audience. Among them, we detail the gossip paradigm, and precise its probabilistic dissemination guarantees.

1.1.1 Structured and unstructured overlays

During the last decade, several peer-to-peer information dissemination methods have been developed. Some are based on a static overlays that nodes have to maintain and through which information is propagated, e.g., dissemination trees [8, 9, 10, 11, 53] or meshes [12, 54], while others like gossip uses an unstructured approach.

In a tree-based structured overlay, nodes are organized in a tree, and the disseminated content flows from the root of the tree towards the leaves. Maintaining a quality of service in presence of churn in trees is challenging because when an internal node disappears its subtree stop receiving the content. In addition, the participation of nodes to the system is not equally distributed among them, as leaves do not forward the content they receive to any nodes. For example, in a binary tree more than half of the nodes are leaves. In addition, nodes higher in a tree have to participate more than the others, and it is necessary to take this into account when creating or maintaining a tree. SplitStream [9] tries to overcome these drawbacks using several trees which simultaneously disseminate parts of the content.

A mesh consists in a static graph, which links all the nodes interested in a similar content. Contrary to trees, where the root is the only node distributing the content, in a mesh several nodes may receive the content in different places of the graph, and simultaneously propagate it in different directions. Typically, nodes retrieve content from several nodes in parallel, rather than just from one node, which improves the resilience of the system to nodes failures. For this reason, it is also easier to repair a mesh than a tree without affecting the service quality.

In gossip, nodes exchange contents the one with the others without using static associations, like in trees and meshes. This unstructured approach is thus simpler to create and

maintain. We detail gossip in details in the following parts.

1.1.2 Principles of gossip

The gossip paradigm has been inspired by the probabilistic studies on epidemic diffusion. Such studies tried to model the mechanisms that nature deploys to spread an infection among a population of individuals. In gossip, once a message is received by a node, the node is then infected and contagious, which means that it can propagate the disease to its neighbors during a given period of time. After some time, using this infect-and-spread approach, all nodes are infected, i.e., received the message. Gossip protocols represent a simple, yet efficient, approach to disseminate a message in a system.

Parameters. In [55], the authors discuss the parameters that are implicitly used in the previous paragraph. Each time a node receives a message for the first time, it forwards this message to a random subset of nodes, whose size is called the dissemination *fanout*. The period during two forwardings is called a dissemination *round*. In addition, nodes forward a message during a limited number r of rounds, which is called the number of dissemination *repetitions*. Following the analogy of an infectious disease, r is the number of rounds after which a node stays alive, or contagious, after its infection. The special case where r equals 1 is called the infect-and-die model.

Illustration. Figure 1.1 provides an example of a file dissemination using gossip in a system of 7 nodes. In this situation nodes forward a newly received message to 2 other random nodes during 1 round. During round number 1, the source of the dissemination owns a file that it transmits to two other nodes. Nodes that have received the file are represented in black, while those who have not are represented in white. The two nodes that received the message from the source forward this message to two other nodes during round 2. During round 3, the nodes that have received the message for the first time forward it, and after four rounds, all the nodes in the system have received the message. One may see that using gossip a node may receive several times the same message, which is the case of many nodes in the illustration. In practice, determining the ideal fanout and number of dissemination repetitions usually depends on the targeted application, and on the assumptions that the system designer takes (e.g., concerning churn).

1.1.3 Probabilistic guarantees

In this section, we provide theoretical results that prove that gossip is both an efficient and scalable method to disseminate information in a set of N nodes.

Atomic dissemination. Kermarrec et al. [56] have analytically and experimentally studied the probability that all nodes receive a given message, a situation which is called atomic dissemination. Their results show that in the infect-and-die model, where nodes forward a message during only one round after its reception, if the fanout of nodes is close to $ln(N) + c$, where c is a constant, then the probability to ensure atomic

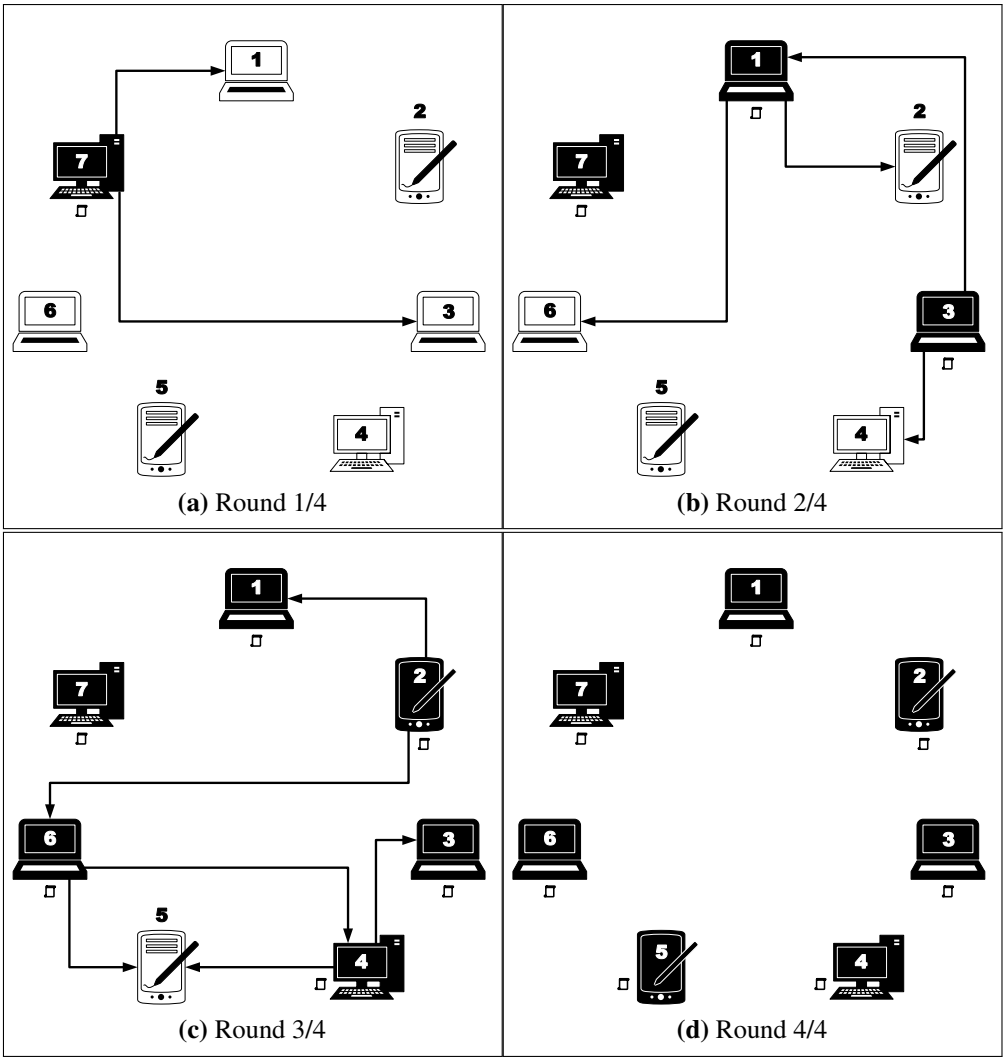


Figure 1.1 – Example of gossip with 7 nodes

dissemination is equal to:

$$P_{atomic} = e^{-e^{-c}} \tag{1.1}$$

Another interesting result is that even if nodes participate differently to the system atomic dissemination can be enforced. Indeed, it is enough that nodes participate to the dissemination in average by forwarding to $O(ln(N))$ other nodes.

Latency of dissemination. When the fanout of nodes is on average close to $ln(N)+c$, nodes receive all the messages with high probability. Under this assumption and in the infect-and-die model, Bollobás [1] has shown that nodes receive a message in average R rounds after its emission, where:

$$R = \frac{ln(N)}{ln(ln(N))} + O(1) \tag{1.2}$$

As R grows slowly with N, this equation proves the scalability of gossip.

We have seen that gossip is efficient to propagate messages inside a population of nodes if the fanout of nodes is correctly chosen as atomic dissemination is then ensured with high probability. In addition, the fanout of nodes and the latency of dissemination increase slowly with the system size, which proves its scalability.

SECTION 1.2

Selfish behaviors

In this section we present the types of deviations that nodes may execute in a gossip protocol. We first introduce the BAR model that consider three types of nodes. The following sections explain how rational nodes can be modeled using utility functions, and how it is possible to protect a protocol against selfish deviations.

1.2.1 BAR model

Assuming the existence of only two categories of nodes, correct and faulty nodes, limit the number of faults that can be tolerated in a system to one third of the membership. Considering a new category of faults, rational deviations, allowed researchers to build protocols that tolerate an unbounded number of rational nodes in addition to a bounded number of Byzantine nodes. Contrary to the two other categories of nodes, altruistic and Byzantine, the precise definition of rational nodes depends on the protocol they participate in.

In the Byzantine Altruistic Rational (BAR) model [13], nodes are classified into three categories.

- **Byzantine nodes.** First defined in [57] Byzantine nodes may deviate arbitrarily from a protocol. Nodes may be broken, for example if they are misconfigured or malfunctioning, but may also behave deterministically, e.g., to harm other users.

- **Rational nodes.** These nodes are self-interested and seek to maximize their benefit according to a known utility function. Rational nodes deviate from the protocol in any way that increases their benefit.
- **Altruistic nodes.** They strictly follow the protocol, and are often named correct nodes.

Nodes may be rational for several reasons, for example, they may want to save their resources [14], or they may limit their participation in order to avoid being detected if they illegally share files [58].

1.2.2 Utility functions

Rational nodes deviate from the protocol in any way that increases their benefit. However, defining this benefit depends on the application that nodes are running. For system designers, tolerating rational nodes can be done in two ways: (i) first, rational nodes can be considered Byzantine, and if all deviations can be detected they would be handled correctly, (ii) second, the protocol can be adapted to encourage rational nodes to follow it. The first approach is usually more costly in terms of resources than the second one.

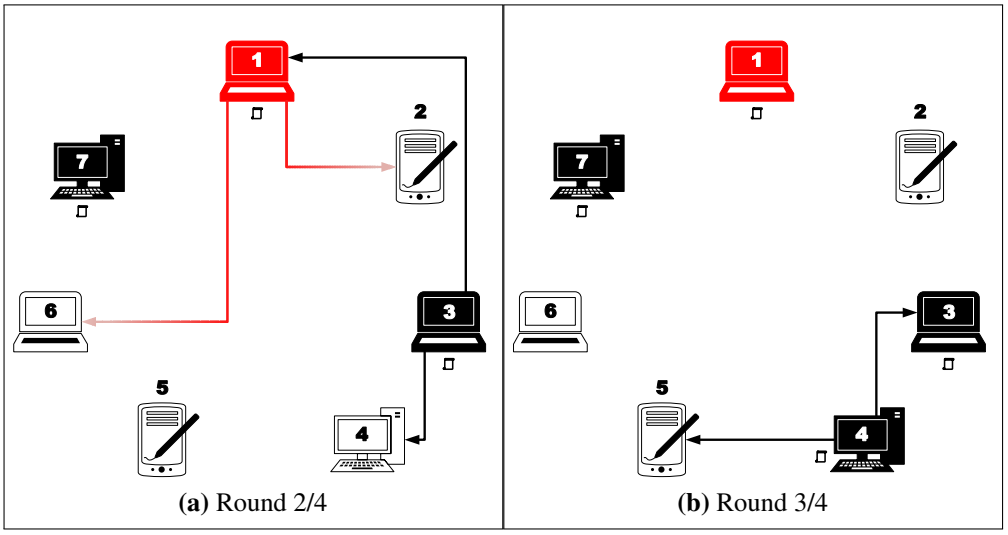
To evaluate precisely the benefit of nodes, and later convince rational nodes to participate correctly, it is conventional to define a utility function that accounts for the costs and the benefits of nodes when they participate in a protocol. The costs of nodes may include computation cycles, storage, network bandwidth, overhead associated to sending or receiving messages, power consumption, etc. Their benefit can be represented, for example, as the fraction of released updates they receive, or as the (lowest possible) jitter in the case of live-streaming. In this last case, jitter is the proportion of time during which the stream is not viewable. Due to the use of error correction methods (e.g., FEC[1]) jitter is not exactly equal to the proportion of missing updates. Erasure codes [15, 16, 17] and Multiple Description Coding [59] are techniques that can help reduce the average jitter of a application that disseminates multimedia content.

In Figure 1.2 we use the same gossip scenario we presented in Figure 1.1 to introduce a situation in which a single rational node has a negative impact on the dissemination of an update. This time, node 1 is a rational node, and is represented in red. During round 2, node 1 chooses not to forward the update it should send to nodes 2 and 6. Consequently, at the end of round 3, nodes 2 and 6 did not receive the update everyone else received. In this situation, these two nodes would have no other possibility to receive this update, while node 1 decreased its participation.

1.2.3 Byzantine fault tolerance or Nash equilibrium

System designers have two ways to protect their protocol from rational nodes: detecting all kind of faults using Byzantine fault tolerant methods, or establishing a Nash equilibrium. The first method consider all deviations equally and is generally costly. The second one, however can be more easily attained.

[1] FEC stands for Forward Error Correction



(a) Round 2/4 **(b)** Round 3/4

Figure 1.2 – Example of gossip with 7 nodes in presence of a rational node

John Nash developed in 1951 some theoretical basis of game theory that are intensively used today in fields like economy, geopolitics and computer science. In [60], he introduced the notion of equilibrium that was later named after him. Simply stated, a Nash equilibrium consists in a situation where the players stick to their behavior because of their assumptions about other players.

Applied to gossip, where the players are the nodes, it means that a node that makes the assumption that the other nodes are correct, or altruistic, because it does not know them will deviate from the protocol if its utility function tells it to do so. Thus, system designers can build their protocols and model the utility of nodes in such a way that the best strategy for individual rational nodes is to stick to the protocol.

SECTION 1.3

Impact of selfish nodes

In this section, we introduce individual and collective rational deviations, and the threats that current detection mechanisms imply on the privacy of users.

1.3.1 Individual deviations

The authors of [61] found that 70% of the users of Gnutella where freeriders in a 24-hour period. In addition, the top 1% of contributors returned 50% of all contributions.

Several protocols have been devised to deal with the problem of rational nodes in different kind of collaborative systems, among which spam-filtering content dissemination [18], N-party transfer [62], and file transfer protocols [61].

In Figure 1.1, we detailed an example of dissemination in presence of a rational

node. This simple example illustrated how selfish deviations can heavily harm the dissemination of updates in a system. Gossip protocols have to consider this situation and be designed to avoid this situation. We will further examine in chapter 2 the protocols that have been devised to treat the problem of individual selfish nodes in gossip-based systems.

1.3.2 Collusions

Collective deviations, or collusions, are situations where some nodes deviate from the protocol according to a strategy collectively defined. This type of deviation have been observed in P2P file sharing systems [19, 20]. To define their strategy, nodes could silently communicate (i.e., outside the official protocol), which has also been observed [21]. Collective deviations are more subtle than the individual ones. They are generally defined in reaction to mechanisms that aim at limiting the impact of individual deviations. As a consequence, depending on the protocol nodes are running they will be different.

As an example, a possible way to force nodes to communicate is to use tit-for-tat, or balanced, exchanges where a node cannot receive more updates from its partner than it is sending to it. In these kinds of systems, one possible colluding strategy is for two (or more) nodes to exchange updates off-the-record the one with the other as soon as one of them received it. This way, the colluding nodes receive updates sooner than other nodes, and do not miss them. However, this strategy does harm the dissemination of updates. When correct nodes would want to exchange updates with colluding nodes the resulting exchanges will be poor in term of number of updates exchanged. The messages that colluding nodes receive are less propagated in the system. We examine more in details the possible strategies of colluding nodes in state-of-the-art protocols, in chapters 2 and 3.

1.3.3 Privacy and selfish nodes

In order to protect a protocol from selfish nodes, which do not participate as they are supposed to, system designers usually try to detect all kind of faults and evict the guilty nodes, or try to design a system in which the best interest of nodes is to participate correctly. As we will see later in this document, in the first case, the details of the interactions of nodes can be learned by other participating nodes, while in the second case all rational behaviors cannot be avoided, e.g., collective deviations. There is a tension between the idea of verifying the behavior of nodes and the idea of protecting their privacy. Nowadays, Internet users are aware that their private information is worth something. For example, two billions dollars are used each year in the United States to buy personal data [85]. As a consequence, users may want to have more control on their data, and may avoid to use protocols that do not offer them this possibility. Conciliating both privacy and collusions resistance is the second problem we tackle in this document detailed in chapters 4 and 5.

PART I

RATIONAL COLLUSIONS IN GOSSIP-BASED DISSEMINATION SYSTEMS

Chapter 2

Gossip in presence of rational nodes

Contents

2.1. Rational resilient dissemination protocols 18
2.1.1. Nash equilibriums 18
2.1.1.1. BAR Gossip and FlightPath 18
2.1.2. Audits and statistical approaches 23
2.1.2.1. Enforcing fairness in a live-streaming protocol . 23
2.1.2.2. SecureStream 25
2.1.2.3. LiFTinG 26
2.2. Accountability techniques 30
2.2.1. Software-only accountability 30
2.2.1.1. PeerReview 30
2.2.1.2. AVM 33
2.2.2. Hardware-assisted accountability 33
2.2.2.1. A2M 34
2.2.2.2. TrInc 35
2.3. BAR-transformation protocols 37
2.4. Summary 39
2.4.1. Requirements 40
2.4.2. Drawbacks of existing solutions 41
2.4.3. Conclusion 43

Using a gossip-based protocol to disseminate information in large-scale P2P systems is a simple yet reliable and scalable approach. However, in the basic version of gossip users are not forced to participate in the dissemination. They can receive the content, maximize their benefit, and yet try to minimize the cost they pay in return, which often consists in upload bandwidth or CPU cycles they dedicate to the application.

In order to tolerate selfish nodes, a first approach consists in designing specific gossip-based dissemination protocols that encourage nodes to participate correctly. The ideas behind these protocols is to provide incentives that will reward the participation of nodes, and/or to detect and punish the deviations of nodes. The second possibility is to apply generic accountability mechanisms on top of a gossip protocol to compare the behavior of nodes with the execution of a correct node.

This chapter, which reviews the works related to collective rational deviations in dissemination protocols, is organized as follows: Section 2.1 provides some background about existing rational-resilient gossip protocols. Section 2.2 details accountability approaches that could be applied on top of any gossip protocol to verify the correctness of nodes. Section 2.3 presents methods that transform a P2P protocol into a Byzantine tolerant protocol. Section 2.4 identifies the requirements of a gossip protocol accounting for collective selfish behaviors, and presents a summary of whether the related works match these requirements. Section 2.4.3 concludes this chapter.

SECTION 2.1

Rational resilient dissemination protocols

In this section, we present the existing rational resilient gossip-based dissemination protocols. For each of them, after providing an overview of their data dissemination scheme and of their defense mechanisms against rational behaviors, we underline the weaknesses that colluding nodes can seize. In sections 2.1.1.1 we present a live-streaming protocol which is built around the principle of Nash equilibrium. We later describe, in sections 2.1.2.1 and 2.1.2.3, solutions which use the principle of distributed auditing to check the correctness of nodes.

2.1.1 Nash equilibriums

In the following, we present two live-streaming protocols, based on gossip, that are shown to be Nash equilibriums, which means that individual selfish nodes stick to the protocol believing that it is the best strategy they could apply.

2.1.1.1 BAR Gossip and FlightPath

BAR Gossip [22] is the first P2P gossip-based live-streaming protocol designed explicitly to tolerate both rational and Byzantine (i.e., arbitrary) behaviors. It was later modified into Flightpath [23] to increase its scalability and improve its churn management.

Streaming model and main ideas. In gossip-based content dissemination systems, chunks of data are exchanged at each round between randomly selected partners. It is precisely this randomness that gives gossip protocols their robustness. However, such non-determinism refrains the system from easily checking the legitimacy of partner selections. For example, without determinism nodes could exchange updates only with their accomplices, or sometimes avoid to forward them. Thus, while traditional gossip elects partners randomly, BAR Gossip employs a verifiable pseudo-random number generator (PRNG) and signature schemes to build the partner selection algorithm. Indeed, the randomness of associations is preserved, and hence so is the robustness of the dissemination algorithm, while the legitimacy of associations can be verified.

BAR Gossip makes several assumptions about nodes and communications synchrony. Nodes are uniquely identified, and maintain their clocks synchronized within δ seconds of each others, and communicate over point-to-point, synchronous and unreliable links using both TCP, and UDP. During the streaming session, time is divided into rounds of duration $(T + \delta)$ seconds, where T is a time interval sufficient to complete the exchanges of updates required by the two possible exchange protocols. The source of the video stream is supposed to be altruistic (i.e., to strictly follow the protocol). Each round, it generates some new updates, and send them to a random fraction of the audience. The membership is assumed to be static: nodes have to subscribe to the session prior to its start, and non-Byzantine nodes remain in the system for the whole duration of the broadcast.

The role of POMs. All messages that a node sends are signed, and when a node does not comply with the protocol it takes the risk of being denounced to the source of the stream via the generation of a verifiable proof of misbehavior (POM). This POM links a fault with its author using cryptographic primitives, possibly joining several messages sent from the guilty node. A centralized trusted entity periodically collects the POMs from nodes. The interaction relies on a *balanced cost* principle: interacting with the trusted entity has a constant cost, independently of the presence of a POM. The identity of non replying nodes is also communicated to the source. These nodes are then *evicted* from the live streaming session by means of an eviction list that the source sends together with the updates it generates.

Exchange protocols. Updates exchanges between two nodes, called partners, can follow two schemes:

- *Balanced exchanged protocol* where the two partners exchange the *same* number of updates.
- *Optimistic push protocol* where the two partners exchange a *different* number of updates.

These two exchange protocols share the same basic structure, illustrated in Figure 2.1, made of four steps. We briefly describe these steps and explain why rational nodes are encouraged to follow them.

- *Partner selection.* A node chooses a gossip partner to exchange updates with. The identity of this node is obtained using a PRNG which is seeded using the current round number signed with the initiator's key. The first number generated is deterministically mapped into the identity of a gossip partner. The chosen gossip partner can verify this computation, and denounce the initiator using the signed message it sent, which in this case constitutes a POM.

- *History exchange.* The two parties learn about the unexpired updates the other party holds. In particular, the initiator sends a hashed history, and only after obtaining the other node's history in clear it sends its history in clear. The contacted node can check that the hashed history and the clear one are consistent the one with the other. Otherwise, these two messages constitute a POM again the initiator.

- *Update exchange.* Each party copies a subset of these updates into a briefcase that is sent, encrypted, to the other party. Depending on the exchange protocol, the number of updates that is placed in these briefcases may not be the same. Particularly, using the balanced exchange protocol nodes are encouraged to announce all their updates if they want to maximize the number of updates they will receive. Each of them sends a briefcase message containing the encrypted updates that it is supposed to send, and their ids in clear.

- *Key exchange.* The nodes swap the keys needed to access the updates in the briefcases they received. If a node does not send the right key, the other one would be able to constitute a POM. Also, if a node does not respond to a key request, the requester node continues sending the request, thus consuming the download bandwidth of the other node.

The two protocols differ in the way they select the updates that will be placed in the briefcases. The balanced exchange protocol uses a tit-for-tat principle where both nodes profit from the same number of new updates. The authors have proved that this exchange protocol is a Nash equilibrium [60], which means that the best strategy nodes could follow is the protocol. The keystone of this protocol is a principle of deferred gratification: when interacting with each others, the best option of nodes is to correctly follow each step of the balanced exchange protocol to finally benefit from it.

On the contrary, the optimistic push protocol is designed to help nodes that have fallen behind in the broadcast, and allow a peer to receive more updates than it can give in return. This protocol is not a Nash equilibrium. However, during unbalanced exchanges, the same quantity of data (but not the same number of updates) is exchanged, through the sending of updates that are about to expire (old updates), or of junk data. This results in a waste of bandwidth, but this is designed to prevent free rides.

FlightPath: an approximate Nash equilibrium. To improve its performance and to obtain a dynamic membership, the authors of BAR Gossip modified it into another protocol named FlightPath. The core of the protocol remains the same, but several modifications are introduced. FlightPath exhibits better performance, decreasing the jitter and the bandwidth consumption of nodes, than its predecessor. In opposition with

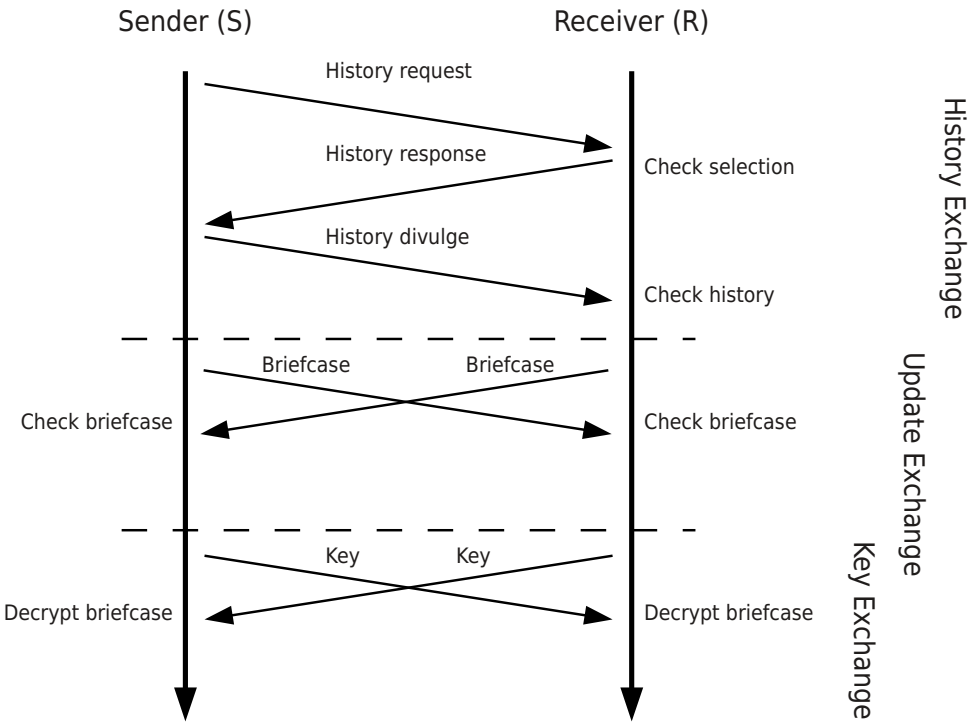


Figure 2.1 – Basic trade illustration of BAR Gossip (*Based on* [22]).

BAR Gossip, where the set of participating peers is defined at the beginning of the streaming session, FlightPath allows a dynamic membership, and deals more efficiently with realistic conditions, where nodes can leave or join the system at any moment. This gain is allowed thanks to an evolution of the system goal, which is now to reach an approximate Nash equilibrium, allowing a bounded imbalance in trades.

In an approximate Nash equilibrium [2], rational peers deviate from the protocol if and only if they expect to increase their benefit by a proportion of ε doing so. The advantages of using a ε-Nash equilibrium consist in giving more freedom to design practical solutions. FlightPathuses BAR Gossip as a basis, but the authors claim to have improved performance while decreasing the overhead of the protocol. For example, nodes do not have to waste network bandwidth by sending garbage data to balance bandwidth consumption, the load can be redirected away from a busy peer, trades with "poor" peers can be avoided and the use of arithmetic coding of data provide more opportunities for useful trades.

FlightPath optimizations. Gossip protocols are well-known for their robustness, but the authors explain that the very randomness of partner selection, basis of this robustness, may induces difficulties to propagate updates by a hard deadline in live streaming system. To overcome this issue, this article introduces two kinds of improvements, the first three have an influence on the peak bandwidth of the protocol, and the other three on jitter.

- *Reservation.* This mechanism seeks at limiting the number of concurrent trades of

any peer, for each round. Too many exchanges may lead a peer to the impossibility of finishing all exchanges in time, but also to receive duplicate updates.

- *Splitting need.* During a round, if a peer has many partners it can split its demands among them instead of demanding every missing update to each. There is a limit between two few demands per peer and risking not to obtain all updates, and demanding too many and wasting bandwidth.
- *Erasure codes.* For each round, the source codes all the stream data in more updates that normally necessary introducing erasure codes. The probability to trade similar updates during a round is decreased.
- *Tail inversion.* to reduce jitter, peers have the possibility to request updates that do not come from the current round. Instead of requesting updates in most-recent-first order, a peer has the option to receive updates coming from the two most recent rounds first and then updates in oldest-first order.
- *Imbalance ratio.* each peer bounds by α the proportionality factor between the downloads and uploads coming from any other node. When $\alpha = 0$, every trade is balanced, allowing an unlucky peer to fall behind. At the opposite, if $\alpha = 1$, free-riders can game the system. Experimentally, the authors set α to 10%. This is the basis of their ε-Nash equilibrium.
- *Trouble detector.* Each peer monitors its performances during each round. If they fall below a certain threshold, peers can initiate more than one trade in a round to avoid jitter. This solution is only a security net, because increasing the number of concurrent trades also increases the bandwidth dedicated to uploading.

Dynamic membership. FlightPath also presents two schemes to handle nodes that want to join or leave the system during a streaming session, a phenomenon that is also called churn. In a BAR environment one has to be careful in providing benefit to any peer who has not earned it. Two mechanisms are proposed : the first allows the tracker to modify the membership list and to disseminate it to all relevant peers, the second lets a new peer immediately begin to trade so that it does not have to wait in silence until the tracker's list takes effect.

In the first solution, a tracker periodically assigns new membership to reflect joins and leaves. An epoch is defined as a set of Δ successive rounds. If a peer joins in epoch e, the tracker places it into the membership that will be used in epoch $e + 2$. At the boundary between epochs e and $e + 1$, the tracker shuffles the membership list and transmits it to the source. Shuffling prevents Byzantine peers from attempting to place themselves at specific indexes. The tracker notifies nodes about this new membership list with a new kind of updates, partial membership lists, that are exchanged in priority compared with other updates (digests or stream updates).

The second proposition organizes peers as tubs such that the first tub contains the oldest peers and subsequent tubs contain younger and younger peers. A peer selects partners from its own tub and also from any other peer older than itself. However, the probability that a peer from tub t selects from a tub $t' < t$ decreases exponentially with $t - t'$.

2.1.2 Audits and statistical approaches

Other protocols, which are not based on Nash equilibriums, have been devised to deal with selfish nodes. These solutions make nodes maintain a log of their interactions and audit each others in order to detect deviations and punish them. These protocols are explained in this section.

2.1.2.1 Enforcing fairness in a live-streaming protocol

The live-streaming system presented in [24], the fault model of nodes and the approach are quite different compared to those of BAR Gossip, and FlightPath, where nodes are free to avoid to initiate some type of exchanges. In this work, any node not contributing its fair share of data may be expelled from the system. Nodes trying to maximize their utility are classified as Byzantine, and a punishment-based system is designed to evict them from the system. This punishing approach differs radically from the one used in BAR Gossip where a rational node is encouraged to stay and participate in the system by mean of a controlled exchange protocol.

Mesh overlay. The basis of the work presented in this publication consists in auditing a live-streaming system in order to encourage nodes to participate. Contrary to gossip, where partners are chosen randomly, nodes are here part of a mesh, and trade with a given static set of neighbors. These neighbors are randomly determined by the streaming system initially, to make it harder for malicious users to take profit of this kind of configuration.

Characterization of a system under attack. The expected average behavior of a streaming system working in ideal conditions, without Byzantine nodes, is determined through simulations which monitor the minimum, average, and maximum download and upload factors of nodes. These simulations showed that when the maximum upload factor authorized per node increases, some nodes participate more in the dissemination of data than other, even though all of them are behaving correctly. When a proportion of the peers is behaving opportunistically the minimum upload factors of correct nodes are increased. Thus, the principle of this solution is to monitor the upload factor of nodes, and when it is greater than a determined threshold evict all nodes that do not participate enough in the dissemination.

Auditing approach. The auditing method consists in monitoring the maximum upload factor measured in the membership to detect the presence of rational nodes. However, the minimum upload factor does not follow a clear pattern, making it hard to identify the minimum contribution of correct nodes under compromised scenarios. Therefore, applying a minimum threshold for the upload factor to detect opportunistic nodes may also punish correct nodes if it is not carefully chosen.

Auditing infrastructure. The auditing infrastructure has two functions: (i) collect accountable information about the download and upload factors of nodes; and (ii) determine the best threshold and apply it to evict nodes that do not participate enough. Each task is assumed by a different component. We illustrate the different components

of the monitoring infrastructure in Figure 2.2. In this figure, node S is the source of the content that nodes A to F share with each others. Local auditors are based on nodes A to F, while global auditors G_1, G_2 and G_3 periodically audit the whole system.

Local Auditors. Local auditors are hosted on the nodes participating in the system, and are not trusted as any node can be malicious. It is important to distinguish a local auditor from the live streaming application from which it obtains the history of exchanged packets.

- Local auditors periodically compile the history of exchanged packets of the node they control. This history is then signed and published to an assigned subset of neighbors. This level of indirection is used to prevent nodes from masking their real upload and download factors by presenting different information to different auditors.
- Each local auditor checks that its node's neighbors sent more data than the minimum threshold. If not, it emits an accusation against the node that is transmitted to a global auditor. Local auditors also verify that the quantity of data received from, or sent to, a neighbor is exactly what it says to have sent, or received. If these checks fail then the local auditor tells its local streaming application not to further exchange packets with the suspect neighbor (in this situation it is not possible to say if it is really misbehaving).

Global Auditors. Global auditors are trusted components that run on dedicated external nodes. They have a global membership knowledge and interact with one another and local auditors. Their number is not fixed, and may vary depending on some parameters, such as the size of the system.

- Global auditors periodically sample the state of the system by querying local auditors. Then they cooperate and establish the minimum threshold according to a given strategy. Once a threshold has been chosen, it is gossiped to all local auditors.
- They also verify accusations issued by local auditors against particular nodes and, after validation, expel misbehaving nodes from the system. The validation is made by verifying the node's local history. Expelling a node consists in informing all its immediate neighbors of its status.

Choice of a threshold. Several strategies may be designed to establish a threshold to detect selfish nodes. We briefly present those that are described in the following:

- *Constant threshold.* Using a fixed threshold would either not detect enough selfish nodes, or evict correct nodes. In addition, selfish nodes that would learn the threshold would simply adapt their contribution in order not to be detected.
- *Increasing threshold.* The session starts with a upload threshold of 0.5, and once the average download factor of nodes is lower than 0.98 the system increases the threshold until the situation is considered normal again, in which case the threshold is decreased.

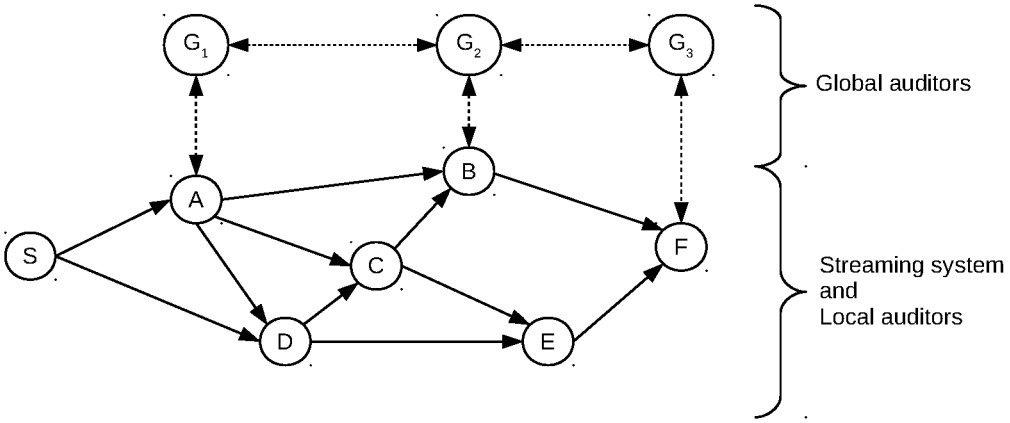


Figure 2.2 – Global auditing architecture (*Based on* [24]).

- *Percentile-based threshold.* In this strategy, the system periodically samples the participation factors of nodes, and if the system seems to be compromised, the threshold is set to the upload factor value that marks the lowest 10 percent.

2.1.2.2 SecureStream

SecureStream is a P2P live-streaming system built upon Fireflies, which is an intrusion-tolerant membership protocol.

Fireflies. Fireflies [25, 63] is a scalable Byzantine membership protocol, which presents nodes with a reasonably up-to-date view of the membership. As illustrated in Figure 2.3, the nodes are organised on several rings. The position of a node on the rings, which depends on their identifier, defines the nodes it monitors and those it is monitored by. In Figure 2.3, node A is monitored by nodes E, F and G, and monitors the nodes B, D, and F.

Fireflies is composed of three subprotocols:

- *Pinging protocol.* Nodes ping their neighbors to detect failures.
- *Gossip protocol.* An intrusion-tolerant protocol is used to disseminate information between nodes with probabilistic guarantees on the delay.
- *Membership protocol.* Nodes can suspect their neighbors, and it is possible for nodes to lift these accusations.

Exchanges. SecureStream relies on Fireflies to define the neighbors of a node, with which it then exchanges updates. The protocol use a pull-based approach, which means that nodes inform their neighbors about the updates they own, and they subsequently request those they are interested in. The integrity of packets is ensured using linear digests: the hashes of n packets are grouped, and the whole is signed. This approach decreases the overhead of cryptographic signatures, while being more secure than message authentication codes (MAC). Nodes cannot request more than a predefined

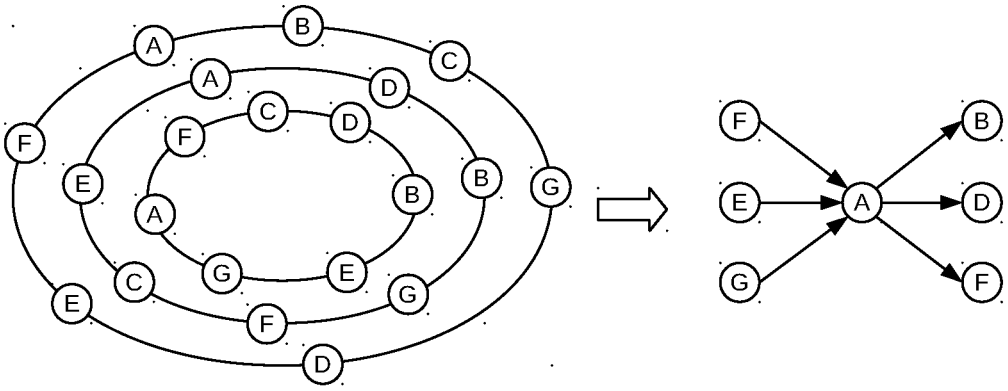


Figure 2.3 – Example of the architecture of Fireflies (*Based on* [25]).

quantity of packets from each of its neighbor, which makes it harder for malicious nodes to over-request their neighbors.

Auditing architecture. The protocol also provides an auditing mechanism, which is very close to the one of [24] we presented previously, to ensure nodes participate in uploading at least a certain percentage of the received stream. Nodes have to participate more than a particular specified threshold (statically defined by the system administrator, or dynamically). To reach this goal, two types of auditors are deployed in the system. Local auditors execute on the participating peers, while global auditors run on external dedicated nodes. The local auditors have to collect information about what a node received, or sent, and have to verify it by asking the neighbors of a node for the exchanges they made. If they discover that a node did not participate enough in the dissemination, they report the case to global auditors, which are also in charge of determining the minimum forwarding threshold value.

2.1.2.3 LiFTinG

LiFTinG [26] is the first protocol to detect colluding freeriders in gossip-based content dissemination systems. LiFTinG approach is close to the one of the protocol [24] we just described. The main difference is that histories can be cross-compared, and associations between nodes must respect statistical randomness. Furthermore, the mechanism of eviction is not the same. Nodes are affected a score, and each detected fault, depending on its gravity, makes this score increase. Once a given threshold has been crossed, the node is evicted from the system.

Exchange protocol. LiFTinG defines a three-phase gossip protocol where data is disseminated using an asymmetric push scheme. Nodes propose packets identifiers to a dynamically changing random subset of other nodes. They, in turn, request packets of interest, which are subsequently pushed by the proposer.

Nodes in the system communicate using lossy links (UDP) and can receive data from any other node in the system. It is assumed that nodes can pick uniformly at random a

set of nodes in the system. This is usually achieved through full membership (as in BAR Gossip or FlightPath) or a peer sampling protocol [27, 28, 64]. A source broadcasts a stream to a set of peers, using the same three-phase gossip protocol that is used by other nodes. The stream content is divided into multiple updates uniquely identified by their ids.

We detail below the three phases of the protocol:

- *Propose phase.* At every gossip period, each node proposes the set of updates it received since its last propose phase to a random set of f nodes. The fanout f is the same for all nodes in the system.
- *Request phase.* Upon reception of a proposal of a set of updates, a node determines the subset it needs and requests these updates.
- *Serving phase.* When a proposing node receives a request corresponding to a proposal, it serves the updates requested. If a request does not correspond to a proposal it is ignored, and non-requested updates are not served.

Rational deviations. Freeriders (or rational nodes) allow themselves to deviate from the protocol in order to minimize their contribution while maximizing their benefit. In addition, freeriders may adopt any behavior not to be expelled, including lying to verifications, e.g., to cover up the bad actions of colluding freeriders. However, it is assumed that freeriders have no interests in wrongfully accusing correct nodes. There are three different ways in which a freerider may deviate from the protocol : bias the partner selection, drop messages they are supposed to send, or modify the content of the messages they send.

The study of the methods with which a node can game the protocol gives three characteristics the system has to provide to prevent them. They are listed below.

- *Quantitative correctness.* a node must propose to f nodes at each period.
- *Causality.* received updates must be proposed in the next gossip period.
- *Statistical validity.* communication partners must be randomly selected.

Verifications. The verifications in LiFTinGare of two kinds, direct and *a posteriori*. Verifications can lead to the emission of blames or expulsions depending on the gravity of the misbehavior.

- *Direct verifications.* They are performed regularly, while the protocol is running and check if requested updates are all served and if received updates are further proposed to f nodes (quantitative correctness and causality).
- *A posteriori verifications.* They are launched sporadically, and require each node to maintain an history of its past interactions. In practice, a node stores a trace of the events of the last h seconds ($n_h = h/T_g$ gossip periods). The history is audited to check the statistical validity of the random choices made when

Attack	Type	Detection
Fanout decrease	Quantitative correctness	Direct cross-check
Partial propose	Causality	Direct cross-check
Partial serve	Quantitative correctness	Direct check
Bias partner selection	Entropy	Entropic check, *a posteriori* cross-check

Table I – Possible attacks on the LiFTinG protocol (*Based on* [26]).

selecting communication partners (entropic check). The veracity of the history is checked by cross-checking the involved nodes with probability p_{cc}.

Direct checks aim at verifying locally that every requested updates is served. This detection is always performed. Direct cross-checks are in charge of ensuring that every served update is further proposed to f nodes during next round. If node p_1 was served a given update by p_0, it has to transmit to p_0 the list of nodes to which it proposed the received update. Then p_0 can contact these nodes to check if p_1 told the truth. In practice, this is done with probability p_{cc}.

We reproduce in table I the classification of all possible deviating behaviors along with the type of verification that allows their detection.

Blaming Architecture. The detection of freeriders is achieved by means of a score assigned to each node. When a node detects that some other node freerides, it emits a blame message containing a blame value against the suspected node. Summing up the blame values results in a score. For scores to be meaningful, blame values are homogenized. Each node is monitored by a set of M other nodes named managers, distributed among the participants. When the score of a node is beyond a fixed threshold, the manager spreads through gossip, a revocation message against the node making the participants progressively expel it from the system. The mapping nodes-managers is done through a hash function, whose entry is the node IP address.

We give in Table II the blame values corresponding to direct verifications (quantitative correctness and causality). For a given node, R is the set of received update and S is the set of those that were served to neighbors. Figure 2.4 presents the type of message exchange involved in cross-checking.

The history of nodes are checked through an entropic check. When inspecting the local history of a node, the verifier computes the number of occurrences of each node in the set of proposals sent by p during the last h seconds. The uniformity of the distribution of these number of occurrences is then compared to a perfectly uniform distribution, with a tolerance threshold equals to γ. The value of γ must be set in order to be tolerant, and to limit the number of false positives (i.e., correct nodes being blamed). *A posteriori*

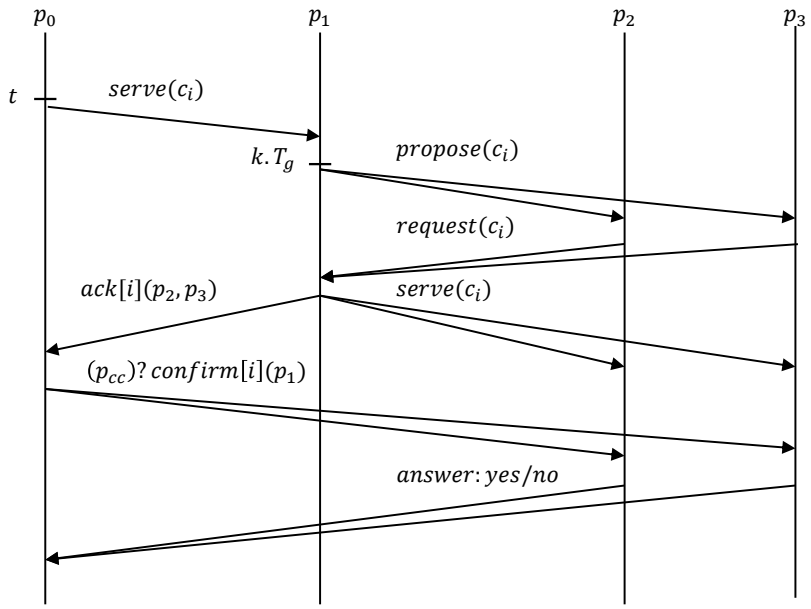


Figure 2.4 – Cross-checking protocol in LiFTinG (*Based on* [26]).

Cause	Blame value	Blame emitter
Non received update (direct)	$\frac{f}{\lvert R\rvert}$ for each of the $(\lvert R\rvert - \lvert S\rvert)$ missing update	requester
Non received ack (cross-checking)	f	initial sender
Missing or negative answer (cross-checking)	1	initial sender

Table II – Values and source of blames emitted during direct verifications in LiFTinG (*Based on* [26]).

cross-checking is achieved by polling all, or a subset of the nodes mentioned in the history for an acknowledgement. Each non verified allegation results in a blame.

Cause	Blame value	Blame emitter
Non acknowledged proposal	1	verifier

SECTION 2.2

Accountability techniques

Contrary to the protocols described in the previous section, accountability mechanisms are not specific to gossip-based dissemination protocols, but are designed to be applied on top of them to verify that the logs of nodes are identical to those that a correct execution could generate. In this section, we present some accountability techniques that either rely purely on software solutions, or also use specialized hardware.

2.2.1 Software-only accountability

The first set of accountability techniques do not rely on trusted hardware but use specific data structures based on cryptographic mechanisms. In this section we describe two protocols that use such accountability techniques.

2.2.1.1 PeerReview

In PeerReview [29], nodes must execute actions that are deterministic. Each of them maintains a secure record of the messages it exchanges with other nodes. The correctness of each node is then periodically, and independently, checked by other nodes, using a reference implementation that replays its log. This reference implementation can make snapshot of its state, and initialize itself from a given snapshot. Each node is in possession of a cryptographic pair of private/public keys, linked to a unique node identifier, that is be used to sign messages. A node that does not answer to a message cannot be exposed (presented as faulty) by the system, but is eventually suspected by every correct node. The reason is that under bad network conditions, which are not rare in the Internet, messages sent by correct nodes can be lost. Correct nodes do not communicate with suspected nodes, consequently a suspected node has to follow correctly a challenge/response protocol to be able to communicate anew with them. Each node is associated to a set of nodes, which are called its witnesses. These witnesses collect information about the node, check it correctness and make the results available to the rest of the system. For the system to work correctly, it is necessary to be able to map each node to its set of witnesses. It is assumed that each node has at least one correct witness. The processing overhead of PeerReview grows linearly with the number of peers which are in charge of checking a node actions. However, the message overhead depends on the targeted strength of fault detection. If every fault has to be eventually detected, then the overhead grows with the square of the number of nodes in the system. If a probabilistic detection is enough, this overhead can be logarithmic with the total number of nodes.

Tamper-evident log. On each node, the core of PeerReview consists in a secure log, which is an append-only list of the node inputs and outputs in chronological order. The log also contains periodical state snapshots and some annotations from the detector module.

The nodes use a hash function H, and h_{-1} is a known value which is needed for the computation of hash values. Each log entry $e_k = (s_k, t_k, c_k)$ contains a sequence number s_k, a type t_k and some type-specific content c_k. Sequence numbers must be strictly increasing. Additionally, each record includes a recursively defined hash value $h_k = H\left(h_{k-1}||s_k||t_k||H(c_k)\right)$. Figure 2.5 illustrates a fraction of a secure log containing 3 log entries. The hash value $H16$ is obtained from the previous hash value $H15$ and from the log entry $(S15, T15, C15)$, and is later used to compute the rest of the hash chain. This chaining mechanism along with the communication of a given hash value prevent nodes from tampering with the log entries added before this hash value.

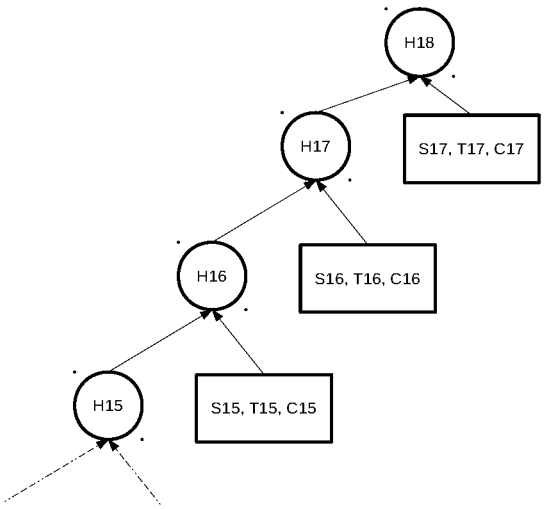


Figure 2.5 – Illustration of a secure log (*Based on* [29]).

An authenticator $\alpha_k^j = \sigma_j(s_k||h_k)$ is a signed statement by node j that its log entry e_k has hash value h_k. $\sigma_j(.)$ means that the argument is signed with j's private key. By sending α_k^j to a node i, a node j commits to having logged entry e_k and to the contents of its log before e_k.

Any node can also use α_k^j to inspect e_k and the entries preceding it in j's log. To inspect x entries, i challenges j to return the last x entries and h_{k-x}. If j responds, i calculates the hash value h_k from the response and compares it with the value in the authenticator. Moreover, i can use α_k^j as verifiable evidence to convince other nodes that an entry e_k exists in j's log.

Commitment protocol. The commitment protocol role is to ensure that a node log is consistent with the set of messages it has sent or received. When two nodes exchange messages, they both engage themselves to log them, by including authenticators in the

message, and in its acknowledgement. These authenticators cover the corresponding log entry. A log entry for a received message must include a matching authenticator; therefore, a node cannot invent log entries for messages it never received. Authenticators cannot be forged because they are signed.

When node i is about to send a message m to node j, it creates a new entry $(s_k, SEND, \{j, m\})$, where s_k is the chosen entry number, attaches h_{k-1}, s_k and $\sigma_i(s_k||h_k)$ to m, and sends the result to j. Thus, j has enough information to calculate h_k and to extract α_k^i, which must be valid. If it is, j creates its own log entry $(s_l, RECV, \{i, s_k, m\})$ and return an acknowledgement with h_{l-1}, s_l and $\sigma_j(s_l||h_l)$ to i. This allows j to extract and verify α_l^j. If it does not receive a valid acknowledgement, i sends a challenge to j's witnesses.

Consistency protocol. If a node i receives authenticators from another node j, it must eventually forward these authenticators to the witnesses of j. Then, periodically, each witness of j picks the authenticators with the lowest and highest sequence number, and challenges j to return all log entries in between. If j is not correct, then its witness obtained a verifiable evidence that j is faulty. Finally, each witness uses the log entries to extract all the authenticators that j has received from other nodes, and sends them to the corresponding witness sets. This propagation of the authenticators is planned to prevent faulty accomplices to mutually protect themselves.

Audit protocol. Each witness w of a node i periodically takes its most recent authenticator from i (say α_k^i), and then challenges i to return all log entries since its last audit, up to and including e_k. Then w appends the new entries to its local copy λ_{wi} of i's log. Node w can also create an instance of i's reference implementation and initialize it with a recent snapshot from λ_{wi}. Then, it replays all the inputs starting from that snapshot and it compares the outputs with those of the log.

Challenge/Response protocol. If nodes decide not to answer to messages, the above protocols are not useful. When a node j concludes that a node i does not answer, it indicates the suspected state for i and creates a challenge for i. Node j sends the challenge to i's witnesses, that forward it to i. If i does not send a response, the witnesses indicate that i is suspected. Challenges can take two forms.

The first kind of challenges, *audit challenge*, consists in demanding to node i to return a log segment which is delimited by two known values a_k^i and a_l^i (whose signatures are controlled). If i is correct, then it should answer the challenge with the correct log segment.

Send challenges are the second possibilities, they are triggered when a node did not acknowledged a message m. After extracting and checking the authenticator from m, any correct node is convinced that i must acknowledge m, and is waiting for this acknowledgement to release its suspicion.

Evidence transfer protocol. When a correct node has collected enough evidence to prove that node j is not correct, it has to propagate this proof. To achieve this,

any correct node is able to obtain the challenges collected by the witnesses of a node. Generally, a node collects the challenges that concern a node it communicates with. Eventually, each correct node come to the same conclusion concerning the correctness of a node.

PeerReview is a general method that can be used with any protocol to check the correct behavior of nodes. Its utilization of tamper-evident log allows the detection of any deviation. Combined with CSAR [30], PeerReview can be applied to non-deterministic protocols. The authors of PeerReview studied how their ideas can be applied with virtual machines. We present their work in the next Section. No specialized hardware is needed. However, nothing forces nodes to execute the audit phase.

2.2.1.2 AVM

Accountable Virtual Machines (AVMs) run a binary software image in a virtualized copy of a computer system. The method presented in [31] adapts the ideas of PeerReview to virtual machines which are in charge of detecting the possible deviations of a hosted program. AVMs are virtual machines that also provide the following services:

- *Logs.* Maintain a tamper-evident log (similar to the one presented in PeerReview) with enough information to reproduce the entire execution of a node.
- *Authenticators.* Associate each outgoing message with a cryptographic signature that links it to the log of the execution that produced it.
- *Snapshot.* Generates a snapshot of its state, and initialize itself with a snapshot to allow the verification of the correctness of a node based on a snapshot and the log of events from the moment this snapshot was taken.

Improvements over PeerReview. The interest of accountable virtual machines is their capability to record non-repudiable information. The log of all inputs and outputs that they generate allows auditors to check that the hosted software behaves correctly, checking if it corresponds with the log of an execution known to be correct. The idea of this paper is close to the one of PeerReview, but using virtual machines does not imply to understand and modify the source code of the untrusted application. In addition, using a virtual machine permits the simultaneous monitoring of several applications.

Audits. When user A wants to audit user B, he has to retrieve a segment of B's log and use B authenticators he previously learned about to check the integrity of the log. If this verification fails, A can transmit to all users the authenticator that will serve as a proof of B's misbehavior. In order to verify the execution, A needs a snapshot of the initial state of A at the beginning of the log segment, and will replay the execution using a reference implementation of the state machine used by B.

2.2.2 Hardware-assisted accountability

In this section, we now describe accountability techniques that rely on specific trusted hardware in addition to software procedures to obtain accountability. The main interest

of these methods is to show that the overhead of accountability can be significantly reduced through the use of simple trusted hardware.

2.2.2.1 A2M

A2M uses trusted hardware to provide an abstraction of a trusted log, that can be used to build accountable systems at a lower cost. The aim of [32] is to demonstrate the advantages of using a minimal trusted hardware in the fault tolerance domain. This work proposes Attested Append-Only Memory (A2M) to provide the abstraction of a trusted log, whose aim is to suppress equivocation (the act of telling different stories to different people) which limits performances of fault-tolerant systems.

A2M Usage. A service can mitigate the effects of Byzantine faults using an A2M to store information that cannot be altered. During setup, the untrusted component must make known to all possible verifiers the authentication keys for its A2M module and the identifier of the A2M log used for each distinct purpose.

To prove that it has committed a data item D in its log, a component can execute *append(q, h(D))*. The hashing operation allows to use the same size for all entries in the A2M. An interested verifier can establish that the data item is, indeed, in the untrusted component's committed state by demanding the attestation given by a call to *lookup*.

Thanks to the collision-resistant properties of the hash function, for a given cumulative hash value, there is a single path of data items appended to a given log.

Attested Append-Only Memory. Using an A2M implementation within the trusted computing base, a protocol can assume that a seemingly correct host can give only a single response to every distinct protocol request. An A2M equips an untrusted host with a set of trusted logs. Each log has a unique identifier q and consists of a sequence of values, each annotated with a log-specific sequence number and an incremental cryptographic digest of all log entries up to itself. Only a suffix of the log is stored in A2M, starting from position $L > 0$ and going up to position $H \geq L$.

Interface. A call to *append(q,x)* takes a value x, appends it to the log with identifier q, increments the highest assigned sequence number H by 1, places it in it x, and computes the cumulative digest $d_H = h(H||x||d_{H-1})$, where $d_0 = 0$.

The *lookup* method takes a log number q, a sequence number n and a nonce z and return an attestation. This method indicates if the sequence number was unassigned ($n > H$), forgotten ($n < L$), assigned (in this case, the log value and the digest value are returned), or skipped (see below). An important fact is that the response given by the A2M is signed by it, and thus cannot be forged. A special call *(q,z)* executes *lookup* on the last value of the log.

truncate(q,n) allows the log to forget all entries with sequence numbers lower than n in log queue q, setting L to n. A call *advance(q,n,d,x)* allows log q to skip ahead by n values, using a computation similar to the one in *append* where d_{H-1} is replaced

by d, and finally to affects x. All the skipped slots will return a skipped status when concerned by a call to *lookup*.

A2M is based on a hardware solution, and provides an API which can be used as a tamper-evident log. The overhead of using tamper-evident log based protocols is then lowered, because logs are made tamper-evident by construction. However, using trusted hardware is far from being a general possibility in nowadays large scale systems, as they are not easily available for customers.

2.2.2.2 TrInc

TrInc [33] is another hardware-based accountability solution whose main contribution is to reproduce the features of A2M at a lower cost. In TrInc, a trusted hardware that consists essentially in a non-decreasing counter, and a shared session key, provides unique attestations. The authors prove that despite its light-weight characteristics, TrInc can reproduce, or improve previous protocols such as A2M and PeerReview.

One goal of TrInc, in comparison with previous works, is to minimize the additional communication overhead and the number of non-faulty participants required. To use TrInc, a participant has to attach a trusted piece of hardware, called a trinket, to its computer. This device is linked with the computer over an untrusted channel. This trinket provides attestations for messages that will be bind with a counter value, but they are further stored in untrusted memory.

When a message has to be send by the computer, it is associated to an attestation from the trinket. In this attestation, the trinket ensures that a counter value is associated to the message and, implicitly, that this value will never be reused for any other message. A trinket provides the possibility to create new counters. Thus, on a given trinket, each counter has a unique identifier. All trinket participating in a same system have the possibility to own the same shared symmetric key, related to the session, and stored in trusted memory, unexposed to untrusted parties.

Trinkets. Each trinket owns a unique identity I, and a public/private key pair (K_{pub}, K_{priv}) which are both provided by its manufacturer. The manufacturer also provides an attestation A that proves the values I and K_{pub} belong to a valid trusted trinket. A trinket can also tell how many counters it has created through a value M, which cannot decrease. A trinket also possess a FIFO queue containing the most recent counter attestations generated by the trinket allowing it to recover after a power failure. These attributes constitute the global state of a trinket, and are summarized in table III.

Each counter has some attributes (see table IV), which are its identity i, its current value c and its associated key K. The identity i is equal to the value M had when the counter was created. Counter c is initialized to 0, and can only increase. The key K contains a symmetric key to use for attestations of this counter.

TrInc API. An attestation $< COUNTER, I, i, c, c', hash(m) >_K$, for a message m, can be generated for two different reasons. If it binds the counter to a message, the

Notation	**Meaning**
K_{priv}	Unique private key of this trinket
K_{pub}	Public key corresponding to K_{priv}
I	ID of this trinket, the hash of K_{pub}
A	Attestation of this trinket's validity
M	Meta-counter: the number of counter this trinket has created so far
Q	Limited-size FIFO queue containing the most recent few counter attestations generated by this trinket

Table III – TrInc: Global state of a trinket

Notation	**Meaning**
i	Identity of this counter , i.e., the value of M when it was created
c	Current value of the counter (starts at 0, monotonically non-decreasing)
K	Key to use for attestations, or 0 if K_{priv} should be used instead

Table IV – TrInc: Per-counter state.

counter value is increased. If it is an attestation of the counter value, this value does not evolve.

A trinket can generate attestations based on the identity of a counter, the value that should replace the counter value, and the hash of the message that should be bound to the counter value. It is the same procedure that is applied to know the value of a counter. Attestations are signed using the shared session key, if it exists, or the private key of the trinket. Some other functions of the TrInc API allow the trinket to return its certificate, to create or to free a counter, and to receive a session key which will be affected to a counter.

When node A wants to send a message to node B, it first obtains an attestation from its trinket, and sends it along with the message to node B. It is also necessary for A to send its certificate $C = (I, K_{pub}, A)$, where I is the trinket identity, K_{pub} its public key and A an attestation that I and K_{pub} belong to a valid trinket. B can then learn A public key and verify that this is a valid trinket's public key. If the trinket uses a session key, B can check whether the attestation is really issued from this key.

Sessions. At the beginning of a session, the session administrator generates a symmetric key K. To allow a certain user to join the session, he asks that user for his trinket certificate C. It can then answer it with the session key through a message $\{KEY,\ K\}_{K_{pub}}$ that can only be decrypted by the user. At the same time, the user can initialize a counter from this message and set the session key associated. Any node that knows this key is allowed to check the node attestations.

SECTION 2.3
BAR-transformation protocols

Nysiad [34] presents a technique which transforms a distributed system which is crash-tolerant into a Byzantine-tolerant one. Relying on the principles of State Machine Replication (SMR) [65], each node is replicated on a given number of other nodes, called the replicas. The number of replicas per node is determined depending on the assumption of the maximum number of replicas that can be simultaneously faulty. The replicas run a replication protocol which ensures that they remain synchronized with the replicated node, named the primary. Nodes are supposed to execute a deterministic state machine that transitions in response to receiving messages or expiring timers. To maintain their synchronization, the replicas have to treat the messages the primary received in the same order, even in case of network failure. If a replicated node does not faithfully follow the protocol, its replicated state machine is stopped, or considered as crashed, which is a case that the original translated protocol can handle.

Nysiad architecture. In a t-guard graph, the state machine replicas of each node, which are also called its guards, are assigned to at least $3t + 1$ nodes, including the node itself. Furthermore, each two nodes have (at least) $2t + 1$ common guards, which are called the monitors of the two nodes. Nysiad makes the assumption that an upper bound

t of guards of a node can be Byzantine, and that message communication between non-Byzantine guards is reliable. The assignments of guards to nodes is made by a centralized, Byzantine-tolerant service called the Olympus. The Olympus certifies the guards of a host, and is involved only when the communication graph changes as a result of churn, or new communications pattern in the pattern.

In the following we present the three protocols that constitutes the basis of the SMR system run by guards, and the epoch protocol that the Olympus executes to reconfigure the guard graph in case of churn. The three protocols of the replicas are the following:

- *Replication protocol.* The guards of a host remain synchronized.
- *Attestation protocol.* Only messages corresponding to a faithful protocol execution are delivered to the guards of a node.
- *Credit protocol.* A node is considered as crashed by other nodes if it does not fairly process all its input, thus forcing a node to either process all its inputs fairly or to ignore all of them.

In the following, we first illustrate the various parts of the replication protocol. Then, we discuss the role of the Olympus.

Nodes state machine replication protocol. We illustrate in the following the functioning of the three sub-protocols forming the state machine replication protocol.

Replication protocol. This protocol ensures the guards (the replicas) are synchronized on the state of the node they replicate. To achieve this, the replicated node uses a reliable ordered broadcast for communication with its guards. This protocol works as follows. When a node h_i wants to send an input message m to its guards, it first sends an order request message to its guards containing the hash of m. Guards reply with an order certificate message, which includes a sequence number c that they maintain on behalf of the replicated node h_i . The node h_i is able to collect (at least) $n_i - t$ order certificates, n_i being the number of guards of h_i. A consistent sequence number in $n_i - t$ order certificates constitutes an order proof for the message m to be delivered. At this point, h_i delivers the messages m to its own running replica of its state machine, and sends the message m with the order proof to all its guards. If a guard assesses the order proof is valid, it delivers the message m to its running replica of h_i's state machine, and gossips it with the other guards of h_i to ensure that if a non-Byzantine guard delivers a message, also all other non-Byzantine guards will be able to do so.

The reliable ordered broadcast protocol ensures synchronization among guards (the replicas) of a node, but it does not preclude to a node the possibility of forging or ignoring inputs. The attestation and credit protocols described next provide the complementary pieces to take these two misbehaviors into account.

Attestation protocol. Attestations prevents a host from forging invalid messages that would be processed by all its replicas. Checking the validity of message events is slightly different than checking timer events. Suppose node h_i wants to send a message

m to node h_j. Each guard of h_i implements a RSM and can produce an attestation for m that it has to send to node h_i. When h_i has received at least $t + 1$ such attestations, it can send them to node h_j. Node h_j is then able to broadcast m along with the attestations from the guards of h_i to its own guards, who will then trust the reception of this message. When a timer event occurs, node h_i needs to collect $t + 1$ attestations, including its own. Doing so, a node cannot produce events at a rate higher than the one of the fastest correct host. In an asynchronous system, doing so prevents nodes from denunciating each other for non-responsiveness, for example using a ping protocol, without waiting for the responses of nodes. Collecting the attestations slows down the eviction process and avoids false accusations.

Credit protocol. The aim of this protocol is to force nodes to consider all inputs and produce the expected output, or to ignore all of them. For example, in gossip applications selfish nodes would typically try to avoid to take into account some messages to avoid retransmitting them. To avoid this, before sending new messages, a host must first have obtained credits from its guards for the previous inputs it received. If a host does not produce any output, it will eventually be considered as a crashed host. Nodes use certificates that order several inputs at a time.

Epoch protocol. The Olympus is in charge of producing signed *epoch certificates* for hosts which can then convince the receivers of a message of their validity. An epoch certificate contains the host identifier, the set of identifiers of all its guards, the epoch number (to avoid replay attacks), and a hash of the final state of the node in the previous epoch. The Olympus does not need to know the underlying protocol that nodes are executing, and is completely independent of the three previous protocols we presented. Several triggers can provoke the changing of the guards of a node among which the failure of a guard (detected by means of ping messages), or the emission of a first message for a given host because two nodes must have at least $2t+1$ guards in common.

When the changing of a node's guards occur, each guard of a node h_i sends a *state certificate* containing the current epoch number and a secure hash of its current state to h_i. When h_i has received $n_i - t$ such certificates, h_i sends the collection of certificates to the Olympus. In response, the Olympus chooses new guards for h_i and begins a new epoch, sending a new certificate to h_i. Upon receiving its new certificate, h_i informs its new guards of their role by sending them its signed state and its certificate. The guards have also ways to check the validity of the assignment. When h_i generates a message for the first time for another node h_j, the Olympus has received $n_i - t$ certificates from the guards of h_i and then asks h_j to change its guards.

SECTION 2.4
Summary

In the previous sections, we presented the existing works that try to limit the impact of rational deviations on content dissemination, or works closely related to it. In this section, we analyze the main needs of a gossip-based content dissemination protocol. We then explain why previous works are not satisfactory in presence of collusions of

rational nodes.

2.4.1 Requirements

In this section, we present the main concerns a system designer should have in mind when building a content-dissemination protocol based on gossip and tolerating rational collusions.

Efficiency of the dissemination. First, as any content dissemination protocol, information should be quickly disseminated among the nodes. Usually, two metrics are used to determine this efficiency: latency and throughput. The latency is equal to the delay between the emission of a new update from the source and its average reception time by the nodes. The throughput is equal to the quantity of data that can be emitted by the source during a period of one second without perturbing the reception by nodes. The design choices, e.g., the choice of using an unstructured or a structured overlay, may impact these metrics. If the protocol is based on random associations, it is necessary to control the randomness of each node's associations.

Scalability and churn. The dissemination protocol must be able to propagate data efficiently among various quantity of nodes, starting from some hundreds nodes to several thousands. For example, the streaming of live events, like the Olympic Games, may interest a large quantity of nodes located all over the world. Being able to tolerate a high number of peers is an important requirement. In addition, the membership may evolve abruptly. If numerous nodes massively join a gossip session, they should be served quickly with content, and old nodes should not suffer from this massive arrival. When massive departure occurs, the nodes that decided to stay should not suffer either.

Incentives to follow the protocol for individual nodes. In the BAR model, rational nodes are interested in decreasing their contribution to the protocol while maximizing their benefit. Several mechanisms have been designed to encourage nodes to contribute. The principle of balanced exchanges, or tit-for-tat, where nodes cannot receive more updates than they contribute in return, is probably the more immediate one. However, other mechanisms exist, and have been presented in the related works of this section. Protecting the protocol from individual rational nodes is an important prerequisite, because this kind of deviation is easy to implement in practice and highly tempting for participating nodes.

Incentives to deter rational coalitions. When assuming selfish behaviors, it is immediate to consider them followed by individuals. However, in presence of mechanisms that aim at limiting them, nodes may also be tempted to make coalitions to protect themselves from being detected and still contribute less than the average. It is also possible for nodes to prefer to interact inside their coalition because of better communications, e.g., if the nodes are geographically close, or if they are part of a high-bandwidth low-latency local network. However, the protocol has to be built in such a way that participating in a coalition does not prevent nodes from also executing their tasks. This way, coalitions would not affect the quality of service of the whole dissemination system.

Forcing the detection of deviations. As rational collusions could still happen it is important to detect them. To do so, mechanisms can be designed, however one has to be sure that nodes would denounce coalitions. If it is costly to do it, rational nodes would avoid to do it. In addition, a coalition has to be observable from the point of view of nodes outside of it. To reach these goals, the randomness of associations is a good advantage, but still a node has to be threatened to denounce other nodes. Knowing if a node has been able to observe a deviation is then important in order to later take sanctions.

2.4.2 Drawbacks of existing solutions

We summarize here why the existing systems presented so far are not satisfactory against coalitions of rational nodes. When possible, we detail which strategies the coalitions could use.

BAR Gossip [22]/FlightPath [23]. As seen previously, BAR Gossip relies on tit-for-tat exchanges of updates of data, which has scalability limitations. First, the data source has to ensure that packets are spread evenly across the system by sending data to a fixed proportion of nodes, which is getting difficult as the number of nodes increases, and by sending different packets to different nodes. In addition, it requires the source and all nodes to have full membership knowledge in order to allow random partner selections. These restrictions affect scalability when the source has bounded upload bandwidth.

In BAR Gossip and FlightPath, colluding groups cannot completely disrupt the protocol, but they can limit their participation. Indeed, any node has the choice not to initiate optimistic push exchange requests, and not to answer positively to them. Colluding nodes, obtaining updates off the record, are free to apply a passive/decline strategy. Two other aspects of the protocol also suggest that correct nodes may suffer from the presence of colluding nodes in the system during balanced exchanges. If a node is part of a colluding group where updates are immediately distributed to everyone, and if for a given round it already obtained all available updates, then it will not initiate a balanced exchange. The direct consequence is that colluding nodes will less frequently initiate balanced exchanges than correct peers. The second point is based on a property, proved in the paper's demonstration, that tells that a rational node has no interest in lying about the updates it owns. Due to the tit-for-tat policy, used in balanced exchanges, a node receives a new update only if it can provide one in return. Thus, the number of updates exchanged in a transaction that includes a "rich" node is small. It results that, when they exchange with colluding nodes, correct nodes see their average benefit decrease.

The scalability of BAR Gossip is limited by three factors. First, the full membership has to be known from every node. Second, balanced exchanges, known as tit-for-tat or symmetric, exchanges, are known to limit the efficiency of the content dissemination [24]. Finally, the source of the streaming session has to send updates to a constant fraction of the audience.

van Renesse et al. 2008 [24] In this article, when the maximum upload rate of the system is greater than a threshold, then nodes that did not upload enough data ,

according to a global auditor decision, are expelled from the system.

Local auditors, located on colluding nodes, may not be able to denounce any other colluding node during the study of their history. The underlying problem is that the history of a node is not verified, and could perfectly not correspond to the reality of its exchanges. If a colluding node tamper with its history, saying that it exchanged with one accomplice, it will fool the auditing protocol, which searches nodes that upload less than a threshold. Thus, colluding nodes are able to protect themselves from audits, even if they do not participate actively in the system.

Another point is the utilization of thresholds concerning the minimum upload factor of each node. The drawback of this solution is its number of false-positive detected. More specifically, the differences between the upload factors of two correct nodes may be important, even in systems where all nodes are correct. Thus, correct nodes that suffered from a bad position in the overlay may be expelled because they were detected as rationals.

LiFTinG [26]. Similarly to the preceding method, the score based punishment approach of LiFTinG can potentially punish a correct node that was not allowed to participate sufficiently, due for example to limited connectivity, message losses and bandwidth limitation. The experimental results show that after 30 seconds, 12% of correct nodes were expelled and 14% of rational nodes were not. This rate of false-positive could be even more important if rational nodes were to incorrectly denounce other nodes in order to protect themselves. In this case, either correct nodes would be evicted instead of rational nodes, or the system designer would be force to avoid the eviction of any node.

Talking about colluding nodes, the research report introduces three ways in which the direct cross-check can be fooled. To counter them, the randomness of partner choices is controlled through a statistical study of the associations made during intervals of h seconds. During this duration, peers can join or leave at any moment, it is then expectable that associations with old nodes will be more frequent than those with peers that were not in the system since a long time. No mechanism is employed in Lifting to consider the utilization of statistical check with churn events. It results that using a pure statistical check for controlling associations may produce accusations of correct nodes.

Let us now suppose that, in a certain extent, nodes choose partners randomly. A surviving problem is that if colluding nodes, after having selected at random a partner, recognize a node that belongs to their own colluding group they are still allowed to deceive the system applying exactly the methods presented in the paper. In a system where 10% of nodes collude together, in average one in ten exchanges will potentially deviate from the protocol. The entropic check limits these misbehaviors but do not prevent them from occurring regularly.

Finally, upon reception of a proposal, a node is free to request or not an update. However, colluding nodes may obtain updates illegally and thus not wish to participate in their

propagation. If a node obtained an update off the record, it will refuse it when receiving a proposition, because it would imply to diffuse it further. The direct consequence of this behavior is that colluding nodes will not participate in the propagation of updates they obtained illegally.

Nysiad [34]. Nysiad assumes that selfish nodes are Byzantine and try to use a Byzantine-tolerant approach to detect them. However, this approach has an important limitation. Since basic distributed computing primitives such as consensus cannot be implemented if more than a third of the audience is Byzantine [57], this approach limits the number of nodes that can be simultaneously faulty. In addition, Nysiad is also not completely decentralized as a special entity named the Olympus is in charge of affecting guards to nodes.

2.4.3 Conclusion

The objectives of this chapter were to present the more relevant publications of the domain that are related to gossip in presence of rational nodes. We first presented rational-resilient protocols that are either based on establishing Nash-equilibriums which encourage nodes to behave correctly or use auditing techniques to detect and punish the deviations of nodes. These approaches have scalability issues, and can suffer from simple collective rational deviations. It should be noted that we performed experiments where these protocols are put in presence of collusions. The results are reported in section 3.4.2.

The second part detailed generic accountability approaches that allow a system to securely record the behavior of nodes and later compare them with a correct execution. Accountability approaches are either entirely software-based, or use specialized trusted-hardware. These interesting methods could be used on top of gossip, but they assume that nodes will voluntarily verify each others through audits, which is not true when we assume that all nodes may behave rationally.

We then presented a protocol, Nysiad, which considers all deviations as Byzantine, and builds a Byzantine-resilient version of a protocol. However, as all protocols that aim at handling Byzantine faults, this protocol is not completely decentralized and limits the possible number of faulty nodes to one third. In our assumption, we desire a system where nodes can all behave rationally, which is closer to the reality.

Chapter 3

AcTinG: accurate freerider detection in gossip

Contents

3.1. Introduction to $AcTinG$ 46
3.1.1. Principal ideas 46
3.1.2. System model 47
3.1.3. Protocol overview 48
3.2. $AcTinG$ 50
3.2.1. Protocol details 51
3.2.2. Membership protocol 51
3.2.3. Partnership management 53
3.2.4. Audit protocol 54
3.2.5. Update exchanges 56
3.3. Proofs 56
3.3.1. Risk versus gain analysis 56
3.3.2. Resilience to (colluding) rational nodes 58
3.4. Evaluation 64
3.4.1. Methodology and parameters setting 65
3.4.2. Impact of colluders 66
3.4.3. Bandwidth consumption 68
3.4.4. Resilience to massive node departure 68
3.4.5. Scalability 69
3.5. Conclusion 71

In this chapter, we present $AcTinG$, a novel gossip protocol that includes accountability techniques that are directly included in a content dissemination protocol. This approach allows this protocol to detect coalitions of rational nodes, and allows the verifying tasks of nodes to be verified.

This chapter is organized as follows. Section 3.1 describes the principal ideas behind the design of $AcTinG$. Section 3.2 provides the detailed implementation. Section 3.3 provides theoretical probabilistic results that show that rational nodes are encouraged to follow the protocol correctly. Finally, Section 3.4 evaluates $AcTinG$ based on several aspects. Section 3.5 summarizes and concludes this chapter.

SECTION 3.1
Introduction to $AcTinG$

In this section we describe the principal ideas behind the design of $AcTinG$. Then we detail the system model we consider, and present a protocol overview.

3.1.1 Principal ideas

In order to reach the requirements we presented in Section 2.4, we propose the mechanisms that follows.

Asymmetric exchanges. Allowing nodes to receive more updates than they can provide in return during an exchange is an important need of content-dissemination systems. It has been observed that applications based on symmetric exchanges have limited scalability. BAR Gossip [22], for example, needs a source that broadcasts to 5% of the audience to balance this lack of scalability.

Pseudo-randomness. Associations between nodes must be random for several reasons. First, to ensure a good dissemination of updates. Second, because if a node can predict its future partners it would be able to decide if the moment is adequate to deviate. However, allowing associations to be completely random allow somehow nodes to deviate without detecting abnormal behaviors accurately. To obtain the best of both solutions, we decided to use pseudorandom associations.

Secure logs and associations. To detect coalitions, nodes will maintain a secure log that will accurately register the behavior of a node during several rounds, and provide enough context to check its correctness. The bigger the number of rounds included in a log the smaller the probability that a rational node will only interact with accomplices. If a correct node observe a log, it should be able to detect coalitions.

Audits: verifiability without predictability. To verify each others, nodes should have access to the logs of their partners. However, nodes could be tempted to avoid receiving a log, or to avoid verifying it. Thus, the execution of audits should be verified as the other steps of the protocol, and should not be predictable (or nodes would deviate only when it would be safe).

3.1.2 System model

We consider a system with N nodes. Each node is uniquely identified, e.g., using a hash value of its IP address. We consider two classes of nodes: *correct* nodes and *rational* nodes. Correct nodes follow the protocol. Rational nodes are defined as in [22] extended with the notion of collusion: they aim at getting the content (i.e., while 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 [13]. 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 misbehavior 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 node as the node that is disseminating a given content. We assume that each content is disseminated from a single source node 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 organize themselves in colluding groups of arbitrary sizes.

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

As in [22] and [23], we assume that nodes maintain clocks synchronized 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 [29, 31, 32, 33]. We build on the secure log presented in [29]. We assume that nodes can join and leave the system (gently or by crashing) at any time.

3.1.3 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 3.2.1.

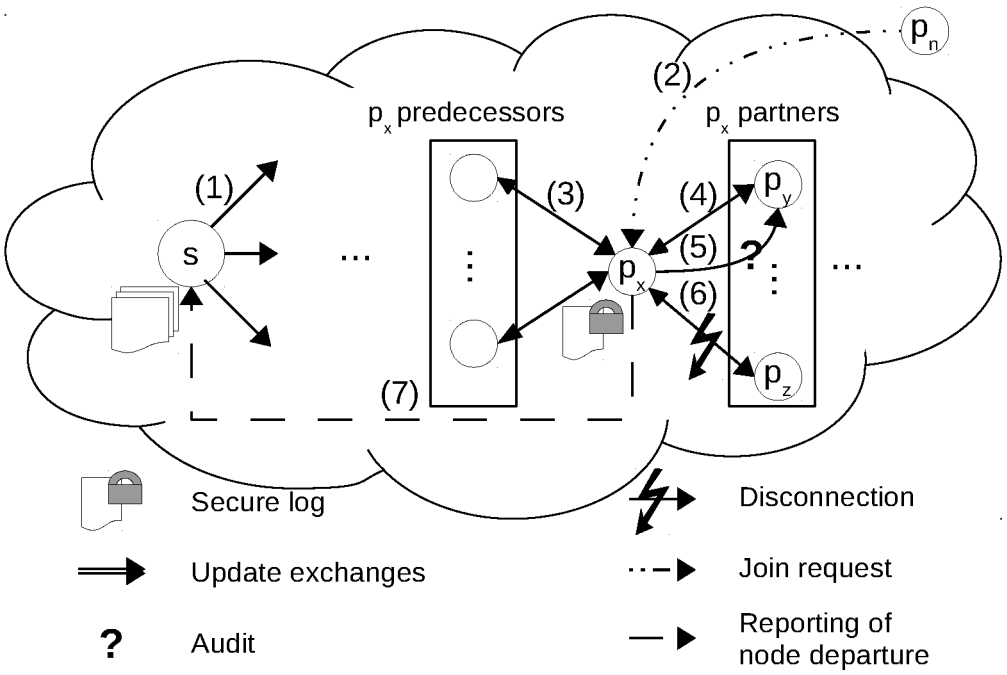


Figure 3.1 – Overview of $AcTinG$.

Figure 3.1 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 3.2.2 (arrow 2 in the figure). In the middle of the figure, a node p_x, which characterizes 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 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 maximize 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 3.2.3. 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 [33]. 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 3.2.4.

"Is not 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 misbehavior 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 3.2.2). 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 partners to collect evidence about the effective unresponsiveness of p_z (as described in Section 3.2.2). 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.

Summarizing, 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*.

SECTION 3.2
$AcTinG$

In this section we present the details of $AcTinG$ which implement the principles previously presented.

3.2.1 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 to a small set of randomly chosen nodes. To get updates, each node initiates, and maintains partnerships with f 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 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. Specifically, this audit checks the behavior of the new partner for the last $Period$ rounds. 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 3.2.2), which allows dealing with new nodes joining the system, nodes leaving it and unresponsive nodes. Then, we present the partnership management (Section 3.2.3), the audit (Section 3.2.4) and the update exchange protocols (Section 3.2.5), which allow handling the partnerships between nodes auditing their logs and exchanging updates between partners, respectively.

3.2.2 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. However, due to the lack of space we describe in this paper only an overview of this protocol.

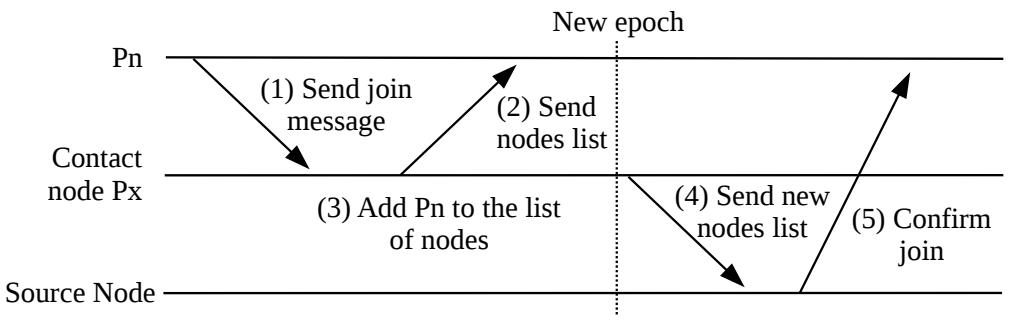


Figure 3.2 – Arrival of a new node.

Node arrival The arrival of a new node follows the sequence of messages depicted in figure 3.2. In this diagram, 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 3.2.3 and contacts each of these nodes to start a new partnership. As such p_n is ready to start receiving the content. At the beginning of the new epoch, each node, including node p_x informs the source of the arrival of new members that have contacted him (step (4)). Using theses messages, the source confirms to the new members their integration in the system and updates the membership list (step (5)).

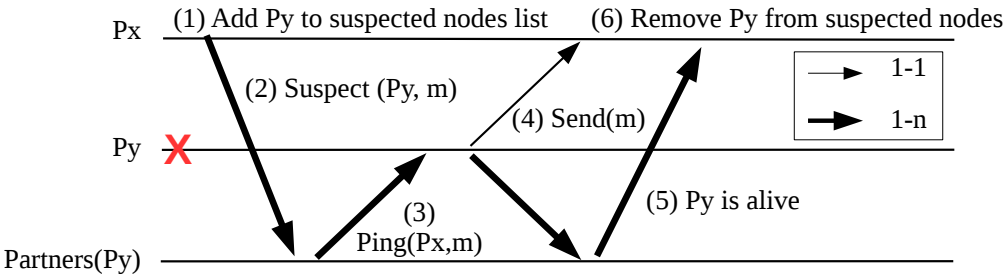


Figure 3.3 – Handling of an omission failure.

Node departure and omission failures If a node p_x is expecting a message from one of its partners p_y for too long[1], it suspects p_y of omission failure as depicted in the diagram of Figure 3.3. 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 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 sent by p_y partners.

In order to make sure that a rational node will never suspect a correct node in order to avoid initiating or accepting an interaction with it, 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 instead of initiating or accepting an interaction with it.

[1]Delays for node suspicion are configured in an implementation dependent manner

Figure 3.4 represents how the departure of a node is handled. In this illustration, node P_x tries to interact with node P_y which is non responsive, either because it left the system or because it does not want to answer, or suffer from network issues. Node P_x thus adds P_y to its list of suspected nodes (step (1) in the diagram), and signals to the nodes that P_y should interact with that it is not responsive (message (2) in the diagram). These partners have to ping node P_y (message (3) in the diagram), and confirm to node P_x that it is not responsive (message (4) in the diagram). Upon reception of these messages, node P_x can store them, and remove P_y from the list of nodes it is maintaining (step (5) in the diagram).

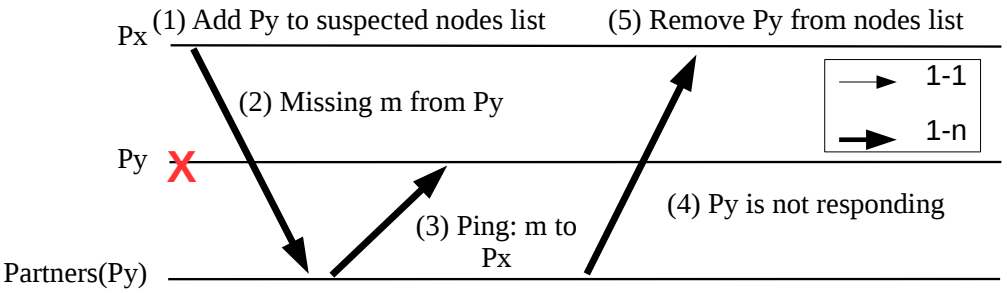


Figure 3.4 – Handling of a node departure.

Membership list update Periodically, nodes that served as contact nodes for others send their list of new nodes to the source node. Furthermore, nodes that hold the evidence of the departure of one of their partner send this evidence to the source node. The latter updates the membership list and sends the updated list at the beginning of each epoch to its partners along with the content. In order to fasten the removal of dead nodes from the membership list of nodes, an optimization consists of letting the source disseminate the list of dead nodes at the beginning of each round 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 in the case where new partnerships have to be established before the new epoch. In order to preserve nodes that are participating to a content dissemination session from the massive arrival of new nodes, which may consume their bandwidth, we adopt the optimization defined in [23], which allows splitting the load between old nodes and new nodes. Specifically, this optimization forces new nodes to establish partnerships with a limited proportion of old nodes and with other nodes that arrived during the same epoch as themselves.

3.2.3 Partnership management

Each node p_x has to maintain partnerships with f other nodes, which are selected with the PRNG function seeded with a deterministically computed seed (e.g., the round number concatenated with p_x's public key) among the non-suspected nodes of the last membership list. This process is depicted in the diagram of Figure 3.5. If a selected node is not responding, node p_x has to propagate a suspicion, and once the suspicion is confirmed, p_x is allowed by the source to find a new partner. Every $Period$ rounds, a

node p_x breaks the partnerships it initiated with f nodes, without informing its partners, which know when the partnerships are supposed to come to an end (step (1) in the diagram). 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 $Period$ previous rounds, and asking them to return their own log of the last $Period$ 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 3.2.4. 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 or not 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 computed number indicates that the audit must take place, p_x contacts p_y partners, and ask for their logs.

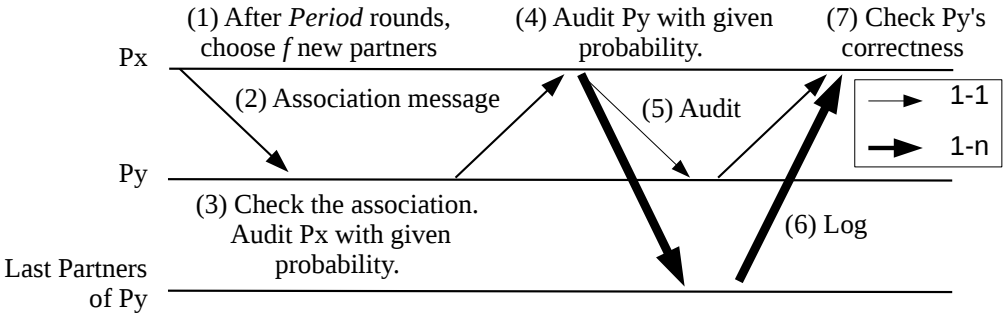


Figure 3.5 – Establishment of new associations between nodes, which may imply audits.

3.2.4 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 log 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 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 content of 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's log, 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, the last time it changed its partners.

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[2]. 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.

[2]We recall that the source is assumed to be a correct node.

3.2.5 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 3.6. 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 those 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. After receiving the updates, each partner 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 these updates.

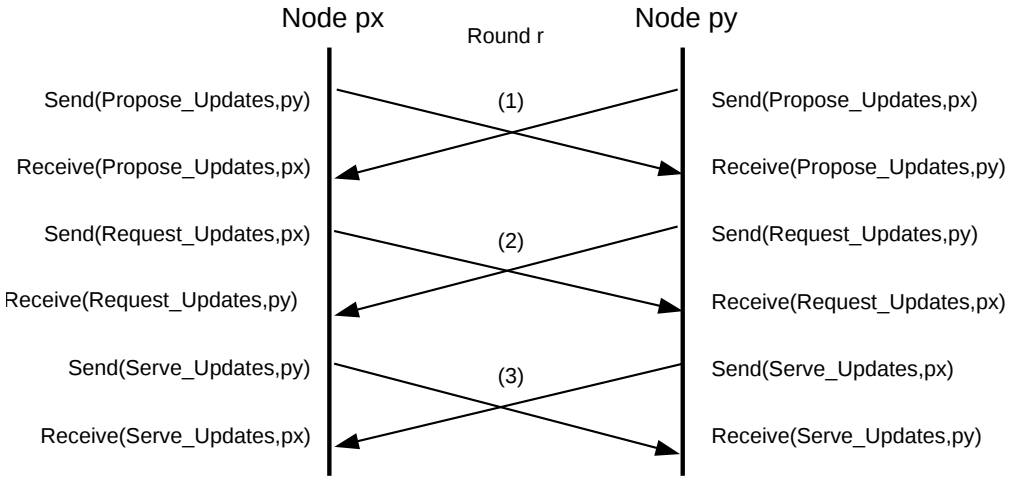


Figure 3.6 – Update exchanges between nodes.

SECTION 3.3

Proofs

3.3.1 Risk versus gain analysis

The aim of this section is to demonstrate that rational nodes will not deviate from the $AcTinG$ protocol, because audits will detect deviations with high probability, and because the estimated gain of collective deviations is low.

First, we evaluate the risk that two colluding nodes would take by deviating, for example when interacting as predicted by the protocol, but without logging the updates they receive, or send. This deviation seems to be the most rewarding one for colluding nodes. We define the risk as the probability that such a deviation would be detected, and denounced by an audit.

In the following rounds, this deviation would allow them not to share with correct nodes, possibly several times, the updates they obtained unofficially, thus to save their upload bandwidth. However, this strategy will force colluding nodes to receive a second time the updates they received unofficially. Exchanging the multiple uploads of an update for one additional download can already be seen as interesting. However, trading upload bandwidth for download bandwidth, can also be typically considered interesting in ADSL environments.

We now calculate the probability that a deviation such as we described is discovered by an audit. If any of the two colluding nodes is audited during the time where the exchange is contained in their log, 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 $fanout$ partnerships with other nodes, which are changed after $period$ rounds. Let P_{audit} the probability that a node audit 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 does not realize the audit. In average, each of the two nodes cumulate $\frac{2 \times fanout \times RTE}{period}$ partners during the time the deviation is visible. Finally, we obtain that the risk that a deviation is detected is equal to

$$\left\{1 - \left(\frac{C}{N} + \left(1 - \frac{C}{N}\right) \times (1 - P_{audit})\right)^{\frac{2.fanout.RTE}{period}}\right\}^2$$

Let us suppose that two colluding nodes exchanged an update, and did not declare it in their logs. We now want to determine the number of interactions that the rational node can hope that it will avoid to send it to correct nodes. To do so, we use a program, and present its code in figure 3.7.

The principle of this program is that during each of the RTE rounds that follow the round at which the deviation occurred, $2 * fanout$ interactions happen. Each of these interactions, has a probability $\frac{C}{N}$ to involve another incorrect node. When it is not the case, this other node owns the missing update with a probability equal to $(2 * fanout)^{round_id}$, according to a traditional result of gossip. When the rational node receives the update from a correct node, it will have to share it with its future partners.

From the output of this program, 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 have to multiply this proportion by the probability that a rational node meets another rational node to be able to execute this deviation, which is $\frac{C}{N}$, and the proportion of the bandwidth that is consumed by updates, which is roughly equal to $\frac{3}{5}$.

$$gain = \frac{C}{N} \times proportion_saved_sends \times \frac{3}{5}$$

Computing the risk, and the gain, with the values of the parameters used in the protocol,

```
saved_sends = 0
for round_id in 1..RTE do

  for association_id in 1..2*fanout do

    if random() > C/N then
      if random() < min((2 * fanout)^round_id), N)/N then
        received_update = true;
      else
        saved_sends = saved_sends + 1;
      end if
    end if
  end for
  if received_update then
    break;
  end if
end for
return saved_sends
```

Figure 3.7 – Pseudocode of the program that is used to estimate the number of time a colluding node avoids to send an update.

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 node are exposed with a high risk each time they execute the deviation, and can only hope for a very small benefit. Finally, we can say that according to the BAR model, rational nodes will not deviate from the protocol.

3.3.2 Resilience to (colluding) rational nodes

In this section, we analyze the major steps of the protocol and show that rational, possibly colluding nodes do not have any interest in deviating from these steps. For each step, we consider all the possible deviations, and provide the incentives that make rational nodes follow the protocol.

We present the rational deviations associated to the integration of a new node in the system, and the incentives that make nodes stick to the protocol.

- Fig 3.2 Step 2. To join a content dissemination session, a new node p_n sends a join request to a contact node p_x (Step 1), which then adds p_n to its list of new nodes.
  - *Rational deviation.* Node p_x does not add p_n to its local list of nodes, thus avoiding to inform other nodes about the arrival of a new node, saving resources and increasing the probability for its group of colluders to be served by other nodes.

  - *Incentive.* During the future rounds, any node auditing the node p_n will see the identity of p_x, its contact node, in its log, and ask for the log of p_x and check that it informed its partners about the arrival of a new node in the system. If it is not the case, the node will be evicted.

- Fig 3.2 Step 3. Node p_n then replies with the list of active nodes of the current epoch.
  - *Rational deviation 1.* Node p_x ignores the request that p_n sent, and does not reply with the list of active nodes to node p_x saving its bandwidth.
  - *Incentive.* If the new node does not receive a reply, it will continue to send the request periodically, and more and more frequently. As a rational node wants to preserve its bandwidth, it will always consider join requests immediately.
  - *Rational deviation 2.* Node p_x does not send the correct list of alive nodes in the system (e.g., it could send a truncated list, to hide the nodes it colludes with).
  - *Incentive.* If the new node is audited in the RTE rounds that follow its arrival, the log of its contact node will be verified, resulting in the eviction of the latter if the list of active nodes it sent was not correct.

- Fig 3.2 Step 4. At the beginning of the next epoch, node p_x sends to the source of the stream the list of new nodes that contacted it. It then logs the acknowledgement of the source to prove that it realized this step.
  - *Rational deviation.* Node p_x does not send the correct list of new nodes to the source of the stream, or does not send it at all.
  - *Incentive.* The node p_n that is willing to join the session is expecting to receive a message from the source, confirming its integration in the list of nodes. If it does not receive this message after some time, it will send join messages to the contact node more and more frequently, consuming its resources. Thus a rational contact node will immediately transfer the source node about the arrival of a node in the system.

We then proceed similarly to prove that the omission failure, and node departure, handling protocols will be observed.

- Fig 3.4 Step 1. If a node p_x is expecting a message from one of its partners p_y for too long, it adds it to its list of suspected nodes.
  - *Rational deviation.* Node p_x does not add p_y to the list of suspected nodes.
  - *Incentive.* Audits check that a node sent, and received, all the messages an association implies, and, if it is not the case, that during the following rounds the node informed the other nodes about the suspected nodes. Otherwise, the audited node will be declared incorrect. Thus, as p_x can not predict whether it will be audited or not in futures rounds, it will emit the necessary suspicion messages regarding p_y.

- Fig 3.4 Step 2. Then, it sends a suspicion message to the other partners of p_y.
  - *Rational deviation 1.* Nodes p_x does not send a suspicion message, for example to protect an accomplice.
  - *Incentive.* Audits check that a node sent, and received, all the messages an association implies, and, if it is not the case, that suspicion messages were sent. Otherwise, the audited node will be declared incorrect. Thus, as p_x can not predict whether it will be audited or not in futures rounds, it will emit the necessary suspicion messages regarding p_y.
  - *Rational deviation 2.* Node p_x, and the partners of node p_y, collectively decide to exclude node p_x from the system, even though it is correct.
  - *Incentive.* Suspecting a node is made more costly than a normal interaction, and the expected benefit of excluding a correct node is low, because correct nodes are a source of updates for other nodes. In addition, the probability for such a situation to occur is extremely small (1 out of 10 millions, when 10% of the nodes collude).
- Fig 3.4 Step 3. Then, each of p_y's partners pings p_y.
  - *Rational deviation.* The partners of node p_y does not ping it, as the node p_x asked them.
  - *Incentive.* If node p_x does not receive a message from a partner p_z of node p_y, it will contact the partners of node p_z and ask them to obtain the missing answer from p_z. Thus node p_z will have to answer, if he wants to avoid being suspected, and to consume more bandwidth than if it answered directly to node p_y. Thus, a rational node will always immediately execute the ping procedure.
- Fig 3.4 Step 4. The partners of p_y then reply to p_x with a signed message certifying whether p_y replied to the ping message or not.
  - *Rational deviation.* The partners of node p_y lie saying that it is responding, either to save bandwidth or to protect it.
  - *Incentive.* Nodes cannot lie because ping messages, and their answers, contain log entries which testify the time at which the messages were sent by nodes. Thus, to say that a node is responding, another node needs to communicate with it, and rational nodes have no way to protect their partners.
- Fig 3.4 Step 5. After having received the confirmation that node p_y is not responding, node p_x removes p_y from its list of nodes.
  - *Rational deviation.* The node p_x does not remove p_y from the list of nodes in the system, thus avoiding it to be evicted from the system.
  - *Incentive.* The partners of node p_y sent messages that can not be forged, and if all of them indicate that the node is not responding, then p_x would be evicted if an audit detected that it did not removed p_y from the list of nodes in the system.

- Fig 3.3 Step 4. Node p_y is informed that node p_x is expecting a given message m. It then sends this message to node p_x and to its partners.
  - *Rational deviation.* The node p_y does not send the message m that was not received to the partners of node p_y and to node p_x.
  - *Incentive.* If the node p_y does not send the expected message, the suspicion will not be released, and it will eventually be evicted from the system, as other nodes will refuse to interact with it.
- Fig 3.3 Step 5. The partners of node p_y confirm to node p_x that they received the message from node p_x, joining to the message the authenticator that node p_y sent along with message m.
  - *Rational deviation.* The partners of node p_y lie to node p_x, saying that it is not responding, to evict it from the system.
  - *Incentive.* Is it not clear if this deviation is rational, however, the probability that such a deviation occurs is small. For example, if nodes maintain 3 partners, and 10% of the audience colludes, then such a deviation can occur with probability 10^{-6}. If one node replies that node p_y is responding, and proves it with the authenticator, then node p_y will not be evicted from the system.
- Fig 3.3 Step 6. Node p_x receives the message m it was expecting, and remove p_y from its list of suspected nodes.
  - *Rational deviation.* The node p_x does not remove p_y from the list of suspected nodes, aiming at isolating it from other nodes.
  - *Incentive.* Evicting correct nodes does not bring a clear benefit, because correct nodes are those that propagate updates. In addition, if an audit occurs, it will appear that the node p_x is suspecting a node even though it received messages that proved that the node sent the message it was supposed to send. If this node does not appear in the node's log, for example because it does not want to log it, then the costly suspicion procedure has to be executed once again, which clearly will degrade the resources consumption of node p_x.

In the following, we show why nodes execute the partnership protocol.

- Fig 3.5 Step 1. Every $Period$ rounds, node p_x stops exchanging with its f partners and deterministically selects f new partners using a pseudo-random number generator seeded with a deterministically computed seed (e.g., the round number concatenated with p_x's public key).
  - *Rational deviation 1.* Node p_x tries to establish a new partnership with nodes whose IDs are other than those computed using the PRNG function, e.g., to interact with colluders.
  - *Incentive.* A rational node p_x will never select such nodes as it risks eviction during the next audit. Indeed, node p_x can be selected by a correct node,

say p_0, in the future, which will verify whether p_x effectively selected the nodes it was supposed to interact with by examining its log. If node p_0 detects such a deviation, it will expose p_x.

  - *Rational deviation 2.* Node p_x tries to establish less than f partnerships to save bandwidth.
  - *Incentive.* The same incentive as above holds.

- Fig 3.5 Step 3. When node p_x proposes to start a new partnership to node p_y, p_y checks that p_x had to contact it by rerunning the PRNG. If the check succeeds, p_x and p_y will exchange during the next round.
  - *Rational deviation 1.* Node p_y does not reply to the proposition message sent by p_x.
  - *Incentive.* If p_x does not receive a reply after it sent its proposition, it will suspect p_y. Not doing so would expose it during the next $Period$ rounds, and prevent it to interact with any correct node. In order not to be evicted, p_y will answer to the partnership request.
  - *Rational deviation 2.* Node p_y replies without verifying the legitimacy of p_x's request, which could happen if nodes p_y and p_x are colluders, and try to protect themselves.
  - *Incentive.* We distinguish two cases: (a) p_x is correct, and (b) p_x is rational and the verification p_y has to perform should not pass. If p_x is correct and p_y attests that p_x has passed the verification without effectively performing it there is no way to detect that p_y is behaving rationally. However, as p_y does not effectively know whether p_x is correct or not, p_y risks eviction as well as p_x. Hence, p_y will prefer to verify whether p_x was supposed to contact it or not. Instead, if p_x is rational and p_y attests that p_x passed the verification without performing it, p_y risks eviction. Indeed, if one of the following nodes among those that will be contacted by p_x or one of the correct nodes that will contact p_x during following rounds, say p_k, is correct and finds out that p_x behaved rationally through an audit, p_k could use the attestations sent by p_y and which are in p_x's log to prove that p_y behaved rationally. This will result in the eviction of p_y. As rational nodes do not want to be evicted, they do not attest for the correctness of a node without performing the corresponding verifications, and colluders do not protect themselves.

- Fig 3.5 Step 3, 4. Node p_x deterministically decides whether to audit p_y's log or not.
  - *Rational deviation.* A node does not audit its partner when it should.
  - *Incentive.* When this node will start new partnerships in future rounds, its log will possibly be audited. During this audit its future partner recomputes the boolean that indicates whether the node should have audited its previous partners or not. If the node did not perform the audits while it was supposed to do so or did not perform them correctly, it will be exposed by its partner. As the node cannot predict the occurrence of its future audits, it will audit

its current partners, and will contact all their own partners to get their logs, following the results of the deterministic computation it has to perform.

- Fig 3.5 Step 5. If the result of the computation implies that p_x must audit p_y, the former contacts all p_y's partners and ask for their log.
  - *Rational deviation.* Node p_x does not contact the right set of nodes to obtain their logs
  - *Incentive.* The same incentive as above holds.
- Fig 3.5 Step 6. The partners of p_y reply with their logs.
  - *Rational deviation.* Nodes do not reply with their logs, either to save their bandwidth, or to protect node p_x from being declared faulty.
  - *Incentive.* A node that does not send its log will be suspected of omission failure, and will have to handle a suspicion procedure, which is costly, and will eventually have to send the missing log to delete this suspicion. A rational node has no interest in refusing to send its log.
- Fig 3.5 Step 7. Upon receiving the logs of p_j and p_i, one of p_y's partners, p_x checks that the two nodes interacted correctly.
  - *Rational deviation 1.* Upon receiving the logs of p_j and p_i, p_x claim that they are correct without effectively performing the necessary verifications to save resources.
  - *Incentive.* If p_x skips some or all the verifications described in the audit protocol, its risks to be exposed by p_y or p_j future partners that will detect their misbehavior. As p_x does not want to take such risk, it will correctly perform the audit of p_y and p_j's logs.
  - *Rational deviation 2.* Node p_x does not denounce node p_y when it is discovered faulty to protect it from being evicted from the system.
  - *Incentive.* The same incentive as above holds.

nodes among those that will be contacted by p_x or one of the correct nodes that will contact p_x during following rounds, say p_k, is correct and finds out that p_x behaved rationally through an audit, p_k could use the attestations sent by p_y and which are in p_x's log to prove that p_y behaved rationally. This will result in the eviction of p_y. As rational nodes do not want to be evicted, they do not attest for the correctness of a node without performing the corresponding verifications, and colluders do not protect themselves.

Finally, we present the deviations and the associated incentives concerning the exchange of updates between nodes.

- Fig 3.6 Step 1. Consider node p_y among the set of partners selected by p_x. Node p_x contacts p_y and sends it a proposition message containing the updates it owns.
  - *Rational deviation 1.* Node p_x does not send a proposition message to node p_y

  - *Incentive.* Each correct partner of p_x expects to receive its proposition at the beginning of each round, and will thus suspect p_x if the latter does not send it. Hence, to avoid being suspected, p_x always sends a proposition to its partners.
  - *Rational deviation 2.* Node p_x sends an invalid proposition message to node p_y (e.g., including less updates than what it holds).
  - *Incentive.* Furthermore, the proposition sent by p_x is necessarily correct (i.e., includes all the updates that appear in p_x's log), otherwise it risks eviction by its future partners if one of them audits its log.

- Fig 3.6 Step 2. Then, p_y replies with a request message containing the updates it is missing.
  - *Rational deviation.* Node p_y formulates an invalid request.
  - *Incentive.* Regarding the formulation of requests by p_y, the latter risks immediate eviction if the request is incorrect. Indeed, p_x would hold a proof of misbehavior of p_y, which it would immediately send to the source of the content dissemination session.

- Fig 3.6 Step 3. Finally, p_x serves the updates requested by p_y.
  - *Rational deviation.* Node p_x serves less updates than what node p_y requested to save resources.
  - *Incentive.* A similar ending would happen to p_x if it does not serve the updates expected by p_y.

- Fig 3.6 Step 1, 2, 3.
  - *Rational deviation.* Node p_x colludes with node p_y and exchange updates, but temper with their logs to make other nodes believe they did not, thus avoiding to exchange updates, and saving their future bandwidth.
  - *Incentive.* If instead, p_x colludes with its partners in order to exchange updates off the record, the group risks to receive most of the updates a second time from correct nodes in future exchanges wasting their bandwidth.

updates expected by p_y.

SECTION 3.4
Evaluation

In this section, we present the performance evaluation of the $AcTinG$ protocol. We start by introducing our methodology. Then, we compare the impact of colluders on $AcTinG$, BAR Gossip, and LiFTinG. 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 $AcTinG$, its performance in the case of massive node departure and its scalability in terms of memory and bandwidth

consumption using simulations involving up to a million nodes.

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 organization (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 $AcTinG$ exhibit 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 when a node joins, or leave, the system. Solving this issue, is classically done by using a tracker, i.e., a centralized server that handles membership, as in the FlightPath protocol [23], which could easily be integrated in our system. The tracker could even be replicated using classical fault-tolerance techniques (e.g., [66]).

3.4.1 Methodology and parameters 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[3], 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 further tolerance to message loss (combined with retransmissions), the source groups packets in windows of 40 packets, including 4 FEC footnoteFEC stands for Forward Error Correction. 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 $\frac{\ln(NbNodes)}{2}$ partnerships that it maintains for a duration of five rounds. For instance, in the fault free case, with $NbNodes = 400$, 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 3.3.1, 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.

[3]Grid5000: https://www.grid5000.fr/

3.4.2 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 log of other nodes and do not reply to messages sent by other nodes asking them to assess the behavior 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 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 3.8. 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 3.9. 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.

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 group. We have measured this overhead and results are depicted in Table I. From this table we observe that the overhead due to collusion is of at least 34% of the size of the stream (case of a group containing only two colluders). In addition, as seen in section 3.3.1, in a scenario where 10% of nodes collude, and where audits are performed 5% of the

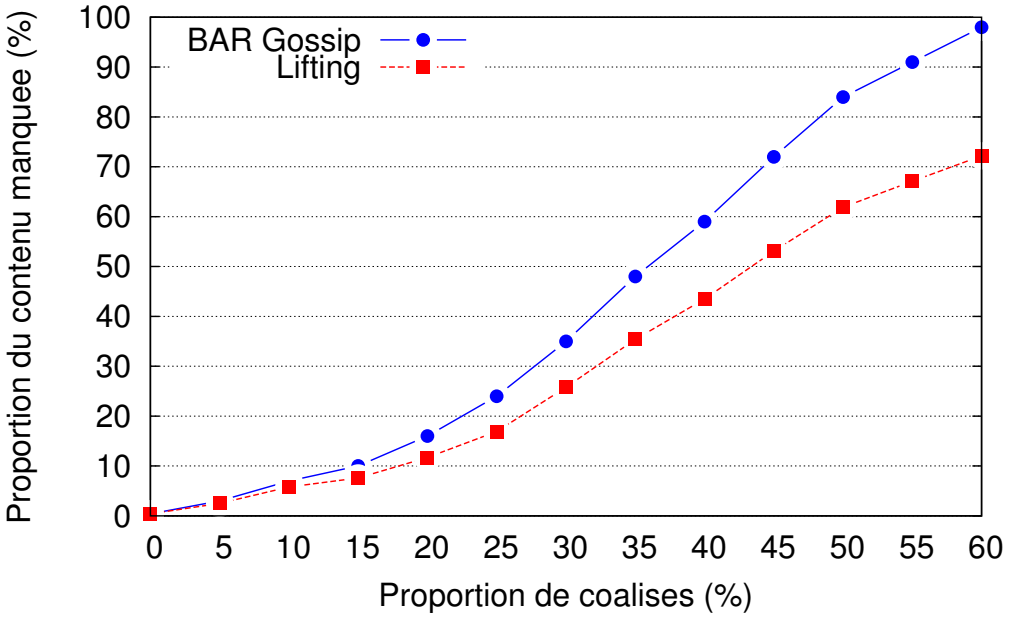


Figure 3.8 – Proportion of missed updates by correct nodes when a given proportion of the audience collude as a single group.

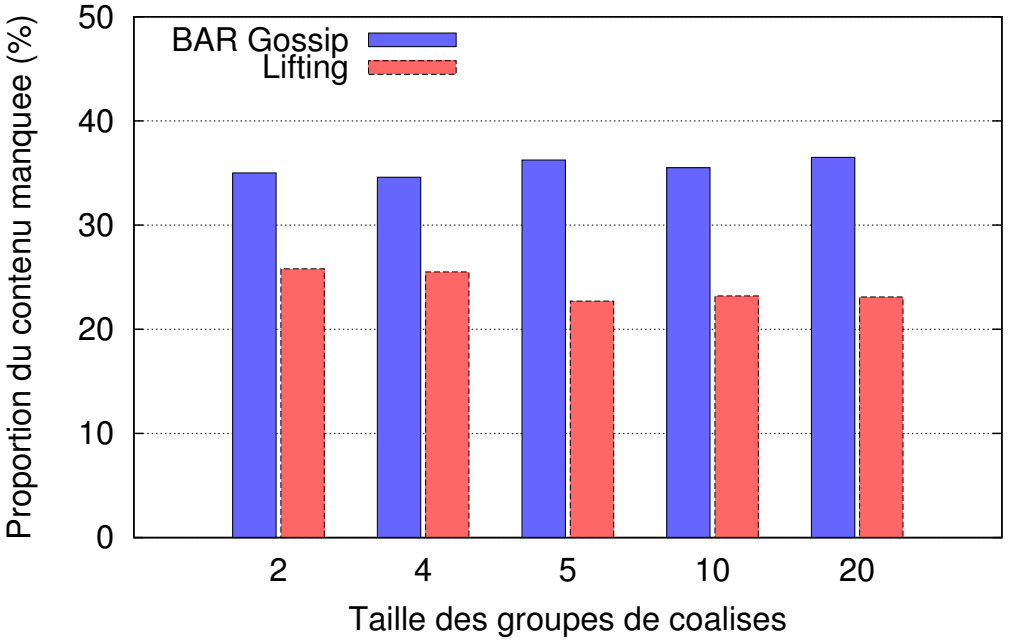


Figure 3.9 – Proportion of missed updates by correct nodes when 30% of the audience is rational, and collude in independent groups of equal sizes.

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.

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

Table I – Overhead of colluders in $AcTinG$.

3.4.3 Bandwidth consumption

To assess the overhead of $AcTinG$, we plot in Figure 3.10 the cumulative distribution of the average bandwidth consumption of nodes. Recall that $AcTinG$ is used to broadcast a 300kbps. Figure 3.10 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.

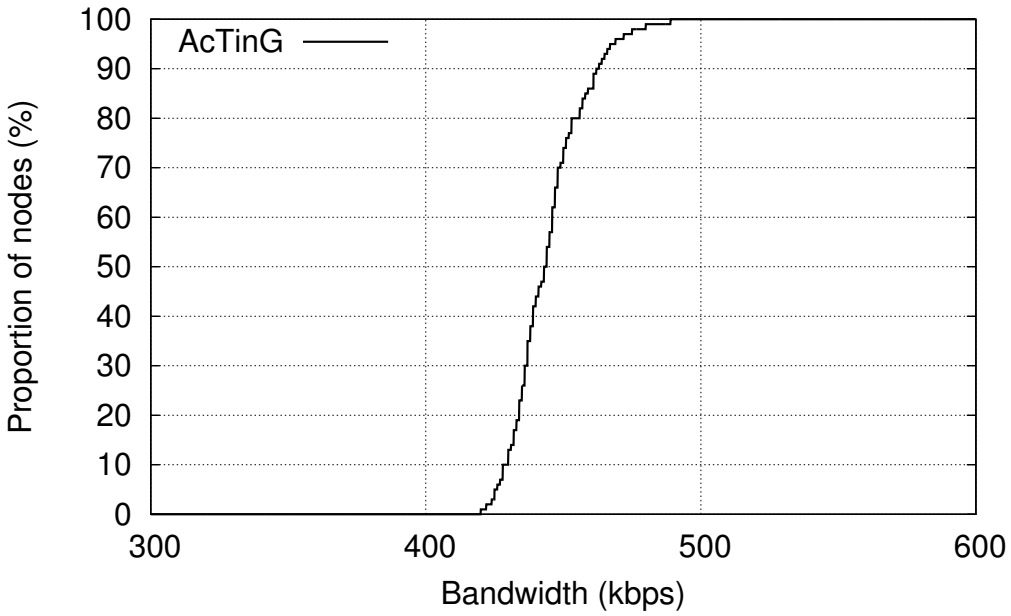


Figure 3.10 – Fault-free case: Cumulative distribution of average bandwidths.

3.4.4 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 3.11 and 3.12 respectively. Specifically, we observe in Figure 3.11 that the massive node departure, i.e., which happens after 500 seconds of the beginning of the experiment, immediately causes a decrease in the average bandwidth consumed by the remaining nodes, as the latter 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 stabilizes around 430 kbs (13% less than the original value), which is due to the decrease of the necessary number of partners per node.

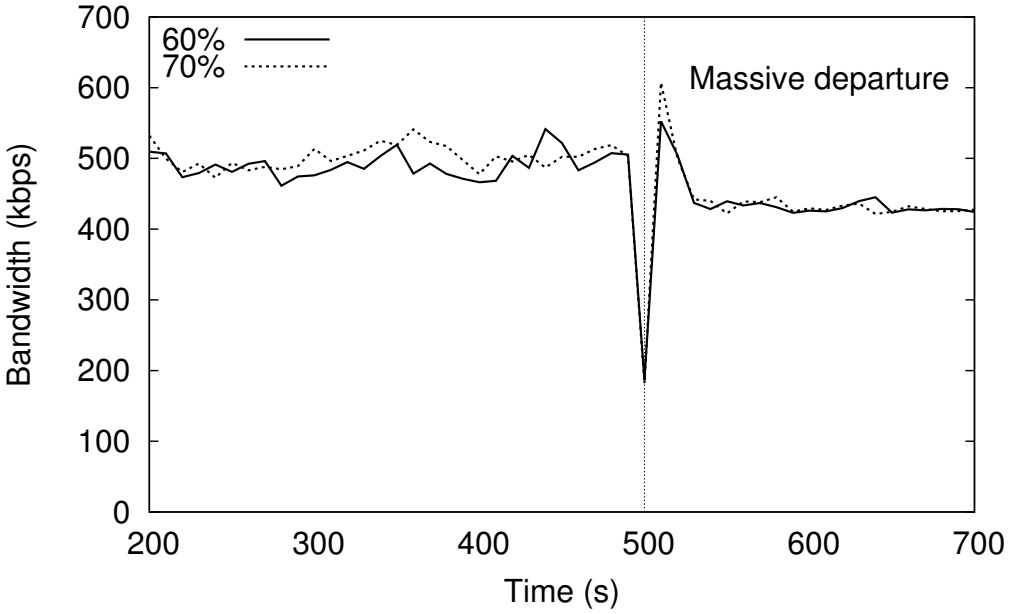


Figure 3.11 – Nodes average bandwidth after a massive departure.

We also compute the percentage of nodes that do not receive a viewable stream[4]. We observe in Figure 3.12 that only 2,5% nodes do not receive a viewable stream during the first second when 60% 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.

3.4.5 Scalability

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

[4]The stream is not viewable when more than 5% of the streaming windows cannot be displayed because of missed updates [22]

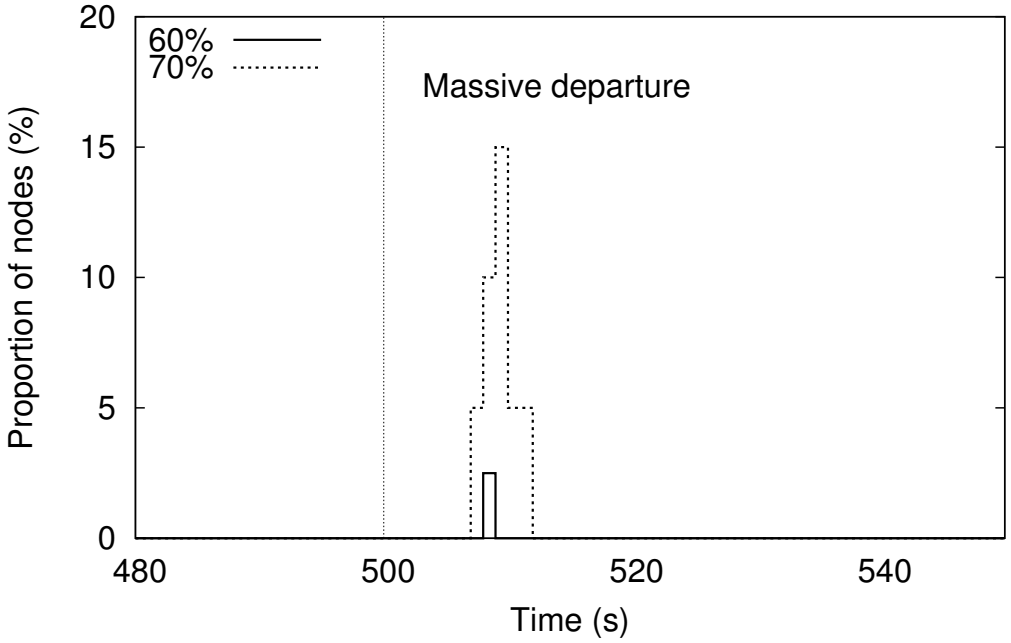


Figure 3.12 – Percentage of nodes that do not receive a viewable stream after a massive departure.

Results depicted in Table II 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 II – Average bandwidth and memory usage of $AcTinG$ in function of the system size.

SECTION 3.5

Conclusion

A number of gossip-based content dissemination protocols tolerating rational behaviors 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. We have then 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.

Part II

Privacy and rational resiliency in gossip-based dissemination systems

Chapter 4

Privacy in rational-resilient gossip

Contents

4.1. Principles of gossip and selfish behaviours 76
4.1.1. Selfish behaviours 77
4.1.2. Requirements against selfish behaviours 79
4.1.3. Accountability solutions 79
4.1.4. Privacy requirements 79
4.2. Rational-resilient gossip protocols 80
4.2.1. Rational resiliency by design 80
4.2.2. Audit-based approaches 81
4.2.3. Virtual currency approach. 82
4.3. Anonymous communication protocols 83
4.3.1. Altruistic relaying 83
4.3.2. Rational resilient relaying 85
4.4. Accountable and privacy preserving approaches 86
4.4.1. Zero-knowledge proofs 87
4.4.2. Collaborative verification protocols 87
4.5. Preserving privacy in other contexts 88
4.5.1. Peer-to-peer protocols 88
4.5.1.1. Interest-based social network 88
4.5.1.2. Collaborative filtering 89
4.5.1.3. Micro-blogging dissemination 89
4.6. Summary 90
4.6.1. Requirements 91
4.6.2. Summary of existing solutions 91
4.6.3. Conclusion 93

In the previous chapters, we identified a kind of rational deviation that had not been studied in the literature: collective deviations. We then proposed a new protocol, $AcTinG$, which is the first one to deter all kind of rational deviations in gossip-based systems. However, this protection comes at a cost, as nodes have to register all their actions in logs and periodically share them with other nodes. Nowadays, users of distributed systems are more and more aware that private information may be collected during their participation in a protocol, and later studied, or leaked. Users may be reluctant to give away information about them in a peer-to-peer context, and thus desire to maintain their privacy.

Protecting the privacy of users in the context of gossip would imply that a node should not learn more information about other participants than the bare minimum, which it learns from its interactions. For example, a node should not be able to discover if any two other nodes exchanged updates, and what updates they exchanged. In addition, a desirable property is to ensure that it is not possible to determine which content a node wants to receive. We formalise later in this chapter the privacy properties that a gossip protocol should enforce.

This chapter is organised as follows. We first recall in section 4.1 the principles of the gossip paradigm, and present the formalism and an example that we will use throughout this chapter. We also give requirements that gossip protocols should enforce to deter selfish behaviours, and those concerning the protection of the users' privacy. Section 4.2 details how the existing rational-resilient gossip protocols may leak private information. Section 4.3 focuses on anonymous communication protocols which are both accountable and privacy-preserving but suffer from poor performance. Section 4.4 describes some recent works that combined privacy and accountability in more general situations. Section 4.5 presents some distributed systems that focused on privacy issues. Section 4.6 concludes this chapter.

SECTION 4.1
Principles of gossip and selfish behaviours

The objective of a peer-to-peer content dissemination system is to reliably distribute a given content (e.g., a video stream, membership updates) among a set of interested nodes. Gossip protocols reach this objective by enforcing random exchanges between nodes in such a way that all nodes receive the whole content with a high probability [56]. Specifically, content dissemination is organised 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, at any given point in time, node X has a set of f_p predecessors $\{P_1, \ldots, P_{f_p}\}$ and a set of f_s successors

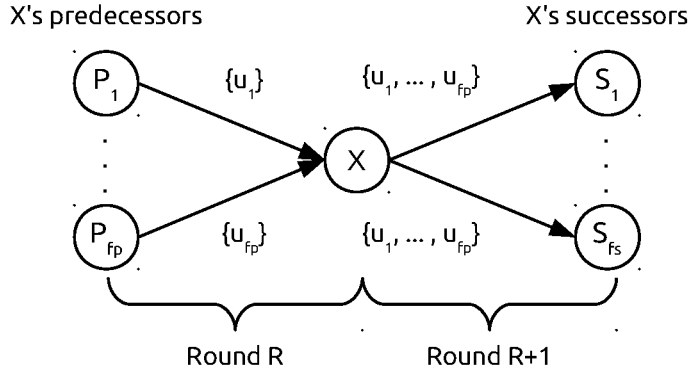


Figure 4.1 – Forwarding of updates in a gossip-based system

$\{S_1, \ldots, S_{f_s}\}$ that have been picked uniformly at random from the nodes participating in the system. This is commonly achieved by relying on a full membership protocol (e.g., [22, 25]), or on a distributed random peer sampling protocol (e.g., [64, 67]). 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). Following the gossip paradigm and if the number of successors per node is correctly chosen, nodes receive the whole content within a small delay.

4.1.1 Selfish behaviours

It has been presented in various studies (e.g., [14, 61]) that in practice 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). Selfish behaviours can have a tremendous impact on the dissemination of updates. For instance, a study performed in a live streaming system [26] has shown that 25% nodes behaving selfishly by decreasing their contribution by 30%, results in up to 60% nodes receiving an unusable video stream.

Figure 4.2 presents examples of selfish behaviours in which a node X (in the middle of the figures) aims at improving its benefit. In these figure, node X is a successor of nodes P_1 and P_2, from which it receives updates, and a predecessor of nodes S_1 and S_2, to which it must forward updates. In case (a), node X forwards a subset of the updates it received (i.e., $\{u_1\}$ instead of $\{u_1, u_2\}$) in order to save bandwidth. In case (b), node X forwards the received updates to a subset of its successors (i.e., it sends $\{u_1, u_2\}$ to S_2 instead of sending it to S_1 and S_2). In case (c), node X does not correctly choose one of its successor, and prefers to send its updates to an accomplice Y (depicted in the right of the figure). This deviation modifies the random propagation of updates and harms their dissemination. Finally, in case (d), X and Y form a selfish coalition, in which they exchange updates off the record (e.g., update u_1 in the figure), for example

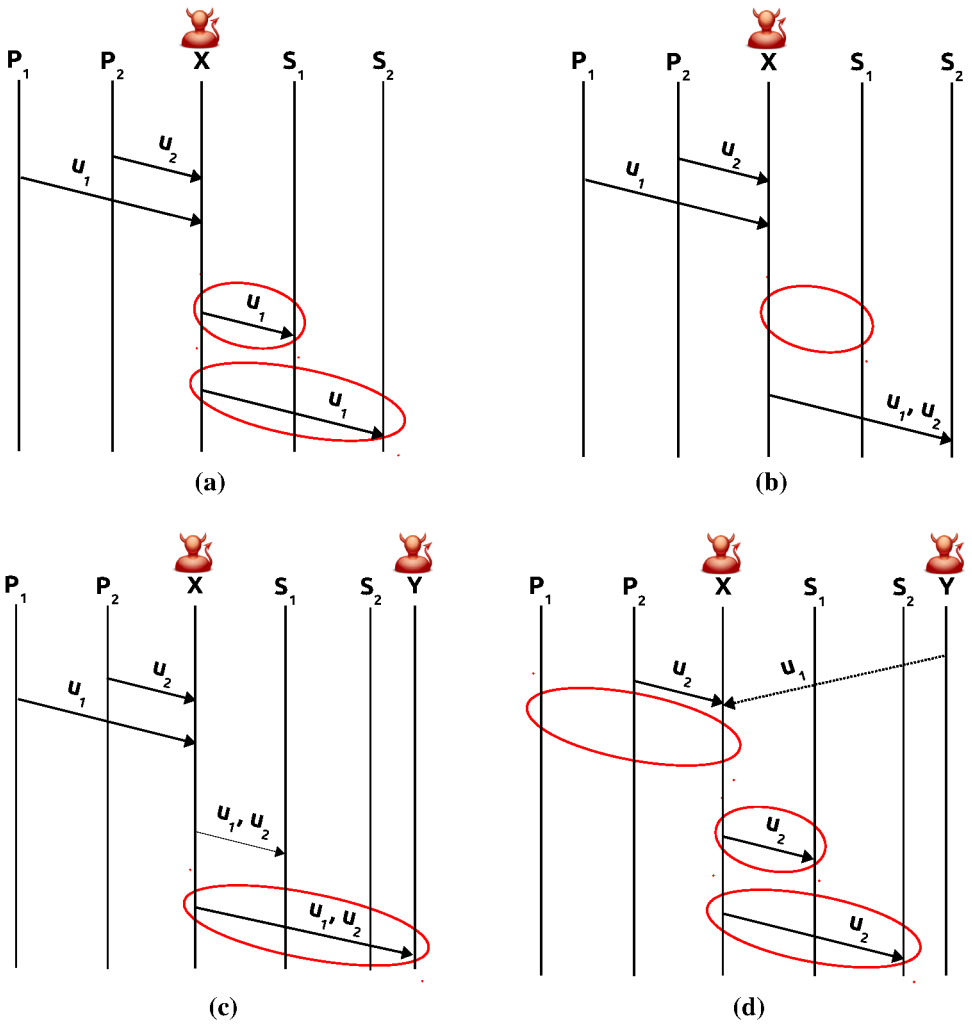


Figure 4.2 – Examples of selfish deviations

because they are connected with a particularly good network. By doing so, node X can skip receiving u_1 officially from node P_1, which prevents it from forwarding it to S_1 and S_2.

4.1.2 Requirements against selfish behaviours

In order to protect a gossip-based system from selfish deviations of nodes, and based on the previous examples, we can intuitively infer three properties, $\mathbf{R_1}$, $\mathbf{R_2}$ and $\mathbf{R_3}$, that have to be verified. Together, these properties force nodes to receive the updates they have not yet received (property $\mathbf{R_1}$ prevents nodes from exchanging updates outside the protocol, and avoiding to receive them to avoid to forward them further) and to correctly forward the content they receive (properties $\mathbf{R_2}$ and $\mathbf{R_3}$).

$\mathbf{R_1}$ **Obligation to receive**: At a given communication round, a node must receive the updates it did not receive officially in the previous rounds.

$\mathbf{R_2}$ **Obligation to forward**: At a given communication round, a node must forward the updates it received to other randomly selected nodes.

$\mathbf{R_3}$ **Random partnerships**: A node predecessors and successors should be selected uniformly at random.

4.1.3 Accountability solutions

Accountability mechanisms (e.g., PeerReview [29], FullReview [35], AVMs [31]) 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., in $AcTinG$ [5]).

Figure 4.3 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 precises 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 PeerReview and AVM [29, 31]) or on secure hardware (as in Trinc [33]) 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 $R + 1$.

4.1.4 Privacy requirements

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., [36, 37]) have shown that the consumed media can disclose information about individuals (e.g., gender, sexual, religious or political preferences). Further to learning

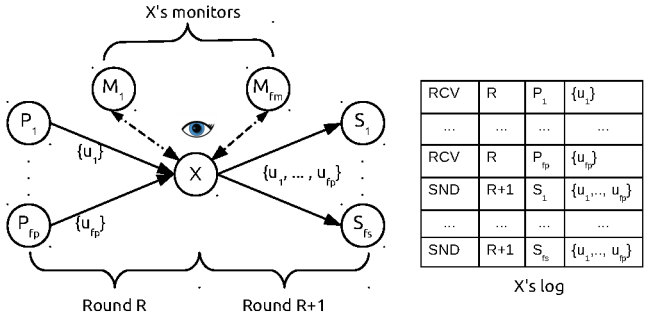


Figure 4.3 – Accountable gossip

nodes interests, it is then possible to infer *links* between nodes sharing similar interests, thus possibly inferring sensitive information about them. In addition, it is not enough to encrypt updates in secure logs. Relying on this method, it is necessary that nodes known the associations between updates and their encryptions in order to register their actions in their log. They could also learn the associations between updates they do not even receive. Thus, when observing the logs of other nodes, they would be able to understand the events they describe.

Hence, in addition to the properties $\mathbf{R_1}$, $\mathbf{R_2}$, and $\mathbf{R_3}$, we aim at enforcing the following privacy properties in our protocol:

$\mathbf{P_1}$ **Private Content Consumption:** It is not possible to state that a given node has consumed a given content.

$\mathbf{P_2}$ **Private Session Membership:** It is not possible to identify the set of members that are interested in a similar content.

$\mathbf{P_3}$ **Private Exchanges:** It is not possible for nodes to observe the precise updates exchanged between two partners (except for the two partners) or to predict that two nodes exchange updates.

SECTION 4.2

Rational-resilient gossip protocols

In this section, we examine to which extent the privacy of nodes is endangered by the design of the mechanisms deployed to deter deviations. In Chapter 2, we presented several gossip protocols that are resilient to rational behaviours. We do not describe these protocols in details again, instead we only indicate the elements that concern the privacy of nodes.

4.2.1 Rational resiliency by design

BAR Gossip [22], and it successor FlightPath [23], are streaming protocols that have been designed to handle both selfish and Byzantine deviations. We previously explained (see Section 2.1.1.1) that they do not force peers to initiate exchanges with other nodes,

which can limit the efficiency of the content dissemination, however we did not studied their impact on the privacy of nodes.

Association with a content. In these two protocols, all nodes have to know the full membership of a session, which is uniquely associated to a disseminated content. Then, each node that is part of a session knows that all the other nodes are interested in the same content. Thus properties $\mathbf{P_1}$ and $\mathbf{P_2}$ are not enforced in these protocols.

Predictability of interactions. Running these protocols, each participant chooses the nodes it contacts using a deterministic, yet random, procedure that depends on its identifier (or its public key) and on the round number. Both information being public all interactions, either coming from the balanced exchange procedure or the optimistic push procedure, can be predicted by any peer in the system. However, it is worth mentioning that the updates that are exchanged between two partners cannot be precisely known by other nodes. Finally, property $\mathbf{P_3}$ is not completely enforced by these protocols.

4.2.2 Audit-based approaches

In this section, we describe several gossip-based protocols that use periodical audits to verify the correct forwarding of updates in a system. LiFTinG and G2G use verifications based on secure logs to enforce properties $\mathbf{R_1}$, $\mathbf{R_2}$, $\mathbf{R_3}$ in presence of non collective rational deviations. However, none of the privacy properties $\mathbf{P_1}$, $\mathbf{P_2}$, $\mathbf{P_3}$ are enforced. We details the privacy leaks in the following parts of the section.

LiFTinG

LiFTinG [26] (see Section 2.1.2.3 for details) is a protocol that forces nodes to disseminate the updates they receive, using cross-checking procedures and audits to check how a node forwarded its updates. In chapter 2, we focused on the false-positive and false-negative detection rates of this protocol, but we study here the mechanisms that threaten the privacy of interactions. First, all nodes participating in LiFTinG know that they share the same interest in the content being distributed. Thus, properties $\mathbf{P_1}$ and $\mathbf{P_2}$ are not enforced in LiFTinG. We present two kinds of verifications that aim at detecting selfish nodes, but leak information about nodes in the following, thus preventing property $\mathbf{P_3}$ to be enforced.

Direct cross-checks. The direct cross-checking procedure has been previously illustrated in Figure 2.4. After a node p_0 sends some updates to a node p_1, using the standard three-phase gossip procedure, it also has to check that p_1 forwarded these updates to other nodes, here p_2 and p_3, selected randomly among the audience. To permit this, node p_1 must send to node p_0 the list of the nodes it contacted. Node p_0 can then ask these nodes to confirm that they receive the updates it sent to node p_1. Any node in the system is then able to discover all the nodes located two hops after itself in the content dissemination path, and associate them with particular updates. This kind of verification allows a node to learn the interactions of the node it is checking, and to which is previously sent some updates. Thus property $\mathbf{P_3}$ is not enforced.

***A posteriori* verifications.** Periodically, each node picks a random node and audits its history, which is a non cryptographically-secure log. The auditing node verifies the randomness of the distribution of peers to which the audited node proposed updates. In addition, the auditing node is able to contact the previous partners listed in the history to ask for confirmations of the events presented. This could also constitutes an attack, as anyone can contact any other nodes to ask for information concerning an hypothetical exchange. Similarly to the previous verification, a node that executes an audit learns in details all the exchanges of the audited node.

Due to the fact that the session membership is known from all nodes, and to the verification procedures that detect selfish nodes, LiFTinG does not enforce the properties $\mathbf{P_1}$, $\mathbf{P_2}$ and $\mathbf{P_3}$.

Give2Get

Give2Get [68] (G2G) uses ideas that are close to those of LiFTinG. While this protocol is specifically designed for mobile wireless networks it can be used in more general systems. G2G uses epidemic forwarding (i.e., gossip) to transmit messages among nodes. While G2G does not explicitly uses audits, nodes check each other. Properties $\mathbf{P_1}$, $\mathbf{P_2}$ are not enforced since all nodes participating in this protocol are known to be interested in the same content.

Forwarding verification. When a node receives a message from another node it has to send back a signed proof of relay (POR) which is a message that contains the hash value of the received message, and the identities of the receiver and the sender nodes. The sender of the message can then use this POR to prove to its predecessors that it correctly forwarded the message. More precisely, each node has to collect two PORs for each message it receives (using a fanout of two will limit the scalability of this protocol, but it can be generalised). This verification prevents property $\mathbf{P_3}$ from being enforced.

In this protocol, the full membership is known to nodes. In addition, the proofs of relays divulge information about nodes. The predecessors of a node are able to learn which messages where transmitted to which nodes, i.e., a node can learn where its messages go two hops after it in the dissemination path. To conclude, properties $\mathbf{P_1}$, $\mathbf{P_2}$ and $\mathbf{P_3}$ are not enforced in G2G.

4.2.3 Virtual currency approach.

Building on the idea of virtual currency, an interesting approach is developed in [38], where it is shown to provide accountability without compromising privacy in a peer-to-peer system. This solution requires two trusted entities: 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. The privacy of exchanges is ensured at the condition that these entities are available and trusted, which is not something we are ready to assume in a P2P environment. However, at these conditions, selfish individual nodes may be forced to participate actively in the dissemination of updates.

Using this approach, trusted entities would prevent properties $\mathbf{P_1}$, $\mathbf{P_2}$ and $\mathbf{P_3}$ from being enforced.

SECTION 4.3

Anonymous communication protocols

In this section we present several existing anonymous communication protocols that could be used to create a one-to-all dissemination protocol, the source sending updates and nodes disseminating and receiving them.

Anonymity is stronger than privacy as it provides the following properties, which were defined in [69]:

- **Sender anonymity.** It is not possible to determine the sender of any given message.
- **Receiver anonymity.** It is not possible to identify the destination of any given message.
- **Unlinkability.** An observer is not able to identify a pair of nodes as communicating with each other.

Thus enforcing anonymity would also enforce the privacy property $\mathbf{P_3}$. However, to obtain the privacy properties $\mathbf{P_1}$ and $\mathbf{P_2}$, it would be necessary to serve several contents simultaneously to nodes to hide the interests of nodes. As we will see in Chapter 5, where we realise a performance evaluation, using anonymous communication protocols to build a privacy-preserving and rational-resilient content dissemination protocol is not possible because of their high overhead.

This section begins with a description of the first developed protocols that are not accountable, and in which selfish nodes may avoid to participate correctly in all steps. We then present protocols that have been designed to prevent these deviations.

4.3.1 Altruistic relaying

The first anonymous protocols DC-Net [70], and Onion Routing [71] have focused on enabling the strongest possible anonymity level for the former, and on providing practical performance for the latter. However, these two protocols take the participation of nodes for granted. Some anonymous communication systems, like Dissent [39] and RAC [40], force nodes to correctly execute their role of relay. However, if we imagine a gossip protocol that uses anonymous communications a node that would be the destination of a message would correctly receive it, but would still not be forced to propagate it to other nodes.

In addition, using anonymous communication systems to provide privacy can also be seen as a performance overkill. For example, for each message sent anonymously, Dissent v1 [39] forces each node to send messages to all the other nodes. The second version of this protocol [41] uses trusted nodes which receive anonymous communication

requests from untrusted nodes, and run a protocol involving all-to-all communications between trusted nodes.

DC-Net

The DC-Net [70] protocol relies on the principle of secret sharing, and is the most robust anonymous communication protocol. For an opponent to break anonymity it is necessary to control all the nodes in the system. Nodes proceed in rounds. During one round, only one node is allowed to send a message. If two nodes try to send messages during the same round a collision occurs and none of the messages is correctly delivered (although some approaches have introduced the possibility to reserve slots [42, 72]). Nodes are organised in a structured network and nodes have to forward to their neighbours the encrypted messages they receive. Nodes apply a XOR-based mechanism on the messages they receive from their neighbours to decrypt messages. The highest drawback of DC-Net is its high overhead in terms of communications and computational costs. At each round, any pair of two nodes in the system needs to exchange messages. Some protocols have been devised to reduce these overheads. For example, Herbivore [73] organise nodes into groups to decrease the total number of messages exchanged. In practice, this protocol is consider to be unusable as soon as there are more than 50 peers in a session [74].

Onion Routing

The onion routing protocol [71] is probably the most famous anonymous communication protocol. This is due to the important number of variants of this protocol that have been implemented, among which Crowds [75], Cashmere [43], Tarzan [44], and TOR [76].

Sending protocol. In this protocol, a node that wants to send a message to another destination node follows the following steps:

- Choose a set of nodes, called the relays, which defines the path that the data will follow until the destination.
- Encrypt the data as a onion, which is a recursively defined layered data structure. Each onion contains another onion, the identity of the following relay on the path and may contain cryptographic information (such as the cryptographic algorithm that is used to cipher data).

Figure 4.4 shows how a node A sends a message anonymously to node B using two relay nodes R_1 and R_2. Each relay is able to decrypt the message it receives that has been previously encrypted using its public key, and finds the identity of the next destination along with the encrypted data it has to transmit.

Relaying and selfishness. Each node on the path uses its private key to decipher the onions it receives, and then forward the internal onion to the next relay. For security concerns, this internal onion is padded to maintain a fixed size. This protocol assumes that nodes are altruistic, i.e., they follow the protocol even though they have no particular interest in doing so. A node could drop all the messages it should relay without being detected.

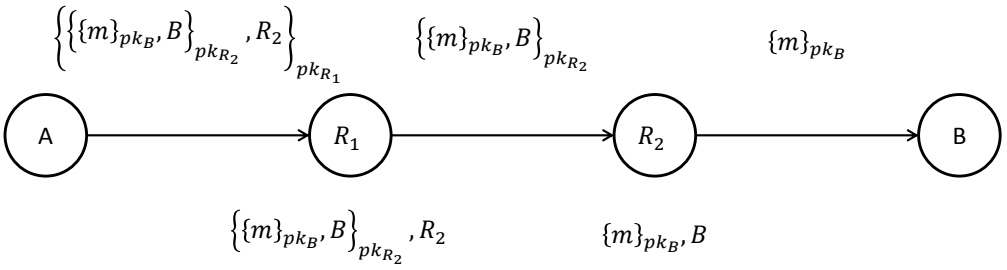


Figure 4.4 – Sending of an onion using two relays.

4.3.2 Rational resilient relaying

We describe in this part anonymous communication protocols that force nodes to act as relays. Theoretically, they could be used to disseminate a content within an audience of nodes. However, the audience would still be associated to a content (properties $\mathbf{P_1}$ and $\mathbf{P_2}$ would not be satisfied), and the cost of these protocols is way too important for practical applications.

Dissent

The first anonymous communication protocol designed to tolerate rational nodes is Dissent [39] (Dining-cryptographers Shuffled-Send Network). This protocol uses DC-Net as a basis and thus suffers from the same scalability issues (it is not intended to be used with groups larger than 50 nodes), and adds a double encryption system. When a node detects the deviation of another node, the execution of a round is stopped and at least one of the misbehaving nodes is anonymously proved to be faulty, and evicted.

Dissent operates in two stages.

- **Shuffle phase** A set of fixed-length messages, one from each group member, is permuted and anonymously broadcast to all nodes. This protocol has two practical limitations: all messages must be of equal length, and the decryption of messages is serial (i.e., it takes a long time if the number of participants or the number of messages is important).

- **Bulk phase.** In each round, all group members emit variable length messages (possibly void messages) to any destination. To do so, they broadcast bit streams based on pseudorandom seeds distributed via the shuffle phase in such a way that XORing all the bit streams together allow nodes to obtain a permutation of all members' variable length messages.

Dissent v2. The protocol was later modified in [41] to improve its performance. In this version, trusted servers that run the first version of Dissent are used by nodes which still benefit from anonymity guarantees. We do not want to make the assumption that trusted servers are available in general P2P solutions.

RAC

RAC [40] is a rational resilient anonymous communication protocol that is based on Onion Routing, and that was designed to scale better than Dissent. In RAC a node that has to relay a message will broadcast it to all the nodes so that the node from which it received the onion can check the forwarding. Thus, when receiving a message all the nodes have to try to decrypt it, and the node that will succeed will have to send it to all other nodes. This protocol is shown to be a Nash equilibrium in the original publication.

Fireflies infrastructure. To limit the number of messages to send, and to make this number independent of the system size, the membership is organised on rings, as is Fireflies [25] [1]. Nodes have a successor and a predecessor on each on these rings. When a node receives a message from one of its predecessors it will first forward the onion to its successors, which will prove its reception. Then, it will try to decipher the onion, and upon success, will also broadcast the internal onion to its successors. If L is the number of relays used in the onion path, and R is the number of rings, then the propagation of one message has a cost of $L \times R \times Bcast(N)$.

Broadcast groups. To avoid that all the messages reach all the nodes the membership is divided in groups of size G where nodes are organised as previously presented. If two nodes that belong to the same group want to communicate they use the previous protocol inside the group they belong to. The cost of the protocol is then reduced to $L \times R \times Bcast(G)$. However, when two nodes that are not part of the same group have to communicate the procedure is different. The source of the message broadcasts it inside its group, but the smallest onion is made in such a way that it informs the last relay that it has to forward the message in the group of the destination node (to allow the reception of the message) and inside its own group (to prove to the source that it did forward the message). The cost of the protocol is then lower to the previous cost and is equal to $((L-1) \times R \times Bcast(G)) + (R \times Bcast(2G))$. Simplifying the expression leads to $L \times R \times Bcast(2 * G)$ which is better than without groups.

Dissent and RAC are anonymous communication protocols that, theoretically, could be used to disseminate a content in an audience of peers in presence of selfish nodes, and would enforce properties $\mathbf{P_1}$, $\mathbf{P_2}$ and $\mathbf{P_3}$ if several contents are simultaneously disseminated. However, Dissent is not able to scale correctly, or would need the assistance of trusted servers, and the overhead of RAC is too important to disseminate standard multimedia contents (e.g., musics or videos).

SECTION 4.4
Accountable and privacy preserving approaches

In this section, we introduce existing approaches that would check the correctness of a node while preserving its privacy. The first possibility is to rely on zero-knowledge proofs, and the other one, is to use very recent solutions that have been developed by

[1] see Section 2.1.2.2 for a description of Fireflies

Haeberlen et al. in several publications. For each solution, we briefly argue why it can not be applied to gossip.

4.4.1 Zero-knowledge proofs

Classical zero-knowledge proofs [77] (ZKPs) are methods by which one party can prove to another party that a given statement is true, without revealing any information apart from the fact that the statement is true. They are difficult to apply to gossip because they are designed to verify functions with a fixed number of inputs, but 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-based systems, the quantity of messages a node receives during a time interval is not predictable. In addition, the high overhead of general-purpose ZKPs would be prohibitive for many applications.

Another approach is to make nodes compute the expected output of the node that is checked using secure multi-party computation (MPC) [45], and then check them to the actual behaviour of the node. But MPC is practical only for very simple functions.

4.4.2 Collaborative verification protocols

We now present some ideas that have been presented by Haeberlen et al. in several papers [46, 47]. They all focus on the same idea: it is possible in practice using a particular data structure to control the behaviour of members in distributed applications, under some assumptions, while preserving their privacy.

Privacy-preserving accountability

In [46], Haeberlen et al. propose an original and clever method that allows nodes to collaboratively check each other in distributed systems, where the behaviour of nodes can be checked by other nodes, while preserving their privacy using a data structure called a Merkle Hash Tree. This proposition works under the assumption that nodes do not collude.

Merkle Hash Tree. Participating nodes maintain a Merkle Hash Tree, which is a tree whose leaves represent all the possible states of peers, and the internal nodes are deterministically computed using a hash function and the value of their sons. Nodes in the system regularly make public the root hash value of their tree. The interest of this data structure lies in the fact that it is difficult to find collisions, i.e., create two trees, or subtrees, whose internal hash values are different but whose root values are equal. Thus, as nodes

Collaborative verification. Two nodes that interacted have to convince each other that they are not in a state that is inconsistent with what they already know. To do so, they send each other the internal values of their hash tree. Basically, all nodes that interacted with another node will then be able to collectively, but anonymously, check its Merkle Hash Tree.

Application. This approach has been applied to a BGP routing system [48]. However very elegant, we believe that this approach cannot be applied to gossip protocols, because it is not possible to represent concisely all the possible states of a node. For example, if a node has to receive X messages, using states to represent what it did receive would give 2^X states. Using a dissemination protocol, the set of states can then become completely unpractical. However, if an optimisation can be found to represent the states of a node this approach could become practical.

SECTION 4.5

Preserving privacy in other contexts

In the previous section, we considered content-dissemination systems and studied how they would preserve, or harm, the privacy of their users. In this section, we present several P2P protocols that aim at preserving the privacy of their users in other contexts.

4.5.1 Peer-to-peer protocols

Several domain-specific P2P systems have been developed by Kermarrec et al., we briefly describe them in the following. These solutions cannot be applied to gossip-based systems.

4.5.1.1 Interest-based social network

GOSSPLE [49] builds an overlay of anonymous nodes depending on their interests. Using a gossip protocol, nodes periodically send their interest profiles, compute their distance in terms of interest to other nodes, and update their connections to other nodes. Nodes can then use their network of acquaintances for various goals, for example to obtain personalised search results.

Profile anonymity. Each user has a profile which describes its interests. The association between users and profiles is hidden. Each node has a proxy, which is another node of the protocol, which disseminates its profile on its behalf. To send its profile to its proxy, a node has to use a relay which will forward the encrypted profile. This mechanism protects the system from single adversary nodes, but can expose it to collusions.

Neighbourhood maintenance. Each node periodically selects its oldest neighbour and exchange the descriptors, which include profiles represented as Bloom filters, of c nodes. Each node then recompute the identities of the c nodes that maximise its distance metric.

The methods used in GOSSPLE could be applied to gossip in the sense that each node could obtain a set of neighbours sharing similar interests. However, the way a content would actually be disseminated inside a GOSSPLE overlay is not defined, and it is not clear if it would preserve the privacy properties $\mathbf{P_1}$, $\mathbf{P_2}$ and $\mathbf{P_3}$. In addition, selfish behaviours would not be prevented.

4.5.1.2 Collaborative filtering

Collaborative Filtering enables users who share the same interests, often revealed by their profiles, to benefit from a content that one of them considered relevant. Such recommendation techniques are more efficient if the profiles are public, however preserving the privacy of these profiles is also an important concern. The mechanisms presented in [50] consist in a decentralised protocol which aim at providing nodes the information they are interested in while preserving the privacy of their profiles.

Interest-based overlay. Nodes maintain an overlay which is constructed using the profiles of nodes. Nodes maintain a list of neighbours which share similar interests. Second, information is disseminated using epidemic forwarding as nodes forward the information they receive and liked to their neighbours. The protocol protects the privacy of nodes when they transmit their profile to their neighbours and when they receive posts that should not reveal their interests.

Privacy and obfuscation. Each disseminated item is associated with a profile that aggregates information from the profiles of users along its dissemination path. This first allows nodes to decide if any item they receive has chances to be appreciated. Second, nodes are able to compute the portion of their profile that is the least sensitive one, i.e., the one that is shared by a large portion of other users in the system. The amount of users' profile that is revealed is a parameter that the system designer can tune.

Random dissemination. Nodes that liked an item may not forward it to their neighbours, and may forward items they are not interested by. An attacker observing the propagation of items would not be able to establish with certainty if nodes like or dislike them. However, when a user creates a new item the solution does not provide privacy to the node that forwarded it to its neighbours.

Similarly to GOSSPLE, it is not clear if disseminating a high-bandwidth content using collaborative filtering would be efficient in terms of bandwidth. In addition, it would not preserve the properties $\mathbf{R_1}$, $\mathbf{R_2}$ and $\mathbf{R_3}$ that aim at preventing selfish behaviours.

4.5.1.3 Micro-blogging dissemination

Micro-blogging is another kind of P2P systems which has emerged as a way to disseminate information quickly and efficiently. Posts on Twitter and Facebook can be seen as micro-blogging examples, and are made visible thanks to forwarding procedures (e.g., *sharing* in Facebook, and *retweeting* in Twitter). Such procedures propagate a post that a user appreciated to its neighbours. The goals of RIPOSTE [51] is to help interesting posts to be disseminated in the system and eliminate other posts, while preserving the privacy of the users' opinion.

Algorithm's principles. RIPOSTE is an algorithm that takes as input the opinion of users about a post (e.g., like/dislike) and as output decides to repost or not. If a user likes a post, the algorithm will decide to repost with a given probability, and if he

does not, the post has another slightly smaller probability to be reposted. For each user and for each post, these two probabilities are computed as a function of the number of connections of the user, and the extent to which the post has already reached the connections. An observer thus cannot say if a user really reposted a post, or if the algorithm made it.

Algorithm details. RIPOSTE uses two real numbers parameters that are used to take decisions: the spreading factor $\lambda > 1$ and the blocking factor $0 < \delta < 1$. If u is a user that receives a new post, and $s > 0$ is the number of its followers that have not yet received the post. This last number is classically easy to obtain as a user has access to the post that each of its followers have received. The reposting decision is taken as follows:

- If u likes the post, it is reposted with probability $r_{like}(s)$ which is equal to $\frac{\lambda}{s}$ if $s \geq \lambda + \delta$, else it is equal to $1 - \frac{\delta(s-\delta)}{\lambda s}$.
- If u does not like the post, it is reposted with probability $r_{dis}(s) := \frac{\delta}{s}$.

Privacy and dissemination guarantees. The values of δ and λ determine the level of privacy that is obtained, and how well appreciated posts will be disseminated. If q is the probability for a given user to like a post, and $\widehat{q}$ is the same probability after observing that the post was reposted from u then

$$\widehat{q} = \frac{q}{q + (1-q)\delta/\lambda}.$$

The closer δ is to λ the better is the privacy, however, ensuring a good dissemination of interesting posts adds other constraints. Building on the theory of branching processes [3] and under some assumptions, there is a popularity threshold $p^* = \frac{1-\delta}{\lambda-\delta}$ such that posts with popularity lower than p^* spreads to at most a constant factor larger than the number of followers of the original poster, and posts with larger popularity spreads to a least some fraction of the network.

Similarly to the previous P2P protocols of this section, RIPOSTE does not prevent selfish behaviours: nodes could avoid to forward a content they received. In addition, to enforce properties $\mathbf{P_1}$ and $\mathbf{P_2}$ it would be necessary to simultaneously disseminate many contents that nodes would receive, which would not be optimal in terms of performance.

SECTION 4.6
Summary

In this section, we summarise what has been said in this chapter. First, we develop the requirements for a gossip-based system that would aim at preserving the privacy of its users while ensuring that nodes participate in the dissemination of updates. Second, we explain why existing approaches are not fully satisfactory. Finally, we conclude the chapter.

4.6.1 Requirements

In this part, we present the main concerns a system designer should have in mind when building a content-dissemination protocol based on gossip tolerating individual rational collusions and preserving the privacy of interactions.

Decentralised solution. We desire to design a gossip based system that does not rely on any trusted entity, or on a central server. This assumption is standard in P2P systems.

Defence against selfish nodes. Properties $\mathbf{R_1}$, $\mathbf{R_2}$ and $\mathbf{R_3}$ should be enforced against individual selfish nodes.

Compatibility with gossip. Gossip is particular, interactions between nodes are random, and the content of exchanges cannot be predicted. Thus, a privacy-preserving accountable gossip protocol has to take this characteristics into account.

Privacy requirements. Properties $\mathbf{P_1}$, $\mathbf{P_2}$ and $\mathbf{P_3}$ have to be enforced. In particular, these properties should hold against a global and active attacker, which can control a collusion of nodes trying to break the privacy.

4.6.2 Summary of existing solutions

We present in table I a summary of the related works. For each work, we indicate if it satisfies the requirements we identified. We also present the characteristics of the PAG protocol, which will be presented in the next chapter.

	Decentralised	**Accountable**	**Applicable to Gossip**	$\mathbf{P_1}$ **and** $\mathbf{P_2}$	$\mathbf{P_3}$	**Good performance**
Accountable anonymous communication systems [39, 71]	✓	✓	✓	✓	✓	×
Zero-Knowledge Proofs [45, 77]	✓	✓	×	-	-	-
BAR Gossip [22, 23]	✓	×	✓	✓	×	✓
LiFTinG [26]	✓	×	✓	×	×	✓
AcTinG [5]	✓	✓	✓	×	×	✓
PeerReview [29], AVM [31], TrInc [33], A2M [32]	✓	✓	✓	×	×	✓
Merkle Hash Tree [46]	✓	✓	×	✓	✓	✓
Virtual Currency [38]	×	✓	✓	✓	✓	✓
PAG	✓	✓	✓	✓	✓	✓

Table I – Detailed summary of existing approaches.

4.6.3 Conclusion

In this chapter, we have identified the requirements one should respect when building a privacy-preserving and accountable gossip protocol. Based on these requirements we have seen that there is currently no solution that could provide both accountability and privacy in gossip in a practical way. In the next chapter, we will detail PAG, which is the first protocol to reach this goal.

Chapter 5

PAG: Private and accountable gossip

Contents

5.1. System model 97
5.1.1. Communications and cryptographic assumptions 97
5.1.2. Gossip sessions 97
5.1.3. Nodes behaviors 98
5.1.4. Adversary model 98
5.2. PAG Overview 98
5.2.1. Global membership and monitoring 98
5.2.2. Exchanges of updates using gossip 99
5.2.3. Enforcing accountability using monitoring 100
5.2.4. Enforcing privacy using homomorphic encryptions 101
5.3. Design of PAG 103
5.3.1. Forwarding updates 103
5.3.2. Encrypting a set of updates 104
5.3.3. Combining all encryptions 106
5.3.4. Practical implementation details. 106
5.4. Security, privacy and accountability 108
5.4.1. $k - PAG$: clustering sessions 108
5.4.2. Enforcing $\mathbf{P_3}$ under global and active attacks 109
5.4.3. Accountability against selfish deviations 110
5.5. Performance evaluation 113
5.5.1. Methodology and Parameter Setting 113
5.5.2. Probabilistic study of the impact of coalitions 114
5.5.3. Comparison with an accountable gossip protocol and impact of the number of contents 115
5.5.4. Comparison with anonymous communication systems . . 116
5.5.5. Impact of updates size 119
5.5.6. Cryptographic costs 119

5.5.7. Scalability 120
5.6. Conclusion 120

As we have seen, there is currently no solution that would protect content-dissemination systems against selfish nodes and at the same time provide privacy to its users. In this chapter, we present PAG, the first protocol that possesses these two properties.

In Section 5.1, we present the assumptions we make about the users and the system, and the architecture of our solution at the level of a session, and at the level of the whole system. Then, in Section 5.2 we introduce the intuition behind PAG which relies on the homomorphic properties of an encryption mechanism close to the one used in RSA. Section 5.3 details the exchanges of messages between nodes in PAG that implement these mechanisms, explain the modifications needed to use this protocol in practice, and give more details on the witnessing protocol. Section 5.4 proves the security guarantees of this protocol using both ProVerif and probabilistic guarantees. Finally, Section 5.5 provides a performance evaluation, and Section 5.6 concludes this chapter.

SECTION 5.1

System model

In this section, we present the assumptions we make for the rest of this chapter.

5.1.1 Communications and cryptographic assumptions

As classically made in gossip-based protocols (e.g., in [22] and [23]) we structure time using rounds whose duration, also called the gossip period, is assumed to be long enough for nodes to complete their interactions. Nodes are roughly synchronized, which allows them to check each others' periodical exchanges based on the specification of the protocol.

Nodes are uniquely identified with an integer identifier, for example deterministically computed using their IP addresses. Further, we assume that nodes can generate prime numbers, and have access to cryptographic primitives (RSA encryptions and signatures), which are supposed to be unbreakable. In addition, nodes have a couple of public/private keys used to generate signatures and encrypt their communications. Classically, we will denote $p_k(X)$ the public key of a node X, $\{a\}_p$ the encryption of a message a using the key p, and $< a >_p$ the signature of a using p.

5.1.2 Gossip sessions

We assume that several gossip sessions disseminating different contents can hold simultaneously in the system. Each content is generated by its source, and is signed with its private key. Updates are propagated along with their signature so that they can be verified by the nodes upon reception, which prevents data tampering. Nodes interested by a content have to obtain the public key of its source using an external service.

In the illustrations of this document, we represent the interest of nodes, which are kept private, as a color, e.g., in Figure 5.1, nodes A and C are colored in black and are assumed to be interested in the same content.

5.1.3 Nodes behaviors

We consider that nodes can be of several types.
Correct nodes strictly follow the protocol and participate actively in the dissemination of updates. In particular, the source of each session is assumed to be correct.
Selfish nodes are interested in increasing their benefit and minimizing the cost they pay for their participation in the protocol, and deviate from the protocol in any way that would improve their benefit (e.g., reduced bandwidth consumption, reduced CPU overhead). These nodes would deviate from properties $\mathbf{R_1}$, $\mathbf{R_2}$, and $\mathbf{R_3}$ if allowed to. We assume that selfish nodes do not collude.

5.1.4 Adversary model

We consider a global and active opponent, which is the strongest possible model of attacker. Global means that the opponent can monitor and record the traffic on all the 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 goal of the opponent is to endanger properties $\mathbf{P_1}$, $\mathbf{P_2}$ and $\mathbf{P_3}$. The higher the number of nodes that the opponent must control to break a protocol, the stronger the privacy guaranteed by this protocol. The only limitation of the global and active opponent is that it is not able to invert encryptions.

SECTION 5.2
PAG Overview

In this section, we present an overview of PAG. We start by presenting how the membership of the whole accountable and privacy-preserving gossip protocol is maintained in part 5.2.1. Then, we detail how nodes select their successors in part 5.2.2. We further present how nodes monitor each other to enforce accountability in part 5.2.3. Finally, we introduce the cryptographic procedures that preserve privacy against a global and active opponent in part 5.2.4.

5.2.1 Global membership and monitoring

The role of a membership protocol is to handle the arrival and the departure of nodes, as well as the distribution of the membership list to nodes. To provide nodes a reasonably up-to-date view of the membership we rely on FireFlies [25, 63] which is a scalable Byzantine-resistant membership protocol. Fireflies ensures that the membership is known by every node in the system with probabilistic guarantees on the delay, and that unresponsive nodes are detected and evicted by their neighbors. We also use FireFlies to assign random and live monitors to nodes in order to enforce accountability in gossip. A new node can be inserted in the system using an access point that is either a tracker, or an already inserted node whose identity is made public.

Figure 5.1 shows an example of how FireFlies handles nodes membership. In this figure, nodes are organized on several rings using random permutations. The positions of a node on the rings, which depend on its identifier, defines the nodes it monitors and

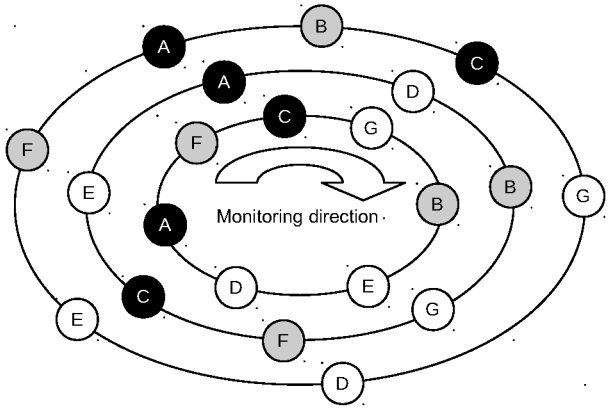


Figure 5.1 – Membership using FireFlies [25]

those it is monitored by. For example, node A is monitored by nodes F, E and D (on the outer, middle and inner rings, respectively), and monitors the nodes B, D, and F (on the outer, middle and inner rings, respectively).

5.2.2 Exchanges of updates using gossip

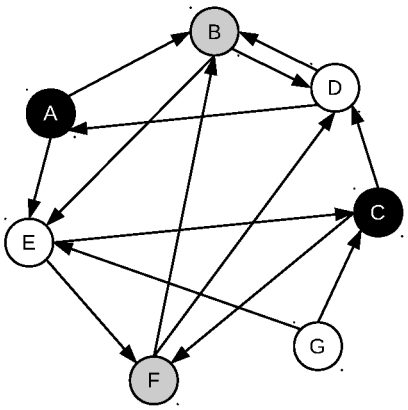


Figure 5.2 – Exchanges of updates between nodes that have different interests

To exchange updates and obtain the content they are interested in, nodes apply the gossip paradigm, and interact with randomly chosen nodes in the membership list. Nodes use a pseudo-random number generator deterministically seeded to choose their successors and forward them their most recently received updates. More precisely, during round $R + 1$ nodes will forward the updates they received during round R to the nodes they have randomly chosen. In Figure 5.2, we give an example of the associations that could occur between the nodes of Figure 5.1 during one round. Each of them selected two random nodes to which they would forward the updates they previously received. For example, node A forwards all the updates it received, independently of its interests, to nodes B and E. The content exchanged in these interactions could be either white, gray or black.

5.2.3 Enforcing accountability using monitoring

To ensure the accountability properties $\mathbf{R_1}$, $\mathbf{R_2}$ and $\mathbf{R_3}$, we rely on a monitoring infrastructure. Each node is assigned a set of monitors, which are its successors on the membership rings of FireFlies. The monitoring relations between nodes are thus known from every node in the system. To detect the deviations of a node, at least one of its monitors need to be correct. In some cases, nodes can exhibit some of the messages they have sent or received to prove their correctness, or that another node deviated from the specification of the protocol (avoiding it is future work).

To ensure a good dissemination of updates nodes need to have random successors (property $\mathbf{R_3}$), however the randomness of associations must be verifiable. To combine both goals, we rely on random yet deterministic associations. To do that, nodes use a pseudo-random number generator to determine their successors among the full membership using a deterministic seed (e.g., obtained from the encryption of the most recently received updates) that only its monitors can verify. The monitors of a node are the only nodes in the system that are able to predict, and later verify, its interactions with other nodes. During the dissemination of updates, a node will typically receive an update at round R and forward it during round $R + 1$. Its monitors are in charge of checking this forwarding during round $R + 1$, which provides property $\mathbf{R_2}$. As it will be presented in Section 5.3, property $\mathbf{R_1}$ comes from the fact that nodes cannot avoid to receive updates they cannot prove to have received in the past.

Figure 5.3 presents a simplified example, in the sense that no signature or encryption are represented, of how the monitoring infrastructure checks that a node correctly forwards the updates it receives. Details of the protocol will be presented in section 5.3, we only give here the intuition of the monitoring process. In this example, node A receives an update u, and has to forward it to node B during round R. Upon reception of this update u, node B acknowledges to its monitors A, D and G, the reception of this update from node A, using the $Ack(u, A)$ message. The monitors of node B transmit this information to the monitors of node A, namely nodes E, D and F, using the $Confirm(u, A, B)$ message. Thus, after these messages, the monitors of node A are able to check that it (i) forwarded the right message, and (ii) correctly chose its successor. Meanwhile, the monitors of node B have learned that it received the update u. During the following round, it is their task to check that node B correctly forwards this update. Finally, the monitoring infrastructure controls the dissemination of updates at each hop, and finally along the whole dissemination path. A special case occurs when the source of a content sends its updates to some nodes. In this case, the source has to inform the monitors of those nodes that they received some updates to forward.

We now briefly explain why nodes have interest in transmitting each kind of message represented in Figure 5.3. Let us assume that node A received an update u, and that its monitors are aware of this reception. In this case, A's monitors expect to receive a message $Confirm(u, A, B)$ where node B is the successor that node A has to choose. If they do not receive this message, they would accuse node A of a selfish deviation (i.e., avoiding to forward an update). If this message is not received, it is either because node A did not send the update u to node B, or because node B did

not send the $Ack(u, A)$ message to its monitors (remember that at least one monitor of node B is assumed to be correct). In reality, when node A sends an update to node B, it expects a signed acknowledgement that it can use to prove that it correctly forwarded u to node B. If not, it will not be able to prove its correctness and finally will be evicted from the system. If node B received the update u and acknowledged it to node A, it will send the $Ack(u, A)$ message because node A has a proof that it received the update, and expect its monitors to receive the $Confirm(u, A, B)$ message.

Finally, using this monitoring infrastructure nodes are forced to interact with correctly chosen successors, receive updates and forward them.

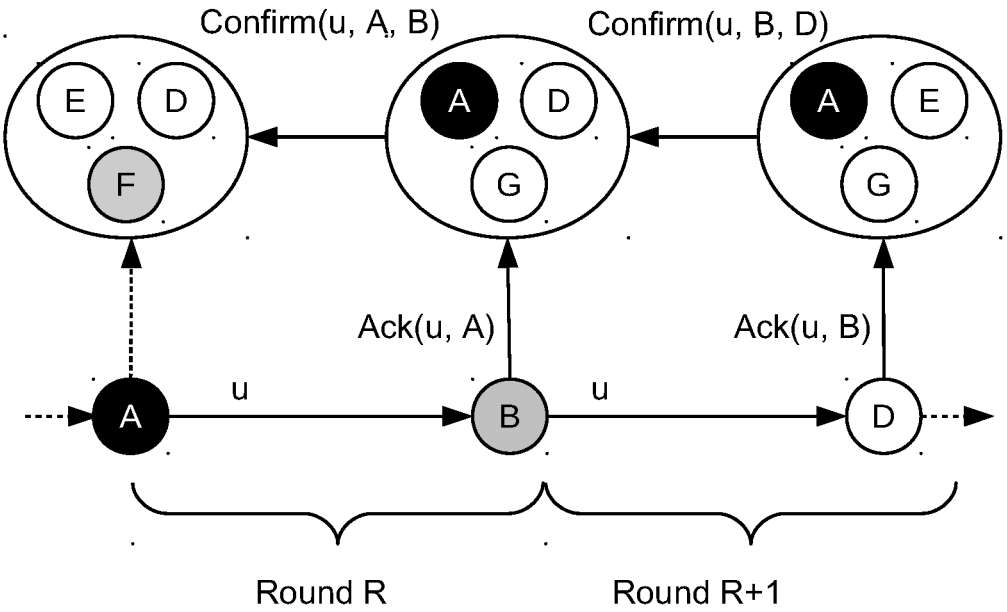


Figure 5.3 – Monitoring of nodes to ensure the forwarding of updates

5.2.4 Enforcing privacy using homomorphic encryptions

The monitoring infrastructure we presented ensures that nodes forward the updates they receive. We now present the homomorphic procedures that allow the encryption of updates, preventing monitors to learn which updates nodes exchange. This encryption ensures that a monitor is not able to understand which updates a node receives or forwards, but that is still able to check that the forwarding is done correctly. Simply relying on updates encryption is not sufficient, 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. Combining the two information, an attacker would be able to break the privacy of all exchanges.

Homomorphic encryption. Monitors verify that nodes correctly forward the updates they receive. To allow these verifications, we use an encryption process that is close to an unpadded RSA encryption, and exploit two interesting properties. Suppose that the public key consists of a modulus m and an exponent e, then the encryption of an update

u is given by $\{u\}_{(e,m)} = u^e \mod m$. Let u_1 and u_2 be two updates. The following homomorphic property can easily be established:

$$\{u_1\}_{(e,m)} \cdot \{u_2\}_{(e,m)} = \{u_1 \cdot u_2\}_{(e,m)} \tag{5.1}$$

Let e_1 and e_2 be two exponents. In addition to the previous property, this encryption also verifies:

$$\left\{\{u\}_{(e_1,m)}\right\}_{(e_2,m)} = \{u\}_{(e_1 \cdot e_2,m)} \tag{5.2}$$

It is not possible to inverse the encryption of updates as the value of the modulus m is smaller than the size of the encrypted content. Any hash function presenting these two homomorphic properties could be used to check the dissemination of updates, however, we are not aware of such functions. Instead, we use this hash mechanism, which is close to the one used in the RSA encryption, but cannot be inversed, and is not costly when the modulus size is not too high (256 or 512 bits is probably enough in most cases).

Intuition. Figure 5.4 illustrates the intuition of PAG. Nodes A and F are the two predecessors of node B, and node D is a successor of node B. We consider only 2 predecessors for node B for the sake of simplicity, even though 3 is a minimum to ensure privacy (for this reason, we use at least 3 successors per node). We only focus on the reception and forwarding of the updates that node B receives. The same steps would also apply to the nodes A, F and D to provide a kind of forwarding chain but are not described for the sake of simplicity.

NOTE ►*La figure doit être mise à jour!*◄

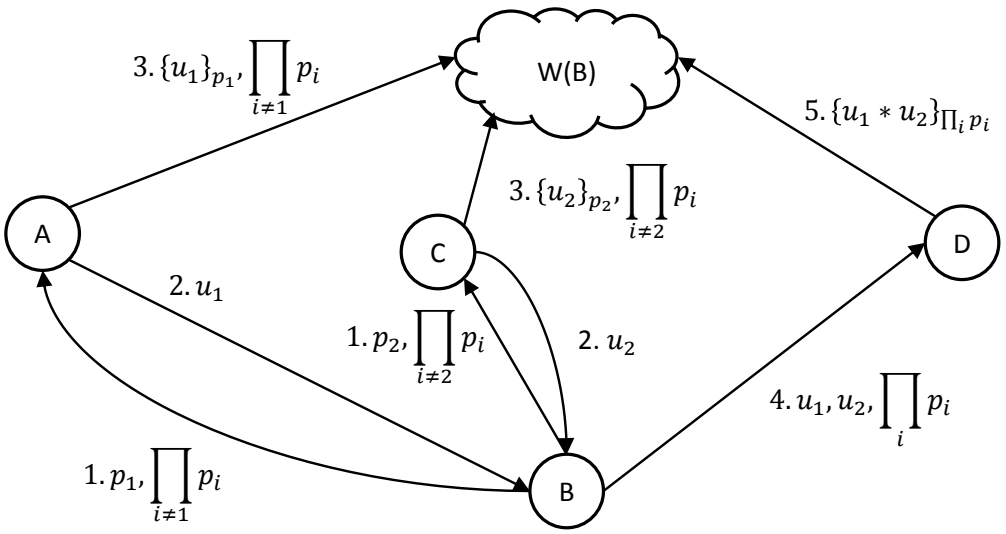


Figure 5.4 – Privacy preserving verification of a forwarding of a node B

The set of monitors of node B is made of nodes A, D and G. Let us suppose that nodes A and F have to respectively send updates u_1 and u_2 to node B. First, they would ask node B to send them a prime number. Node B chooses two prime numbers

p_1 and p_2 and respectively sends $(p_1, \prod_{j \neq 1} p_j)$ and $(p_2, \prod_{j \neq 2} p_j)$ (messages 1.) to nodes A and F. Nodes A and F can then send their two updates to node B (messages 2.), which would be encrypted. Nodes A and F also have to declare (messages 3.) to the monitors of node B, that they sent some updates to node B whose encryptions are respectively equal to $\{u_1\}_{(p_1,m)}$ and $\{u_2\}_{(p_2,m)}$. Later, when node B will have to forward its updates u_1 and u_2 to node D, it will join the product $\prod_j p_j$ (message 4.). Node D will be able to acknowledge the reception of u_1 and u_2 using the encrypted value $\{u_1 \cdot u_2\}_{(\prod_j p_j, m)}$. The monitors of node B can verify that the following equation is verified, which proves that B did forward exactly what it received:

$$\left(\{u_1\}_{(p_1,m)}\right)^{\prod_{j \neq 1} p_j} \cdot \left(\{u_2\}_{(p_2,m)}\right)^{\prod_{j \neq 2} p_j} = \{u_1 \cdot u_2\}_{(\prod_j p_j, m)} \tag{5.3}$$

Privacy of exchanges. Although incomplete this short example shows the main idea of the dissemination protocol: the monitors of a node are able to check that what a node receives is forwarded without learning the actual content, thus preserving the privacy of the node being monitored. To obtain the content of exchanges it would be necessary to learn the prime numbers a node chose. With this information the monitors would decrypt the exchanges a node had with its predecessors, or successors. Predecessors and monitors of a node receive the product of prime numbers, and are not able to factorize it efficiently, as it is a notoriously known hard problem. There is currently no polynomial time algorithm to factorize integers [4], even though no proof of difficulty has been published.

SECTION 5.3

Design of PAG

We have presented the principles of PAG in the previous sections. In this section, we detail the steps of the protocol, which has been designed in such a way that selfish nodes can not deviate from the protocol without providing at least another node with a proof of misbehavior it can use to evict this node from the system.

5.3.1 Forwarding updates

Two important requirements guided the design of PAG. First, the encryption of updates must change from one forwarding to the other. Otherwise, a global attacker could follow the transmission of updates in a system. Second, for performance reasons, it must be possible to combine the encryption of several updates to reduce the overhead of the protocol. The use of a classical hash function does not provide these points. To the best of our knowledge, our protocol is the first to provide both these properties.

Figure 5.5 presents the exchange of messages that occur when a node A has to forward a set S_A of updates to a node B, which already possess the set of updates S_B, during round number R. First, node A asks to node B for a prime number that it will later use to encrypt the product of the updates in S_A (message 1.). For this step, node B has to wait for all its predecessors to ask for a prime number before answering them. We note $K(R, B)$ the product of the prime numbers that node B chose during round R to

receive updates from its predecessors.

In message 2., node B replies with the primary key p_j that B must use in a signed and then encrypted message. It also joins the homomorphic encryption of the updates in S_B using p_j (for optimization reasons, it may be possible to encrypt only a subset of S_B). 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.

In message 3., node A serves in an encrypted, and then signed, message using node B's public key the updates in $S_A \setminus S_B$ 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 it to acknowledge the reception of the updates in a message encrypted using its public key $K(B)$.

In message 4., node A sends to node B a signed attestation that declares the value of the encryption of the updates 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 updates in S_A using the encryption of their product with $K(R-1, A)$. If necessary, node A can use this message as a proof that it did forward the right set of updates to node B during round R.

5.3.2 Encrypting a set of updates

The role of monitors is to check that the node they monitor (i) contact all its successors, (ii) forward all the updates it received at round R during round $R+1$. For this last verification, monitors have to compute the homomorphic encryption of the product of all the updates that the node receives during a given round, and check that its successors during the following round acknowledge this encryption.

Figure 5.6 illustrates the mechanisms that allow the monitors to perform these tasks. At each round, the monitors of a node expect to receive messages from it, and from the monitors of its successors. In this figure, the monitors of node B are nodes A, D and G, while the monitors of its predecessor, node A, are nodes D, E and F.

Monitoring details. When node B receives the set S_A of updates from node A, it has to send two messages to one of its own monitors. The first message (message 6.) is a copy of the acknowledgement that node B already sent to node A in (message 5. of Figure 5.5). Message 7., which is signed, contains the attestation that node A sent in message 4. of Figure 5.5, and the product of the prime numbers that node B used to receive updates from its other predecessors during round R.

The monitor, here node D, that receives these two messages from node B has to compute

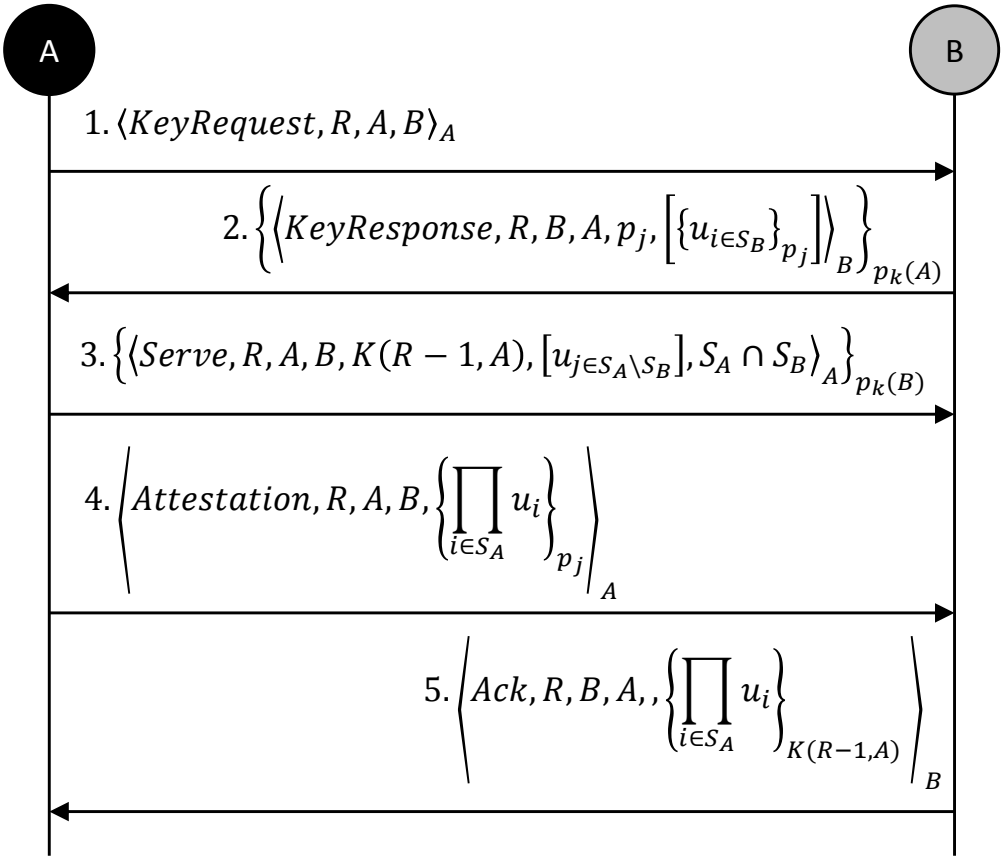


Figure 5.5 – Propagation of messages inside a session

the value $\left\{\prod_{i\in S_A} u_i\right\}_{(p_j,m)}^{(\prod_{k\neq j} p_k,m)} = \left\{\prod_{i\in S_A} u_i\right\}_{(\prod_k p_k,m)} = \left\{\prod_{i\in S_A} u_i\right\}_{(K(R,B),m)}$ and broadcast it to the other monitors of node B, which are nodes A and G, along with message 6. To be sure that a monitor correctly computes and forward the encryptions of updates, a node can inform her other monitors before sending a message to one of them.

5.3.3 Combining all encryptions

During a round, and after each broadcast, each monitor of node B computes the product of all the encryption values forwarded by the other monitors of node B. Finally, at the end of the round, the monitors of node B knows the encryption of the set of updates that node B received using the product of the prime numbers that node B chose. This encryption must be acknowledged by the successors of node B during the following round, allowing its monitor to validate its forwarding.

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

$$\{S_A \cup S_F\}_{(\prod_j p_j,m)} = \{S_A\}_{(\prod_j p_j,m)} \times \{S_F\}_{(\prod_j p_j,m)}$$

To allow this verification, the monitor that has been contacted by node B also has to forward the acknowledgement (message 9.) to the monitors of node A, which are node D, E and F. The monitors of node A can then verify that node B received the correct set of updates from node A.

5.3.4 Practical implementation details.

While the main ideas of the protocol have been presented, important details have to be ruled out before having a practical implementation of a gossip protocol. In this section we present the important details or optimizations that allow the protocol to become practical.

Number of monitors per node. To maximize the privacy of exchanges while minimizing the bandwidth overhead of the protocol, we advise to select the same number of monitors and successors per node.

Updates encryptions. A node is able to communicate to its predecessors the encryptions of a fraction of the updates it owns, in order to avoid to receive them again. Determining how many encryptions to send is dependent on the applications, and more particularly on the updates' and on the size of their encryptions. In our scenario, updates were bigger than their encryptions, and the best results in terms of bandwidth consumptions were obtained when the updates of the last 4 rounds were encrypted and transmitted.

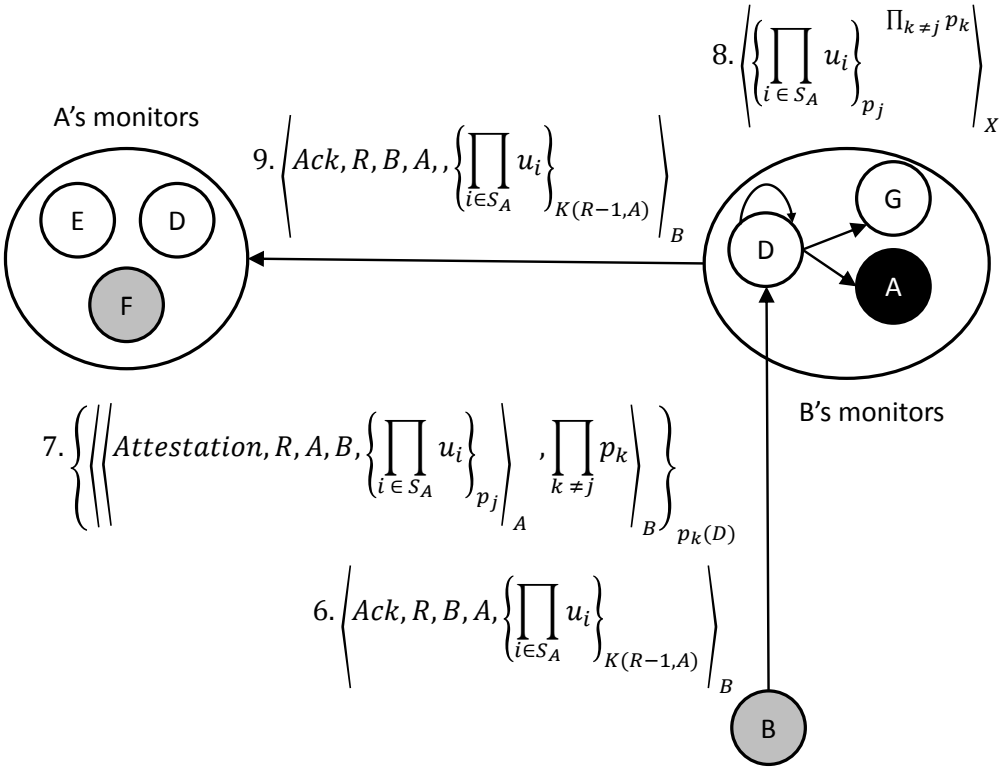


Figure 5.6 – Monitoring part of an interaction between two nodes

Simultaneous multiple receptions of an update from several predecessors. While our protocol limits the possibilities for a node to receive several times a given update, it can still occur. More particularly, it is possible when a node simultaneously receives updates from different predecessors. However, to limit bandwidth consumption it is necessary to 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 compute the encryption of the set of received updates, and the monitors to match the encryptions of received messages with the one of forwarded messages.

Allowing updates to disappear. Generally, updates have a date of expiration 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 sends updates 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 have to be forwarded. A small modification of the messages and monitoring exchanges allow updates to expire. The monitors of a node would check the propagation of the second list, and still acknowledge the reception of the first list.

Increasing the size of updates. It is possible to reduce the bandwidth overhead of our protocol if the size of updates is increased. Indeed, the propagation of updates is

checked through their encryption. Transferring more data using a single encryption is more efficient. For example, using 5 updates instead of 40 to disseminate a 300Kbps stream allows the total bandwidth consumption to go from 1100Kbps to 600Kbps in a system with 1000 nodes. Such optimization is up to the system designer, and in our experiments to have fair comparisons with other systems we consider updates of 938 Bytes.

SECTION 5.4

Security, privacy and accountability

When nodes participate in PAG, they have to receive and forward all the contents that are being disseminated in the system. However, as nodes have a finite amount of bandwidth, PAG may be too costly depending on the contents being shared. In this section, we present an optimization of PAG that we named $k - PAG$, which reduces its bandwidth overhead by grouping sessions into clusters of k sessions. We show that properties $\mathbf{P_1}$ and $\mathbf{P_2}$ are enforced with k-anonymity (part 5.4.1). We then present the results of the proof of security we made using ProVerif, which is an automatic cryptographic protocol verifier, that proves that property $\mathbf{P_3}$ holds against a global and active attacker if it controls less than f nodes (part 5.4.2). Finally, we provide details about the incentives that enforce properties $\mathbf{R_1}$, $\mathbf{R_2}$ and $\mathbf{R_3}$ (part 5.4.3).

5.4.1 $k - PAG$: clustering sessions

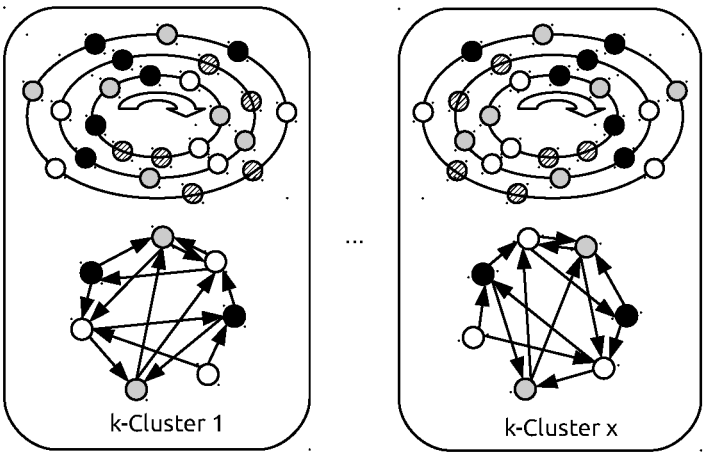


Figure 5.7 – $k - PAG$ illustration with k=3

As depicted in Figure 5.7, sessions in $k - PAG$ are grouped into clusters of a predefined minimal size, i.e., k. As such, a node inserted into a cluster receives k contents, instead of receiving the S contents that are disseminated in the whole system. Similarly to PAG, the membership of nodes as well as the assignment of monitors is performed using one FireFlies ring structure per cluster.

According to [78], a system ensures k-anonymity if the information released about a user cannot be distinguished from those of $k - 1$ individuals whose information

also appear in the release. In $k - PAG$, as k contents are distributed in each cluster, it is not possible for an observer to determine which one of the k contents a node is interested in. Thus, the property privacy $\mathbf{P_1}$ is ensured with k-anonymity in $k-PAG$.

Not being able to determine the list of nodes that are interested in a content is a consequence of the previous privacy property. If the system contains k different contents and N nodes, it is not possible to determine which repartition of nodes among the C_N^k possible subsets of k nodes among the N is the right one. Thus property $\mathbf{P_2}$ is also enforced.

It is interesting to study the extreme values of k. If k is equal to S then k-PAG is identical to PAG. On the contrary, if k equals 1, then property $\mathbf{P_1}$ and $\mathbf{P_2}$ do not hold, as the cluster a node is inserted in directly refers to the content it is interested in.

To receive a content, a node must be inserted into the membership list of the cluster that disseminates the content it is interested in. To do that, it can contact any node of this cluster in order to be inserted in the cluster's membership list.

To enforce k-anonymity, the creation of a new k-cluster is done only when an existing cluster reaches a size equal to $2 * k$ sessions. At that time, the cluster of size $2 * k$ is split into two clusters of size k. As such, at any point in time, the interest of any node in the system is hidden among at least $k - 1$ other contents. To meet this objective even at the beginning of the system, or if at any point in time there are less than k parallel sessions in the system (i.e., $S < k$), nodes have to wait for the starting of enough sessions to enforce k-anonymity. An alternative, which we will investigate in future work, would be to generate fake gossiping sessions. Finally, if sessions inside a cluster do not have the same duration, the source nodes of the shortest sessions send garbage data while waiting for the others to end.

5.4.2 Enforcing $\mathbf{P_3}$ under global and active attacks

ProVerif [52] is a well-known automatic cryptographic protocol verifier that uses Horn clauses to detect possible attacks. Using ProVerif, we modeled the cryptographic mechanisms of PAG[1] (that were illustrated in Figures 5.5 and 5.6). The aim of using this experiment is to show that there is no attack on the privacy property $\mathbf{P_3}$ that involves less than f nodes, where f is the number of predecessors, successors and monitors per node.

We considered the representative situation where a node B receives updates from three predecessors A_1, A_2 and A_3, and have to forward them to one of its successors C. Checking that node B correctly forwards the updates it receives also implied to instantiate the monitors of nodes A_1, A_2, A_3, B and C. We modeled the case where $f = 3$, because 3 is the smallest value where the protocol can be proved secure. Increasing the value of f would reinforce the security of the protocol, as the necessary number

[1]The code is available in appendix and at
https://gist.github.com/anonymous/5d9d542ffa47e1f64a7a

of colluding nodes sharing information in order to break the privacy would also increase.

We assume that node B is correct, otherwise, its exchanges may not be kept private, even without exterior attacks. In addition, we consider that the aim of an opponent is to obtain the value of a prime number that node B chose for one of its predecessor in order to obtain the detail of the exchange between node B and this node. We also 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, once the prime number used is known, the attacker would have to encrypt 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 subset of a set of size N is equal to 2^N, but we choose not to ignore it. We thus make the assumption that the attacker has an accomplice that communicates all the updates of the session, or that the attacker receives the content.

We modeled several attack scenarios to assess the privacy property $\mathbf{P_3}$. The model of the protocol can be found in Appendix .1. These scenarios can be grouped under two cases:

- **Case (1).** The attacker listens to all communications on the network, and actively 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 proved that no attack exists on the cryptographic procedures of PAG. Our experiments in case (2) allowed us to 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. If the colluding monitors of a node receive the right messages, and are controlled by the attacker, then the opponent is able to obtain the private numbers that B generated and thus gain access to the identities of the updates a node received. As we said, this attack is very costly, but we can not underestimate it.

5.4.3 Accountability against selfish deviations

If nodes execute correctly their exchanges, as specified by Figure 5.5, then properties $\mathbf{R_1}$, $\mathbf{R_2}$ and $\mathbf{R_3}$ are enforced. In the following, we briefly explain the incentives that force a selfish node, say node A, to follow the protocol. Remember that nodes register the messages they send or receive, and can use these messages to prove their correctness or that another node deviated.

Random successors. At the beginning of a new round, node A contacts its successors asking them to answer with a prime number to encrypt the updates it must forward (message 1). A selfish node will execute this step correctly because the identities of its

successors are known by its monitors, and they will check that the exchanges take place (through message 9 of Figure 5.6). Thus, property $\mathbf{R_3}$ is enforced by the monitoring infrastructure. The successors of node A answer with a prime number, and use it to encrypt some updates they wish to avoid to receive again (message 2). Correctly following this step is in the interest of selfish nodes as it will minimize the number of updates they will receive. In addition, a node can not avoid to receive updates it does not have, which enforces $\mathbf{R_1}$.

Serving updates. Node A computes the set of updates that its successor does not have and send them, along with the identifiers of the updates it should have sent but that its successor already had (message 3). If a node does not send the right set of updates to its successors then the verification its monitors will run will fail. Eventually, as its successors received 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 5.6). 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 $\mathbf{R_2}$.

We now detail for each step of the protocol the incentives that encourage selfish nodes to follow the protocol. For each step of the protocol, we identify the various deviations that selfish nodes could follow and provide the associated incentive.

- Fig 5.5 Step 1. At the beginning of a new round node A sends a KeyRequest message to each of its successor. Suppose that node A contacts node B during this step.
  - *Selfish deviation 1.* Node A sends an incorrect message to node B, or does not choose the right successor.
  - *Incentive.* An incorrect message, or a message sent to the wrong destination would constitute a proof of misbehavior that node B could held against node A.
  - *Selfish deviation 2.* Node A does not send any message.
  - *Incentive.* If node A does not send any message to one of its successor during a round, say node B, its witnesses will not receive any message from node B, and will want to find out who is guilty.
- Fig 5.5 Step 2. Node B replies with a prime number to node A, and the encryption of some of the updates it already has using the prime number it chose for node A.
  - *Selfish deviation 1.* Node B sends an incorrect number (e.g., not a prime number) or the encryptions of updates are incorrect.
  - *Incentive.* Node B can verify that the number it received is a prime number, and denounce node A if the number is not prime. If updates are not correctly

encrypted, node A will send all the updates it has to propagate. It is not in the interest of node B to receive updates it already owns.

- Fig 5.5 Step 3. Node A computes the set of updates that node B does not have and thus that it should send, it also computes the indexes of the updates that node B has. It can then send these information along with the key that node B will use to acknowledge the reception of updates from node A.
  - *Rational deviation 1.* The key is not correct.
  - *Incentive.* If the key is not correct, the verification that node A will eventually execute will fail, and node B has a signed message that will prove that node A is faulty. It is possible to obtain the key that node A used by asking its predecessors what it is.
  - *Rational deviation 2.* The set of updates and the set of indexes are not correct.
  - *Incentive.* Once again the verifications will fail and it will be possible for any node in the session to see that node A did not sent the right set of updates.
  - *Rational deviation 3.* The updates are modified by node A.
  - *Incentive.* If the updates are modified by node A then the source's signature that is propagated with them will not match the content.
- Fig 5.5 Step 4. Node A sends an attestation to node B. This message contains the encryption of the product of all updates that node A sent using the prime number that node B chose. This encryption is then used by node B's witnesses to check the forwarding of these updates.
  - *Rational deviation* The encryption is incorrect.
  - *Incentive.* If the encryption is incorrect node B must denounce it, if it does not the acknowledgements from node B's successors will not match with the attestation, and node B will be denounced.
- Fig 5.5 Step 5. Node B sends an acknowledgement to node A using the key that node A used to forward the updates it received during the previous round.
  - *Rational deviation 1.* It is not the correct set of updates that is encrypted.
  - *Incentive.* If the encryption is incorrect, node A should not accept it. The combination of the serve message (step 3.) that node B received and the acknowledgement that node A received would prove that node A is correct while node B is not.
  - *Rational deviation 2.* It is not the correct key that is used.
  - *Incentive.* The same incentive as above holds.

SECTION 5.5

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 5.5.1). We then evaluate the proportion of exchanges that an active and global attacker could discover if it controls more than f nodes in the system (part 5.5.2). We further evaluate the overhead of our protocol in terms of bandwidth consumption using both simulations and real code deployments compared to state-of-the art competitors (parts 5.5.3 and 5.5.4) while varying the values of k in $k - PAG$, as well as the size of the content being disseminated (part 5.5.5). Finally, we evaluate the cryptographic costs of PAG (part 5.5.6) as well as the scalability of PAG with respect to the number of users (part 5.5.7).

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

5.5.1 Methodology and Parameter Setting

PAG **and its competitors.** To assess the performance of PAG compared with other solutions, we implemented it in Java and used it as a video live streaming application. In this context, a source node broadcasts a video stream at a fixed rate, during 5 minutes, and sends each generated update to 3 random nodes. When it is not precized, PAG is configured with 3 monitors per node. Updates are then disseminated using PAG or one of the protocols we compare ourselves with. Among these protocols are an accountable gossip protocol, and two anonymous communications protocols. $AcTinG$ [5] is an accountable gossip protocol that is not designed to preserve the privacy of nodes as they maintain a secure log, and audit each others. To give a fair comparison, we also choose to compare PAG to two anonymous communication protocols. **RAC** [40] is an anonymous communication system that forces nodes to relay the messages that other nodes send. Using it, a source could send a content to all nodes anonymously and with accountability. We do not study Dissent [39] at it was shown to be even more costly than RAC in [40]. **OR+PR** is a combination of the Onion Routing [71] protocol with PeerReview [29]. This protocol forces nodes to relay the messages they receive. This second anonymous communication protocol is less costly, and its advantage is that its throughput does not depend on the size of the membership. Using PeerReview allows the relaying of messages to be checked. However, it cannot be considered privacy-preserving as nodes running PeerReview maintain a secure log. Studying this protocol gives an insight on the cost of an hypothetical ideal protocol that would combine anonymity with accountability.

Real deployment settings. We deployed PAG on 48 machines of the Grid5000 cluster,

using 9 instances per machine, thus totaling 432 nodes. The machines were interconnected using a 1Gb/s network. Each machine is composed of an Intel Xeon L5420 processor clocked at 2.5Ghz with 32GB of RAM. To provide further tolerance to message loss (combined with retransmissions), a source groups packets in windows of 40 packets, including 4 FEC [2] coded 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. The cryptographic primitives consisted in 1024-bit RSA signatures. The size of the generated prime numbers and of the modulus used in the homomorphic encryptions are set to 512 bits.

Simulations settings. Our simulations consisted in an implementation of the protocol in the OMNeT++ [86] simulator, using the same parameters value we used in our deployment. The simulation code is a C++ version of the one we deployed in the previous experiments. We also used computations to obtain the scalability of the protocol and its memory consumption when the number of nodes was too high to be observable in practice.

5.5.2 Probabilistic study of the impact of coalitions

A coalition of at least f nodes can break the privacy of some interactions of nodes in the system. We now evaluate the privacy guarantees of PAG in presence of a global and active attacker. We also compare these guarantees to those of an existing state of the art protocol, i.e., $AcTinG$ [5].

In AcTinG audits check the secure logs of nodes, which contain the detail of up to 30 interactions, with a probability of 10% whenever a new interaction occurs. The aim of the global and active attacker we consider is to break the privacy of exchanges by controlling several nodes. Corrupted monitors reveal the identity of nodes that exchange updates. Merging the knowledge of several nodes, it is possible to obtain more information about users in the system. Apart from learning the identities of partners, 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 decrypt them. The probability of this situation to happen is however low enough for the protocol to be practical.

To evaluate this risk, we consider the probability for an exchange between two nodes, which can be controlled by the attacker, to be discovered by the attacker. Depending on the size of the coalition, we want to evaluate the probability that the privacy of an exchange is broken and observed by the attacker. Attacks have to be evaluated in terms of probabilities because nodes are randomly affected predecessors and successors in their session, and monitors among the entire membership.

In Figure 5.8, we evaluate the proportion of all exchanges in a session that an attacker, which controls a variable proportion of the membership, can discover. The ideal privacy

[2] FEC stands for Forward Error Correction.

in this case is represented in black, and express the probability that at least one of the two nodes that interact is corrupted, and divulge the content of the exchange. In case of a collective attack, increasing the number of monitors, and the fanout of nodes, makes the privacy guarantees close to ideal. The code that was used to create this figure can be found in Appendix .2.

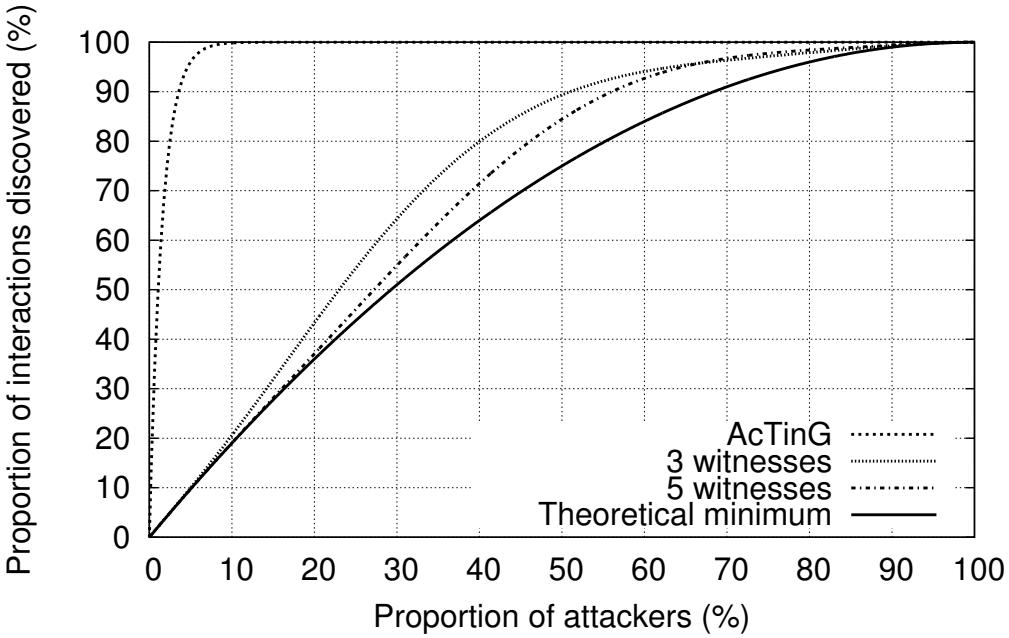


Figure 5.8 – Privacy presence of a global and active attacker controlling a varying proportion of the membership.

5.5.3 Comparison with an accountable gossip protocol and impact of the number of contents

We first compare the bandwidth consumption of our protocol to the one of $AcTinG$ [5]. We present in Figure 5.9 the cumulative distribution functions of the bandwidth consumptions of nodes during a 300 Kbps streaming session. In average nodes running $AcTinG$ consume 460 Kbps, while using PAG they need 1050 Kbps.

The privacy of nodes increases with the number of 300Kbps contents per cluster being disseminated. This number ranges from 1 to 3, and the associated bandwidths are respectively equal to 1050, 2075 and 3092 Kbps. This cost is not exactly linear with the number of session, as some messages (e.g., the encryption of updates) can be factorized when a node receives several contents.

The biggest part of the cost is due to the forwarding policy: what is received by a node at round R must be forwarded at round $R+1$. The same update may be received several time by a node and then forwarded, and we cannot avoid it currently. $AcTinG$ is less costly because nodes can refuse updates, and it is controlled using their log during audits. Increasing the number of monitors does not significantly increase the bandwidth

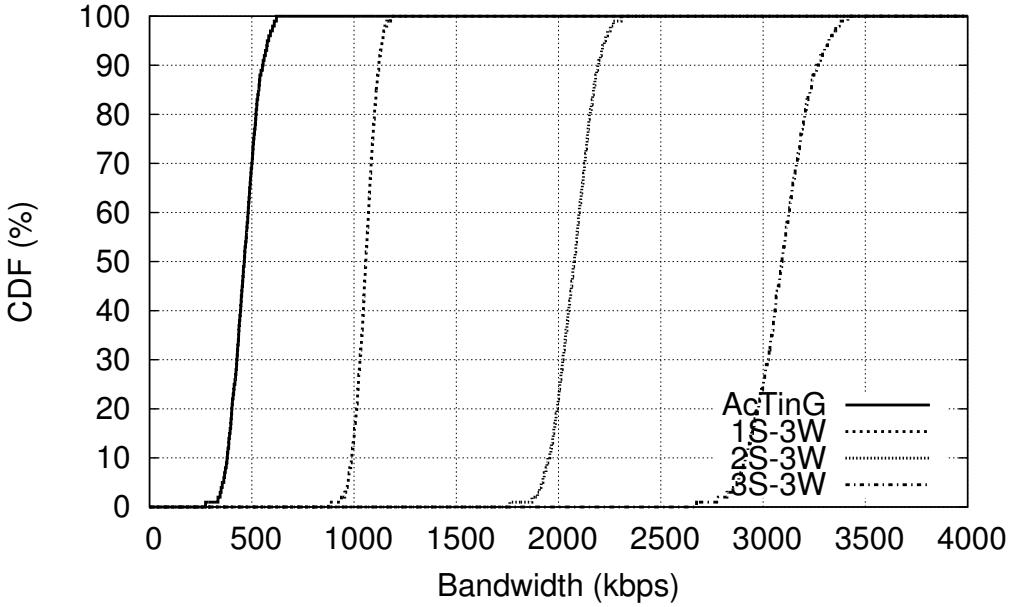


Figure 5.9 – Bandwidth consumption of $k-PAG$ with several 300kbps contents per cluster and 3 monitors per node [sim]

cost of the protocol, and allows a better resilience to collective deviations between nodes. However, having more monitors than successors/predecessors does not increase the privacy guarantees.

In Figure 5.10, we measure the average bandwidth consumption of our protocol using 3 monitors, and successors per node. After the initialization of the session, the average bandwidth consumption of node is established around 1000Kbps.

5.5.4 Comparison with anonymous communication systems

Relying on anonymous communication systems to run a gossip protocol would provide privacy to nodes, however it would not force nodes to participate actively in the dissemination of updates. These protocols are costly and cannot be applied in the exact same settings we used with PAG and $AcTinG$. Thus, we designed a second set of experiments that consisted in determining the maximum video quality that protocols could provide in function of the network capacity. We present in the first two lines of Table II the video qualities we considered and the associated payload size.

Table I summarizes the results of our experiments 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 a network of 1.5Mbps $AcTinG$ can provide a 480p video using 1.4Mbps.

As we have seen previously, PAG is more costly than $AcTinG$ which is also accountable but not privacy preserving. Using 10Mbps network links, PAG can provide at most a 480p video, consuming 6.9Mbps of bandwidth. In comparison, $AcTinG$ would

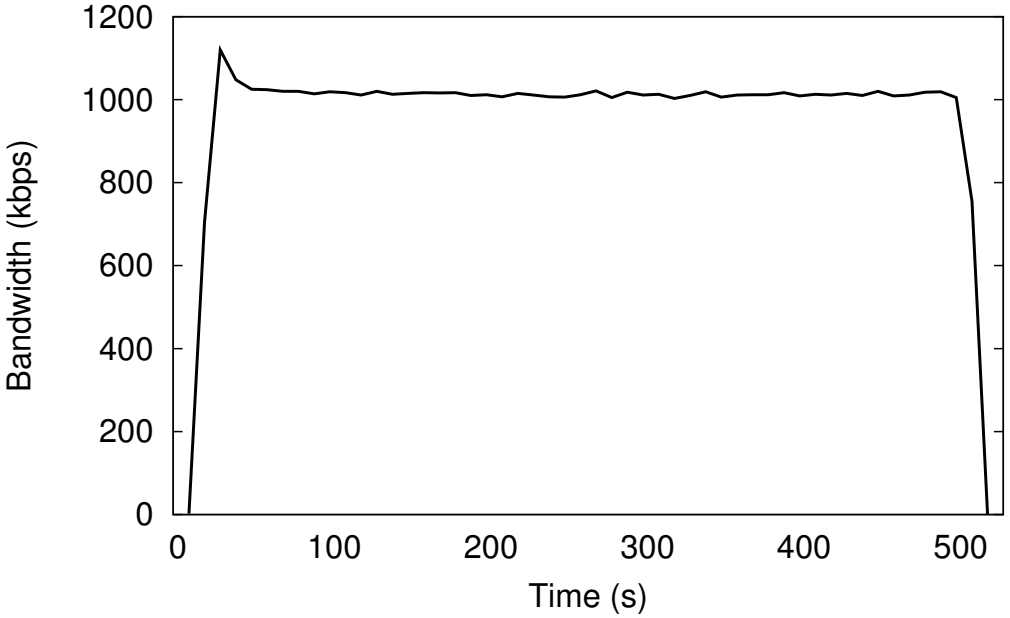


Figure 5.10 – Average bandwidth consumption of nodes running PAG with a 300Kbps payload.

be able to send a 1080p video using 6Mbps of bandwidth. Decreasing the video quality in exchange for privacy is a tradeoff that users may be willing to take.

However, anonymous communication systems would not provide accountability, and 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. In comparison, OR+PR is a better solution which would need at least a 1Gbps network to provide a 1080p video, consuming 103Mbps. In comparison, PAG would send a 1080p video consuming only 31Mbps.

	Privacy	Accountability	**1.5Mbps ADSL Lite**	**10Mbps Ethernet**	**100Mbps Fast Ethernet**	**1 Gbps Gigabit Ethernet**	**10Gbps 10 Gigabit Ethernet**
PAG	✓	✓	144p (660 Kbps)	480p (6.9 Mbps)	1080p (31 Mbps)	1080p (31 Mbps)	1080p (31 Mbps)
$AcTinG$	×	✓	480p (1.4 Mbps)	1080p (6 Mbps)	1080p (6 Mbps)	1080p (6 Mbps)	1080p (6 Mbps)
OR + PR	×	✓	∅	144p (2.8 Mbps)	720p (55.1Mbps)	1080p (103Mbps)	1080p (103Mbps)
RAC	✓	×	∅	∅	∅	∅	∅

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

5.5.5 Impact of updates size

Although we considered 938B updates in the previous experiments, to be fair with the other solutions, Figure 5.11 shows that using bigger updates can further decrease the bandwidth consumption of our protocol. This is due to the fact that more content can be represented under each encryption. For example, nodes propagating 10Kb updates needed to perform 370 homomorphic encryptions per second, while propagating 100Kb updates decreased this number to 52 encryptions per second.

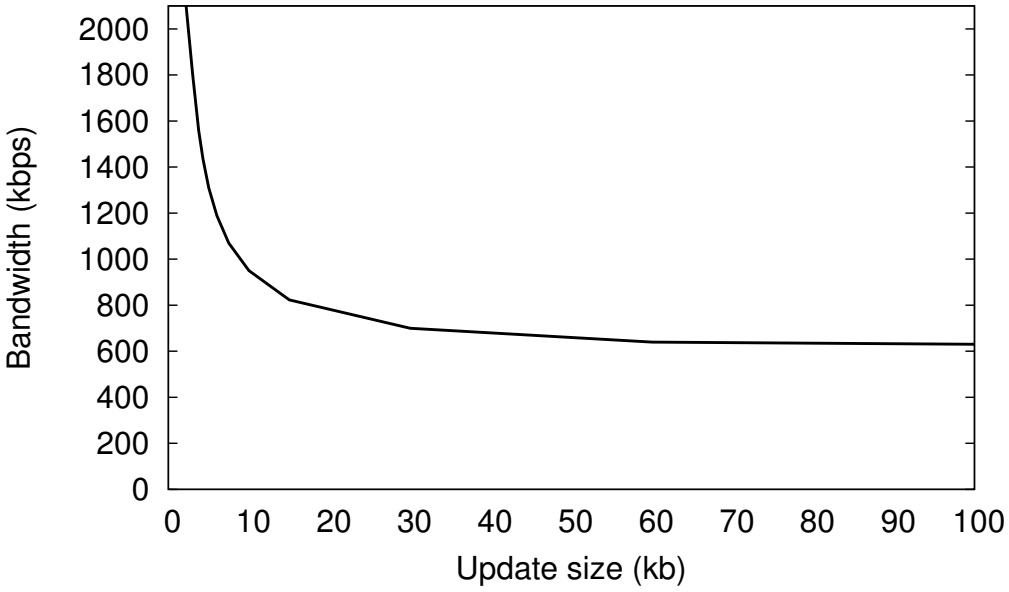


Figure 5.11 – Bandwidth consumption with 1000 nodes and a 300Kbps stream in function of the size of updates [sim]

5.5.6 Cryptographic costs

Our protocol relies on cryptographic mechanisms, which dominate the CPU cost of our protocol. To evaluate this cost, we measured the number of RSA encryptions and the number of homomorphic encryptions per second that each node performs rather than the CPU load, which depends on the hardware used. We measured the number of RSA signatures and homomorphic encryptions per second and per node (generated both from the monitoring and the participation in a session) depending on the video quality. The results are depicted in Table II. The number of RSA signatures is constant and equals 33, as it depends on the number of messages generated by the protocol, while the number of homomorphic encryptions performed depends on the video quality, and more precisely on the number of 938B updates in which the video is divided. Using a simple benchmarking tool[3], we determined that each core of the machines we used in our deployment is able to perform 900 RSA-1024 and 4800 RSA-512 encryptions per second. Thus using a single core for homomorphic encryptions is enough to obtain a video quality up to 720p using a 512 bits modulus, which would generate 3924

[3]We used the command *openssl speed rsa*.

encryptions per second. If the system size or the video quality desired were bigger it would be necessary to use more cores to have enough cryptographic power. In addition, using a 256 bits modulus can also be considered secure enough in many situations, and it would significantly reduce the cryptographic and bandwidth overhead of the protocol. As modern machines use several cores, we believe that our protocol can be used by a wide range of users.

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
Homomorphic encryptions	133	475	1170	1560	3934	7200

Table II – Number of RSA-1024 signatures and homomorphic encryptions per second in a system of 1000 nodes [sim]

5.5.7 Scalability

In this experiment we increase the number of nodes in the system and measure the associated bandwidth consumption. Typically, in a gossip-based system if N is the number of nodes that are interested in a similar content, then each node has $log(N)$ successors, and the same number of monitors. The bandwidth scalability of PAG comes from its gossip nature.

Adapting the number of monitors/successors per node allows the protocol to scale. It is possible to limit the number of monitors per node, which would decrease the bandwidth overhead. Making this choice depends on the security guarantees that the system designer aims, as increasing the number of monitors would increase the privacy of a node.

Figure 5.12 presents the bandwidth consumption of $AcTinG$ and PAG depending on the system size when a 300Kbps video stream is disseminated. We do not represent anonymous communication protocols as RAC is not able to scale correctly (it cannot provide more than 63Kbps when there are more than a thousand nodes), and OR+PR would consume a fixed amount of bandwidth independently of the system size (28Mbps for a 300Kbps video). In these conditions, PAG is able to provide nodes with the full 300Kbps stream, while consuming less bandwidth than anonymous communication systems. With a million nodes it consumes 2.5Mbps, where $AcTinG$ needs 840Kbps.

SECTION 5.6
Conclusion

A number of gossip-based content dissemination protocols tolerating selfish behaviors have been proposed in the past. A limitation of these protocols is that they do not

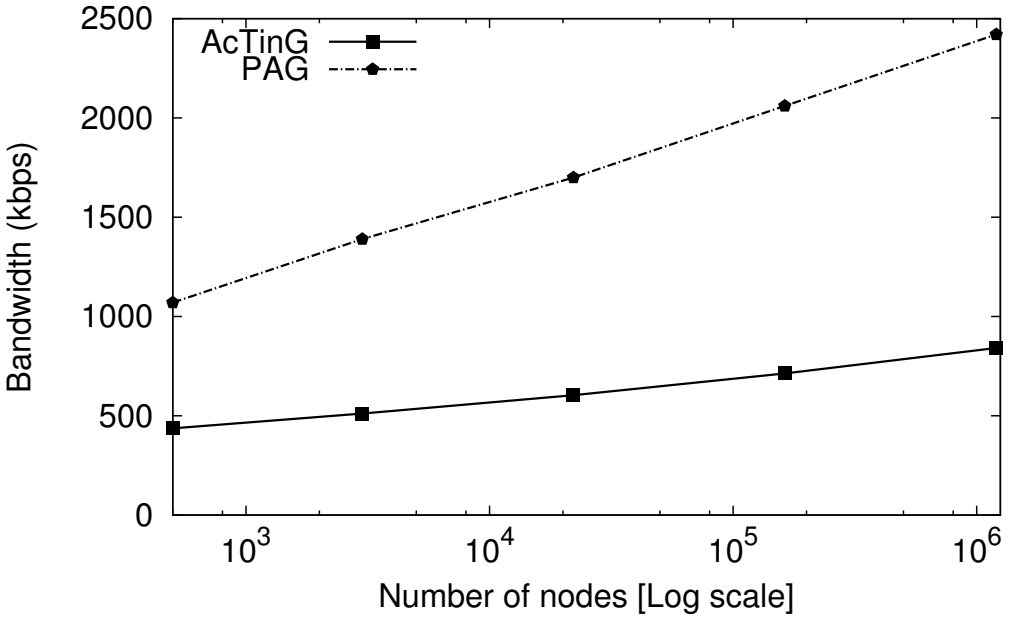


Figure 5.12 – Scalability of PAG compared to $AcTinG$ with a 300kbps content [sim]

preserve the privacy of users. On the other side of the spectrum, accountable anonymous communication protocols are too costly to be used in practice to disseminate multimedia content. In this chapter, we have presented PAG, the first content dissemination protocol that uses accountability through a monitoring infrastructure and still preserves the privacy of users thanks to homomorphic cryptographic procedures. Performance evaluation combining both a real deployment and simulations has demonstrated that it has good bandwidth properties and that the 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. Future work includes designing a privacy-preserving mechanism that would decrease the bandwidth overhead of PAG, reduce the participation of nodes in contents in which they are not interested, and an accountable eviction procedure that that would avoid correct nodes to exhibit messages they sent or received.

Conclusion

Peer-to-peer (P2P) content dissemination systems aim at distributing a given content (e.g., a video) to a set of interested users. In these systems users may be tempted to behave selfishly. Indeed, these systems need users that receive a content to share it with others. In practice, selfish users may try to save their resource (e.g., upload bandwidth), and may deviate from the protocol. Such deviations endanger the good dissemination of the content being distributed. These selfish users aim at increasing their benefit (e.g., they receive the content earlier) while decreasing their participation to the system (e.g., they contribute less bandwidth to distributing the content). Among the existing P2P paradigms that can be applied to the dissemination of a content, we focused on gossip, which is currently the most robust approach. Gossip-based protocols do not rely on an infrastructure, and consists in organizing random exchanges between nodes. This randomness helps gossip to tolerate the joins or departures of nodes during the dissemination of a content. In addition, this paradigm enforces probabilistic guarantees concerning the efficient dissemination of updates inside a set of nodes. One drawback of gossip however, is that the size of the system (i.e., the number of users) has to be known by each user in order to calibrate correctly the exchanges of nodes. This thesis focused on designing novel mechanisms to prevent selfish behaviors in gossip under different assumptions.

While the existing gossip protocols correctly prevent individual selfish deviations, we showed that selfish nodes may collude to lure the mechanisms that aim at detecting them, or to further increase their benefit. We measured the impact of such collusions in state-of-the-art protocols, and proved that selfish nodes applying an adequate strategy are able to increase their benefit and reduce their participation to the dissemination of a content, while not being detected. We then designed a new gossip protocol, named $AcTinG$, specially designed to prevent both individual and collective selfish behaviors. We showed that $AcTinG$ is able to serve the entire content being disseminated even in presence of selfish nodes, deviating either individually or collectively. A theoretical study shows that nodes colluding in $AcTinG$ do not improve their benefit, and do not perturb the dissemination of contents. Relying on secure logs, $AcTinG$ also guarantees zero false positive detections, and eventually detects selfish nodes. Performance evaluation combining both a real deployment and simulations has demonstrated 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.

However, nodes running $AcTinG$ have to be ready to share their secure logs, which detail all their exchanges and the content they received or sent. In practice, depending on the context, users may not be ready to pay such a price, and may avoid to use an application that may allow other users to collect information about them. This observation was the basis of the reflexion that led to PAG, which is the second contribution of this thesis. We explained that current methods that aim at preventing selfish behaviors leak information about users of content dissemination systems. We were able to quantify these leaks in $AcTinG$, and detail them in the existing protocols. PAG is a gossip protocol based on novel cryptographic procedures that forces nodes to forward the content they receive (i.e., forbid selfish behaviors) while preserving the privacy of their interactions, even in presence of a global and active opponent.

In the second part of this document, we have presented PAG, the first content dissemination protocol that uses accountability through a monitoring infrastructure and still preserves the privacy of users thanks to homomorphic cryptographic procedures. Performance evaluation combining both a real deployment and simulations has demonstrated that it possesses good bandwidth properties and that the 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. However, PAG contrary to $AcTinG$ does not tolerate collusions of selfish nodes, which is currently the price of enforcing privacy.

In the following, we describe some possible improvements of PAG and $AcTinG$ aiming at obtaining the best from both protocols, and future research directions that, we believe, could be interesting to follow.

Possible improvements

Efficient dissemination and privacy

We presented PAG, which is a novel gossip protocol that allows nodes to preserve their privacy using homomorphic cryptographic procedures, and is able to detect if a node behaves selfishly. The existing gossip protocols allow nodes to collect information about each others, which may dissuade users to run them. In particular, users that are interested by the same content may be able to obtain the identities of each other. A solution we proposed in PAG is to merge several sessions, which would provide a node with the content it expects but also with some contents it does not desire. Doing so, it is not possible to identify which particular content interests a user. We believe that it is possible to maintain the privacy properties of PAG while avoiding to pay this cost. An approach we are studying at the time of writing is based on probabilities to preferentially serve nodes with their content of interest, and less with the other contents. However, work is needed to be fully convinced of its effectiveness. Namely, it

is necessary to prove that all nodes correctly receive the content they desire, and do not receive all the other contents, but only a portion of them. We aim at using simulations first to be convinced of this approach, which will require little effort thanks to the codes of $AcTinG$ and PAG. Then, we plan to theoretically evaluate the guarantees of this approach concerning privacy and performance.

Collusions in PAG

We presented $AcTinG$, which is a gossip protocol that is resistant to collusions of selfish nodes. Selfish nodes may collude in order to increase their benefit, or to protect themselves from detection mechanisms. Differently, nodes in PAG are assumed to be honest-but-curious and not to try to disrupt the system, or to collude. In PAG, we found that collective attacks may harm the dissemination of updates. For example, in PAG, when a node A forwards updates to a node B, it has to send to its partner an attestation that consists in a declaration of the updates that node A forwarded and that node B will have to forward. In this attestation, it would be possible for node A to declare less updates than what is really forwarded. Such a deviation would allow node B to participate less in the dissemination of updates, thus to increase its benefit. As successors are randomly affected to nodes, our protocol does not allow nodes to choose their successors, which limit the probability of this deviation to happen. However, currently it cannot be completely avoided. Combining a resilience to collective deviations and a protection of users' privacy would be an interesting result in the sense that it could be applied to other situations.

Future research directions

Privacy-preserving proofs of misbehavior.

When a node is detected guilty proving its deviation to other nodes may reveal information about the interactions it had with others. Such procedure is then a leak of privacy. Often, publications refer to zero-knowledge proofs (ZKPs) to protect the privacy of interactions while proving that the node is faulty. We are not aware of a practical method that could be used in gossip, as ZKPs are known to be costly in term of bandwidth consumption and cryptographic costs. We plan to study more in depth these mechanisms, and see if it is possible to adapt one of them to gossip.

References

[1] B. Bollobás, *Random graphs*. Springer, 1998.

[2] N. Nisan, T. Roughgarden, E. Tardos, and V. V. Vazirani, *Algorithmic game theory*. Cambridge University Press Cambridge, 2007, vol. 1.

[3] K. B. Athreya and P. E. Ney, *Branching processes*. Springer Science & Business Media, 2012, vol. 196.

[4] H. C. Van Tilborg and S. Jajodia, *Encyclopedia of cryptography and security*. Springer Science & Business Media, 2011.

[5] S. B. Mokhtar, J. Decouchant, and V. Quéma, "Acting: Accurate freerider tracking in gossip," in *Proc. of the 33rd IEEE International Symposium on Reliable Distributed Systems, SRDS 2014, Nara, Japan, October 6-9, 2014*, 2014, pp. 291–300.

[6] H. Yin, X. Liu, T. Zhan, V. Sekar, F. Qiu, C. Lin, H. Zhang, and B. Li, "Design and deployment of a hybrid cdn-p2p system for live video streaming: experiences with livesky," in *Proceedings of the 17th ACM international conference on Multimedia*. ACM, 2009, pp. 25–34.

[7] M. Zhao, P. Aditya, A. Chen, Y. Lin, A. Haeberlen, P. Druschel, B. Maggs, B. Wishon, and M. Ponec, "Peer-assisted content distribution in Akamai NetSession," in *Proceedings of the 13th ACM SIGCOMM Conference on Internet Measurement (IMC'13)*.

[8] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz, "Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination," in *Proc. of the 11th international workshop on Network and operating systems support for digital audio and video*, 2001, pp. 11–20.

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

[10] S. Banerjee, B. Bhattacharjee, and C. Kommareddy, "Scalable application layer multicast," vol. 32, no. 4. Proc. of the ACM SIGCOMM, 2002.

[11] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek *et al.*, "Overcast: reliable multicasting with on overlay network," in *Proc. of the 4th Symposium on Operating System Design & Implementation*. USENIX Association, 2000.

[12] D. Kostić, A. Rodriguez, J. Albrecht, and A. Vahdat, "Bullet: High bandwidth data dissemination using an overlay mesh," in *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5. ACM, 2003, pp. 282–297.

[13] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth, "Bar fault tolerance for cooperative services," in *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5. ACM, 2005, pp. 45–58.

[14] R. Krishnan, M. D. Smith, Z. Tang, and R. Telang, "The impact of free-riding on peer-to-peer networks," in *Proc. of the 37th Hawaii International Conference on System Sciences*. IEEE, 2004, pp. 10–pp.

[15] M. G. Luby, M. Mitzenmacher, M. A. Shokrollahi, D. A. Spielman, and V. Stemann, "Practical loss-resilient codes," in *Proc. of the 29th ACM Symposium on Theory of Computing*, 1997, pp. 150–159.

[16] M. Luby, "Lt codes," in *Proc. of the 43rd IEEE Symposium on Foundations of Computer Science*. IEEE, 2002, pp. 271–282.

[17] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege, "A digital fountain approach to reliable distribution of bulk data," in *ACM SIGCOMM Computer Communication Review*, vol. 28, no. 4, 1998, pp. 56–67.

[18] S. B. Mokhtar, A. Pace, and V. Quema, "Firespam: Spam resilient gossiping in the bar model," in *Proc. of the 29th IEEE Symposium on Reliable Distributed Systems*. IEEE, 2010, pp. 225–234.

[19] Q. Lian, Z. Zhang, M. Yang, B. Y. Zhao, Y. Dai, and X. Li, "An empirical study of collusion behavior in the maze p2p file-sharing system," in *Proc. of the 27th International Conference on Distributed Computing Systems*. IEEE, 2007, pp. 56–56.

[20] E. L. Wong and L. Alvisi, "What's a little collusion between friends?" in *Proc. of the 2013 ACM symposium on Principles of distributed computing*. ACM, 2013, pp. 240–249.

[21] R. Eidenbenz, T. Locher, and R. Wattenhofer, "Hidden communication in p2p networks steganographic handshake and broadcast," in *Proc. of the INFOCOM IEEE*. IEEE, 2011, pp. 954–962.

[22] H. C. Li, A. Clement, E. L. Wong, J. Napper, I. Roy, L. Alvisi, and M. Dahlin, "Bar gossip," in *Proc. of the 7th symposium on Operating Systems Design and Implementation*. USENIX Association, 2006, pp. 191–204.

[23] H. C. Li, A. Clement, M. Marchetti, M. Kapritsos, L. Robison, L. Alvisi, and M. Dahlin, "Flightpath: Obedience vs. choice in cooperative services." in *Proc. of the 8th symposium on Operating Systems Design and Implementation*, vol. 8, 2008, pp. 355–368.

[24] M. Haridasan, I. Jansch-Porto, K. Birman, and R. van Renesse, "Enforcing fairness in a live-streaming system," in *Proc. of 15th symposium on Multimedia Computing and Networking (MMCN'08)*, January 2008.

[25] H. Johansen, A. Allavena, and R. Van Renesse, "Fireflies: scalable support for intrusion-tolerant network overlays," in *ACM SIGOPS Operating Systems Review*, vol. 40, no. 4. ACM, 2006, pp. 3–13.

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

[27] M. Jelasity, R. Guerraoui, A.-M. Kermarrec, and M. Van Steen, "The peer sampling service: Experimental evaluation of unstructured gossip-based implementations," in *Proc. of the 5th ACM/IFIP/USENIX international conference on Middleware*. Springer-Verlag New York, Inc., 2004, pp. 79–98.

[28] A.-M. Kermarrec, A. Pace, V. Quema, and V. Schiavoni, "Nat-resilient gossip peer sampling," in *Proc. of the 29th IEEE International Conference on Distributed Computing Systems, ICDCS'09*. IEEE, 2009, pp. 360–367.

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

[30] M. Backes, P. Druschel, A. Haeberlen, and D. Unruh, "Csar: A practical and provable technique to make randomized systems accountable." in *Proc. of the 16th symposium on Network and Distributed System Security*, vol. 9, 2009, pp. 341–353.

[31] A. Haeberlen, P. Aditya, R. Rodrigues, and P. Druschel, "Accountable virtual machines." in *Proc. of the 9th symposium on Operating Systems Design and Implementation*, 2010, pp. 119–134.

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

[33] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda, "Trinc: Small trusted hardware for large distributed systems." in *Proc. of the 6th Symposium on Networked Systems Design and Implementation*, vol. 9, 2009, pp. 1–14.

[34] C. Ho, R. Van Renesse, M. Bickford, and D. Dolev, "Nysiad: Practical protocol transformation to tolerate byzantine failures." in *Proc. of the 5th Symposium on Networked Systems Design and Implementation*, vol. 8, 2008, pp. 175–188.

[35] A. Diarra, S. B. Mokhtar, P. Aublin, and V. Quéma, "Fullreview: Practical accountability in presence of selfish nodes," in *Proc. of the 33rd IEEE Symposium on Reliable Distributed Systems*, 2014.

[36] E. Zheleva and L. Getoor, "To join or not to join: the illusion of privacy in social networks with mixed public and private user profiles," in *Proc. of the 18th international conference on World wide web*, 2009.

[37] H. Hu, G.-J. Ahn, and J. Jorgensen, "Detecting and resolving privacy conflicts for collaborative data sharing in online social networks," in *Proc. of the 27th Computer Security Applications Conference*, 2011.

[38] M. Belenkiy, M. Chase, C. C. Erway, J. Jannotti, A. Küpçü, A. Lysyanskaya, and E. Rachlin, "Making p2p accountable without losing privacy," in *Proc. of the 2007 ACM workshop on Privacy in electronic society*. ACM, 2007, pp. 31–40.

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

[40] S. Ben Mokhtar, G. Berthou, A. Diarra, V. Quéma, and A. Shoker, "Rac: a freerider-resilient, scalable, anonymous communication protocol," in *Proc. of the 33rd International Conference on Distributed Computing Systems*. IEEE, 2013, pp. 520–529.

[41] D. I. Wolinsky, H. Corrigan-Gibbs, B. Ford, and A. Johnson, "Dissent in numbers: Making strong anonymity scale." in *Proc. of the 10th Symposium on Operating Systems Design and Implementation*, 2012, pp. 179–182.

[42] P. Golle and A. Juels, "Dining cryptographers revisited," in *Proc. of the Advances in Cryptology-Eurocrypt 2004*. Springer, 2004, pp. 456–473.

[43] L. Zhuang, F. Zhou, B. Y. Zhao, and A. Rowstron, "Cashmere: Resilient anonymous routing," in *Proc. of the 2nd conference on Symposium on Networked Systems Design and Implementation*. USENIX Association, 2005, pp. 301–314.

[44] M. J. Freedman and R. Morris, "Tarzan: A peer-to-peer anonymizing network layer," in *Proc. of the 9th ACM conference on Computer and Communications Security*. ACM, 2002, pp. 193–206.

[45] O. Goldreich, S. Micali, and A. Wigderson, "How to play any mental game," in *Proc. of the 19th ACM symposium on Theory of Computing*. ACM, 1987, pp. 218–229.

[46] A. Papadimitriou, M. Zhao, and A. Haeberlen, "Towards privacy-preserving fault detection," in *Proc. of the 9th Workshop on Hot Topics in Dependable Systems*. ACM, 2013, p. 6.

[47] A. Narayan, A. Feldman, A. Papadimitriou, and A. Haeberlen, "Verifiable differential privacy," in *Proc. of the EuroSys conference*, 2015.

[48] M. Zhao, W. Zhou, A. J. Gurney, A. Haeberlen, M. Sherr, and B. T. Loo, "Private and verifiable interdomain routing decisions," in *Proc. of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. ACM, 2012, pp. 383–394.

[49] M. Bertier, D. Frey, R. Guerraoui, A.-M. Kermarrec, and V. Leroy, "The gossple anonymous social network," in *Proc. of the ACM/IFIP/USENIX 11th International Conference on Middleware*. Springer-Verlag, 2010, pp. 191–211.

[50] A. Boutet, D. Frey, R. Guerraoui, A. Jégou, and A.-M. Kermarrec, "Privacy-preserving distributed collaborative filtering," in *Proc. of the International Conference on Networked Systems*. Springer, 2014, pp. 169–184.

[51] M. Jovic, A. Adamoli, and M. Hauswirth, "Catch me if you can: performance bug detection in the wild," in *ACM SIGPLAN Notices*, vol. 46, no. 10. ACM, 2011, pp. 155–170.

[52] B. Blanchet, "An efficient cryptographic protocol verifier based on prolog rules," in *Proc. 14th IEEE Computer Security Foundations Workshop (CSFW)*. IEEE, 2001, pp. 82–96.

[53] M. Castro, P. Druschel, A.-M. Kermarrec, and A. I. Rowstron, "Scribe: A large-scale and decentralized application-level multicast infrastructure," *IEEE Journal on Selected Areas in Communications*, vol. 20, no. 8, pp. 1489–1499, 2002.

[54] V. Pai, K. Kumar, K. Tamilmani, V. Sambamurthy, and A. E. Mohr, "Chainsaw: Eliminating trees from overlay multicast," in *Peer-to-peer systems IV*. Springer, 2005, pp. 127–140.

[55] P. T. Eugster, R. Guerraoui, A.-M. Kermarrec, and L. Massoulié, "Epidemic information dissemination in distributed systems," *Computer*, vol. 37, no. 5, pp. 60–67, 2004.

[56] A.-M. Kermarrec, L. Massoulié, and A. J. Ganesh, "Probabilistic reliable dissemination in large-scale systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 3, pp. 248–258, 2003.

[57] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, no. 3, pp. 382–401, 1982.

[58] D. Hughes, G. Coulson, and J. Walkerdine, "Free riding on gnutella revisited: the bell tolls?" *Distributed Systems Online, IEEE*, vol. 6, no. 6, 2005.

[59] V. K. Goyal, "Multiple description coding: Compression meets the network," *Signal Processing Magazine, IEEE*, vol. 18, no. 5, pp. 74–93, 2001.

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

[61] E. Adar and B. A. Huberman, "Free riding on gnutella," *First Monday*, vol. 5, no. 10, 2000.

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

[63] H. D. Johansen, R. V. Renesse, Y. Vigfusson, and D. Johansen, "Fireflies: A secure and scalable membership and gossip service," *ACM Trans. Comput. Syst.*, vol. 33, no. 2, pp. 5:1–5:32, May 2015. [Online]. Available: http://doi.acm.org/10.1145/2701418

[64] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. Van Steen, "Gossip-based peer sampling," *ACM Transactions on Computer Systems (TOCS)*, 2007.

[65] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990.

[66] T. C. Bressoud and F. B. Schneider, "Hypervisor-based fault tolerance," *ACM Transactions on Computer Systems (TOCS)*, vol. 14, no. 1, pp. 80–107, 1996.

[67] A. J. Ganesh, A.-M. Kermarrec, and L. Massoulié, "Scamp: Peer-to-peer lightweight membership service for large-scale group communication," in *Networked Group Communication*, 2001.

[68] A. Mei and J. Stefa, "Give2get: Forwarding in social mobile wireless networks of selfish individuals," *Dependable and Secure Computing, IEEE Transactions on*, vol. 9, no. 4, pp. 569–582, 2012.

[69] A. Pfitzmann and M. Hansen, "Anonymity, unlinkability, undetectability, unobservability, pseudonymity, and identity management-a consolidated proposal for terminology," *Version v0*, vol. 31, p. 15, 2008.

[70] D. Chaum, "The dining cryptographers problem: Unconditional sender and recipient untraceability," *Journal of cryptology*, vol. 1, no. 1, pp. 65–75, 1988.

[71] D. Goldschlag, M. Reed, and P. Syverson, "Onion routing," *Communications of the ACM*, vol. 42, no. 2, pp. 39–41, 1999.

[72] M. Waidner, B. Pfitzmann *et al.*, "The dining cryptographers in the disco: Unconditional sender and recipient untraceability with computationally secure serviceability," *J.-J. Quisquater and J. Vandewalle, editors, Advances in Cryptology—EUROCRYPT*, vol. 89, p. 690, 1989.

[73] S. Goel, M. Robson, M. Polte, and E. Sirer, "Herbivore: A scalable and efficient protocol for anonymous communication," Cornell University, Technical Report, 2003.

[74] M. Edman and B. Yener, "On anonymity in an electronic society: A survey of anonymous communication systems," *ACM Computing Surveys (CSUR)*, vol. 42, no. 1, p. 5, 2009.

[75] M. K. Reiter and A. D. Rubin, "Crowds: Anonymity for web transactions," *ACM Transactions on Information and System Security (TISSEC)*, vol. 1, no. 1, pp. 66–92, 1998.

[76] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router," DTIC Document, Tech. Rep., 2004.

[77] O. Goldreich, S. Micali, and A. Wigderson, "Proofs that yield nothing but their validity or all languages in np have zero-knowledge proof systems," *Journal of the ACM (JACM)*, vol. 38, no. 3, pp. 690–728, 1991.

[78] L. Sweeney, "k-anonymity: A model for protecting privacy," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 2002.

[79] "Skype," http://www.skype.com, accessed: 2015-08-31.

[80] "Jabber," http://www.jabber.org, accessed: 2015-08-31.

[81] "Icq," http://www.icq.com/en, accessed: 2015-08-31.

[82] "Sopcast," http://www.sopcast.org, accessed: 2015-08-31.

[83] "Veetle," http://www.veetle.com, accessed: 2015-08-31.

[84] "Diaspora," https://diasporafoundation.org, accessed: 2015-08-31.

[85] F. Khatibloo, "Personal identity management," 2011.

[86] O. D. E. Simulator, https://omnetpp.org/, accessed: 2015-08-31.

Appendix

In this appendix, we present the model of PAG that has been used with ProVerif to prove that it preserves the privacy of users in presence of individual selfish nodes, and is resilient to limited-size coalitions. Indeed, if nodes have f successors then an attacker needs to control at least f very precise nodes to discover the content of an exchange between two nodes. We then present the code that has been used to evaluate the theoretical resiliencies of PAG and $AcTinG$ to coalitions. Using probabilities, we were able to show that PAG ensures a close to ideal security to nodes.

SECTION .1

ProVerif code of PAG

```
(*
Phases that include the reception of updates from A1, A2, A3
   by B,
and the forwarding to C

   Send prime numbers X
Message 1: B -> A1: { { p1 }skB }pkA1
Message 2: B -> A2: { { p2 }skB }pkA2
Message 3: B -> A3: { { p3 }skB }pkA3

   Send updates X
Message 4: A1 -> B: { { u1, NA1 }skA1 }pkB
Message 5: A2 -> B: { { u2, NA2 }skA2 }pkB
Message 6: A3 -> B: { { u3, NA3 }skA3 }pkB

   Attestations X
Message 7: A1 -> B: { { u1 }p1 }skA1
Message 8: A2 -> B: { { u2 }p2 }skA2
Message 9: A3 -> B: { { u3 }p3 }skA3

   Forward to C, and obtain an ack X
Message 10: C -> B: { { p4 }skC }pkB
Message 11: B -> C: { { u1, u2, u3, p1.p2.p3 }skB }pkC
Message 12: C -> B: { { u1.u2.u3}p1.p2.p3 }skC

   Acks from B to A1, A2, A3 X
Message 13: B -> A1: { {u1}NA1 }skB
Message 14: B -> A2: { {u2}NA2 }skB
```

```
Message 15: B -> A3: { {u3}NA3 }skB

   Acks from B to its witness
Message 16: B -> WB: { {u1}NA1 }skB
Message 17: B -> WB: { {u2}NA2 }skB
Message 18: B -> WB: { {u3}NA3 }skB

   Attestation from B to its witness
Message 19: B -> WB: { { { u1 }p1 }skA1, p2.p3 }skB
Message 20: B -> WB: { { { u2 }p2 }skA2, p1.p3 }skB
Message 21: B -> WB: { { { u3 }p3 }skA3, p1.p2 }skB

   Transfer of ack from WB to WA1, WA2, WA3
Message 22: WB -> WA1: { {u1}NA1 }skB
Message 23: WB -> WA2: { {u2}NA2 }skB
Message 24: WB -> WA3: { {u3}NA3 }skB

   Ack from C to its witness X
Message 25: C -> WC: { {u1.u2.u3}p1.p2.p3 }skC

   Transfer of ack from WC to WB
Message 26: WC -> WB: { {u1.u2.u3}p1.p2.p3 }skC
*)

free c: channel.

type host.
(*type nonce.*)
type pkey.
type skey.
type int.
type msg.

(* Public key encryption *)

fun pk(skey): pkey.
fun encrypt(bitstring, pkey): bitstring.
reduc forall x: bitstring, y: skey;
   decrypt(encrypt(x,pk(y)),y) = x.

fun encryptPrime(bitstring, int): bitstring.
fun product(int, int): int.
(* reduc forall x: int, y: int; divide(product(x,y),y) = x.
*)

fun productUp(bitstring, bitstring): bitstring.
(* reduc forall x: bitstring, y: bitstring;
   divideUp(productUp(x,y),y) = x.
*)

fun int_to_bitstring(int): bitstring [data,typeConverter].
fun bitstring_to_int(bitstring): int[data,typeConverter].
```

```
(* Signatures *)

fun spk(skey): pkey.
fun sign(bitstring, skey): bitstring.
reduc forall x: bitstring, y: skey; verif(sign(x,y), pk(y),
   x) = true.
reduc forall x: bitstring, y: skey; getmess(sign(x,y)) = x.

(* Secrecy assumptions *)

not attacker(new skA1).
not attacker(new skA2).
not attacker(new skA3).
not attacker(new skB).
not attacker(new skC).

(* host names *)

free A1, A2, A3, B, C, WA1, WA2, WA3, WB, WC: host.
free attestation: msg.

(* Queries *)

free u1, u2, u3, NA1, NA2, NA3: bitstring [private].
query attacker(u1).

(* Role of A1, A2, A3 *)

let processInitiatorA(hostA: host, skA: skey, pkB: pkey, u:
   bitstring, NA: bitstring) =
   in(c, (=hostA, m : bitstring));

   let m1 = decrypt(m, skA) in
   let p = getmess(m1) in
   if verif(m1, pkB, p) = true then
      out(c, (B, encrypt(sign((u, NA), skA), pkB))); (* serve
         *)
      out(c, (B, sign((attestation, encryptPrime(u,
         bitstring_to_int(p))), skA))); (* attestation *)
      in(c, (=hostA, m2 : bitstring))
   else
      0.

(* Role of B *)

let processInitiatorB(skB: skey, pkA1: pkey, pkA2: pkey,
   pkA3: pkey, pkC: pkey) =

   new p1: int;
   new p2: int;
   new p3: int;
```

```
out(c, (A1, encrypt( sign(int_to_bitstring(p1), skB),
    pkA1) ));
out(c, (A2, encrypt( sign(int_to_bitstring(p2), skB),
    pkA2) ));
out(c, (A3, encrypt( sign(int_to_bitstring(p3), skB),
    pkA3) ));

(* serve messages *)
in(c, (=B, m4: bitstring)); (* what if the messages arrive
    in a different order? or are replayed? *)
in(c, (=B, m5: bitstring));
in(c, (=B, m6: bitstring));

let n4 = decrypt(m4, skB) in
let (u10: bitstring, NA10: int) = getmess(n4) in
let n5 = decrypt(m5, skB) in
let (u20: bitstring, NA20: int) = getmess(n5) in
let n6 = decrypt(m6, skB) in
let (u30: bitstring, NA30: int) = getmess(n6) in
if verif(n4, pkA1, (u10, NA10)) = true && verif(n5, pkA2,
    (u20, NA20)) = true && verif(n6, pkA3, (u30, NA30)) =
    true then

   in(c, (=B, m7: bitstring));
   in(c, (=B, m8: bitstring));
   in(c, (=B, m9: bitstring));

   let (=attestation, n7: bitstring) = getmess(m7) in
   let (=attestation, n8: bitstring) = getmess(m8) in
   let (=attestation, n9: bitstring) = getmess(m9) in
   if verif(m7, pkA1, n7) = true && verif(m8, pkA2, n8) =
       true && verif(m9, pkA3, n9) = true then
      out(c, (A1, sign(encryptPrime(u10, NA10) , skB)));
      out(c, (A2, sign(encryptPrime(u20, NA20) , skB)));
      out(c, (A3, sign(encryptPrime(u30, NA30) , skB)));

      out(c, (WB, sign(encryptPrime(u10, NA10) , skB)));
      out(c, (WB, sign(encryptPrime(u20, NA20) , skB)));
      out(c, (WB, sign(encryptPrime(u30, NA30) , skB)));

      out(c, (WB, sign((m7, product(p2,p3)), skB)));
      out(c, (WB, sign((m8, product(p1,p3)), skB)));
      out(c, (WB, sign((m9, product(p1,p2)), skB)));

      in(c, (=B, m10: bitstring));
      let n10 = decrypt(m10, skB) in
      let p4 = getmess(n10) in
      if verif(n10, pkC, p4) = true then
         out(c, (C, encrypt(sign( (u1, u2, u3,
            product(product(p1, p2), p3)), skB) ,pkC)));
         in(c, (=B, m12: bitstring))
      else
         0
```

```
      else
         0
   else
      0.


let processInitiatorC(skC: skey, pkB: pkey) =
   new p4: int;
   out(c, (B, encrypt(sign(int_to_bitstring(p4), skC), pkB)));

   in(c, (=C, m11: bitstring));
   let n11 = decrypt(m11, skC) in
   let (u10: bitstring, u20: bitstring, u30: bitstring, p:
      int) = getmess(n11) in
   if verif(n11, pkB, (u10, u20, u30, p)) = true then
      out(c, (B,
         sign(encryptPrime(productUp(u10,productUp(u20,
         u30)), p), skC)));
      out(c, (WC,
         sign(encryptPrime(productUp(u10,productUp(u20,
         u30)), p), skC)))
   else
      0.

let processInitiatorWB(pkB: pkey, pkC: pkey) =
   in(c, (=WB, m16: bitstring));
   in(c, (=WB, m17: bitstring));
   in(c, (=WB, m18: bitstring));
   out(c, (WA1, m16));
   out(c, (WA2, m17));
   out(c, (WA3, m18));

   in(c, (=WB, m19: bitstring));
   in(c, (=WB, m20: bitstring));
   in(c, (=WB, m21: bitstring));
   let (n19: bitstring, p23: bitstring) = getmess(m19) in
   let (n20: bitstring, p13: bitstring) = getmess(m20) in
   let (n21: bitstring, p12: bitstring) = getmess(m21) in
   if verif(m19, pkB, (n19, p23)) = true && verif(m20, pkB,
      (n20, p13)) = true && verif(m21, pkB, (n21, p12)) =
      true then
      in(c, (=WB, m26: bitstring));
      let n26 = getmess(m26) in
      if verif(m26, pkC, n26) = true then
      (* TODO: Verification homomorphique *)
         0
      else
         0
   else
      0.

let processInitiatorWC() =
```

```
   in(c, (=WC, m25: bitstring));
   out(c, (WB, m25)).

let processInitiatorWA(hostWA: host, pkB: pkey) =
   in(c, (=hostWA, m22: bitstring));
   let n22 = getmess(m22) in
   if verif(m22, pkB, n22) = true then
      0
   else
      0.

(* Start process *)

process
   new skA1: skey;
   let pkA1 = pk(skA1) in
   out(c, pkA1);
   new skA2: skey;
   let pkA2 = pk(skA2) in
   out(c, pkA2);
   new skA3: skey;
   let pkA3 = pk(skA3) in
   out(c, pkA3);
   new skB: skey;
   let pkB = pk(skB) in
   out(c, pkB);
   new skC: skey;
   let pkC = pk(skC) in
   out(c, pkC);

   (
      (* Launch an unbounded number of sessions of the
         initiator *)
      (!processInitiatorA(A1, skA1, pkB, u1, NA1)) |
      (!processInitiatorA(A2, skA2, pkB, u2, NA2)) |
      (!processInitiatorA(A3, skA3, pkB, u3, NA3)) |
      (!processInitiatorB(skB, pkA1, pkA2, pkA3, pkC)) |
      (!processInitiatorC(skC, pkB)) |
      (!processInitiatorWB(pkB, pkC)) |
      (!processInitiatorWC()) |
      (!processInitiatorWA(WA1, pkB)) |
      (!processInitiatorWA(WA2, pkB)) |
      (!processInitiatorWA(WA3, pkB))
   )
```

SECTION .2

Probabilistic evaluation of PAG resiliency to collusions

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
```

```
#define ld long double

/* Factorial utility function */
ld fact(ld x) {
   ld res = 1.0;
   for (ld i = 1.0; i <= x; i++)
      res *= i;
   return res;
}

/* Binomial coefficient utility function */
ld coefBin(ld k, ld n) {
   return fact(n) / (fact(k) * fact(n-k));
}


int main(void) {

   FILE *file = fopen("../probaPrivacy.data", "w");

   ld n = 1000.0;     // Number of nodes in the system
   ld paudit = 0.1;   // Audit probability in AcTinG
   ld RTE = 10.0;     // Delay between an update release and
      its expiration
   ld fActing = 3.0; // Fanout of nodes in AcTinG
   ld period = 5.0;   // Period with which nodes change their
      successors in AcTinG

   for (ld p = 0.0; p <= n; p++) {

      /* Probability that one of the two partners is an
         opponent and makes an audit, which would allow him
         to discover the interaction */
      ld probaComplexe = 0.0;
      ld X = (2.0 * 2.0 * fActing * RTE) / period; /* Total
         number of partners of the two nodes */
      X = 2*X + pow(X, 2);

      for (ld j = 0.0; j <= X; j++) { /* Binomial law on the
         number of opponents, with probability that at least
         one realizes an audit */
         ld tmp = coefBin(j, X) * pow(p/n, j)* pow((n-p)/n,
            X-j);
         tmp *= (1.0 - (ld) pow(1.0 - paudit, j)); /*
            Probability that none of the j nodes makes an
            audit */
         probaComplexe += tmp;
      }

      ld probaActing = (ld) (1-pow((n-p)/n, 2.0)) + (ld)
         pow((n-p)/n, 2.0) * probaComplexe;
```

```
/* Probability for PAG with 3 successors/witnesses per
    node */
ld f = 3.0;
ld pf2 = 0.0; /* f-2 of the predecessors are colluders,
    and at least one of the two critical message goes to
    a colluding attacker */
ld k = f-2;
pf2 += (coefBin(k, f) * ((ld) pow(p/n, k)) * ((ld)
    pow(1-p/n,f-k))) * (1-pow(1-p/n, 2));
k = f-1; /* or f-1 predecessors are attackers */
pf2 += (coefBin(k, f) * ((ld) pow(p/n, k)) * ((ld)
    pow(1-p/n,f-k)));
ld privacy3 = (1.0 - (ld) pow(1.0-p/n, 2.0)) + ((ld)
    pow(1.0-p/n, 2.0)) * pf2 * (1.0 - (ld) pow(1.0-p/n,
    f));

f = 5.0;
pf2 = 0.0; /* f-2 of the predecessors are colluders,
    and at least one of the two critical message goes to
    a colluding attacker */
k = f-2;
pf2 += (coefBin(k, f) * ((ld) pow(p/n, k)) * ((ld)
    pow(1-p/n,f-k))) * (1-pow(1-p/n, 2)); /* f-2
    predecessors collude and both critical messages go
    to and attacker witness */
k = f-1; /* or f-1 predecessors are attackers */
pf2 += (coefBin(k, f) * ((ld) pow(p/n, k)) * ((ld)
    pow(1-p/n,f-k)));
ld privacy5 = (1.0 - (ld) pow((n-p)/n, 2.0)) + ((ld)
    pow((n-p)/n, 2.0)) * pf2 * (1.0 - (ld) pow((n-p)/n,
    f));
/* one of the two nodes is incorrect, or all the other
    predecessors except one are incorrect and there is
    at least one incorrect witness (1-all are correct) */

f = 7.0;
pf2 = 0.0; /* f-2 of the predecessors are colluders,
    and at least one of the two critical message goes to
    a colluding attacker */
k = f-2;
pf2 += (coefBin(k, f) * ((ld) pow(p/n, k)) * ((ld)
    pow(1-p/n,f-k))) * (1-pow(1-p/n, 2));
k = f-1; /* or f-1 predecessors are attackers */
pf2 += (coefBin(k, f) * ((ld) pow(p/n, k)) * ((ld)
    pow(1-p/n,f-k)));
ld privacy10 = (1.0 - (ld) pow(1.0-p/n, 2.0)) + ((ld)
    pow(1.0-p/n, 2.0)) * pf2 * (1.0 - (ld) pow(1.0-p/n,
    f));

fprintf(file, "%Lf\t%Lf\t%Lf\t%Lf\t%Lf\t%Lf\n", (ld)
    (p/n)*100.0, (ld) (1-pow(1.0-p/n, 2.0))*100.0,
    probaActing*100.0, privacy3*100.0, privacy5*100.0,
    privacy10*100.0);
```

```
    }

    fclose(file);

    return 0;
}
```