DOCTORAL THESIS

Cotutelle-de-thèse

INSTITUT NATIONAL DES SCIENCES APPLIQUÉES DE LYON

ÉCOLE DOCTORALE **ED 512** – INFORMATIQUE ET MATHÉMATIQUES DE LYON

SPÉCIALITÉ INFORMATIQUE

DIRECTOR: PROF. LUCA Q. ZAMBONI

UNIVERSITÁ DEGLI STUDI DI MILANO

DEPARTMENT OF COMPUTER SCIENCE

CORSO DI DOTTORATO IN INFORMATICA (XXVIII CYCLE) – **INF/01**

COORDINATOR: PROF. PAOLO BOLDI

Defended on 24 March 2017, by :

Guido LENA COTA

Addressing Selfishness in the Design of Cooperative Systems

Supervisors:	Prof. Lionel BRUNIE	INSA de Lyon
	Prof. Ernesto DAMIANI	Università degli Studi di Milano
Cosupervisors:	Dr. Sonia BEN MOKHTAR	INSA de Lyon
	Dr. Gabriele GIANINI	Università degli Studi di Milano

EXAMINATION COMMITTEE:

Reviewers:	Prof. Harald KOSCH	Universität Passau, Germany
	Prof. Vivien QUEMA	Grenoble INP / ENSIMAG, France
Reviewer (Università degli Studi di Milano) , Examiner:		
	Dr. Julia LAWALL	Sorbonne Universités, UPMC, LIP6, France
Examiners:	Prof. Sara BOUCHENAK	INSA de Lyon, France
	Prof. Paola BONIZZONI	Università degli Studi di Milano-Bicocca, Italy
	Prof. Mariagrazia FUGINI	Politecnico di Milano, Italy

ABSTRACT

Cooperative distributed systems, particularly peer-to-peer systems, are the basis of several mainstream Internet applications (e.g., file-sharing, media streaming) and the key enablers of new and emerging technologies, including Blockchain and the Internet of Things. Essential to the success of cooperative systems is that nodes are willing to cooperate with each other by sharing part of their resources, e.g., network bandwidth, CPU capability, storage space. However, as nodes are autonomous entities, they may be tempted to behave in a selfish manner by not contributing their fair share, potentially causing system performance degradation and instability. Addressing selfish nodes is, therefore, key to building efficient and reliable cooperative systems. Yet, it is a challenging task, as current techniques for analysing selfishness and designing effective countermeasures remain manual and time-consuming, requiring multi-domain expertise.

In this thesis, we aim to provide practical and conceptual tools to help system designers in dealing with selfish nodes. First, based on a comprehensive survey of existing work on selfishness, we develop a classification framework to identify and understand the most important selfish behaviours to focus on when designing a cooperative system. Second, we propose RACOON, a unifying framework for the selfishness-aware design and configuration of cooperative systems. RACOON provides a semi-automatic methodology to integrate a given system with practical and finely tuned mechanisms to meet specified resilience and performance objectives, using game theory and simulations to predict the behaviour of the system when subjected to selfish nodes. An extension of the framework (RACOON++) is also proposed to improve the accuracy, flexibility, and usability of RACOON. Finally, we propose SEINE, a framework for fast modelling and evaluation of various types of selfish behaviour in a given cooperative system. SEINE relies on a domain-specific language for describing the selfishness scenario to evaluate and provides semi-automatic support for its implementation and study in a state-of-the-art simulator.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

A cooperative system is a complex distributed system that relies on the voluntary resource contribution from its participants to perform the system function. Peer-to-peer (P2P) systems are the most widespread and well-known examples of cooperative systems. Other examples are cooperative distributed computing (e.g., grid computing [62], volunteer computing [18]) and self-organizing wireless networks (e.g., delay tolerant networks [56]). The popularity of these systems has rapidly grown in recent years [44]. This trend is expected to continue, driven by the ever-increasing demand for digital content, notably multimedia content (music, movies, TV series), and fuelled by the emergence of P2P-assisted content delivery technologies [85, 147].

In cooperative systems, nodes from different administrative domains are expected to contribute to the overall service provision, e.g., downloading files [2, 45], watching live events streaming [1, 5], or making video and phone calls [23]. This collective contribution opens up the potential for scalable, self-sustained and robust distributed systems, without requiring costly dedicated servers.

Essential to the success of cooperative systems is that nodes are willing to cooperate with each other by sharing part of their resources — e.g., network bandwidth, CPU capability, storage space. However, in practice [78, 87, 114, 116, 144, 182], real systems often suffer from selfish nodes that strategically withdraw from cooperation to satisfy their individual interests. For instance, users of file-sharing applications may decide not to share any files [87, 182], or to share files only with a small group of partners [114]. Predictably, these selfish behaviours can severely undermine the systems performance and lead to widespread service degradation [111, 135, 141, 148]. For example, Guerraoui et al. [72] demonstrated experimentally that if 25% of nodes participating in a P2P live streaming system download a given video file without sharing it with other nodes, then half of the remaining nodes are not able to view a clear stream [72].

Despite the vast amount of research that has been conducted to address selfishness in cooperative systems, designing an efficient selfishness-resilient system remains a challenging, time-consuming, and error-prone task [15, 118, 160]. A system designer has to account for a plethora of design decisions (e.g., choosing the proper routing algorithm, selecting the appropriate overlay structure, deploying mechanisms to foster cooperation), each of which requires significant effort to be evaluated [64, 98, 108]. The complexity is further enhanced by the many and varied possibilities for a selfish node to deviate from the correct behaviour in a given system, as well as by the diverse — and often conflictual — application-specific objectives that the system designer seeks to achieve.

Part of the difficulty stems from the lack of a convenient unifying framework to *design*, *strengthen*, *tune* and *evaluate* real-world cooperative systems in selfish-prone environments. Existing tools and methodologies provide only a partial solution. Analytical frameworks, particularly game theory [130], provide theoretical tools to reason about selfishness and cooperation

in competitive situations like those underlying a cooperative system. On the other side, the application of analytical models to real systems can be extremely complex [118, 150]. In contrast, practical frameworks for designing and deploying robust distributed systems do not provide any guidance for taking the problem of selfish behaviours into consideration [98, 108, 128]. This issue is typically handled by integrating into the system mechanisms designed to foster cooperation and punish selfish nodes. However, the practical use of these mechanisms is not straightforward, because they are either tailored to a particular application [45, 47, 111, 135] or they are general but difficult to test and configure [53, 72, 76].

The aim of our research is to facilitate the task of system designers in dealing with selfishness in cooperative systems. To this end, we present a classification framework and a descriptive language for describing motivations and executions of selfish behaviours in cooperative distributed systems. Furthermore, we provide general methodologies, along with their software implementations, for supporting the semi-automatic design and evaluation of cooperative systems deployed over a network of selfish nodes.

1.1 SELFISHNESS IN COOPERATIVE SYSTEMS

Most cooperative systems are characterised by untrusted autonomous individuals with their own objectives — not necessarily aligned with the system objectives — and full control over the device they use to interact with the system [127]. This applies especially to P2P systems, due to the open nature of most of them [30, 116]. Autonomy and self-interest are the defining characteristics of selfish nodes.

A selfish node is a strategic individual that cooperates with other nodes only if such behaviour increases its local benefits.[1] Selfish behaviours have been observed in P2P [72, 78, 87, 114, 116, 144, 182] and in other cooperative systems [125, 135, 176]. The emergence of selfishness in cooperative systems results in substantial degradation of performance, unpredictable or limited availability of resources, and may even lead to a complete disruption of the system functionalities [28, 71, 135, 152].

The large body of literature on selfishness in cooperative systems documents not only the importance of this problem but also the number and variety of possible selfish behaviours. In the remainder of this section, we present three examples of selfish behaviours in P2P and volunteer computing, and we examine their impact on the system performance.

1.1.1 *Example: selfishness in P2P file-sharing*

By far, file-sharing has been the most popular and widely-deployed application of P2P. A P2P file-sharing application (e.g., BitTorrent [45], eDonkey [78], Gnutella [87]) enables its users to share files directly among each other via the Internet. The goal of these systems is to offer a high variety and availability of digital content, supported by the voluntary contribution of files and bandwidth by the users.

1 For the moment, we are content with this informal definition, but we formalise this concept later in Chapter 2.

File-sharing has become the most representative showcase for the problem of selfishness in cooperative systems. In 2005, Hughes et al. [87] reported that almost 85% of participants of the file-sharing system Gnutella do not share any files. The next year Handurukande et al. [78] observed a similar situation in the eDonkey network, with almost four-fifths of the clients not sharing anything. In two measurement studies between 2008 and 2010, Zghaibeh et al. [181, 182] showed that the population of selfish nodes in the BitTorrent community increased by almost 80% in two years.

These selfish behaviours are pursued through rather simple actions, such as changing a configuration parameter (e.g., the upload bandwidth) or exiting the client application once the files of interest have been downloaded. Locher et al. [116] proposed a more complex, but highly effective, approach to downloading files in the BitTorrent system without uploading any data. Specifically, the authors implemented and openly distributed a selfish client (BitThief) that exploits several features of the BitTorrent protocol to attain fast downloads and no contribution.

The example of BitThief is particularly instructive because it shows the easy accessibility of selfish tools also to non-experts. In fact, even if manipulating a client software to implement a selfish behaviour is costly and requires detailed technical knowledge, once this effort has been made, it is simple to distribute the "hacked" client to the Internet community.

1.1.2 *Example: selfishness in P2P live streaming*

P2P live streaming applications, such as PPLive [5] and BitTorrent Live [1], are large-scale cooperative systems that allow millions of users to watch streams of events at the same time. The high scalability of P2P live streaming is ensured by the users' contribution in disseminating the video chunks in the stream, which alleviates the load on the streaming sources. The video chunk dissemination is typically based on gossip protocols. In practice, each user proposes his available chunks to randomly selected partners, who in turn request any chunks they need; the interaction ends when the user delivers the requested chunks.

Selfish users in P2P live streaming systems have many possibilities to receive video chunks while reducing their contribution [53, 72, 111]. For instance, a user may under-report the video chunks available to avoid future requests, or may simply decide to ignore all the requests. Recent studies performed on real live streaming systems [141, 163] have confirmed that the quality of the stream (measured in terms of video discontinuity and latency) received by the cooperative users is substantially reduced by the presence of selfish users in the system.

Collusion among nodes is another manifestation of selfishness, which has been addressed, for example, by Guerraoui et al. in [72]. In their study, the authors observed that colluding nodes could significantly lower the streaming quality by giving each other a higher priority.

1.1.3 *Example: selfishness in volunteer computing*

Volunteer computing aggregates the computational power of millions of Internet-connected personal computers that donate their spare CPU cycles to a computational project. The most

famous of these projects is probably SETI@home, which aims to discover extraterrestrial intelligence by processing radio telescope data [19]. The enormous success of SETI@home led to the release of BOINC (Berkeley Open Infrastructure for Network Computing) [18], which rapidly became the most popular platform for voluntary computing.

Using a BOINC client, volunteers download computational tasks (work units) from a project server, solve the tasks locally, and send the final results back to the server. To encourage cooperation, BOINC rewards volunteers with credit points proportionally to their contribution [18]. The credits are typically displayed on web-based leaderboards accessible worldwide, allowing volunteers to compare their ranking with other users. From a survey conducted by BOINC [12], as well as from the behaviour of BOINC users on online forums [9], the credit system appears to be very motivating for the volunteers. It is so motivating to induce some selfish users to cheat the system only for achieving a better position in leaderboards [7, 10, 11]. Concretely, similarly to BitThief, the original client was hacked and replaced with a selfish client that speeds up the computation by sending untrustworthy results to the server labelled as completed work units.

The selfish behaviour of few "volunteers" may ruin the results of an entire experiment [7], and thus undermine the correctness — or worse, the attractiveness — of a volunteer computing system. The typical countermeasure to this issue is to detect and isolate selfish clients using redundant task allocation, whereby the same work unit is assigned to several clients for result comparison [18]. The drawback of this approach is the large overhead imposed on the clients, which poses an unfavourable trade-off between correctness and performance.

1.2 DEALING WITH SELFISHNESS IN COOPERATIVE SYSTEMS

The ways selfishness has been addressed in the literature on cooperative systems can be broadly divided into two categories: *analysis* and *design*.

Studies in the first category provide analytical models to understand the motivations driving a selfish node, as well as for predicting its expected behaviour. The most comprehensive framework available for this purpose is Game Theory (GT), a branch of economics that deals with strategic interactions in conflict situations [130]. GT provides a set of mathematical tools for modelling competition and cooperation between *rational* (i.e., strategic and selfish) individuals, like the autonomous and self-interested nodes in cooperative systems.

In the last decades, the distributed system community has extensively used GT to assess the robustness of a variety of systems [26, 35, 36, 111, 112, 142, 167], taking advantage of the predictive power and general applicability of the tool. However, as for any analytical approach, applying game-theoretic analysis to real systems tends to be complex, and requires many simplifying assumptions to obtain a tractable model [150]. More precisely, the system designer needs to create a mathematical model of the system (the *game*), including the alternative *strategies* available to the nodes (the *players*) and their preferences over the possible *outcomes* of the system (defined by an *utility function*). Then, the designer has to use game-theoretic arguments to assess what strategy is the most likely to be played by the players, with respect to the given utility function. The answer to this question is the *solution* of a game. The most common so-

lution considered in the literature is the *Nash Equilibrium* [130], which describes the situation in which no player wants to change unilaterally her strategy. Carrying out this process is inherently difficult for complex systems, especially because it is manual, time-consuming and error-prone [118].

The second category of approaches to deal with selfishness in cooperative systems proposes design solutions and mechanisms to enforce cooperation among selfish nodes. Yumerefendi and Chase [178] advocate accountability as a viable solution for dealing with non-cooperative behaviours. Distributed accountability systems [53, 72, 76] provide an implementation of this solution that has been proven feasible and effective for cooperative systems. Particularly, the FullReview system described in [53] can operate in the presence of selfish nodes. On the downside, enforcing accountability incurs a non-negligible cost on the system, mainly due to the high message overhead and the intensive use of cryptography. This results in a fundamental trade-off between performance and resilience to selfish nodes, which poses a difficult configuration problem for the system designers. Since no reference solution is given in the studies cited above, the configuration of such accountability mechanisms is a trial-and-error and time-consuming procedure.

Accountability systems usually address selfishness with a strictly punitive approach, by isolating or evicting selfish nodes [53, 72]. A complementary approach is to introduce incentives for sharing resources, thereby making cooperation more profitable for selfish nodes. The large number of incentive mechanisms proposed in the literature can be classified into reciprocity-based and economy-based. Incentives based on direct reciprocity requires that nodes maintain a history of past interactions with other nodes and use this information to influence the present interaction. For example, BitTorrent applies a direct reciprocity incentive [45], whereby each node prioritises the requests of other nodes based on their history of cooperation. Direct reciprocity offers a lightweight and scalable incentive mechanism, but it can work only if nodes have long-term relationships; otherwise, they may not have the opportunity to reciprocate appropriately. To overcome this issue, indirect reciprocity-based incentives make the interactions between two peers depend not only on the past interactions between them but also on the interactions between them and other nodes. Reputation is the most common incentive based on indirect reciprocity, and it has been widely applied to large-scale dynamic environments like cooperative systems [121]. Reputation offers high flexibility and scalability, and can be implemented in a fully decentralised manner [79, 94, 129, 185]. On the other hand, reputation mechanisms are subject to many types of attacks [121], such as the dissemination of false information (e.g., bad mouthing, unfair praise), and other strategic behaviours (e.g., whitewashing).

In economy-based incentive mechanisms, nodes pay for obtaining services or resources (as consumers) and get paid for sharing resources (as providers). These mechanisms use virtual currency as the commodity for trading resources and allow its later expenditure [25, 67]. The major drawbacks of economy-based incentives are that they also introduce economic issues in the system (e.g., price negotiation, inflation, deflation) [67], and they may require an authority (virtual bank) to issue and certify the currency [25].

Mechanism Design (MD) represents the link between analysis and design approaches for dealing with selfishness in cooperative systems. MD is a branch of game theory that provides

the mathematical framework for designing games, which, as we discussed above, are suitable to model distributed systems. This argument has been well supported by many authors, among which Shneidman et al. [160] and Aiyer et al. [15], who proposed to use MD as a design tool for cooperative systems. The goal of MD is to design a system in such a way that selfish behaviours can be proved to be an irrational choice for selfish nodes. A notable example of the mechanism design approach is the BAR Model of Aiyer et al. [15], which provides an architecture for building cooperative distributed systems that are robust to selfish and Byzantine nodes. However, using mechanism design for designing real systems presents the same limitations mentioned above for game theory analysis, particularly, the complexity added by its analytical approach [15, 26, 27, 111, 118, 150]. Moreover, the resulting design solution suffers from poor maintainability and flexibility: every change in the system parameters requires a full revision of the design, which hinders the reuse of a successful solution in other systems.

1.3 RESEARCH CHALLENGES

Based on the discussion above, we can summarise the *Main Research Challenge* (MRC) of this thesis as follows:

(MRC) Provide integrated support for *designing*, *tuning* and *evaluating* cooperative systems deployed over a network of selfish nodes.

We decompose the MRC into two parts, namely Analysis (A) and Design (D), each facing three sub-challenges.

(A) Provide (conceptual and practical) support for *understanding*, *modelling* and *evaluating* selfish behaviours in cooperative systems.

- (A.1) Develop a framework to understand and classify the motivations as well as the strategies of selfish behaviours in cooperative systems.
- (A.2) Develop tools to specify selfish behaviours in cooperative systems.
- (A.3) Develop a framework to facilitate the development and evaluation of a behavioural model of selfish nodes in a given cooperative system.

(D) Provide practical support for *designing* and *configuring* cooperative systems that meet targeted functionality and performance objectives in the presence of selfish nodes.

- (D.1) Identify general and practical mechanisms to enforce cooperation in cooperative systems.
- (D.2) Develop a methodology to facilitate the set-up and configuration of cooperation enforcement mechanisms in a given cooperative system.
- (D.3) Develop a framework to facilitate the assessment of the resilience of a given cooperative system against different types of selfish behaviours.

1.4 THESIS CONTRIBUTIONS & OUTLINE

Motivated by the challenges presented in the previous section, in this thesis, we present three distinct but interrelated contributions. A brief outline of the contributions is depicted in Figure 1 and discussed below.

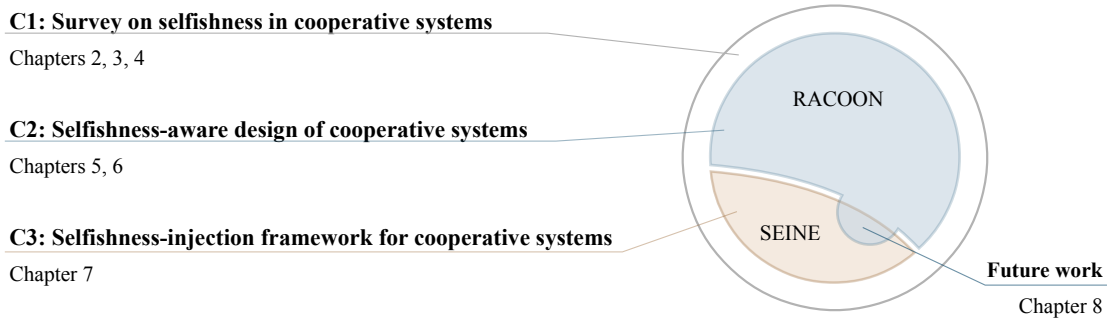


Figure 1: Main contributions of the thesis.

> **(C.1)** A survey on selfishness and its countermeasures in cooperative systems.
> *Related to all challenges.*

In **Chapter 2** we provide a comprehensive overview of the underlying motivations and strategies of selfish nodes in cooperative systems. Based on a systematic review of the extensive literature on the subject, we develop a classification framework to describe and analyse selfish behaviours. Furthermore, in **Chapter 3** and **Chapter 4**, we present the state-of-the-art to analyse and cope with selfishness in cooperative systems.

With **(C.1)**, we provide the conceptual basis on which build the practical contributions introduced next.

> **(C.2)** A framework for the selfishness-aware design of cooperative systems.
> *Related to challenge (A.3), (D.2), and (D.3).*

In **Chapter 5**, we propose RACOON, an integrated framework for designing, tuning and evaluating cooperative systems resilient to selfish nodes. RACOON facilitates the work of system designers by proposing an end-to-end and largely automated design methodology. In particular, RACOON automates the set-up and configuration of accountability and reputation mechanisms into the system under design, using a novel approach based on game theory and simulations. To illustrate the usefulness of our framework, we use it to design a P2P live streaming system and an anonymous communication system. Experimental results and simulations show that the designed systems can meet the targeted level of performance and selfish-resilience.

Chapter 6 presents RACOON++, which extends RACOON. In particular, RACOON++ includes a declarative model that allows parametrizing selfish behaviours, and introduces a new

model to predict the evolution of the system based on evolutionary game theory. We illustrate the benefits of using RACOON++ by designing a P2P live streaming system, a load balancing system, and an anonymous communication system. Extensive experimental results using the state-of-the-art PeerSim simulator, which is completely integrated into our framework, verify that the systems designed using RACOON++ achieve both resilience to selfish nodes and high performance. We released a Java implementation of RACOON++ as an open-source project: code and other data useful for evaluation are available on the project's page on GitHub.[2]

(C.3) A framework for describing and injecting selfish behaviours in cooperative systems.
Related to challenge (A.2), (A.3) and (D.3).

In **Chapter 7** we introduce SEINE, a framework for modelling various types of selfish behaviours in a given cooperative system and automatically understanding their impact on the system performance through simulations. SEINE relies on a domain-specific language, called SEINE-L, for describing the behaviour of selfish nodes, along with a run-time system that generates an implementation of the described behaviours for the state-of-the-art simulator PeerSim [128].

We conclude the thesis in **Chapter 8**, in which we summarise the contributions with respect to the initial research challenges. In this chapter, we also discuss some future work, notably, the scheduled integration between the RACOON and SEINE frameworks.

LIST OF PUBLICATIONS, POSTERS AND SUBMISSION RELATED TO THIS THESIS.

- G. Lena Cota, S. Ben Mokhtar, J. Lawall, G. Muller, G. Gianini, E. Damiani, L. Brunie. **A framework for the design configuration of accountable selfish-resilient peer-to-peer systems.** In Proceedings of the 34th IEEE Symposium on Reliable Distributed Systems (IEEE SRDS'15). 2015.
- G. Lena Cota, S. Ben Mokhtar, G. Gianini, L. Brunie, E. Damiani. **RACOON: A framework to design Cooperative Systems resilient to selfish nodes.** Poster at ACM EuroSys Conference. 2015.
- G. Lena Cota, S. Ben Mokhtar, J. Lawall, G. Muller, G. Gianini, E. Damiani, L. Brunie. **Analysing Selfishness Flooding with SEINE.** Accepted for publication in the 47th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2017).

- G. Lena Cota, S. Ben Mokhtar, J. Lawall, G. Muller, G. Gianini, E. Damiani, L. Brunie. **RACOON++: A Semi-Automatic Framework for the Design and Simulation of Selfishness-Resilient Cooperative Systems.** Under review for the IEEE Transactions on Dependable and Secure Computing (TDSC).

2 RACOON++ on GitHub: https://github.com/glenacota/racoon.

TECHNICAL REPORTS AND OTHER RESOURCES RELATED TO THIS THESIS.

- G. Lena Cota, P-L. Aublin, S. Ben Mokhtar, G. Gianini, E. Damiani, L. Brunie. **A Semi-Automatic Framework for the Design of Rational Resilient Collaborative Systems.** Technical report, http://liris.cnrs.fr/Documents/Liris-6739.pdf.
- **RACOON++** on GitHub: https://github.com/glenacota/racoon.
- **SEINE** on GitHub: https://github.com/glenacota/seine.

LIST OF PUBLICATIONS NOT DIRECTLY RELATED TO THIS THESIS.

- G. Gianini, M. Cremonini, A. Rainini, G. Lena Cota, L. G. Fossi. **A game theoretic approach to vulnerability patching.** In IEEE International Conference on Information and Comm. Technology Research (ICTRC). 2015.
- V. Bellandi, P. Ceravolo, E. Damiani, F. Frati, G. Lena Cota, J. Maggesi. **Boosting the innovation process in collaborative environments.** In IEEE International Conference on Systems, Man, and Cybernetics. 2013.

Part I

SELFISHNESS IN COOPERATIVE DISTRIBUTED SYSTEMS

2

A SURVEY ON SELFISHNESS IN COOPERATIVE SYSTEMS

As highlighted in Chapter 1, cooperation is a necessary condition and a fundamental challenge for any cooperative system such as peer-to-peer (P2P). The success of these systems depends on the resource contribution of the participating nodes, which are autonomous individuals with different goals, roles, and abilities. However, given this autonomy, self-interested nodes may prefer to use the system services while contributing minimal or no resources in return. In the last decade, a significant amount of research has investigated the presence and extent of selfishness in cooperative systems [78, 87, 125, 135, 176, 182]. All these studies confirm that selfish behaviours can lead to a significant degradation of the overall system performance. Thus, a clear understanding of the underlying motivations and strategies of selfish nodes is crucial for optimal design and evaluation of cooperative systems.

The primary contributions of this chapter can be summarised as follows:

- We develop a classification framework to understand motivations and executions of selfish behaviours in cooperative distributed systems.
- We use our classification framework to survey relevant research works related to selfishness in cooperative distributed systems.

Roadmap. We introduce the notion of cooperative systems in Section 2.1. Section 2.2 provides a definition of selfish behaviours and discusses the types of nodes that can participate in a cooperative system, classified according to their goals and capabilities. In Section 2.3, we propose a classification framework to describe motivations and executions of selfish behaviours. We apply the framework to some illustrative cases in Section 2.4, and we use it in Section 2.5 to analyse relevant research work. Finally, we summarise and draw some conclusions in Section 2.6.

2.1 AN OVERVIEW OF COOPERATIVE SYSTEMS

A cooperative system is a complex distributed system that relies on the voluntary resource contribution from its participants to perform the system function. Examples of cooperative systems are peer-to-peer applications (e.g., P2P file-sharing, P2P media streaming, P2P instant messaging), collaborative computing (e.g., volunteer computing, grid computing), and self-organizing wireless networks (e.g., delay tolerant networks). Common to all these systems is the active role of participants in providing the system service, whether this is downloading files, solving scientific problems or routing data in sparse wireless networks. In practice, each participant shares a part of its local resources (e.g., network bandwidth, processing power, or storage space) for the common good of the system.

Participants of a cooperative system are autonomous individuals that cooperate with the system using heterogeneous computational devices (*nodes*). For instance, a node can be a desktop computer, a cluster of processors, a mobile phone, or a sensor device. Nodes are operated independently by their respective owners and typically reside in different administrative domains.

Cooperative systems can leverage on cooperation, heterogeneity and autonomy of the participating nodes to obtain the following characteristics:

Scalability. The capacities of a cooperative system depend on the number of participants that are contributing. Thus, the system's efficiency can grow organically with the system size. It is quite telling that BitTorrent [45] and Skype [23], two of the most popular large-scale applications involving millions of users, are examples of cooperative systems.

Cost-effectiveness. The costs needed to operate the system are shared among a large set of contributors, and not borne by a single entity.

Resilience. Cooperative systems are less vulnerable to faults, attacks, and censorship, due to the high diversity of the potential targets [151], e.g., different hardware and network specifics, software architectures, and geographic locations.

2.2 SELFISH BEHAVIOUR IN COOPERATIVE SYSTEMS

A necessary starting point for our discussion on selfishness in cooperative systems is to provide a clear understanding of what is a cooperative behaviour and what is a selfish behaviour. Our definitions build on the seminal work "Rationality and self-interest in P2P Networks" by Shneidman and Parkes [160], and are also indebted to the taxonomy of non-cooperative behaviours proposed by Obreiter et al. [140].

Definition 2.1 (cooperative behaviour)**.** A cooperative behaviour corresponds to the correct and faithful execution of the protocols underpinning a cooperative system.

Example 2.2.1 (file-sharing). In data distribution systems such as BitTorrent or Gnutella, cooperative behaviours are to advertise the available files and allocate bandwidth to implement the data sharing protocol.

Example 2.2.2 (volunteer computing). Volunteer nodes participating in projects like SETI@home and Folding@home behave cooperatively if they donate their spare CPU cycles to complete the computational job assigned [18, 19].

Example 2.2.3 (P2P live streaming). Live streaming applications such as PPLive [5] allow millions of users to watch streams of events at the same time. In these systems, the cooperative behaviour of peers is to advertise available video chunks and dedicate some upload bandwidth to support the chunks propagation.

Ideally, each participant of a cooperative system implements the relevant cooperative behaviour. Unfortunately, this is very unlikely in real systems, as they are composed of au-

tonomous individuals with own objectives — which may not be aligned with the system objectives — and complete control over the hardware and software of the nodes they use to interact with the system [127]. Long story short: system participants have the ability to modify their behaviour if they are motivated enough to do so.

Notation. *For convenience, henceforth we use the term node to denote both the system participant and its controlled device, assuming that each participant controls exactly one node.*

We can distinguish between two forms of non-cooperative behaviours: selfish behaviours and malicious behaviours. Selfish behaviours have been observed in many real systems [72, 78, 87, 114, 116, 135, 144, 182], in which nodes deviate from the cooperative behaviour because they devised a more profitable behaviour to adopt. The notion of profitability introduces an economic aspect into our discussion. Specifically, cooperation comes at a cost to the contributors, who are expected to consume their own resources (e.g., bandwidth, CPU, memory, storage) for the common good of all nodes. Thus, we can safely assume that the satisfaction that a node derives from participating in the system depends not only on the quality of the service it receives but also on its cooperation costs. The utility of a given behaviour for a given node is a measure (a value) of the node's satisfaction for implementing that behaviour. For the purposes of this chapter, and in conformity with earlier works on cooperative systems design [15, 26, 30, 111], we provide the following informal definition of node's utility:[1]

Definition 2.2 (utility)**.** The utility that a node participating in a cooperative system receives from a given behaviour is defined by the benefit gained from the service provided by the system and the cost of sharing its own resources when performing that behaviour.

Definition 2.3 (profitable behaviour)**.** A behaviour is more profitable than another behaviour if it yields a higher utility.

We can now present our definition of selfish behaviour.

Definition 2.4 (selfish behaviour)**.** A selfish behaviour is an intentional and profitable deviation from the cooperative behaviour.

Example 2.2.4 (message delivery). In wireless networks such as DTN [169] that rely on cooperative message propagation, a node may decide not to relay the traffic of other mobile nodes, in order to extend its battery lifetime and preserve its bandwidth [176, 188].

Example 2.2.5 (cooperative storage). A node of a cooperative storage system (e.g., Pastiche [48] OceanStore [103]) may selfishly withdraw from the resource allocation protocol and discard some data of other nodes to save local storage space [47].

1 A formal definition can be found in Chapters 5 and 6, in which we present our game theoretic model of a cooperative system.

Example 2.2.6 (distributed accountability). PeerReview [76] is a decentralised accountability system, also suitable for P2P networks, which detects misbehaviours by distributing the auditing workload among nodes. However, Diarra et al. [53] showed that PeerReview might suffer from selfish behaviours, because the auditors have no incentive for supporting the overhead introduced by the audit mechanism.

Example 2.2.7 (file-sharing). The Maze file-sharing system [86] includes an incentive system in which peers earn points for uploading and expend points for successful downloads. Despite this incentive, in their empirical study of the Maze network, Lian et al. [114] have detected groups of peers sharing files only among each other while refusing to share any file with peers outside their group.

Based on these examples, we can further elaborate Definition 2.4 with some considerations on the motivations behind selfish behaviours. There are many reasons why a node would choose to stop following the cooperative behaviour, such as:

1. The cost of contributing resources to other nodes outweighs the benefits received from the system.

2. The system does not impose punishments for selfish behaviours nor incentives to encourage contributions [38].

3. The punishment for selfish behaviours (resp. the incentive for contributions) is not fast, certain and large enough to foster cooperation [30].

4. Nodes have economic or social reasons for cooperating only with a restricted group of nodes [114, 121].

5. Nodes suffer from persistent resource shortage, due for example to hardware or software limitations of the device that hosts the node (e.g., battery-powered devices) [75, 176].

6. An exogenous event that occurs in the node's environment, such as temporary connectivity problems, or message loss [30].

According to the taxonomy of non-cooperative behaviours proposed by Obreiter et al. [140], motivations (1-3) have to be considered unjustifiable, and the selfish behaviours they produce should be deterred; conversely, selfish behaviours dictated by motivations (4-5) should be considered justifiable misbehaviours and be exempted from punishments.

Another interesting fact illustrated by the examples above is that selfish behaviours can also occur in protocols specifically designed to address misbehaviours (Examples 2.2.6 and 2.2.7). Note that this is consistent with Definition 2.4 as soon as the protocols in question rely on the cooperation of autonomous nodes.

Another form of non-cooperative behaviours are malicious behaviours, which are defined next.

Definition 2.5 (malicious behaviour)**.** A malicious behaviour is an intentional deviation from the cooperative behaviour that aims at harming the system and disrupting its protocols.

Example 2.2.8 (file-sharing). File pollution is the malicious introduction of tampered contents in file-sharing networks, with the goal of discouraging the distribution of copyright protected contents [115]. Frequent downloads of polluted files can degrade the quality of file-sharing service to the point that the users decide to leave the system.

Example 2.2.9 (P2P overlay). The eclipse attack [164] is a malicious and coordinated behaviour that aim to isolate correct nodes from the P2P overlay and its services by taking control of their neighbour set.

In contrast with selfish behaviours, the only goal of a malicious behaviour is to deteriorate the system performance as much as possible and, possibly, at any cost.

In the remainder of this section, we describe the types of nodes that can participate in a cooperative system, and we suggest the taxonomy shown in Figure 2. Our taxonomy is inspired by the classification of node types proposed by Feigenbaum and Shenker [57] and refined by Shneidman and Parkes [160]. First, we distinguish nodes according to their ability and will to strategize over different behaviours. Nodes that do not strategize can be either correct or faulty (left side of Figure 2).

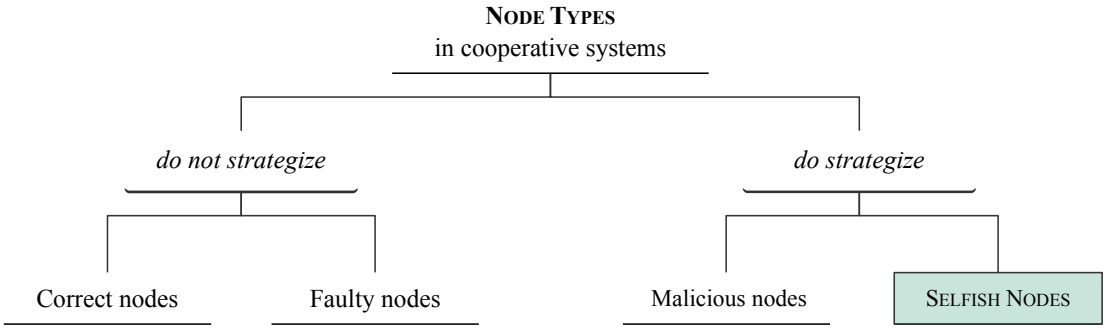


Figure 2: A taxonomy of node types in cooperative systems. The framed box indicates the focus of this thesis.

Correct nodes are also called altruistic [137] or obedient [57, 160], because they faithfully follow the system protocols without pursuing individual interests.

Definition 2.6 (faulty nodes)**.** A node is faulty when it fails to execute the cooperative behaviour due to hardware or software bugs, misconfigurations, or other unintentional faults.

Faulty nodes are incorrectly functioning nodes that may suffer from different types of failures, including fail-stop (the node stops working), omissions (the node drops incoming or outgoing messages), and Byzantine (the node behaves arbitrarily) failures [21, 160].

On the right side of Figure 2 are nodes that behave strategically to pursue personal objectives. We distinguish between malicious and selfish nodes, according to the nature of their objectives.

Definition 2.7 (malicious nodes)**.** A node is malicious when it executes a malicious behaviour (see 2.5) with the intent to degrade the system performance and disrupt the service.

Definition 2.8 (selfish nodes)**.** A node is selfish when it strategically executes the most profitable behaviour, either cooperative (see 2.1) or selfish (see 2.4), with the aim of maximising its expected utility.

When designing a cooperative system, the designer should take all the types of nodes into consideration [57, 136, 160]. More precisely:

- Correct nodes, of course, do not need any special handling.
- Faulty nodes should be detected and repaired (or replaced) promptly, typically using redundancy, cryptographic signing, and Byzantine fault-tolerance techniques [21, 39, 40, 160].
- Malicious nodes can be handled using traditional security approaches for attacks prevention, detection, and removal (e.g., accountability [72, 76, 179], intrusion detection and prevention systems [88, 156], trusted hardware [109, 154]).
- **Selfish nodes have to be incentivised to behave correctly, by making the cooperative behaviour the most profitable behaviour to adopt.**

Multiple mechanisms have been developed to address selfishness in cooperative systems. In general, we can define these mechanisms as follows:

Definition 2.9 (Cooperation enforcement mechanisms)**.** A cooperation enforcement mechanism is a method or collection of methods that aim to increase the utility of a cooperative behaviour by reducing (or eliminating) deviation opportunities, or by rewarding cooperation, or both.

In general, cooperation enforcement mechanisms create a relationship between the contribution that a node provides to the system and the resources that it can consume from it. We refer to Chapter 4 of this thesis for a detailed discussion of how such a relationship can be implemented.

In summary, the ultimate goal of a cooperative system designer is to build the system in such a way that faulty nodes are tolerated, malicious nodes are expelled, and selfish nodes are motivated to choose the cooperative behaviour. This thesis concentrates on the last part of the goal. For a better understanding of the state-of-the-art techniques for handling faulty and malicious nodes, we refer the interested reader to the literature cited above.

2.3 A CLASSIFICATION FRAMEWORK FOR SELFISH BEHAVIOURS

In the previous section, we have already presented a few examples of selfish behaviours, taken from the extensive literature on the subject. Despite their diversity and specificity, we found that these behaviours share several common features in their conception and execution. We believe that identifying these commonalities will provide new clues to understanding and address selfishness in cooperative systems.

To support our claims, we develop a classification framework for the analysis and comparison of selfish behaviours in cooperative systems. The framework, illustrated in Figure 3, is based on the findings of a systematic review of 25 published studies related to our research (more details can be found at the end of this section). Our classification is based on six dimensions, grouped into two categories: the *motivation* for adopting a particular selfish behaviour, and its practical *execution* in a given system. In practice, the execution of a selfish behaviour describes an illegal implementation of the system protocols, which is initiated by a selfish node in attempting to satisfy some personal motivation. We discuss each dimension in the following subsections.

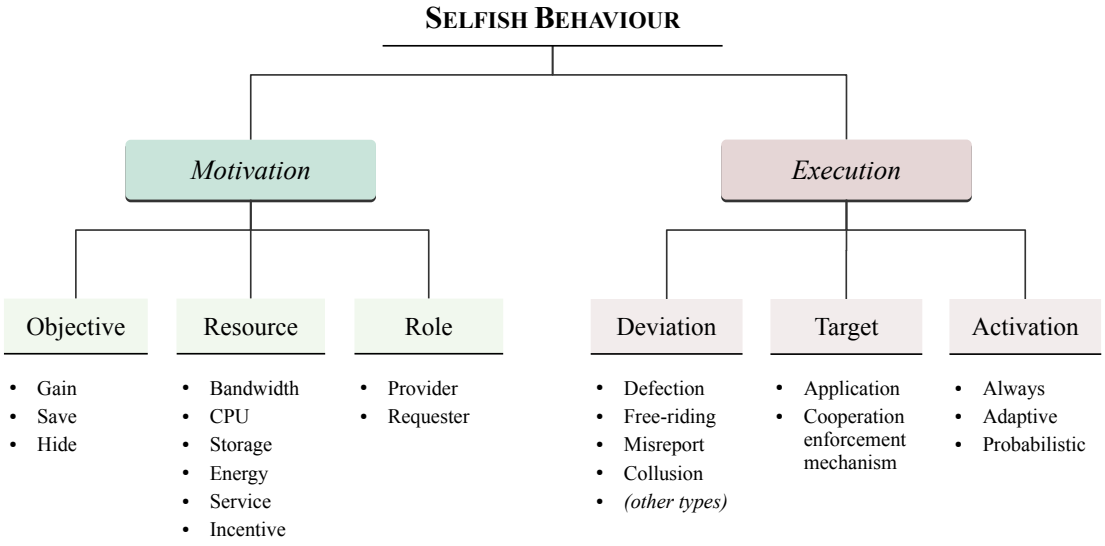


Figure 3: Classification of selfish behaviours in cooperative systems.

2.3.1 *Motivation*

The *motivation* of a selfish behaviour provides information about who commits the behaviour and why. Specifically, a motivation is defined by three dimensions, which describe an *objective* for a given *resource* of interest, along with the *role* being played in the system by the selfish node when it deviated from the cooperative behaviour.

OBJECTIVE. The driving force of any selfish behaviour is to increase the utility of the selfish node that performs it. In Definitions 2.2 and 2.9, we presented the three factors that have a substantial impact on the utility, namely resource consumption, resource contribution, and the incentives introduced by the cooperation enforcement mechanisms. The objective of a selfish behaviour suggests the strategy to increase the utility, which falls into one of the following options:

- *Gain* more resources, to increase the benefits from consumption.
- *Save* local resources, to lower the contribution costs.
- *Hide* misbehaviours from the cooperation enforcement mechanisms, to escape from detection and associated penalties.

RESOURCE. A resource is a commodity that increases the personal utility of nodes that possess it. We distinguish between physical and logical resources. A physical resource is a node's capacity, such as *bandwidth*, *CPU* power, *storage* space, or *energy*. These resources are usually rival in consumption due to congestion.[2] For example, in delay tolerant networks, a mobile node that has reached its storage capacity cannot carry further messages until it frees enough space. Similarly, if the volunteer of a distributed computing project like SETI@home is contributing all its computational power, then it cannot volunteer for another project.

Logical resources are the high-level and application-specific *service* offered by the cooperative system (e.g., file-sharing, message routing), and the *incentive* created by the cooperation enforcement mechanism (e.g., money, level of trust).

ROLE. The motivation behind a selfish behaviour also depends on the role being played by the selfish node when it decided to deviate from the correct protocol execution. In our classification framework, we consider two general roles that apply to all cooperative systems: resource *provider* and resource *requester*. The motivations that drive the behaviour of a selfish provider are usually very different from those of a selfish requester. To illustrate this difference, and as a summary example of the *motivation* of selfish behaviours, consider the situation below.

2 In economics, a good is said to be rival if its consumption by one consumer can negatively affect the simultaneous consumption by another consumer.

Summary Example 1 (*P2P file-sharing*)

P2P file-sharing applications such as BitTorrent and Maze assign a higher priority in download to the peers that share more data. In this setting, a selfish node can increase its utility by performing the following selfish behaviours:

- Lowering the contribution cost of hosting and delivering the files, even if this would reduce the download priority acquired.[3]
  Motivation: (the peer aims to) *save bandwidth* and *storage* when playing as a *provider*.
- Artificially increasing the benefit gaining higher priority in download.
  Motivation: (the peer aims to) *gain* in quality of *service* when playing as a *requester*.

2.3.2 *Execution*

The *execution* of a selfish behaviour provides information about how the behaviour has been implemented by a selfish node. More precisely, an execution is defined by three dimensions, which describe the *deviation* type from a *target* functionality of the system, along with the *activation* policy that has triggered the deviation.

DEVIATION. A (selfish) deviation is the implementation of a selfish behaviour for a particular cooperative system. The wide range of motivations behind a selfish behaviour, as well as the application-specific nature of their implementation, generate a tremendous number of possible deviations for any given cooperative behaviour. Nevertheless, based on our review of the available literature, we could identify four types of deviation that match most of the selfish behaviours analysed.

- *Defection*. A defection is an intentional omission in the execution of a system protocol.[4] Example 2.2.4 presented a case of defection, whereby a mobile node may decide not to participate in the *store-carry-forward* message propagation protocol in order to extend its battery lifetime [176]. A selfish node performs a defection to stop the protocol execution, so as to prevent requesters from consuming or even asking for its resources. A defection can be put into practice by ignoring incoming requests, like in the example above, or by refusing to join a protocol, like the selfish auditors in Example 2.2.6 that withdraw from auditing other nodes.
- *Free-riding*. A free-ride is a selfish deviation that can reduce the amount of resources contributed by a node without stopping the protocol execution. For instance, a peer participating in a P2P live streaming application (see Example 2.2.3) might free-ride the data-exchange protocol by sending fewer video chunks than what was requested by the other

3 For example, the node has already downloaded the desired files, and it is not interested in downloading other files for the moment.

4 Avizienis et al. [21] define an omission as the "absence of actions when actions should be performed".

party [72]. As another example, in volunteer computing (see Example 2.2.2), a node may decide to save some computational time for speeding up the credit collection process [7], and thus free-ride by returning a result without performing the entire computation. The literature on cooperative systems offers other definitions of free-riding, especially in the context of peer-to-peer. Some authors define free-riding the complete lack of contribution [116, 135, 165], while others characterise it as downloading more data than uploading [27, 38, 87]. Our definition is more general because it applies to any type of resource, and it is more precise because it can be clearly distinguished from deviations that achieve the same result by stopping the system protocols (i.e., defections).

- *Misreport*. The correct functioning of a system protocol depends not only on the faithful implementation of operations, such as data transmission or task computation, but also on the truthful exchange of information. Such information may include the hardware specification of nodes (e.g., computational power, memory size, network connections) or their current availability of data and resources. Based on this information, the protocols underlying the cooperative system can operate the system function and optimise the workload distribution among nodes. However, a selfish node may have some motivation for providing false or inaccurate information, for example, to avoid contribution or gain better access to resources. We define this type of deviation as misreport. Consider for instance a P2P application for file-sharing (see Example 2.2.1) or media streaming (see Example 2.2.3), in which peers advertise their sharable contents (files or video chunks). In this scenario, under-reporting the list of sharable contents allows a selfish peer to save upload bandwidth by reducing the number of requests [72, 148].

- *Collusion*. Up to this point, we have considered only deviations performed by a single node. On the other hand, in Example 2.2.7 we introduced a collective form of selfish behaviour in P2P file-sharing, in which multiple peers act together to increase their utility. Similarly to previous works [43, 57, 114, 129, 137], we call this type of deviation a collusion, and we define it as the coordinated execution of a selfish behaviour performed by a group of nodes called colluders. A collusion is more difficult to detect than individual deviations [28, 43, 72], because colluders can reciprocally hide their misbehaviours. For example, the distributed accountability protocol of Example 2.2.6 can be cheated if a sufficient number of colluding nodes stop auditing the logs of their colluders [53, 76].

TARGET. A selfish behaviour can affect different protocols or functionalities of a cooperative system, which we define as the target of a deviation. At a high level, there are two categories of targets: those representing application-related functionalities, and those specific to the cooperation enforcement mechanism. Table 1 reports some examples of the first category of targets, along with a description of typical deviations. Concerning the second category, common targets for a cooperation enforcement mechanism are the *monitoring* and *detection* protocols (e.g., watchdogs, log auditing), and the *incentive* scheme (e.g., reputation, credits).

Target	Functionality	Typical deviation
Data hosting	Contribution of memory space for storing contents, such as files or messages	Discarding stored contents to free memory space
Data transmission	Contribution of bandwidth capacity for delivering contents to other nodes	Transmitting less data than requested
Content sharing	Contribution of contents from a node to other nodes	Advertising only part of the available contents, if any
Overlay management	Maintenance of an overlay network, including resource lookup for finding the hosts of a requested resource	Ignoring lookup queries
Information routing	Maintenance of routing tables and contribution of bandwidth capacity for routing messages through the system	Refusing to forward a message
Partner management	Discovery of new partners for future interactions, and selection of known partners for current interactions [a]	Selecting only colluders
Resource allocation	Distribution of resources among a set of selected nodes [b]	Allocating more resources to colluding partners
Information providing	Provision of truthful information about current state, resource capacities and availabilities	Providing false or inaccurate information
Task computation	Contribution of CPU time for executing a computational task	Returning a result before completing the task

[a] A selection policy might be based for example on the resource capabilities of the known partners (e.g., considering only high-bandwidth nodes), on their availability, or on direct experience.

[b] An allocation policy may, for example, allocate more resources to known partners, or to partners with lower resources.

Table 1: Examples of application-related targets of a selfish deviation.

ACTIVATION. Deviation type and target help provide a picture of *how* a selfish behaviour has been implemented. What is still missing in the picture is the time dimension. This information in provided by the activation policy of the behaviour execution. More precisely, the activation of a selfish behaviour defines the rules or events that trigger its execution. We propose the following activation policies:

- *Always*, when nodes perform the same given selfish behaviour during their whole stay period in the cooperative system. For instance, Locher et al. [116] developed a selfish client of BitTorrent that always implements the same deviations.
- *Probabilistic*, when nodes perform a given selfish behaviour according to some probability distribution. A selfish node may, for example, adopt a probabilistic activation policy to make their deviations unpredictable.

- *Adaptive*, when a given selfish behaviour is triggered by an activation event, such as exceeding a threshold amount of resource consumption, or obtaining a service. For instance, selfish participants of a P2P file-sharing application may decide to deviate from the correct protocol execution only when they are not downloading [114]. Another typical situation, described for example also by Aditya et al. [14], is to provide false information to a monitoring mechanism to cover up previous deviations. To put the last example another way, a selfish behaviour can be the trigger of other selfish behaviours.

To conclude the presentation of the six dimensions of our classification framework for selfish behaviours, we complete the Summary Example 1 by suggesting possible executions for the proposed motivations.

Summary Example 2 (*P2P file-sharing*)

P2P file-sharing applications such as BitTorrent and Maze assign a higher priority in download to the peers that share more data. In this setting, a selfish node can increase its utility by performing the following selfish behaviours:

- Lowering the contribution cost of hosting and delivering the files, even if this would reduce the download priority acquired.
  Motivation: (the peer aims to) *save bandwidth* and *storage* when playing as a *provider*.
  Execution#1: (the peer can) *free-ride* the *data transmission* protocol when not downloading (*adaptive* activation) [87].
  Execution#2: (the peer can) *always misreport* the list of sharable files to other nodes, when executing the *information providing* protocol [116].
- Artificially increasing the benefit gaining higher priority in download.
  Motivation: (the peer aims to) *gain* in quality of *service* when playing as a *requester*.
  Execution: (the peer can) *always collude* with other nodes to cheat the *incentive* mechanism and get rewarded in download priority. Lian et al. [114] describe several collusion strategies observed in the Maze network. For instance, colluders can upload large amounts of traffic among each others in order to artificially inflate their download priority even without contributing to the community at large.

2.3.3 *Details of the papers included in the review*

The classification framework outlined above is based on the review of 25 research papers on the subject. We selected these papers with the following criteria in mind:

Relevance. The papers should be representative of (but not limited to) the types of cooperative systems used as examples in Section 2.2, namely, peer-to-peer systems, distributed computing, and delay tolerant networks.

Quality. The papers should be of particular interest to the research community (estimated considering publishers and number of citations). Also, they should provide detailed descriptions of concrete selfish behaviours.

Coverage. As the commonalities among deviations began to emerge, we selected papers also based on the type of the deviations described therein. In particular, the papers should contribute to a relatively uniform distribution of deviation types, in order to avoid biases due to over-representation of certain deviations.

Table 2 presents the list of papers considered for our review, along with some information on the cooperative system investigated (i.e., application domain, service provided, architecture type). From the analysis of these papers, we could identify 56 selfish behaviours. Almost all behaviours can be classified using one of the four deviation types listed above. The only exception describes a deviation that is very specific to the implementation of the target system — i.e., the rarest-first policy for requesting file pieces in BitTorrent [116]. The right columns of Table 2 show the contribution of each paper to the final distribution of deviation types.

Remark. *We intentionally excluded from our classification the category of selfish deviations that target the authentication protocol of reputation-based cooperation enforcement mechanisms (see Section 4.1 for more details). Deviations in this category, such as whitewashing and Sybil attack [102], exploit the difficulty in establishing node identities in cooperative systems so as to cheat the incentive mechanism, for instance by escaping bad reputations or spreading false reputation information. Although common and well-studied, we decided not to include these behaviours in our classification framework because they are heavily tied to a specific type of cooperation enforcement mechanism, and, therefore, not suitable for generalisation.*

2.4 EXAMPLES OF SELFISH BEHAVIOUR CLASSIFICATION

Hereafter, we present in more details four papers considered for our survey, and we show how to apply our classification framework to the selfish behaviours described by the respective authors. The papers have been selected so as to fulfil the same criteria of relevance, quality and coverage applied to the survey. In particular, the authors of the last paper investigate the impact of selfishness in anonymous communication systems.

2.4.1 *BAR Gossip, Li et al. [111]*

Li et al. [111] address the problem of selfishness in P2P live streaming systems based on gossip protocols, and develop a new protocol (BAR Gossip) that can tolerate both selfish and Byzantine behaviours by the peers receiving the stream. In addition, the authors show that BAR Gossip is stable in the presence of significant collusion (up to 40% of colluders in the system).

Table 2: Papers considered in our review, along with the characteristics of the cooperative systems investigated and the types of selfish deviations therein described.

	Cooperative System			*Deviation types* [a]				
Reference	**Domain**	**Name and/or Service**	**Architecture**	D	F	M	C	O
Ben Mokhtar et al. [28]	Data Distribution	File-sharing, live streaming	P2P	✓	x	x	✓	x
Ben Mokhtar et al. [26]	Data Distribution	FireSpam	P2P (struct.)	✓	✓	x	x	x
Guerraoui et al. [72]	Data Distribution	Media streaming	P2P (unstruct.)	x	✓	✓	✓	x
Hughes et al. [87]	Data Distribution	Gnutella (file-sharing)	P2P (unstruct.)	✓	x	x	x	x
Li et al. [111]	Data Distribution	BAR Gossip (live streaming)	P2P (unstruct.)	x	✓	✓	✓	x
Lian et al. [114]	Data Distribution	Maze (file-sharing)	P2P (unstruct.)	x	x	x	✓	x
Locher et al. [116]	Data Distribution	BitThief (file-sharing)	P2P (unstruct.)	✓	x	✓	x	✓
Piatek et al. [148]	Data Distribution	PPLive (live streaming)	P2P (hybrid)	x	✓	x	✓	x
Sirivianos et al. [165]	Data Distribution	Dandelion (file-sharing)	P2P (unstruct.)	✓	x	✓	x	x
Yang et al. [175]	Data Distribution	Maze (file-sharing)	P2P (unstruct.)	✓	x	x	x	x
Anta et al. [20]	Computing	BOINC	Client-server	x	✓	x	x	x
Anderson [18]	Computing	BOINC	Client-server	x	✓	x	✓	x
Kwok et al. [104]	Computing	Grid Computing	Client-server	x	✓	✓	x	x
Shneidman and Parkes [161]	Computing	Leader-election	Client-server	x	x	✓	x	x
Yurkewych et al. [180]	Computing	BOINC	Client-server	x	✓	x	✓	x
Cox and Noble [47]	Backup & Storage	Samsara	P2P (struct.)	x	✓	x	x	x
Gramaglia et al. [71]	Backup & Storage	P2P Storage	P2P (struct.)	✓	x	x	x	x
Buttyán et al. [36]	Networking	Bulletin board	DTN	✓	x	x	x	x
Blanc et al. [31]	Networking	Overlay management	P2P (struct.)	✓	x	x	x	x
Li et al. [113]	Networking	Message switching	DTN	x	x	x	✓	x
Mei and Stefa [124]	Networking	Message switching	DTN	✓	✓	x	x	x
Shneidman and Parkes [161]	Networking	Inter-domain routing	Internet	x	x	✓	x	x
Zhu et al. [188]	Networking	Message switching	DTN	✓	x	x	✓	x
Ben Mokhtar et al. [27]	Anonym. Comm.	RAC	Proxy servers	✓	✓	x	x	x
Jansen et al. [89]	Anonym. Comm.	Tor [54]	Proxy servers	✓	x	x	x	x
Ngan et al. [135]	Anonym. Comm.	Tor [54]	Proxy servers	✓	✓	x	x	x

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

COOPERATIVE SYSTEM DETAILS In a P2P live streaming system, the dissemination of video chunks throughout the network can be achieved using gossip-based protocols. The typical implementation of these protocols is that each peer proposes its available chunks to randomly selected partners, which in turn request any chunks they need; the protocols end when the peer delivers the requested chunks.

SELFISH BEHAVIOURS The selfish behaviours described by Li et al. in their paper are summarised in Tables (3-5). Consider for example the free-riding and misreport deviations reported in Tables 3 and 4, respectively. A selfish peer may adopt either behaviour for the same motiva-

tion: reducing the bandwidth consumption when participating in the gossip protocol as video chunks provider. The way this motivation is brought into practice is quite different between the two cases. In the first case, the free-riding deviation, a selfish peer first advertises all its shareable chunks, but, once requested, it delivers only a part of them. The selfish behaviour in Table 4 describes a different execution, whereby a misreporting peer under-reports its chunk availability to reduce the probability of receiving chunk requests. Note that the activation policy for both the free-riding and misreport deviations has been classified as adaptive. In fact, Li et al. assume that selfish peers will behave selfishly as long as they experience a video stream of high quality; otherwise, they would stick to the cooperative behaviour to contribute improving the system performance, and enjoy a good quality stream again.

Table 3: Selfish behaviour in P2P live streaming systems: free-riding (source: [111]).

Motivation	**Objective**	**Resource**	**Role** [a]	Description
	Save	Bandwidth	P	Save bandwidth consumption for current activities
Execution	**Deviation**	**Target**	**Activation**	Description
	Free-riding	Data transmission	Adaptive	Send less video chunk than what requested by the partners

[a] P = Provider, R = Requester

Table 4: Selfish behaviour in P2P live streaming systems: misreport (source: [111]).

Motivation	**Objective**	**Resource**	**Role**	Description
	Save	Bandwidth	P	Save expected bandwidth consumption for future activities
Execution	**Deviation**	**Target**	**Activation**	Description
	Misreport	Information providing	Adaptive	Under-report the available chunks

The selfish behaviour reported in Table 5 describes the situation in which peers coordinate their actions to increase their collective utility. The motivation here is to contribute resources only with a subset of the network, thus, decreasing the overall contribution cost. Selfish peers can implement this behaviour by colluding in a coordinated deviation against the partner selection mechanism, so as to establish interactions only among themselves.

The impact of the preceding behaviours on the live streaming performance has been evaluated by Li et al. through experiments and simulations [111]. The results show that, without any cooperation enforcement mechanism in place, the presence of 50% of selfish nodes can prevent correct nodes from watching a good quality stream.

Table 5: Selfish behaviour in P2P live streaming systems: collusion (source: [111]).

Motivation	**Objective**	**Resource**	**Role**	Description
	Save	Bandwidth	P	Save bandwidth consumption colluding group
Execution	**Deviation**	**Target**	**Activation**	Description
	Collusion	Partner management	Always	Bias the partner selection and prefer colluders to other peers

2.4.2 *BOINC client, Anta et al. [20], Anderson [18], Yurkewych et al. [180]*

BOINC [18] is an open-source client for volunteer computing, which aggregates the computational power of millions of Internet-connected personal computers (*volunteers*) that donate their idle resources to a computational project (e.g., SETI@home [19]). The paper of Anderson [18] marginally, and the papers of Anta et al. [20] and Yurkewych et al. [180] particularly, address the problem of selfish volunteers in the BOINC system, whose main goal in to earn credits faster even if at the expense of results accuracy and system efficiency. Specifically, all authors agree on using redundant task allocation techniques to detect selfish activities and punish selfish volunteers with (virtual [18] or real money [20, 180]) monetary fines.

COOPERATIVE SYSTEM DETAILS As already introduced in Chapter 1, each project in BOINC is hosted on a server that provides volunteers with work units. Once a volunteer has computed (offline) the work unit, it sends the result back to the server [18]. To motivate cooperation, BOINC rewards volunteers with credit points proportionally to their contribution, and makes available their scores on web-based leaderboards accessible worldwide.[5]

SELFISH BEHAVIOUR It is well documented that the BOINC credit system has attracted not only volunteers but also selfish users, whose main goal is to rise in the credit leaderboards ranking rather than making significant and reliable contributions to the project. For example, the "The SETI@home problem" article by David Molnar [7] reports on two selfish BOINC clients that have been developed, notably, also by Microsoft, to speed up the computation by sending untrustworthy results to the server labelled as completed work units (more details can be found here [8]). More recently, other volunteer computing projects that rely on the BOINC platform have faced severe issues with selfish behaviours. Particularly, in 2013, a small group of 17 selfish volunteers in the PrimeGrid project[6] — i.e., collaborative searching for prime numbers — have faked their results leading to approximatively one month's worth of work to be redone [10]. Similar behaviours, though with almost no negative consequences on the validity of results, have also been detected in some World Community Grid's [7] research projects [11].

5 http://boincstats.com/
6 https://www.primegrid.com/
7 https://www.worldcommunitygrid.org/

Table 6 reports the classification of the selfish behaviour described by Anta et al. [20], Anderson [18], and Yurkewych et al. [180] in their papers. The motivation for such behaviour is to earn more credits and faster in order to climb the ladder of the leaderboards ranking. With this intention, a selfish volunteer may use a hacked BOINC client that is optimised for fast computation rather than trustworthiness of results. More precisely, Table 6 describes a free-riding deviation aiming at reducing the amount of task computation activities (e.g., data input and accuracy checks).

Table 6: Selfish behaviour in volunteer computing: free-riding (source: [18, 20, 180]).

Motivation	**Objective**	**Resource**	**Role** [a]	Description
	Gain	Incentive	P	Obtain more credits than is due
Execution	**Deviation**	**Target**	**Activation**	Description
	Free-riding	Task computation	-	Send an untrustworthy result

[a] P = Provider, R = Requester

2.4.3 *Delay tolerant network, Zhu et al. [188]*

In their presentation of an incentive scheme (SMART) to foster cooperation in DTNs, Zhu et al. also provide some examples of selfish behaviours that may occur in these systems. To address this problem, the authors introduce a virtual currency to charge for and reward the cooperative delivery of messages within the network. The SMART scheme has been evaluated through simulations, which showed its applicability, reliability, and effectiveness also in the presence of selfish nodes [188].

COOPERATIVE SYSTEM DETAILS Delay tolerant networks achieve end-to-end connectivity over a disrupted network by asking mobile nodes to participate in a message propagation process called *store-carry-forward*. This process relies on the cooperation of intermediate nodes, which carry the messages of other nodes until the next hop of a communication path appears [56].

To motivate selfish nodes to participate in a *store-carry-forward* protocol, Zhu et al. introduce in DTNs a secure virtual currency (called layered coin) along with a payment scheme for rewarding the provision of message forwarding [188]. This scheme works as follows: the source node generates a multi-layered coin to transmit together with the message. The base layer created by the source contains information about reward amount (credit value) and conditions. Each intermediate node in the *store-carry-forward* process adds a layer to the coin, providing details about its identity. Once the message has reached its destination, the recipient of the message can distribute the credit to all intermediate nodes by using the identities stored in the layered coin.

SELFISH BEHAVIOURS As already mentioned in Example 2.2.4, the hypothesis that each node in a DTN is willing to forward messages to others might be unrealistic. In fact, participating in the *store-carry-forward* constitutes an important cost for mobile nodes with limited energy, bandwidth and storage constraints. A selfish node motivated by the objective of saving these resources may perform the selfish behaviour classified in Table 7. The simplest implementation of this behaviour is to ignore any request to serve as an intermediate node during the whole stay period in the DTN.

Table 7: Selfish behaviour in delay tolerant networks: defection (source: [188]).

Motivation	**Objective**	**Resource**	**Role** [a]	Description
	Save	Bandwidth, storage, energy	P	Save resources dedicated to cooperation activities
Execution	**Deviation**	**Target**	**Activation**	Description
	Defection	Information routing	Always	Never participate in the message propagation protocol

[a] P = Provider, R = Requester

The selfish behaviour described in Table 8 is related to the cooperation enforcement mechanism proposed by Zhu et al. [187, 188]. The motivation for this behaviour is to increase the amount of layered coins received by the payment scheme, to gain more purchasing power when acting as a service requester. With this intention, a selfish node may collude with other nodes to forge a valid credit and reward themselves for "forward" operations they have never done. In their paper, Zhu et al. also discuss a solution for this behaviour, based on the concatenation of different layers.

Table 8: Selfish behaviour in delay tolerant networks: collusion (source: [188]).

Motivation	**Objective**	**Resource**	**Role**	Description
	Gain	Incentive	R	Obtain more credits than is due
Execution	**Deviation**	**Target**	**Activation**	Description
	Collusion	Incentive mechanism	-	Collude with other nodes to increase the rewards

2.4.4 *Tor network, Dingledine et al. [135]*

Tor [54] is an anonymous communication system that uses Onion Routing [70] to protect its users from traffic analysis on the Internet. This system works by relaying traffic over a network of voluntary nodes located around the world. However, Ngan et al. have reported that while the number of Tor users keeps growing, the number of relay nodes is not [135]. In their paper,

the authors investigate the reasons for this lack of cooperation and propose an incentive scheme to reward Tor relays with a higher quality of service.

COOPERATIVE SYSTEM DETAILS Using Tor, when a source node wants to send a message to a destination node, it first fetches the list of available relay nodes from trusted *directory authorities*. Then, the source node builds a circuit of voluntary relay nodes. Circuits are updated periodically, and relays can participate in multiple circuits at the same time. To protect a message, the source encrypts it with the public key of the destination. Furthermore, to protect the communication channel, the source uses the public key of each relay node in the circuit to encrypt the address of the next relay node. The resulting message is called an *onion*. A relay uses its private key to decrypt one layer of the onion and contributes a part of its bandwidth to forward the resulting message to the next relay until the message eventually reaches its destination.

To incentivise Tor users to act as a relay, the directory authorities measures the performance of relays to prioritise the traffic of the most active ones, i.e., the so-called "gold star" relays [135]. In practice, a gold star relay's traffic always gets relayed ahead of other traffic.

SELFISH BEHAVIOURS Ngan et al. [135] reported two selfish behaviours in Tor. The motivation of both behaviours, summarised in Tables 9 and 10, is to reduce the communication and computational burden of nodes when actively playing as a traffic relay (i.e., a service provider). A relay node executing the selfish behaviour described in Table 9 behaves cooperatively until it gets rewarded with a gold star; then, it stops participating in any relaying activity. Table 10 reports a different adaptive strategy, whereby nodes cap their relaying at a given bandwidth threshold and free-ride any other request that would exceed that threshold.

Table 9: Selfish behaviour in Tor networks: defection (source: [135]).

Motivation	**Objective**	**Resource**	**Role** [a]	Description
	Save	Bandwidth, CPU	P	Save resources dedicated to cooperation activities
Execution	**Deviation**	**Target**	**Activation**	Description
	Defection	Data transmission	Adaptive	Never participate in the system as relay node

[a] P = Provider, R = Requester

To evaluate the performance of the preceding behaviours, Ngan et al. built "a packet-level discrete event simulator that models a Tor overlay network" [135]. Then, they studied the

Ngan et al. evaluated the performance of the preceding behaviours by simulating an onion routing network composed by a third of correct relays, another third of relays performing the selfish behaviour in Table 9, and the remaining relays following the behaviour in Table 10. The authors considered as performance measures the download and ping time of a relay node when acting as a service requester (i.e., a Tor user). Results demonstrate that when no incentive

Table 10: Selfish behaviour in Tor networks: free-riding (source: [135]).

Motivation	**Objective**	**Resource**	**Role**	Description
	Save	Bandwidth, CPU	P	Contribute less bandwidth than the fair share
Execution	**Deviation**	**Target**	**Activation**	Description
	Free-riding	Data transmission	Adaptive	Relay traffic only up a given threshold

mechanism is in place, the performance of selfish relays is slightly better than correct nodes. Therefore, according to the evaluation by Ngan et al. [135], the utility of a selfish relay in Tor can increase not only due to the lower contribution costs, but also to an increase (even if modest) of the service benefits.

2.5 GENERAL ANALYSIS OF SELFISHNESS IN COOPERATIVE SYSTEMS

In Section 2.3, we have developed a new framework to classify the behaviour of selfish nodes in cooperative systems, based on a systematic review of the state-of-the-art. We now apply the framework to the reviewed papers, and we use it as a comprehensive analysis toolkit to identify emerging patterns and characteristics of selfish behaviours. Our presentation of this analysis is structured in accordance with the six dimensions of a selfish behaviour as presented in Section 2.3.

Notation. *Hereafter, we refer to the list of reviewed papers as the "paper dataset", and we refer to the entire set of selfish behaviours described in the paper dataset as the "behaviour dataset".*

2.5.1 *Analysis of the motivations*

The most common motivation for a selfish node to behave selfishly is to save bandwidth when participating in the cooperative system as a resource provider. Every other motivation in the behaviour dataset follows this pattern, regardless of the application domain. Below, we elaborate on this point and take a closer look at each dimension of a selfish motivation.

RESOURCE The application domain of a cooperative system has a strong influence on the resources of interest for its selfish participants. For example, selfish volunteers in distributed computing systems are more sensitive to their CPU availability and credit scores [18, 20, 180], whereas selfish mobile nodes in DTNs takes energy consumption in more account [31, 36, 113, 124]. Notably, the energy consumption in battery-powered nodes is strictly related to the CPU utilisation and, even more, to network traffic.[8]

8 Al-Karaki and Kamal reported that sending a bit over 10 or 100 m distance consumes as much energy as performing thousands to millions of arithmetic operations [16].

A second observation we can draw from the behaviour dataset is that selfish nodes are often concerned about their bandwidth capacity, regardless of the application domain at hand (e.g., in data distribution [28, 87, 111, 116, 148], networking [113, 124, 188], backup [47, 71], and anonymous communication [27, 27, 135]). This finding is perhaps expected since distributed systems, in general, and cooperative systems, in particular, require intensive message-passing. As a result, bandwidth consumption places a significant burden on the utility of any selfish node.

ROLE & OBJECTIVE A clear message we get from the analysis of the behaviour dataset is that the vast majority of selfish behaviours are initiated by resource providers. The most likely cause of this result is that providers are typically in charge of the most cost-intensive functionalities of a cooperative system, such as data transmission [72, 87, 113, 148], task computation [18, 20, 104, 180], and monitoring [27, 28, 180]. This argument is reinforced by the finding that saving resources is the typical objective of a selfish behaviour. On the contrary, we found that resource requesters usually behave selfishly to further increase their benefits, i.e., gaining more resources than what they should according to the system protocols [26, 116, 165, 188].

The two studies [28, 180] from the paper dataset that address selfish behaviours with a "hide" objective are the only ones that do not take the robustness of the selfishness detection mechanism in place for granted. We recall from Section 2.3 that the rationale of this objective is to conceal misbehaviours, possibly colluding, in order to avoid punishments that could decrease a selfish node's utility. For example, redundant task allocation is a common mechanism in volunteer computing systems to detect misbehaviours [18]. However, Yurkewych et al. [180] described how selfish colluders could avoid detection if they agree on providing the same wrong result to the project server, so as to cover each others' deviations.

2.5.2 *Analysis of the executions*

Before proceeding with the analysis of the behaviour dataset, we would like to remind the reader that one criterion to create the paper dataset was to ensure an as uniform as possible representation of all deviation types, in order not to be biased by over-representation of certain types. On the other hand, it is important to bear in mind that the uniform distribution is not representative of the actual attention given to each deviation type in the literature; on the contrary, we found that most of the studies on the topic focus on free-riding.

After these preliminary remarks, we can now complete the analysis of the behaviour dataset by introducing the three execution dimensions into consideration. As we will see, this broader perspective will allow us to identify interesting patterns across the motivations and executions of selfish behaviours.

DEVIATION We start by examining the relationship between selfish objectives and deviation types. Figure 4 illustrates the typical deviations that selfish nodes perform to put their objectives into practice. As we already mentioned, the most common objective that motivates a

selfish behaviour is to save resources (green areas in the figure). In our behaviour dataset, a significant proportion of selfish nodes tries to fulfil this objective performing either free-riding or defection deviations, which is reasonable in general as these deviations are the most convenient and direct in cutting off resource contribution. In contrast, we found that increasing the benefits obtained from the system is not as simple as reducing costs. Rather, it requires more sophisticated strategies (pink areas in Figure 4), chiefly providing false information [111, 116, 161, 165] or collude [72, 114, 188].

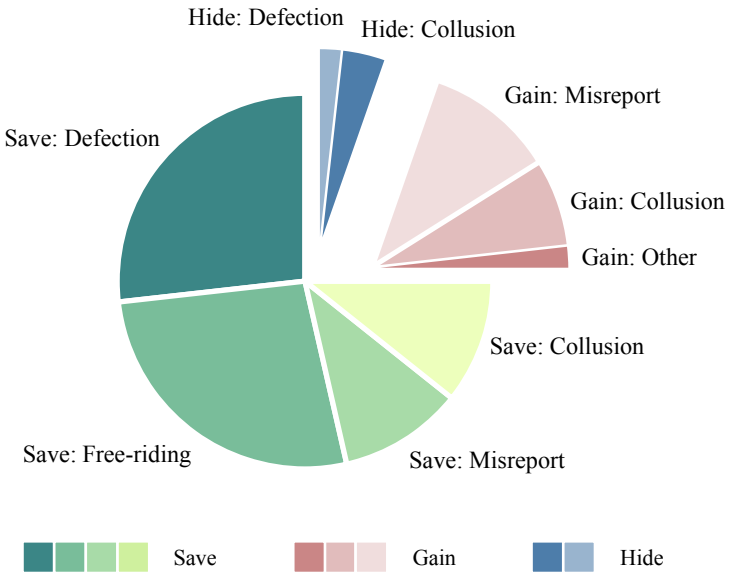


Figure 4: Overview of the deviation types implementing the objectives of the selfish behaviours considered for our analysis.

We already know that saving bandwidth is the principal motivation behind a selfish behaviour. Remarkably, it is also the most common motivation associated with each deviation type in our behaviour dataset, especially for defection and free-riding deviations. To be more precise, if we consider collusion, saving bandwidth is the most frequent motivation together with receiving more incentives. For example, the selfish collusion in P2P live streaming presented in Table 5 aimed to save bandwidth by biasing the selection of interaction partners [111], whereas the collusion described in Table 8 could raise the reward distributed to the colluders [188].

Also noteworthy about collusion is that this deviation type can be executed in combination with other deviations, which, of course, can exacerbate the negative impact on the cooperative system. As an illustration, let us consider again the example of collusion in volunteer computing described by Yurkewych et al. [180]. In that case, the collusion consisted in covering-up a free-ride (stopping the task computation before completing it) by coordinating a misreport deviation (sending the same incomplete result to the project server).

TARGET Figure 5 illustrates the primary targets of selfish deviations in the behaviour dataset. For clarity, we aggregated some functionalities listed in Section 2.3.2: *data management* includes

data transmission, data hosting and data sharing, and *partner and overlay management* includes overlay management, information routing, partner management and resource allocation. A very large majority of selfish behaviours in our dataset target application-related functionalities. Indeed, most of the papers reviewed assumed no selfishness in the cooperation enforcement mechanisms, but only in the application at hand. However, assuming perfect selfishness-resilience of such mechanisms is as unrealistic as assuming a cooperative system of all correct nodes [28, 53, 180]. We will further investigate on this in the next chapters of this thesis.

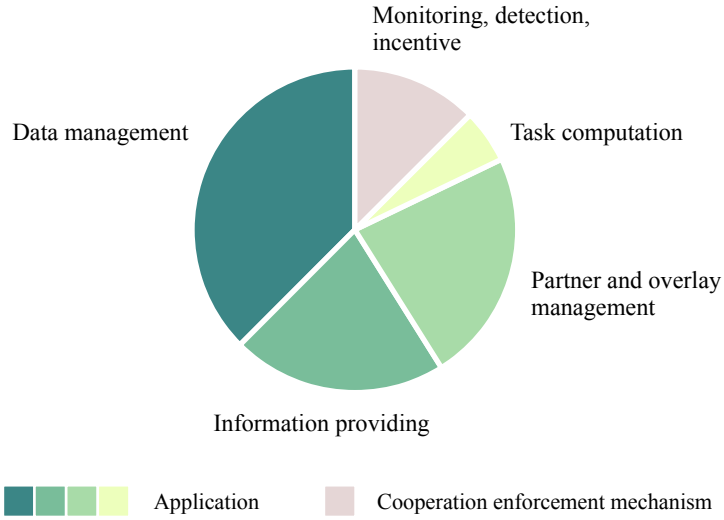


Figure 5: Overview of the main functionalities of a cooperative system that are targeted by the selfish behaviours considered for our analysis.

The application domain of a cooperative system can affect the target of a selfish behaviour. For instance, deviations from task computation are the main issue in distributed computing [18, 20, 104, 180], whereas data hosting is the primary target in cooperative storage systems [47, 71]. Also, the system architecture has an impact on the target of a selfish deviation. As an illustration, in centralised systems such as anonymizing networks and Grids, there are no selfish behaviours that target partner selection or overlay management protocols, as such functionalities are either absent or more rigidly controlled.

ACTIVATION The last dimension of our classification framework defines the rule or event that activates a selfish deviation. However, of the 56 selfish behaviours included in our dataset, we cannot be completely confident about the reliability of more than half of the activation policies specified for their execution (e.g., the authors provided no rules that may trigger a deviation, or failed to provide a clear justification for their choice). Unfortunately, given this gap, we could not pursue a faithful and trustworthy analysis of the behaviour dataset along the activation dimension.

The activation policies available in our behaviour dataset are either based on empirical observations of real-world cooperative systems (e.g., studies on Gnutella [87] and Maze [114, 175]), or constructed by design (e.g., BitThief [116]).

2.6 SUMMARY

In this chapter, we provided the theoretical basis along with conceptual tools for reasoning on selfishness in cooperative distributed systems. Although related to all the research challenges presented in Section 1.3, this chapter directly addressed challenge (A.1) and (A.2), i.e., developing adequate support for understanding, classifying and specifying selfish behaviours.

After a brief overview of representative examples of cooperative systems, we gave a definition of selfish behaviour based on the notions of autonomy and utility, arguing that selfish behaviours are intentional and profitable deviations from the faithful execution of the protocols underpinning a cooperative system. Furthermore, we proposed a taxonomy of nodes participating in cooperative systems. Specifically, we identified four types of nodes — i.e., correct, faulty, malicious, and selfish nodes — based on their correctness and capability to strategise.

Then, we developed a classification framework to enable the analysis and comparison of selfish behaviours in cooperative distributed systems. The framework is based on the systematic review of 25 published studies on the subject. Our classification is based on six dimensions, grouped into two categories: the *motivation* for adopting a particular selfish behaviour, and its practical *execution* in a given system. The three dimensions for motivating a selfish behaviour provide information about who commits the behaviour (*role*) and why (*resource* and *objective*). On the other hand, the execution of a selfish behaviour describes the *deviation* type (i.e., defection, free-riding, misreport, or collusion) from a *target* functionality of the cooperative system, along with the *activation* policy that has triggered the deviation. Finally, we tested the classification framework by describing eight selfish behaviours in four different cooperative systems.

Lastly, we used our classification framework to conduct a comprehensive analysis of the studies mentioned above, with the aim to identify emerging patterns and key characteristics of selfish behaviours. In particular, we found that one of the most common motivation for a selfish node to behave selfishly is to save bandwidth when participating in the cooperative system as a resource provider. Also, we noted that defection and free-riding are often selected as deviation types for achieving this objective, because they can be very effective in cutting down resource consumption.

3

ANALYSING SELFISHNESS IN COOPERATIVE SYSTEMS

The last two decades have witnessed a notable growth in the literature addressing the problem of selfish behaviours in cooperative systems. We can broadly distinguish between studies that focus on the *analysis* of selfishness and studies that *design* countermeasures to foster cooperation among selfish nodes. The scope of this chapter is limited to the first category of studies, especially game theory, leaving the discussion of the countermeasures to Chapter 4.

The contributions of this chapter are summarised below:

- We provide a comparative review of state-of-the-art approaches to analyse selfishness in cooperative systems.
- We present a detailed description of the main characteristics, advantages and limitations of using game theory for system analysis.

Roadmap. In Section 3.1 we introduce the main approaches to analyse selfish behaviours in cooperative systems, namely, analytical and experimental. In Section 3.2 we present a brief discussion of game theory and review research in which game theory has been applied to system design problems. Then, in Section 3.3, we provide a comparative review of analytical and experimental approaches to analyse selfishness. Finally, we summarise the chapter in Section 3.4.

3.1 APPROACHES TO SELFISHNESS ANALYSIS

In this section, we discuss the related work for research challenge (A), i.e., understanding, modelling and evaluating selfish behaviours in cooperative systems. The existing approaches can be broadly divided into analytical and experimental. We discuss each of these categories below.

3.1.1 *Analytical approaches*

Analytical approaches provide mathematical tools to reason about selfishness and cooperation in competitive situations like those underlying a cooperative system [26, 28, 71, 82, 111]. Game theory [130] has become the most prominent analytical framework for modelling selfishness in cooperative systems, due to the predictive power and general applicability of the tool. Much work on the potential of game theory has been carried out in the context of content-disseminating applications [28, 111, 112, 142], wireless and mobile networking [35, 36, 167], cryptography [13, 95], security [69], anonymity and privacy mechanisms [27, 63].

One reason for the success of game-theoretic applications in the context of cooperative systems can be explained by the socio-economic dimension of these systems [143]. In fact, autonomy, self-interest and economic competition are not only the main characteristics of the participants of a cooperative system (see Section 2.2), but also the basic ingredients of a game [130]. Therefore, game theory appears as the natural candidate for the formal representation of selfishness in these systems.

However, applying formal approaches to study real systems tends to be complex [118, 150]. Particularly, game theory approaches require manually creating a mathematical model of the system (the game), including the alternative strategies available to the system participants (the players) and their preferences over the possible outcomes of the system. Then, game-theoretic arguments have to be formulated to assess what strategy is the most likely to be played by the players. In addition to being complex and time-consuming, carrying out this process is also prone to modelling errors, due to assumptions and simplifications to make the model tractable [150].

A more detailed discussion of the objectives, characteristics and limitations of game theory as a tool to analyse selfishness in cooperative systems is deferred to Section 3.2.

3.1.2 *Experimental approaches*

Experimental analysis of distributed systems has a long tradition in Computer Science [50, 51, 58] and can be an appropriate (or even necessary) solution to overcome the shortcomings of analytical modelling [24, 66, 74]. In fact, an empirical approach based on experiments allows gathering some knowledge through observations in the case where the scenario to investigate is hard to formalise analytically. This is particularly true in complex distributed systems such as cooperative systems, due to their large scale, the high dynamics of the environment, and the high heterogeneity and autonomy of their participants (see Chapter 2).

METHODOLOGIES FOR EXPERIMENTAL ANALYSIS. Gustedt et al. [74] proposed a general classification of methodologies for experimental analysis of large-scale distributed systems, depending on whether the experiment needs to execute (i) a real application or a model of an application on (ii) a real environment or a model of a system environment. For instance, an application can be a downloadable software program (e.g., BitTorrent [45], eMule [2], PPLive [5]) or a set of communication protocols, whereas a system environment can be a distributed network of real machines, operating systems or middleware. Table 11 shows the four resulting classes of methodologies, namely, real experiments (named “in-situ", by Gustedt et al. [74]), emulation, benchmarking, and simulation. We discuss each class below.

Real experiments. Real experiments offer the least abstraction and the most accurate system analysis, as they consist in observing the behaviour of a real system implementation running on a real network. However, because cooperative systems typically consist of very large numbers of nodes, performing real experiments could be very costly in terms of hardware requirements and administration effort. Moreover, using real machines may hinder the re-

		Environment	
		Real	*Model*
Application	*Real*	Real experiments	Emulation
	Model	Benchmarking	Simulation

Table 11: Classification of methodologies for experimental analysis.

producibility of an experiment, due to external factors that could influence the experiment results but are extremely difficult to control (e.g., network traffic and CPU usage). In order to reduce hardware and effort costs, as well as to improve reproducibility, it is possible to use experimental testbeds designed for performing experiments in real-scale distributed environments. Among these testbeds, we cite Grid'5000 [37] and PlanetLab [41], which are more suitable for large-scale distributed systems — particularly, the focus of Grid'5000 is more on grid computing, whereas PlanetLab is more oriented to peer-to-peer.

Emulation. An emulator supports the execution of (almost) unmodified applications on various models of system environments. Such models must be realistic enough to reproduce some of the characteristics of a real network (e.g., delays, message loss), while trying to give a controllable, predictable, and repeatable experimental environment. Emulation experiments, and specifically those based on ModelNet [168] and Emulab [173], are becoming very popular in many research areas [162], including complex distributed systems such as cooperative systems.

Benchmarking. The goal of a benchmarking experiment is to help understand how a real system environment will perform when a particular model of an application is running on it. Such a model is carefully designed and configured in such a way as to analyse specific systems' performance, such as dependability [81, 117, 155].[1]

Simulation. Simulation is a classical approach in Computer Science, and it is widely used to perform experiments on cooperative systems [26, 71, 104, 111, 124, 135]. A simulation consists in the execution of a model of an application on a model of the system environment. This allows for perfect control over the experimental conditions, high reproducibility, faster execution time, and the possibility of simulating millions of nodes on a single host [24]. These features come at the price of a high level of abstraction, which can introduce some bias in the experimental results [74]. Relying on a well-established and extensively tested simulator provides more certainty on the accuracy of the results. Naicken et al. [131] surveyed nine existing simulators suitable for performing experiments in cooperative systems, especially in peer-to-peer. Among these, and according to a recent survey [24], the most used are the general network simulators ns (ns-2 [3] and its recent update ns-3 [80]) and OMNeT++ [4], and the P2P PeerSim simulator [128].

1 Dependability of a system is a measure of its ability to perform its function in a trustworthy way [21].

EXPERIMENTAL ANALYSIS OF SELFISHNESS. An experimental analysis of the impact that selfish behaviours have on the system performance can be done only by modelling and injecting such behaviours into the system under consideration, which is a non-trivial task. Indeed, to carry out this task, one has to first analyse the functional specification of the considered system and identify those steps (e.g., functions) for which selfish nodes may behave in a non-cooperative way. Then, on each of the identified steps, one has to decide what are the possible types of selfish behaviours that are meaningful in the system context, e.g., defection, free-ride, misreport, or collusion deviation types (see Section 2.3.2 for a detailed description). Finally, one has to integrate the selected behaviours into the system application, and then invest considerable effort in launching experiments to analyse how the system reacts.

For the purpose of these experiments, existing frameworks supporting the experimental evaluation of cooperative systems can be of some help. For example, the SPLAY environment proposed by Leonini et al. [108] can simplify the evaluation of large-scale distributed systems in real and emulated testbeds such as PlanetLab, ModelNet and EmuLab. Furthermore, SPLAY provides mechanisms to control the dynamics (churn) of the system, which can be used to inject faulty behaviours (node failures). Churn management and basic fault injection support are also provided by various simulation frameworks, such as PeerSim [128], NS [3], and ProtoPeer [66]. Particularly, the RCourse library [122] extends the PeerSim simulator with new and more flexible fault injection capabilities. For example, RCourse can simulate various types of hardware and software faults (e.g., omission failures, message loss, message tampering), and it parameterizes some aspects of the faulty behaviours (e.g., occurrence rate or probability, fraction of nodes involved in a certain fault). For completeness, it is also worth mentioning that benchmark suites for evaluating the dependability of distributed systems allow conducting more sophisticated and accurate fault-injection experiments, but are tailored to very specific application domains — e.g., MRBS for MapReduce systems [155], WS-FIT [117] and Gremlin [81] for web-services.

What is interesting with the possibility of injecting faults into the system is that a fault may have the same implementation as a certain type of selfish deviations, namely, defection. Consider, for instance, Example 2.2.4, presented in the previous chapter and repeated below for convenience.

> *Example* 2.2.4 (message delivery). In wireless networks such as DTN [169] that rely on cooperative message propagation, a node may decide not to relay the traffic of other mobile nodes, in order to extend its battery lifetime and preserve its bandwidth [176, 188].

The selfish deviation described in Example 2.2.4 might be implemented as an omission failure, where a node omits to respond to a request for forwarding messages of other nodes.

Apart from the very simple type of selfish deviations mentioned above, the existing tools for experimental analysis in cooperative systems do not provide any specific support for modelling and injecting selfishness, which has to be done manually. In practice, it is necessary to hard-code both the control and logic of selfish behaviours into the parts of the system implementation that are affected by that behaviours. This activity typically results in generating variant implementations of the same system (i.e., one for each node behaviour), or in creating a

single implementation that incorporates all the possible behaviours as well as the algorithmic functionalities for their control (e.g., variables, if-clauses). The increased complexity and redundancy of the source code reduce its readability, maintainability, and constant evolution. Finally, once a system implementation is available, an extensive experimental campaign is required to quantify the harm caused by various selfishness manifestations of different proportions of selfish behaviours. Such a domain-specific evaluation has to be conducted manually, which is tedious and time-consuming.

3.2 RELATED RESEARCH: GAME THEORY

Game Theory (GT) is a mathematical framework originating from economics that deals with strategic interactions in conflict situations. GT has become increasingly popular for modelling selfishness in P2P and other cooperative systems, due to the predictive power and general applicability of the framework.

In this section, after a short introduction to the basic concepts of GT, we present some relevant applications to cooperative systems, and we discuss the main issues from a system design perspective. For a detailed introduction to game theory, we refer to the recent book [130] by Myerson.

3.2.1 *Basic concepts*

Game theory describes conflict situations as *games* between *rational* (i.e., strategic and self-interested) decision-makers known as *players*.[2] The game model includes the set of *actions* available to each player, along with their constraints, costs and benefits throughout the game. The *strategy* for a player is a complete plan of actions to be taken during the play. If the strategy indicates a unique action for each decision point, then it is called a *pure strategy*; otherwise, it is called a *mixed strategy* if it specifies a probability distribution over possible actions.

Each player is assumed to be a rational entity, who makes decisions and takes appropriate actions to lead the game to the best possible *outcome* for herself, while anticipating the strategies of other players. The preference of a player for the possible outcomes is expressed by a *utility function*, which maps every outcome of the game to a real number (*payoff*). When all the components of the game (players, actions, outcomes and payoffs) are common knowledge to all the players, this game is called of *complete information*.

A game describes all the factors that characterise a strategic interaction but says nothing about what actions are the most likely to be taken by the players. This information is provided by the *solution* of a game, which offers a precise description of how the game will be played and what the outcomes might be. The most famous solution is the *Nash Equilibrium* (NE), presented in 1951 by John F. Nash [130]. An equilibrium describes a steady state condition of the game, and the NE is the situation in which no player can improve her payoff by unilaterally switching

2 Note that there exists no convention to refer to the players with a particular gender. In this thesis, we will use the female pronouns, following the textbook by Myerson [130].

to an alternative strategy. Nash proved that every game has at least one NE in mixed strategies; however, in general, there are many. To date, there is no consensus on which NE should be considered as a solution in the case of multiple equilibria.[3]

3.2.2 *Game types and applications to cooperative systems*

CLASSICAL GAME THEORY. Classical game theory (CGT) is the most mature and well-developed branch of GT. It is based on two main assumptions: players are fully rational utility maximisers, and the identity of players remains fixed during the play. The focus of CGT is to analyse the equilibria of temporally isolated interactions between players. Figure 6 illustrates the basic classification of CGT. The central distinction is between *cooperative* and *non-cooperative* games, based on whether players act independently or form collaborative coalitions. The interest of collaborative games is on how to distribute the total profit generated by a coalition to individual participants, whereas the goal of non-cooperative games is to predict the strategic behaviour of rational players. For this reason, non-cooperative games appear to be more appropriate to reason about selfishness.

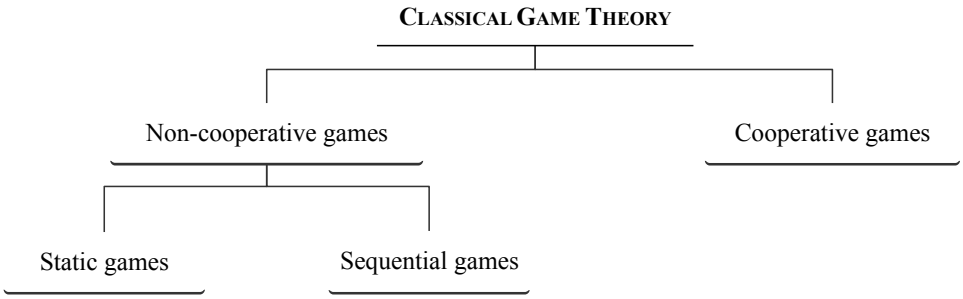


Figure 6: Classification of classical games.

We can further classify non-cooperative games into *static* (or strategic) and *sequential* (or extensive) games (see Figure 6). In static games, the players make their decisions at the beginning of the game, without knowledge of the decisions of the other players. The "Prisoner's dilemma" and the "Battle of the sexes" are two classic examples of static games (we refer the interested reader to [130] for the details). Conversely, in a sequential game, the players act in turns and can observe previous actions by the other players. Sequential games are more versatile than static games, and can model a wider range of real life situations; on the other hand, they are more complex to model and analyse [139].

Application to Cooperative Systems (Peer-to-peer). Feldman et al. [60] developed a non-cooperative static game to model selfish behaviours in P2P systems. According to this model, selfish nodes cooperate only if the current contribution cost is below a given threshold (also known as the node's generosity level). The contribution cost is assumed to be inversely proportional to the number of contributors in the system. The authors use GT reasoning to determine the

3 There is an entire literature on equilibrium selection which seeks to address reasons for players of a game to select a certain equilibrium over another.

minimum generosity level that allows the system to be stable and efficient. Also, the model is used to evaluate the effects of different incentive mechanisms (e.g., isolation, reputation) on the nodes' behaviours and system performance.

Application to Cooperative Systems (Peer-to-peer). Gupta and Somani [73] proposed a reputation-based mechanism to stimulate cooperation in P2P systems and develop a static game to evaluate its performance. The game assumes that each node is a utility maximiser that strategises about whether or not to serve requests of other nodes. Based on the analysis of the Nash equilibria, the authors demonstrate that their incentive mechanism can effectively discourage selfish nodes from avoiding contributions.

Application to Cooperative Systems (Peer-to-peer file-sharing). To investigate selfishness in P2P file-sharing, Park and van der Schaar [145] model cooperative behaviour as a three-stage sequential game: production of contents, notification of the contents available for sharing, and transfer of the requested contents. The game analysis shows that sharing is not an equilibrium strategy and that peers receive only a small amount, if any, of the requested contents. As a countermeasure, Park and van der Schaar propose two economy-based incentive mechanisms to reward sharing with monetary payments [145], which are proven effective through an equilibrium analysis.

EVOLUTIONARY GAME THEORY. Evolutionary game theory (EGT) originated as an adaptation of classical game theory to biological contexts [172]. Recently, the application of EGT has been rapidly expanding to other fields, including information systems [142, 171, 183]. One reason for the interest within the systems community is that EGT allows investigating the dynamics of large populations of players while relaxing the strict assumptions of full rationality and static identities of CGT.

An evolutionary game involves the repetition of strategic interaction between myopic players with bounded rationality, who can gradually adjust their strategy over time in response to the payoffs that they have historically received. Specifically, EGT assumes that players can learn what are the most remunerative strategies from previous interactions, and tend towards those strategies by imitation [172]. An *evolutionary stable strategy* (ESS) is a strategy such that, if adopted by (almost all) the population of players, cannot be invaded by any other strategy. In plain words, an ESS is a strategy that cannot be improved by an alternative strategy. Note that this definition is similar to that of the NE. In fact, it has been proven that each ESS is also a Nash equilibrium, but the converse is not always true [172].

To summarise, the components of an evolutionary game are: (i) a static representation of the system interactions, called a *stage game*; (ii) one or more *populations* of players; (iii) a function to calculate the utility of a given behaviour; and (iv) the *evolution dynamics* that describes the learning and imitation processes. We will discuss these components in more detail in Chapter 6.

Application to Cooperative Systems (BitTorrent). Wang et al. [171] use EGT to investigate the dynamics of incentive mechanisms based on service reciprocation, such as the one enforced in BitTorrent [45]. For this, the authors develop an evolutionary game to study the evolution of

three simple strategies: always cooperate (ALLC), always defect (ALLD), and reciprocator (R) — an R player cooperates with ALLC and R players, but not with ALLD players. Playing the R strategy incurs a cost to the player, due to the information seeking operations required to identify the strategy of the interacting players. Based on this model, Wang et al. theoretically demonstrate that when the learning capabilities of players are weak and the reciprocation cost is not negligible, players will never converge to the ALLC strategy [171].

Application to Cooperative Systems (Peer-to-peer). Zhao et al. [183] provide a general framework based on EGT to analyse the performance and robustness of reciprocity-based incentive mechanisms in P2P systems. An interesting finding emerging from their analysis is that the overall degree of cooperation in the system is limited not only by the presence of selfish nodes but also by unselective altruism. The explanation provided for this result is that selfish nodes have no incentive to cooperate with others if they receive resources in any case. The evolutionary game proposed by Zhao et al. allows controlling the trade-off between the degrees of altruism and cooperation.

MECHANISM DESIGN. Mechanism design (MD) provides a mathematical framework to design games that produce desired outcomes for the designer [139]. MD can be thought of as an *inverse* game theory, because rather than studying what are the equilibrium strategies of a given game, it specifies how the played strategies should map to the desired outcome.

As the potentials of MD as a framework to design cooperative systems have become apparent, researchers have started to take the computational complexity of mechanisms into consideration. The first theoretical model that combined mechanism design with computation tractability is due to Nisan and Ronen [138] and is known as *algorithmic mechanism design* (AMD). AMD assumes centralised decision making, whereby players report their private information to a trusted centre that regulates the players' participation. Feigenbaum and Shenker [57] relaxed this assumption and extended AMD to *distributed algorithmic mechanism design* (DAMD), in which the mechanisms is carried out in a fully distributed fashion.

A notable example of the MD approach to distributed systems was proposed by Aiyer et al. [15] under the name of BAR Model — Byzantine, Altruistic, Rational (BAR). The BAR Model provides an abstract architecture for designing cooperative systems while taking into consideration both rational behaviours and Byzantine failures. The design process leads to a so-called BAR-tolerant system, which includes both fault-tolerance and cooperation enforcement mechanisms to achieve provable resilience to protocol deviations. In practice, the BAR Model design methodology consists in making every step of a distributed protocol a Nash Equilibrium, such that it is in the best interest of rational nodes to follow the cooperative behaviour.

Application to Cooperative Systems (Peer-to-peer live streaming). In Section 2.4 we have already presented BAR Gossip, a P2P live streaming protocol designed by Li et al. [111] using the BAR Model. In BAR Gossip, peers help disseminate the video stream using gossip-based protocols. The dissemination process is designed so as to ensure a fair exchange of data and to make selfish behaviours detectable. Once detected, a misbehaving peer (rational or Byzantine) is evicted from the system. Li et al. illustrate the effectiveness of BAR Gossip

through experiments and simulations [111], proving the BAR-tolerance of the protocol even in the presence of significant collusion (up to 40% of colluders in the system). However, for the sake of tractability, the authors assume that peers are risk-averse (i.e., they never deviate from the protocol if there is any risk of being evicted from the system) and remain in the system for very long periods of time. Moreover, the fair exchange of data may require peers to waste some network bandwidth to balance bandwidth consumption [112].

3.2.3 *Discussion and open issues*

Game theory has been extremely successful in its application to cooperative systems, as evidenced by the ever-increasing body of literature devoted to this subject. However, several critiques have been made on some of its assumptions and applicability. In the remainder of this section, we present two issues in GT that are most related to our work.

ACCURACY VS TRACTABILITY. For a tractable analysis of complex systems, an analytical approach requires a high level of abstraction of the system implementation. In this regard, game theory is no exception [30, 118]. GT has been designed for general applicability to conflict situations and not to capture the particularities of a given cooperative system. Deciding what particularities might be unimportant from a strategic perspective, and thus suitable for abstraction, is not straightforward. In fact, some implementation details might have significant effects on system design as well as on the players' decision making. For instance, to the best of our knowledge, there is no game-theoretic model of BitTorrent (e.g., [93, 149]) that takes into consideration the *rarest-first* selection policy for downloading file pieces. Nevertheless, this policy has been gamed by the selfish client BitThief to increase download speed without cooperating [116].

Another issue related to abstraction was raised by Rahman et al. [150], who demonstrated that choosing different abstractions for modelling the same system may lead to equally valid but contradictory results. As an example, the authors presented an equilibrium analysis of two game models of BitTorrent based on different abstractions, and they showed that the cooperative behaviour was a Nash equilibrium only for one of the two models. The key point made by Rahman et al. is that the results obtained from a game-theoretic analysis should not be taken as given, but they always need to be examined in more detail.

The trade-off between accuracy and tractability in GT applications is apparent not only when modelling complex systems as games, but also the other way round when applying insights from GT analysis in a workable system implementation. Real cooperative systems have properties that game theory fails to accommodate, such as the availability of cheap to change identities in P2P (instead, classical GT assumes fixed and unforgeable identities [118]) or the difficulty of measuring another's node capacities or behaviour (conversely, GT often assumes complete information). For more practical examples, we refer the reader to the publication of Mahajan et al. [118], describing the issues encountered by the authors in designing two cooperative protocols, namely, a packet forwarding protocol for wireless networks and a routing protocol for Internet Service Providers. The authors also commented on the costs of implementing those

solutions (e.g., communication and computational overhead, required infrastructures), which may not be worth the potential gain for the system [118].

USABILITY. As for any analytical approach, applying GT analysis to real systems tends to be complex [30, 118, 150]. The system designer needs to create a mathematical model of the system (the game), including the alternative strategies available to the nodes (the players) and their preferences over the possible outcomes of the system. Then, the designer has to use game-theoretic arguments to assess what strategy is the most likely to be played by the players, with respect to a utility function to be formalised as well as to a solution concept to be chosen (e.g., Nash equilibrium). Carrying out this process is inherently difficult for complex systems, especially because it is manual, time-consuming and error-prone [118].

Likewise, using MD for designing real systems presents the same limitations mentioned above for analysis [15, 26, 27, 111, 118, 150]. Moreover, the resulting design solution suffers from poor maintainability and flexibility: every change in the system parameters requires a full revision of the design, which hinders the reuse of a successful solution in other systems.

In conclusion, game theory should not be considered as an off-the-shelf solution to deal with selfish behaviours in cooperative systems. Still, its prescriptive analysis provides system designers with a better understanding of the causes and impact of selfishness, and could be a source of interesting ideas for addressing this issue in practice.

3.3 EVALUATION OF THE APPROACHES TO SELFISHNESS ANALYSIS

In the previous section, we presented state-of-the-art approaches and corresponding tools to analyse selfishness in cooperative systems. We provide hereafter a comparative evaluation of some of these approaches, as an effort to gain a better understanding of their characteristics and limitations. With such an understanding, we shall be better prepared to meet the research challenge (A) of this thesis.

3.3.1 *Evaluation methodology*

SELECTION CRITERIA. The analysis approaches considered for the comparative evaluation are selected based on the following criteria:

Relevance. The approaches should be appropriate for the analysis of selfishness in cooperative systems. Consequently, it should not be surprising that all the approaches used to study selfish behaviours in the papers reviewed in the previous chapter (see Table 2 for their complete list) have been selected. Due to the relevance criterion, we do not consider benchmarking approaches in our evaluation. Indeed, benchmark suites are usually tailored to a specific application domain as well as to a specific model of application behaviour [74]. Given this lack of generality, and to the best of our knowledge, we could not find any benchmarking approach that is relevant to our study.

Quality. The approaches should be of particular interest to the research community. The selection is based on knowledge acquired from recent surveys [24, 32, 74].

EVALUATION CRITERIA. The analysis approaches will be assessed according to a number of criteria grouped into three classes: *General*, *Cooperative systems*, and *Selfishness* properties. For evaluating the performance of the selected approaches on these criteria, we relied on manuals [130], similar comparative surveys [24, 74], and other available documentation (e.g., project wikis).

The *General* criteria concern properties that are considered for any analytical and experimental approach in Computer Science. The evaluation of such criteria will be made on a scale from 1 to 5, where 1 (graphically represented as "●○○○○") indicates the worst performance and 5 ("●●●●●") is the best. The list of *General* criteria is presented below:

- *Usability*, gives an indication of the ease of use of the existing tools to carry out the considered analysis approach. In particular, the evaluation considers whether the tool provides software support and if it allows for some form of automation in the analysis procedure as well as for some control over the analysis conditions.
- *Reproducibility*, is the ability of the analysis approach to yield the same results when the same methodology is applied to the same inputs. An analysis approach that ensures good reproducibility must be considered better than an approach that ensures less.
- *Abstraction*, indicates the level of detail at which the cooperative system is specified and analysed. A higher level of abstraction is easier to understand and allows for improved control and reproducibility; on the other hand, it can affect direct empirical observation and interpretation as it might introduce artefacts that do not pertain to real-world behaviours [74]. Therefore, it is important to know the level of abstraction of an analysis approach to deduce what confidence can be put into the results. For consistency with the evaluation of the other *General* criteria (i.e., the higher, the better), we will consider the inverse of abstraction — i.e., the *Refinement* level.

The *Cooperative systems* criteria are related to approaches and tools suited for the analysis of large-scale, heterogeneous cooperative systems. Specifically, we consider the following criteria:

- *Scalability*, measured as the maximum number of nodes supported by the considered analysis approach.
- *Heterogeneity*, indicates whether the considered approach supports control over the heterogeneity of nodes ("Controllable") or not ("Fixed").

Finally, the *Selfishness* class of criteria evaluates the adequacy of the analytical and experimental approaches as a means for analysing selfishness:

- *Rationality*, indicates whether the considered approach provides support for modelling (or reproducing) selfish nodes, according to the definition we provided in Chapter 2 — i.e.,

a selfish node is a node that strategically executes the most profitable behaviour, either cooperative or selfish, with the aim of maximising its expected utility (Definition 2.8).

- A set of criteria (*Defection*, *Free-ride*, *Misreport*, *Collusion*) indicating whether the considered approach supports the analysis of the impact on the system performance of the deviation types described in Section 2.3.2. Note that these criteria focus on the execution and consequences of a selfish deviation, without making any assumptions on the rationality of the nodes.

The evaluation of the *Selfishness* criteria is done in a qualitative way, and comprises "√" (the capability is supported) and "χ" (the capability is not supported) as possible values.

3.3.2 *Evaluation results*

Table 12 presents the results of the comparative evaluation of the selected approaches suitable for selfishness analysis. Several important conclusions can be drawn from these results. First and foremost, game theory is the only tool that provides comprehensive and integrated support for analysing selfishness in cooperative systems. On the contrary, as already discussed in Section 3.1.2, the existing experimental approaches can provide very limited support. Specifically, they allow to reproduce only one type of selfish deviation (i.e., defection) using fault injection techniques, but without taking the rationality of nodes into consideration.

Secondly, if the control of the analysis conditions or reproducibility of the results is the most wanted feature, then analytical approaches, as well as emulation and simulation approaches, should be considered. Particularly, game theory and simulations seem more suitable for analysing extreme-scale systems. On the other hand, if it is more important that the analysis results are not biased by a high level of abstraction, then emulation (or, even better, real experiments) should be preferred to simulation and analytical approaches.

The last observation that can be readily drawn from Table 12 is that game theory shows the worst usability. The possible reasons for this result were discussed earlier in this chapter, and are mainly due to the lack of a reliable and easy tool to simplify the game theoretic analysis — for example, by semi-automating the analysis procedure. In contrast with analytical approaches, simulation approaches promise better usability for studying selfish behaviours in cooperative systems. Significantly, simulation approaches can rely on good supporting tools, such as the ns-2 simulator [3] and, notably, the simple and highly scalable PeerSim simulator [128] along with its extension library RCourse [122].

To conclude, none of the existing approaches for analysing selfishness in cooperative systems has proved to be completely satisfactory. On the one hand, game theory is the only tool available for modelling the dynamics and understanding the impact of different types of selfish nodes on a given system; however, it delivers poor usability and can be too abstract to know how the results could apply to real-world settings. On the other hand, experimental approaches offer more practical and usable tools for analysing the behaviour of distributed systems in particular conditions, but they lack specific support for selfish behaviours. What,

Table 12: Comparative evaluation of analytical and experimental approaches for selfishness analysis in cooperative systems.

	General and cooperative systems criteria [a]					Selfishness criteria [b]				
Approaches	**Usa**	**Rep**	**Ref**	**Sc**	**He**	**Ra**	**D**	**F**	**M**	**C**
Analytical										
Game theory [130]	●○○○○	●●●●●	●○○○○	Unlimited	Controllable	✓	✓	✓	✓	✓
Experimental: real experiments										
Grid'5000 [37]	●●●○○	●●●○○	●●●●●	1000	Fixed	x	✓ [c]	x	x	x
PlanetLab [41]	●●○○○	●●●○○	●●●●●	1000	Fixed	x	✓ [c]	x	x	x
Experimental: emulation										
ModelNet [168]	●●●●○	●●●●●	●●●●○	100	Controllable	x	✓ [c]	x	x	x
Emulab [173]	●●●●○	●●●●●	●●●●○	1000	Controllable	x	✓ [c]	x	x	x
Experimental: simulation										
NS-2 [3]	●●●●○	●●●●●	●●●○○	100	Controllable	x	✓ [c]	x	x	x
OMNeT++ [4]	●●●●○	●●●●●	●●●○○	10^5	Controllable	x	✓ [c]	x	x	x
PeerSim [122, 128]	●●●●●	●●●●●	●●○○○	10^6	Controllable	x	✓ [c,d]	x	x	x

[a] Usa = usability, Rep = reproducibility, Ref = refinement (inverse of abstraction), Sc = scalability, He = heterogeneity.
[b] Ra = rationality, D = defection , F = free-ride, M = misreport, C = collusion.
[c] Implemented as faults or churn.
[d] Highly controllable using the RCourse library [122].

therefore, emerges is the need for a unifying tool that could bridge the gap between analytical and experimental approaches.

3.4 SUMMARY

In this chapter, we provided a comparative review of the state-of-the-art approaches to analyse selfish behaviours in cooperative system, which represent the related work for the research challenge (A) of this thesis.

We organised the ways that selfishness can be analysed in cooperative systems into two broad categories, namely, analytical and experimental approaches, and we presented an overview of the characteristics, advantages and limitations of representative approaches for each category. First, we showed that analytical approaches, notably game theory, provide mathematical tools for understanding the dynamics of selfish individuals in competitive situations like those underlying a cooperative system. However, we argued that applying such formal approaches to study real systems tends to be complex, time-consuming, and prone to modelling errors, due to assumptions and simplifications to make the mathematical model tractable.

Secondly, we reviewed the approaches for experimental analysis in cooperative systems (i.e., real experiments, emulation, benchmarking, and simulation). The adoption of an experimental approach can be in fact an appropriate solution to overcome the shortcomings of analytical modelling. Although some the existing tools suitable for the experimental analysis of cooperative systems have the ability to investigate simple faulty behaviours, we found that none of them explicitly support the generation and assessment of selfish behaviours.

We concluded this chapter with a comparative evaluation of the approaches herein described. The results showed that no satisfactory solution is presently available to provide a usable, reliable and comprehensive analysis of selfishness in cooperative systems. This motivates the need for the methodologies and tools proposed in this thesis.

4

DEALING WITH SELFISHNESS IN COOPERATIVE SYSTEMS

In the previous chapter, we reviewed state-of-the-art approaches for analysing the impact of selfish behaviours in cooperative systems. This chapter completes the literature review by presenting common and effective countermeasures against selfishness, grouped under the headings of *incentive mechanisms* and *accountability techniques*.

The contributions of this chapter are listed below:

- We present characteristics, advantages and limitations of different types of incentive mechanisms designed for cooperative systems.
- We discuss how accountability techniques can be used to detect and isolate selfish nodes.

Roadmap. Section 4.1 provides an overview of incentive mechanisms, including a classification and comparative review of their incentive strategies. In Section 4.2, we present the basic principles and practical issues of using accountability techniques in distributed systems. In the same section, we provide a more detailed description of an illustrative example (i.e., FullReview [53]). Finally, we conclude the chapter in Section 4.3 with a summary of the key insights.

4.1 INCENTIVE MECHANISMS

An incentive is any factor that can stimulate or inhibit certain behaviours before their actual execution. In the context of cooperative systems, an incentive should foster cooperation between nodes and discourage selfish behaviours. From an economic perspective, this corresponds to modifying the utility function of selfish nodes, so that they can expect greater benefits — or, conversely, lower costs — if they cooperate. An effective implementation of an incentive should provide convincing arguments to support such an expectation.

We define an *incentive mechanism* as the set of protocols that implement the incentive in a given system. These protocols may include, for instance, a protocol for monitoring the actions of nodes, and protocols for assigning punishments or rewards. In the literature, a wide range of incentive mechanisms has been proposed for addressing selfishness in cooperative systems. For example, some incentive mechanisms focus on detection and punishment of selfish behaviours [47, 68, 72, 111], others compensate the contribution costs and reward cooperation [45, 135, 184], and yet other mechanisms seek to reduce the risk of interacting with selfish nodes [27, 94, 114].

Based on a thorough review of relevant studies, we identify three basic components that characterise every incentive mechanism: evaluation methods, incentive schemes and enforcement

methods. The *evaluation methods* assess the cooperation level of nodes to regulate the provision of incentives. The evaluation of a node typically consists of two activities: acquiring information about its behaviour and assessing its history of contributions. Examples of evaluation methods proposed in the literature are log audits [72, 76], real-time monitoring [104, 114, 135], redundant computation [18, 20, 175], periodic inspections [47, 68, 71, 124] and verifications [165, 188]. Such variety has emerged to accommodate the particular characteristics and requirements of each cooperative system. For instance, centralised systems such as Grids or Tor can rely on the central authority to monitor the nodes' behaviour [104, 135], whereas decentralised evaluation methods (e.g., [53, 72, 76]) are more suitable for cooperative distributed systems.

The *incentive scheme* specifies the overall strategy of an incentive mechanism to foster cooperation and reduce selfishness opportunities. Possible strategies might be to remunerate nodes as compensation for their cooperation costs, to condition access to services, or to reduce the risk of interacting with selfish nodes. In general, the incentive schemes proposed in the literature can be classified into reciprocity-based and economy-based. The strategy adopted by a reciprocity-based scheme is to reward nodes in terms of quality of service based on their history of contributions, whereas an economy-based scheme aims to create an economic market in which cooperation is the tradable good. Note that reciprocity schemes are more trust-oriented, whereas economy schemes are more trade-oriented. We discuss incentive schemes in more detail later in this section.

The last component of an incentive mechanism consists of the practical methods to enforce the incentive scheme in a given system. Following the examples of scheme strategies given above, some possible enforcement methods are virtual payments to remunerate contributions [18, 188], reputation mechanisms as a basis to regulate service provision [114, 135], and blacklists to prevent interactions with selfish nodes [26, 27].

In the remainder of this section, we propose a classification of incentive mechanisms based on their scheme, along with some illustrative examples. Then, we discuss some issues and requirements of incentive mechanisms from a cooperative system designer perspective.

4.1.1 *Classification of incentive mechanisms*

We classify incentive mechanisms based on the scheme that defines their strategy. To this end, we propose a taxonomy of incentive schemes for cooperative systems, along with a clear definition of the advantages and drawbacks of each class. Our taxonomy builds on previous literature reviews, notably the works of Feldman and Chuang [59], and Haddi and Benchaïba [75]. As illustrated in Figure 7, we broadly distinguish between *reciprocity* based and *economy* based schemes.

4.1.1.1 *Reciprocity-based schemes*

In reciprocity schemes, the cooperation level of a node is related to the benefits it can receive from interaction with other nodes. In practice, every node evaluates the history of the contributions of other nodes and uses this information to decide with whom to interact as resource

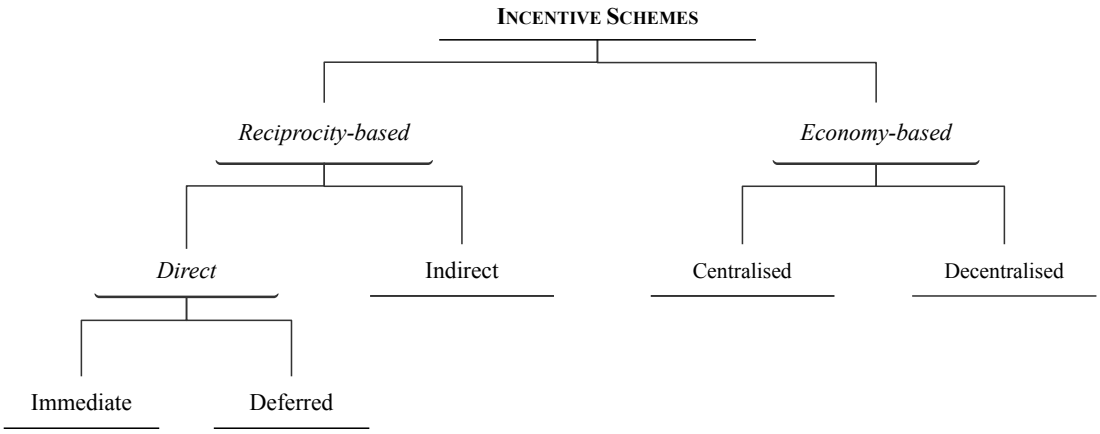


Figure 7: Taxonomy of incentive schemes for cooperative systems.

provider or requester. Reciprocity-based schemes are either *direct* or *indirect*, depending on whether the nodes' cooperation level is evaluated using only first-hand information or by aggregating information from different nodes.

DIRECT RECIPROCITY. Incentive schemes based on direct reciprocity require that every node maintains a (recent or past) history of interactions with all other nodes and use this information to influence present interactions. The actual reciprocation can occur during the present interaction or in future interactions. Based on this feature we can distinguish between *immediate* and *deferred* reciprocity schemes.

Direct reciprocation is immediate when nodes exchange resources at the same time. An application of this scheme is BAR Gossip, the BAR-tolerant live streaming protocol presented in Section 3.2.2. BAR Gossip enforces reciprocity through the *balanced exchange* method, which ensures that every peer will forward a video chunk to its partner only if it can receive in return a chunk that it had not played out yet [111]. Immediate direct reciprocity has many advantages, among which are the following: (i) it is fully decentralised and does not require any third party to operate; (ii) it is lightweight, because nodes do not have to maintain a history of past contributions but they focus only on the present interaction; (iii) it does not require establishing long duration relationships, as the reciprocation opportunity occurs on the fly; (iv) it is privacy-preserving, because nodes can reciprocate without taking the identity of the other node into consideration; (v) it enables real-time detection and handling of selfish behaviours.

Despite the numerous advantages, applying immediate direct reciprocity schemes to cooperative systems is quite problematic, due to the temporal constraint and the high heterogeneity of nodes. In fact, the probability that two nodes have immediate and mutual interest for the currently owned resources — a condition known in economics as "double coincidence of wants" — is usually very low. This may result in low efficiency and resource waste [59, 112].

A deferred direct scheme tackles the problems of the double coincidence of wants by allowing nodes to reciprocate in the future a contribution received in the past. BitTorrent is a prominent example of this class of scheme [45]. Particularly, peers in BitTorrent enforce a tit-for-tat

strategy, whereby they prioritise the download requests of other nodes based on their history of cooperation. The basic requirement of a deferred reciprocity scheme is that nodes maintain contribution information history for every node with which they have interacted. However, this introduces the necessity of strong and persistent identification of nodes, and might impose a non-negligible overhead to update and store such information. Moreover, deferred direct schemes require nodes to establish long duration relationships to ensure adequate opportunities for reciprocation. BitTorrent partly solves the last issue by restricting the number of peers with which each peer can interact, thereby increasing their interaction rate [139].

INDIRECT RECIPROCITY. Indirect reciprocity schemes allow nodes to contribute services (or resources) to a set of nodes at a given time, and receive reciprocation from a different set of nodes at a different time, even if they have never interacted before. The strategy adopted by these schemes is to make the interactions between two nodes depend not only on the past interactions between them but also on the interactions between them and other nodes in the system. Indirect reciprocity appears to be practical and scalable, which makes it particularly suitable for large and dynamic systems like cooperative systems.

The most common method to enforce indirect reciprocity in cooperative systems is using a *reputation* mechanism. The reputation of a node is derived from its past behaviour and represents a good indicator of its cooperation probability in future interactions. Specifically, a node (the *trustor*) relies on the reputation of another node (the *trustee*) to decide whether to interact with it or not. The reputation value of the trustee is the mathematical representation of its reputation, which is obtained from aggregating the recommendations (*feedback*) of other nodes called the *recommenders*. In a decentralised reputation mechanism (e.g., EigentTrust [94]), nodes play interchangeably the roles of the trustor, trustee and recommender. Conversely, in *global* mechanisms, the reputation values are calculated by special nodes based on the opinions of all the system participants [102, 121].

Feedback is a rating on a single interaction experience or formed by aggregation of several experiences. Ratings can be positive, negative or neutral, and can be expressed in several formats, such as binary (cooperative, non-cooperative), textual and numeric (discrete or continuous values) formats.

Remark. *Allowing only positive or only negative ratings may result in unfair and biased reputation values. For example, when only positive ratings are considered, a selfish but resource-rich node might have a higher reputation than a cooperative but resource-poor node, because the former has more opportunities to receive positive rating without incurring the risk of receiving negative ones. On the other hand, allowing only negative ratings can be unfair for cooperative nodes, as these nodes would have the same reputation value of newcomers with no history of contributions.*

In mechanisms like OCEAN [22] and Scrivener [133], trustors assess the reputation of a trustee based only on their direct experiences of interaction. On the one hand, this strategy leads to trustful and consistent assessment, as the process is carried out by the same node that will use the resulting reputation values. On the other hand, relying exclusively on feedback

about first-hand experiences does not scale well in the size of the system [83], because the probability to interact with nodes with a known reputation would be too small. By contrast, indirect reputation mechanisms such as EigenTrust [94], PowerTrust [185] and GossipTrust [186] allow trustors to collect opinions from other recommenders, thereby achieving higher scalability and offering more timely recommendations to assess a node's reputation. These features are particularly suitable for large-scale and highly dynamic cooperative systems [59]. However, indirect reputation approaches may also introduce substantial communication and computational costs on the nodes to disseminate and aggregate recommendations. Moreover, selfish or malicious recommenders can issue untruthful opinions, thus compromising the reliability of the reputation values [83, 102].

Once the reputation value of a node is available to a trustor, the trustor uses this information to decide how to interact with it. Koutroli and Tsalgatidou [102] suggest the following strategies: (i) the trustor interacts only with trustees with a reputation value above (resp. below) a given threshold, (ii) the trustor interacts only with the trustee with the highest (resp. lowest) reputation value, (iii) the probability that the trustor will interact with a trustee is proportional to the reputation value of the latter.

The major drawback of reputation mechanisms is that they open up new opportunities for selfish nodes to misbehave, as comprehensively surveyed by Hoffman et al. [83]. The most studied misbehaviours against reputation are the dissemination of false recommendations (e.g., bad mouthing and false praising), whitewashing [60], collusion [114], and Sybil attack [55]. The common roots of these misbehaviours are that many reputation systems do not require recommenders to provide proof of interactions that can justify their feedback, along with the lack of persistent identities. Although accountability techniques (see Section 4.2) and strong identity systems (e.g., public key infrastructures) can effectively mitigate the impact of these issues [83, 110], they would add further overhead on the nodes and may conflict with the (explicit or implicit) anonymity requirements of some systems (e.g., Tor).

4.1.1.2 *Economy-based schemes*

In economy-based schemes, nodes pay for obtaining resources and are compensated for the resources they provide. Payment is a compensation for the contribution costs and is a solution to the problem of coincidence of wants observed in reciprocity-based schemes. In fact, economy-based schemes do not rely on the mutual interest of owned resources but introduce a new resource wanted by all peers, i.e., real money or virtual money (e.g., BitCoin [132] and Nuglets [33]). Money is typically issued and certified by a Trusted Authority (TA), also called *bank*. The activity of establishing a correspondence between a unit of a resource (e.g., a CPU cycle, a file, a message) and money is called *pricing*. The pricing rules and requirements can vary considerably in different cooperative systems, provided that the settled price is a good incentive for collaboration.

The goal of an economy scheme is to create an economic market in which cooperation is the tradable good. However, this introduces economic issues as well, such as price negotiations,

inflation and deflation. Moreover, money becomes a new resource that selfish and malicious nodes might seek to exploit.

We distinguish *centralised* and *decentralised* economy-based schemes, depending on whether banks are directly involved in the payment mechanism.

CENTRALISED ECONOMY SCHEMES. Centralised economy schemes strongly rely on *banks*. A bank is a Trusted Authority that holds accounts of nodes and mediates every payment. More precisely, after every interaction, the bank subtracts a fair amount of money (*credit*) from the account of the resource consumer and adds it to the account of the resource provider. If the consumer and provider have accounts with different banks, then an inter-bank transaction is needed.

Centralised economy schemes have received wide acceptance in the literature of cooperative systems [17, 29, 119, 134, 184]. For instance, Sprite [184] is a selfishness-resilient economy-based mechanism for mobile ad-hoc networks. To prevent selfish behaviours, Sprite employs dedicated bank nodes (called Credit Clearance Services) that allocate credit to a provider only if the provider can produce verifiable proofs of cooperation, i.e., signed receipts. A similar method, though computationally less demanding, is implemented by the Express system [90], which uses hash chains instead of digital signatures as the proof of cooperation. Another example of a centralised economy scheme has been proposed by Gramaglia et al. [71] for P2P backup systems, whereby nodes pay to store their backup data on others' disks and are paid for sharing their local storage space. The authors introduce payments via digital checks so that a central bank can support payments when nodes are off-line.

Centralised economy schemes are computationally cost-effective for nodes, because most of the burden to maintain dependable and secure payments is on banks. On the other hand, the deployment and operation of a bank require a costly infrastructure that might be infeasible in some environments. Moreover, as for any centralised system, banks constitute a bottleneck for performance, a single point of failure, and — even more compelling — an appealing target for malicious attackers.

DECENTRALISED ECONOMY SCHEMES. In the decentralised realisation of an economy-based scheme, the payment mechanism consists of a direct exchange of money between resource consumer and provider [33, 89, 123, 159, 170, 188]. Banks do not play any role in this exchange but are the only authority that can issue valid money (as in Nuglets [33]) or they simply disappear (as in Bitcoin [132]).

The decentralisation of the payment mechanism offers two main advantages. First, it is privacy-preserving, as it does not require disclosing the identity of the interacting nodes. The only information a resource provider needs to know is that the consumer can pay for the requested resource. Second, it allows the decentralised economy scheme to be more scalable and robust to failures than the centralised scheme because it does not rely on banks' dependability. On the downside, decentralising the monetary exchange induces a substantial increase in the overhead on the nodes.

A means for improving the robustness of decentralised economy schemes is to establish trust among trading nodes. For instance, the incentive mechanisms presented by Buttyán et al. [33, 34] use tamper-proof devices installed in each node to enforce the payment mechanism. This approach, although effective in principle, is not applicable in most cooperative systems [153], due to the impossibility of forcing system participants to install trusted hardware. A possible software-based solution to establish trust is to combine reputation and economy based schemes. As an illustrative example, the ARM system [159] enforces a pricing method that relates the price of a resource to the reputation value of the resource consumer. Hence, cooperation is more attractive because it leads to high reputation values and, consequently, lower prices to buy resources.

Table 13 summarises advantages and drawbacks of the incentive schemes discussed above.

Table 13: Advantages and drawbacks of incentive schemes for cooperative systems.

Incentive scheme	Advantages	Drawbacks
Immediate direct reciprocity	• Decentralised • Cost-effective • Suitable for short-term relationships • Anonymity preserving • Real-time detection and sanction of selfish nodes	• Limited applicability • Poorly scalable • Immediate double coincidence of wants
Deferred direct reciprocity	• Eliminates immediate double coincidence of wants	• Requires persistent and strong identities • Requires long-term relationships • Requires maintaining history of contributions
Indirect reciprocity	• Eliminates immediate double coincidence of wants • Suitable for short-term relationships • Decentralised • Highly scalable • Shows broad adaptability	• Requires persistent and strong identities • Correctness of the reputation value depends on information truthfulness • Possibly high communication and computational overhead due to reputation dissemination and assessment
Centralised economy	• Low computational overhead (for nodes) • Shows broad adaptability	• High communication overhead with banks • Scalability bounded by the banks' capacity • Requires persistent and strong identities • Requires costly infrastructure and mechanisms to enforce banks' dependability
Decentralised economy	• Decentralised • Highly scalable • Anonymity preserving • Shows broad adaptability	• High communication and computation overhead • Requires costly devices or mechanisms to enforce payments' dependability

4.1.2 *Classification of incentive mechanisms from relevant studies*

In Chapter 2, we presented a classification framework to describe selfish behaviours in cooperative systems (see Section 2.3), built on a systematic review of relevant papers selected from the literature. Table 14 summarises the characteristics of the incentive mechanisms proposed in the same studies and ordered by incentive scheme.[1]

Table 14: Characteristics of the incentive mechanisms proposed in the papers considered to build the classification framework presented in Section 2.3.

			Enforcement methods [a]							
Reference	**Evaluation methods**	**Incentive scheme**	A	Ba	Bl	E	R	S	MP	VP
Anderson et al. [18]	Redundant computation	Centralised economy	x	x	x	x	x	x	x	✓
Anta et al. [20]	Redundant computation	Centralised economy	x	x	x	x	x	x	✓	✓
Gramaglia et al. [71]	Inspection	Centralised economy	x	x	✓	x	x	x	x	✓
Yurkewych et al. [180]	Redundant computation	Centralised economy	x	x	x	x	x	x	✓	x
Shneidman and Parkes [161]	Redundant computation	Centralised economy	x	x	x	x	x	x	✓	x
Jansen et al. [89]	-	Decentralised economy	x	x	x	x	x	✓	x	✓
Sirivianos et al. [165]	Crypto. verification	Decentralised economy	x	x	x	x	x	x	✓	✓
Zhu et al. [188]	Crypto. verification	Decentralised economy	x	x	x	x	x	x	x	✓
Li et al. [111]	Reporting	Direct reciprocity	x	✓	x	✓	x	x	x	x
Buttyán et al. [36]	Reporting	Direct reciprocity	x	✓	x	x	x	x	x	x
Ben Mokhtar et al. [26]	Reporting	Indirect reciprocity	x	x	✓	✓	x	x	x	x
Ben Mokhtar et al. [27]	Broadcasting protocols	Indirect reciprocity	x	x	✓	✓	x	x	x	x
Ben Mokhtar et al. [28]	Accountability	Indirect reciprocity	x	x	x	✓	x	x	x	x
Blanc et al. [31]	TA Monitoring	Indirect reciprocity	x	x	x	x	✓	✓	x	x
Cox and Noble [47]	Inspection	Indirect reciprocity	✓	x	x	x	x	✓	x	x
Guerraoui et al. [72]	Accountability	Indirect reciprocity	x	x	x	✓	✓	x	x	x
Kwok et al. [104]	TA Monitoring	Indirect reciprocity	✓	x	x	✓	✓	x	x	x
Lian et al. [114]	TA Monitoring	Indirect reciprocity	x	x	x	x	✓	✓	x	x
Mei and Stefa [124]	Inspection	Indirect reciprocity	x	x	x	✓	x	x	x	x
Ngan et al. [135]	TA Monitoring	Indirect reciprocity	x	x	x	x	✓	✓	x	x
Piatek et al. [148]	Accountability	Indirect reciprocity	✓	x	x	x	x	✓	x	x
Yang et al. [175]	TA Monitoring	Indirect reciprocity	x	x	x	x	✓	✓	x	x

[a] A = agreements , Ba = barter, Bl = blacklists, E = eviction, R = reputation,
S = service differentiation or interruption, MP = money payments, VP = virtual payments.

As illustrated in Figure 8, more than half of the incentive mechanisms rely on the indirect reciprocity scheme. This is most likely due to the high scalability and decentralisation of the scheme, which properties well match with the particular characteristics of cooperative systems. Among the principal methods to enforce indirect reciprocity is reputation, often in combination with service differentiation and eviction techniques. For example, Ngan et al. [135] present a reputation-based service priority regime designed for Tor, whereby the traffic of a

1 The papers by Hughes et al. [87], Li et al. [113] and Locher et al. [116] are not included in Table 14 because no incentive mechanism has been proposed by the authors.

high-reputation relay has priority over the others. Also Guerraoui et al. [72] use reputation as an indicator of the cooperation level of a node, but rather than rewarding nodes with higher reputation values, they propose to evict from the system nodes with reputation below a certain threshold.

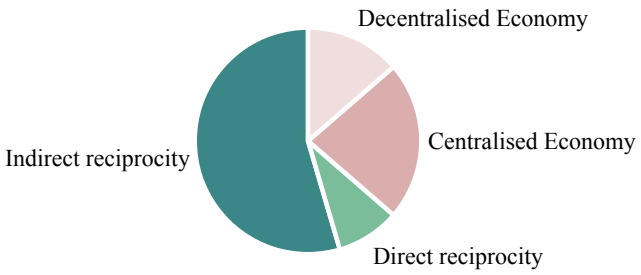


Figure 8: Overview of the incentive schemes adopted by the incentive mechanisms listed in Table 14.

If we group the studies reported in Table 14 by the category of the cooperative system that they investigate (e.g., data distribution, distributed computing), indirect reciprocity is again among the most implemented schemes. The results shown in Figure 9 indicate distributed computing systems as the only exception. The reason is that the client/server architecture of these systems makes centralised economy schemes more suitable and effective.

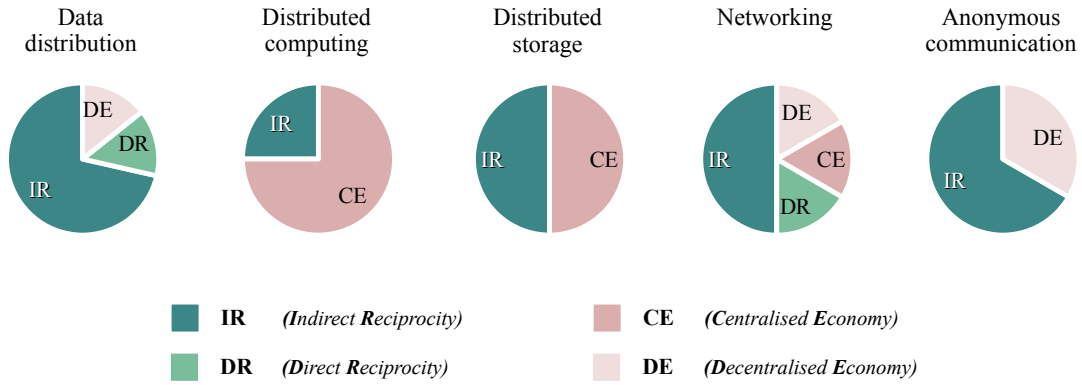


Figure 9: Overview of the incentive schemes adopted by the incentive mechanisms listed in Table 14 and grouped by cooperative system categories.

4.1.3 *Desirable requirements for incentive schemes in cooperative systems*

An incentive mechanism designed for a cooperative system needs to accommodate the particular features and requirements of these systems. In the following, we propose a list of desirable requirements that an incentive mechanism should meet.

DECENTRALISATION. Incentive mechanisms should retain the decentralised nature of cooperative systems. For example, a mechanism designed for a fully decentralised cooperative system or a delay tolerant network should not rely on any central entity or authority, as this would introduce delays, bottlenecks and overhead [153]. Similarly, economy-based schemes might not be the appropriate solution, as they require a trusted authority to issue money and, possibly, manage payments.

OVERHEAD. Enforcing an incentive scheme imposes a substantial cost on the nodes, due to the computational and communication overhead. This problem is especially critical in mobile environments, in which nodes have limited resources. The overhead imposed by an incentive mechanism should never exceed its expected benefits.

ADAPTABILITY. Cooperative systems cover a wide range of functionalities (e.g., file sharing, distributed computing), communication infrastructures (e.g., the Internet, delay tolerant networks), and many other system or application-specific properties (e.g., type of P2P overlay). General applicability and flexibility of an incentive scheme are two desirable requirements to fulfil, as they would reduce the development and configuration cost of the incentive mechanism. A downside of generality, though, is a loss in efficiency, because the scheme does not consider the proper functioning and the performance metrics of a particular system and application [75].

FAIRNESS. Nodes have different physical capacities and level of access to resources, e.g., mobile nodes may temporarily suffer from bad connectivity. Due to this heterogeneity, two undesirable situations may arise:

- Resource-poor nodes might be penalised by incentive strategies based on rewards because their contribution level is lower than that of resource-rich nodes.
- Resource-rich nodes tend to receive more resource requests than a resource-poor node, thus becoming a performance bottleneck and a point of failure.

A fair incentive scheme should take the heterogeneity of nodes into consideration, by avoiding to further penalise resource-poor nodes as well as to overload the resource-rich ones. However, meeting this requirement can be challenging for two reasons. First, distinguishing between lack of cooperation due to resource shortage and selfish behaviour is still an open problem [140]. Second, reducing the load on resource-rich nodes can lead to a substantial degradation of the system performance, particularly in those systems that rely on a small core of highly cooperative nodes [78, 87].

SELFISH RESILIENCE. An incentive scheme may introduce new motivations and opportunities for selfish nodes to misbehave [102, 140]. For example, a group of nodes may decide to collude to increase their remuneration [114], or a node participating in the enforcement methods of a mechanism may decide to limit its contribution so as to reducing the associated overhead [27].

For this reason, rather paradoxically, selfish nodes may need incentives to be convinced to participate in an incentive mechanism.

PRIVACY AND ANONYMITY. An incentive mechanism might require guaranteeing the anonymity of its participants for many reasons, such as (i) to protect the privacy of the interacting nodes, (ii) to avoid retaliation from sanctioned nodes, (iii) to hide sensitive information that could be exploited by malicious nodes (e.g., account credentials in centralised economy schemes), (iv) to meet an application-specific requirement (e.g., anonymous communication systems), and (v) to save costs and time to implement an identity infrastructure. Regardless of the reason, there are different enforcement methods to enforce anonymity, ranging from hiding real-world identity behind a pseudonym, to using state-of-the-art cryptographic mechanisms [79, 102, 157]. Note that privacy and anonymity requirements influence the choice of a given incentive scheme.

Remark. *Meeting the requirements of privacy and anonymity in incentive mechanisms for cooperative systems is out of the scope of this thesis. We refer the interested reader to the references contained in the papers cited above, including the two surveys on which we based our taxonomy of incentive schemes [59, 75].*

4.1.4 *Perspectives on the research challenges*

The review of the literature on incentive mechanisms presented above provides useful insights to address the design challenges (D) of this thesis, and especially the research challenge (D.1) — i.e., identifying *general* and *practical* mechanisms to enforce cooperation in cooperative systems.

Concerning generality, indirect reciprocity, as well as economy-based schemes, offer broad adaptability to various application domains. From our survey, the indirect reciprocity scheme appears to be the most adopted incentive in four out of the five categories of cooperative systems investigated (see Figure 9). Notably among these categories is data distribution, which includes the most popular applications of cooperative systems (e.g., P2P file-sharing, P2P live-streaming).

Indirect reciprocity can also meet the second requirement posed by the research challenge (D.1), which is to provide a cooperation enforcement mechanism to use in practice in cooperative systems. Such a mechanism is represented by reputation systems. In fact, reputation systems can well fit with the decentralised nature of cooperative systems [94, 185, 186], and are particularly suitable for large-scale and highly dynamic environments [59]. On the contrary, the other incentive schemes discussed above cannot be considered completely satisfactory. For instance, as can be seen in Table 13, direct reciprocity schemes either show poor scalability (immediate direct reciprocity) or are inadequate in highly dynamic systems (deferred direct reciprocity). The central role of banks in economy-based incentives (either as transaction brokers or trusted issuers of valid currency), not only poses some practical limitations to the scalability of these schemes [153], but also fails to meet the desired requirement of decentralisation.

Although reputation systems have good potential for meeting the research challenge (D.1), we must be aware of the limitations of such mechanisms and provide appropriate solutions. The first two limitations listed for indirect reciprocity schemes in Table 13 concern their robustness to (i) attacks against the authentication system, such as whitewashing and Sybil attacks, and (ii) the dissemination of false information by single nodes or groups of colluders. The accountability techniques described in the next section have been advocated as a viable solution for preventing these and other attacks [178].

The last limitation of reputation systems listed in Table 13 is related to the communication and computational overhead imposed by the incentive mechanism, which might greatly reduce the system performance. Solving the trade-off between system performance and the effectiveness of the reputation scheme poses a difficult configuration problem, which is often a trial-and-error and time-consuming procedure. This motivates the need for an approach that facilitates the set-up and configuration of incentive mechanisms in a given cooperative system, which represents research challenge (D.2) of this thesis.

4.2 ACCOUNTABILITY IN DISTRIBUTED SYSTEMS

Accountability is the ability to hold individuals responsible for their actions [105, 177]. In distributed systems, accountability techniques have been used to detect and isolate misbehaviour, as well as to assign non-repudiable responsibility for faulty actions and system failures [76, 126, 179]. Yumerefendi and Chase [178] promote accountability as a "first-class design principle" for building dependable and trustworthy systems on top of untrusted environments, such as those of cooperative systems. In fact, on the one side, traditional dependability techniques (e.g., fault prevention, fault tolerance, fault removal) can identify errors and misbehaviours but fail to point to the responsible node. On the other side, traditional security approaches (e.g., authentication, authorization) can protect the system against misbehaviours pursued by external actors, but are not equally effective in preventing misbehaviours of nodes within the security perimeter.

An accountability system provides practical means to meet all the above challenges. Also, as demonstrated in several studies [53, 72, 76, 77, 178], it can detect and isolate selfish behaviours in cooperative systems.

In the remainder of this section, we present the general characteristics of accountability techniques for distributed systems, along with a more detailed presentation of the solution provided by Diarra et al. [53], FullReview. Finally, we discuss some issues of interest to system designers.

4.2.1 *Basic concepts*

Accountability systems require each node to maintain a *secure log* to record its interactions with other nodes [179]. Each node is further associated with a set of *witness* nodes, which periodically check whether the log entries correspond to a correct execution of the underlying protocol. If

any deviation is detected, then the witnesses build a proof of misbehaviour that can be verified by any correct node, and punishment is inflicted on the misbehaving one. Fig 10 provides a graphic representation of this approach, in which a node "Alice" logs the interactions with its partners and is audited by a set of witnesses.

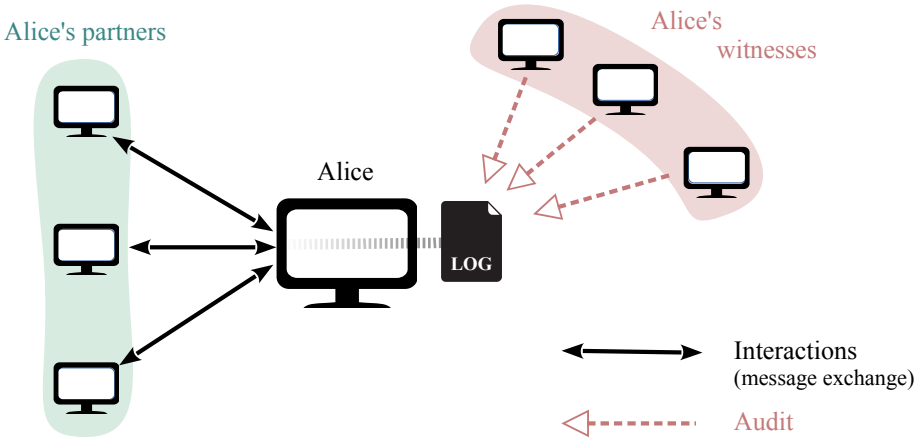


Figure 10: Overview of an accountability system.

More in detail, an accountability system relies on the following building blocks.

Secure hashes and digital signatures. They create binding and non-repudiable evidence of nodes' behaviour. In practice, each node is assigned at least one asymmetric key pair. Keys can be distributed, for example, by a Public Key Infrastructure (PKI) or decentralised protocols. When nodes communicate, they sign each request or response with their private key. Requests and responses are generically referred as *actions*.

Secure logs. Secure logs retain all the signed actions, including the ones received from other nodes, along with other relevant information. Roughly speaking, a log entry should indicate what node performed what action on what resource, with what access level and at what time. Logs are the primary source of information for an accountability system; thereby their reliability must be guaranteed. Secure logs [76, 158] provide a tamper-evident authenticated data structure to store signed actions, so that any attempt to corrupt the log is detectable. More in detail, each log entry e_w is associated with (i) a recursively defined hash value h_w, computed as a hash of e_w concatenated with the value of h_{w-1}, and (ii) an *authenticator* α_w^i, which is a statement signed by node i using its private key to claim that its log entry e_w has hash value h_w. The resulting hash chain, along with the generated authenticators, allows verifying that a node has not been tampered with. For example, let the node i send a pair of authenticators α_0^i and α_w^i (corresponding to the entries e_0 and e_w of its log) to one of its witnesses, denoted as j. Then, j can ask i for its log entries $e_0, e_1, ...e_w$ and recompute $h_0, h_1, ..., h_w$; if the resulting h_w does not match the hash value in α_w^i, the witness has verifiable evidence that i has tampered with its log. Moreover, j can use the signed authenticators α_0^i and α_w^i along with the log entries sent by i to convince any other correct node of i's misbehaviour.

Auditing interfaces. They allow to access nodes' secure logs and provide cryptographic proofs of correctness of their behaviour. In practice, during an audit, the witness verifies that (a sequence of) the signed actions of a node comply with the expected behaviour. This can be done in several ways, for instance by using state machine replication [40, 53, 76], or by checking if the log satisfies a set of invariants that must hold in any correct execution of the system (e.g., a node can serve only content that it has previously received).

Depending on the result of the audit, a witness can classify nodes into three categories: (i) *exposed*, if there is evidence of misbehaviour, (ii) *suspected*, if there is evidence that the node is ignoring requests of interaction, and (iii) *trusted*, in all other cases. An accurate accountability system must guarantee that no cooperative node is ever exposed, nor that a cooperative node is suspected for a long time [76].

Remark. *Accountability techniques can be either entirely based on software solutions (e.g., PeerReview [76], FullReview [53] and AVM [77]) or may require specialised trusted hardware (e.g., TrInc [109] and A2M [42]). Using hardware-based solutions has some advantages; for example, they can significantly reduce the computational cost of maintaining secure logs, making them tamper-evident by construction. However, these techniques are not suitable for the problem domain of this thesis. In fact, the large scale and heterogeneity of cooperative systems, which are often distributed across multiple administrative domains, make the use of trusted hardware unfeasible. Therefore, in this thesis, we focus only on purely software-based accountability techniques.*

4.2.2 *Related work: FullReview*

FullReview is a general software solution for enforcing accountability in a distributed system [53]. Of particular interest for our work, the protocols used by FullReview are designed to be resilient to selfish behaviours, embedding convincing incentives to force selfish nodes to participate in the accountability process. To this end, FullReview makes some assumptions about selfish nodes: they are risk averse,[2] they do not collude, and they remain in the system for a long time.

Concretely, FullReview applies to a set of nodes executing a reference implementation P of the system.[3] Each protocol is defined as a deterministic finite state machine [84]. The FullReview protocols F are also defined as deterministic state machines. As already introduced in Section 4.2.1, each node $\mathtt{i}$ maintains a secure log of the messages it exchanges with other nodes. To sign messages, $\mathtt{i}$ holds a pair of public/private keys bound to a unique node identifier. The correctness of $\mathtt{i}$ is checked periodically by its witness nodes, by replaying $\mathtt{i}$'s log and comparing the outputs with a reference implementation of P. If a witness finds discrepancies, then it *exposes* $\mathtt{i}$ as faulty and generates a verifiable proof of misbehaviour that it makes available to the rest of the system. In FullReview, exposed nodes are directly evicted from the system.

2 If a risk averse node estimates the probability to be detected for a deviation be greater than zero, then it will stick to the protocol.

3 The reference implementation of the system is not the formal specification of the whole system, but it concerns only the subset of functionalities (protocols) to monitor.

Hereafter, we present the five protocols executed by FullReview to enforce accountability in the presence of selfish nodes.

Commitment protocol. This protocol ensures that the sender and the receiver of a message have provable evidence that the other party has logged the exchange. Note that the message exchange may take place during the execution of both P and F. Consider, for instance, that node i wants to send a message m to node j. First, i adds a new entry e_w to its secure log. Then, i generates an authenticator α_w^i to prove it has logged the action, and sends the authenticator to node j along with m. When j receives the message, it checks the attached authenticator, logs the reception of m in a new entry e_z, and generates the corresponding authenticator α_z^j. Finally, j sends its authenticator to i along with a message to ack the reception of m to its original sender.

Consistency protocol. This protocol checks that each node i maintains a single linear log, which has to be consistent with all the authenticators it had issued. In practice, every time a node i receives an authenticator from another node j, i forwards the authenticator to the witnesses of j. Periodically, each of these witnesses picks the authenticators with the lowest and the highest sequence number, and challenges node j to return all log entries in this range; if node j is not correct, then it is exposed as faulty. Moreover, the witnesses use the log entries of j to extract all the authenticators j has received from other nodes. Then, they propagate such authenticators to the corresponding witness sets, to prevent collusion between a witness and its monitored node.

Audit protocol. This protocol specifies the steps that a witness w periodically undertakes to verify whether the actions of a node i are consistent with the reference implementations of P and F. To avoid that a selfish witness would be tempted not to audit in order to save local resources, the audit protocol is started by i, which proactively asks its witnesses for an inspection. On the other hand, i wants to be inspected, because a successful audit releases the *Certificate of Correctness* (CoC) that i needs to communicate with the other nodes. In contrast, if i fails the audit, its witnesses issue and disseminate a *Proof of Misbehaviour* (PoM) of i, thus, exposing i and leading i to an eviction. Finally, to ensure that selfish witnesses do not lie about the outcome of an audit (e.g., claiming that the monitored is correct without performing the verification, in order to save CPU), FullReview requires i to forward all the audit results (either CoCs or PoMs, or both) to the witness set of each of its witnesses, asking them to replicate the audit. If the majority of witnesses of a given witness w of i obtains a different audit result than w, then w is exposed and evicted from the system.

Challenge/Response protocol. This protocol ensures that if a node does not answer to a message, then, instead of being directly evicted from the system, it is *suspected* by the interacting node. This is to avoid wrongful evictions of correct nodes operating under bad network conditions.[4] However, being suspected is not good for nodes, because correct nodes do not communicate with suspected nodes. To get *trusted* again, a correct but suspected node must correctly execute the challenge/response protocol of FullReview. There are two types of chal-

4 Message loss is commonplace in applications deployed over the Internet, which is the case for most cooperative systems.

lenges defined in FullReview. First, an *audit challenge* is generated by a witness w when a node i does not reply to its log request; i stops being suspected by w when it provides the requested log entries. Secondly, a *send challenge* is triggered when a node i does not acknowledge the reception of a message during the commitment protocol; also, in this case, i stops being suspected when it provides its witnesses with the expected ack.

Evidence transfer protocol. This protocol ensures that faulty nodes are eventually *exposed* by all correct nodes in the system. To this end, every correct node i periodically fetches and replays the challenges collected by the witnesses of a node that i is interested in, for example, an interacting partner.

We refer to the technical report of the authors for an exhaustive summary of each incentive designed to discourage selfish deviations from the FullReview protocols [52, 53].

4.2.3 *Discussion and open issues*

Accountability techniques have been successfully applied to detect and punish selfishness in cooperative systems, e.g., in P2P live streaming [53, 72, 76] and anonymous communication systems [53]. On the downside, enforcing accountability incurs a non-negligible cost on the system, which poses a difficult configuration problem for the system designer. We discuss cost and configuration issues in turn below.

COST. The benefits of using accountability techniques come at the expense of a greater cost to the system, mainly due to the high message overhead and the intensive use of cryptography. Costs can be decomposed into three dimensions, namely, computation, communication and storage. Table 15 summarises the activities and parameters that have a larger impact on each cost dimension. For example, in FullReview [53], the computational overhead grows linearly with the number of witnesses assigned to each node and with the frequency of the audits. Moreover, the communication and storage costs imposed by FullReview, measured in terms of network traffic overhead and size of the secure log, grow with the square of the number of nodes in the system.

Cost dimension	Influencing activity	Influencing parameter
Computation	• Cryptographic operations • Auditing	• Key length • Hash function • Frequency of audit • Size of the log portion to verify
Communication	• Log transmission • Evidence dissemination	• Number of witnesses • Frequency of audit • Size of the log portion to verify
Storage	• Logging	• Log size

Table 15: Activities and parameters that influence the cost of enforcing accountability.

CONFIGURATION. The fine tuning of the accountability mechanisms is a necessary but challenging task for building a dependable and high-performance cooperative system. Configuring accountability mechanisms requires that a system designer select values for a number of parameters that directly affect the system performance (see Table 15). Particularly, among the possible parameters, we note the following:

- The `number of witnesses` associated with each node. More witnesses imply more computation and communication overhead.
- The `audit period` between two log audits performed by the witnesses of a node. A short audit period allows for rapid faults detection, but it also increases the computational and communication overhead, due to the greater frequency of log requests and audits.
- The `severity of the punishment` of a faulty node in the case of a successful audit. Depending on the severity, punishments could vary from the eviction of the faulty node to the reduction of its reputation value — if the accountability mechanism is coupled with a reputation management system. A low severity of punishment may increase the number of selfish deviations; whereas high severity may lead to the wrongful eviction of a correct node, for example, if the network is not reliable (e.g., in a mobile environment), or if the node gets suddenly disconnected (e.g., characterised by churn in P2P systems).

In the literature [53, 72, 76, 178], no indication is provided for the setting of these parameters, leaving the designer with a critical and complex trade-off to deal with: on the one hand, achieving the desired resilience to misbehaviours without wrongly penalising correct nodes, and on the other hand, imposing minimal costs. Resolving these conflicting requirements requires the systematic evaluation of a large number of experiments, to investigate the impact of the value of each parameter on the system performance. Moreover, such experiments require the ability to inject and to automatically reason about selfish behaviours, which is not supported by state-of-the-art experimental environments, such as Splay [108], PlanetLab [41], NS-3, ProtoPeer, and PeerSim [24]. Overall, in the absence of convenient tools that support system designers with the configuration of an accountability mechanism, its tuning is a time-consuming trial-and-error procedure.

To highlight the trade-offs that a system designer has to take into account when setting up the parameters of an accountability mechanism, we performed an experiment, involving a gossip-based live streaming protocol monitored by the selfishness-resilient accountability mechanism FullReview [53]. The live streaming protocol is inspired to the one described by Guerraoui et al. [72]. In this protocol, a source node disseminates a set of video chunks to a subset of nodes over an unreliable network. Periodically, each node sends the video chunks it received to a set of randomly chosen partners and asks them for any video chunks it is missing. In this scenario, we consider selfish nodes that always (i) free-ride the chunk-exchange protocol by sending fewer video chunks than what was requested by the other party, and (ii) under-report their chunk availability to reduce the probability of receiving chunk requests. The above deviations have been described in Tables 3 and 4 using the classification framework for selfish behaviours we presented in Chapter 2.

FullReview applies to each node i in the system, requiring i to log all its activities, and assigning it to a set of witnesses that periodically verify whether i is correctly executing the live streaming protocol. If any deviation is detected, i's witnesses inflict a punishment of a given severity on i. In our experiment, we assume that a system designer aims to create a selfishness-resilient live streaming protocol such that:

1. The video received by correct nodes maintains good quality despite the presence of up to 50% selfish nodes. Traverso et al. [166] set the quality threshold to 3% of jitter, above which the quality of experience is significantly impaired.

2. Correct nodes are not wrongfully expelled from the system, even if the network suffers from up to 5% of message loss.

3. The average bandwidth consumption per node including both the video stream and FullReview does not exceed 1 Mbps.

To reach this objective, we start with the FullReview default configuration (e.g., with an audit period being 10 seconds) and vary the severity of punishments inflicted on nodes by the accountability mechanism.

Figure 11 shows the percentage of correct nodes wrongly evicted by FullReview and the percentage of selfish deviations observed in the system for various values (in our experiment less than 10% selfish deviations for the selfish nodes translates into an experienced jitter lower than 3%). As expected, the results show a clear increase in the percentage of correct nodes wrongly evicted from the system. Nevertheless, the punishment values 1 and 1.5 satisfy the first two requirements set by the designer.

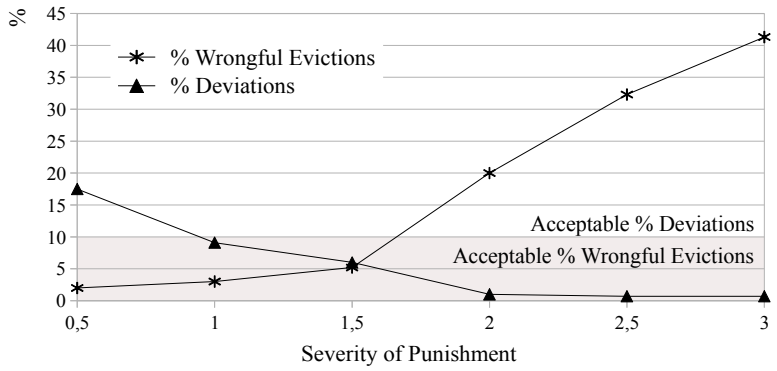


Figure 11: Impact of the punishment values. The gray box indicates the acceptable percentage of deviations and of wrongful evictions (up to 10%).

We thus go further and measure the communication overhead incurred in the system for 1.5, which has the lowest % of deviations, while varying the FullReview audit period, as this parameter highly impacts the communication overhead. The results, depicted in Figure 12, show that, on the one hand, increasing the audit period decreases the overhead, because logs are requested and audits are made less often by monitors; on the other hand, the longer the audit period, the

slower faults are deterred, thereby increasing the percentage of selfish deviations. However, none of the tested values achieves a bandwidth consumption that meets the third requirement set by the designer. The designer has thus to continue the manual calibration process by testing other pairs of parameter values and running further experiments.

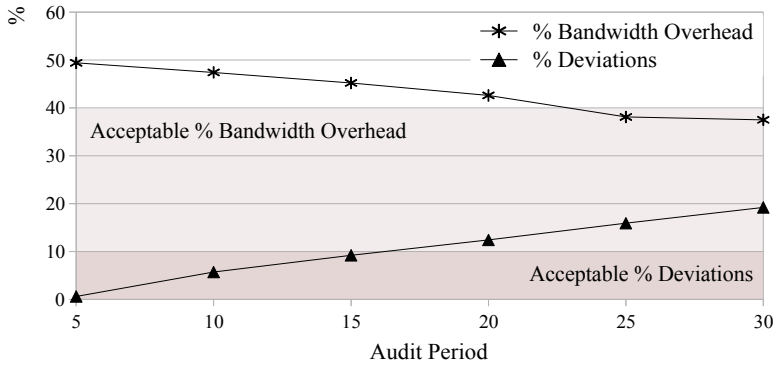


Figure 12: Impact of the audit period. The light gray box indicates the acceptable percentage of deviations (up to 10%), whereas the dark gray box shows the acceptable percentage of overhead (up to 40%).

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

4.3 SUMMARY

This chapter concludes our survey on selfishness and its countermeasures in cooperative systems, which constitutes the first contribution of this thesis (C.1). More particularly, in this chapter, we addressed the research challenge (D.1) by reviewing two groups of state-of-the-art countermeasures against selfish nodes, namely, incentive mechanisms and accountability.

We started by presenting objectives, characteristics, and limitations of incentive mechanisms. To this end, we proposed a classification framework based on a taxonomy of incentive schemes. According to this framework, we could categorise incentive mechanisms into reciprocity-based and economy-based mechanisms. We showed that reciprocity-based mechanisms were more suitable for large scale and heterogeneous systems, but require that nodes have mutual interest for owned resources. In contrast, economy-based solutions overcome the problem of mutual interest by introducing money as universal compensation for any contribution cost, but rely on trusted authorities (banks) and introduce economic issues in the system, such as price negotiation, inflation and deflation. We concluded the first part of this chapter by advocating reputation systems as a candidate solution to address and overcome the design challenges (D) set by this thesis. However, we showed that reputation might also introduce new deviation opportunities for selfish nodes, such as the dissemination of false information and collusion.

In the second part of this chapter, we argued that accountability techniques can be the perfect complement to reputation systems. In fact, on the one hand, enforcing accountability into a reputation system can improve its robustness and reliability; on the other hand, sanction schemes for accountability mechanisms can rely on the flexibility of reputation to keep a fair balance between severity and tolerance. We organised the overview of accountability techniques by, first, introducing basic concepts, such as audits and secure logs. Then, we presented the Full-Review system and described in greater detail its selfishness-resilient protocols for enforcing accountability in distributed systems. Finally, we identified and discussed the main challenges for enforcing accountability in cooperative systems, which are the additional (computational, communication and storage) overhead on the nodes, and the difficulty in finding a satisfactory trade-off between accountability and performance without any tool support.

Supporting the design of selfish-resilient cooperative systems while solving the trade-offs between system performance and the effectiveness of a cooperation enforcement mechanism will be the focus and main contribution of the next part of the thesis.

Part II

SELFISHNESS-AWARE DESIGN OF COOPERATIVE SYSTEMS: THE RACOON FRAMEWORK

5

THE RACOON FRAMEWORK

In Chapter 2, we described many and varied examples of selfish behaviours in cooperative systems. Then, in Chapter 3 and Chapter 4, we discussed different approaches to analyse and counter such behaviours, showing their advantages and drawbacks. Based on the findings described in these chapters, and taking a system design perspective, we can draw three conclusions. First, in real cooperative systems, it is not realistic to assume that all nodes behave as expected. Thus, a defensive design approach should be adopted, bearing the selfishness of nodes in mind. Second, existing tools for addressing selfishness (e.g., game theory, incentive mechanisms, accountability systems) introduce new design decisions themselves, in the form of trade-offs between selfish-resilience and efficiency of the system, as well as between generality and applicability of the mechanism to enforce cooperation. We also showed that finding the right balance in these trade-offs is not trivial, and usually requires extensive manual effort. Finally and consequently to the previous conclusions, designing cooperative systems resilient to selfish nodes is a challenging task.

To facilitate the work of designers of cooperative systems, we propose in this chapter a unified framework, *RACOON*, that addresses the design challenges stated above. *RACOON* provides a semi-automatic and general methodology, along with its software implementation, for designing, tuning and evaluating cooperative systems resilient to selfish nodes. The *RACOON* framework adopts a model-based approach, in which models direct the design of a cooperative system. To begin, the system designer (hereafter referred as "Designer") provides the functional specification of the system (i.e., communication protocols) and a set of performance objectives. *RACOON* uses these models to support and extensively automate the following activities: (i) enforcement of practical mechanisms to foster cooperation (i.e., accountability and reputation mechanisms for distributed systems), (ii) development of a behavioural model of selfish nodes to predict their strategic choices, (iii) tuning of the accountability and reputation mechanisms so as to meet the Designer's objectives, for which we propose a novel combination of simulations and game theory. *RACOON* results in a redesign of the system specification, which includes finely tuned mechanisms to meet selfish-resilience and performance objectives. This output serves as a reference to developers for the eventual implementation phase of the system.

In summary, this chapter makes the following contributions:

- We present specification models to define the communication protocols underlying a cooperative system and the list of design objectives that the system must fulfil.
- We describe how *RACOON* integrates accountability and reputation mechanisms to foster cooperation in the system under design.
- We propose an automatic method to generate a representative set of selfish deviations from any communication protocol specified using the framework.

- We develop a game-theoretic model to predict the strategic (possibly, selfish) choices of nodes. Also, we provide an automatic methodology to generate such games using the information contained in the *RACOON* specification model.
- We present a method to evaluate alternative configurations of the incentive mechanisms to meet the Designer's objectives, using simulations based on game-theoretic reasoning.
- We illustrate the benefits of using *RACOON* by designing a P2P live streaming system and an anonymous communication system. Their extensive evaluation via simulations and real testbed deployment shows that the systems designed using *RACOON* achieve both resilience to selfish nodes and high performance.

Roadmap. In Section 5.1 we provides an overview of *RACOON*. A detailed explanation of the Design and Tuning phases is given in Section 5.3 and Section 5.4, respectively. Section 5.5 presents a performance evaluation of *RACOON*. Finally, we conclude this chapter in Section 5.6.

Remark. *Work presented in this chapter has been published in Proceedings of the 34th IEEE Symposium on Reliable Distributed Systems (IEEE SRDS'15) [106].*

5.1 OVERVIEW

RACOON is a design and simulation framework aimed at supporting Designers in building a selfish-resilient cooperative system that meets desired performance objectives. As illustrated in Figure 13, the operation of *RACOON* consists of two phases: (i) the assisted *design* of the system specification and behavioural models, and (ii) the objective-oriented *tuning* of the system parameters. The pink boxes at the top and on the right of the figure are the inputs provided by the Designer. The output of *RACOON* (the blue boxes at the bottom of Figure 13) includes a specification of the cooperative system that includes finely tuned mechanisms to achieve selfish-resilience and desired performance. In the following, we give an overview of the design and tuning phases of *RACOON*, and then provide more detail in Sections 5.3 and 5.4.

The *design* phase is initiated by the Designer, who provides a specification of the cooperative system. More precisely, the Designer specifies the communication protocols composing the system as a set of state machines, each called Protocol Automaton. In Step (1) in Figure 13, *RACOON* integrates the system specification with mechanisms to encourage nodes to cooperate. Specifically, *RACOON* uses two configurable Cooperation Enforcement Mechanisms (*CEM*): a general accountability system to audit nodes' behaviour and a reputation system to assign rewards or punishments depending on the audit results. Then, the framework extends the state machine representation of the system by adding new states and transitions that represent selfish behaviours (Step (2)). The result is an *Extended Specification* of the cooperative system, which includes selfish behaviours and cooperation enforcement mechanisms. Finally, in Step (3), *RACOON* creates a *Behavioural Model* to describe the rationality expected in the selfish nodes

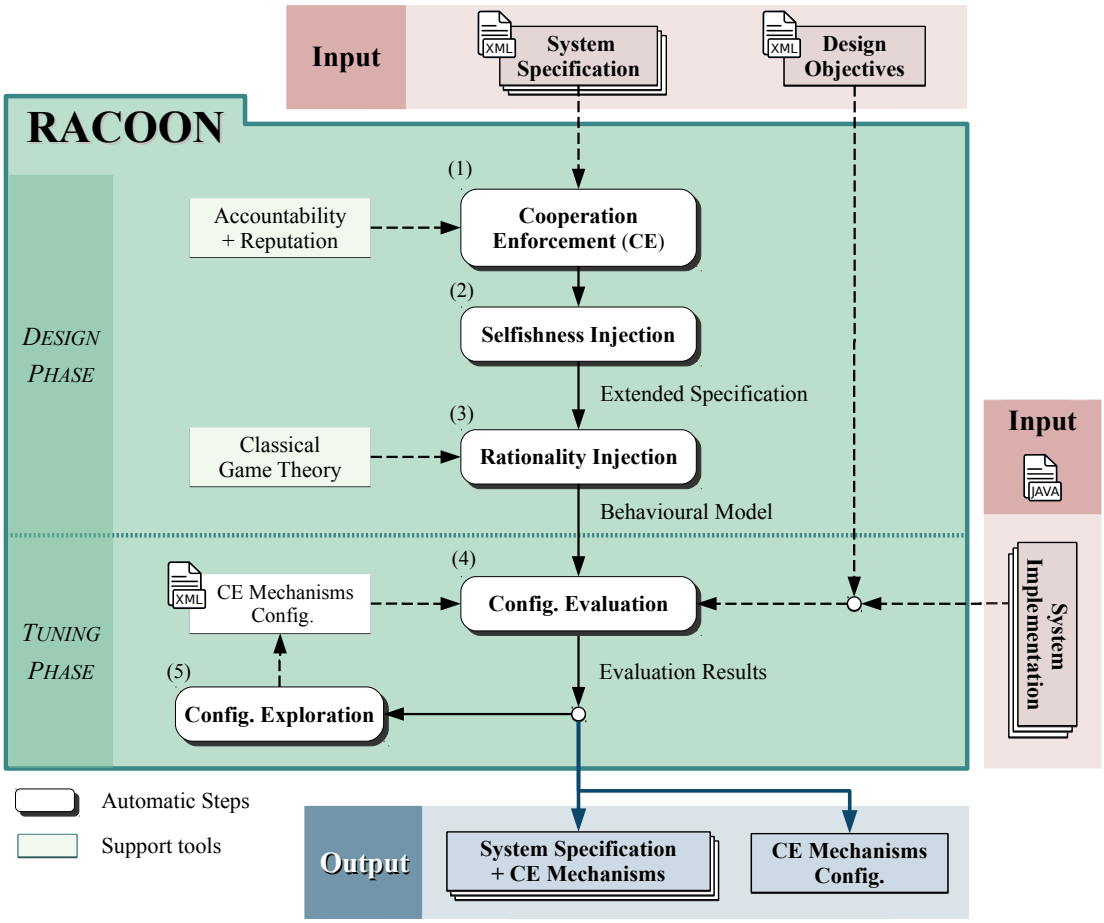


Figure 13: *RACOON* Overview.

under consideration, in order to predict their strategic decisions and actions (i.e., sticking to the communication protocols or deviating from them). To this end, the framework transforms the Extended Specification into a game model of classical Game Theory (GT), which provides the mathematical framework to model nodes' rationality.

The goal of the *tuning* phase is to find a configuration setting for the *CEM* that makes the system meet a list of *Design Objectives* set by the Designer. Tuning is an iterative refinement process consisting of a sequence of two steps: configuration evaluation (Step (3) in Figure 13) and configuration exploration (Step (4)). The evaluation is done using GT-driven simulations, carried out automatically by the custom-built simulator included in the framework. More precisely, the *RACOON* simulator uses the Behavioural Model to simulate the strategic behaviour of selfish nodes into an implementation, provided as input by the Designer, of the system specification in the simulator. Then, the framework uses the results of the evaluation to traverse the configuration space and evaluate new configuration candidates for the *CEM*. Once *RACOON* has found a configuration that meets the Design Objectives, the Designer can proceed with the implementation of the system.

5.2 ILLUSTRATIVE EXAMPLE: THE O&A PROTOCOL

In the next sections, to support the description of the *RACOON* framework, we use the simple protocol *O&A* (*Offer & Accept*) shown in Figure 14 as illustrative example. In the *O&A* protocol, a node i offers some of its available resources (i.e., files) to a group of nodes collectively named J,[1] by sending a message g_0^i with a list of such resources. Upon receiving this message, each node $j \in J$ replies with the list of resources offered by i that it accepts (message g_1^j in Figure 14). For instance, let i offer two files named `file1` and `file2`. Then, assuming that j would not accept files that it already owns, the message g_1^j may be one of the following:

- $g_1^j : \{\texttt{file1}, \texttt{file2}\}$, if j is missing both files specified in g_0^i;
- $g_1^j : \{\texttt{file1}\}$, if j has only the second file offered by i;
- $g_1^j : \{\texttt{file2}\}$, if j has only the first file offered by i;
- $g_1^j : \{\}$, if j already owns the files listed in g_0^i.

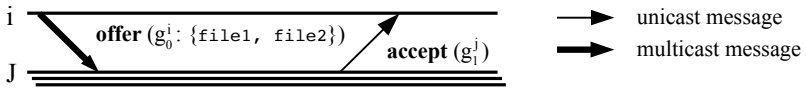


Figure 14: The *O&A* protocol between nodes i and J.

If the node i is selfish, it may decide to limit the amount of resources to share with J. To this end, i may provide false information to J, offering fewer files than what it owns. For example, Figure 15(a) shows that node i offers only `file1`, thus preventing any acceptance of the second file `file2`. In the extreme, the selfish node i might refuse to send any offer messages to J whatsoever, as shown in Figure 15(b).

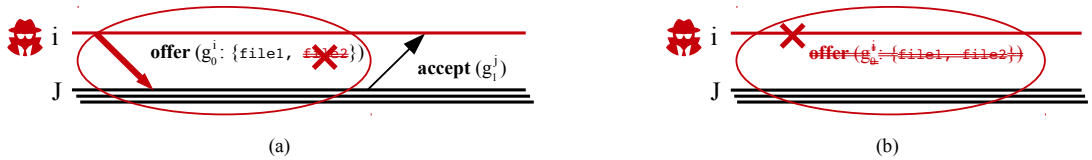


Figure 15: Selfishness manifestations in the *O&A* protocol shown in Figure 14.

5.3 RACOON DESIGN PHASE

The design phase helps the Designer in specifying a cooperative system that embeds mechanisms for enforcing cooperation as well as in defining a behavioural model of the system participants. These activities result in the generation of two artefacts, namely, the *Extended Specification* of the system and the *Behavioural Model* of selfish nodes.

1 The capital letter for J denotes a set of unique nodes.

In this section, we introduce the input of the design phase, we describe the accountability and reputation mechanisms used in *RACOON*, we present the algorithm used to generate selfish deviations, and, finally, we describe an automatic approach for building a game-theoretic behavioural model.

5.3.1 *Input of the Design Phase*

The input of the design phase is the *functional specification* of the cooperative system that should be made resilient to selfish behaviours. The functional specification describes the correct (or *cooperative*) behaviour of nodes using communication protocols, i.e., a set of rules of interaction that define what actions each node can take at each step. Like in many existing approaches [15, 53, 76, 98, 98], each communication protocol is specified using a state machine representation [84]. More precisely, in *RACOON*, the Designer specifies the correct behaviour of the nodes using a notation based on acyclic Deterministic Finite State Machines (DFSM), called a *Protocol Automaton*.[2]

Remark. *Over time, various formal approaches for describing communication protocols have been proposed in the literature. From these, in his 2012 book on "Protocol Engineering", König [101] reports Finite State Machines (FSM), Petri nets, and process calculi as the most important and most common. In particular, the author regards FSM as an adequate, intuitive and practical means for protocol development, while considering the other approaches more useful for theoretical research.*

Finite state machines have been defined and discussed in numerous publications, such as the book "Introduction to Automata Theory, Languages and Computation" by Hopcroft et al. [84]. Among the various types of FSM (e.g., extended state machines, pushdown automata, timed automata), our choice to base the notation of the Protocol Automaton on DFSM is due to compatibility with the accountability mechanism that we plan to use in RACOON, namely, FullReview [53], in which protocols are also defined as DFSM (see Section 4.2.2).

The Protocol Automaton notation allows describing the participants, operations and messages of communication protocols by enriching the DFSM notation (*states* and *transitions*) with additional information (*roles*, *methods*, *messages*, *contents*, and *constraints*). The elements of a Protocol Automaton PA are not tied to a specific run of the protocol, but they rather provide a template to describe any particular instance of it. To be more precise, the specification of a given element in the PA (e.g., a role, a message, a content) describes a spectrum of possible valid instances of that element in any protocol run. For example, in Section 5.2, we listed four alternative contents that may be included in the message g_1^j in the *O&A* protocol; these alternatives are captured by a single content definition in the PA, as we will discuss below.

Roles describe the parties involved in the protocol execution. More precisely, a role determines the responsibilities of a party (whether a node or a group of nodes) and constrains the actions that the party is allowed to execute in a protocol run.

2 In this thesis, the terms "state machine" and "automaton" will be used interchangeably.

Definition 5.1 (Role). A *role* $r \in R$ is a triple $\langle rId, cardinality, isSelfish \rangle$, with:

- rId: the alphanumeric identifier of the role,
- $cardinality$: the number of nodes represented by r. It can either be a single number or a variable number designated by a greater than ($>$), greater than or equal to ($\geqslant$), less than ($<$), less than or equal to ($\leqslant$) conditions, and
- $isSelfish$: a boolean valued `true` if r can be played by a selfish node; `false` otherwise.

For instance, in the protocol *O&A*, there are two roles. The first role, i, has cardinality 1 and can be selfish, as shown in Section 5.2. The second role, J, is played by a non-empty set of nodes (cardinality "> 1"); also, nodes have no interest in behaving selfishly when playing this role.

Remark. *The Protocol Automaton specifies the correct interactions among individual nodes, one for each role, regardless of the cardinality of their current role. For instance, the* PA *of the protocol O&A in Figure 14 defines the interactions between two individuals: the node playing as* i *and any node* $j \in J$*. The cardinality attribute allows to express how many instances of the same communication protocol are described by a certain Protocol Automaton. As an example, let the cardinality of role* J *be* 10*; then, the* PA *of O&A describes the correct behaviour in ten separate but procedurally identical protocol runs between the node* i *and each node playing the role* J*.*

The states that the cooperative system goes through when implementing a communication protocol can be formalised as follows.

Definition 5.2 (State). A *state* $s \in S$ is a triple $\langle sId, roleId, sType \rangle$, with:

- sId: the alphanumeric identifier of the state,
- $roleId$: identifies the *active role* $r \in R$ that with its actions can trigger a transition from s or that can terminate the protocol execution, and
- $sType$: specifies whether s is an `initial`, `final`, or `intermediate` state [84].

A transition between two states of the Protocol Automaton corresponds to a protocol step, i.e., the method call that determines the next protocol state.

Definition 5.3 (Transition). A *transition* $t \in T$ is a quadruple $\langle tId, state1Id, state2Id, methodId \rangle$, with:

- tId: the alphanumeric identifier of the transition,
- $state1Id$ and $state2Id$: identify the source and target states (defined in S) of t, and
- $methodId$: identifies the method $m \in M$ executed in t.

A method is the set of actions performed by a certain role that can trigger a protocol transition. In particular, a *communication method* represents the delivery of a message from one role to another, while a *computation method* performs local computations. For instance, in the *O&A* protocol, there are only communication methods, named `offer` and `accept`. Examples of computation methods include data encryption and compression, logging of messages, physical storage and retrieval of data from a local database.

Definition 5.4 (Method)**.** A *method* $m \in M$ is a couple $\langle mId, messageId \rangle$, with:

- mId: the alphanumeric identifier of the method, and
- $messageId$: identifies the message $g \in G$ sent by m, if m is a communication method, null otherwise.

A message conveyed by a communication method can be formalised as the couple below.

Definition 5.5 (Message)**.** A *message* $g \in G$ is a couple $\langle gId, contentId \rangle$, with:

- gId: the alphanumeric identifier of the message, and
- $contentId$: identifies the content $c \in C$ carried by g.

Figure 14 shows that the roles participating in the *O&A* protocol exchange the messages g_0^i and g_1^j. As discussed above, the PA specifies the correct interactions among individual nodes. Thus, the message g_1^j may convey different contents (i.e., accepting different files), depending on the node $j \in J$ that is interacting with node i. Note that the sender and receiver of a message g can be inferred from the transition t that executes the communication method m sending g. Specifically, the sender of g is the active role of the t's source state (i.e., $t.state1Id.roleId$), whereas the receiver of g is the active role of t's target state (i.e., $t.state2Id.roleId$).

The data carried by a message, i.e., a content, can be either a single data unit (e.g., a binary file) or a collection (e.g., a list of integers) of data units.

Definition 5.6 (Content)**.** A *content* $c \in C$ is a quadruple $\langle cId, cType, cSize, cLength \rangle$, with:

- cId: the alphanumeric identifier of the content,
- $cType$: provides information about the data type,[3]
- $cSize$: indicates the memory size of a single data-unit in c (given in bytes), and
- $cLength$: specifies the number of data units that comprise the content c. It can either be a single number or a variable number designated by a greater than ($>$), greater than or equal to ($\geqslant$), less than ($<$), less than or equal to ($\leqslant$) conditions.

3 Defined by the XML Schema type system.

A constraint prescribes a relationship that has to be fulfilled by two contents. In the specification model of *RACOON*, a constraint is defined as follows.

Definition 5.7 (Constraint)**.** A *constraint* $k \in K$ is a quadruple $\langle kId, content1Id, content2Id, kType \rangle$, with:

- kId: the alphanumeric identifier of the constraint,
- content1Id and content2Id: identify the two contents $\in C$ that are subject to k, and
- kType: specifies the type of relationship required by k, which can be either an ordering relation (=, <, >) or a set operation (`subset`, `strict_subset`, `equal`).

Finally, we can formalise a Protocol Automaton as follows.

Definition 5.8 (Protocol Automaton)**.** A *Protocol Automaton* is a tuple $\langle R, S, T, M, G, C, K \rangle$, with:

- R: finite, non-empty set of *roles*,
- S: finite, non-empty set of *states*,
- T: finite set of *transitions*,
- M: finite set of *methods*,
- G: finite set of *messages*,
- C: finite set of *contents*, and
- K: finite set of *constraints* on contents.

Table 16 summarises the value of the Protocol Automaton of the *O&A* protocol.

Element	Value
Roles	$R = \{\langle i, 1, true \rangle, \langle J, > 1, false \rangle\}$
States	$S = \{\langle s_0, i, initial \rangle, \langle s_1, J, intermediate \rangle, \langle s_2, i, final \rangle\}$
Transitions	$T = \{\langle t_0, s_0, s_1, offer \rangle, \langle t_1, s_1, s_2, accept \rangle\}$
Methods	$M = \{\langle offer, g_0^i \rangle, \langle accept, g_1^j \rangle\}$
Messages	$G = \{\langle g_0^i, c_0 \rangle, \langle g_1^j, c_1 \rangle\}$
Contents	$C = \{\langle c_0, integer, 4, \geqslant 0 \rangle, \langle c_1, integer, 4, \geqslant 0 \rangle\}$
Constraints	$K = \{\langle k_0, c_1, c_0, subset \rangle\}$

Table 16: The Protocol Automaton of the *O&A* protocol.

The Protocol Automaton of the *O&A* protocol can be represented by the diagram in Figure 16. The label on a transition provides information about the method that triggers the transition,

and about the message that might be sent. For example, the label between states s_1 and s_2, indicates that a node j, while playing the role J, sends the message g_1^j to role i by invoking the communication method `accept`.

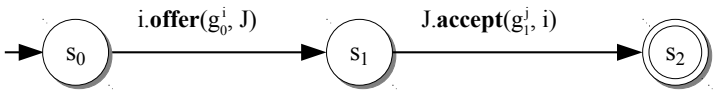


Figure 16: The state diagram representation of the Protocol Automaton specified in Table 16.

In the *RACOON* framework, protocol automata are encoded in an XML-based format, to support portability, reuse, and extensibility of the specification. The Designer specifies a new Protocol Automaton as an XML document.[4] For example, Listing 1 shows the XML representation of the PA described in Table 16.

```
<racoon name="OfferAndAccept">
  <protocol_automaton>
    <roles>
      <role id="i" cardinality="1" isSelfish="true" />
      <role id="J" cardinality=">1" isSelfish="false" />
    </roles>
    <states>
      <state id="s0" roleId="i" type="initial" />
      <state id="s1" roleId="J" type="intermediate" />
      <state id="s2" roleId="i" type="final" />
    </states>
    <transitions>
      <transition id="t0" state1Id="s0" state2Id="s1" methodId="offer" />
      <transition id="t1" state1Id="s1" state2Id="s2" methodId="accept" />
    </transitions>
    <methods>
      <method id="offer" messageId="g0i" />
      <method id="accept" messageId="g1j" />
    </methods>
    <messages>
      <message id="g0i" contentId="c0" />
      <message id="g1j" contentId="c1" />
    </messages>
    <contents>
      <content id="c0" type="integer" size="4" length=">=0" />
      <content id="c1" type="integer" size="4" length=">=0" />
    </contents>
    <constraints>
      <constraint id="k0" content1Id="c1" type="subset" content2Id="c0" />
    </constraints>
  </protocol_automaton>
  <design_objectives>
    <!-- specification of the design objectives (Section 5.4.1) -->
  </design_objectives>
</racoon>
```

Listing 1: The XML document that specifies the Protocol Automaton described in Table 16.

Remark. *The XML document provided as input by the Designer includes the specification of the Protocol Automaton as well as of the design objectives presented later in Section 5.4.1.*

4 The XML Schema for this document can be found in Appendix A.

5.3.2 *Cooperation enforcement*

The *RACOON* framework uses accountability and reputation mechanisms to make cooperation the most profitable behaviour for all nodes. In practice, the quality of service received by nodes depends on their reputation values, which are updated based on accountability audits.

The first step of the design phase of *RACOON* is the integration of the Cooperation Enforcement Mechanisms (*CEM*) into the functional specification provided by the Designer. Concretely, in addition to the protocols specified by the Designer, *RACOON* applies to each node a set of accountability protocols based on FullReview [53] and a distributed reputation system.

The *CEM* used in *RACOON* are discussed hereafter.

5.3.2.1 *Accountability Mechanism*

RACOON uses accountability techniques for detecting misbehaviours and assigning nodes non-repudiable responsibility for their actions. Concretely, we developed the *R-acc* mechanism, based on the FullReview protocols presented in the previous chapter. *R-acc* also shares some assumptions with FullReview about nodes' behaviours (i.e., no collusion) and the system (i.e., a Public Key Infrastructure is available to create trusted identities by means of digital signatures), whereas it differs from other assumptions (i.e., nodes are not risk averse and can remain in the system for a short time).

RACOON can automatically integrate *R-acc* into the Protocol Automaton specified by the Designer, thus enriching the system specification with accountability operations and protocols. To begin, *R-acc* requires each node to maintain a secure log to record all the observable actions occurred during its execution of the Protocol Automaton and of *R-acc* as well. Further, it assigns each node i to a set of other nodes, called its witness set $ws(i)$. A witness is in charge of auditing the log of its monitored nodes, generating provable evidence of their behaviour and assigning punishments or rewards accordingly. Such operations are defined by the five protocols outlined below. Overall, *R-acc* uses the same protocols of FullReview, apart from a simple but significant modification in the audit protocol.

Commitment protocol: ensures that the sender and the receiver of a message have provable evidence that the other party has logged the exchange. Figure 17 shows the integration between the Protocol Automaton *PA* of the *O&A* protocol and the commitment protocol. As an example, consider the node i in the start state s_0. Before it sends the message g_0^i to J, i records the action in a new log entry e_w. Then, i generates an authenticator α_w^i and sends it to J along with the message g_0^i.[5] Upon the reception of the message, each node $j \in J$ logs this event in a new log entry e_z, and generates the corresponding authenticator α_z^j. Finally, each node in J sends its authenticator to i (transition from state f_0 to s_1, in Figure 17), to acknowledge the reception of g_0^i.

Consistency protocol: ensures that each node maintains a single and consistent linear log. We refer to Section 4.2.2 for details.

5 For further details, we refer to the discussion on accountability techniques provided in Section 4.2 of the previous chapter.

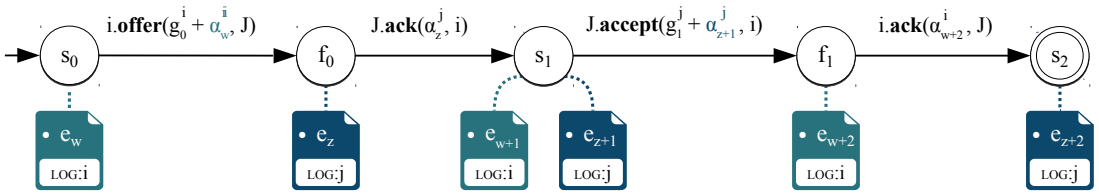


Figure 17: The integration between the commitment protocol of *R-acc* with the *O&A* protocol shown in Figure 16.

Audit protocol: a proactive and periodic inspection of a node's behaviour, based on the examination of its log. In contrast with FullReview, *R-acc* introduces the *probability of audit* parameter, which allows more control over the number of audits instead of auditing at every audit period. Figure 18 shows the *PA* of the audit protocol between a monitored node r_m and one of its witnesses r_w. Upon receiving the audit request g_{a0}, the witness may either (i) terminate the protocol in state f_7 without performing the audit, or (ii) demand (message g_{a1}) and obtains (g_{a2}) from r_m all log entries since r_m's last audit. Following the second case, r_w can then verify if r_m's log conforms to the correct Protocol Automaton making up the functional specification of the cooperative system (transition "audit" in Figure 18). The witness sends the audit result back to the monitored node (message g_{a3}). Finally, once verified the correctness of its audit as described in Section 4.2.2, r_m terminates the protocol. If instead the witness does not receive the requested log from r_m (state f_8 in Figure 18), it will address the issue by using the challenge/response protocol of *R-acc*. Concerning the accuracy of the

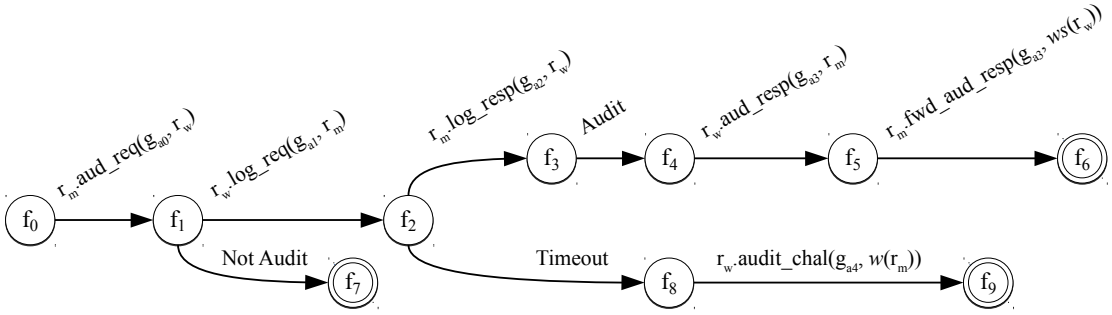


Figure 18: The Protocol Automaton of the *R-acc* audit protocol. For ease of reading, we do not represent in the figure the required execution of the commitment protocol on each message exchange of the audit protocol.

audit protocol, if a message loss occurs during the delivery of a log file, and in the absence of retransmission functions (e.g., all data is sent using UDP[6]), the witness might falsely accuse a cooperative node based on the audit of a corrupted log segment [174]. The challenge/response protocols account for this issue. False negative detections, on the other hand, may occur because of deviations from the audit protocol. For instance, a selfish witness may re-

6 Though using TCP is the natural choice for implementing a distributed accountability system, there are some reasons why a developer might use UDP. For example, because a certain level of error tolerance can result in a big performance gain, or due to legacy constraints.

turn a negative audit result without actually checking the log, to save computation resources. The FullReview protocols [53], on which *R-acc* is based, make this deviation detectable.

Challenge/response protocols: deal with nodes that do not respond to messages as provided in *PA* (e.g., an `offer` message in the *O&A* protocol) or in *R-acc* (e.g., log requests, reception acknowledgements), allowing certain tolerance for correct nodes that are slow or suffering from network problems (e.g., message losses). Specifically, if a node i has been waiting too long for a given message from another node j, i indicates the suspect state for j, and creates one of the challenges described in Section 4.2.2 to allow j to prove it is correct and get trusted again. Nodes in *R-acc* communicate only with non-suspected nodes.

Evidence transfer protocol: ensures that faulty nodes are eventually exposed by all correct nodes in the system (see Section 4.2.2).

As a final remark, the commitment protocol is the only *R-acc* protocol that modifies the functional specification of the system. The remaining protocols run in separate threads, scheduled to execute periodically.

5.3.2.2 *Reputation Mechanism*

RACOON uses a reputation system (*R-rep*) to assist nodes in choosing a cooperative partner to transact with. To provide this function, *R-rep* needs to collect information on the behaviour of each node and aggregate this information to produce a reputation value for it. The *R-acc* accountability system described in the previous section offers a practical and reliable solution to carry out the information gathering task. Once the reputation of a node has been computed, it can be used by the reputation mechanism to take action against selfish nodes while rewarding the cooperative ones. Like in other studies (e.g., [72, 104]), we use reputation in combination with an eviction condition, such that if the reputation of a node goes below a given threshold, then no other node will accept to interact with it. This creates a credible deterrent to selfish nodes, as being ignored would prevent them from receiving any service or resource from other nodes.

Notation. *In the remainder of this thesis, we say that a node is "evicted" from the system if it cannot interact with any other node in the system.*

INFORMATION GATHERING AND FEEDBACK. The functionalities of the *R-rep* reputation mechanism are distributed and decentralised, to better fit with the characteristics of cooperative systems. In particular, using the terminology defined in Section 4.1.1, every node of the system plays interchangeably the role of the trustor, trustee and recommender. The witness node in *R-acc* plays also the role of recommender in *R-rep*, as it can form an opinion of a monitored node (the trustee) based on the audit result. This solution keeps the computational overhead of the *CEM* under control, as it does not require to perform the same type of operation twice (that is, the evaluation of a certain behaviour). Furthermore, basing feedback on provable evidence offers an effective defence against false feedback (e.g., bad mouthing, false praising).

Apart from gathering information and provide feedback, the *R-rep* and *R-acc* cooperation enforcement mechanisms share also other features, to reduce design complexity and reuse available knowledge. First, *R-rep* identifies nodes using the same unique and permanent identity assigned by *R-acc*. A strong identity system prevents many attacks against the registration policy of a reputation system, such as a Sybil attack or whitewashing [121]. Second, *R-rep* relies on *R-acc* for storing the reputation data in a reliable manner. More precisely, nodes store their reputation locally. To prevent manipulations, only witnesses — in their role of recommender — can update the reputation value. Also, the update must be recorded in the *R-acc* secure log, so that any tampering can be detected.

REPUTATION ESTIMATION AND REPRESENTATION. In *R-rep*, the reputation ρ_i of a node i is an integer value between 0 (i.e., the eviction threshold) and ρ_{max}. The upper limit ρ_{max} is a parameter of the reputation mechanism.

Every time a witness of the node i in *R-acc* performs an audit on i's log, the reputation ρ_i is updated depending on the audit result. In the case of a positive audit (i.e., the witness detected at least one deviation of i from the correct behaviour), the reputation is decreased; otherwise, the node i is rewarded with an increase of its reputation value. The decrease and increase of reputations are calculated by the *punishment function* and *reward function*, defined in Definition 5.9 and Definition 5.10, respectively.

Definition 5.9 (Punishment function)**.** The punishment function $f_P : \mathbb{N} \to \mathbb{N}$ is:
$f_P(\rho_i) = \lceil (\rho_{max} - \rho_i) \cdot dp \rceil$, with:

- $\rho_{max} \in \mathbb{N}$: the upper limit of the reputation values in *R-rep*,
- $\rho_i \in \{0, \rho_{max}\}$: the reputation of a node i, and
- $dp \in \mathbb{R}_0^+$: the *degree of punishment* that controls the intensity of the reputation decrease.

The punishment function returns an integer that is proportional to the distance of the reputation from the maximum reputation that (cooperative) nodes can achieve in *R-rep*. The rationale for this choice is to punish with greater severity nodes that already have a bad reputation, in order to inhibit recidivism. The rapidity of the reputation decrease can also be modulated by the degree of punishment parameter, such that the greater dp, the greater the decrease.

Definition 5.10 (Reward function)**.** The reward function $f_R : \mathbb{N} \to \mathbb{N}$ is defined as:
$f_R(\rho_i) = \lfloor \rho_i \cdot dr \rfloor$, with:

- $\rho_i \in \{0, \rho_{max}\}$: the reputation value of a node i, and
- $dr \in \mathbb{R}_0^+$: the *degree of reward* that controls the intensity of the reputation increase.

In the case of a negative audit, the reward function f_R allows calculating the reward to assign to the cooperative node in terms of a reputation increase. In the formula, the amount of increase

is proportional to the reputation value of the node, in such a way as to reward more the nodes that have been more cooperative in the past. Also, the magnitude of the reward can be further tuned using the degree of reward parameter.

The reputation update mechanism of *R-rep* can be thus formalised as follows.

Definition 5.11 (Reputation update mechanism)**.** Let ρ_i^{old} be the reputation of a node i before being audited. Then, the reputation $\rho_i \in [0, \rho_{max}]$ of node i after the audit is:

$$\rho_i = \begin{cases} \max\{\rho_i^{old} - f_P(\rho_i^{old}), 0\}, & \text{in the case of a positive audit of node i} \\ \min\{\rho_i^{old} + f_R(\rho_i^{old}), \rho_{max}\}, & \text{in the case of a negative audit of node i} \end{cases}$$

, with:

- f_P: the *punishment function* (Definition 5.9), and
- f_R: the *reward function* (Definition 5.10).

Example 5.3.1. Let the reputation be in the range $\{0, 20\}$ and the degree of reward be 0.2. Also, let the current reputation of a cooperative node i be 5. Then, after being audited, the reputation update mechanism will assign i with the reputation $\rho_i = \min\{5 + \lfloor 0.2 \cdot 5 \rfloor, 20\} = 6$.

Example 5.3.2. Let the reputation be in the range $\{0, 10\}$ and the degree of punishment be 2. Assuming the current reputation of node i be 6, then its new reputation after a positive audit will be: $\rho_i = \max\{6 - (\lceil 2 \cdot (10 - 6) \rceil), 0\} = 0$.

The set up of the *R-rep* parameters can yield different results, with varying effects on the nodes' behaviour. In Section 5.4, we will show how the tuning phase of *RACOON* can support the automatic configuration of these parameters to induce the desired behaviour.

5.3.3 *Selfishness injection*

In the previous step of the design phase, *RACOON* extended the functional specification of the system with accountability and reputation mechanisms to sustain the selfishness-resilience of the system. Then, in the selfishness injection step, the framework creates the basis for evaluating their effectiveness. Concretely, *RACOON* provides automatic support to identify representative types of deviations from the cooperative behaviour of the system, as well as to model their execution into its functional specification (i.e., the Protocol Automaton).

In our survey of research work on selfishness in cooperative systems, presented in Chapter 2, we found that the most common motivation for a node to behave selfishly is to save bandwidth, and that defection and free-riding are the primary types of deviation to achieve this objective.[7] On the basis of these findings, we designed the selfishness injection step of *RACOON* with a special focus on selfish deviations that aim to cut down the bandwidth consumed for coopera-

7 For ease of reference, we recall that a defection is an intentional omission in the execution of a system protocol, whereas a free-ride is a reduction of the amount of resources contributed by a node without stopping the protocol execution.

tion. The investigation of other types of selfishness (e.g., computational or information-related) will be the subject and main contribution of the last part of this dissertation.

The bandwidth consumed in a communication protocol mainly depends on the number and size of the messages that are exchanged between nodes. Based on this observation, *RACOON* automatically generates three types of communication-related deviations: (i) *timeout* deviation: the node does not perform the prescribed method within the time limit; (ii) *subset* deviation: the node sends only a subset of the correct message content; and (iii) *multicast* deviation: the node sends a message only to a subset of the legitimate recipients. Note that timeout deviations can be classified as a defection, whereas subset and multicast as free-riding deviations.

Remark. *In the present version of the RACOON framework, the selfishness injection algorithm can create only subset and multicast deviations with the largest departure from the correct behaviour. In practice, message contents are shrunk down to a single data unit by subset deviations, whereas multicast deviations reduce the number of receivers of a message to one. Such a "worst-case" approach can be useful to assess the effectiveness of the cooperation enforcement mechanisms, thereby providing a conservative test of the selfishness-resilience of the system under design. Also, it makes the specification model simpler, avoiding to introduce additional parameters (i.e., the intensity of each deviation). On the other hand, focusing only on the most severe deviations can be too rigid and restrictive. We will come back to this issue in the next chapter.*

RACOON generates the three deviations listed above relying on the *Communication Selfishness Injection* (CSI) algorithm shown in Alg. 1. The algorithm takes a Protocol Automaton as input and extends it with new elements (states, transitions, roles, etc.) representing deviations. For instance, Figure 16 shows the result of the application of the *CSI* algorithm to the Protocol Automaton PA of the *O&A* case study. The figure shows that, in the correct execution of PA, the role $\mathtt{i}$ sends a message ($g_0^{\mathtt{i}}$) to J; however, if $\mathtt{i}$ is played by a selfish node, it may also (from top to bottom in the figure): timeout the protocol, send a message with a smaller payload ($g_0^{\mathtt{i}'}$), or send the message to a subset of recipients (J″). Notice that the `accept` transitions have no deviations, because — as specified in Table 16 — their active role J is not selfish.

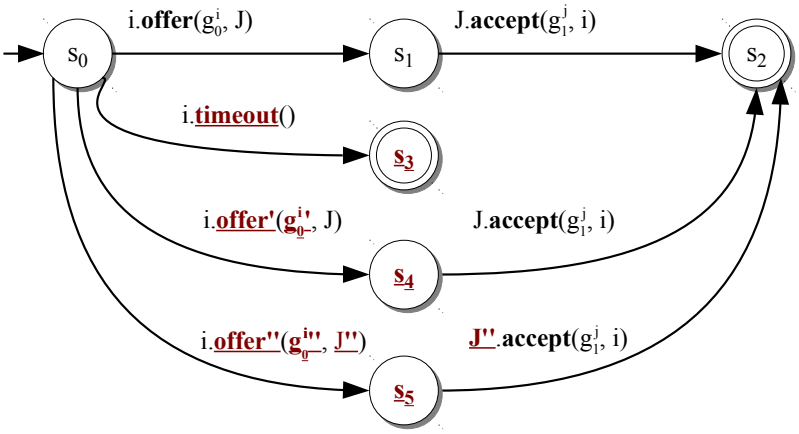


Figure 19: The Protocol Automaton of the *O&A* protocol, extended with selfish deviations.

Hereafter, we describe the pseudo-code of the *CSI* algorithm in more detail. For rapid identification, the parts of the pseudo-code in Alg. 1 that are specific to the same type of deviation are highlighted in the same colour. Furthermore, for brevity, in the pseudo-code we use the notation $\mathbf{get}(\mathtt{elementId})$ to refer to the element of PA to which the *elementId* identifier is associated. We assume the $\mathbf{get}$ notation as recursive; for example, the statement $\mathbf{get}(\mathtt{t.state1Id.roleId})$ refers to the active role of the source state of the transition t.

A *deviation point* is a transition of the Protocol Automaton in which a deviation can take place. To determine if a transition is a deviation point, the *CSI* algorithm first checks whether the active role can be played by a selfish node (line 4 in Alg. 1). If so, the algorithm verifies if the transition is a timeout (line 6), subset (line 10), or multicast (line 14) deviation point, and invokes the generative procedures accordingly. Each procedure is discussed separately below.

Timeout Deviations. For each transition $t \in T$ that is not triggered by a legit timeout method, the *CSI* algorithm generates a timeout deviation by calling the procedure *InjectTimeoutDev* (line 7 in Alg. 1). The procedure creates a new final state s' along with an empty transition connecting the source state of t with s'.

Subset Deviations. For each transition $t \in T$ triggered by a communication method, the *CSI* algorithm checks whether its conveyed message content c is a collection of data-units (line 10). If so, line 11 calls the procedure *InjectSubsetDev*, which creates new elements to represent the deviation. In particular, the procedure creates the new content c' (line 21), which has the same data type and size as c, but has only a single data-unit. The generation of a new content might also require the creation of new constraints. In fact, since c' replaces the correct content in the communication protocol execution, it must be subject to the same constraints as c. This operation is addressed by the *UpdateConstraints* procedure (invoked in line 22 in Alg. 1), which duplicates any constraint $k \in K$ related to c and replaces the correct content's identifier with that of c'. The *InjectSubsetDev* procedure terminates by connecting the new state generated in line 25 to the rest of the PA. This is done by copying the outgoing transitions of the target state s of t (lines 28-20). As a concrete example, this task creates the transition between states s_4 and s_2 in Figure 19, based on the transition connecting s_1 to s_2.

Multicast Deviations. For each transition $t \in T$ triggered by a communication method, the *CSI* algorithm checks whether the recipient of the message sent during t has a cardinality greater than 1 (line 13 in Alg. 1). If so, line 15 calls the procedure *InjectMulticastDev* to create the new role r' (line 31) with cardinality equal to 1. Then, the procedure proceeds following the same pattern as seen before: creating and adding the new elements in PA (lines 32-37), and adding the outgoing transitions of the new state created (line 38).

The *Extended Specification* of the cooperative system resulting from the execution of the *CSI* algorithm is the output of the selfishness injection step.

Alg. 1: The *Communication Selfishness Injection* (CSI) algorithm.

Input: A *Protocol Automaton PA* := ⟨R, S, T, M, G, C, K⟩.
Output: *PA*.

Algorithm `CSI(PA)`

```
1  origT := T  // transitions originally included in PA
2  foreach t ∈ origT do
3      activeRole := get(t.state1Id.roleId)
4      if activeRole.isSelfish = true then
5          method := get(t.methodId)
6          if method.mId ≠ "timeout" then
7              InjectTimeoutDev(t)
8          if get(method.messageId) ≠ null then
9              c := get(method.messageId.contentId)  // sent content
10             if c.cLength > 1 then
11                 InjectSubsetDev(t, s, c)
12             s := get(t.state2Id)  // target state
13             r := get(s.roleId)  // recipient role
14             if r.cardinality > 1 then
15                 InjectMulticastDev(t, s, r)

   Procedure InjectTimeoutDev(t)
16     s' := ⟨new_sId, null, final⟩
17     m' := ⟨"timeout", null⟩
18     sourceState := get(t.state1Id)
19     t' := ⟨new_tId, sourceState.sId, s'.sId, m'.mId⟩
20     add s', m', and t' to PA

   Procedure InjectSubsetDev(t, s, c)
21     c' := ⟨new_cId, c.cType, c.cSize, 1⟩
22     UpdateConstraints(c'.cId)
23     g' := ⟨new_gId, c'.cId⟩
24     m' := ⟨new_mId, g'.gId⟩
25     s' := ⟨new_sId, s.roleId, s.sType⟩
26     t' := ⟨new_tId, t.state1Id, s'.sId, m'.mId⟩
27     add s', c', g', m', and t' to PA
28     foreach ot ∈ T | ot.state1Id = s.sId do
29         ot' := ⟨new_otId , s' , ot.state2Id , ot.methodId⟩;
30         add ot' to PA;

   Procedure InjectMulticastDev(t, s, r)
31     r' := ⟨new_rId, 1, r.isSelfish⟩
32     s' := ⟨new_sId, r'.rId, s.sType⟩
33     correctMessage := get(t.methodId.messageId)
34     g' := ⟨new_gId, correctMessage.contentId⟩
35     m' := ⟨new_mId, g'⟩
36     t' := ⟨new_tId, t.state1Id, s'.sId, m'.mId⟩
37     add r', s', g', m', and t' to PA
38     add out-transitions of s'      ▷ as in lines 28-30
```

The worst case running time of the *CSI* algorithm is $O(|T| \cdot \max\{|K|, |S|\})$, where $|T|$ is the number of transitions in the PA, $|K|$ the number of constraints, and $|S|$ the number of states. This bound follows from the fact that, for each transition of the Protocol Automaton, the algorithm calls three methods having the following orders of complexity:

- `InjectTimeoutDev` is $O(1)$, because it involves operations having constant-time performance, such as the creation of new PA elements as well as the retrieval (function **get**) and storage (function **add**) of elements from the PA.[8]
- `InjectSubsetDev` includes two non-constant time activities: (i) the `UpdateConstraints` function (line 22 in Alg. 1), which is linear to the size of the constraints set K, and (ii) the copy of out-transitions of a state in lines 28-30, which is linear to the number of states in S.[9] Thus, the overall complexity of `InjectSubsetDev` is linear to the cardinality of the larger set between K and S, i.e., $O(\max\{|K|, |S|\})$.
- `InjectMulticastDev` is $O(|S|)$, due to the complexity of copying the out-transitions of a state (line 38), already discussed in the previous bullet.

The complexity of `InjectSubsetDev` dominates that of the other functions executed in the main loop of the *CSI* algorithm, which explains the total complexity of *CSI* presented above.

5.3.4 *Rationality injection*

Rationality is an inherent quality of selfish nodes, which determines their decision-making in choosing and performing the behaviour that they expect to be the most profitable to adopt. We recall from Chapter 2 that a behaviour can be considered more or less profitable than another based on the utility that it yields to the node (Definition 2.3), and, therefore, based on the benefits and costs of performing that behaviour when participating in the system (Definition 2.2). Thus, in a nutshell, a selfish node relies on its rationality to decide whether to stick to the protocol specification or to deviate from it, depending on which option maximises its utility.

A *Rationality Model* is a mathematical or logical (rule-based) description of the decision-making process of selfish nodes. The Extended Specification created in the previous step of the *RACOON* methodology cannot serve for this purpose, because it is limited to describing possible behaviours of selfish nodes without giving any indication of the likelihood of a particular behaviour. In *RACOON*, the rationality model is built on Game Theory (GT) [130], which appears as the natural candidate for the formal representation of the rationality of selfish individuals.[10] Particularly, we rely on the most mature and well-developed branch of GT, i.e., classical GT. *RACOON* provides an automatic tool to describe the Extended Specification within the mathematical framework of GT, thereby translating a Protocol Automaton PA into a game, referred to as the *Protocol Game* PG.

8 The **get** and **add** operations are implemented using hash tables, and, specifically, the Java class HashMap (JDK 8), which enables $O(1)$ lookups in the average case and $O(\log n)$ in the worst case.

9 Given a transition t and a target state s, the maximum number of out-transitions of s is $|S| - 2$, i.e., all states in $|S|$ but s and the source state of t.

10 We refer to Section 3.2 for notation, terminology and background material.

The Protocol Game is the final output of the design phase, i.e., the *Behavioural Model*, which provides the necessary structure for predicting the dynamics of selfish nodes during the tuning phase of the *RACOON* framework. Hereafter, we describe the Protocol Game as well as the process to generate it in more detail.

GAME TYPE. From the classification of game types in classical GT (see Section 3.2.2), we consider non-cooperative sequential games as the most suitable type for describing the Protocol Game. The reason to choose non-cooperative GT is twofold: first, it studies rational utility-maximisers that act individually, like the selfish nodes participating in cooperative systems;[11] second, it aims to make predictions about the strategic behaviour of rational individuals, which is one of the objectives of the *RACOON* framework. At the same time, the reason to choose sequential games to describe a Protocol Game is that they allow modelling interactions in which the participants act in turns, like the nodes participating in a communication protocol.

GAME DESCRIPTION. A non-cooperative sequential game is usually represented as a rooted tree, also called *extensive form* representation. In the following, we provide a formal definition of *game tree* and *extensive form game*.

Definition 5.12 (Game tree)**.** A *game tree* $\mathcal{T}$ is a quadruple $\langle \mathcal{N}, n0, \mathcal{L}, pred \rangle$, with:

- $\mathcal{N}$: finite, non-empty set of *nodes* (Definition 5.15),
- $n0$: a unique *root* node $\in \mathcal{N}$,
- $\mathcal{L}$: finite, non-empty set of terminal nodes (*leaves*), such that $\mathcal{L} \subset \mathcal{N}$. Also, let $\mathcal{D} = \mathcal{N} \setminus \mathcal{L}$. $\mathcal{D}$ is called the set of *decision nodes*, and
- $pred : \mathcal{N} \rightarrow \mathcal{D} \cup \{ null \}$: the immediate predecessor of a node $n \in \mathcal{N}$.

Definition 5.13 (Extensive form game)**.** A finite *extensive form game* consists of:

- $\mathcal{T} = \langle \mathcal{N}, n0, \mathcal{L}, \mathtt{pred} \rangle$: a *game tree*,
- $\mathcal{P}$: finite, non-empty set of *players* (Definition 5.14),
- $\mathcal{A}$: finite set of *actions* (Definition 5.16),
- $\mathcal{H}$: finite set of *information sets* (Definition 5.18),
- A set of functions that describe for each $n \in \mathcal{D}$,
  - $moves : \mathcal{N} \rightarrow \mathcal{A}$: finite set of possible actions at n,
  - $player : \mathcal{N} \rightarrow \mathcal{P}$: the player who moves at n,
  - $succ : \mathcal{N}, \mathcal{A} \rightarrow \mathcal{N}$: the successor node of n from action a, and
  - $info : \mathcal{N} \rightarrow \mathcal{H}$: the information set that contains n.
- $u_p(l) \rightarrow \mathbb{R}$: the *utility* assigned to player p as a function of the leaf $l \in \mathcal{L}$.

11 This is because in Section 5.3.2 we made the assumption that nodes do not collude, similarly to FullReview and many other cooperation enforcement mechanisms.

GAME MAPPING. A Protocol Automaton $PA = \langle R, S, T, M, G, C, K \rangle$ is translated into elements of a Protocol Game PG, which is modelled as an extensive form game. Each player of PG is assigned to exactly one role $\in R$, and it is defined as follows.

Definition 5.14 (Player)**.** A *player* $p \in \mathcal{P}$ is a couple $\langle \texttt{pId}, \texttt{roleId} \rangle$, with:

- pId: the alphanumeric identifier of the player, and
- roleId: identifies the role of the Protocol Automaton PA that corresponds to p.

The game tree $\mathcal{T}$ of the Protocol Game comprises a set of nodes $\mathcal{N}$, with each node representing a state in the Protocol Automaton. Particularly, each leaf node $\in \mathcal{L}$ translates to a final state in the PA, and represents a possible *outcome* of the stage game. The edges of the game tree correspond to the transitions of the PA, whose methods are translated into actions of the Protocol Game. In the following, we provide the definition of nodes and actions.

Definition 5.15 (Node)**.** A *node* $n \in \mathcal{N}$ is a tuple $\langle \texttt{nId}, \texttt{stateId}, \texttt{predId}, \texttt{infosetId} \rangle$, with:

- nId: the alphanumeric identifier of the node,
- stateId: identifies the state of the Protocol Automaton PA that corresponds to n,
- predId: indicates the predecessor decision node $\in \mathcal{D}$, and
- infosetId: specifies the information set that contains n.

Definition 5.16 (Action)**.** An *action* $a \in \mathcal{A}$ is a tuple $\langle \texttt{aId}, \texttt{methodId}, \texttt{playerId}, \texttt{sourceNodeId}, \texttt{targetNodeId} \rangle$, with:

- aId: the alphanumeric identifier of the action,
- methodId: refers the method of the Protocol Automaton PA that corresponds to a,
- playerId: indicates the player $\in \mathcal{P}$ that performs the action,
- sourceNodeId: specifies the node $\in \mathcal{D}$ where the action can be triggered, and
- targetNodeId: specifies the node $\in \mathcal{N}$ where the action leads to.

A *play* y is a path of the game tree, i.e., a sequence of actions, elements of $\mathcal{A}$, from the root to a leaf. It describes a particular interaction between players. The *strategy* of a player is the ordered sequence of actions that she takes in a certain play. The strategy set (or strategy space) Σ is the set of possible strategies available to a player. A *strategy profile* is the vector that specifies a strategy for every player.

Definition 5.17 (Strategy)**.** The *strategy* $\sigma_{i,j} \in \Sigma_i$ of a player $p_i \in \mathcal{P}$ for a given play y_j is the ordered sequence of actions $\{a \in \mathcal{A} \mid a \in y_j \, , a.\texttt{playerId} = p_i.\texttt{pId}\}$.

The Protocol Game is a game with complete information [130], because all the factors of the game mechanisms (e.g., players, order of moves, possible strategies) are specified in the PA, and, therefore, are common knowledge to the players. Furthermore, the PG is a game with imperfect information, because players, when making a decision, are not informed of all the events that have previously occurred.[12] Thus, when a player performs an action, she does not know exactly in which decision node of $\mathcal{T}$ she currently is, because she cannot determine if the previous player's action was a deviation or the correct method. Such condition of uncertainty is defined by an information set [130], which is the set of alternative decision nodes that might be played by a given player at a certain turn and among which the player cannot distinguish. In the PG, the definition of an information set consists only of its identifier, since the association with the nodes is made in their definition (element `infosetId`).

Definition 5.18 (Information set)**.** An *information set* $h \in \mathcal{H}$ is a singleton $\langle \mathtt{hId} \rangle$, where `hId` is the alphanumeric identifier of the information set.

The *RACOON* framework translates any Protocol Automaton PA into the elements defining a Protocol Game PG using the algorithm *PAtoPG* shown in Alg. 2. For clarity, we highlighted with different colours the parts of the pseudo-code that generate players (blue), nodes (green), and actions (red). Also, note that we used the same `get` notation introduced in the previous section to describe Alg. 1. Hereafter, we comment on some parts of the *PAtoPG* algorithm.

The first part of Alg. 2 populates the players set $\mathcal{P}$ (lines 2-4) and translates the initial state of the PA into the root of the game tree (lines 6-7). The other nodes, along with actions and information sets, are generated by the *TraversePA* procedure, during a recursive traversal of the states of the Protocol Automaton. Each traversal iteration starts with the creation of a new information set h (line 10), which will be assigned to the decision nodes created during the iteration (line 18). Then, for each outgoing transition `t` of the visited state `s`, the *TraversePA* procedure (i) creates a new node n to map the target state `tState` of the transition `t`, and (ii) creates a new action a to map the `method` in the PA that triggers `t` (lines 22-23). In particular, if `tState` is a final state, then n is also added to the set of leaves $\mathcal{L}$ (lines 15-16); otherwise, after the creation of n (lines 18-19), the procedure invokes the recursive call on the target state (line 20). Finally, the algorithm *PAtoPG* terminates in line 9 with the calculation of the utilities. This task is supported by the *CalculateUtilities* procedure, which assigns a utility to each player for each outcome of the game. We discuss this task later in this section.

Table 17 reports the values of the players, nodes, actions, and information sets that translate the extended Protocol Automaton of the *O&A* protocol shown in Figure 19. A graphical representation of these values is provided by the diagram in Figure 20.

In Definition 5.13, we listed four functions that are needed to define an extensive form game, namely, *moves*, *play*, *succ*, *info*. Alg. 3 shows how these functions can be implemented in *RACOON* using the elements of the Protocol Game created by the *PAtoPG* algorithm.

12 This knowledge can be gained only after an audit.

Alg. 2: Algorithm for the translation of a Protocol Automaton PA into a Protocol Game. Highlighted in blue are the instructions to generate players, in green to generate nodes, and in red to generate actions.

Input: A *Protocol Automaton* $PA := \langle R, S, T, M, G, C, K \rangle$.
Output: A *Game Tree* $\mathcal{T}$, the *players* $\mathcal{P}$, the *actions* $\mathcal{A}$, the *information sets* $\mathcal{H}$, and the *utilities*.

Algorithm `PAtoPG(PA)`

1 **create** empty sets $\mathcal{P}, \mathcal{N}, \mathcal{L}, \mathcal{H}, \mathcal{A}$
`/* create the players */`
2 **foreach** role $\in$ R **do**
3 $p := \langle$new_pId, role.rId$\rangle$
4 **add** p to $\mathcal{P}$
`/* create the root of the game tree */`
5 s := the initial state of PA
6 $n0 := \langle$"n0", s.sId, null, null$\rangle$
7 **add** $n0$ to $\mathcal{N}$
`/* traverse the Protocol Automaton */`
8 TraversePA(s,$n0$)
`/* calculate the utility values */`
9 CalculateUtilities()

Procedure `TraversePA(s,pred)`

10 $h := \langle$new_hId$\rangle$
11 $p :=$ GetPlayer(s.roleId) `// return p such that p.roleId = s.roleId`
12 **foreach** t $\in$ T | t.state1Id = s.sId **do**
13 tState := **get**(t.state2Id)
14 **if** *tState.sType = final* **then**
15 $n := \langle$ new_nId, tState.sId, pred.nId, null $\rangle$ `// leaf node`
16 **add** n to $\mathcal{N}, \mathcal{L}$
17 **else**
`/* all decision nodes that can be reached from s belong to the same information set h */`
18 $n := \langle$ new_nId, tState.sId, pred.nId, h $\rangle$ `// decision node`
19 **add** n to $\mathcal{N}$
20 TraversePA(tState, n)
21 method := **get**(t.methodId)
22 $a := \langle$ new_aId, method.mId, p.pId, n.predId, n.nId $\rangle$
23 **add** a to $\mathcal{A}$
24 **add** h to $\mathcal{H}$

Element	Value
Players	$\mathcal{P} = \{\langle p_0, \mathrm{i}\rangle, \langle p_1, \mathrm{J}\rangle\}$
Nodes	$\mathcal{N} = \{\langle n0, s_0, \mathrm{null}, \mathrm{null}\rangle, \langle n1, s_1, n0, h_1\rangle, \langle n2, s_4, n0, h_1\rangle, \langle n3, s_5, n0, h_1\rangle,$ $\langle n4, s_2, n1, \mathrm{null}\rangle, \langle n5, s_2, n2, \mathrm{null}\rangle, \langle n6, s_2, n3, \mathrm{null}\rangle, \langle n7, s_3, n0, \mathrm{null}\rangle\}$
Actions	$\mathcal{A} = \{\langle a_0, \mathrm{offer}, p_0, n0, n1\rangle, \langle a_1, \mathrm{offer}', p_0, n0, n2\rangle, \langle a_2, \mathrm{offer}'', p_0, n0, n3\rangle,$ $\langle a_3, \mathrm{timeout}, p_0, n0, n7\rangle, \langle a_4, \mathrm{accept}, p_1, n1, n4\rangle, \langle a_5, \mathrm{accept}, p_1, n2, n5\rangle,$ $\langle a_6, \mathrm{accept}, p_1, n3, n6\rangle\}$
Information sets	$\mathcal{H} = \{\langle h_1\rangle\}$

Table 17: Players, nodes, actions, and information sets that translate the Protocol Automaton in Figure 19.

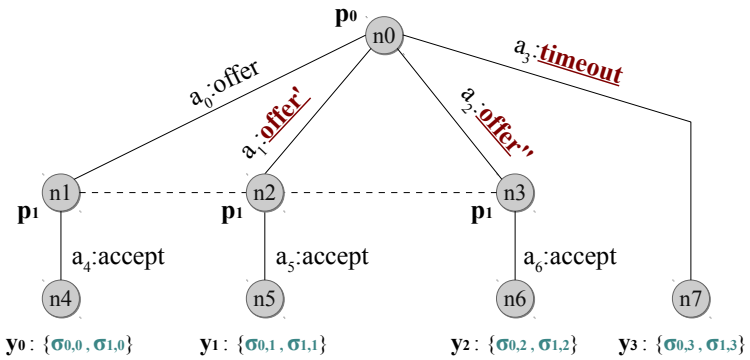


Figure 20: A visual representation of the Protocol Game described in Table 17. The label besides each decision node indicates the player that takes action at that node. The label on each edge denotes an action along with its corresponding method in the PA. Nodes in the same information set are connected by a dashed line. The labels below each leaf denote the strategies of that play.

Algorithm 3: Implementation of the *moves*, *play*, *succ*, *info* functions of an extensive form game.

Input: A *Game Tree* $\mathcal{T} = \langle \mathcal{N}, n0, \mathcal{L}, pred \rangle$, the set of *players* $\mathcal{P}$, and the set of *actions* $\mathcal{A}$.

Function *moves* ($n \in \mathcal{N}$)

```
create the empty set actionsResult
foreach a ∈ A | a.targetNodeId = n.nId do
    add a to actionsResult
return actionsResult
```

Function *play* ($n \in \mathcal{N}$)

```
foreach a ∈ A do
    if a.sourceNodeId = n.nId then
        return get(a.playerId)
```

Function *succ* ($n \in \mathcal{N}$, $a \in \mathcal{A}$)

```
if a.sourceNodeId = n.nId then
    return get(a.targetNodeId)
```

Function *info* ($n \in \mathcal{N}$)

```
return get(n.infosetId)
```

The complexity of the *PAtoPG* algorithm is determined by three operations:

- Generation of players, which is linear to the number of roles in $R \in PA$.
- Generation of the game tree, performed by the recursive method `TraversePA`. The method traverses each state of each path of the Protocol Automaton; the traversal has linear complexity $O(|S| + |T|)$, where $|S|$ and $|T|$ are the cardinalities of the states and transitions sets in the PA.[13] At each state visit, `TraversePA` executes only constant-time operations (e.g., generation of new elements, retrieval and storage of information from the PA). Hence, the generation of the game tree has complexity $O(|S| + |T|)$.
- Calculation of the utilities associated with each outcome of the game tree, which, in the worst case, it requires exploring every node in $\mathcal{N}$ and every edge (i.e., every action in $\mathcal{A}$). The time complexity of this activity can be expressed as $O(|\mathcal{N}| + |\mathcal{A}|)$ [46].

Based on the above observations, we can conclude that the overall complexity of the *PAtoPG* algorithm is $O(|R| + |S| + |T| + |\mathcal{N}| + |\mathcal{A}|)$.

UTILITY FUNCTION. The *utility function* of a player assigns a value — called *utility*, or *payoff* — to each outcome of a game [130]. In *RACOON*, the utility obtained by a player from playing the Protocol Game has two terms: the cost of sharing resources and the incentives (benefits or additional costs) provided by the cooperation enforcement mechanisms.[14] Since our focus is on selfish deviations that aim at saving bandwidth consumption, we calculate the cost k_i incurred by the player p_i as the bandwidth necessary to implement a certain player's strategy. Furthermore, we calculate the incentives as a function f_I of the selected strategy $\sigma_{i,j}$ and of the expected reputation that the player would get after her accountability audit in the immediate future.

Definition 5.19 (Utility function)**.** The *utility* function $u : \Sigma_i \to \mathbb{Z}$ for player p_i when implementing the strategy $\sigma_{i,j}$ is calculated as follows:

$u(\sigma_{i,j}) = -k(\sigma_{i,j}) + f_I$, with:

- $k \in \mathbb{N}$: the communication costs incurred by the player, and
- $f_I : \mathbb{N}, \mathbb{N} \to \mathbb{Z}$: the *incentive function* (Definition 5.21).

The intuition behind the incentive function is based on two considerations. First, the underlying assumption of any incentive mechanism is that the threat of future punishments (resp. the promise of future rewards) has an impact on the decision-making of selfish nodes, and thus on the utility they expect to derive from the execution of a certain strategy. In *RACOON*, the rationality model encourages the players to deviate from the protocol as much as possible (to save resources), as long as this can be accomplished without being evicted by the *CEM*. Second, the *CEM* links the probability of eviction to the reputation of the node. Therefore, if the reputation

13 The Protocol Automaton can be seen as a Directed Acyclic Graph, where each state corresponds to a node and each transition to an edge. Finding the total number of paths in a DAG can be done in $O(|nodes| + |edges|)$ [46].

14 We assume that all players obtain the same benefit from playing the same game.

ρ_i of a player p_i increases at a point in time, then p_i is more likely to deviate in the future, as she will be further from the eviction threshold than before. On the other hand, if ρ_i decreases, then the player will refrain from deviating in the future because she will get closer to eviction.

From the above considerations, we define the *incentive function* as follows.

Notation. *For ease of exposition, we refer to the "reputation of a player" $p \in \mathcal{P}$ as the reputation of the node in the system that is playing as p.*

Definition 5.20 (Expected reputation)**.** The *expected reputation* ρ^e of a player p is the reputation that the player would get after an audit and consequent execution of the reputation update mechanism.[15]

Definition 5.21 (Incentive function)**.** Let ρ_i be the current reputation of a player p_i playing the strategy $\sigma_{i,j}$, and let ρ_i^e be the expected reputation of the same player. Then, the *incentive function* $f_I : \mathbb{N}, \mathbb{N} \to \mathbb{Z}$ is defined as:

$$f_I(\rho_i, \rho_i^e) = \begin{cases} I_R, & \text{if } \rho_i^e - \rho_i > 0 \\ 0, & \text{if } \rho_i^e - \rho_i = 0 \\ -I_P, & \text{if } \rho_i^e - \rho_i < 0 \end{cases}$$

, with:

- $I_R \in [0, +\infty)$: the benefits expected from future rewards, and
- $I_P \in [0, +\infty)$: the costs expected from future punishments.

The incentive function f_I quantifies the expected variations of a reputation value in terms of costs or benefits. When a player gets a reward in the form of a reputation increase, then the function f_I assumes a positive value: since she is farther from the eviction threshold, she can afford to perform more deviations in the future. This corresponds to the utility increment I_R. On the contrary, when a player p_i gets a punishment in the form of a reputation decrease, then the incentive function assumes a negative value. This value can be computed as follows. If the expected reputation value ρ_i^e goes below the eviction threshold, then p_i is evicted. This negative event by itself corresponds to a cost, called eviction cost, conventionally set at a very high nominal value. If instead, the expected reputation value ρ_i^e stays above the eviction threshold, then the player, wishing to return to the previous reputation level, must behave correctly in the future and give up to potential benefits from deviations.

SUMMARY EXAMPLE. Let us consider the following setting of the *O&A* protocol case study (see Figure 14). The role i has to send an `offer` message g_0^i to role J, with J having cardinality 5. The `offer` message conveys a content c_0, containing the identifiers of the files that i wants

15 See Definition 5.11.

Outcome	Player	Strategy	J.cardinality	c_0.cSize	c_0.cLength	Communication Cost
o_0	p_0	$\sigma_{0,0}$	5	4	10	200
	p_1	$\sigma_{1,0}$	1	4	10	40
o_1	p_0	$\sigma_{0,1}$	5	4	1	20
	p_1	$\sigma_{1,1}$	1	4	1	4
o_2	p_0	$\sigma_{0,2}$	1	4	10	40
	p_1	$\sigma_{1,2}$	1	4	10	40
o_3	p_0	$\sigma_{0,3}$	0	0	0	0
	p_1	$\sigma_{1,3}$	0	0	0	0

Table 18: Illustrative example of the computation of the communication costs incurred by the players of the Protocol Game shown in Figure 20.

to offer to J. Let the identifiers be encoded as integers, each of size 4 byte, and the number of identifiers sent via c_0 be 10. Finally, let J accept all the identifiers offered by i. Given this setting, and given the Protocol Game PG modelling the *O&A* protocol (see Figure 20), in the following, we calculate the utilities of the nodes participating in the protocol.

First, we calculate the communication costs. Consider for instance the communication cost of the correct strategy $\sigma_{0,0}$, played by the player p_0 who maps the role i in the *O&A* protocol. The cost is as follows:

$$k(\sigma_{0,0}) = J.cardinality \times (c_0.cSize \times c_0.cLength) = 200 \,.$$

Table 18 summarises the communication costs of all strategies for each player. As another example, consider the strategy $\sigma_{1,1}$, played by the player p_1 that maps the role J in PG. In this case, the player received a single file identifier because the other player has performed a subset deviation. Thus, the communication cost of p_1 is:

$$k(\sigma_{1,1}) = 1 \times (4 \times 1) = 4 \,.$$

Second, to calculate the (positive or negative) incentives provided by the cooperation enforcement mechanisms of *RACOON*, let the reputation mechanism *R-rep* be configured as in the Example 5.3.2 in Section 5.3.2: the maximum reputation be 10 and the degree of punishment be 2. Also, let the degree of reward be 0, like in FullReview [53].[16] If the current reputation of player p_0 and player p_1 is 6, then the incentive values contributed by the incentive function are as reported in Table 19.

Finally, we can calculate the utility derived by each player for each strategy defined in the PG by replacing the communication costs and incentive values calculated above into the utility function presented in Definition 5.19. The results are displayed in Figure 21: the pairs of numbers (one for each player) below each leaf are the utility values.

16 The only incentive included in FullReview is the threat of eviction from the system.

Outcome	Player	Strategy	Audit result	Expected reputation	Incentive value
o_0	p_0	$\sigma_{0,0}$	Negative	6	0
	p_1	$\sigma_{1,0}$	Negative	6	0
o_1	p_0	$\sigma_{0,1}$	Positive	0	-10000 [a]
	p_1	$\sigma_{1,1}$	Negative	6	0
o_2	p_0	$\sigma_{0,2}$	Positive	0	-10000 [a]
	p_1	$\sigma_{1,2}$	Negative	6	0
o_3	p_0	$\sigma_{0,3}$	Positive	0	-10000 [a]
	p_1	$\sigma_{1,3}$	Negative	6	0

[a] The nominal cost assigned to an eviction.

Table 19: Illustrative example of the calculation of the incentive values assigned to the players of the Protocol Game shown in Figure 20.

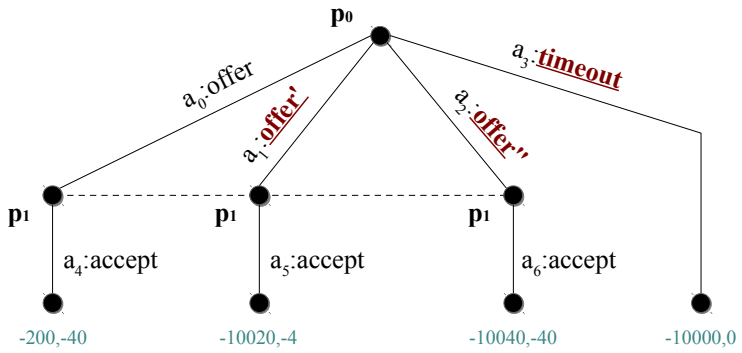


Figure 21: Illustrative example of the utility values that each player would obtain from playing a certain strategy in the Protocol Game described in Table 17.

5.4 RACOON TUNING PHASE

The tuning phase of *RACOON* aims at configuring the accountability and reputation mechanisms according to a list of design objectives input by the Designer. Tuning involves an iterative two-step refinement process, which alternates evaluation with the tuning of the configuration parameters. The evaluation involves Game Theory (*GT*) analysis and simulations to study the system dynamics in a given configuration setting. Then, an exploration algorithm uses the evaluation results to optimise the parameters of the *CEM*. The tuning process ends after a set number of iterations, or when a configuration that satisfies the Designer's objectives is found.

5.4.1 *Input of the Tuning phase*

RACOON allows defining selfish-resilience and performance objectives for the cooperative systems designed within its framework. Each of these design objectives specifies a predicate over

a system metric, which can be evaluated by the *RACOON* evaluation tool. Examples of predicates are "at most" and "at least". Hereafter, we present the objectives natively supported by *RACOON*:

- *Deviation rate*: the frequency of the deviations performed by selfish nodes.
- *Cooperation level*: the fraction of cooperative nodes in the system.
- *Audit precision*: the number of correct positive audits divided by the total number of positive audits.
- *Wrongful eviction rate*: the fraction of correct nodes wrongly evicted by the *CEM*, due to false-positive audit results.
- *CEM message overhead*: the costs of the accountability and reputation mechanisms in terms of extra messages.
- *CEM bandwidth overhead*: the costs of the accountability and reputation mechanisms in terms of bandwidth consumed.

We selected these metrics based on two factors: the metrics should be application-independent, so as to be meaningful for any cooperative system designed using *RACOON*; the metrics should be commonly used in related work on incentive mechanisms (e.g., [28, 53, 72, 100, 111, 124]). Examples of design objectives are "cooperation level *at least* 0.8" and "CEM message overhead *at most* 0.6".[17]

The Designer can also specify custom application-specific requirements, e.g., on throughput [100, 165], jitter [72, 111], anonymity [27], or download time [89, 116, 135]. For each custom objective added to the list of design objectives, the Designer needs to implement the specific methods to collect and evaluate the related metrics in the evaluation tool.

Definition 5.22 (Design Objective)**.** A *design objective* $o \in O$ is a quintuple $\langle \mathtt{oId}, \mathtt{predicate}, \mathtt{value}, \mathtt{minValue}, \mathtt{maxValue} \rangle$, with:

- `oId`: the alphanumeric identifier of the design objective. The design objectives natively supported by the framework have reserved identifiers (i.e., `deviation_rate`, `wrongful_eviction_rate`, `CEM_message_overhead`, `CEM_bandwidth_overhead`, `audit_precision`, `cooperation_level`),
- `predicate`: the predicate imposed by the design objective o over a system metric. It can be either `at_least` or `at_most`,
- `value`: the threshold value $\in \mathbb{R}$ imposed by the design objective o over a system metric,
- `minValue`: the minimum value $\in \mathbb{R}$ that can be associated with the design objective (default: 0.0), and
- `maxValue`: the maximum value $\in \mathbb{R}$ that can be associated with the design objective (default: 1.0).

17 Despite not all the *metric-predicate* combinations are logical or desirable (e.g., "audit precision *at most* 0.0"), the evaluation tool of *RACOON* accepts their specification for the sake of flexibility.

In the *RACOON* framework, the Designer specifies the design objectives in the same XML document she used to specify the Protocol Automaton (see Section 5.3.1). Listing 2 shows an example of such specification.

```
<racoon name="OfferAndAccept">
  <protocol_automaton>
    <!-- specification of the design objectives (Section 5.3.1) -->
  </protocol_automaton>
  <design_objectives>
    <objective id="cooperation_level" predicate="at_least" value="0.8" />
    <objective id="CEM_message_overhead" predicate="at_most" value="0.6"/>
    <custom_objective id="jitter" predicate="at_most" value="0.03" />
  </design_objectives>
</racoon>
```

Listing 2: The XML document that specifies the Design Objectives and the Protocol Automaton.

5.4.2 *Configuration evaluation and exploration*

Configuration evaluation and exploration are the two interleaving steps of the *RACOON* framework that aim to find a satisfactory configuration for the cooperation enforcement mechanisms that meets the design objectives set by the Designer. The automatic approach adopted by *RACOON* is to simulate the cooperative system under design in different regions of the parameter space, using the Behavioural model resulting from the design phase to drive the behaviour of selfish nodes. The exploration ends after a predefined number of iterations, or when a configuration satisfying the Designer's objectives is found.

To simulate the behaviour of selfish nodes, we developed and integrated into the *RACOON* framework an event-based simulator (*R-sim*) specialised for peer-to-peer overlay networks. The simulator takes as input an implementation of the Extended Specification of the system (provided by the Designer), the Behavioural Model (i.e., the Protocol Game PG), and the design objectives to achieve. *R-sim* supports a cycle-based simulation model. Specifically, at each cycle, every node executes the Extended Specification in turn. Correct nodes will never deviate from the correct execution of the system specification, while the behaviour of selfish nodes can change from one cycle to another according to the action that maximises their utility. To this end, the *R-sim* simulator performs a game-theoretic analysis of the Protocol Game PG. In particular, at each simulation cycle, *R-sim* proceeds as follows:

1. It creates a new instance of the PG, which reflects the current state of the interacting nodes (e.g., reputation values, available resources).
2. It conducts a game analysis of the PG to identify the rational strategy of each node.
3. It uses the solution of the game analysis to drive the execution of the system implementation.

The game-theoretic analysis performed at step (2) determines the possible steady states of PG, which are the equilibrium points (see Section 3.2). *RACOON* uses the Sequential Equilibrium (SE) solution [130], a refinement of the Nash Equilibrium for sequential game with imperfect

information. To find the SE of PG, *R-sim* relies on *Gambit*,[18] an open-source library of tools for solving non-cooperative games. Specifically, Gambit implements the algorithm by Koller, Megiddo and von Stengel [99], using linear programming. Consider for instance the Protocol Game created in the summary example of Section 5.3.4, and shown in Figure 21. In this game, the SE found by Gambit is the cooperative strategy profile *(offer, accept)*, which indicates that the best strategy of the two players is not to deviate from the correct execution of the system. In fact, any selfish strategy played by the player p_0 (e.g., omitting to send the `offer` message) would worsen her utility due to the punishment imposed by the *CEM*. The *RACOON* simulator uses this information and simulates the players' behaviour accordingly. Thus, if at a given cycle the equilibrium strategy of a selfish player p_0 is to perform a timeout deviation, for example, because her reputation is high enough to alleviate the severity of future punishments, then *R-sim* will skip any execution of the communication protocol of that node.

A single simulation allows verifying the selfish-resilience and performance guarantees offered by a given configuration of the accountability and reputation mechanisms. If the results are not satisfactory with respect to the design objectives set by the designer, the last step of the *RACOON* framework supports the automatic exploration of the configuration space of the *CEM*. A configuration candidate is an assignment of the parameters for the *R-acc* and *R-rep* mechanisms, summarised in Table 20.

***R-acc* accountability mechanism**		
Name	**Allowed Values**	**Definition**
ws: *witness set size*	$ws \in \mathbb{N}$	The number of witnesses associated with each node.
a_{pe}: *audit period*	$a_{pe} \in \mathbb{N}$	The time period (in seconds) between two executions of the audit protocol.
a_{pr}: *audit probability*	$a_{pr} \in [0, 1]$	The probability that a witness sends a log request during the execution of the audit protocol.
***R-rep* reputation mechanism**		
Name	*Range*	**Definition**
dp: *degree of punishment*	$dp \in \mathbb{R}_0^+$	The parameter of the *punishment function* (see Definition 5.9) that controls the intensity of a reputation decrease.
dr: *degree of reward*	$dr \in \mathbb{R}_0^+$	The parameter of the *reward function* (see Definition 5.10) that controls the intensity of a reputation increase.

Table 20: The configuration parameters of the *CEM*.

The exploration is an iterative process, which generates new candidates based on the evaluation of the previous ones until a configuration is found that satisfies all the Designer's objectives. *RACOON* explores the configuration space using a greedy constraint-satisfaction algorithm, which is guided by a set of simple rules that govern the updating of the configuration candidate. For instance, if when simulating a given configuration the overhead is already above the

18 Gambit: http://sourceforge.net/projects/gambit/

threshold fixed by the Designer, *RACOON* will not increase the number of witnesses, the probability of audit or the audit period in the next configuration to be explored as this would further increase the overhead [53, 76]. If no feasible solution is found after a pre-defined number of iterations (e.g., because the objectives are contradictory or too demanding), the framework stops the search, asking the Designer to improve the design manually or to relax the design objectives.

The tuning phase concludes the execution of the semi-automatic design and configuration process of cooperative systems supported by the *RACOON* framework. The outcome of *RACOON* is an automatic redesign of the system specification provided by the Designer, which integrates general and practical mechanisms (accountability and reputation) aiming to provide resilience against selfish nodes that try to contribute less than their fair share to the system. *RACOON* also supports the automatic tuning process of such mechanisms, with the goal of achieving a satisfactory level of selfish-resilience and performance set by the system designer. The selfish-resilience specification, as well as the finely tuned configuration of the cooperation enforcement mechanisms, can be used by the Designer as a reference for later development and deployment of the cooperative system.

5.5 EVALUATION

In this section, we demonstrate the benefits of using the *RACOON* framework to design selfish-resilient cooperative systems. First, we assess the *design effort* required by the system designer to specify a P2P live streaming protocol, along with a set of objectives she wants to achieve. Second, we assess the *effectiveness* of *RACOON* by comparing the quality of a configuration it finds with a set of *R-acc* configurations. Third, we assess the *accuracy* of the simulations performed by *RACOON* compared to a real implementation of this accountable live streaming system. Further, we evaluate the *performance* of *RACOON* by measuring the average time necessary to find satisfactory configurations in 30 different use cases. Finally, we show the degree of *re-usability* of the *RACOON* specification and simulation code by evaluating the effort required by the Designer to specify and simulate an anonymous communication protocol starting from the live streaming protocol.

5.5.1 *Design and development effort*

We show in this section, the effort necessary for the Designer to describe a cooperative system using *RACOON*. As a use case, we consider the P2P live streaming system described below.

P2P Live Streaming. The basic design of this system consists of a source node that disseminates video chunks to a set of nodes over a network. Periodically, each node sends the chunks it has received to a set of randomly chosen partners, and asks them for the chunks they are missing. Each chunk is associated with a playback deadline, which, if missed, would render a chunk unusable and the corresponding portion of the video unplayable. For the chunk

exchange, we use the three-phase gossip-based live streaming protocol (3P) studied by Guerraoui et al. [72] and depicted in Figure 22.

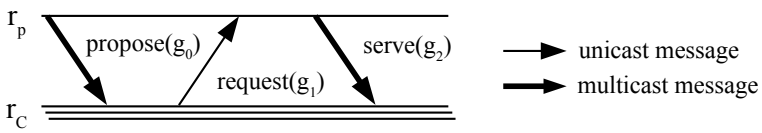


Figure 22: The sequence diagram of the chunk exchange protocol 3P, studied by Guerraoui et al. [72].

The functional specification of the protocol 3P involves two roles: the provider r_P proposes the set of chunks it has received to a set of consumers r_C, which in turn request any chunks they need. The protocol ends when r_P sends to the nodes playing the role r_C the requested chunks. Figure 23 illustrates the Protocol Automaton (PA) of the protocol 3P.

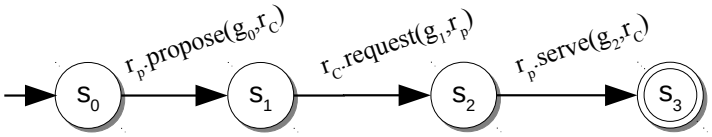


Figure 23: The Protocol Automaton of the chunk exchange protocol 3P.

We recall from Section 5.3.1 that a Protocol Automaton describes the interactions among individual nodes, regardless of the cardinality of the role they are currently playing. Therefore, the PA in Figure 23 defines the correct and separate executions of the protocol 3P between the node playing as r_P and each node playing as r_C.

In our experiment, we assume that the Designer wants to meet the following objectives:

O1: A deviation rate lower than 10%;

O2: A bandwidth overhead lower than 40%;

O3: A wrongful eviction rate lower than 10%.

Overall, the XML specification of protocol and objectives contains 43 Lines of Code (LoC).[19] In addition to writing this specification, the Designer has to develop a new module for the *R-sim* simulator, which implements the P2P live streaming system. The implementation classes of this module contain 500 LoC. This includes the implementation of the 3P protocol and the custom monitors to measure live-streaming metrics (e.g., the number of chunks transmitted/received).

5.5.2 *Meeting design objectives using RACOON*

Given the above specification and the corresponding simulation code, *RACOON* explores the space of possible configurations by running a set of simulations to find a configuration that satisfies the objectives set by the designer. To carry out these simulations, we configured the live streaming system using the parameters depicted in the column "Simulation" of Table 21.

19 The full specification is presented in Appendix B.

Parameter	Simulation	Experiment
Network size (nodes)	1000	100
Broadcast bandwidth (Kbps) [a]	674	674
Partner set size (nodes) [b]	7	6
Play-out delay (rounds) [c]	6	6
Bandwidth capacity (Kbps) [a]	1000	1000

[a] Same as by Guerraoui et al. [72].

[b] Slightly larger than the logarithm of the network size [96].

[c] Same as by Traverso et al. [166].

Table 21: Simulation and real deployment parameters.

The configuration proposed by *RACOON* is depicted in the last column of Table 22. We compare the performance of this configuration with the five FullReview configurations depicted in the same table. We selected these configurations by varying two parameters: the audit period and the degree of punishment. The first four configurations correspond to four combinations of low and high values of these parameters (referred to as L and H, respectively in the configuration names). These values are the lowest and highest values tested in the experiments of Section 4.2.3, respectively. Besides these combinations, we included the best configuration found in Section 4.2.3 (labelled M-M in Table 22) as it already satisfies the first two requirements set by the designer.

	L-L	L-H	H-L	H-H	M-M	*RACOON*
Audit period	5	5	30	30	15	5
Audit probability	1.0	1.0	1.0	1.0	1.0	0.5
Degree of punishment	0.5	3.0	0.5	3.0	1.5	1.0

Table 22: FullReview Configurations

Figure 24 shows the simulation results of the six configurations. The *RACOON* configuration is the only one that fulfils all the design objectives, which are depicted as horizontal dotted lines in the figure. Furthermore, this configuration provides up to 33% fewer deviations, 42% fewer wrongful evictions and 17% lower overhead than the others.

The results of this experiment show that *RACOON* can greatly facilitate the work of Designers in making a cooperative system meet the desired design objectives. In fact, the automatic tuning of the cooperation enforcement mechanisms performed by our framework can effectively replace the manual trial-and-error calibration required by the existing approaches from the literature (i.e., FullReview, in this experiment).

5.5.3 *Simulation compared to real system deployment*

To demonstrate the accuracy of *RACOON* simulations, we implemented a prototype of the live streaming protocol described above. We configured the accountability and reputation mechanisms using the parameters depicted in the last column of Table 22. Then, we deployed the

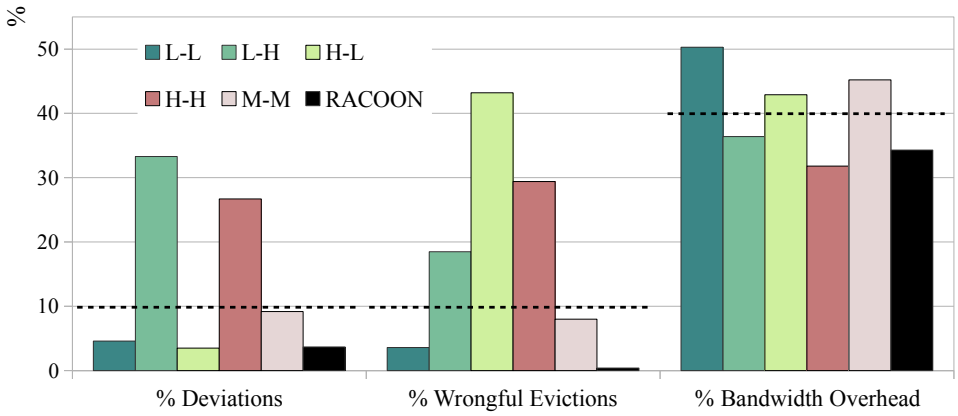


Figure 24: *RACOON* vs FullReview Configurations.

prototype on real machines of the Grid'5000 testbed.[20] Specifically, we ran 100 clients on 10 eight-core physical machines. Each machine is clocked at 2.5GHz with 32GB of RAM, and is interconnected with the others via a Gigabit switch. We performed our experiment in this environment in order to measure the impact of selfish nodes on the video chunk loss without the risk of fluctuating networking conditions. Otherwise, it would be always necessary to verify whether a deteriorated stream quality comes from selfish nodes or the transient network, which is tedious, imprecise, and time-consuming.

The column "Esperiment" of Table 21 describes our experimental settings. The only differences with the simulation settings are the lower number of nodes in the network and in the partner set. Note also that the bandwidth of each client is still capped to 1000 Kbps in uplink. In this experiment, we measure the chunk loss experienced by correct nodes as a function of the fraction of selfish nodes in the system.

Figure 25 presents the results of our evaluation. This figure contains a curve showing the impact of selfish nodes on traditional Gossip (i.e., without any accountability mechanisms) as well as the two curves for the system designed using *RACOON*. The "SIM - *RACOON*" curve is obtained using *RACOON* simulations, whereas the "G5K - *RACOON*" curve is obtained using the real deployment. From this figure, we observe that without accountability mechanisms, correct nodes experience 10% chunk loss with only 10% of nodes behaving selfishly, which prevents them from watching the video stream. Further this figure shows that the simulated curve and the real one, overlap up to the inclusion of 50% of selfish nodes in the system. Above this value, the curves still exhibit a comparable shape.[21] Finally, this figure shows that the configuration found by *RACOON* is effective — i.e., correct nodes watch a high quality video stream (video chunk loss lower than 3% [166]) — even when 90% of the network is composed of selfish nodes.

20 Grid'5000: http://www.grid5000.fr
21 Notice that the maximum difference between them is 0.43% ("SIM - *RACOON*" = 0.5%, "G5K - *RACOON*" = 0.93%), when the percentage of selfish nodes in the network is 90%.

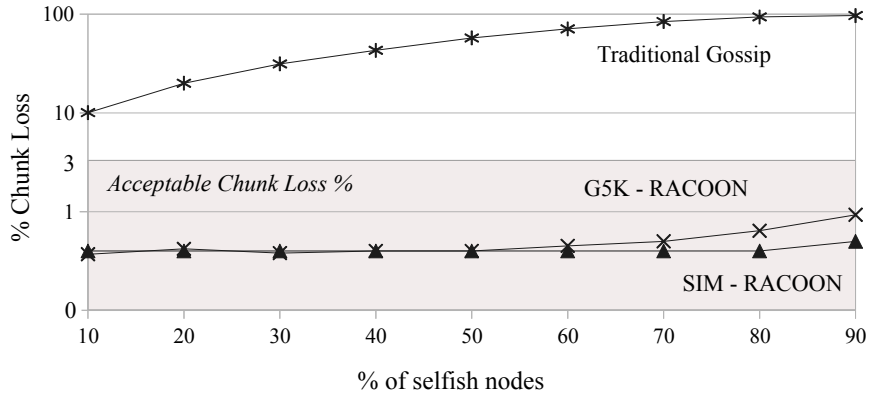


Figure 25: Simulation vs real deployment (logarithmic scale).

5.5.4 *Execution time*

To evaluate the time necessary for *RACOON* to find a satisfactory configuration, we performed the following experiment. First, we defined a set of 30 different scenarios in the live streaming application. Each scenario is a unique combination of the following elements: system objectives, simulation settings (e.g., number of nodes, bandwidth capacity, play-out delay), percentage of selfish nodes in the system, and message loss rate. Second, we measured the number of configurations that *RACOON* explores in each case before finding a satisfactory solution. Finally, we averaged these numbers over the total number of scenarios that have been considered. The results show that for each scenario, an average of 26 configurations are explored by *RACOON* before finding a satisfactory one (standard deviation 7, minimum configurations explored 8, maximum 41). Considering that each configuration corresponds to one executed simulation and that each simulation lasts approximately 42 seconds,[22] the exploration algorithm takes on average about 18 minutes to complete. This duration appears to be reasonable as all the activities performed by *RACOON* are done offline at design time. Finally, note that the execution time of the *RACOON* framework is entirely dominated by the *Configuration Evaluation* step, as the remaining steps take in average less than 30 milliseconds to be executed.

5.5.5 *Expressiveness*

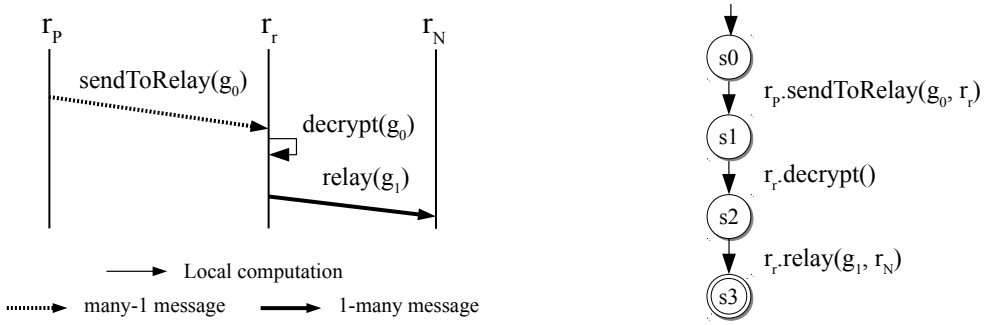
To illustrate the generality of our framework, we use *RACOON* to design the simple anonymous communication protocol described below.

Anonymous Communication. The system is based on a simplified version of the Onion Routing protocol for communication channel anonymity [70]. In Onion Routing, when a source node wants to send a message to a destination node, the source node builds a circuit of voluntary *relay* nodes. Circuits are updated periodically, and relays can participate in multiple circuits at the same time. To protect a message, the source encrypts it with the public

22 Average value over 1000 simulations, run on a 2.8 GHz machine with 8 GB of RAM.

key of the destination. Furthermore, to protect the communication channel, the source uses the public key of each relay node in the circuit to encrypt the address of the next relay node. The resulting message is called an *onion*. A relay uses its private key to decrypt one layer of the onion and contributes some of its bandwidth to forward the resulting message to the next relay, until the message eventually reaches its destination.

Figure 26a illustrates the protocol enabling the forwarding of onions. In this protocol, each relay R: (i) receives onion messages from their predecessor in the circuit (PR); (ii) decrypts the external layer of each onion; (iii) forwards the resulting onions to their respective successor NR.



(a) Forwarding protocol

(b) Protocol automaton

Figure 26: Onion Forwarding Protocol.

To design a selfish-resilient onion forwarding protocol using *RACOON*, the Designer follows the same steps we have seen to design the live streaming system. First, she provides the specification of the system. This specification describes three roles, which are the relay r_r, and its previous (r_P) and next (r_N) hops. Note that r_P and r_N have cardinality $\geqslant 1$, because they represent a set of relay nodes. The Protocol Automaton of the forwarding protocol can be modelled as in Figure 26b. The protocol includes a communication method and a computation method. The communication methods *sendToRelay* and *relay* send messages that carry the single onion to forward. The method *decrypt* is a decryption operation (i.e., local computation). A selfish relay r_r that aims at saving bandwidth strategically drops onions that are not intended for it.

Designing this system using *RACOON* only required writing 39 lines of XML specification and 350 LoC for the implementation of the simulation module.[23] The full XML specification is presented in Appendix B.

We simulate an anonymous network of 1000 nodes, with different fractions of selfish participants. Each node is the source of 50 onions. Every 5 rounds, a source node creates a circuit of 3 node relays, which are randomly selected. This setting is similar to other state-of-the-art protocols (e.g., [54]). We set the period of the audit to 10 rounds, with a probability of 0.5. In the case of a positive audit, the misbehaving node incurs a punishment with a degree of 1. Measurements in simulations are averaged over 50 independent runs, which ensures a standard deviation less than 1%. As the evaluation metric, we consider the *onion loss rate* (i.e., the aver-

23 We assumed the same design objectives set for the P2P live streaming use case.

age percentage of onions missed by the destination node) of nodes as a function of the fraction of selfish nodes in the system.

Figure 27 presents the results of our evaluation. The figure contains two curves. The "Traditional Anon. Comm." logarithmic curve shows the results of an anonymous communication system designed without any mechanism to prevent deviations. Unsurprisingly, in this case, the onion loss rate increases significantly as the number of selfish nodes increases, up to above 90% of loss onions when one out of two nodes is selfish. On the contrary, the onion loss rate of the anonymous communication system designed using *RACOON* ("SIM - Racoon" curve) increases slowly and never exceeds 7% as the number of selfish nodes increases.

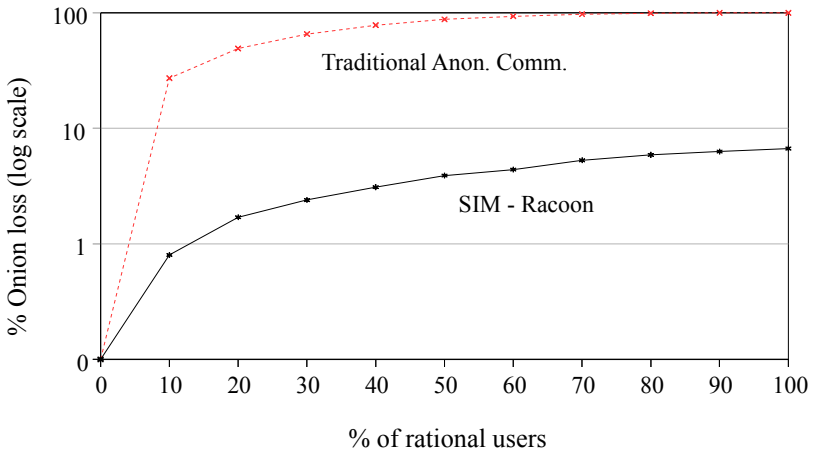


Figure 27: Onion loss rate as a function of the percentage of selfish nodes in the system (logarithmic scale).

5.6 SUMMARY

In this chapter, we presented *RACOON*, an integrated framework for the design and configuration of cooperative systems resilient to selfish nodes. *RACOON* consists of two phases: the assisted design of the system and the performance-oriented tuning of its parameters.

The first phase begins with the description of the protocols of the system to make resilient to selfish nodes. *RACOON* includes a specification model, the Protocol Automaton, to facilitate the designers in this task. To enforce cooperation, the framework extends the system specification with two general, scalable and distributed mechanisms, namely, an accountability system for auditing the nodes' behaviour and a reputation mechanism to assign incentives depending on the audit results. *RACOON* enables the later evaluation of these mechanisms by further extending the system specification with selfish behaviours. For this purpose, it includes an algorithm for the automatic injection into the Protocol Automaton of three types of selfish deviations, chosen based on the survey on selfishness presented in Chapter 2. The output is a static representation of (cooperative and selfish) behaviours that might be played in the system. To predict which behaviours are more likely to be played by selfish nodes, *RACOON* relies on classical game theory: it transforms the extended Protocol Automaton into a game, thereby providing the behavioural model to simulate selfishness in the next phase of the framework.

The goal of the tuning phase is to configure the cooperation enforcement mechanisms according to a list of selfishness-resilience and performance objectives set by the designer. Tuning involves an iterative two-step refinement process, which alternates evaluation with the tuning of the configuration parameters. The evaluation is done using the game theory-driven simulator developed for *RACOON*. The simulator performs an automatic game analysis of the behavioural model, thereby simulating the interactions of selfish nodes into a software implementation of the Protocol Automaton provided by the designer. If the evaluated configuration fails to meet the design objectives, the framework explores the configuration space of the accountability and reputation mechanisms automatically, until it finds a satisfactory solution or after a set number of iterations.

We demonstrated the benefits of using the *RACOON* framework by designing two selfish-resilient cooperative systems: a P2P live streaming system and an anonymous communication system based on the onion routing protocol [70]. Experiments in simulation and (at a smaller scale) in a real deployment on Grid'5000 showed that the live streaming system configuration chosen by *RACOON* allows correct nodes to visualize a stream of good quality in the presence of selfish nodes, as well as to meet other performance requirements set by the designer (e.g., limiting the bandwidth overhead to a fixed threshold). Also, these experiments proved that the *RACOON* simulations are accurate compared to the performance of the corresponding real system, with a maximum absolute difference of less than 1%. Finally, we evaluated the performance of the framework in finding a satisfactory configuration in the live streaming use case, showing that in the 30 scenarios considered this task took on average less than 20 minutes.

The *RACOON* framework is the central contribution of this thesis (C.2). It provides a general and semi-automatic methodology, along with its software implementation, to address three research challenges: (D.2) it facilitates the work of designers in enforcing practical and fine-tuned mechanisms to foster cooperation in their particular system; (A.3) and (D.3) it simplifies the development, evaluation, and use of behavioural models of selfish nodes for testing the selfishness-resilience of cooperative systems. Related to the last two challenges, Table 23 positions the automatic tools included in *RACOON* in relation to the state-of-the-art approaches for analysing selfishness in cooperative systems presented in Chapter 3. In particular, the table shows that *RACOON* offers more domain-specific support to selfishness analysis than experimental approaches, while still being more usable than pure analytical tools like game theory.

Nevertheless, some limitations in *RACOON* were found, revealing space for improvement. In particular, we note three major issues in the current version of the framework, both theoretical and practical: (i) the selfish behaviours that can be generated during the selfishness injection step are fixed and poorly customizable, (ii) the behavioural model builds on the assumptions of classical game theory, notably, the notion of perfect rationality, which requires making some unrealistic assumptions on the nodes' capabilities, and (iii) the simulator at the heart of the tuning phase is custom-built. The second issue explains the lowest score of *RACOON* with respect to the refinement level (column *Ref* in Table 23), whereas the last issue determines its lower usability than the other well-established simulation tools.

Providing solutions to the above issues is the aim and focus of the next chapter.

	General and cooperative systems criteria [a]					Selfishness criteria [b]				
Approaches	**Usa**	**Rep**	**Ref**	**Sc**	**He**	**Ra**	**D**	**F**	**M**	**C**
Analytical										
Game theory [130]	●○○○○	●●●●●	●○○○○	Unlimited	Controllable	✓	✓	✓	✓	✓
Experimental: real experiments										
Grid'5000 [37]	●●●○○	●●●○○	●●●●●	1000	Fixed	x	✓ [c]	x	x	x
PlanetLab [41]	●●○○○	●●●○○	●●●●●	1000	Fixed	x	✓ [c]	x	x	x
Experimental: emulation										
ModelNet [168]	●●●●○	●●●●●	●●●●○	100	Controllable	x	✓ [c]	x	x	x
Emulab [173]	●●●●○	●●●●●	●●●●○	1000	Controllable	x	✓ [c]	x	x	x
Experimental: simulation										
NS-2 [3]	●●●●○	●●●●●	●●●○○	100	Controllable	x	✓ [c]	x	x	x
OMNeT++ [4]	●●●●○	●●●●●	●●●○○	10^5	Controllable	x	✓ [c]	x	x	x
PeerSim [122, 128]	●●●●●	●●●●●	●●○○○	10^6	Controllable	x	✓ [c,d]	x	x	x
Analytical + Experimental: simulation										
RACOON	●●●○○	●●●●●	●○○○○	10^4	Controllable	✓	✓	✓	x	x

[a] Usa = usability, Rep = reproducibility, Ref = refinement (inverse of abstraction), Sc = scalability, He = heterogeneity.
[b] Ra = rationality, D = defection , F = free-ride, M = misreport, C = collusion.
[c] Implemented as faults or churn.
[d] Highly controllable using the RCourse library [122].

Table 23: Comparison between *RACOON* and the existing approaches for selfishness analysis.

6

THE RACOON++ FRAMEWORK: RACOON MEETS EVOLUTION

In this chapter, we present *RACOON++*, an enhanced and extended version of the framework described in the previous chapter. Capitalising on the modularity of the *RACOON* methodology, in *RACOON++* we introduce new models and tools to address a number of limitations as well as to provide new features. Figure 28 provides an overview of the new framework.

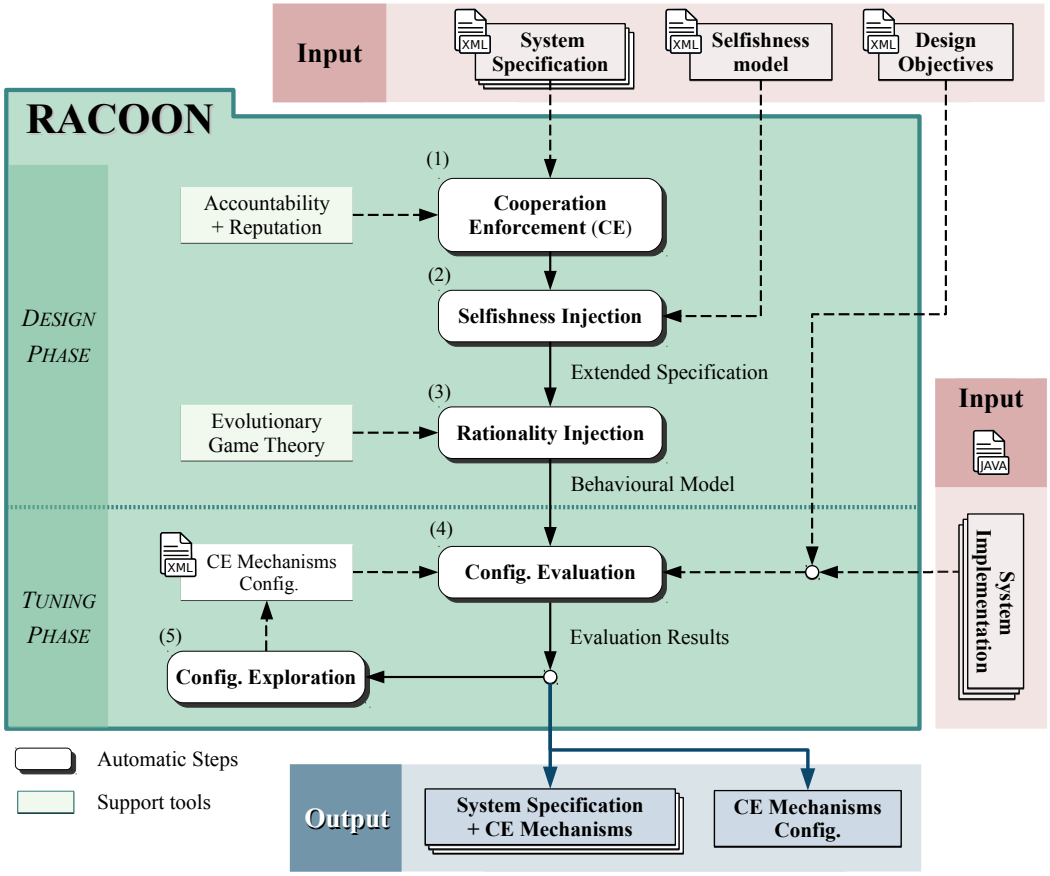


Figure 28: The *RACOON++* framework overview.

RACOON++ have different inputs than the previous version of the framework. In particular, *RACOON++* provides a more expressive model for the system specification, along with a richer model to describe the design objectives to meet. Furthermore, the new framework provides the system designer with a simple yet expressive specification model to define the utility function and the behaviour of selfish nodes, which in *RACOON* were predefined and fixed for all application scenarios. This model, called the *Selfishness Model*, shapes the utility function of a node by assigning costs and benefits to specific actions of the communication protocols, and

parametrizes some aspects of selfish behaviours (*who* deviates, from *which action*, with *what type* of deviation). The introduction of this input affects the operation of the *Selfish Deviation Generation* component of the original *RACOON* framework, requiring the definition of a new algorithm to generate selfish behaviours given a system specification and a selfishness model.

As a second major contribution of *RACOON++*, we model the behaviour of selfish nodes using Evolutionary Game Theory (EGT) [172] instead of the traditional non-cooperative approach adopted in the previous version of the framework. Using EGT, we can relax the strict assumption of perfect rationality of the nodes and consider them as active learners, who can adjust their strategy over time in response to repeated observations of their own and others' utilities. Such learning and adaptation processes fit better with the computational and decisional capabilities of real nodes [120, 171, 183]. Furthermore, as noted by Palomar et al. [142], an evolutionary approach is more appropriate for modelling the dynamic behaviour of cooperative systems.

We integrate the *RACOON++* functionalities with the state-of-the-art P2P simulator PeerSim [92]. To the best of our knowledge, the simulator we developed in *RACOON* was the first P2P simulator able to dynamically simulate selfish behaviours, based on GT analysis. However, like all custom built simulators, it has neither the maturity, nor the efficiency, nor the acceptance of well-known tools like PeerSim [24].

To summarise, the primary contributions of this chapter are the following:

- We present a declarative model for defining the utility function of a node and parametrizing some aspects of selfish behaviours.
- We develop a more suitable model for reasoning on the strategic and dynamic interactions of nodes, based on *EGT*.
- We integrate the state-of-the-art P2P simulator PeerSim into our framework, for scalability and reproducibility purposes.
- We propose a new automatic configuration method for the accountability and reputation mechanisms used by the framework.
- We assess the design effort and the effectiveness of using *RACOON++* by designing three cooperative systems: a live streaming system [72], a load balancing protocol [92], and an anonymous communication system based on Onion Routing [70].

We released a Java implementation of the *RACOON++* framework as an open-source project. The code and data to reproduce the experiments presented in this chapter are freely available on GitHub: `https://github.com/glenacota/racoon`.

Roadmap. The remainder of this chapter is organised as follows. In Section 6.1 we provide a quick overview of the *RACOON++* framework. Section 6.3 and Section 6.4 present an updated version of the design and tuning phases of the framework, which accounts for the new contributions. We report a performance evaluation of *RACOON++* in Section 6.5, and we conclude the chapter in Section 6.6.

The contents of this chapter are currently under review for the IEEE Transactions on Dependable and Secure Computing (TDSC).

6.1 OVERVIEW

RACOON++ is an extension of the *RACOON* framework presented in Chapter 5. The framework aims at supporting system designers (hereafter "Designer", for brevity) in building a selfish-resilient cooperative system that meets desired design objectives. As depicted in Figure 28, the operation of *RACOON++* consists of two phases: the assisted design of the system and the objective-oriented tuning of its parameters. These phases share the same vision and goals of the previous version of the framework, as we briefly show in the following.

The Designer initiates the *design* phase providing a state-machine representation of the communication protocols composing the system, which *RACOON++* integrates with a slight modification of the two Cooperation Enforcement Mechanisms (CEM) used in *RACOON*, namely, accountability and reputation mechanisms (Step (1) in Figure 28). In Step (2), *RACOON++* proceeds with the injection of selfish behaviours into the system specification, which results in the *Extended Specification* of the system. For a better control over the injection process, we provide the Designer with a *Selfishness Model* to describe the preferences and capabilities of selfish nodes. Finally, in Step (3), *RACOON++* transforms the Extended Specification into a game model of Evolutionary Game Theory (EGT), which provides the mathematical framework to describe the *Behavioural Model* of the selfish nodes under consideration.

The goal of the *tuning* phase is to find a configuration setting for the *CEM* that makes the system meet a list of *Design Objectives* set by the Designer. As in *RACOON*, tuning is an iterative refinement process consisting of game-theory driven simulations (Step (3)) to evaluate a configuration candidate, and an exploration process to traverse the configuration space of the *CEM* (Step (4)). Once *RACOON++* has found a configuration that meets the design objectives, the Designer can proceed with the implementation of her system.

6.2 ILLUSTRATIVE EXAMPLE: THE S-R-R PROTOCOL

In the next sections, to support the description of the *RACOON++* framework, we use the simple communication protocol *S-R-R* (*Search, Request & Response*) shown in Figure 29 as an illustrative example. In the *S-R-R* protocol, a node r_0 queries other nodes for some desired resources (e.g., files). To this end, r_0 sends a query message g_0 to a group of nodes collectively named R_1 (the capital letter denotes a set of nodes). Each node in R_1 processes the query and replies with the list of available resources (message g_1). Upon receiving the list, r_0 sends a new message g_2 to R_1, requesting (a subset of) the resources listed in g_1. Finally, each node in R_1 sends the requested data (message g_3).

6.3 RACOON++ DESIGN PHASE

In this section, we introduce the new inputs of the design phase of *RACOON++*, and we describe the updates in the accountability and reputation mechanisms adopted. Then, we present the modifications in the algorithm to generate selfish deviations, along with the new Behavioural Model based on Evolutionary Game Theory.

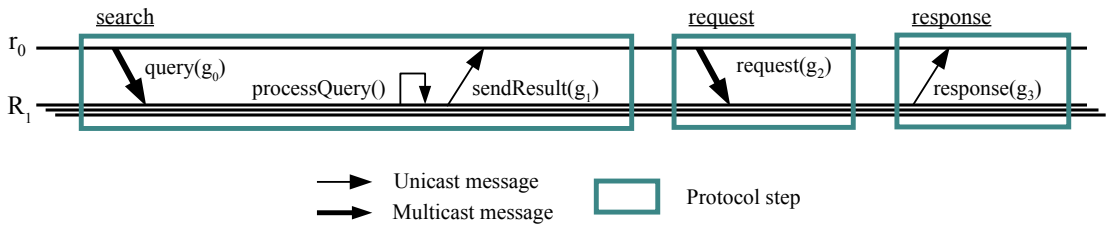


Figure 29: The *S-R-R* protocol between nodes r_0 and R_1.

6.3.1 *Input of the Design phase*

The inputs of the design phase are (i) the *functional specification* of the functionalities of the cooperative system that should be made resilient to selfish behaviours, and (ii) the *selfishness model* adopted by selfish nodes.

6.3.1.1 *Functional specification*

The functional specification describes the correct behaviour of nodes by means of a state-machine representation of the communication protocols of the system. In *RACOON++*, we extend the Protocol Automaton PA defined in Chapter 5 (Definition 6.5) with new information about the role, transition, message, and content elements. We denote the new Protocol Automaton as PA^{++}.

A role determines the responsibilities of a party (whether a node or a group of nodes) and constrains the actions that the party is allowed to execute in a protocol run. Every PA^{++} has at least two types of roles: the provider of a resource or service, and the requester. However, other types are also possible (e.g., brokers, auditors, recommenders). For example, in the *S-R-R* protocol, there are two roles: r_0 and R_1, where r_0 is a requester, and R_1 corresponds to a given number of potential providers.

Definition 6.1 (Role in *RACOON++*)**.** A *role* $r \in R^{++}$ is a triple $\langle \texttt{rId}, \texttt{cardinality}, \texttt{rType} \rangle$, with:

- `rId`: the alphanumeric identifier of the role,
- `cardinality`: the number of nodes represented by `r`. It can either be a single number or a variable number designated by a greater than ($>$), greater than or equal to ($\geqslant$), less than ($<$), less than or equal to ($\leqslant$) conditions, and
- `rType`: specifies whether `r` is a `provider`, `requester`, or has `other` types of role.

A transition corresponds to a protocol step, i.e., the set of method calls that determine the next protocol state. *RACOON++* supports the definition of three types of transitions: *abstract*, *communication*, and *computation*. An abstract transition groups many method calls into a single "black box" transition, which may simplify the protocol representation by hiding some

implementation details. On the contrary, the remaining transition types allow to characterise a (communication or computation) method that triggers the transition. For example, the protocol *S-R-R* has three transitions: *search* (abstract), *request* and *response* (communication).

Definition 6.2 (Transition in *RACOON++*)**.** A *transition* $t \in T^{++}$ is a quadruple $\langle tId, state1Id, state2Id, methodId \rangle$, with:

- tId: the alphanumeric identifier of the transition,
- $state1Id$ and $state2Id$: identify the source and target states $\in S$ of t, and
- $methodId$: identifies the method $m \in M$ executed in t. It is defined only for non abstract transitions, null otherwise.

A message conveyed by a communication method (Definition 5.4 in the previous chapter) is formalised as the quadruple below.

Definition 6.3 (Message in *RACOON++*)**.** A *message* $g \in G^{++}$ is a quadruple $\langle gId, senderId, receiverId, contentId \rangle$, with:

- gId: the alphanumeric identifier of the message,
- $senderId$ and $receiverId$: identify the sender and receiver roles $\in R^{++}$ of g, and
- $contentId$: identifies the content $c \in C^{++}$ carried by g.

The content sent via a communication message in the Protocol Automaton can be either a single data unit (e.g., a binary file) or a collection (e.g., a list of integers) of data units. With respect to the definition provided in the previous chapter, we removed the information about the memory size of a single data unit (element $cSize$). In *RACOON*, we used this information to calculate the communication costs contributing to the utility function of selfish nodes (see Definition 5.19); by contrast, in *RACOON++* we provide a more expressive means to define costs and benefits of selfish nodes, which we will describe in the next section.

Definition 6.4 (Content in *RACOON++*)**.** A *content* $c \in C^{++}$ is a triple $\langle cId, cType, cLength \rangle$, with:

- cId: the alphanumeric identifier of the content,
- $cType$: provides information about the data type,[1] and
- $cLength$: specifies the number of data units that comprise the content c. It can either be a single number or a variable number designated by a greater than ($>$), greater than or equal to ($\geqslant$), less than ($<$), less than or equal to ($\leqslant$) conditions.

1 Defined by the XML Schema type system.

Finally, we can formalise the Protocol Automaton used in *RACOON++* as follows.

Definition 6.5 (Protocol Automaton in *RACOON++*)**.** A *Protocol Automaton* PA^{++} in *RACOON++* is a tuple $\langle R^{++}, S, T^{++}, M, G^{++}, C^{++}, K \rangle$, with:

- R^{++}: finite, non-empty set of *roles*,
- S: finite, non-empty set of *states* (the same as in *RACOON*, see Definition 5.2),
- T^{++}: finite set of *transitions*,
- M: finite set of *methods* (the same as in *RACOON*, see Definition 5.4),
- G^{++}: finite set of *messages*,
- C^{++}: finite set of *contents*, and
- K: finite set of *constraints* on contents (the same as in *RACOON*, see Definition 5.7).

Figure 30 shows the state diagram of the *S-R-R* protocol. The label on a transition provides information about the method that triggers the transition, and about the message sent. For example, the label between states s_1 and s_2 indicates that role r_0 invokes the communication method `request`, which conveys the message g_2 to role R_1. The role indicated in the label of an abstract transition, such as the one between states s_0 and s_1 in Figure 30, is the one that executes the first method encapsulated in the abstract transition.

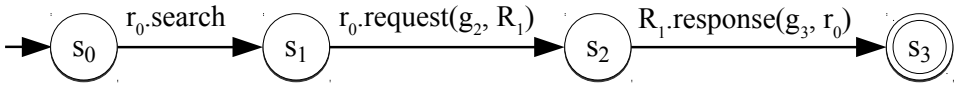


Figure 30: The Protocol Automaton of the *S-R-R* protocol.

Notation. *For simplicity, in the remainder of this chapter we use the notation* PA *instead of* PA^{++} *to refer to the updated Protocol Automaton used in RACOON++. Similarly, for the set of roles* R*, transitions* T*, messages* G*, and contents* C *(instead of* R^{++}*,* T^{++}*,* G^{++}*, and* C^{++}*, respectively).*

As in the *RACOON* framework, protocol automata in *RACOON++* are encoded in an XML-based format.[2] The Designer specifies a new Protocol Automaton as an XML document, which also includes the specification of the Selfishness Model and design objectives presented later in this chapter. For example, Listing 3 shows the XML representation of the PA described in Figure 30.

2 The XML Schema can be found in Appendix A.

```
<racoon name="SearchRequestResponse">
<protocol_automaton>
  <roles>
  <role id="r0" cardinality="1" type="requester" />
  <role id="R1" cardinality=">=1" type="provider" />
</roles>
<states>
  <state id="s0" roleId="r0" type="initial" />
  <state id="s1" roleId="r0" type="intermediate" />
  <state id="s2" roleId="R1" type="intermediate" />
  <state id="s3" roleId="r0" type="final" />
</states>
<transitions>
  <transition id="search" state1Id="s0" state2Id="s1" methodId="" />
  <transition id="t1" state1Id="s1" state2Id="s2" methodId="request" />
  <transition id="t2" state1Id="s2" state2Id="s3" methodId="response" />
</transitions>
<methods>
  <method id="search" messageId="" />
  <method id="request" messageId="g2" />
  <method id="response" messageId="g3" />
</methods>
<messages>
  <message id="g2" contentId="c2" senderId="r0" receiverId="R1" />
  <message id="g3" contentId="c3" senderId="R1" receiverId="r0" />
</messages>
<contents>
  <content id="c2" type="integer" size="4" length=">=0" />
  <content id="c3" type="integer" size="4" length=">=0" />
</contents>
<constraints>
  <constraint id="k0" content1Id="c3" type="equal" content2Id="c2" />
</constraints>
</protocol_automaton>
<selfishness_model>
  <!-- specification of the selfishness model (subsection below) -->
</selfishness_model>
<design_objectives>
  <!-- specification of the design objectives (Section 6.4.1) -->
</design_objectives>
</racoon>
```

Listing 3: The XML document that specifies the Protocol Automaton described in Figure 30.

6.3.1.2 *Selfishness Model*

The selfishness model carries the information about the economic drivers of a node. It does so by specifying the utility (i.e., benefits and costs) that a node obtains in participating in the cooperative system; furthermore, it indicates the possible deviations from the node's correct execution. A definition of selfishness model is provided below.

Definition 6.6 (Selfishness Model)**.** A *Selfishness Model* SM is a couple $\langle V, D \rangle$, with:

- V: finite set of *valuations* (Definition 6.7), and
- D: finite set of selfish *deviations* (Definition 6.10).

VALUATIONS. Valuations describe the contributions to the overall utility of a certain behaviour. As in *RACOON*, the utility that a node receives from participating in the system is given by the benefit obtained by consuming resources and the cost of sharing resources. A

valuation specifies this information at the granularity of transitions and messages of a Protocol Automaton PA.

Definition 6.7 (Valuation)**.** A *valuation* $v \in V$ is a tuple $\langle vId, scope, scopeId, roleId, benefit, cost \rangle$, with:

- vId: the alphanumeric identifier of the valuation,
- $scope$: specifies whether v applies to a `transition` or a `message`,
- $scopeId = \{t.tId : t \in T\} \cup \{g.gId : g \in G\}$: identify a transition or a message in the PA,
- $roleId$: the identifier of the role $r \in R$ associated to the valuation, and
- $benefit$ and $cost$: the numerical values $\in \mathbb{N}$ that quantify the benefit and cost.

In practice, a valuation specifies the costs and benefits obtained by a given role when a certain transition takes place or a particular message is sent or received. We evaluate the contribution of a given valuation to the overall utility as defined below.

Definition 6.8 (Valuation function)**.** A valuation function $\upsilon : V \to (Z)$ is defined as:

$$\upsilon(v) = \begin{cases} v.benefit - v.cost, & \text{if } v.scope = \text{“transition”} \quad \text{(a)} \\ r.cardinality \cdot c.cLength \cdot (v.benefit - v.cost), & \text{if } v.scope = \text{“message”} \quad \text{(b)} \end{cases}$$

with:

- r: the receiver of the message $g \in G$ such that $g.gId = v.scopeId$, corresponding to the role $r \in R$ such that $r.rId = g.receiverId$, and
- c: the content $\{c \in C \mid c.cId = g.contentId\}$.

As an example, consider the *search* transition of the *S-R-R* protocol. It is reasonable to expect that role r_0 receives more benefit than cost from the transition because the node will eventually receive useful information. This consideration can be expressed by the valuation $\langle v_0, \text{“}transition\text{”}, search, r_0, 10, 1 \rangle$, which results in a contribution to the utility of $\upsilon(v_0) = 9$ (case (a), in Definition 6.8). Note that another system designer may have valued the same transition differently, according to her expertise and knowledge of the system.

By contrast, to evaluate the contribution to the overall utility of a valuation that applies to a message, Definition 6.8 presents a formula (case (b)) that accounts for the cardinality of the receiver role of the message as well as the number of data units comprising the delivered content. The rationale for this formula is based on the observation that costs and benefits of a message are usually proportional to the number of data units transmitted or received (e.g., the communication costs of a message depends on its length and number of recipients). Consider, for instance, the *request* transition of the *S-R-R* protocol, which involves the transmission of message g_2 to role R_1. Let c_2 be the content transmitted by g_2, and let $v_1 = \langle v_1, \text{“}message\text{”}, g_2, r_0, 5, 1 \rangle$

be the valuation associated with g_2. Then, the contribution of v_1 to the overall utility is: $v(v_1) = R_1.cardinality \cdot c_2.cLength \cdot 4$. Note that it is also possible to define a valuation associated to g_2 that specifies benefits and costs of the receiver R_1 of the message; for instance, $v_2 = \langle v_2, \text{“}message\text{”}, g_2, R_1, 5, 1\rangle$).

SELFISH DEVIATIONS. The Selfishness Model of *RACOON++* allows the Designer to specify where in the Protocol Automaton selfish deviation can occur, along with information about their type and degree. The types of deviations supported by the framework are the same proposed in *RACOON*, namely, timeout, subset, and multicast deviations (see Section 5.3.3). The degree of a selfish deviation indicates the distance from the correct behaviour, as defined next.

Definition 6.9 (Deviation degree)**.** The $degree \in [0, 1]$ of a selfish deviation specifies the intensity of a deviation as follows:

- The $degree$ of a timeout deviation is always equal to 1.
- The $degree$ of a subset deviation specifies the fraction of data units comprising a message content that is to be dropped.
- The $degree$ of a multicast deviation specifies the fraction of the intended receivers of a multicast message to whom the message is not to be sent.

According to the above definition, timeout deviations can only occur to the maximum degree. This is because there is no gradation in the intensity of timing out a prescribed action: when a timeout occurs the protocol is stopped.

We can now present a definition of selfish deviation in the SM.

Definition 6.10 (Deviation)**.** A selfish *deviation* $d \in D$ is a tuple $\langle dId, transitionId, dType, degree\rangle$, with:

- dId: the alphanumeric identifier of the deviation,
- $transitionId$: identifies the transition of the Protocol Automaton that is the subject of the deviation d. The wildcard value “*” specifies that all transitions in PA are subject of d,
- $dType$: specifies whether d is a `timeout`, `subset` or a `multicast` deviation, and
- $degree$: the deviation degree (Definition 6.9).

For instance, $\langle d_0, t_2, timeout, 1\rangle$ describes the behaviour of a selfish node that never replies to a request (see Figure 30). As another example, suppose the Designer wants to account for selfish nodes that only send half of the content in any message exchange of the *S-R-R* protocol (e.g., half of the requested resources). The selfish deviation $\langle d_1, *, subset, 0.5\rangle$ represents this behaviour. As a final example, let the Designer want to describe a selfish behaviour such that selfish nodes send one-fourth of the requested contents only to half of the intended recipients.

Such combination of subset and multicast deviations can be described by defining a pair of deviations, e.g., $\langle d_2, *, \texttt{subset}, 0.25\rangle$ and $\langle d_3, *, \texttt{multicast}, 0.5\rangle$.

To conclude, notice that different types of transition (abstract, communication, or computation) are subject to different types of deviation. For example, a computation transition is not affected by subset or multicast deviations because there is no message exchange. Table 24 lists the possible combinations of deviation and transition types, showing the combinations that may lead to an actual deviation ("✓") and those that cannot be applied ("χ"). Inapplicable deviations will be ignored by the *RACOON++* framework during the selfishness injection step.

Transition type	**Deviation type**	*Applicable?*	**Brief description**
Abstract	Timeout	✓	The node terminates the protocol before the execution of the abstract transition.
Abstract	Subset	χ	Cannot be applied because communication methods may be encapsulated into the transition.
Abstract	Multicast	χ	Cannot be applied because communication methods may be encapsulated into the transition.
Communication	Timeout	✓	The node terminates the protocol before the execution of the communication method.
Communication	Subset	✓	The node sends a subset of the correct message content.
Communication	Multicast	✓	The node sends a message to a random subset of the legitimate recipients.
Computation	Timeout	✓	The node terminates the protocol before the execution of the computation method.
Computation	Subset	χ	Cannot be applied because the transition involves no message exchange.
Computation	Multicast	χ	Cannot be applied because the transition involves no message exchange.

Table 24: Selfish deviations .

The Designer specifies the Selfishness Model in the same XML document she used to specify the Protocol Automaton. Listing 4 reports the specification of some of the valuations and selfish deviations presented as examples in this section.

```
<selfishness_model>
  <valuations>
    <valuation id="v0" scope="transition" scopeId="search" roleId="r0"
                                                    benefit="10" cost="1" />
    <valuation id="v1" scope="message" scopeId="g2" roleId="r0"
                                                    benefit="5" cost="1" />
    <valuation id="v2" scope="message" scopeId="g2" roleId="R1"
                                                    benefit="1" cost="1" />
  </valuations>
  <deviations>
    <deviation id="d0" transitionId="t2" type="timeout" degree="1" />
    <deviation id="d2" transitionId="*" type="subset" degree="0.25" />
  </deviations>
</selfishness_model>
```

Listing 4: The XML document that specifies the Selfishness Model and the Protocol Automaton.

6.3.2 *Cooperation enforcement*

The first automatic step of the design phase of *RACOON++* is the integration of the Cooperation Enforcement Mechanisms into the functional specification provided by the Designer. The *CEM* includes protocols (i) to enforce accountability, (ii) to form, aggregate, and disseminate reputation values, and (iii) to ensure that requester nodes can get the requested resource or service with a probability proportional to their reputation. In particular, if the reputation of a node hits the lower bound, no other node will accept its requests, thus preventing the node from receiving any benefit from the system.

The two *CEM*s used in *RACOON++* are discussed hereafter.

ACCOUNTABILITY MECHANISM. *RACOON++* can detect misbehaviours and assign nodes non-repudiable responsability for their actions by relying on a refinement of the *R-acc* accountability mechanism presented in the previous chapter (Section 5.3.2). The main differences of the current version, named *R-acc++*, with respect to the one used in *RACOON* involve the challenge/response and evidence transfer protocols, as described below:

Challenge/response protocols: deal with nodes that do not respond to messages as provided in PA or in *R-acc++*, allowing certain tolerance for correct nodes that are slow or suffering from network problems (e.g., message loss). Specifically, if a node i has been waiting too long for a given message from another node j, i indicates the suspect state for j (see Section 4.2.2), and creates a challenge for it. In the previous *R-acc* mechanism, nodes communicate only with non-suspected nodes. *R-acc++* adopts a more tolerant approach: while in the suspect state, the probability of j to communicate with i is locally decreased by a fixed amount, until j responds to the challenge and gets trusted again.

Evidence transfer protocol: *R-acc++* does not include the evidence transfer protocol used in *R-acc*. The same goal of ensuring that faulty nodes are eventually exposed by all correct nodes in the system is accomplished by the reputation mechanism described next.

RACOON++ includes a Protocol Automaton specification for each protocol. The Designer can refer to these specifications when writing the Selfishness Model, to define valuations and deviations also for *R-acc++*, and test whether accountability still holds when this mechanism is enforced by selfish nodes.

For the sake of completeness, we report in Figure 31 the result of the integration of the Protocol Automaton of the *S-R-R* protocol (Figure 29) and the commitment protocol of *R-acc++*.

REPUTATION MECHANISM. The *CEM* used in *RACOON++* includes the reputation system *R-rep* designed for *RACOON* and described in Section 5.3.2. In a nutshell, the reputation of a node is the summary of its history of behaviours, which is extracted by the accountability system. Cooperation leads to a good reputation, while selfish behaviours lead to a bad reputation. The difference with the previous framework lies in the use of the reputation value to encourage nodes to cooperate.

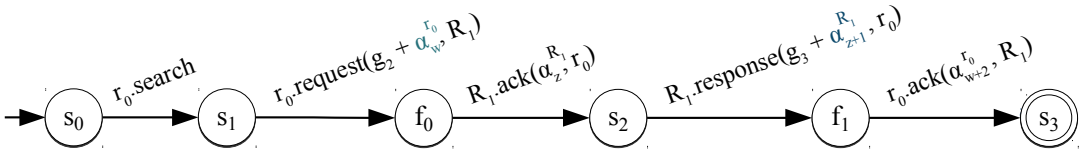


Figure 31: The integration between the commitment protocol of *R-acc++* with the *S-R-R* protocol shown in Figure 30.

In *RACOON*, reputation was used to trigger the eviction of the node from the system. By contrast, the *CEM* of the *RACOON++* framework imposes an allocation regime such that the probability of a node receiving a service or a resource in the future is proportional to its current reputation. In particular, a node with zero reputation can still act as a service provider, because the *CEM* prevents only those interactions in which it plays as a requester. The advantage of this strategy over the eviction is twofold: first, it allows nodes with zero reputation to redeem themselves by cooperating again (a win-win situation from both the node and the system point of view); second, it alleviates the impact of false positive audits, which in *RACOON* could lead to the irreversible eviction of a node.

6.3.3 *Selfishness injection*

In the selfishness injection step of the framework, *RACOON++* automatically generates selfish deviations from the functional specification of the system as well as the *CEM*. This is implemented by the *Selfish Deviation Injection* (*SDI*) algorithm given in Alg. 4. The algorithm takes as input a Protocol Automaton PA and the Selfishness Model SM. Then, it extends the PA with new elements (states, transitions, roles, etc.) representing the deviations specified in the SM.

We now describe the pseudo-code of the *SDI* algorithm in more detail. For rapid identification, the parts of the pseudo-code in Alg. 4 that are specific to the same type of deviation are highlighted in the same colour. Furthermore, for brevity, we use the same `get(elementId)` notation presented in the previous chapter to refer to the element of PA to which the *elementId* identifier is associated.

A *deviation point* is a transition of the PA in which a deviation can take place. To determine if a transition $t \in T$ is a deviation point, the *SDI* algorithm first checks if the selfishness model contains a selfish deviation d that affects t (line 3). In the affirmative case, Alg. 4 looks for deviation points in lines 4 (timeout), 8 (subset), and 11 (multicast).

Timeout Deviations. For each deviation point $t \in T$, the algorithm generates a timeout deviation by calling the procedure *InjectTimeoutDev* (line 5 in Alg. 4). This procedure creates a new final state s' and a new abstract transition connecting the source state of t with s'.

Subset Deviations. For each deviation point $t \in T$ triggered by a communication method, *SDI* checks if the message content c comprises more than a single data unit (line 8). If so, line 9 calls the procedure *InjectSubsetDev*, which creates new elements to represent the deviation. In

particular, the procedure creates a new content c' (line 18) that shares the same data type as c, but has a shorter length, calculated by taking *d.degree* into consideration (line 17).[3]

Multicast Deviations. For each deviation point $t \in T$ triggered by a communication method, the algorithm checks if the receiver of the message sent during t has a cardinality greater than 1 (line 11 of Alg. 4). If so, line 12 calls the procedure *InjectMulticastDev* to create the role r' (line 31) with a smaller cardinality than the correct one (calculated in line 30 using the deviation degree $d.degree$).

Figure 32 shows the result of executing the *SDI* algorithm on the Protocol Automaton of Figure 30. Consider for example state s_2. In the correct execution of the PA, the role R_1 sends a response message (g_3) to r_0. However, if R_1 is selfish, it may also timeout the protocol, or send a message with a smaller payload (g_3').

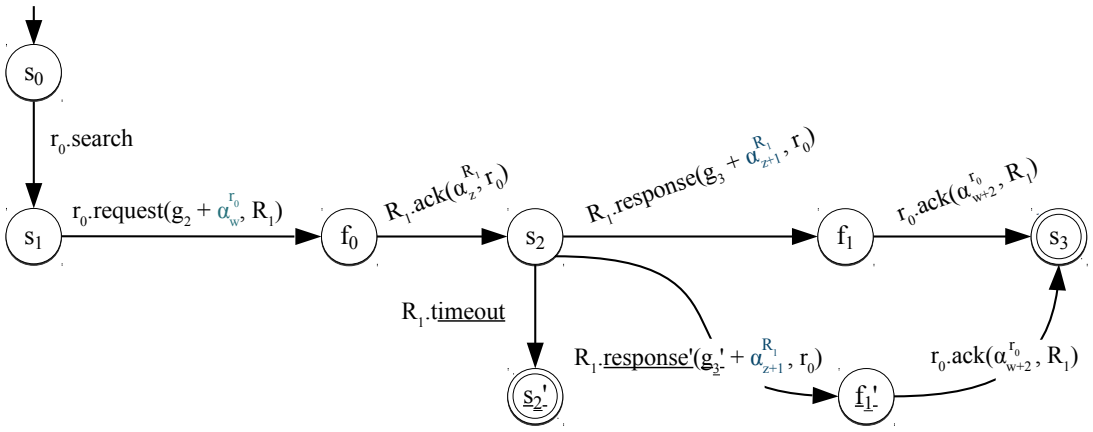


Figure 32: The Protocol Automaton of the *S-R-R* protocol, extended with selfish deviations.

In order to assess the complexity of the *SDI* algorithm, we observe that each iteration of the main loop (over the transitions of the PA) involves the following activities:

- Verification of deviation points, i.e., finding a deviation in the set $D \in SM$ that satisfies certain properties. The time complexity of such verification is linear to the size of D.
- The possible execution of the *InjectTimeoutDev*, *InjectSubsetDev*, and *InjectMulticastDev* methods to generate selfish deviations. These methods have the same complexity of their counterparts in the *CSI* algorithm used in *RACOON*. In fact, their implementations differ only for constant-time performance operations.[4]

Based on the above observations, we can conclude that the overall complexity of the *SDI* algorithm is $O(|T| \cdot |D| \cdot \max\{|K|, |S|\})$, where $|T|$ is the number of transitions in the PA, $|D|$ is the number of selfish deviations defined in the SM, $|K|$ the number of constraints in the PA, and $|S|$ the number of states.

3 We refer to Section 5.3.3 of the previous chapter for a more detailed discussion of the *UpdateConstraints* procedure call in line 19 of Alg. 4, as well as for the instructions in lines 27-29.
4 We refer to Section 5.3.3 for a discussion on the complexity analysis of these methods.

Alg. 4: The *Selfish Deviation Injection* (SDI) algorithm.

Input: A *Protocol Automaton PA* := ⟨R, S, T, M, G, C, K⟩, a *Selfishness Model* SM := ⟨V, D⟩.
Output: *PA*.

Algorithm `SDI(PA,SM)`

```
1  origT := T  // transitions originally included in PA
2  foreach t ∈ origT do
3      if ∃ d ∈ D | d.transitionId = {t.tId, "*"} then
4          if d.dType = "timeout" then
5              InjectTimeoutDev(t)
6          if get(t.methodId).messageId ≠ null then
7              c := get(t.methodId.messageId.contentId)  // sent content
8              if d.dType = "subset" and c.cLength > 1 then
9                  InjectSubsetDev(t, c, d)
10             r := get(t.state2Id.roleId)  // recipient role
11             if d.dType = "multicast" and r.cardinality > 1 then
12                 InjectMulticastDev(t, r, d)

   Procedure InjectTimeoutDev(t)
13     s' := ⟨new_sId, null, final⟩
14     sourceState := get(t.state1Id)
15     t' := ⟨new_tId, sourceState.sId, s'.sId, null⟩  // defined as abstract transition
16     add s' and t' to PA

   Procedure InjectSubsetDev(t, c, d)
17     length' := ⌊c.cLength (1 - d.degree)⌋
18     c' := ⟨new_cId, c.cType, length'⟩
19     UpdateConstraints(c'.cId)
20     message := get(t.methodId.messageId)
21     g' := ⟨new_gId, message.senderId, message.receiverId, c'.cId⟩
22     m' := ⟨new_mId, g'.gId⟩
23     targetState := get(t.state2Id)
24     s' := ⟨new_sId, targetState.roleId, targetState.sType⟩
25     t' := ⟨new_tId, t.state1Id, s'.sId, m'.mId⟩
26     add c', g', m', s', and t' to PA
27     foreach ot ∈ T | ot.state1Id = targetState.sId do
28         ot' := ⟨new_otId , s' , ot.state2Id , ot.methodId⟩
29         add ot' to PA

   Procedure InjectMulticastDev(t, r, d)
30     cardinality' := ⌊r.cardinality (1 - d.degree)⌋
31     r' := ⟨new_rId, cardinality'⟩
32     s' := ⟨new_sId, r'.rId, s.sType⟩
33     message := get(t.methodId.messageId)
34     g' := ⟨new_gId, message.contentId⟩
35     m' := ⟨new_mId, g'.gId⟩
36     t' := ⟨new_tId, t.state1Id, s'.sId, m'.mId⟩
37     add r', s', g', m', and t' to PA
38     add out-transitions of s'      ▷ as in lines 27-29
```

6.3.4 *Rationality injection*

In the *RACOON++* framework, the rationality of a selfish node is described by a Behavioural Model, which determines its decision-making in choosing and carrying out the behaviour that it expects to be the most profitable. From our extensive review of the related work on selfishness, game theory has appeared as the most suitable candidate to formalise the rationality of selfish individuals (see Chapter 3). Differently than in the previous version of the framework, in which we used theoretical tools from classical Game Theory [130], to develop the Behavioural Model in *RACOON++* we rely on Evolutionary Game Theory (EGT) [172].

The first advantage of using EGT is that we can relax the strict assumption of perfect rationality of the nodes, which was imposed by classical game theory. Perfect rationality brings, in fact, a great burden to nodes' computational and analytical abilities, because it requires them to solve a complex combinatorial problem [99] to assess the best action to take at each protocol step. On the contrary, EGT allows modelling nodes as active learners with bounded rationality [172], who can adjust their strategy over time in response to repeated observations of their own and others' utilities. Such learning and adaptation processes assume much lighter capabilities on the part of the nodes, thereby providing a more feasible behavioural model for real nodes [120, 183]. Furthermore, as argued by several authors (e.g., Palomar et al. [142], Wang et al. [171]), the second advantage of adopting an evolutionary approach is that it appears more appropriate for modelling and studying the dynamic behaviour of cooperative systems.

In the remainder of this section, we present the evolutionary game used in the framework for modelling how selfish nodes evolve their behaviour in the cooperative system under design. The components of an evolutionary game are: (i) a static representation of the system interactions, in some cases called Stage Game; (ii) one or more populations of players; (iii) a function to calculate the utility of a given behaviour; and (iv) the dynamics of the learning and imitation processes. We describe each component separately below.

STAGE GAME. Evolutionary games involve the repetition of strategic interaction between self-interested individuals. We model this interaction as a sequential game called the Stage Game (SG), which we represent as in *RACOON* using the *extensive form* (or *game tree*) [130]. Figure 33 shows the game tree of the stage game derived from the *S-R-R* protocol illustrated in Figure 32. To generate the SG, *RACOON++* relies on the *PAtoPG* algorithm developed for *RACOON* (pseudo-code in Alg. 2), which uses the information contained in the Extended Specification resulting from the selfishness injection step. Specifically, the tool translates the Protocol Automaton included in the Extended Specification into elements of a stage game, as informally described below.[5]

Players. A *player* $p \in \mathcal{P}$ corresponds to a role in the PA. For example, players p_0 and p_1 in Figure 33 map to roles r_0 and R_1 of the *S-R-R* protocol. For ease of notation, let $p_k.type$ refer to the type of the role mapped by player p_k.

5 A formal definition of each element and of the mapping rule is provided in Section 5.3.4 of Chapter 5.

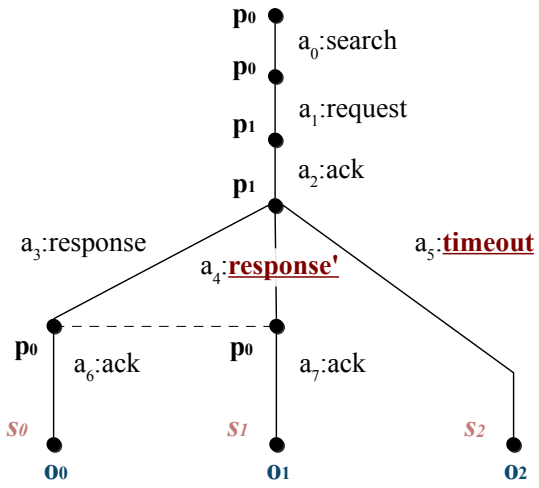


Figure 33: The SG derived from the *S-R-R* protocol in Figure 31.

Node. A *node* $n \in \mathcal{N}$ of the stage game is derived from a state in the PA, and is labelled with the player who has to take action. A leaf node of the SG corresponds to a final state of the PA, and represents a possible *outcome* of the stage game. In Figure 33, each leaf is labelled with the corresponding outcome o_k.

Actions. An *action* is a move of the player in the Stage Game, and is derived from a method in the PA. Note that an edge of the game tree in Figure 33 corresponds to a transition in the Protocol Automaton.

Strategies. A *play* is a path through the game tree from the root to a leaf. It describes a particular interaction between two (or more) players. The ordered sequence of actions that a player takes in a certain play constitutes her *strategy*. Consider for instance the left-most play in Figure 33, which represents the correct execution of the *S-R-R* protocol: Table 25 reports the strategies of players p_0 and p_1 to implement it.

POPULATION OF PLAYERS. A *population* is a group of individuals with common economic and behavioural characteristics. Because of the symmetric nature of cooperative systems, in *RACOON++* we consider a single population of nodes, who can play the strategies in the strategy space defined by the stage game. In conformity with the previous works [142] and [171], we divide the strategy space into non-overlapping subsets, each representing a distinct combination of behaviours for the nodes (i.e., cooperative, selfishness of a certain type). We call these subsets *strategy profiles* $s \in \mathcal{S}$. *RACOON++* creates a strategy profile s_k for each play k of the SG, such that s_k includes the strategies carried out by all players participating in that play. Thus, for example, and with reference to Figure 33, the strategy profile s_0 represents the behaviour of cooperative nodes and includes the strategies presented in Table 25.

We partition the overall population into *sub-populations*, so as to establish a one-to-one mapping with the strategy profiles. A sub-population ω_k represents the group of nodes that adopt the behaviour defined by s_k. In accordance with the *EGT* model, a member of ω_k participates

Player	Strategy
p_0	{a_0:*search*, a_1:*request*, a_6:*ack*}
p_1	{a_2:*ack*, a_3:*response*}

Table 25: The strategies comprising the strategy profile s_0, implementing the correct execution of the stage game in Figure 33.

in the system by repeatedly playing what is specified by her strategy profile, regardless of the outcome of the play. However, a member of ω_k can join another sub-population ω_j if she expects to increase her utility by playing s_j. Thus, the size of a sub-population reflects the success of the associated strategy profile. As the system evolves, the distribution of members of the sub-populations can vary. We call this information the *population state* of the system.

UTILITY FUNCTION. The utility function of a player assigns a value (i.e., the utility) to each outcome of a game. An outcome o of the SG depends on the sub-populations of the interacting players, whose strategies determine the particular play that leads to o. For example, consider the stage game in Figure 33, and let players p_0 and p_1 be members of sub-population ω_1 and ω_3, respectively. Table 26 lists the planned sequence of actions of the two players. The interaction starts with player p_0 executing the *search* transition and then sending a request message to the other player. Player p_1 will first acknowledge the reception of the message, and then she will terminate the protocol. The interaction described above corresponds to the play {a_0:*search*, a_1:*request*, a_2:*ack*, a_5:*timeout*} in Figure 33, which leads to the outcome o_2 induced by the strategy profile s_2. The outcomes of a stage game describe the results of the interaction between every possible combination of players from different sub-populations.

Player	Strategy profile	Strategy
p_0	s_1	{a_0:*search*, a_1:*request*, a_6:*ack*}
p_1	s_3	{a_2:*ack*, a_5:*timeout*}

Table 26: The strategies implemented in the SG of Figure 33 when players p_0 and p_1 are from sub-populations ω_1 and ω_3.

In *RACOON++*, the utility received from playing a stage game has two terms: the *protocol payoff*, and the *incentives* introduced by the *CEM*. The protocol payoff γ_j evaluates the costs and benefits of a player when the outcome of SG is o_j. To calculate this value, *RACOON++* evaluates the valuation elements defined in the selfishness model by the Designer (see Section 6.3.1.2). Let us illustrate the procedure to evaluate the protocol payoff γ_0 in the stage game of Figure 33, in the case of interaction between members of the cooperative sub-population ω_0. Consider the following valuations associated to role r_0 and, thus, to player p_0:

- $v_0 = \langle v_0, \text{“}\texttt{transition}\text{”}, \texttt{search}, r_0, 10, 1\rangle$, which specifies benefits and costs associated to role r_0 when performing the abstract transition `search` (a_0:*search* action in Figure 33).

- $v_1 = \langle v_1, \text{“message”}, g_3, r_0, 3, 1 \rangle$, which specifies benefits and costs associated to role r_0 when receiving the message g_3 from role R_1 (*a_3:response* action in the figure).

Let the content $c \in C$ transmitted by the message g_3 comprise a list of 10 data units. Then, the protocol payoff of player p_0 is:

$$\gamma_0(p_0) = v(v_0) + v(v_1) = 9 + 2 \cdot r_0.\text{cardinality} \cdot c.\text{cLength} = 9 + 2 \cdot 1 \cdot 10 = 29.$$

The protocol payoff is the expected utility that would be received if no incentives for cooperation were attached to the system. However, the *CEM* used in *RACOON++* establishes that the ability of a player to obtain a requested service is proportional to her reputation value (see Section 6.3.2). Thus, the utility $u_j \in \mathbb{R}$ obtained by a player p_i depends on whether she plays as a service requester in the stage game. Formally:

$$u_j(p_i) = \begin{cases} \gamma_j(p_i) \cdot \rho(p_i) & \text{if } p_i.\text{type} = \text{“requester"} \\ \gamma_j(p_i) & \text{otherwise} \end{cases}$$

, where the function $\rho : \mathcal{P} \to [0, 1]$ determines the probability that player $p_i \in \mathcal{P}$ will receive the protocol payoff, calculated as the reputation of p_i divided by the upper bound ρ_{max} of reputation values.

Following on the previous example, let the reputation mechanism allow values between 0 and 10, and let the requester player p_0 have reputation 6. Then, her utility can be calculated as:

$$u_0(p_0) = \gamma_0(p_0) \cdot \rho(p_0) = 29 \cdot 0.6 \simeq 17.4 \ .$$

EVOLUTIONARY DYNAMICS A common assumption in traditional game theory is that players have the information and skills to assess and choose the best strategy to play in the current system's state [130]. However, as other works have highlighted [142, 171, 183], this assumption places a heavy burden on nodes' computational and communication capabilities, which is infeasible in most cooperative systems. On the contrary, EGT assumes that individuals are not fully rational, but tend to implement the most remunerative strategies through learning and imitation [172].

In *RACOON++*, each node monitors the utility it has obtained for playing the strategy profile of its sub-population. If the utility decreases for more than a given number of consecutive observations, or if a specified time has elapsed, then the node will look for a fitter sub-population to join. The accountability audits of *R-acc++* provide the means to learn what are the fittest sub-populations in the system. More precisely, we assume that a witness can infer the sub-population and the utility of a node by auditing its logs, as the recorded actions can be traced back to a particular strategy profile (the space of strategy profiles, as well as the costs and benefits of each action, are common knowledge to all nodes, because we assume a single population). After an audit, the witness compares its own utility against that of the monitored node. If the witness has a lower utility, it will join the sub-population of the monitored node with a given

probability [171, 183]. Note that this probability determines the evolution rate: the smaller its value, the slower the fittest sub-population in the system increases.

6.4 RACOON++ TUNING PHASE

The tuning phase of *RACOON++* aims at configuring the accountability and reputation mechanisms according to a list of design objectives input by the Designer. Tuning involves an iterative two-step refinement process, which alternates evaluation with the tuning of the configuration parameters. The evaluation involves Evolutionary Game Theory (*EGT*) analysis and simulations to study the system dynamics in a given configuration setting. This task is performed by the *R-sim* simulator integrated into the framework. Then, an exploration algorithm uses the evaluation results to optimise the parameters of the *CEM*. The tuning process ends after a set number of iterations, or when a configuration that satisfies the Designer's objectives is found.

6.4.1 *Input of the Tuning phase*

RACOON++ offers a predefined set of selfish-resilience and performance objectives for the cooperative systems designed within its framework. As in *RACOON*, each of these design objectives defines a predicate over a system metric (see Definition 5.22), which can be evaluated by the *RACOON++* evaluation tool, namely, the *R-sim* simulator. Examples of predicates are *at most* and *at least*. Hereafter, we present the application-independent objectives natively supported by *RACOON++*.

- *Cooperation level*: the fraction of cooperative nodes in the system;
- *Cooperation persistence*: the probability that a cooperative node stays cooperative;
- *Cooperation attractiveness*: the probability that a selfish node becomes cooperative;
- *Audit precision*: the number of correct positive audits divided by the total number of positive audits;
- *Audit recall*: the number of correct positive audits divided by the number of audits that should have been positive;[6]
- *CEM bandwidth overhead*: the costs of the accountability and reputation mechanisms in terms of additional bandwidth;
- *CEM message overhead*: the costs of the accountability and reputation mechanisms in terms of extra messages.

Examples of design objectives are "cooperation level *at least* 0.8" and "CEM message overhead *at most* 0.6". *RACOON++* allows specifying further objectives on application-specific metrics (e.g., throughput, jitter, anonymity level). For each custom objective, the Designer needs to implement the methods to collect and evaluate the related metrics in the evaluation tool.

6 We recall that a selfish witness of the accountability system may decide not to perform an audit in order to save local resources and return a false negative audit result instead. More details are provided in Section 4.2.2.

The second input of the tuning phase is an implementation of the functional specification of the designed system in the *R-sim* simulator, which we discuss in more detail in the next section.

6.4.2 *Configuration evaluation*

To evaluate a configuration setting for the *CEM*, *RACOON++* simulates the system behaviour using the simulation framework *R-sim*, which is based on the PeerSim simulator [128]. The simulation results indicate whether the evaluated *CEM* configuration has satisfied the list of design objectives set by the Designer or not. In the following, after a brief overview of the main components of PeerSim, we present *R-sim* in detail.

6.4.2.1 *PeerSim overview*

PeerSim is a scalable simulation environment written in Java and released under the GPL open source licence. Although originally designed for P2P systems, it can be easily adapted to simulate cooperative systems in general, as we will show in Section 6.5.

A simulation in PeerSim consists of different components (objects), where every component is easily replaceable by another one implementing the same functionality (interface). Among the main interfaces are `Node`, `Protocol`, `Linkable`, and `Control`. Concretely, in PeerSim, the network in composed of *nodes*, which are containers of *protocols*. A *linkable* is a protocol that enables communication between neighbour nodes, thereby defining an overlay network. Finally, a *control* can monitor or modify every other component,[7] and can be executed before or during the simulation. The controls executed before the simulation are also called *initializers*.

The simulation engine of PeerSim offers cycle-based and event-based simulations. The cycle-based model achieves extreme scalability and performance, at the cost of some loss of realism: nodes communicate with each other directly (no message passing), and each node protocol is executed in turn at every cycle. In contrast, the event-based model enables more realistic simulations of the underlying communication network but is less scalable.[8]

An experiment in PeerSim is fully specified by a plain text configuration based on Java property files, i.e., collections of pairs associating a property name to a property value. Properties specify the simulation engine to use, the implementation (Java classes) of components to load at runtime, and the numeric or string parameters for these components. Each component must implement the appropriate interface of the PeerSim Java API.

The example configuration in Figure 34(a) defines a network composed of one million nodes. Figure 34(b) illustrates the resulting PeerSim components. The simulation is run using the cycle-based engine for 600 cycles. At each cycle, each node runs (i) an overlay protocol labelled `overlay` that is implemented by the class `Newscast` [91],[9] with parameter `cache` set to 20, and (ii) an implementation of the *S-R-R* protocol (class `SRR`), labelled as `srr` in the configuration file.

7 For example, a control can add new nodes to the network or remove existing ones; or it can interact with protocols providing them with external inputs of modifying their parameters.

8 The PeerSim developers tested up to 10^5 nodes in event-based simulations and 10^7 nodes in the cycle-based ones [128].

9 The Newscast overlay protocol maintains a robust random topology, based on the continuous exchange of up-to-date information about neighbours (node addresses and timestamps). The amount of information exchanged depends on the fixed-sized cache at the nodes, which is a protocol parameter.

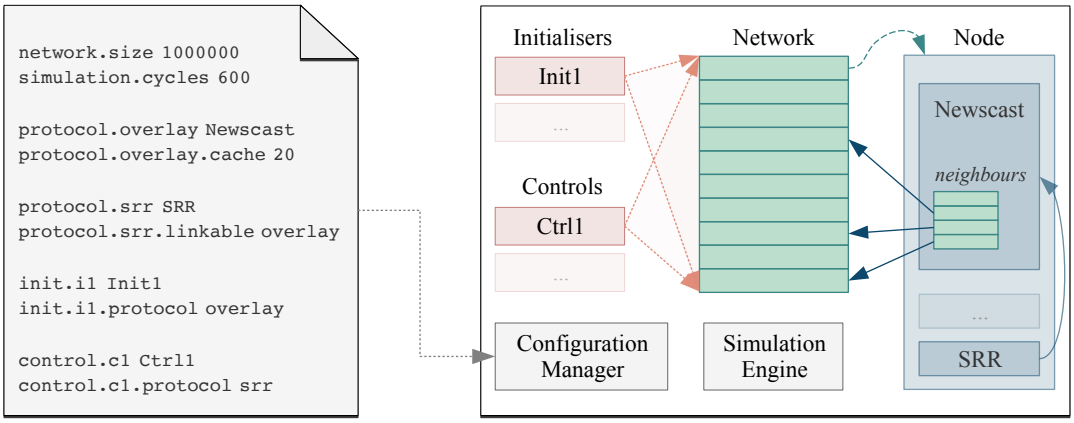


Figure 34: A simple PeerSim configuration file (a) and the corresponding PeerSim components (b). Depicted in light colour in (b), are the additional components that can easily be added.

The property "`protocol.srr.linkable`" assigns the Newscast overlay to *SRR*. The configuration in Figure 34(a) also defines an initializer `i1` and a control `c1` for the protocols `overlay` and `srr`, respectively.

As said above, the class `SRR` provides an implementation of the *S-R-R* protocol, and, specifically, of its Protocol Automaton (Figure 30). Listing 5 presents the basic element of this class. In particular, `SRR` needs to implement the `CDProtocol` interface[10] and to provide the method `nextCycle` that initiates the PA execution. Note that the `SRR` object calling this method acts as the role r_0 in the PA and interacts with each node that plays as R_1 (lines 6-7 in Listing 5). More precisely, r_0 interacts with the `SRR` instance associated with a node in R_1, which is maintained in the list `roleR1` declared in line 2. In Listing 5, the implementation of PA transitions is provided in separate methods. Although this is is not a necessary restriction, it facilitates the conversion of the PA into source code. The arguments of the `search`, `request`, and `response` methods are the `SRR` instance of the interacting node and the content of the possible message exchange (e.g., the list `messageG2` of resources requested by r_0 in the message $g_2 \in PA.G$).

```
1  public class SRR implements CDProtocol {
2    List<SRR> roleR1; // the role 'R1' in the PA
3    // ...
4    public void nextCycle( Node node, int protocolID ) {
5         // ...
6         for(SRR r1 : roleR1)
7           search( r1 );
8    }
9    // impl. of transition 'search' in the PA
10   public void search( SRR r1 ) { /* .... */ }
11   // impl. of transition 't1' (method 'request') in the PA
12   public void request( SRR r1, List messageG2 ) { /* ... */ }
13   // impl. of transition 't2' (method 'response') in the PA
14   public void response( SRR r0, List messageG3 ) { /* ... */ }
15 }
```

Listing 5: Source code of the main elements of the `SRR` class imlpemented in PeerSim.

10 The cycle-based extension of the `Protocol` interface.

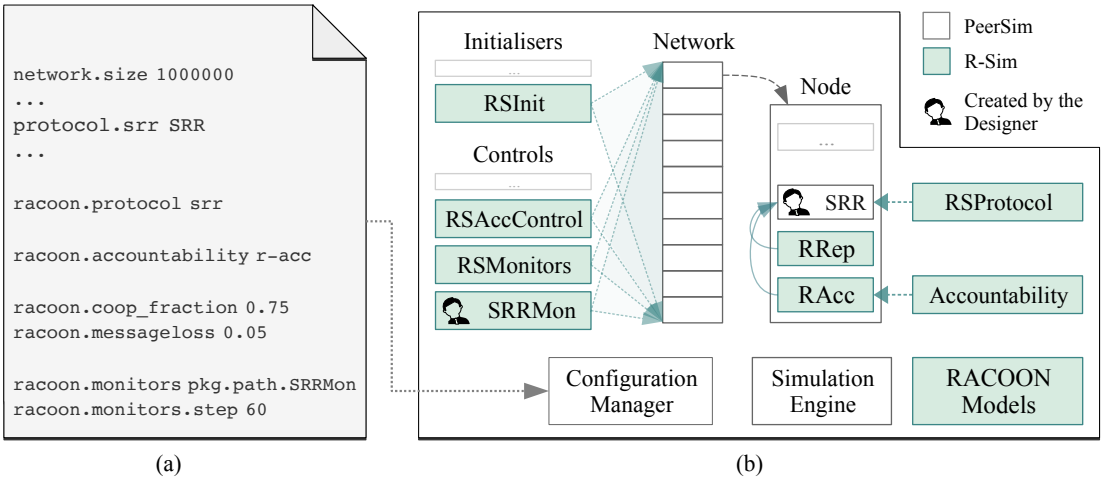


Figure 35: Integration between the *R-sim* and PeerSim configuration properties (a) and components (b).

6.4.2.2 *R-sim simulator*

The *RACOON++* simulation framework *R-sim* uses the evolutionary game model of the cooperative system to simulate the system dynamics in the candidate configuration setting. *R-sim* supports a cycle-based model, in which time is structured into rounds. At each round, each node plays a certain strategy of the SG, according to the evolutionary dynamics described in Section 6.3.4. During the simulation, *R-sim* collects statistics about such dynamics, to evaluate the design objectives. To the best of our knowledge, *R-sim* is the only available software tool for the dynamic simulation of selfish and strategic behaviours in distributed systems.

In contrast with *RACOON*, which includes a custom-built simulator for cooperative systems, *RACOON++* relies on the state-of-the-art PeerSim simulator, thereby improving the usability, accuracy and performance of the framework. We have chosen PeerSim among other simulation tools (see [24] for a comprehensive review) for the following reasons: (i) it meets the requirements of scalability and dynamicity imposed by the evolutionary model; (ii) it supports the integration with *RACOON++* thanks to its modular architecture; (iii) it is an active project, with a good developer community and support.

As shown in Figure 35, *R-sim* exploits the modular architecture of PeerSim extending it with new components to develop, simulate and evaluate the cooperative system resulting from the design phase of the *RACOON++* framework. Hereafter, we present the main components.

CONFIGURATION PARAMETERS. *R-sim* introduces new configuration parameters to set up its additional components. These parameters can be specified in the PeerSim configuration file by using the prefix "`racoon.`". The portion of configuration file depicted in Figure 35(a), for example, specifies six *R-sim* parameters, which we discuss in the remainder of this section.

RACOON MODELS. The set of Java classes collectively denoted as `RACOON Models` in Figure 35(b) allow representing in PeerSim the specification artefacts used in *RACOON++*, i.e., the Protocol Automaton PA, Selfishness Model SM, design objectives, and Evolutionary Game EG. The instances of these classes contain the information to simulate the behaviour of each sub-

population of nodes defined in EG. Also, `RACOON Models` includes the `CEMConfigCandidate` class to model the *CEM* configuration to evaluate.

CEM CLASSES. The `RAcc` and `RRep` protocol components implement the accountability and reputation mechanisms used in *RACOON++*. In particular, `RAcc` extends the abstract class `Accountability`, which maintains a `SecureLog` and provides the methods to invoke the accountability protocols described in Section 6.3.2 (e.g., **`commit`** the reception of a message, **`audit`** the log of a node, create challenges with **`createAuditChallenge`** and **`createSendChallenge`**). *R-sim* also includes an implementation of the `FullReview` accountability mechanism.[11] The configuration parameter "`racoon.accountability`" indicates the accountability mechanism to enforce between *R-acc++* ("`r-acc`" or "`racc`") and FullReview ("`fullreview`"). Note that the properties values are case insensitive and "`r-acc`" is the default.

R-SIM PROTOCOLS. The `RSProtocol` class extends a standard PeerSim protocol to enable its interaction with the *RACOON++* framework. In the following, we discuss the additional functionalities provided by `RSProtocol`, with reference to the re-implementation of the `SRR` class shown in Listing 6. For ease of presentation, we denote the node that runs the extended protocol as its *actor*.

```
1  public class SRR extends RSProtocol implements CDProtocol {
2    List<SRR> roleR1; // the role 'R1' in the PA
3    // ...
4    public void nextCycle( Node node, int protocolID ) {
5      // ...
6      for(SRR r1 : roleR1) {
7        if( r1.acceptInteraction( this ))
8          search( r1 );
9      }
10   }
11   // impl. of transition 'search' in the PA
12   public void search( SRR r1 ) {
13     processRTransition( "Search", this, "r0" );
14     // ...
15   }
16   // impl. of transition 't1' (method 'request') in the PA
17   public void request( SRR r1, List messageG2 ) {
18     processRMessage( "g2", messageG2.size(), this, "r0", r1, "r1" );
19     // ...
20   }
21   // impl. of transition 't2' (method 'response') in the PA
22   public void response( SRR r0, List messageG3 ) {
23     processRMessage( "g3", messageG3.size(), this, "r1", r0, "r0" );
24     switch( getDeviationType( "t2" ) ){
25       case TIMEOUT:
26         break;
27       case SUBSET:
28         double deviationDegree = getDeviationDegree( "t2" );
29         /* implementation of the subset deviation */
30         break;
31       default:
32         /* implementation of the correct behaviour */
33     }
34   }
35 }
```

Listing 6: Source code of the main elements of the `SRR` class implemented in *R-sim*.

11 To enable a comparative evaluation of the *R-acc++* and FullReview performance, presented in Section 6.5.5.

`RSProtocol` provides attributes and methods to maintain the current sub-population of an actor as well as the utility obtained since its last accountability audit. If on the one hand, the sub-populations management is automatically carried out by *R-sim* at run-time, on the other hand, the Designer needs to put information to calculate the utility directly into the source code. `RSProtocol` simplifies this task by exposing two methods to retrieve such information from the valuations defined in the SM (see Section 6.3.1.2):

- **`processRTransition`** `( String tId, RSProtocol rsp, String rId )`. It retrieves the valuations $v \in SM.V$ that satisfies the following: $v.scope =$ "transition", $v.scopeId = tId$, and $v.roleId = rId$. Then, it updates the utility of the protocol instance rsp based on the costs and benefits specified in each valuation v retrieved. E.g., line 13 in Listing 6.
- **`processRMessage`** `( String gId, int cLength, RSProtocol sender, String senderId, RSProtocol receiver, String receiverId )`. It retrieves the valuations $v \in SM.V$ such that $v.scope =$ "message", $v.scopeId = gId$, and $v.roleId = senderId \cup receiverId$. Then, it updates the utility of the protocol instances $sender$ and $receiver$ based on the costs and benefits specified in the respective valuations retrieved, also taking into account the length of the content conveyed by the message (see Definition 6.8). E.g., lines 18 and 23.

To facilitate the implementation of the selfish behaviours defined in the SM, the `RSProtocol` class exposes the following methods:

- `DeviationType` **`getDeviationType`** `( String tId )`. It returns the type[12] of the deviation that an actor may perform from the transition $\{t \in PA.T \mid t.tId = tId\}$. The result depends on the strategy implemented by the current sub-population of the actor. The method can be used as the argument of a switch-case statement, where each case-block implements the execution of the transition according to the deviation type (lines 24-33 in Listing 6).
- `double` **`getDeviationDegree`** `( String tId )`. If the deviation type of $t \in PA.T$ with $t.tId = tId$ is *NONE*, then the method returns 0; otherwise, it returns the degree of the deviation performed by the actor in t, which depends on the strategy implemented by its current sub-population. For example, line 28 aims to get the degree of possible deviations from the *t2* transition of the *S-R-R* protocol (i.e., the execution of the *response* method).

Finally, the `RSProtocol` class enables the interaction between the extended protocol and the `RRep` and `RAcc` instances ran by the same actor. In particular, `RSProtocol` exposes the **`acceptInteraction`** method that returns *true* if the actor of the extended protocol can start an interaction with the actor of the protocol passed as an argument;[13] otherwise, the method returns *false*. For example, in Listing 6, the actor playing as R_1 in line 7 decides whether to accept or not to interact with the actor playing the role r_0.

The binding of the extended protocol to the other components of the *R-sim* simulator occurs at run-time based on the value of the configuration parameter "`racoon.protocol`". For example, the configuration file in Figure 35(a) specifies that the protocol to extend is the one labelled as `srr` and implemented by the `SRR` class.

12 `DeviationType` is an enum type whose values are *TIMEOUT*, *SUBSET*, *MULTICAST*, and *NONE*.

13 We recall that a node i may accept to interact with another node j based on (i) the current reputation value of j, and (ii) whether j has been suspected by the *R-acc++* mechanism (see Section 6.3.2).

R-SIM CONTROLLERS. The `RSInit` control component initializes every `RSProtocol` instance involved in the simulation. The initialization consists of three operations. First, it assigns a sub-population in EM to each node in the network, in such a way that the selfish sub-populations are uniformly represented. The size of the cooperative sub-population, instead, depends on the configuration parameter "`racoon.coop_fraction`", which indicates the proportion of cooperative nodes in the system. For example, the configuration file in Figure 35(a) specifies that only one-fourth of the network consists of selfish nodes. Second, `RSInit` initializes the accountability protocol (*R-acc++* or FullReview) and the initial reputation value of each node. Finally, the initializer sets the message loss rate to simulate in *R-sim*, which allows evaluating the performance of systems deployed over unreliable networks. The rate of the message loss is specified by the "`racoon.messageloss`" configuration parameter (default value: 0).

The `RSAccControl` component controls the operation of the `Accountability` instances in *R-sim*, by checking at every simulation cycle what are the accountability protocols to execute.

R-SIM MONITORS. *R-sim* includes three control components for monitoring a set of simulation metrics and performance. Specifically, it includes monitors for the design objectives natively supported by *RACOON++* (class `NativeDOMonitor`), the performance of the accountability mechanisms (class `AccountabilityMonitor`), and the performance of the sub-populations (class `SubPopulationMonitor`) such as their size, average utility, and average reputation of their members. Each of these monitors implements the PeerSim `Control` interface and extends the *R-sim* abstract class `DOMonitor`, which defines what are the basic functions to implement for measuring and evaluating a system metric. The Designer can implement her custom monitors — e.g., the `SRRMon` class in Figure 35(b) — by extending the `DOMonitor` class and implementing its abstract methods.

The `RSMonitors` controller is responsible for the periodic execution of the `DOMonitor` instances, namely, the three monitors included in *R-sim* and the set of custom monitors indicated in the configuration parameter "`racoon.monitors`". The value of this parameter is the list of full class names of each custom monitor to enable, each separated by a space. The number of cycles between two monitoring executions is determined by the configuration parameter "`racoon.monitors.step`" (default value: 1, i.e., every cycle). Finally, note that the `RSMonitors` controller relies on the service class `OutputWriter` to write the simulation results as comma-separated values (CSV) files.

6.4.3 *Configuration Exploration*

The output of the *RACOON++* framework is the design and configuration of a cooperative system that achieves the objectives set by the Designer. Thus far, we have described how *RACOON++* fosters cooperation using accountability and reputation mechanisms (Section 6.3.2), and how it evaluates the system performance using *EGT* and simulation (Section 6.4.2). The last step of the framework relies on the evaluation results to tune the configuration parameters of its cooperation enforcement mechanisms, aiming to achieve the desired design objectives.

THE TUNING ALGORITHM. *RACOON++* uses an iterated local search algorithm to support the automatic exploration of the *CEM* configuration space, moving from one configuration candidate to a neighbouring one until all design objectives are met [107]. A configuration candidate $\chi \in X$ is a vector containing the *CEM* parameters, namely, *witness set size*, *audit period*, *audit probability*, *degree of punishment*, and *degree of reward* (defined in Table 20 in the previous chapter). Two configuration candidates are considered neighbours if they differ in exactly one parameter value. In local search, neighbours are selected based on the performance achieved by the last evaluated solution, i.e., based on the number and type of design objectives met by the previous configuration candidate.

Alg. 5 presents the `Tuning` algorithm used in *RACOON++*, which is an instantiation of the local search process described above. The algorithm takes as input an initial configuration $conf \in X$, and iteratively decides whether to explore or not neighbour configurations (lines 7-14) depending on the performance computed by the `Evaluate` and `Score` functions (lines 5-6). Both functions, along with the `Neighbour` function (line 14) for exploring the configuration neighbourhood, are discussed next.

If no feasible solution is found after a number of iterations $tMax$,[14] then the `Tuning` algorithm stops the search, asking the Designer to improve the system design manually or to relax the objectives. Nevertheless, the algorithm returns the best configuration found during the exploration process, i.e., the configuration that has achieved the highest performance score (lines 11-13 in Alg. 5).

Alg. 5: The `Tuning` algorithm to explore the configuration space of the *CEM* in *RACOON++*.

Data: A list of design objectives O.
Input: The maximum number of iterations $tMax \in \mathbb{N}$, and the initial configuration $conf \in X$.
Output: A configuration $bestConf \in X$.

Algorithm `Tuning` (*tMax*, *conf*)

```
1  bestConf := conf
2  bestScore := 0
3  t := 0
4  while t < tMax do
5      performance := Evaluate(conf)
6      score := Score(performance)  // return 1.0 if all the objectives were met
7      if score = 1.0 then
8          bestConf := conf
9          break  // stop the search
10     else
11         if score > bestScore then
12             bestConf := conf  // store the best configuration evaluated so far
13             bestScore := score
14         conf := Neighbour(conf, performance)
15     t := t + 1
16 return bestConf
```

14 For example, because the design objectives were contradictory or too demanding.

PERFORMANCE EVALUATION. The `Tuning` algorithm uses the `Evaluate` function (lines 1-5 in Alg. 6) to compute the performance of a configuration on the objectives O set by the Designer. This function invokes the *R-sim* simulator to simulate the behaviour of the cooperative system configured as specified in `conf` (line 1). The simulation results are saved in a vector of real numbers, `results`, where the i-th element stores the metric value associated with the i-th objective in O. Each simulation result is then evaluated against the corresponding design objective by the `EvaluateObjective` function reported in lines 6-12 in Alg. 6. This function returns a real number $\in [0, 1]$, in such a way that the higher the value the best the performance — notably, the function returns 1.0 if the objective was met (lines 8-12). Note that as the evaluation metrics can be defined in very different ranges,[15] they are normalised into $[0, 1]$ before to be used in the `EvaluateObjective` function, in order to give to each metric a similar weight.

Alg. 6: The functions to calculate the performance and the score of a configuration candidate.

Data: A list of design objectives *O*.

Function Input: The configuration *conf* $\in$ X to evaluate.
Function Output: The vector *performance*.

Function `Evaluate` (*conf*)

```
1  results := RSim.simulate(conf)
2  performance := {}
3  foreach o ∈ O do
4  |  performance[o] := EvaluateObjective(o, results[o])
5  return performance
```

Function Input: An objective *o* and the associated value *r* resulting from the simulation.
Function Output: A value $\in [0, 1]$ that indicates the level of achievement of *o*.

Function `EvaluateObjective` (*o*, *r*)

```
6  r' := r/(o.maxValue − o.minValue)  // rescaled result value
7  t' := o.value/(o.maxValue − o.minValue)  // rescaled threshold
8  if o.predicate = at_least then
9  |  eval := r/t'
10 else
11 |  eval := (1.0 − r')/(1.0 − t')
12 return min(eval, 1.0)
```

Function Input: The vector *performance*, i.e., the output of the `EvaluateObjective` function.
Function Output: The value *score* $\in [0, 1]$.

Function `Score` (*performance*)

```
13 score := 1.0
14 foreach p ∈ performance do
15 |  score := score · p
16 return score
```

The execution of the `Evaluate` function returns a vector of real numbers, called the `performance` vector. The `Score` function converts this vector into a single real number in $[0, 1]$, calculated as

15 See the Definition 5.22 of design objectives presented in Chapter 5.

the product of its values (lines 13-16). The score value is used in the `Tuning` algorithm to verify whether the current configuration candidate has met all the design objectives (line 7 in Alg. 5) as well as to keep track of the best configuration evaluated (lines 11-13).

GENERATING CONFIGURATION NEIGHBOURS. If a configuration candidate $conf \in X$ fails to meet all the design objectives set by the Designer, then the `Tuning` algorithm explores the neighbourhood of $conf$ searching for better configurations. This process is carried out by the `Neighbour` function in lines 1-17 in Alg. 7, which takes as input the current configuration $conf$ and its $performance$ vector, and returns a neighbour configuration $conf'$ that has not been tested yet.

Alg. 7: The functions to generate the neighbours of a configuration candidate.

Data: A list of design objectives *O* and the set of configurations already *tested*.

Function Data: The set of configurations *to_test*.
Function Input: A configuration *conf* $\in$ X and the associated vector *performance*.
Function Output: The neighbour configuration *conf'* $\in$ X.

Function `Neighbour` (*conf*, *performance*)

```
1  add conf to tested
2  pScore := ParametersScore(performance)
3  max_p := max absolute value in pScore
   /* Create the new configurations to test */
4  foreach p ∈ pScore such that p = max_p do
5      newConf := conf
6      if p > 0 then
7          newConf[p] := Increment(conf[p])
8      else
9          newConf[p] := Decrement(conf[p])
10     add newConf to to_test;
   /* Return a new configuration to test */
11 conf' := conf
12 while conf' ∈ tested do
13     if to_test ≠ ∅ then
14         conf' := Poll(to_test)
15     else
16         conf' := Random(conf') // random update of a random parameter of conf'
17 return conf'
```

Function Input: The vector *performance*, i.e., the output of the `EvaluateObjective` function.
Function Output: A vector of values *pScore* to guide the update of a configuration candidate.

Function `ParametersScore` (*performance*)

```
18 pScore := [0, 0, 0, 0, 0]
19 foreach o ∈ O do
20     if performance[o] < 1.0 then
21         UpdatePScore(pScore, o)
22 return pScore
```

The neighbour selection is guided by a set of rules derived from an empirical analysis of the *CEM* parameters and their impact on the design objectives natively supported by *RACOON++*. Concretely, we performed a systematic evaluation of 250 configuration candidates in three cooperative systems,[16] for a total of 750 evaluations. For each system and each configuration parameter, we evaluated the performance of the design objectives when increasing that parameter value, while keeping constant the other parameters. The relations emerging from every combination parameter-objective were classified into the five categories listed below:

- *Direct relation*: an increase in the parameter value corresponds to an improvement in the design objective performance greater than 5%.
- *Weak direct relation*: an increase in the parameter value corresponds to a slight improvement in the design objective performance (between 1% and 5%).
- *Unrelated*: there is no significant correlation between the parameter value and the performance of the design objective.
- *Weak indirect relation*: an increase in the parameter value corresponds to a slight degradation in the design objective performance (between 1% and 5%).
- *Indirect relation*: an increase in the parameter value corresponds to a degradation in the design objective performance greater than 5%.

Table 27 summarises the results of the empirical analysis (for ease of presentation, "unrelated" results are omitted). For instance, we observe that the higher the number of witnesses, the higher the *CEM* bandwidth overhead, because each witness increases the amount of log transmissions and checking. As another example, we observe that the shorter the audit period, the higher the cooperation level, because selfish nodes are detected earlier and punished more often. Note that the relations listed in Table 27 have been observed in all the cooperative systems considered for the empirical analysis; any other relation that did not reach uniform consensus in the three systems was discarded.

The integration of these observations into the neighbour generation process is realised by the function `ParametersScore` (lines 18-22 in Alg. 7). The function takes as input the `performance` vector associated with a configuration candidate, and uses information derived from Table 27 to determine whether the *CEM* parameters could be modified (and how) to improve the overall performance. To this end, `ParametersScore` creates a support vector of five elements, named `pScore`, where each element is associated with a *CEM* parameter. For each design objective that was not satisfied (lines 19-20), the function `UpdatePScore` increments or decrements the `pScore` values as specified by the updating rules reported in Table 28. These rules are derived from the relations in Table 27: direct and weak direct relations map to an increment of the `pScore` value ($+1$ and $+0.5$, respectively), indirect and weak indirect relations map to a decrement (-1 and -0.5), and unrelated cases do nothing. For example, let the unsatisfied design objectives be the *cooperation level* and the *CEM bandwidth overhead*. Then, the overall `pScore` vector returned by `ParametersScore` is $\{1-1,-1,1+1,0.5+0.5,0.5+0.5\} = \{0,-1,2,1,1\}$.

16 The use cases selected for the evaluation of the *RACOON++* framework, which we present in Section 6.5.1.

Native Design Objective	**Parameter**	*Relation*
Cooperation Level	Witness set size	Direct
Cooperation Level	Audit period	Inverse
Cooperation Level	Audit probability	Direct
Cooperation Level	Degree of punishment	Direct (weak)
Cooperation Level	Degree of reward	Inverse (weak)
Cooperation Persistence	Degree of punishment	Direct (weak)
Cooperation Persistence	Degree of reward	Inverse (weak)
Cooperation Attractiveness	Witness set size	Direct (weak)
Cooperation Attractiveness	Degree of punishment	Direct
Audit Precision	Witness set size	Direct
Audit Precision	Audit probability	Direct
Audit Recall	Witness set size	Direct
Audit Recall	Audit probability	Direct
CEM Bandwidth Overhead	Witness set size	Inverse
CEM Bandwidth Overhead	Audit probability	Direct
CEM Bandwidth Overhead	Degree of punishment	Direct (weak)
CEM Bandwidth Overhead	Degree of reward	Inverse (weak)
CEM Message Overhead	Witness set size	Inverse
CEM Message Overhead	Audit period	Direct
CEM Message Overhead	Degree of punishment	Direct (weak)
CEM Message Overhead	Degree of reward	Inverse (weak)

Table 27: Observed relations between the design objectives natively supported by *RACOON++* and the *CEM* configuration parameters.

	pScore update variations [a]				
Unsatisfied Design Objective	ws	a_{pe}	a_{pr}	dp	dr
Cooperation Level	+1	-1	+1	+0.5	-0.5
Cooperation Persistence	0	0	0	+0.5	-0.5
Cooperation Attractiveness	+0.5	0	0	+1	0
Audit Precision	+1	0	+1	0	0
Audit Recall	+1	0	+1	0	0
CEM Bandwidth Overhead	-1	0	+1	+0.5	-0.5
CEM Message Overhead	-1	+1	0	+0.5	-0.5

[a] ws = witness set size, a_{pe} = audit period, a_{pr} = audit probability , dp = degree of punishment, dr = degree of reward.

Table 28: Rules to update the pScore vector created by the `ParametersScore` function in Alg. 7.

The `Neighbour` function uses the pScore vector for two purposes: (i) to determine the parameters to modify for generating the neighbour configuration, i.e., the parameters corresponding to the pScore elements with the greatest absolute value (lines 3-4 in Alg. 7),[17] and (ii) to determine whether to increment or decrement the selected parameter, based on the sign of the associated pScore value. For instance, given the pScore vector $\{0, -1, 2, 1, 1\}$, the `Neighbour` function will generate a neighbour configuration $newConf$ characterised by an increment of the audit probability (line 6-7), and will add it to the set of configurations to_test (line 10). As another example, consider the vector $\{-2, 2, 0, 0, 0\}$. In this case, `Neighbour` will generate two neighbour configurations to add to the to_test set: one configuration that decrements the current witness set size, and the other one that increments the audit period. If no rules are available for updating a particular configuration (i.e., all elements in pScore are equal to 0), then no neighbour configuration is generated.

The last part of the `Neighbour` function returns a configuration candidate $conf'$ to the `Tuning` algorithm. Note that $conf'$ is either retrieved from the set of pending configurations to_test (line 14 in Alg. 7) or is randomly generated if the to_test set is empty (line 16). Note that in order to avoid the re-exploration of the regions of the configuration space, the algorithm keeps memory of the previously generated candidates by storing them into the tested set.

6.5 EVALUATION

In this section, we demonstrate the benefits of using *RACOON++* to design selfish-resilient cooperative systems. We start by introducing the three use cases considered in the evaluation, namely, a live-streaming protocol, a load balancing protocol, and an anonymous communication system. Then, we assess the effort required by a Designer to specify and implement the use cases. Second, we evaluate the capability of *RACOON++* to auto-configure the *CEM*, by measuring the time needed to find a satisfactory configuration in 90 different scenarios. Third, we evaluate the effectiveness of the *RACOON++* cooperation enforcement mechanisms in withstanding the impact of selfish nodes on a set of performance objectives. Finally, we compare the performance of the *CEM*'s accountability mechanism with FullReview, showing that *R-acc++* achieves better results while imposing less overhead than FullReview.

The *RACOON++* framework is provided as a program, which is freely available on the Racoon project website [6]. The framework consists of roughly 8,300 lines of Java code. To facilitate the reproducibility of our results, the implementation of the use cases as well as the configuration files related to the experiments reported in this section can also be downloaded from the project website.

6.5.1 *Use cases*

We consider the following use cases.

17 The rationale behind this selection criterion is that the higher the pScore value of a parameter, the more empirical observations were supporting its choice.

Live Streaming. We consider the P2P live streaming system presented by Guerraoui et al. [72] and already discussed in the previous chapter (Section 5.5.1).

Load Balancing. The heterogeneity of nodes and the high dynamics of P2P systems can lead to a load imbalance.[18] We assume a P2P system in which nodes are allowed to transfer all or a portion of their load among themselves. The goal of a load balancing protocol is to regulate these transfers in a way that evenly distributes the load among nodes, to optimise the use of node capabilities. The load balancing protocol considered as a use case is the one proposed by Jelasity et al. [92].

Anonymous Communication. We consider the basic anonymous communication protocol presented in the previous chapter (Section 5.5.5), based on a simplified version of the Onion Routing protocol for communication channel anonymity [70].

6.5.2 *Design and development effort*

To show the benefits of using *RACOON++* in terms of design and development effort, we present the operations that allow the Designer to specify, develop, and test the use cases.

INPUTS SPECIFICATION. The first step for the Designer is to decide what parts of the system should be included in the *RACOON++* functional specification (i.e., the Protocol Automata). The selected parts should fulfil two criteria. On the one hand, these parts should represent system functionalities that are sensitive to selfish behaviours — specifically, to the deviation types described in Section 6.3.1.2. On the other hand, the selected parts should involve actions that can be observed by other nodes (e.g., a message exchange), to allow accountability audits [76]. Figures 36-38 illustrate the protocol automata defined for our use cases, presented in turn hereafter.

The live streaming protocol (see Figure 36) involves two roles and three protocol steps: the provider r_p proposes the set of chunks it has received to a set of consumers r_C, which in turn request the chunks they need. The protocol ends when r_p sends the requested chunks to r_C.

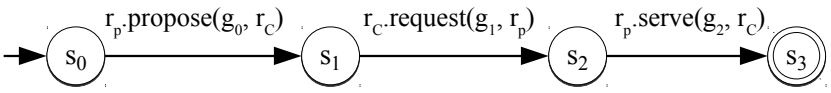


Figure 36: The PA of the live streaming protocol [72].

In the load balancing protocol, each node starts with a certain amount of load. We assume that time is divided in cycles and that each node has a limited amount of load (*quota*) that it can transfer in a given cycle. The basic idea proposed by Jelasity et al. [92] is that each node periodically picks the neighbour that has the maximally different load (larger or smaller) from its local load and has sufficient residual quota. If such a node can be found, then the load transfer is performed. Figure 37 illustrates the Protocol Automaton designed using *RACOON++*. First, the node playing as role r_0 looks in its neighbourhood for a partner R_1 for the load transfer,

18 The load can be measured in terms of different metrics, such as the number of queries received per time unit.

asking the neighbours for their current load and quota. Then, based on this information, r_0 negotiates with R_1 the amount of load to transfer (abstract transition *negotiation*), so that the load transfer can finally take place (abstract transition *transfer*).

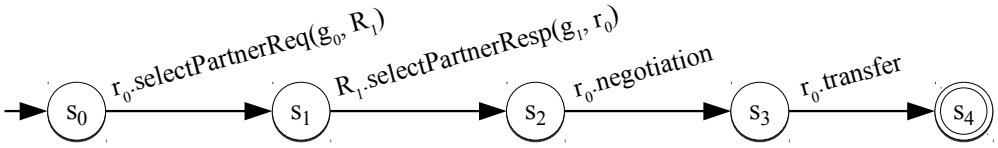


Figure 37: The PA of the load balancing protocol [92].

In the anonymous communication protocol, every time a relay r_r receives an onion message from its predecessors (r_P) in the circuit, r_r decrypts the external layer of the onion, and forwards the resulting onion to the next hops r_N in the circuit. If r_r is the final destination of the onion, then the protocol will end after the *decrypt* transition (state s_2 of Figure 38).

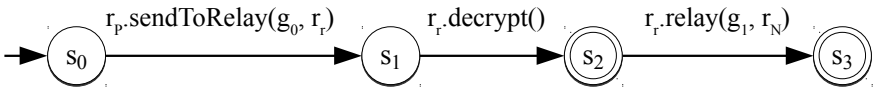


Figure 38: The PA of the anonymous communication protocol.

Once the Designer has provided the functional specification of the system, she defines the selfishness model. For example, consider the anonymous communication protocol. A selfish relay r_r that wants to save bandwidth may strategically avoid to forward onions that are not intended for itself. Concretely, r_r could avoid to relay any onion to its successors (*timeout* deviation) or relay onions only to a subset of them (*multicast* deviation). As another example, consider a selfish provider r_p that wants to participate in the live streaming protocol but limits its bandwidth consumption. A possible strategy for r_p is to propose fewer chunks than it has available (*subset* deviation), or send proposals to only a subset of its neighbours (*muticast* deviation), in such a way as to reduce the number of chunks that could be requested.

Finally, the Designer provides *RACOON++* with a list of design objectives that the system must satisfy. Recall from Section 6.4.2 that an objective can be application-independent or application-specific. Examples of application-specific objectives related to our use cases are (i) a load distribution with a Coefficient of Variation (CoV) close to zero,[19] (ii) a low fraction of onions that do not reach their final destination, or (iii) a low fraction of video chunks that are not played in time.

The Designer provides the *RACOON++* specification inputs as an XML document. The first column of Table 29 illustrates the conciseness of the XML representation of the inputs, showing that the full specification of a use case does not require many Lines of Code (LoC). The full XML specifications are presented in Appendix B.

R-SIM IMPLEMENTATION. The *RACOON++* methodology requires the Designer to implement the functional specification of the designed system in the integrated *R-sim* simulator,

19 The Coefficient of Variation is defined as the ratio of the standard deviation to the mean, and it indicates the extent of variability in relation to the mean of the data set.

Use case	**Specification**	**R-sim Implementation** [a]			
		Std	RS	*TOT*	CM
Live Streaming	55	384	28	*444*	32
Load Balancing	49	232	28	*290*	30
Anonymous Communication	49	212	23	*289*	31

[a] Std = standard operation, RS = *R-sim* functionalities, TOT = Std + RS, CM = custom monitor.

Table 29: Lines of Code needed for the use cases.

based on PeerSim. As discussed in Section 6.4.2, *R-sim* facilitates this task by providing a set of ready-to-use components and an intuitive API to interface a standard PeerSim protocol with the *RACOON++* models and functionalities. For example, the framework includes an implementation of the *CEM*, the algorithms to simulate the rationality of selfish nodes, and native monitors to assess application-independent system performance (e.g., audit precision and recall, bandwidth overhead). These software facilities reduce the number of functionalities to code, allowing the Designer to focus only on implementing the application specific parts of her system, such as the code to implement the correct execution of the protocol and the selfish deviations from it.

The "R-sim Implementation" columns in Table 29 summarise the LoC of the use cases' implementations, distinguishing between the LoC needed to implement the standard operation ("Std" column) from those introduced to invoke the *R-sim* functionalities ("RS" column). The RS lines of code are in the range $6.3\% - 9.6\%$ of the total implementation code (average 8%, standard deviation 0.02), which appears reasonable as it corresponds to only 28 additional LoC, at most. Finally, the column "CM" in Table 29 indicates the size in lines of code of the *R-sim* custom monitors created for each use case. Note that implementing the same monitors without using *R-sim* would require almost 110 additional LoC, with a more than fourth-fold increase in the size of the Java classes.

6.5.3 *Meeting design objectives using RACOON++*

To evaluate the capability of *RACOON++* to find a satisfactory configuration for its cooperation enforcement mechanisms, we performed the following experiment. First, we defined 30 different scenarios for each use case, for a total of 90 scenarios, where a scenario is a unique and random combination of design objectives, system parameters (e.g., number of nodes, message loss rate), application specific parameters (e.g., play-out delay associated with a video chunk, length of a circuit of relays, initial distribution of loads), and fraction of selfish nodes in the system.[20] Second, we used *RACOON++* to find a satisfactory configuration for each scenario,

20 The full specification of each scenario is available on the project website [6]. A specification includes: Protocol Automaton, Selfishness Model, design objectives, and *R-sim* configuration.

while measuring the number of tested configurations and the duration of the process. The results of the experiment are summarised in Table 30.

Use Case	**Configurations tested**				**Time duration** [s]		
	Avg	Median	Range	Not found	Avg	Median	Range
Live Streaming	7.1	4.0	1–56	2	1,954	829	201–12,479
Load Balancing	6.1	3.5	2–25	0	1,151	407	183–9,695
Anonymous Communication	7.4	4.0	1–37	1	61	28	10–282
All use cases	6.8	4	1–56	3	1,046	355	10–12,479

Table 30: Performance of the tuning process of *RACOON++* in terms of time duration and number of configurations tested.

We observe that *RACOON++* evaluates between 1 and 56 configurations before finding a satisfactory one. This wide range is mainly due to the random characteristics of the scenarios generated, which makes meeting the design objectives in some scenarios more challenging than in others. Consider, for instance, the two Live Streaming scenarios compared in Table 31: the objectives set for the scenario that required only one configuration to evaluate (column "LS Scenario #6") are apparently less strict than those of the other scenario ("LS Scenario #12").

Design objective	**LS Scenario #6**	**LS Scenario #12**
(at least) Cooperation level	0.8	0.95
(at most) CEM bandwidth overhead	0.6	0.45
(at most) Chunk loss	0.05	0.03
Configuration tested	1	56

Table 31: Design objectives of two scenarios generated for the Live Streaming (LS) use case.

Although the large variations of number of configurations tested, Figure 39 shows that their distribution is skewed toward low values in all use cases.[21] Thus, we consider the median of 4 configurations tested as a representative indicator of the *RACOON++* performance in meeting the design objectives, rather than the average value of almost 7 configurations.

The column "Not found" of Table 30 indicates the number of scenarios for which *RACOON++* could not find a satisfactory configuration within a set number of attempts (i.e., 100). Overall, the tuning process failed to meet all the design objectives in only three scenarios over 90, which we consider as an acceptable result. The failures were due to too hard constraints on the efficiency and effectiveness of the cooperation enforcement mechanisms, which were expressed as cost overhead and custom performance objectives (such as low video chunk loss rate), respectively. In these cases, *RACOON++* returns to the Designer the tested configuration that has obtained the best performance over the design objectives (see Section 6.4.3). However, if

21 This explains the large difference between average and median in Table 30, especially in the time duration performance, as we will discuss next.

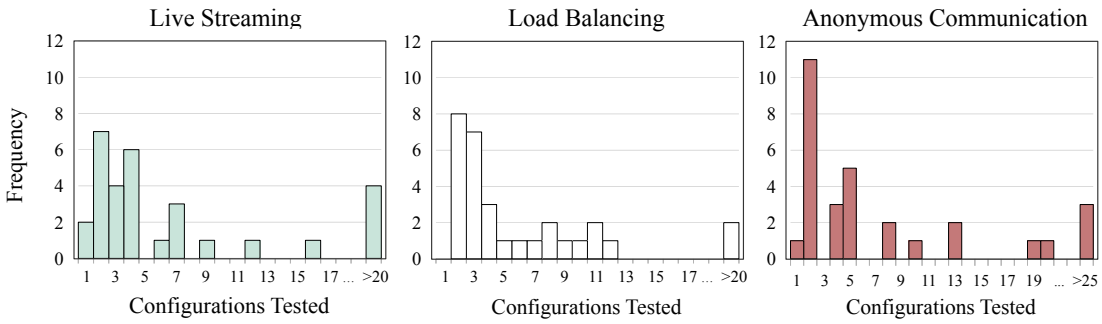


Figure 39: Frequency of the number of configurations tested for each use case.

not satisfied with this outcome, the Designer can either relax the performance requirements or optimise some application-specific operation or parameter. For instance, in the live streaming use case, the Designer could increase the partner set of each node to enhance its possibility to interact with a cooperative partner and thereby receiving more video chunks.

Finally, we analyse the time performance of the tuning process. As reported in Table 30, the median duration of the process is less than 6 minutes (average 18 min, range 10 s–208 min),[22] which we consider as reasonable, as *RACOON++* runs offline at design time. The large time variation of the process is due not only to the number of configurations that have been explored but also to the simulation time of a given scenario, which depends, among the other factors, on the use case as well as the network size. For example, simulating the live streaming use case takes longer than the other use cases, mainly because of the very large number of video chunk objects to generate, disseminate, and process.

6.5.4 *RACOON++ effectiveness*

In this section, we show that the cooperative systems designed using *RACOON++* can effectively foster cooperation as well as achieve application-specific objectives in the presence of an increasing proportion of selfish nodes. To this end, we evaluated 9 scenarios, 3 scenarios per use case, randomly selected from the ones generated for the previous experiment. The Protocol Automaton, Selfishness Model, and *CEM* configuration of each scenario are reported in Appendix B.2.1-B.2.2, whereas the code and *R-sim* configurations are available on the project website [6].

In the first experiment, we assess the effectiveness of the *CEM* in fostering cooperation in the tested systems. The experiment consists of a set of simulations, which monitor the dynamics of 2,000 nodes for 3,000 simulation cycles. We initialize each simulation with an increasing proportion of cooperative nodes (from 0.1 to 1), and we measure the cooperation level achieved at the end of the simulation. Median results in Figure 40 show that the *CEM* succeeds in making the nodes behave cooperatively in all use cases, with the worst results showing a dramatic increase of the cooperation level, from 0.1 to 0.94 (Live Streaming use case)

22 Measures made on a 2.8 GHz machine with 8 GB of RAM.

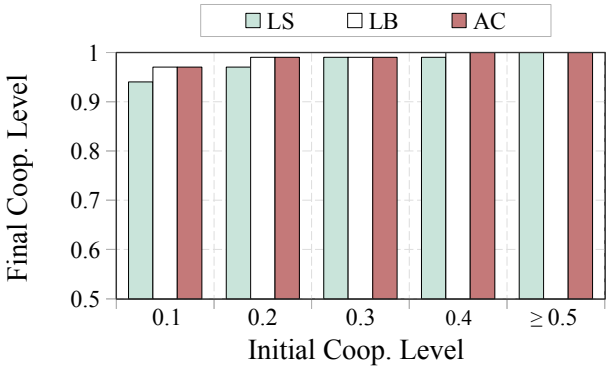


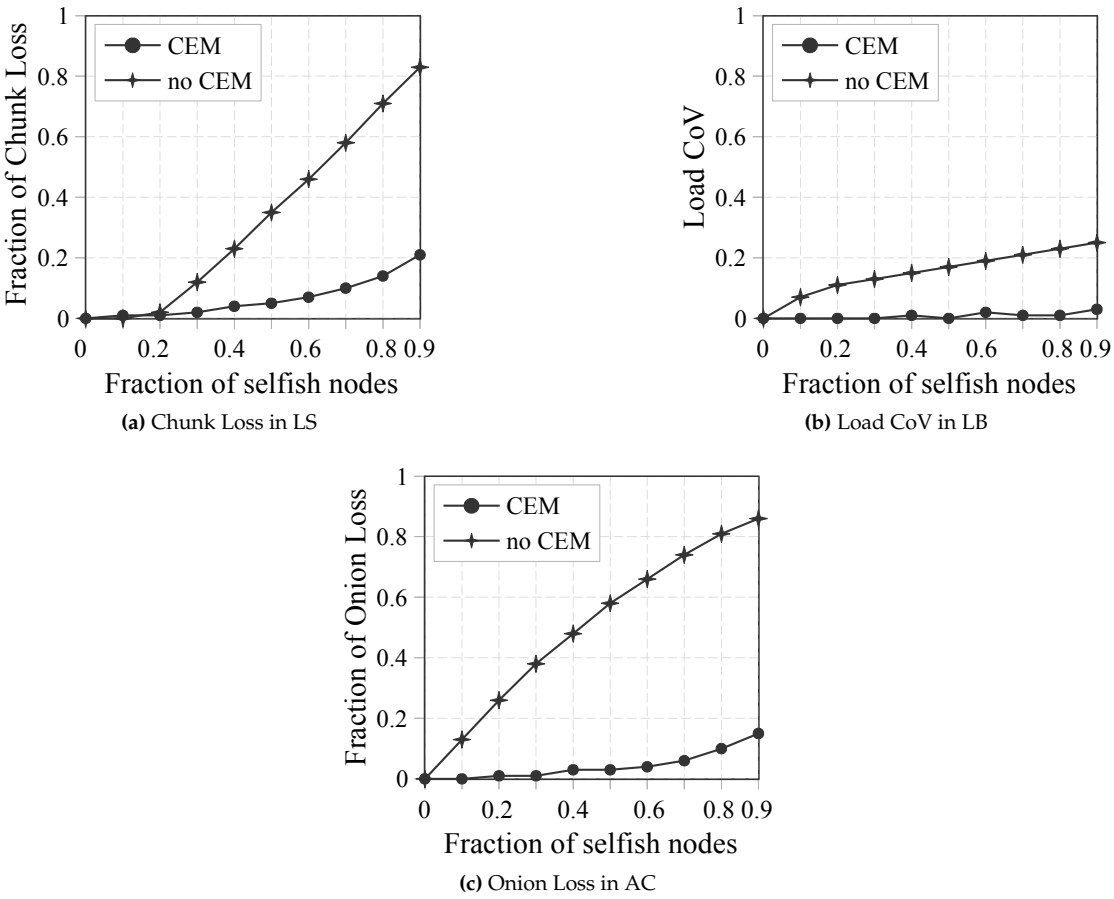
Figure 40: Cooperation levels of the Live Streaming (LS), Load Balancing (LB), and Anonymous Communication (AC) use cases, when varying the initial fraction of selfish nodes.

We now focus on the correlation between cooperation level and application-specific performance. Figures 41(a-b-c) present the median results of our evaluation for the three use cases.

The figures display a curve showing the impact of selfish nodes when no cooperation enforcement mechanism is adopted (curve *no CEM* in the figures), and another curve for the results obtained when using *RACOON++* (curve *CEM*). For example, Figure 41(c) shows that without any mechanism to prevent selfishness the fraction of onions that do not reach destination in the anonymous communication use case increases linearly with the number of selfish nodes in the system and reaches very high values (e.g., 40% of selfish nodes lead to a loss of almost half of the transmitted onions, thereby making the system ineffective in practice). Similar conclusions hold for the number of chunks in the live streaming use case Figure 41(a). The initial cooperation level also has an impact on the performance of the load balancing protocol, which we measured in terms of CoV of the load distribution (the lower the CoV, the better the performance). As we can observe in Figure 41(b), when no mechanism to foster cooperation is in place the CoV increases with the number of nodes that refuse to participate in the balancing protocol. In contrast, the results achieved by the systems designed using *RACOON++* show that the *CEM* can effectively withstand the impact of large populations of selfish nodes.

6.5.5 *RACOON++ vs FullReview*

In this section we present the benefits of using the *RACOON++ CEM* instead of the original FullReview protocols [53]. The main differences between these mechanisms, already discussed in Section 6.3.2, are (i) the approach to punishing selfish and suspect nodes, which is more tolerant in the *CEM*, (ii) the possibility in *R-acc++* to control the probability of auditing other nodes, (iii) the dissemination of proofs of misbehaviour in the system, which in *RACOON++* is realized by *R-rep*. To compare the performance of the *RACOON++ CEM* and of FullReview in our use cases, we initialized the tested systems with a scenario randomly chosen from the set created for the previous experiment. Then, we performed two sets of simulations for each system. In one set we used the *RACOON++ CEM* to foster cooperation, and in the other set we used FullReview. Both the *CEM* and FullReview were optimised for the scenario. In particular,



(a) Chunk Loss in LS

(b) Load CoV in LB

(c) Onion Loss in AC

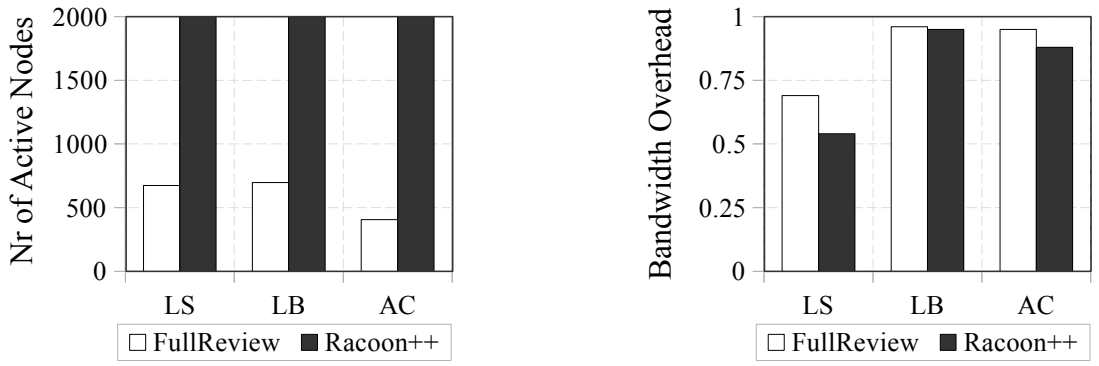
Figure 41: Application-specific performance of the Live Streaming (LS) (a), Load Balancing (LB) (b), and Anonymous Communication (AC) (c) use cases, when varying the initial fraction of selfish nodes.

the *CEM* was automatically configured by the *RACOON++* tuning phase, whereas FullReview was tuned manually.

The first important benefit of using *CEM* is shown in Figure 42(a), which represents the fraction of nodes that are participating in the cooperative system at the end of the simulation. This figure readily illustrates the opposite approaches adopted by *RACOON++* and FullReview to deal with selfishness: *RACOON++* aims to motivate selfish nodes to change their strategy and behave cooperatively, while FullReview operates by isolating non-cooperative nodes. We advocate our approach as the most appropriate for cooperative systems, for two reasons. First, it takes into account the high heterogeneity of nodes and allows low-resource nodes to occasionally behave selfishly because of resource shortages (e.g., low battery in mobile devices). Second, it fits better with the cooperative design principles, which are based on participation and inclusion rather than on punitive restrictions.

On the performance side, Figure 42(b) shows that the *RACOON++ CEM* can decrease the bandwidth overhead in the tested systems, notably by up to 22% in the Live Streaming use

case. This is mainly due to the replacement of the evidence transfer protocol of FullReview with a lightweight reputation system, in which reputation values are exchanged by piggybacking on the accountability protocols messages. Also, the *CEM* allows probabilistic audits, which further reduces the traffic and computation overhead associated with the audit activities. As is apparent from Figure 42(b), the types of cooperative system that gain the most from these optimisations are the ones performing an extensive message exchange, such as P2P live streaming and anonymous communication. In fact, *R-acc++* scales better than FullReview as the communication activities to log increase.



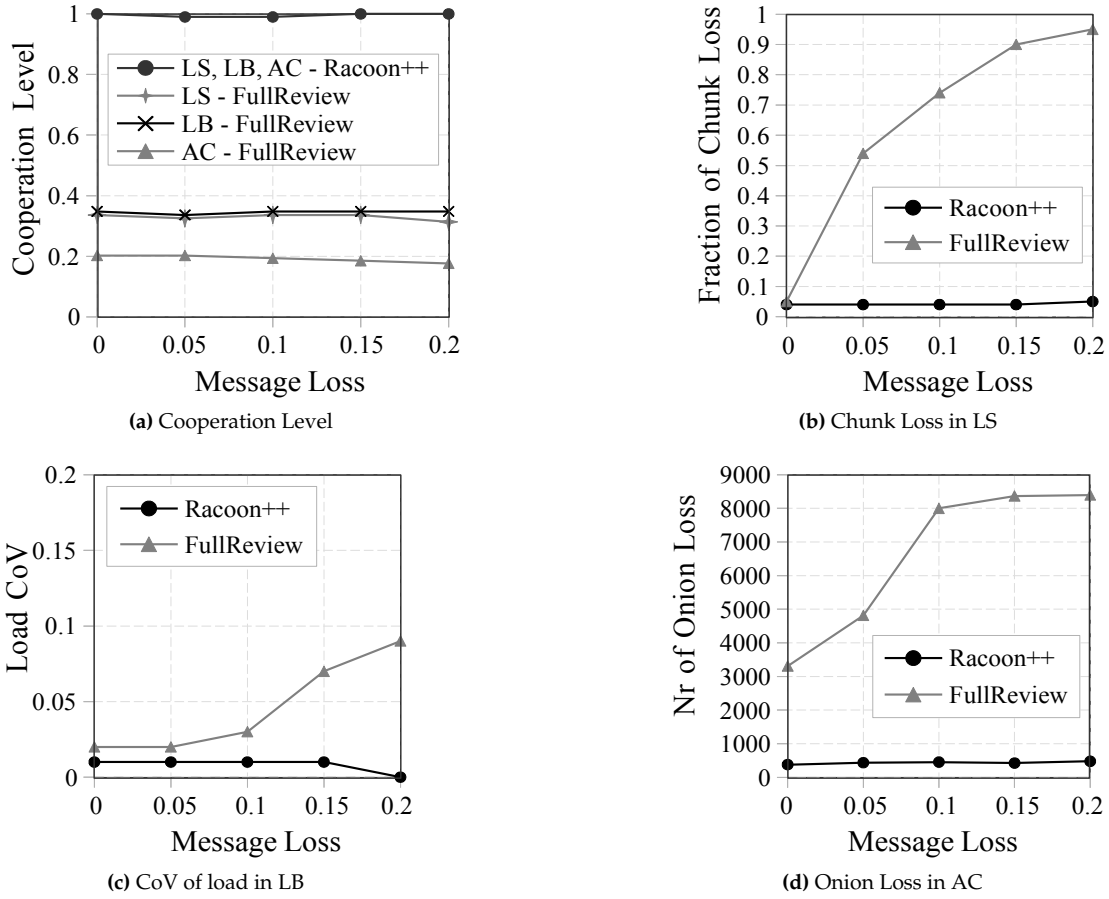
(a) Active Nodes

(b) Bandwidth Overhead

Figure 42: Performance comparisons between FullReview and *RACOON++ CEM* in the Live Streaming (LS), Load Balancing (LB), and Anonymous Communication (AC) use cases.

In the previous experiments, we assumed the systems running over a reliable network. However, as shown in earlier work [106], FullReview is very sensitive to message loss, which can significantly increase the number of suspect nodes, and might even lead to the wrongful eviction of a correct node. We evaluated the robustness of the *RACOON++ CEM* against message loss by assessing the performance of the tested systems when running over an unreliable network with up to 20% message loss. Figures 43(a) illustrates the cooperation levels achieved by the tested systems at the end of the simulations when using the *RACOON++ CEM* and FullReview. The curves show that message loss has a small impact on the cooperation, due to the mitigating effect of the challenge/response protocol used by both mechanisms (see Section 6.3.2). Notice that the FullReview curves in Figure 43(a) confirm what was already discussed for Figure 42(a), that is the dramatic decrease of active nodes because of the extreme punishment enforced by the accountability mechanism. Such performance degradation is much more severe for application-specific objectives, as can be observed in Figures 43(b-c-d). The main reason is the FullReview suspicion mechanism, which prevents a suspect node from interacting with others. Because temporary message loss can trigger node suspicion, the larger the message loss rate, the longer a node could be stuck in a suspect state. Conversely, in the *RACOON++ CEM*, a suspect node can continue to interact with other nodes, though with a lower probability. This gives the suspect node more opportunities to get out of the suspect state by behaving cooperatively, which is also beneficial for the system. The *Racoon++* curves in Figures 43(b-c-d) demonstrate that this simple strategy is enough to guarantee resilience from selfish nodes while being tolerant to mes-

sage loss. Particularly, in the load balancing use case, the CoV of the load distribution decreases as the message loss rate increases (see Figure 43(c)). This is because, on the one hand, message loss does not significantly impair the protocol performance; on the other hand, because the additional penalisation caused by the suspicion mechanism makes selfish nodes more likely to stop deviating and persist in cooperation.



(a) Cooperation Level

(b) Chunk Loss in LS

(c) CoV of load in LB

(d) Onion Loss in AC

Figure 43: Experiment results with different proportions of message loss.

6.6 SUMMARY

In this chapter, we presented *RACOON++*, an enhancement of the *RACOON* framework for the selfishness-aware and performance-oriented design of cooperative systems. It improves most of the functionalities and supporting tools of the original framework, with a special focus on addressing the following issues:

1. *RACOON offers limited customisation of the characteristics of the selfish behaviours to evaluate.* ***RACOON++, instead,*** provides system designers with a simple yet expressive means, the Selfishness Model, to parametrize various aspects of selfish behaviours, including details

on their execution (*who* deviates, from *which step of the protocol*, with *what type* of deviation) as well as the utility obtained by the nodes. To account for this additional information, we designed a new algorithm for the automatic generation and injection of selfish behaviours into the specification of a given system.

2. *The behavioural model used by RACOON is based on classical game theory, which requires restrictive and rather unrealistic assumptions on nodes' rationality.*
   RACOON++, instead, relies on evolutionary game theory, which provides a more suitable behavioural model for making predictions about the strategic and dynamic interactions of nodes in cooperative systems.

3. *RACOON relies on a custom-built simulator, which might hinder the usability and general acceptance of the framework.*
   RACOON++, instead, has been integrated with the state-of-the-art simulator PeerSim.

Table 32 illustrates the improvements of *RACOON++* over *RACOON* with respect to the evaluation criteria presented in Chapter 3 (Section 3.3). The use of the well-established and extensively used simulator PeerSim increases the usability (column "Usa"), performance (column "Sc"), and accuracy (column "Ref") of the framework. Furthermore, the new behavioural model, along with the application-specific information provided by the Selfishness Model, are also beneficial for the accuracy of the framework results, as they decrease the level of abstraction with which the cooperative system is specified and analysed (column "Ref").

	General and cooperative systems criteria [a]					Selfishness criteria [b]				
Approaches	**Usa**	**Rep**	**Ref**	**Sc**	**He**	**Ra**	**D**	**F**	**M**	**C**
RACOON	●●●○○	●●●●●	●○○○○	10^4	Controllable	✓	✓	✓	x	x
RACOON++	●●●●○	●●●●●	●●○○○	10^6	Controllable	✓	✓	✓	x	x

[a] Usa = usability, Rep = reproducibility, Ref = refinement (inverse of abstraction), Sc = scalability, He = heterogeneity.
[b] Ra = rationality, D = defection , F = free-ride, M = misreport, C = collusion.

Table 32: Comparison between *RACOON* and the existing approaches for selfishness analysis.

We evaluated *RACOON++* on three use cases: a P2P live streaming system, a load balancing protocol, and an anonymous communication system. First, we described the operations to specify and implement the use cases, showing that the facilities made available by the framework can effectively reduce the design and development effort required from system designers. Second, we assessed the effectiveness of the *RACOON++* cooperation enforcement mechanisms (accountability and reputation) in achieving application-specific objectives in each use case in the presence of selfish nodes. Third, we evaluated the capability of *RACOON++* to auto-configure the *CEM*, by measuring the time needed to find a satisfactory configuration in 90 different scenarios. Results showed that the process took on average less than 18 minutes to complete. Finally, we compared the performance of the *CEM*'s accountability mechanism with FullReview, showing that the accountability system included in *RACOON++* can achieve

dramatic improvements while imposing less overhead than FullReview.

This chapter concludes the central part of the dissertation, devoted to the presentation of the second contribution of this thesis (C.2), i.e., *RACOON* (along with its extension *RACOON++*), a framework for designing cooperative systems that achieve desired performance objectives in the presence of selfish nodes. For determining the achievement of such objectives, *RACOON* includes an analysis tool to assess the impact of certain types of selfish behaviours on a simulated model of the system. However, as readily apparent in Table 32, the selfish behaviours supported by the *RACOON* and *RACOON++* frameworks do not cover all the types that we identified in the survey on selfishness presented in Chapter 2. Moreover, the scope of the selfishness analysis supported by the *RACOON* frameworks is further restricted to systems that use accountability and reputation mechanisms.

Such limitations motivated the search for a general and comprehensive tool for assessing the impact of selfishness on cooperative systems, which will be the last contribution of this thesis and the subject of the next part.

Part III

SELFISHNESS INJECTION ANALYSIS IN COOPERATIVE SYSTEMS: THE SEINE FRAMEWORK

7

THE SEINE FRAMEWORK

As extensively discussed in the previous chapters, selfishness is one of the key issues that designers of cooperative systems have to deal with. It has the potential to severely degrade the system performance and to lead to instability and failures. For instance, in the peer-to-peer live streaming system described by Guerraoui et al. [72], experiments shown that if a quarter of nodes stop sharing their available resources, then half of the remaining 75% of nodes receive a degraded stream. In the evaluation of the *RACOON* and *RACOON++* frameworks (Chapters 5-6), we presented similar experiments, showing that selfishness is a many-faceted and widespread problem in cooperative systems.

In this context, understanding the impact that selfish behaviours may have on the system is crucial for the design of reliable and effective countermeasures to mitigate such impact. In Chapter 3, we argued that this can be done only by designing and injecting selfish behaviours into the system under consideration, which is a non-trivial task, requiring multi-domain expertise. Indeed, to carry on this task, the system designer has to first analyse the functional specification of the considered system and identify those steps (e.g., functions) for which selfish nodes may behave in a non-cooperative way. Then, on each of the identified steps, the designer has to determine what are the possible selfish behaviours that are meaningful in the context of her application and implement the corresponding behaviours. Finally, the designer has to invest considerable effort in experiments to assess the impact of the introduced behaviours on the performance of the system.

To the best of our knowledge, the *RACOON* frameworks developed in this thesis are the first tools to provide designers with domain-specific support for analysing selfishness in cooperative systems. In fact, the frameworks include a semi-automatic simulation environment that can assess the selfish-resilience of a system against a fixed set of three simple deviations. However, such deviations cover only a small subset of the many possibilities of deviation presented at the beginning of this dissertation, in Chapter 2. Moreover, the deviations need to be manually implemented for the simulators integrated into the *RACOON* frameworks.

To overcome these difficulties and limitations, we propose *SEINE*, a simulation framework for rapid modelling and evaluation of selfish behaviours in a given cooperative system, aimed at supporting the design and testing of selfishness-resilient systems. *SEINE* relies on a *Domain-Specific Language* (DSL), called *SEINE-L*, for describing the behaviour of selfish nodes, along with an annotation library to associate such specification with a system implementation for the state-of-the-art simulator PeerSim [128]. We based our design of *SEINE-L* on the extensive survey on selfishness in cooperative systems that we presented in Chapter 2. *SEINE-L* provides a unified semantics for defining *Selfishness Scenario*s, which allow describing capabilities, interests and behaviours of the different types of nodes participating in the system. The *SEINE* framework provides a compiler and the run-time system supporting the automatic and system-

atic evaluation of different Selfishness Scenarios in the PeerSim simulation framework. Simulations return a set of statistics on the behaviour of the system in the presence of the specified type of selfish nodes.

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

Overall, this chapter aims to make the following contributions:

- We present the design of the *SEINE-L* language and we evaluate its expressiveness by showing that it can capture the semantics of the selfish behaviours described in the papers from the survey.
- We assess the impact of a Selfishness Scenario in a PeerSim simulation. Through the evaluation of three complete use cases, namely, a gossip-based dissemination protocol, a live streaming protocol (i.e., BAR Gossip [111]), and a file sharing system (i.e., BitTorrent [45, 116]), we show that the simulations enabled by *SEINE* are accurate with respect to real measurements performed on these systems.
- We show that *SEINE* facilitates a substantial reduction of the effort required to model, code, and evaluate selfish behaviours in a given system.

Roadmap. The chapter is organised as follows. In Section 7.1 we present the Selfishness Scenario model, based on the survey on selfishness presented in Chapter 2. Section 7.2 provides an overview of *SEINE*, followed by a detailed description of its components: the DSL for modelling a Selfishness Scenario (Section 7.3) and the support tools for injecting it into a PeerSim simulation (Section 7.4). Section 7.5 presents a performance evaluation of *SEINE*, and, finally, the chapter concludes in Section 7.6.

Work presented in this chapter has been accepted for publication in the 47th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2017).

7.1 DOMAIN ANALYSIS

In Chapter 2, we performed a systematic analysis of selfish behaviours in cooperative systems. Given the vast body of literature on the subject, we selected 25 state-of-the-art papers that are of particular interest to the research community and provide detailed and concrete descriptions of selfish behaviours. For ease of presentation, we report fifteen of these papers in Table 33 below (the complete list can be found in Table 2). For each paper, the table indicates some characteristics of the studied system (i.e., application domain, service provided, architecture type), along with further information to characterise the selfishness manifestation therein investigated.

Table 33: Subset of the papers considered in our review, along with the characteristics of the cooperative systems investigated and the types of selfish deviations therein described.

	Cooperative System			*Deviation types* [a]				
Reference	**Domain**	**Name and/or Service**	**Architecture**	D	F	M	C	O
Ben Mokhtar et al. [28]	Data Distribution	File-sharing, live streaming	P2P	✓	x	x	✓	x
Ben Mokhtar et al. [26]	Data Distribution	FireSpam	P2P (struct.)	✓	✓	x	x	x
Guerraoui et al. [72]	Data Distribution	Media streaming	P2P (unstruct.)	x	✓	✓	✓	x
Hughes et al. [87]	Data Distribution	Gnutella (file-sharing)	P2P (unstruct.)	✓	x	x	x	x
Li et al. [111]	Data Distribution	BAR Gossip (live streaming)	P2P (unstruct.)	x	✓	✓	✓	x
Lian et al. [114]	Data Distribution	Maze (file-sharing)	P2P (unstruct.)	x	x	x	✓	x
Locher et al. [116]	Data Distribution	BitThief (file-sharing)	P2P (unstruct.)	✓	x	✓	x	✓
Piatek et al. [148]	Data Distribution	PPLive (live streaming)	P2P (hybrid)	x	✓	x	✓	x
Sirivianos et al. [165]	Data Distribution	Dandelion (file-sharing)	P2P (unstruct.)	✓	x	✓	x	x
Anderson [18]	Computing	BOINC	Client-server	x	✓	x	✓	x
Kwok et al. [104]	Computing	Grid Computing	Client-server	x	✓	✓	x	x
Cox and Noble [47]	Backup & Storage	Samsara	P2P (struct.)	x	✓	x	x	x
Gramaglia et al. [71]	Backup & Storage	P2P Storage	P2P (struct.)	✓	x	x	x	x
Mei and Stefa [124]	Networking	Message switching	DTN	✓	✓	x	x	x
Ngan et al. [135]	Anonym. Comm.	Tor [54]	Proxy servers	✓	✓	x	x	x

[a] D = defection , F = free-riding, M = misreport, C = collusion, O = other types.

The goal of the domain analysis was to identify possible commonalities in the motivations and executions of selfish behaviours, so as to build a domain-specific terminology and semantics for their representation and understanding. The output of such analysis is the *Selfishness Scenario* model, which provides the conceptual framework to characterise interests and capabilities of different types of nodes participating in a cooperative system, as well as possible executions of selfish behaviours. A visual representation of the overall model is given by the feature diagram in Figure 44.[1]

Remark. *The Selfishness Scenario model presents many common features with the classification framework of selfish behaviours introduced in Chapter 2. Nevertheless, the two models differ in their focus and objectives. On the one hand, the classification framework provides the conceptual template to categorise one particular behaviour of a selfish node, based on the node's motivation and a high-level description of the behaviour execution. On the other hand, the Selfishness Scenario aims to describe in greater details the implementation aspects of each selfish behaviour that may affect a given system, in order to enable their replication for (analytical or experimental) analysis.*

Hereafter, we define and discuss each component of the Selfishness Scenario model.

1 The feature diagram in the figure is a cardinality-based extension of the *FODA* notation [49].

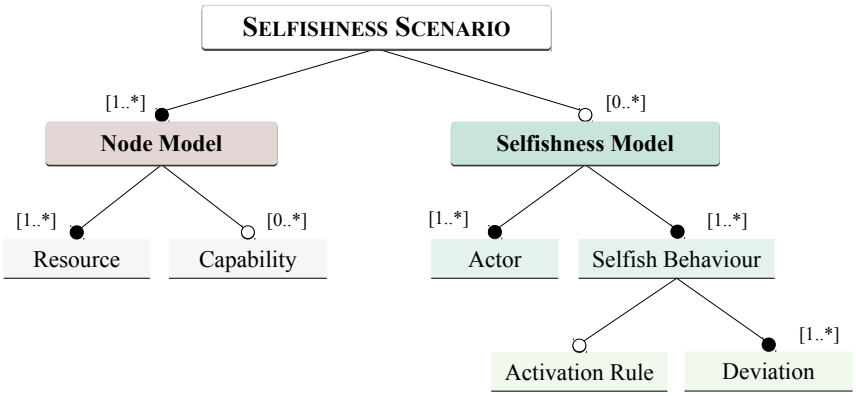


Figure 44: Feature diagram of a Selfishness Scenario.

Definition 7.1 (Selfishness Scenario)**.** A *Selfishness Scenario* is a tuple $\langle N, \Sigma \rangle$, with:

- N: finite, non-empty set of *node models* (Definition 7.2) to describe the types of nodes in the system, and
- Σ: finite set of *selfishness models* (Definition 7.5) to describe the behaviours of selfish nodes.

The participants of a cooperative system constitute a heterogeneous population both in their personal interests and capabilities. A *node model* describes interests and capabilities of each type of participants in terms of resources.

Definition 7.2 (Node Model)**.** A *node model* $\eta \in N$ is a tuple $\langle E, \Pi \rangle$, with:

- E: finite, non-empty set of *resources* of interest (Definition 7.3), and
- Π: finite set of *capabilities* that might hold over the resources in E (Definition 7.4).

Definition 7.3 (Resource)**.** A *resource* $\epsilon \in E$ is a physical or logical commodity that increases the personal utility of the nodes that possess it.

A physical resource represents a node's capacity, such as bandwidth, CPU power, storage space, or energy. A logical resource can be a high-level and application-specific service offered by the cooperative system (e.g., file-sharing, message routing), or the incentive created by a cooperation enforcement mechanism (e.g., money, level of trust).

Definition 7.4 (Capability)**.** A *capability* $\pi \in \Pi$ of a node model $\eta \in N$ is a constraint on a resource $\epsilon \in \eta.E$.

Consider, for example, a cooperative system consisting of mobile and desktop nodes. Mobile nodes usually have lower communication and computation capabilities than desktop nodes.

Then, a Selfishness Scenario may describe this system using two node models (*mobile* and *desktop*) characterised by the same resources of interest (*bandwidth* and *CPU*) but different capabilities (less bandwidth and CPU capacities for mobile nodes).

According to Definition 7.1, a Selfishness Scenario describes the possible behaviours of selfish nodes as a set of *selfishness models*,[2] whereby each selfishness model is characterised by a set of *selfish behaviours* as well as the *actors* that might perform them.

Definition 7.5 (Selfishness Model)**.** A *selfishness model* $\sigma \in \Sigma$ is a tuple $\langle A, B \rangle$, with:

- $A \subset N$: finite, non-empty set of *actors* (Definition 7.6), and
- B: finite, non-empty set of *selfish behaviours* (Definition 7.7).

Definition 7.6 (Actor)**.** An *actor* $\alpha \in (A \subset N)$ of a selfishness model $\sigma \in \Sigma$ is a node model that might exhibit the selfish behaviours described by σ.

In the Selfishness Scenario model, a selfish behaviour is characterised by a set of *deviations* from the correct execution of the cooperative system, along with, possibly, the *activation rule* that motivates an actor to adopt that behaviour.

Definition 7.7 (Selfish Behaviour)**.** A *selfish behaviour* $\beta \in B$ is a tuple $\langle ar, \Delta \rangle$, with:

- ar: the optional *activation rule* (Definition 7.8), and
- Δ: finite, non-empty set of selfish *deviations* (Definition 7.9).

Definition 7.8 (Activation Rule)**.** The *activation rule* ar of a selfish behaviour $\beta \in B$ describes the condition that will trigger each deviation specified in $\beta.\Delta$ to be executed.

Examples of activation rules are exceeding a threshold amount of resource consumption or the delivery of a service (e.g., a file download). Another typical situation is providing false information to a monitoring mechanism to cover up previous deviations. Thus, a selfish behaviour may be the activator of other selfish behaviours.

Definition 7.9 (Deviation)**.** A selfish *deviation* $\delta \in \Delta$ describes a type of deviation from the correct execution of the protocols underlying a cooperative system. A deviation point is a step of a system protocol in which the deviation may take place.

The wide range of motivations behind selfish behaviours, as well as the application-specific nature of their implementation, generates a tremendous number of possible deviations for any

2 If the set of selfishness models is empty, then the Selfishness Scenario describes a cooperative system composed solely of cooperative nodes.

given cooperative behaviour. Nevertheless, based on our review of the available literature, we could identify four recurring types of deviations, named defection, free-riding, misreport, and collusion. As shown in the last columns of Table 33, these types match almost all the selfish behaviours analysed in our review. The only exception is the rarest-first policy for requesting file pieces, which is very specific to the implementation of BitTorrent [116].

A *defection* is an intentional omission in the execution of a system protocol. A selfish node performs a defection to stop the protocol execution, so as to prevent requesters from consuming or even asking for its resources. *Free-riding* is a reduction of resources contributed by a node without stopping the protocol execution. The literature on cooperative systems offers other definitions of free riding, such as the complete lack of contribution [116, 135, 165], or downloading more data than what is uploaded [87]. Our definition is more general because it applies to any resource, and it is more precise because it can be clearly distinguished from deviations that achieve a similar result by stopping the system protocols. A *misreport* is the communication of false or inaccurate information, to avoid contribution or gain better access to resources. Finally, a *collusion* is the coordinated execution of a selfish behaviour by a group of nodes that act together to increase their benefits. Collusions are more difficult to detect than individual deviations [28, 72], because colluders can reciprocally hide their misbehaviours.[3]

7.2 SEINE OVERVIEW

The *SEINE* framework aims to help cooperative system designers to evaluate the impact of selfish behaviours on the system performance. The framework builds on the lessons learnt from the domain analysis presented above, providing designers with modelling and simulation tools to describe and experiment with Selfishness Scenarios in a given system. *SEINE* relies on the PeerSim open-source simulator for large-scale distributed systems [128]. The results of the simulation experiments are the output of *SEINE*.

Figure 45 provides an overview of the *SEINE* framework. To begin, the system designer (hereafter "Designer", for brevity) produces the input files required by the framework, namely, a Selfishness Scenario (*S* in the figure) specified using the *SEINE-L* DSL, a configuration file to set up the simulation (*C*), and a Java implementation of the application protocols used in the system (*P*). The clear separation between selfishness and implementation concerns facilitates the maintenance and reuse of the *S* and *P* artefacts. To associate the DSL declarations to the protocol implementation components (e.g., classes, variables, methods), the Designer decorates such components using a library of *Annotations* provided by the *SEINE* framework.

Upon creating all the input files, the Designer uses *SEINE* to study the behaviour of the system defined by *C* and *P* when faced with the Selfishness Scenario described in *S*. First, the *SEINE-L Compiler* generates the configuration file *C**, which extends *C* with instructions for injecting selfish behaviours into *P* as well as for monitoring the system performance. Second, *SEINE* calls the *Configuration Manager* included in the PeerSim library to build the experiment at run-time via reading *C** and instantiating the specified simulation components. These com-

3 We refer to Section 2.3.2 in Chapter 2 for a more detailed presentation of the deviation types considered in our work.

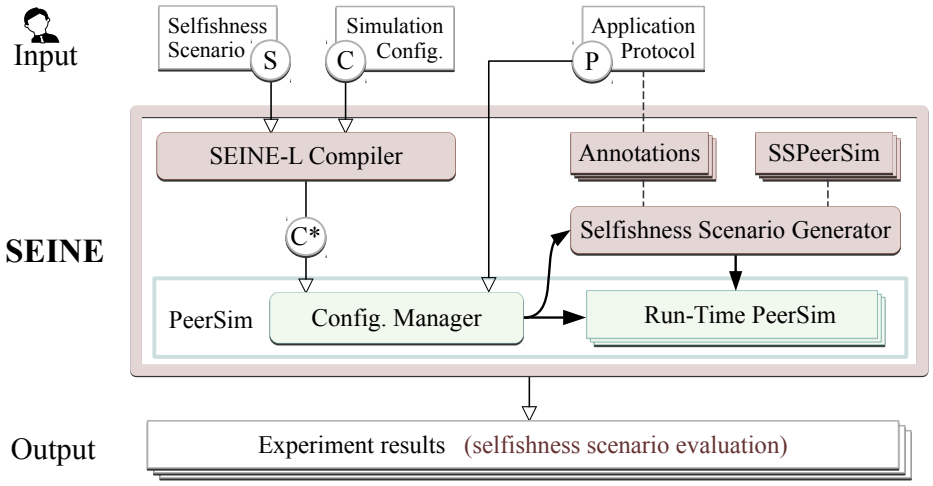


Figure 45: Overview of the *SEINE* framework.

ponents are Java classes that implement (i) the nodes that compose the network, (ii) the set of protocols hosted by each node (including *P*), (iii) control objects to monitor or modify the behaviour of the simulated system, and (iv) the *Selfishness Scenario Generator* that injects the Selfishness Scenario into the simulation run-time. In particular, the Selfishness Scenario Generator uses Aspect-Oriented Programming techniques [61] and relies on a library of Java classes (*SSPeerSim*, in Figure 45) to interact with the PeerSim simulator. Finally, the *SEINE* framework presents the results of the simulation as a collection of statistics describing the behaviour of the simulated cooperative system for the given Selfishness Scenario.

7.3 MODELLING SELFISHNESS IN SEINE-L

SEINE-L provides a clear and concise description of the capabilities, interests and behaviours of different models of node participating in the protocols of a cooperative system. The semantics of the DSL builds on the domain analysis presented in Section 7.1, while its syntax is designed in the style of the PeerSim configuration file to offer a more coherent integration with this simulation framework. In fact, the system designer can write the *SEINE-L* program as a separate file or integrate it directly into a PeerSim configuration file. Concretely, the *SEINE-L* syntax is based on Java property files, i.e., collections of pairs associating a property name to a property value. A formal specification of *SEINE-L* is provided as a context-free grammar, which can be found in Appendix C.

The outline of a *SEINE-L* program is illustrated in Figure 46. The entry point is the keyword `seine` followed by a dot and the name of the Selfishness Scenario. Then, the DSL provides five top-level language constructs: *resource*s of interest, *indicator*s of the system state, *node* models, *selfishness* models, and *observers* to monitor the system behaviour. The declarations of the first four constructs have the format `keyword.[construct_name]`, whereas the observers are defined inside a block of statements in curly braces.

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

Figure 46: The outline of a *SEINE-L* specification.

We illustrate the usage of the *SEINE-L* constructs by describing in detail the specification of a Selfishness Scenario for the live streaming system described by Guerraoui et al. [72] and already presented as use case of the *RACOON* and *RACOON++* frameworks (see Section 5.5.1). Listing 7 shows the *SEINE-L* specification of this scenario, called *LSS*, which we refer to in the remainder of this section.

```
1  seine.LSS { # Live Streaming Selfishness scenario
2    resource.bwCapacity uniform(1000) # kbps
3    indicator.batteryLeft
4
5    node.mobile {
6      fraction 0.3 # fraction of mobile nodes in the system
7      selfish 0.5 # fraction of mobile nodes that are selfish
8      capability bwCapacity(500)
9    }
10   node.desktop { selfish 0.1 }
11   node.exclude 0 # the id of the node for which LSS does not apply
12
13   selfishness.smMobile {
14     actor mobile(0.8) # fraction of selfish mobile nodes and actors of smMobile
15     behaviour.bhvAggressive { # 1st behaviour of smMobile, with activation rule
16       activation batteryLeft < 30
17       freeriding { # characteristics of the deviation of bhvAggressive
18         degree 0.8 # degree of deviation
19         on send_SERVE # deviation point
20       }
21     }
22     behaviour.bhvWeak { # 2nd behaviour of smMobile, always triggered
23       freeriding { degree 0.3 on send_SERVE } # 1st deviation of bhvWeak
24       misreport { degree 0.3 on send_PROPOSE } # 2nd deviation of bhvWeak
25     }
26   }
27   selfishness.smColluders {
28     actor desktop mobile(0.2)
29     behaviour.bhvCollusive {
30       collusion { probability 0.15 }
31     }
32   }
33   observers {
34     period 100 # monitoring period
35     name package.path.LSSObserver # class implementing the monitoring process
36   }
37 }
```

Listing 7: *SEINE-L* specification of a Selfishness Scenario for the live streaming system studied in [72].

In *LSS*, a node can be either mobile or desktop, depending on its device type (lines 5-10). The scenario shows that mobile nodes have severely constrained resources, and, particularly, their upload bandwidth capacity is half of the desktops' capacity (line 8). The selfish behaviours declared in the *LSS* example aim to reduce the bandwidth dedicated to other nodes (lines 13-26 in Listing 7) or non-colluders (lines 27-32).

COMMENTS. Comments begin with `#` and continue to the end of the line, as illustrated, for example, in lines 1, 2, 6 and many others.

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

INDICATORS. An `indicator` declaration specifies a quantifiable attribute of either the system or a node model that depends upon its current state. For instance, the *batteryLeft* indicator in line 3 of Listing 7 can be used to guard the battery level of mobile nodes. Indicators cannot be initialised within a *SEINE-L* program.

NODE MODEL. Each node model is declared using the `node` keyword followed by a name and a block of properties delimited by curly braces. The *LSS* scenario declares mobile and desktop nodes, respectively, in lines 5-9 and line 10. The properties that can characterise a node model are listed below:

- `fraction` is the proportion of nodes in the system that hold this node model. If omitted, the fraction is set evenly by the preprocessor so that all node fractions sum up to 1. For instance, *desktop* nodes in the considered scenario will have the fraction set to 0.7.

- `selfish` is the fraction of selfish nodes within this node model. The default value is 1, i.e., all nodes holding this model are selfish.

- `capability` is a list of constraints over the values of the declared resources. Line 8 of the *LSS* scenario, for example, halves the upload bandwidth capacity of mobile peers, to 500 kbps. The DSL syntax prevents the definition of capabilities on resources that are not specified in the program.

There might be reasons to exclude a given set of nodes from the scope of the Selfishness Scenario, for instance, because they represent special devices or trusted parties. In *SEINE-L*, this can be achieved using the `node.exclude` keywords followed by the identifiers (i.e., integers) of the nodes to exclude. As an example, line 11 of Listing 7 excludes the first node from

the *LSS* scenario, because it represents the streaming source, which is assumed to be always cooperative.

SELFISHNESS MODEL. The `selfishness` declaration specifies the set of selfish behaviours adopted by a certain configuration of node models. Such a configuration is expressed by the `actor` keyword followed by a list of terms, each defining the fraction of nodes of a given model to associate with the selfishness under specification. In Listing 7, the selfish behaviours of the *LSS* scenario described above are grouped into two selfishness declarations, namely, *smMobile* and *smColluders*. The actors of the *smMobile* model are defined in line 14 as 80% of the selfish population of the *mobile* nodes. In practice, given that 15% of nodes in the live streaming system were described in lines 6-7 as selfish mobile nodes (i.e., 50% of 30% of the overall population), the percentage of nodes that adopt the *smMobile* selfishness model is 12%. Consider now the `actor` declaration in line 28 in Listing 7, which associates the *smColluders* selfishness model to two node models, namely, all the selfish *desktop* nodes and 20% of the selfish population of the *mobile* nodes (i.e., those that were not associated with the *smMobile* model in the declaration in line 14). Notice that the fraction of *desktop* nodes could be omitted from the declaration, as the default value is 1.

Each selfish behaviour that constitutes a selfishness model is described by a `behaviour` declaration. A behaviour is a list of selfish deviations from the correct execution of a system protocol; such deviations are strategically interrelated and triggered by the same activation rule, which is introduced in *SEINE-L* by the `activation` keyword. An activation rule defines a condition (e.g., greater-than-or-equal-to) over the current value of a resource or indicator declared in the Selfishness Scenario. The *LSS* specification, for example, indicates in line 16 that every mobile node with a *smMobile* selfishness model switches to a more aggressive behaviour to reduce bandwidth consumption when it is running out of battery (i.e., the battery level drops below 30%). In contrast, if no activation rule is specified, then the selfish behaviour is always triggered. This is the case of the *bhvWeak* (lines 22-25) and *bhvCollusive* (lines 29-31) behaviours. Support for the specification of logical expressions to combine multiple activation rules is left to future work.

A selfish behaviour specifies a non-empty set of deviations from the correct execution of certain steps (deviation points) of the system protocols. The practical specification of these steps, called deviation points, is given in Section 7.4. The *SEINE-L* syntax allows to declare five types of deviations, based on the classification developed from the domain analysis. Each deviation type is introduced by its own keyword, namely, `defection`, `freeriding` (`free-riding` is also accepted), `misreport`, and `collusion`. In addition, *SEINE-L* includes the keyword `other` if none of the previous types applies — e.g., incentive-specific deviations such as whitewashing and Sybil attacks in reputation systems [102], or application-specific deviations such as cheating the rarest-first policy for requesting file pieces in BitTorrent [116].

The execution of a deviation can be further characterised by the following additional properties of the deviation declaration:

- `probability` indicates the probability to deviate if the activation rule of the corresponding behaviour is met. The default value is 1.
- `on` constrains the possible deviation points of a deviation. For instance, in the free-riding declaration of the *bhvAggressive* behaviour (lines 17-20), the `on` property ties the execution of this deviation to the deviation point named *send_SERVE*. Multiple deviation points can be listed as illustrated below, separated by whitespace.

  ```
  on send_PROPOSE send_REQUEST send_SERVE
  ```

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

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

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

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

To conclude, *SEINE-L* also allows a compact declaration for deviations with only one property. For example, the compact declaration for the *collusion* deviation in line 30 of Listing 7 is as follows.

```
collusion.probability 0.15
```

OBSERVERS. The language constructs presented so far focus on the description of a Selfishness Scenario. Also, *SEINE-L* provides a means to set-up monitoring components for assessing the performance of a cooperative system under that scenario. This can be done using the `observers` declaration. In practice, an observer is a Java object that collects statistics on the system performance during its simulation with PeerSim (see Section 7.4 for more details). The `observers` declaration specifies the full class `name`s of each observer object to enable, as well as the `period` between two monitoring events in terms of simulated seconds (the default value is 100). For instance, the *LSS* scenario sets up the periodic execution of the *LSSObserver* object every 100 simulated seconds. This can also be specified using the compact form below.

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

7.4 INJECTING SELFISHNESS IN PEERSIM USING SEINE

The *SEINE* framework comprises the PeerSim simulator [128], a library of annotations for linking the *SEINE-L* specification of a Selfishness Scenario to the source code of the application protocols implemented in PeerSim, a compiler for *SEINE-L*, and a generator of simulation components for modelling, executing and monitoring a Selfishness Scenario in PeerSim. In the remainder of this section, we present each of the novel tools developed for *SEINE*.

7.4.1 *Library of Annotations*

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

The remaining annotation types allow indicating deviation points in PeerSim protocols. Concretely, a deviation point is the Java method that implements the part of the protocol in which one or more deviations may take place. These annotations are named after their deviation type (i.e., `@Defection`, `@Freeriding`, `@Misreport`, `@Collusion`, and `@OtherDeviation`) and target method declarations. As an example, let Listing 8 be a fragment of the PeerSim implementation of the live streaming system to experiment with the *LSS* Selfishness Scenario presented in Section 7.3. The annotations in lines 4 and 8 indicate the deviation points for the corresponding deviation types provided in *LSS*. For instance, the `collusion` deviation in line 28 in Listing 7 is implicitly associated with all the methods that have been annotated with `@Collusion`, such as `send_PROPOSE` in Listing 8.

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

Listing 8: A fragment of the PeerSim protocol implementing the system to associate with the *LSS* scenario in Listing 7.

The arguments of a deviation point might be linked to the execution of a selfish deviation. For instance, the `@Misreport` deviation point in lines 4-6 in Listing 8 has an argument (`cnksId`) that refers to the object representing the information to manipulate (a Java collection data type). Table 34 summarises the argument required from each type of deviation point.

Annotation	**Required argument for the deviation point**	
	Description	Type
`@Defection`	/	/
`@Freeriding`	The object representing the resource to manipulate	numerical [a], collection [b]
`@Misreport`	The object representing the information to manipulate	numerical [a], collection [b]
`@Collusion`	The protocol instance ran by another node and potential colluder	PeerSim protocol [c]
`@OtherDeviation`	/	/

[a] Subclasses of the abstract class `Number` in Java (i.e., the primitive data types `byte`, `short`, `int`, `long`, `float`, and `double`).
[b] Subclasses of the abstract class `Collection` in Java (i.e, implementations of the `List` and `Set` interfaces).
[c] Implementations of the `Protocol` interface in PeerSim.

Table 34: The arguments required for each type of deviation point in *SEINE*.

The annotations used to indicate deviation points can have different attributes, depending on the deviation type that is represented. For instance, the `@Collusion` annotation in Listing 8 (line 4) specifies the attribute `ref_arg`, which indicates what argument of the *send_PROPOSE* method is the reference to the protocol instance that might be ran by a potential colluder. This is the first argument by default. The same attribute is also supported by the `@Freeriding` and `@Misreport` annotation types, identifying the argument of the deviation point that may be affected by the deviation. For instance, the free riding annotation in line 8 of Listing 8 can modify the value of the list of chunks `cnks` to deliver to the requester `req`. Table 35 outlines the attributes of each annotation type supported by *SEINE* to indicate deviation points.

7.4.2 *SEINE-L Compiler*

As shown in Figure 45, the *SEINE-L* compiler performs a source-to-source transformation of the *SEINE-L* specification (*S*, in the figure) into the PeerSim configuration file format.

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

@Defection

Attribute	Type	Default value	Description
`return_value`	String	"null"	The value to return in place of the one resulting from the correct method execution. The parameter accepts the following values: "null" (default), "true", "false", and any numeric value.

@Freeriding, @Misreport

Attribute	Type	Default value	Description
`ref_arg`	int	0	The ordering position (in the method declaration) of the argument that is subject to deviation.
`decrease`	boolean	true	True if the deviation decreases the correct value (if a number) or length (if a data collection) of the argument subject to deviation; false, if the deviation increases the correct value or length.

@Collusion

Attribute	Type	Default value	Description
`ref_arg`	int	0	The ordering position (in the method declaration) of the argument that refers to the protocol instance that might be ran by a potential colluder.
`return_value`	String	"null"	The value to return in place of the one resulting from the correct method execution, if `ref_arg` refers to a colluder. The parameter accepts the following values: "null" (default), "true", "false", and any numeric value.
`mtd_colluder`	String	""	The name of the method to execute in place of the correct one if `ref_arg` refers to a colluder.
`mtd_not_colluder`	String	""	The name of the method to execute in place of the correct one if `ref_arg` does not refer to a colluder.

@OtherDeviation

Attribute	Type	Default value	Description
`mtd`	String	""	The name of the method to execute in place of the correct one.

Table 35: The attributes of the annotation types to indicate deviation points in *SEINE*.

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

7.4.3 *Selfishness scenario generation*

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

At run time, the Configuration Manager gives instructions to the *Selfishness Scenario Generator* to properly instantiate the classes in *SSPeerSim* so as to generate the objects that support the simulation of the Selfishness Scenario. Also, the Selfishness Scenario Generator uses Aspect-Oriented Programming (AOP) [61] techniques to modify the execution of the PeerSim protocol components that have been annotated by the Designer, in such a way as to inject selfish behaviours and mode model capabilities. AOP is a relatively recent programming paradigm, which allows seamlessly integrating cross-cutting functionalities into existing software implementations, in an automatic and reliable manner. As we have already argued in this chapter, we treat selfishness as a separate concern from the main logic of a cooperative system. Therefore, AOP is a good choice for our framework. For coherence with the *SEINE* and PeerSim frameworks, both written in Java, we chose AspectJ [97] as the aspect-oriented language. In AspectJ, cross-cutting behaviours are described in class-like modules, called *aspects*. An aspect includes *advice* constructs for describing code to be inserted at given locations (*joinpoints*) of a standard Java program. Such locations are specified by *pointcut* constructs. An advice can insert the code *before* or *after* such locations, or it can replace existing code (*around* advice). The Selfishness Scenario Generator includes the aspects listed below.

- `SeineProtocolAspect` can extend PeerSim protocol classes by adding fields for storing selfishness-related information (e.g., the name of the node model implemented, the selfish behaviours that can be performed) as well as methods for behaving accordingly to the Selfishness Scenario provided. The pointcut of this aspect intercepts all the classes decorated with the `@Seine` annotation.

- `ResourceIndicatorAspect` replaces getters/setters of the fields decorated with `@Resource` and `@Indicator` annotation types with a new implementation that (i) checks the fulfilment of each activation condition that may trigger a selfish behaviour, and, only for resources, (ii) constrains the values to the range specified by a capability condition.

- `SelfishInjectionAspect` specifies the advice that replaces the correct implementation of an annotated deviation point with that of a selfish deviation. More precisely, first it checks whether the deviation can take place. To this end, the advice verifies two conditions: (i) the node executing the method can perform a deviation of that type (i.e., its node model is associated with a selfishness model that includes the deviation), and (ii) the deviation is currently activated. Finally, if these conditions are satisfied, then the deviation implemented in the advice can be executed; otherwise, the execution proceeds according to the reference implementation.

The code snippet in Listing 9 outlines the integration of deviation code into the *send_SERVE* method by the `SelfishInjectionAspect`. According to the *LSS* Selfishness Scenario presented in Listing 7, this method is a deviation point only for selfish mobile nodes that adopt the *smMobile* selfishness model, i.e., 12% of the overall system population (see Section 7.3).

```
1  @Freeriding
2  public void send_SERVE( ... ) {
3        /** SelfishInjectionAspect advice */
4        boolean can_deviate = /* Verification */ ;
5        if(can_deviate) {
6              /* Execution of the deviation code */
7        }
8        /** End of SelfishInjectionAspect advice */
9
10       /* reference method implementation */ ...
11 }
```

Listing 9: A code fragment representing code injection into the *send_SERVE* method.

Another type of simulation component instantiated by the Selfishness Scenario Generator is the set of observers that monitor and gather statistics on the system performance. The *SEINE* framework helps the Designer in creating application domain-specific observers, by providing in the *SSPeerSim* library an abstract class that defines the methods that need to be implemented. The execution of the Designer's observers is coordinated by a configurable controller that is automatically operated by the Selfishness Scenario Generator. Particularly, the set-up of the controller is specified by the `observers` declaration of the *SEINE-L* program (see Section 7.3).

7.4.4 *SEINE Implementation*

All tools and components in *SEINE* are written in Java. The entire implementation consists of almost 4000 lines of code, not including third-party components (i.e., the PeerSim simulator) and automatically generated code (i.e., the *SEINE-L* parser, built using ANTLR [146]). We developed the *SEINE* framework in a modular and loosely coupled manner, which promotes extensibility and reuse of its core components. For example, the interaction with the PeerSim simulator is implemented in a separate and independent module (the *SSPeerSim* library), which is less than 25% of the entire source code.

The *SEINE* framework is freely available at `https://github.com/glenacota/seine`.

7.5 EVALUATION

In this section, we demonstrate the benefits of using *SEINE* to describe selfish behaviours and evaluate their impact on cooperative systems. We start by assessing the generality and expressiveness of the *SEINE-L* language by outlining some of our experiences in describing Selfishness Scenarios with our DSL. Then, we evaluate the accuracy of the *SEINE* output, developing and

testing three use cases selected from our literature review, namely, a gossip-based live streaming protocol, a selfish-resilient media streaming protocol, and a selfish client for the BitTorrent protocol. Also, we assess the effort required by a Designer to implement and test these use cases. Finally, we show that *SEINE* imposes a small time overhead on the normal execution of the PeerSim simulator.

To facilitate the reproducibility of our results, the configuration files related to the experiments reported in this section are available for download on the project page on GitHub.

7.5.1 *Generality and expressiveness of SEINE-L*

We have used *SEINE-L* to express the Selfishness Scenarios described in the fifteen studies from the domain analysis reported in Table 33. Many of these works present various strategies to save bandwidth in data distribution applications, such as Gnutella [87], BitTorrent [116], and PPLive [148]. Other works investigate selfishness in different domains, like the paper of Kwok et al. [104] that studies Grid computing systems, and specifically the impact of task dispatching policies within a Grid site that allocate resources only to local tasks. Overall, the number and variety of the cooperative systems considered, as well as the different degrees of complexity of the Selfishness Scenarios therein described, demonstrate the general applicability and the expressive power of *SEINE-L*.

Table 36 shows that *SEINE-L* files are concise: the Selfishness Scenarios specified are between 14 and 37 Lines of Code (LoC), with an average of 25 LoC. The complete specification of each Selfishness Scenario can be found in Appendix D

Reference	LoC	Reference	LoC
Ben Mokhtar et al. [28]	29	Sirivianos et al. [165]	24
Ben Mokhtar et al. [26]	34	Anderson et al. [18]	17
Kwok et al. [104]	24	Cox and Noble [47]	14
Guerraoui et al. [72]	19	Gramaglia et al. [71]	35
Hughes et al. [87]	20	Mei and Stefa [124]	28
Li et al. [111]	37	Ngan et al. [135]	30
Lian et al. [114]	17	Piatek et al. [148]	18
Locher et al. [116]	23		

Table 36: Lines of Code for expressing the Selfishness Scenarios of the papers considered in the domain analysis review.

7.5.2 *Accuracy of SEINE-R*

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

7.5.2.1 *Live Streaming*

We consider the gossip-based streaming system presented by Guerraoui et al. [72] and already described in Section 7.3. Despite its simplicity, this system is realistic enough to serve as a representative example of a practical live streaming application. Guerraoui et al. deployed the system over PlanetLab,[4] in which a source node streams video chunks with a bit rate of 674kbps to 300 nodes having upload bandwidth limited to 1000kbps. They tested one scenario with only cooperative nodes and another scenario with a quarter of the nodes being selfish, following the selfishness model described in Section 7.3. To assess the system performance in both scenarios, Guerraoui et al. considered the fraction of cooperative nodes perceiving a clear stream (i.e., viewing at least 99% of the streamed chunks) when varying the playout deadline from 0 to 60 seconds.

We developed this gossip-based live streaming system as a PeerSim protocol, and we used *SEINE* to describe and simulate the same Selfishness Scenario as well as the same experiment setting as investigated by Guerraoui et al. [72]. We ran ten simulations for each set of parameters, obtaining a fairly low standard deviation (0.02 on average), and we used the mean value to compare with the reference results. Figure 47 shows the high level of accuracy of *SEINE-R*, with an almost perfect match of the curves representing the scenario with selfish nodes and a remarkable correspondence also in the selfish-free scenario. [5] Note that in the latter scenario our results show high accuracy when the playout deadline is above 7 seconds, whereas, below this time limit, *SEINE* simulations indicated better results. This is mainly due to the lower, real-world reliability of the PlanetLab network compared with the perfect but simulated reliability of the PeerSim network.

7.5.2.2 *BAR Gossip*

Proposed by Li et al. [111], BAR Gossip is a P2P live streaming system designed to tolerate both Byzantine and selfish peers. To this end, BAR Gossip includes mechanisms to enforce cooperation, namely, verifiable partner selection and data exchange mechanisms that make non-cooperative behaviours detectable and punishable. We select this use case to show that *SEINE* can also be used as a tool for testing performance and robustness of selfish-resilient protocols. For instance, BAR Gossip has been proven vulnerable to colluding nodes [28], which exchange video chunks off-the-track among each other, thereby decreasing the system efficiency and particularly the streaming experience of non-colluding nodes.

4 PlanetLab: https://www.planet-lab.org/
5 Our results are plotted over a copy of the figure published in [72].

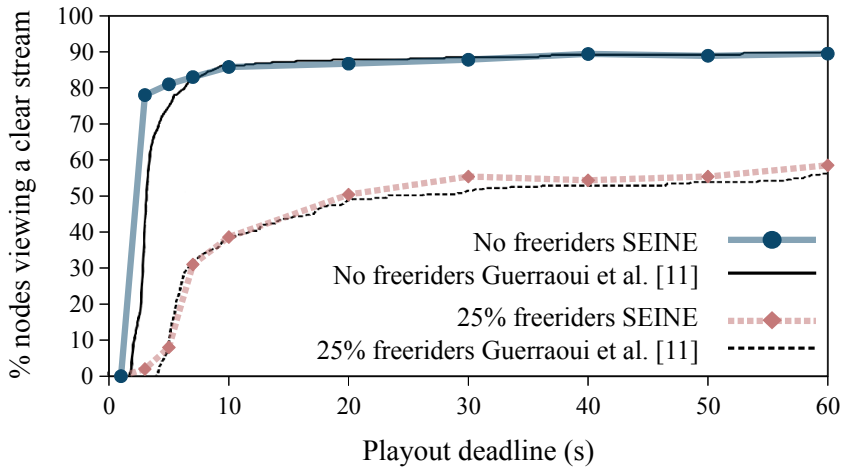


Figure 47: Comparison between the results published by Guerraoui et al. [72] and the results obtained with *SEINE*.

We assessed the accuracy of *SEINE* in reproducing the BAR Gossip Selfishness Scenario that was presented and experimentally studied by Ben Mokhtar et al. [28]. That study deployed 400 nodes in the Grid'5000 testbed,[6] each node running either a compliant version of BAR Gossip or a collusion-enabled implementation. We developed BAR Gossip [111] in PeerSim and set up its protocols using the configuration reported by Ben Mokhtar et al. [28]. Then, we simulated the system when varying the proportion of colluding nodes, from 0 to 50, and we measured the fraction of missed updates by cooperative nodes. Again, we ran ten simulations for each setting, obtaining an average standard deviation below 0.01. As clearly depicted in Figure 48, the accuracy of the results output by *SEINE-R* with respect to the ones provided by the authors of the reported study is very high (0.996 Pearson correlation score). The gap between the results when the proportion of colluders is above 50% is due to some missing parameters in the description of the experiment setting, especially the maximum of chunks that can be exchanged during the optimistic push protocol.

7.5.2.3 *BitThief*

Locher et al. [116] developed and released software to download files in the BitTorrent protocol without uploading any data. Specifically, they implemented and openly distributed a selfish client, *BitThief*, capable of attaining fast downloads without contributing. The purpose of this use case is twofold. First, it shows the accuracy of *SEINE* on empirical experiments performed on real-world applications. Second, it proves the simplicity of using *SEINE* when a PeerSim implementation of the system is already available. Specifically, we used the BitTorrent code published on the PeerSim project website.[7]

We used *SEINE* to reproduce the real world experiment described by Locher et al. [116], which consists in monitoring the download times for one torrent in a BitTorrent network with

6 Grid'5000: https://www.grid5000.fr/
7 http://peersim.sourceforge.net/code/bittorrent.tar.gz.

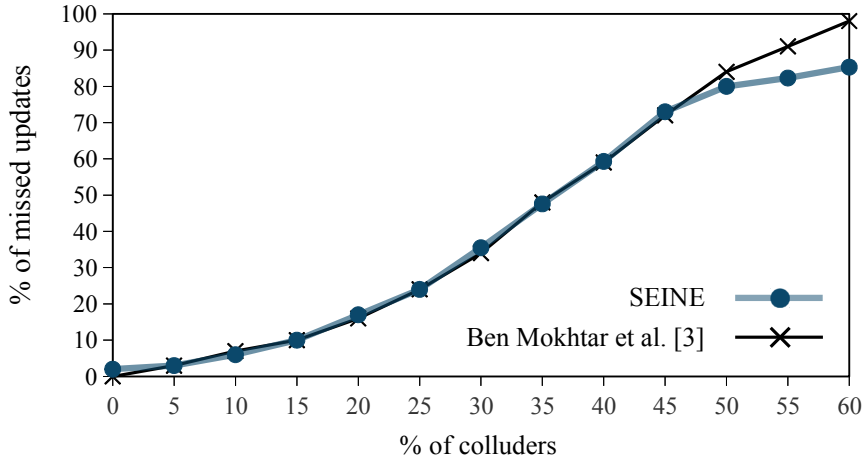


Figure 48: Comparison between the results published by Ben Mokhtar et al. [28] and the results obtained with *SEINE*.

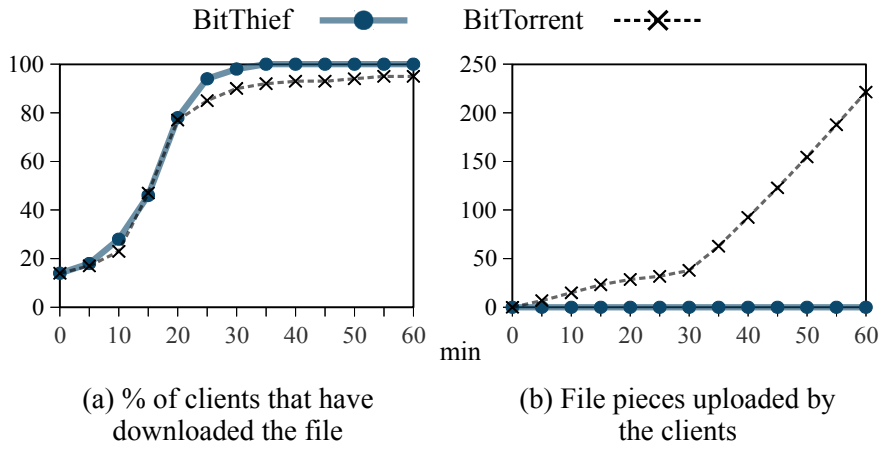


Figure 49: Performance and contribution of BitTorrent and BitThief when downloading the same file, measured using *SEINE*.

5% BitThief clients. BitThief exploits several features of the BitTorrent protocol by means of a set of selfish deviations from the default client implementation. For example, a BitThief client can open up to 500 connections with other peers (the default value is 80), to increase its probability of receiving useful file pieces. In their experiment, Locher et al. showed that such deviations allow BitThief clients to download the file with performance comparable to (if not better than) the default clients, while not contributing any file pieces to other peers. The same result has been obtained in our experiment using *SEINE*, as can been observed in Figures 49(a)-(b).

7.5.3 *Development effort*

To show the benefits of using *SEINE* in terms of design and development complexity, we discuss the effort required to describe, implement and maintain selfish behaviours in our use cases.

Using *SEINE*, the specification of the Selfishness Scenario to test is clearly separated from its actual integration into the cooperative system code. Such separation of concerns facilitates the description and maintenance of node models and selfish behaviours, allowing the Designer to focus only on the *SEINE-L* program and on a few annotations of the system code. On the contrary, without using *SEINE*, the Selfishness Scenario must be hard coded into the PeerSim Java programs and configuration files.

As can be noted from Figure 50(a), implementing a Selfishness Scenario using *SEINE* is not only easier but also extremely concise regarding Lines of Code. The bars in the figure provide a graphical representation of the volume and distribution of code to modify with respect to the faithful implementation of the cooperative system (the white part of the bar). The figure also reports the exact number of LoC to add for each use case. We can derive two observations from these results. First, implementing a Selfishness Scenario using *SEINE* requires four to five times less code to write than without using our framework. Second, when not using *SEINE*, such implementation (i.e., parameters, variables and methods) is scattered across the code of the PeerSim Java program and configuration file; on the other hand, the annotation library included in *SEINE-R* requires the Designer only to annotate existing fields and methods of the PeerSim program, and to write a *SEINE-L* program directly into a configuration file.

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

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

Figure 50(b) illustrates the number of Lines of Code to modify (i.e., add, remove, or edit) in each use case to implement the modifications listed above. Using *SEINE*, modifying the Java class requires modification of at most 1 line, which corresponds to inserting or dropping an annotation. Furthermore, when updating the configuration file, the Designer operates on coherent and consecutive blocks, such as the selfishness model block. In contrast, as can be observed in Figure 50(b), implementing the modifications to the Selfishness Scenarios when not using *SEINE* leads to more LoC to modify, which are scattered across the sources.

To demonstrate how *SEINE* facilitates the fast development and testing of different Selfishness Scenarios, we present two test cases for the BAR Gossip cooperative system. In the first test case, we start from the Selfishness Scenario described in [28] and discussed in Section 7.5.2, and we evaluate the impact of the size and number of colluding groups. More precisely, we fix the fraction of selfish nodes in the system to 20%, and we evaluate the streaming quality perceived by selfish and cooperative nodes when varying from one big colluding group to 10 smaller groups of equal size. Results depicted in Figure 51(a) show that the percentage of updates missed by selfish nodes increases as they form colluding groups of smaller size. This is due to the lower probability for colluders to meet, given the random nature of the underlying

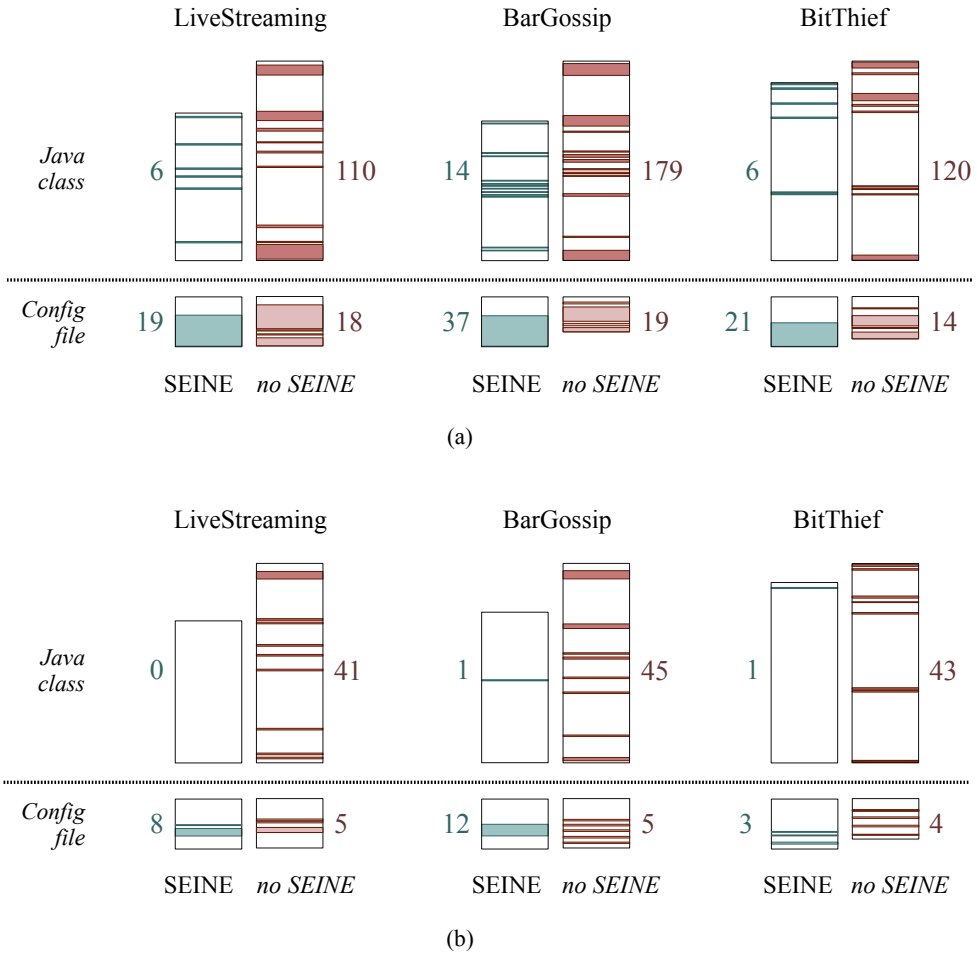


Figure 50: Number and distribution of Lines of Code (a) to specify the Selfishness Scenario into the faithful implementation of the use cases and (b) to modify such scenarios, with and without using *SEINE*.

gossip protocol for chunk dissemination, which cannot be cheated in BAR Gossip by design. On the contrary, the absence of significant changes in the performance for cooperative nodes indicates that they are not affected by how colluders organise themselves into groups. The system Designer can implement this test case using *SEINE* without modifying a single line of code in the PeerSim implementation of BAR Gossip, but only operating on the *SEINE-L* code. Specifically, the Designer first duplicates the selfishness model describing the collusive behaviour as many times as the number of colluding groups she wants to create. Then, the Designer modifies the fractions of the `actor` declarations, in such a way that they sum to one.

As a second test case, we investigate the impact of mobile nodes with lower bandwidth capabilities on the performance of BAR Gossip. Similarly to the previous test case, this scenario modification does not change the system implementation but only the *SEINE-L* description of the Selfishness Scenario. In particular, the Designer has to create a new node model block (i.e., *mobile*) which limits the bandwidth capacity with respect to the original node model (i.e., *desk-*

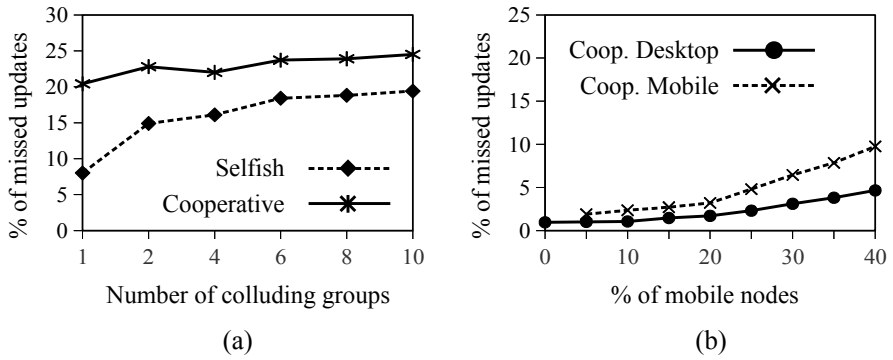


Figure 51: Performance of BAR Gossip when varying (a) the number of colluding groups and (b) the fraction of resourceless mobile nodes.

top). For example, the bandwidth capacity of desktop nodes is 1000 kbps, whereas it is capped to 300 kbps for mobile nodes. This modification to the scenario corresponds to adding 4 lines to the *SEINE-L* program and changing a few numeric values (e.g., refactoring the fraction of desktop nodes). Figure 51(b) reports the results of this test case, showing that the percentage of updates that are missed by cooperative nodes decreases as the fraction of mobile nodes increases. This result can be explained by the lower contribution that mobile nodes make to the chunk dissemination protocols, due to their limited resource capabilities.

To conclude, all these test cases demonstrate how the development and testing of cooperative systems greatly benefit from the *SEINE* functionalities. Evaluating a new Selfishness Scenario in a complex system like BAR Gossip only takes less than an hour, including the simulation time, for instance.

7.5.4 *Simulation time*

In this section, we evaluate the extra execution time imposed by *SEINE* on the regular PeerSim performance. To this end, we defined 10 different Selfishness Scenarios for each use case described in Section 7.5.2, and we ran 40 simulations for each scenario. The results, summarised in Table 37, show that *SEINE* imposes an average extra execution time of 5% (standard deviation 0.03), ranging from the 1.6% extra time achieved by the BAR Gossip use case to the 7.8% extra time of BitThief. Such a short duration increase — 11 seconds out of 154 seconds, at most — appears to be very reasonable in light of the benefits provided by the *SEINE* framework.

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

Table 37: Average execution time to evaluate a Selfishness Scenario using *SEINE* and the additional time it imposes.

7.6 SUMMARY

In this chapter, we presented *SEINE*, a semi-automatic framework for fast modelling and evaluation of the impact of selfish behaviours on cooperative distributed systems. The framework builds on the lessons learnt from the survey on selfishness presented in Chapter 2, providing system designers with modelling and simulation tools to describe and experiment selfishness in a given system.

At the heart of *SEINE* is the *Selfishness Scenario* model, which can capture the characteristics of the various models of nodes and behaviours in a cooperative system. Based on this model, we developed *SEINE-L*, an expressive domain-specific language to describe selfishness scenarios in a clear and concise way. A *SEINE-L* specification allows describing interests and capabilities of nodes in terms of system resources (e.g., bandwidth capacity, battery level) as well as implementation details of the selfish behaviours that they may take. In particular, a selfish behaviour in *SEINE-L* is specified as a set of deviations triggered by an activation condition (e.g., a threshold over the value of a resource), where each deviation has its own characteristics, such as type, target and intensity. Once a *SEINE-L* specification of the selfishness scenario to analyse is provided, the *SEINE* framework proceeds with its evaluation through simulation experiments. To this end, the framework provides a compiler and the run-time system supporting the automatic and systematic evaluation of different selfishness scenarios in the state-of-the-art PeerSim simulation framework. Simulations return a set of statistics on the behaviour of the system when in the presence of the specified model of selfish nodes. The simulation results are also the output of the *SEINE* framework.

We demonstrated the benefits of using *SEINE* by evaluating the framework in four aspects. First, we evaluated the generality and expressiveness of *SEINE-L*, showing that the language can capture the semantics of 38 selfish behaviours described in fifteen papers from the literature. Second, to assess the accuracy of our framework, we used *SEINE* to develop and test three use cases selected from the literature (i.e., a gossip-based live streaming protocol, a selfish-resilient media streaming protocol, and a selfish client for the BitTorrent protocol), demonstrating that the *SEINE* outputs were accurate with respect to real measurements performed on these use cases. Third, we showed that *SEINE* could effectively reduce the effort required by a designer to implement and test selfishness scenarios. Particularly, we evaluated the effort both quantitatively, in terms of lines of code (almost an order of magnitude lower when using *SEINE*), and qualitatively with concrete examples. Finally, we showed that *SEINE* imposes on average 5% of time overhead on the normal execution of the PeerSim simulator, which we believe is reasonable in light of the other benefits described above.

Part IV

CONCLUSIONS AND FUTURE WORK

8

CONCLUSIONS

8.1 SUMMARY

Selfishness is one of the key problems that confront designers of cooperative systems such as peer-to-peer (P2P) live streaming systems and file-sharing networks. Indeed, selfish nodes have the potential to severely degrade the system performance and to lead to instability and failures. For example, in experiments conducted using the tools developed in this thesis, we showed that if 25% of nodes participating in P2P live streaming stop sharing the video chunks they have downloaded, then half of the remaining nodes are not able to view a clear stream (Chapter 7).

Despite the magnitude of the problem, current techniques for predicting the behaviour of selfish nodes, understanding their possible influence on the system, and designing cost-effective countermeasures remain manual and time-consuming, requiring multi-domain expertise.

The research reported in this thesis aimed to further the understanding of selfish behaviours in cooperative systems, as well as to provide conceptual and practical tools for helping designers cope with the associated challenges. In the introductory chapter of this dissertation, we identified two categories of challenges: the first category (A) aiming to provide support for understanding, modelling and evaluating selfish behaviours in cooperative systems, and the second category (D) aiming to facilitate the design and configuration of systems that meet targeted cost-benefit trade-offs in the presence of selfish nodes. Motivated by these challenges, we made three distinct but interrelated contributions, which are addressed in the three parts of the dissertation as summarised below.

Part I: Selfishness in cooperative distributed systems

(C.1) A survey on selfishness and its countermeasures in cooperative systems.

The first part presented the background and motivation for our research through a comprehensive survey of existing work on selfishness and its countermeasures. In Chapter 2, we focused on characterising the behaviour of selfish nodes in cooperative systems. After introducing the concepts and definitions used in the rest of the thesis, along with practical examples from various application domains, we described a classification framework to specify and compare selfish behaviours. We illustrated the general applicability of this framework in four detailed use cases, namely, a P2P live streaming system, the BOINC volunteer computing infrastructure, a message propagation protocol for delay tolerant networks, and the Tor anonymous communication network. We used the classification framework for conducting a systematic analysis of selfishness manifestations in the selected literature. Results from this analysis provided valu-

able insights to understand what are the most common and important selfish behaviours to focus on when designing a cooperative system.

In the two remaining chapters of the first part, we reviewed previous work that has attempted to address selfish behaviours. We started in Chapter 3 by presenting analytical and experimental approaches suitable for the analysis of selfishness in cooperative systems. Among the analytical approaches, game theory was discussed in greater detail, due to its important role in the technical contributions of this thesis. The chapter concluded with a comparative evaluation of the presented approaches. It showed that no satisfactory approach is presently available to conduct a comprehensive, usable, and reliable analysis of selfishness in cooperative systems. This result provided the first motivation for the tools proposed in Part II.

Chapter 4 then focused on the study of two state-of-the-art countermeasures against selfish nodes, i.e., incentive mechanisms and accountability, selected based on their importance in the literature and relevance to our technical contributions. To begin, we described different incentive schemes to foster cooperation (resp. discourage selfishness), grouped into reciprocity-based and economy-based schemes. Then, from the subsequent comparative evaluation of these schemes, we found that reciprocity-based incentives — notably, reputation systems — are the most suitable for dealing with various types of selfish nodes in any cooperative system. In the second part of the chapter, we showed that the reliability and robustness of a reputation system could be greatly improved if paired with accountability techniques. Hence, we concluded Chapter 4 presenting the basic principles of accountability, along with a detailed description of the FullReview system developed by Diarra et al. [53]. Finally, we discussed the cost and configuration issues of accountability, which introduced the second motivation for the framework proposed in the next part of the dissertation.

Part II: Selfishness-aware design of cooperative systems

> **(C.2)** *RACOON*/*RACOON++*: a framework for the selfishness-aware design of cooperative systems.

In the second part of the dissertation, we described the second contribution of the thesis: *RACOON*, an integrated framework for the selfishness-aware and performance-oriented design of cooperative systems. *RACOON* puts into practice the knowledge about selfishness gained in the previous part, bringing together different approaches with the goal of facilitating the enforcement and configuration of efficient mechanisms to foster cooperation in any system deployed over a network of selfish nodes.

The *RACOON* framework was presented in Chapter 5, which provided a detailed description of each step of its largely automated methodology. The operation of *RACOON* consists of two phases: the assisted *design* of the cooperative system and the performance-oriented *tuning* of its parameters. To begin, the system designer provides the functional specification of the system (Protocol Automaton), along with a set of selfish-resilience and performance objectives that the system should satisfy. *RACOON* then augments the specification with (i) two cooperation enforcement mechanisms, namely, a reputation system and an accountability sys-

tem based on FullReview, as well as (ii) possible deviations, drawn from a fixed pool of selfish behaviours. Then, during the tuning phase, *RACOON* evaluates different configurations of the cooperation enforcement mechanisms until it finds the configuration that allows achieving the specified objectives. Such evaluation is performed through simulations that use classical game theory analysis to drive the behaviour of selfish nodes. Overall, *RACOON* results in the redesign of the system specification that includes finely tuned mechanisms to meet the selfish-resilience and performance objectives set by the designer. We illustrated the benefits of using the *RACOON* framework by designing a P2P live streaming system and an anonymous communication system. Experiments in simulation and (at a smaller scale) in a real deployment on Grid'5000 showed that *RACOON* could greatly simplify the design and configuration of the use cases.

In the beginning of Chapter 6, we identified three major weaknesses of the *RACOON* framework, both theoretical and practical: (i) it considers a predefined set of selfish behaviours, which are fixed and poorly customizable, (ii) it relies on classical game theory to predict the behaviours of selfish nodes, which requires making some strong assumptions on the capabilities of nodes, and (iii) it relies on a custom-built simulator. Motivated by these weaknesses, in the remainder of the chapter, we presented an enhanced and extended version of the framework, called *RACOON++*. *RACOON++* allows system designers to parametrize various aspects of selfish behaviours, including details on their execution (*who* deviates, from *which step of the protocol*, with *what type* of deviation) as well as the utility obtained by the nodes. Furthermore, *RACOON++* uses evolutionary game theory to better represent the dynamic behaviour of cooperative systems, whereby selfish nodes can change their strategy over time through learning and imitation. Finally, *RACOON++* relies on the state-of-the-art open source simulator PeerSim, thus improving the accuracy, reproducibility and performance of the original framework. We evaluated *RACOON++* on three use cases: a P2P live streaming system, a load balancing protocol, and an anonymous communication system. The evaluation showed that the systems designed using *RACOON++* could maintain a good behaviour as the rate of selfish nodes raised.

To summarise, the *RACOON* framework provides a general and semi-automatic methodology, along with its software implementation, for designing cooperative systems that achieve desired performance objectives in the presence of (particular types of) selfishness. To determine whether or not these objectives are met, *RACOON* includes an analysis tool based on game theory and simulation, which is used to predict and automatically inject selfish behaviours into the simulated system. However, despite some improvements made in *RACOON++*, the types and characteristics of the selfish behaviours supported for evaluation are quite limited if compared with those identified during the survey reported in Chapter 2. Moreover, the scope of the selfishness analysis supported by *RACOON* is further limited to systems that enforce a particular type of incentive and accountability techniques, i.e., those automatically enforced by the framework. Such limitations motivated the search for a more general and comprehensive analysis tool, which is the last contribution of this thesis and the subject of the next part.

Part III: Describing and injecting selfish behaviours in cooperative systems

(C.3) *SEINE*: a framework for describing and injecting selfish behaviours in cooperative systems.

The third part of the dissertation, consisting of Chapter 7, presented *SEINE*, a semi-automatic framework for fast modelling and evaluation of selfish behaviours in cooperative systems. *SEINE* relies on a domain-specific language, called *SEINE-L*, for describing the behaviour of selfish nodes, along with an annotation library to associate such specification with a system implementation for the simulator PeerSim. To design *SEINE-L*, we based on the domain analysis presented in the first part of this dissertation, especially, in Chapter 2. *SEINE-L* provides a unified semantics for defining *selfishness scenarios*, which allow describing capabilities, interests and behaviours of different types of nodes participating in the system. The *SEINE* framework provides a compiler and the run-time system supporting the automatic and systematic evaluation of different selfishness scenarios in the PeerSim simulation framework. Simulations return a set of statistics on the behaviour of the system when in the presence of the specified type of selfish nodes. We illustrated the generality of *SEINE-L* by describing fifteen selfishness scenarios extracted from the domain analysis. Then, we showed the accuracy and ease of use of *SEINE* in evaluating the impact of selfish behaviours in three use cases selected from the literature, namely, a gossip-based dissemination protocol, a live streaming protocol (BAR Gossip), and a file sharing system (BitTorrent).

8.2 POSSIBLE IMPROVEMENTS AND FUTURE RESEARCH DIRECTIONS

In the following, we discuss possible improvements of the contributions presented in this thesis, as well as future research direction that we consider worth pursuing.

8.2.1 *Integration of RACOON and SEINE*

The selfish behaviours considered by the *RACOON* framework to evaluate the selfish-resilience of a system are quite limited if compared with the possibilities emerged from the survey presented in Chapter 2. We addressed this limitation by developing *SEINE*, which provides a simpler, more general and comprehensive framework for selfishness analysis. The natural follow-up work is the integration of the *SEINE* functionalities into the *RACOON* framework, so as to combine the best analysis and design features from each tool, as summarised in Table 38.

Figure 52 illustrates how *RACOON* and *SEINE* might be composed into a single coherent framework. In particular, the figure shows which steps of *RACOON* could be replaced by *SEINE*. How to realise such integration in practice, especially the adaptation of the different inputs, is an engineering task that will require careful design and implementation.

Table 38: Evaluation of performance and capabilities of the integration between *RACOON* and *SEINE*.

	Analysis									Design
	General and cooperative systems criteria [a]				Selfishness criteria [b]					CE [c]
Approaches	**Usa**	**Rep**	**Ref**	**Sc**	**Ra**	**D**	**F**	**M**	**C**	
RACOON(++)	●●●●○	●●●●●	●●○○○	10^6	✓	✓	✓	x	x	✓
SEINE	●●●●●	●●●●●	●●○○○	10^6	✓	✓	✓	✓	✓	x
RACOON(++) + SEINE	●●●●●	●●●●●	●●○○○	10^6	✓	✓	✓	✓	✓	✓

[a] Usa = usability, Rep = reproducibility, Ref = refinement (inverse of abstraction), Sc = scalability.
[b] Ra = rationality, D = defection , F = free-ride, M = misreport, C = collusion.
[c] CE = cooperation enforcement

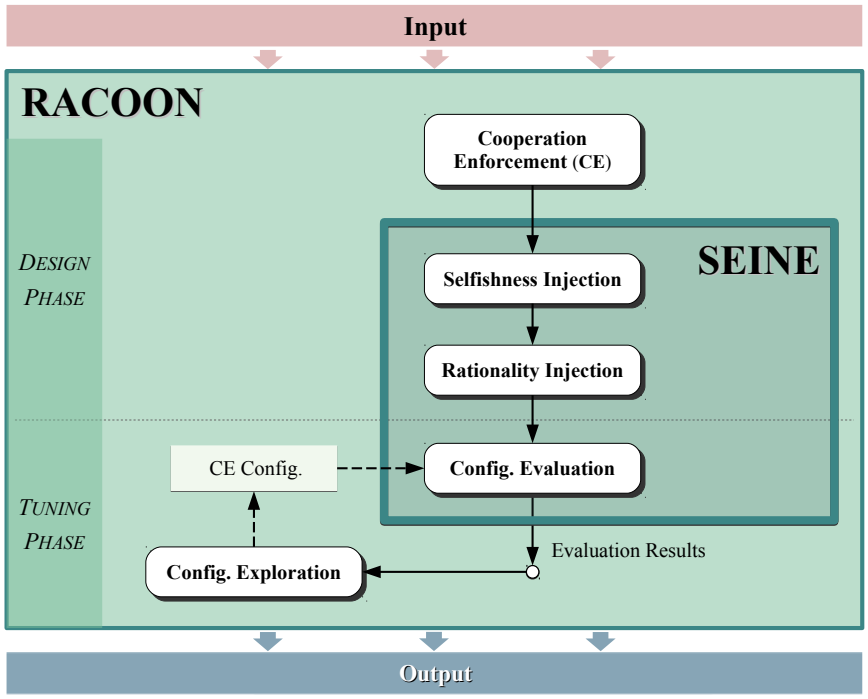


Figure 52: Conceptual integration of *SEINE* into the *RACOON* framework.

8.2.2 *Additional types of selfish deviations*

In the review of research work on selfishness, presented in Chapter 2, we identified four recurring types of deviations from the cooperative behaviour: defection, free-riding, misreport, and collusion. In this thesis, we showed that these types are sufficient to describe most of the deviations described in the reviewed literature. However, other types of deviations are possi-

ble. For example, as was noted in Chapter 2, we intentionally excluded from our consideration incentive-specific deviations such as whitewashing and Sybil attacks in reputations systems, or forgery and double spending in economy-based mechanisms. As another example of deviation type, we might consider deviations that violate the timing requirements of an intended behaviour, e.g., executing a certain task at a slower speed or delaying the sending of a message. A practical example is provided by Guerraoui et al. [72] in the context of a gossip-based live streaming system. In this system, each node proposes its available video chunks to randomly selected partners, who in turn request any chunks they need; the interaction ends when the proposing node delivers the requested chunks. The "timing deviation" described by the authors consists in increasing the period between two proposals, so as to propose old chunks that have a greater probability of not being requested because already received by the partners from previous interactions with other nodes. As a result, the selfish node is asked to deliver fewer chunks, which reduces its bandwidth consumption.

We plan to conduct a wider and deeper survey of the literature about selfishness, in order to formalise the deviation types outlined above as well as to search for additional types. The acquired knowledge would represent not only a conceptual contribution to enrich our understanding of the great variability of selfish behaviours, but it will also have practical importance in providing the theoretical background to widen the range and scope of the selfishness injection tools proposed in this thesis.

8.2.3 *Extending the RACOON framework*

The modularity of the *RACOON* framework provides many possibilities to extend its capabilities in different ways. For example, in Chapter 6, we presented *RACOON++*, which extended the original framework with a new rationality model for selfish nodes (i.e., evolutionary game theory) and with a model for parameterizing some aspects of the selfish behaviours supported. Figure 53 highlights (coloured in yellow) several possibilities of extension, which are discussed hereafter grouped under the heading of the respective step of the *RACOON* workflow.

COOPERATION ENFORCEMENT. The *RACOON* framework uses accountability and reputation mechanisms to make cooperation the most profitable behaviour for selfish nodes. Although these mechanisms are general and practical enough to be enforced in most cooperative systems, which were the requirements that led to their selection, a system designer might prefer or even need to rely on different countermeasures against selfishness. For example, *tit-for-tat* mechanisms have proved to be a less expensive yet workable solution in various cooperative systems, including BitTorrent [45]. As another example, consider cooperative systems such as ad-hoc networks in which nodes are likely to participate only for a short period. In these systems, reputation might not be perceived as a sufficient motivating incentive, because the benefits from cooperation could be long delayed after a node's cooperative action. A suitable alternative is offered by decentralised economy schemes, in which cooperative nodes get instant remuneration in units of currency (e.g., real money, Bitcoin [132] or Nuglets [33]). The Bitcoin in-

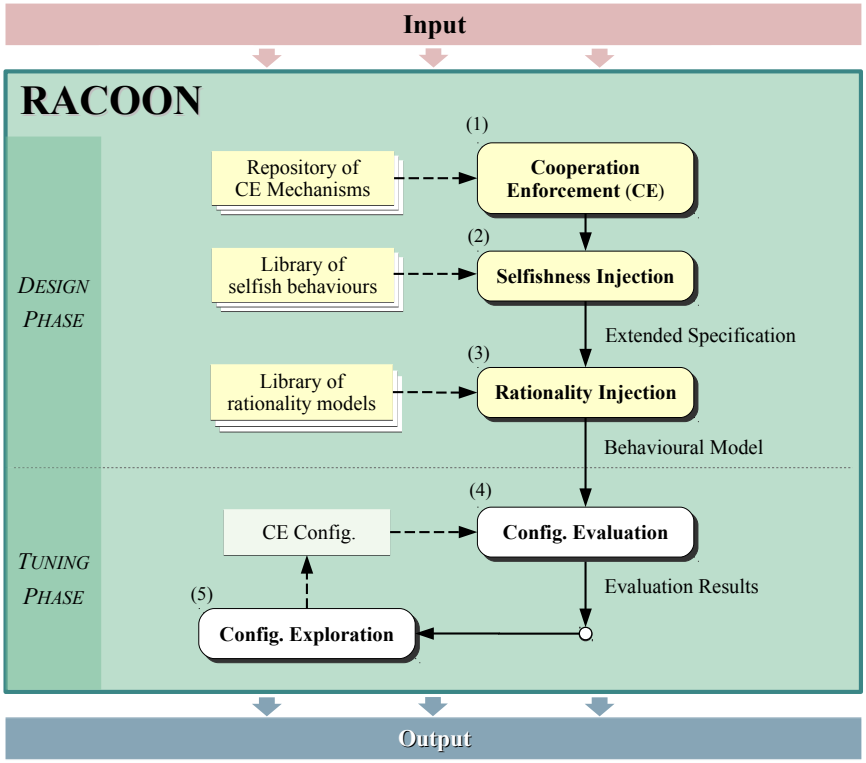


Figure 53: Possible extensions of the *RACOON* framework (coloured in yellow).

frastructure and blockchain in general are particularly suited for cooperative systems, because very robust and completely distributed (including the currency generation process) [157].

In future work, we intend to expand the set of cooperation enforcement mechanisms supported by *RACOON*, in such a way as to provide system designers with the flexibility to choose the most appropriate according to the specific characteristics of their system.

SELFISHNESS INJECTION. As discussed in Section 8.2.2, we plan to extend our set of selfish deviation types with new and more specific types. Then, building on these definitions, we can extend the selfishness injection algorithms included in *RACOON* to generate and integrate new types of deviations into the functional specification of the system, i.e., the Protocol Automaton (see Chapter 5). However, this might require some modifications of the original Protocol Automaton specification. For instance, to account for the "timing deviation" outlined in the previous section, we might extend the tuples that specify methods and messages of the Protocol Automaton with the *duration* and *delay* properties, as reported in Table 39. Then, we can use these properties to slow down the execution of a method as well as to postpone the delivery of a message.

RATIONALITY INJECTION. The rationality of a selfish node is described by a rationality model, which determines its decision-making in choosing and carrying out the (expected) most profitable behaviour. In this thesis, we have presented three rationality models:

PA Element	Original Tuple	Modified Tuple
Methods	$\langle \texttt{mId}, \texttt{messageId} \rangle$	$\langle \texttt{mId}, \texttt{messageId}, \texttt{duration} \rangle$ where *duration* indicates the method execution time
Messages	$\langle \texttt{gId}, \texttt{contentId} \rangle$	$\langle \texttt{gId}, \texttt{contentId}, \texttt{delay} \rangle$ where *delay* specifies the amount of time to wait before sending the message

Table 39: Possible modifications of some elements of the Protocol Automaton (PA) in order to account for the new type of "timing deviation" sketched in Section 8.2.2.

- In *RACOON*, we used non-cooperative sequential games from classical game theory, which are based on the assumption of perfect rationality [130]. Although this assumption might appear to be unrealistic in practice [120], it can serve as a worst-case scenario analysis, since it models the most strategic and powerful types of selfish nodes.
- In *RACOON++*, we used evolutionary games from evolutionary game theory [172], whereby selfish nodes evolve their behaviours according to the dynamics of the system, tending to implement the most remunerative strategies through learning and imitation. A drawback of this rationality model is that it increases the computational cost and execution time for evaluating its impact on the system performance.
- In *SEINE*, we developed the selfishness scenarios model along with the *SEINE-L* language to express it. A *SEINE-L* specification can describe under what conditions a certain type of nodes shall execute certain types of selfish deviations.

In future work, we plan to provide the users of the *RACOON* framework with the ability to choose what rationality model they want to consider, depending on its usability, timeliness, and reliability requirements. Furthermore, it could be interesting to investigate other models of rationality in future research, such as models based on activity logs or network traces [114, 124, 155, 175], or consider different types of games (e.g., inspection games [68], repeated games and Bayesian games [130]).

8.2.4 *Support for distributed testbeds*

In this thesis, we resorted to a combination of analytical and simulation approaches to analyse the impact of selfish behaviours on the performance of a given cooperative system. These approaches operate on a high level of abstraction, since both the system and the behaviour of nodes are fully modelled. However, in general, as the level of abstraction increases, the accuracy and confidence of the results go down.

A possible improvement of the analysis and design frameworks proposed in this thesis is to extend their support to other experimental approaches, namely, real experiments and emulation. Concretely, the frameworks should provide the necessary interfaces for running the

same experiment in a seamless manner on local machines through simulations or deployed in real (e.g., Grid'5000, PlanetLab) or emulated (e.g., ModelNet, EmuLab) networks. Among the various approaches that might be considered, we plan to take inspiration from the SPLAY evaluation environment proposed by Leonini et al. [108] and the work of Friedman et al. [64, 65].

Part V

APPENDIX

A

RACOON AND RACOON++: XML SCHEMA FOR THE INPUTS OF THE FRAMEWORK

Table 40 summarises the inputs of the *RACOON* and *RACOON++* frameworks that the system designer can provide as XML documents. Particularly, the inputs of each framework are included in the same XML document.

Framework	**Input**	*Ref. Section*
RACOON	Protocol Automaton PA	Section 5.3.1
	Design Objectives	Section 5.4.1
RACOON++	Protocol Automaton PA^{++}	Section 6.3.1
	Selfishness Model	Section 6.3.1
	Design Objectives	Section 6.4.1

Table 40: The inputs of the *RACOON* and *RACOON++* frameworks.

In this appendix, we provide the XML Schema definitions for the XML documents to specify the inputs of the *RACOON* and *RACOON++* frameworks.

A.1 SCHEMA FOR THE XML INPUTS OF RACOON

General structure.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
<xs:element name="racoon">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="protocol_automaton"/>
      <xs:element ref="design_objectives"/>
    </xs:sequence>
    <xs:attribute name="name" use="required" type="xs:NCName"/>
  </xs:complexType>
</xs:element>
```

Protocol Automaton.

```
<xs:element name="protocol_automaton">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="roles"/>
      <xs:element ref="states"/>
      <xs:element ref="transitions"/>
      <xs:element ref="methods"/>
      <xs:element ref="messages"/>
      <xs:element ref="contents"/>
      <xs:element ref="constraints"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="roles">
```

```
  <xs:complexType>
    <xs:sequence>
      <xs:element minOccurs="1" maxOccurs="unbounded" ref="role"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="role">
  <xs:complexType>
    <xs:attribute name="cardinality" use="required"/>
    <xs:attribute name="id" use="required" type="xs:NCName"/>
    <xs:attribute name="isSelfish" use="required" type="xs:boolean"/>
  </xs:complexType>
</xs:element>
<xs:element name="states">
  <xs:complexType>
    <xs:sequence>
      <xs:element minOccurs="1" maxOccurs="unbounded" ref="state"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="state">
  <xs:complexType>
    <xs:attribute name="id" use="required" type="xs:NCName"/>
    <xs:attribute name="roleId" use="required" type="xs:NCName"/>
    <xs:attribute name="type" use="required" type="stateType"/>
  </xs:complexType>
</xs:element>
<xs:simpleType name="stateType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="initial"/>
    <xs:enumeration value="final"/>
    <xs:enumeration value="intermediate"/>
  </xs:restriction>
</xs:simpleType>
<xs:element name="transitions">
  <xs:complexType>
    <xs:sequence>
      <xs:element maxOccurs="unbounded" ref="transition"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="transition">
  <xs:complexType>
    <xs:attribute name="id" use="required" type="xs:NCName"/>
    <xs:attribute name="methodId" use="required" type="xs:NCName"/>
    <xs:attribute name="state1Id" use="required" type="xs:NCName"/>
    <xs:attribute name="state2Id" use="required" type="xs:NCName"/>
  </xs:complexType>
</xs:element>
<xs:element name="methods">
  <xs:complexType>
    <xs:sequence>
      <xs:element maxOccurs="unbounded" ref="method"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="method">
  <xs:complexType>
    <xs:attribute name="id" use="required" type="xs:NCName"/>
    <xs:attribute name="messageId" type="xs:NCName"/>
  </xs:complexType>
</xs:element>
<xs:element name="messages">
  <xs:complexType>
    <xs:sequence>
      <xs:element maxOccurs="unbounded" ref="message"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="message">
  <xs:complexType>
    <xs:attribute name="contentId" use="required" type="xs:NCName"/>
    <xs:attribute name="id" use="required" type="xs:NCName"/>
```

```
  </xs:complexType>
</xs:element>
<xs:element name="contents">
  <xs:complexType>
    <xs:sequence>
      <xs:element maxOccurs="unbounded" ref="content"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="content">
  <xs:complexType>
    <xs:attribute name="id" use="required" type="xs:NCName"/>
    <xs:attribute name="length" use="required"/>
    <xs:attribute name="size" use="required" type="xs:integer"/>
    <xs:attribute name="type" use="required" type="xs:NCName"/>
  </xs:complexType>
</xs:element>
<xs:element name="constraints">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="constraint"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="constraint">
  <xs:complexType>
    <xs:attribute name="content1Id" use="required" type="xs:NCName"/>
    <xs:attribute name="content2Id" use="required" type="xs:NCName"/>
    <xs:attribute name="id" use="required" type="xs:NCName"/>
    <xs:attribute name="type" use="required" type="constraintType"/>
  </xs:complexType>
</xs:element>
<xs:simpleType name="constraintType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="="/>
    <xs:enumeration value="&lt;"/>
    <xs:enumeration value="&gt;"/>
    <xs:enumeration value="subset"/>
    <xs:enumeration value="strict_subset"/>
    <xs:enumeration value="equal"/>
  </xs:restriction>
</xs:simpleType>
```

Design Objectives.

```
<xs:element name="design_objectives">
  <xs:complexType>
    <xs:sequence>
      <xs:element maxOccurs="unbounded" ref="objective"/>
      <xs:element maxOccurs="unbounded" ref="custom_objective"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="objective">
  <xs:complexType>
    <xs:attribute name="id" use="required" type="objId"/>
    <xs:attribute name="predicate" use="required" type="objPredicate"/>
    <xs:attribute name="value" use="required" type="xs:double"/>
  </xs:complexType>
</xs:element>
<xs:element name="custom_objective">
  <xs:complexType>
    <xs:attribute name="id" use="required" type="xs:NCName"/>
    <xs:attribute name="predicate" use="required" type="objPredicate"/>
    <xs:attribute name="value" use="required" type="xs:double"/>
    <xs:attribute name="minValue" use="required" type="xs:double" default="0.0" />
    <xs:attribute name="maxValue" use="required" type="xs:double" default="1.0" />
  </xs:complexType>
</xs:element>
<xs:simpleType name="objId">
  <xs:restriction base="xs:string">
    <xs:enumeration value="deviation_rate"/>
    <xs:enumeration value="cooperation_level"/>
    <xs:enumeration value="audit_precision"/>
```

```
    <xs:enumeration value="wrongful_eviction_rate"/>
    <xs:enumeration value="CEM_message_overhead"/>
    <xs:enumeration value="CEM_bandwidth_overhead"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="objPredicate">
  <xs:restriction base="xs:string">
    <xs:enumeration value="at_least"/>
    <xs:enumeration value="at_most"/>
  </xs:restriction>
</xs:simpleType>
</xs:schema>
```

A.2 SCHEMA FOR THE XML INPUTS OF RACOON++

General structure.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
<xs:element name="racoon">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="protocol_automaton"/>
      <xs:element ref="selfishness_model"/>
      <xs:element ref="design_objectives"/>
    </xs:sequence>
    <xs:attribute name="name" use="required" type="xs:NCName"/>
  </xs:complexType>
</xs:element>
```

Protocol Automaton.

```
<xs:element name="protocol_automaton">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="roles"/>
      <xs:element ref="states"/>
      <xs:element ref="transitions"/>
      <xs:element ref="methods"/>
      <xs:element ref="messages"/>
      <xs:element ref="contents"/>
      <xs:element ref="constraints"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="roles">
  <xs:complexType>
    <xs:sequence>
      <xs:element minOccurs="1" maxOccurs="unbounded" ref="role"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="role">
  <xs:complexType>
    <xs:attribute name="cardinality" use="required"/>
    <xs:attribute name="id" use="required" type="xs:NCName"/>
    <xs:attribute name="type" use="required" type="roleType"/>
  </xs:complexType>
</xs:element>
<xs:simpleType name="roleType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="provider"/>
    <xs:enumeration value="requester"/>
    <xs:enumeration value="other"/>
  </xs:restriction>
</xs:simpleType>
<xs:element name="states">
  <xs:complexType>
    <xs:sequence>
      <xs:element minOccurs="1" maxOccurs="unbounded" ref="state"/>
```

```
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="state">
  <xs:complexType>
    <xs:attribute name="id" use="required" type="xs:NCName"/>
    <xs:attribute name="roleId" use="required" type="xs:NCName"/>
    <xs:attribute name="type" use="required" type="stateType"/>
  </xs:complexType>
</xs:element>
<xs:simpleType name="stateType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="initial"/>
    <xs:enumeration value="final"/>
    <xs:enumeration value="intermediate"/>
  </xs:restriction>
</xs:simpleType>
<xs:element name="transitions">
  <xs:complexType>
    <xs:sequence>
      <xs:element maxOccurs="unbounded" ref="transition"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="transition">
  <xs:complexType>
    <xs:attribute name="id" use="required" type="xs:NCName"/>
    <xs:attribute name="methodId" use="required" type="xs:NCName"/>
    <xs:attribute name="state1Id" use="required" type="xs:NCName"/>
    <xs:attribute name="state2Id" use="required" type="xs:NCName"/>
  </xs:complexType>
</xs:element>
<xs:element name="methods">
  <xs:complexType>
    <xs:sequence>
      <xs:element maxOccurs="unbounded" ref="method"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="method">
  <xs:complexType>
    <xs:attribute name="id" use="required" type="xs:NCName"/>
    <xs:attribute name="messageId" type="xs:NCName"/>
  </xs:complexType>
</xs:element>
<xs:element name="messages">
  <xs:complexType>
    <xs:sequence>
      <xs:element maxOccurs="unbounded" ref="message"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="message">
  <xs:complexType>
    <xs:attribute name="contentId" use="required" type="xs:NCName"/>
    <xs:attribute name="id" use="required" type="xs:NCName"/>
    <xs:attribute name="receiverId" use="required" type="xs:NCName"/>
    <xs:attribute name="senderId" use="required"/>
  </xs:complexType>
</xs:element>
<xs:element name="contents">
  <xs:complexType>
    <xs:sequence>
      <xs:element maxOccurs="unbounded" ref="content"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="content">
  <xs:complexType>
    <xs:attribute name="id" use="required" type="xs:NCName"/>
    <xs:attribute name="length" use="required"/>
    <xs:attribute name="type" use="required" type="xs:NCName"/>
  </xs:complexType>
```

```
</xs:element>
<xs:element name="constraints">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="constraint"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="constraint">
  <xs:complexType>
    <xs:attribute name="content1Id" use="required" type="xs:NCName"/>
    <xs:attribute name="content2Id" use="required" type="xs:NCName"/>
    <xs:attribute name="id" use="required" type="xs:NCName"/>
    <xs:attribute name="type" use="required" type="constraintType"/>
  </xs:complexType>
</xs:element>
<xs:simpleType name="constraintType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="="/>
    <xs:enumeration value="&lt;"/>
    <xs:enumeration value="&gt;"/>
    <xs:enumeration value="subset"/>
    <xs:enumeration value="strict_subset"/>
    <xs:enumeration value="equal"/>
  </xs:restriction>
</xs:simpleType>
```

Selfishness Model.

```
<xs:element name="selfishness_model">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="valuations"/>
      <xs:element ref="deviations"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="valuations">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="valuation"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="valuation">
  <xs:complexType>
    <xs:attribute name="benefit" use="required" type="xs:integer"/>
    <xs:attribute name="cost" use="required" type="xs:integer"/>
    <xs:attribute name="id" use="required" type="xs:NCName"/>
    <xs:attribute name="roleId" use="required" type="xs:NCName"/>
    <xs:attribute name="scope" use="required" type="valuationScope"/>
    <xs:attribute name="scopeId" use="required" type="xs:NCName"/>
  </xs:complexType>
</xs:element>
<xs:simpleType name="valuationScope">
  <xs:restriction base="xs:string">
    <xs:enumeration value="transition"/>
    <xs:enumeration value="message"/>
  </xs:restriction>
</xs:simpleType>
<xs:element name="deviations">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="deviation"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="deviation">
  <xs:complexType>
    <xs:attribute name="degree" use="required" type="xs:decimal"/>
    <xs:attribute name="id" use="required" type="xs:NCName"/>
    <xs:attribute name="transitionId" use="required" type="xs:NCName"/>
    <xs:attribute name="type" use="required" type="deviationType"/>
  </xs:complexType>
```

```
</xs:element>
<xs:simpleType name="deviationType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="timeout"/>
    <xs:enumeration value="subset"/>
    <xs:enumeration value="multicast"/>
  </xs:restriction>
</xs:simpleType>
```

Design Objectives.

```
<xs:element name="design_objectives">
  <xs:complexType>
    <xs:sequence>
      <xs:element maxOccurs="unbounded" ref="objective"/>
      <xs:element maxOccurs="unbounded" ref="custom_objective"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="objective">
  <xs:complexType>
    <xs:attribute name="id" use="required" type="objId"/>
    <xs:attribute name="predicate" use="required" type="objPredicate"/>
    <xs:attribute name="value" use="required" type="xs:double"/>
  </xs:complexType>
</xs:element>
<xs:element name="custom_objective">
  <xs:complexType>
    <xs:attribute name="id" use="required" type="xs:NCName"/>
    <xs:attribute name="predicate" use="required" type="objPredicate"/>
    <xs:attribute name="value" use="required" type="xs:double"/>
    <xs:attribute name="minValue" use="required" type="xs:double" default="0.0"/>
    <xs:attribute name="maxValue" use="required" type="xs:double" default="1.0"/>
  </xs:complexType>
</xs:element>
<xs:simpleType name="objId">
  <xs:restriction base="xs:string">
    <xs:enumeration value="cooperation_level"/>
    <xs:enumeration value="cooperation_persistence"/>
    <xs:enumeration value="cooperation_attractiveness"/>
    <xs:enumeration value="audit_precision"/>
    <xs:enumeration value="audit_recall"/>
    <xs:enumeration value="CEM_cost_overhead"/>
    <xs:enumeration value="CEM_message_overhead"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="objPredicate">
  <xs:restriction base="xs:string">
    <xs:enumeration value="at_least"/>
    <xs:enumeration value="at_most"/>
  </xs:restriction>
</xs:simpleType>
</xs:schema>
```

B

RACOON AND RACOON++ EVALUATION: USE CASES SPECIFICATION

In this appendix we provide the specification of the cooperative systems used as a use case for evaluating the *RACOON* and *RACOON++* frameworks.

B.1 RACOON USE CASES

Gossip-based P2P Live Streaming (XML Specification).

```
<racoon name="P2PLiveStreaming">
<protocol_automaton>
  <roles>
    <role id="rp" cardinality="1" isSelfish="true" />
    <role id="rC" cardinality="7" isSelfish="false" />
  </roles>
  <states>
    <state id="s0" roleId="rp" type="initial" />
    <state id="s1" roleId="rC" type="intermediate" />
    <state id="s2" roleId="rp" type="intermediate" />
    <state id="s3" roleId="rC" type="final" />
  </states>
  <transitions>
    <transition id="t0" state1Id="s0" state2Id="s1" methodId="propose" />
    <transition id="t1" state1Id="s1" state2Id="s2" methodId="request" />
    <transition id="t2" state1Id="s2" state2Id="s3" methodId="serve" />
  </transitions>
  <methods>
    <method id="propose" messageId="g0" />
    <method id="request" messageId="g1" />
    <method id="serve" messageId="g2" />
  </methods>
  <messages>
    <message id="g0" contentId="c0" />
    <message id="g1" contentId="c1" />
    <message id="g2" contentId="c2" />
  </messages>
  <contents>
    <content id="c0" type="integer" size="4" length=">=0" />
    <content id="c1" type="integer" size="4" length=">=0" />
    <content id="c2" type="binary" size="6000" length=">=0" />
  </contents>
  <constraints>
    <constraint id="k0" content1Id="c1" type="subset" content2Id="c0" />
    <constraint id="k1" content1Id="c2" type="equal" content2Id="c1" />
  </constraints>
</protocol_automaton>
<design_objectives>
  <objective id="deviation_rate" predicate="at_most" value="0.1" />
  <objective id="CEM_bandwidth_overhead" predicate="at_most" value="0.4"/>
  <objective id="wrongful_eviction_rate" predicate="at_most" value="0.1"/>
</design_objectives>
</racoon>
```

Anonymous communication protocol (XML Specification).

```
<racoon name="AnonymousCommunication">
<protocol_automaton>
  <roles>
    <role id="rP" cardinality="10" isSelfish="false" />
    <role id="rr" cardinality="1" isSelfish="true" />
    <role id="rN" cardinality="10" isSelfish="false" />
  </roles>
```

```
  <states>
    <state id="s0" roleId="rP" type="initial" />
    <state id="s1" roleId="rr" type="intermediate" />
    <state id="s2" roleId="rr" type="intermediate" />
    <state id="s3" roleId="rN" type="final" />
  </states>
  <transitions>
    <transition id="t0" state1Id="s0" state2Id="s1" methodId="sendToRelay" />
    <transition id="t1" state1Id="s1" state2Id="s2" methodId="decrypt" />
    <transition id="t2" state1Id="s2" state2Id="s3" methodId="relay" />
  </transitions>
  <methods>
    <method id="sendToRelay" messageId="g0" />
    <method id="decrypt" messageId="" />
    <method id="relay" messageId="g1" />
  </methods>
  <messages>
    <message id="g0" contentId="c0" />
    <message id="g1" contentId="c1" />
  </messages>
  <contents>
    <content id="c0" type="binary" size="10000" length=">=0" />
    <content id="c1" type="binary" size="10000" length=">=0" />
  </contents>
  <constraints />
</protocol_automaton>
<design_objectives>
  <objective id="deviation_rate" predicate="at_most" value="0.1" />
  <objective id="CEM_bandwidth_overhead" predicate="at_most" value="0.4"/>
  <objective id="wrongful_eviction_rate" predicate="at_most" value="0.1"/>
</design_objectives>
</racoon>
```

B.2 RACOON++ USE CASES

B.2.1 *Experiments: Design and development effort*

Live Streaming (XML Specification).

```
<racoon name="P2PLiveStreaming">
<protocol_automaton>
  <roles>
    <role id="rp" cardinality="1" type="provider" />
    <role id="rC" cardinality="10" type="requester" />
  </roles>
  <states>
    <state id="s0" roleId="rp" type="initial" />
    <state id="s1" roleId="rC" type="intermediate" />
    <state id="s2" roleId="rp" type="intermediate" />
    <state id="s3" roleId="rC" type="final" />
  </states>
  <transitions>
    <transition id="t0" state1Id="s0" state2Id="s1" methodId="propose" />
    <transition id="t1" state1Id="s1" state2Id="s2" methodId="request" />
    <transition id="t2" state1Id="s2" state2Id="s3" methodId="serve" />
  </transitions>
  <methods>
    <method id="propose" messageId="g0" />
    <method id="request" messageId="g1" />
    <method id="serve" messageId="g2" />
  </methods>
  <messages>
    <message id="g0" contentId="c0" senderId="rp" receiverId="rC" />
    <message id="g1" contentId="c1" senderId="rC" receiverId="rp" />
    <message id="g2" contentId="c2" senderId="rp" receiverId="rC" />
  </messages>
  <contents>
    <content id="c0" type="integer" length=">=0" />
    <content id="c1" type="integer" length=">=0" />
    <content id="c2" type="binary" length=">=0" />
  </contents>
```

```
  <constraints>
    <constraint id="k0" content1Id="c1" type="subset" content2Id="c0" />
    <constraint id="k1" content1Id="c2" type="equal" content2Id="c1" />
  </constraints>
</protocol_automaton>
<selfishness_model>
  <valuations>
    <valuation id="v0" scope="transition" scopeId="t0" roleId="rp" benefit="200" cost="0" />
    <valuation id="v1" scope="transition" scopeId="t0" roleId="rC" benefit="1000" cost="0" />
    <valuation id="v2" scope="message" scopeId="g0" roleId="rp" benefit="0" cost="4" />
    <valuation id="v3" scope="message" scopeId="g1" roleId="rC" benefit="20" cost="4" />
    <valuation id="v4" scope="message" scopeId="g2" roleId="rp" benefit="0" cost="10000" />
    <valuation id="v5" scope="message" scopeId="g2" roleId="rC" benefit="10000" cost="0" />
  </valuations>
  <deviations>
    <deviation id="d0" transitionId="t0" type="multicast" degree="0.45" />
    <deviation id="d1" transitionId="t0" type="subset" degree="0.5" />
  </deviations>
</selfishness_model>
<design_objectives>
  <custom_objective id="chunk_loss" predicate="at_most" value="0.03" min="0.0" max="1.0" />
</design_objectives>
</racoon>
```

Load Balancing (XML Specification).

```
<racoon name="LoadBalancing">
<protocol_automaton>
  <roles>
    <role id="r0" cardinality="1" type="requester" />
    <role id="R1" cardinality="20" type="provider" />
  </roles>
  <states>
    <state id="s0" roleId="r0" type="initial" />
    <state id="s1" roleId="R1" type="intermediate" />
    <state id="s2" roleId="r0" type="intermediate" />
    <state id="s3" roleId="r0" type="intermediate" />
    <state id="s4" roleId="R1" type="final" />
  </states>
  <transitions>
    <transition id="t0" state1Id="s0" state2Id="s1" methodId="selectPartnerReq" />
    <transition id="t1" state1Id="s1" state2Id="s2" methodId="selectPartnerResp" />
    <transition id="negotiation" state1Id="s2" state2Id="s3" methodId="" />
    <transition id="transfer" state1Id="s3" state2Id="s4" methodId="" />
  </transitions>
  <methods>
    <method id="selectPartnerReq" messageId="g0" />
    <method id="selectPartnerResp" messageId="g1" />
  </methods>
  <messages>
    <message id="g0" contentId="c0" senderId="r0" receiverId="R1" />
    <message id="g1" contentId="c1" senderId="R1" receiverId="r0" />
  </messages>
  <contents>
    <content id="c0" type="integer" length="1" />
    <content id="c1" type="integer" length="1" />
  </contents>
  <constraints />
</protocol_automaton>
<selfishness_model>
  <valuations>
    <valuation id="v0" scope="transition" scopeId="negotiation" roleId="r0" benefit="20" cost="40" />
    <valuation id="v1" scope="message" scopeId="g0" roleId="r0" benefit="50" cost="40" />
    <valuation id="v1" scope="message" scopeId="g1" roleId="R1" benefit="50" cost="40" />
  </valuations>
  <deviations>
    <deviation id="d0" transitionId="t0" type="multicast" degree="0.5" />
    <deviation id="d1" transitionId="t1" type="timeout" degree="1" />
    <deviation id="d2" transitionId="transfer" type="timeout" degree="1" />
  </deviations>
</selfishness_model>
<design_objectives>
  <custom_objective id="cov" predicate="at_most" value="0.1" min="0.0" max="1.0" />
</design_objectives>
</racoon>
```

Anonymous communication (XML Specification).

```
<racoon name="AnonymousCommunication">
<protocol_automaton>
  <roles>
    <role id="rP" cardinality="10" type="requester" />
    <role id="rr" cardinality="1" type="provider" />
    <role id="rN" cardinality="10" type="requester" />
  </roles>
  <states>
    <state id="s0" roleId="rP" type="initial" />
    <state id="s1" roleId="rr" type="intermediate" />
    <state id="s2" roleId="rr" type="final" />
    <state id="s3" roleId="rN" type="final" />
  </states>
  <transitions>
    <transition id="t0" state1Id="s0" state2Id="s1" methodId="sendToRelay" />
    <transition id="t1" state1Id="s1" state2Id="s2" methodId="decrypt" />
    <transition id="t2" state1Id="s2" state2Id="s3" methodId="relay" />
  </transitions>
  <methods>
    <method id="sendToRelay" messageId="g0" />
    <method id="decrypt" messageId="" />
    <method id="relay" messageId="g1" />
  </methods>
  <messages>
    <message id="g0" contentId="c0" senderId="rP" receiverId="rr" />
    <message id="g1" contentId="c1" senderId="rr" receiverId="rN" />
  </messages>
  <contents>
    <content id="c0" type="binary" length="1" />
    <content id="c1" type="binary" length="1" />
  </contents>
  <constraints />
</protocol_automaton>
<selfishness_model>
  <valuations>
    <valuation id="v0" scope="transition" scopeId="t1" roleId="rr" benefit="256" cost="0" />
    <valuation id="v1" scope="transition" scopeId="t1" roleId="rN" benefit="4096" cost="0" />
    <valuation id="v2" scope="message" scopeId="g0" roleId="rN" benefit="256" cost="512" />
    <valuation id="v3" scope="message" scopeId="g1" roleId="rr" benefit="256" cost="512" />
  </valuations>
  <deviations>
    <deviation id="d0" transitionId="t2" type="multicast" degree="0.4" />
    <deviation id="d1" transitionId="t2" type="timeout" degree="1" />
  </deviations>
</selfishness_model>
<design_objectives>
  <custom_objective id="onion_loss" predicate="at_most" value="0.1" min="0.0" max="1.0" />
</design_objectives>
</racoon>
```

B.2.2 *Experiments: RACOON++ effectiveness*

Deviations defined in the Selfishness Models.

Live Streaming

Id Scenario	*Deviations* [a]
LS #2	$\{\langle d_0, t_0, \text{timeout}, 1\rangle, \langle d_1, t_0, \text{multicast}, 0.45\rangle, \langle d_2, t_2, \text{subset}, 0.7\rangle\}$
LS #5	$\{\langle d_0, t_0, \text{timeout}, 1\rangle, \langle d_1, t_2, \text{timeout}, 0.1\rangle, \langle d_2, t_2, \text{subset}, 0.2\rangle\}$
LS #19	$\{\langle d_0, t_0, \text{multicast}, 0.45\rangle, \langle d_1, t_2, \text{subset}, 0.5\rangle\}$

[a] With reference to Figure 36, the PA transitions are triggered by the following methods: t_0 by `propose`, t_1 by `request`, and t_2 by `serve`.

Load Balancing

Id Scenario	*Deviations* [a]
LB #4	$\{\langle d_0, t_0, \texttt{timeout}, 1\rangle, \langle d_1, t_0, \texttt{multicast}, 0.5\rangle, \langle d_2, t_1, \texttt{timeout}, 1\rangle\}$
LB #5	$\{\langle d_0, t_0, \texttt{timeout}, 1\rangle, \langle d_1, t_0, \texttt{multicast}, 0.5\rangle, \langle d_2, t_1, \texttt{timeout}, 1\rangle, \langle d_3, \texttt{transfer}, \texttt{timeout}, 1\rangle\}$
LB #23	$\{\langle d_0, t_0, \texttt{timeout}, 1\rangle, \langle d_1, t_0, \texttt{multicast}, 0.5\rangle, \langle d_2, \texttt{transfer}, \texttt{timeout}, 1\rangle\}$

[a] With reference to Figure 37, the PA transitions are triggered by the following methods: t_0 by `selectPartnerReq` and t_1 by `selectPartnerResp`. `negotiation` and `transfer` are two abstract transitions.

Anonymous Communication

Id Scenario	*Deviations* [a]
AC #3	$\{\langle d_0, t_2, \texttt{timeout}, 1\rangle, \langle d_1, t_2, \texttt{multicast}, 0.4\rangle\}$
AC #18	$\{\langle d_0, t_2, \texttt{multicast}, 0.3\rangle\}$
AC #25	$\{\langle d_0, t_2, \texttt{multicast}, 0.6\rangle\}$

[a] With reference to Figure 38, the PA transitions are triggered by the following methods: t_0 by `sendToRelay`, t_1 by `decrypt`, and t_2 by `relay`.

CEM Configurations.

	Live Streaming			**Load Balancing**			**Anonymous Comm.**		
Parameter	LS #2	LS #5	LS #19	LB #4	LB #5	LB #23	AC #3	AC #18	AC #25
Witness set size (nodes)	2	2	3	1	2	2	3	3	3
Audit period (cycles)	18	16	20	20	14	14	20	20	20
Audit probability	0.75	0.65	0.6	0.6	0.7	0.65	0.6	0.6	0.6
Degree of punishment	4.0	4.0	4.0	4.0	5.0	4.0	4.0	4.0	4.0
Degree of reward	0.15	0.4	0.4	0.4	0.15	0.15	0.4	0.4	4.0

C

SEINE: GRAMMAR OF THE SEINE-L LANGUAGE

In this appendix we provide the context-free grammar of the *SEINE-L* domain-specific language introduced in Section 7.3 in Chapter 7.

```
seine : 'seine' '.' ID '{' declarations+ '}' ;
declarations : resource | indicator | node | selfishness | observers;

// Resource and indicator declarations
resource : 'resource' '.' ID (init_resource)? ;
init_resource : 'random' '(' DOUBLE ',' DOUBLE ')'
              | 'linear' '(' DOUBLE ',' DOUBLE ')'
              | 'uniform' '(' DOUBLE ')'
              ;
indicator : 'indicator' '.' ID ;

// Node type declarations
node : 'node' '.' ID ('{' node_body* '}')? # nodeDecl
     | 'node.exclude' node_exclude_list # nodeExclude
     ;
node_body : node_fraction | node_selfish_fraction | node_capabilities ;
node_fraction : 'fraction' DOUBLE ;
node_selfish_fraction : 'selfish' DOUBLE ;
node_capabilities : 'capability' capability_item+ ;
capability_item : ID '(' DOUBLE ',' DOUBLE ')' | ID '(' DOUBLE ')';
node_exclude_list : DOUBLE* ;

// Selfishness model declarations
selfishness : 'selfishness' '.' ID '{' selfishness_body '}' ;
selfishness_body : 'actor' actor_list behaviour+
                 | behaviour+ 'actor' actor_list
                 ;
actor_list : actor_item+ ;
actor_item : ID ('(' DOUBLE ')')? ;

behaviour : 'behaviour' '.' ID '{' behaviour_body '}' ;
behaviour_body : ('activation' activation_decl)? deviation_decl+
               | deviation_decl+ ('activation' activation_decl)?
               ;
activation_decl : ID BIN_OP DOUBLE ;
```

```
deviation_decl : DEV_TYPE ('{' deviation_param+ '}')?
               | DEV_TYPE '.' deviation_param
               ;
deviation_param : 'probability' DOUBLE
                | 'degree' DOUBLE
                | 'on' methods_decl+
                ;
methods_decl : ID | '!' ID ;

// Observers declarations
observers : 'observers' '{' observers_body '}'
          | 'observers' '.' observers_names
          ;
observers_body  : ('period' observers_period)? observers_names ;
observers_period : DOUBLE ;
observers_names : 'name' observers_names_item+ ;
observers_names_item : ID ('.' ID)+ ;

// LEXER
BIN_OP :  '<' | '>' | '<=' | '>=' | '=' ;
DEV_TYPE : ('free-riding' | 'freeriding')
         | 'defection'
         | 'misreport'
         | 'collusion'
         | 'other'
         ;
DOUBLE : ('0'..'9')+ ('.' ('0'..'9')+)? ;
ID : ('a'..'z'|'A'..'Z'|'0'..'9'|'_'|'-')+ ;
```

D

SEINE: EVALUATION OF THE GENERALITY AND EXPRESSIVENESS OF SEINE-L

In the following, we report the *SEINE-L* description of the Selfishness Scenarios specified to evaluate the generality and expressiveness of the *SEINE-L* language in Section 7.5 in Chapter 7.

P2P Backup, Gramaglia et al. [71]

```
seine.P2PBackup  {
  resource.spaceUtilisation uniform(75000) # MB
  resource.backupSize uniform (10000) # MB
  resource.onlinePeriodProb
  node.peer1 {
    fraction 0.25
    selfish 0.5
    capability onlinePeriodProb(1.0)
  }
  node.peer2 {
    fraction 0.25
    selfish 0.5
    capability onlinePeriodProb(0.75)
  }
  node.peer3 {
    fraction 0.25
    selfish 0.5
    capability onlinePeriodProb(0.5)
  }
  node.peer4 {
    fraction 0.25
    selfish 0.5
    capability onlinePeriodProb(0.25)
  }
  selfishness.sm {
    actor peer1 peer2 peer3 peer4
    behaviour.bhv {
      defection {
        probability 0.5
        on storePartnerData
      }
    }
  }
  observers.name p2pbackup.P2PBackupObserver
}
```

BAR Gossip, Li et al. [111]

```
seine.BarGossip {
  resource.bandwidthCapacity
  indicator.uploadDownloadRatio
  node.peer {
    selfish 0.6
    capability bandwidthCapacity(0,150000)
  }
  node.exclude 0 # source
  selfishness.proactive {
    actor peer(0.3)
    behaviour.proactiveDecline {
      activation uploadDownladRatio > 1
      defection.on startOptimisticPushProtocol
    }
    behaviour.proactiveJunk {
      activation uploadDownladRatio < 1
      misreport { degree 0.5 on sendPUSHmessage }
      defection { probability 0.5 on sendBRIEFmessage }
    }
  }
  selfishness.passive {
    actor peer(0.5)
    behaviour.passiveDecline {
      activation uploadDownladRatio > 1
      defection.on sendPUSH_RESPmessage
    }
    behaviour.passiveJunk {
      activation uploadDownladRatio < 1
      misreport { degree 0.5 on sendPUSH_RESPmessage }
    }
  }
  selfishness.colluding {
    actor peer(0.1)
    behaviour.collusive { collusion }
  }
  observers.name bg.BarGossipObserver
}
```

Dandelion, Sirivianos et al. [165]

```
seine.Dandelion {
  resource.bandwidth random(0,1000) # Kbps
  indicator.downloadedFraction
  node.Leecher { fraction 0.995 selfish 0.7 }
  node.Seeder { fraction 0.005 }
  node.exclude 0 # source
  selfishness.selfish_seeding {
    actor Leecher(0.7)
    behaviour.bhvs {
      activation downloadedFraction = 100
      defection { probability 0.5 on seedFile }
    }
  }
  selfishness.misreporting {
    actor Leecher(0.3)
    behaviour.bhvm {
      misreport {
        probability 0.5
        on notifyNeighboursToTracker
      }
    }
  }
  observers.name dandelion.DandelionObserver
}
```

Grid computing, Kwok et al. [104]

```
seine.GridComputing  {
  resource.reputationIndex uniform(0.5)
  indicator.servicePromise # 1 if delivered, 0 otherwise
  node.GridSiteManager { fraction 0.085 selfish 0.2 }
  node.GridMachine { fraction 0.815 selfish 0 }
  node.exclude 0 # the global task dispatcher
  selfishness.smGSM_promise {
    actor GridSiteManager(0.5)
    behaviour.bhvGSM_promise {
      activation servingMemberPromise = 1
      freeriding.on dispatchRemoteTask
      misreport.on notifyComputationalCapacity
    }
  }
  selfishness.smGSM_RI {
    actor GridSiteManager(0.5)
    behaviour.bhvGSM_RI  {
      activation RI > 5
      freeriding.on  dispatchRemoteTask
      misreport.on notifyComputationalCapacity
    }
  }
  observers.name gc.GridComputingObserver
}
```

GiveToGet, Mei and Stefa [124]

```
seine.GiveToGet {
  resource.bandwidth random(0,600000)
  resource.storage random(0,100)
  node.DeviceResourceful { fraction 0.25 selfish 0.2 }
  node.DeviceResourceless {
    fraction 0.75
    selfish 0.7
    capability bandwidth(0,200000)
  }
  selfishness.smDR {
    actor DeviceResourceless(0.7)
    behaviour.bhv_store {
      activation storage > 60
      defection { probability 0.8 on storeMessages }
    }
    behaviour.bhv_forward {
      activation bandwidth > 150000
      freeriding { degree 0.5 on forwardMessages }
    }
  }
  selfishness.smCollusion {
    actor DeviceResourceful DeviceResourceless(0.3)
    behaviour.collusive {
      collusion.on storeProofOfRelays
    }
  }
  observers.name gtg.GiveToGetObserver
}
```

Tor, Ngan et al. [135]

```
seine.TorStar {
  resource.bandwidth random(0,1000)
  indicator.priority
  node.Client { fraction 0.9 selfish 0 }
  node.Relay {
    fraction 0.1
    capability bandwidth(0,500)
  }
  selfishness.smSelfish {
    actor Relay(0.33)
    behaviour.bhvS {
      defection.on relayOnion
    }
  }
  selfishness.smCooperative_reserve {
    actor Relay(0.33)
    behaviour.bhvC {
      activation bandwidth > 50
      defection.on relayOnion
    }
  }
  selfishness.smAdaptive {
    actor Relay(0.34)
    behaviour.bhvA {
      activation priority = 1
      defection.on relayOnion
    }
  }
  observers.name ts.TorStarObserver
}
```

BOINC, Anderson [18]

```
seine.VolunteerComputing {
  resource.CPU random(1.5,2.5)
  indicator.recentPoints
  node.Volunteer { selfish 0.1 }
  node.exclude 0
  selfishness.smV {
    actor Volunteer
    behaviour.task_computation {
      activation recentPoints < 10000
      freeriding.on processTask
    }
    behaviour.verification {
      collusion.on coordinateResults
    }
  }
  observers.name vc.VolunteerComputingObserver
}
```

Maze, Lian et al. [114]

```
seine.Maze {
  resource.sharedFiles random(0,1000)
  node.mazeClient {
    fraction 0.9
    selfish 0.1
  }
  node.mazeSybilClient {
    fraction 0.1
  }
  selfishness.sm {
    actor mazeClient mazeSybilClient
    behaviour.bhv {
      collusion.on uploadFilePieces notifySharedFiles
    }
  }
  observers.name maze.MazeObserver
}
```

FireSpam, Ben Mokhtar et al. [26]

```
seine.FireSpam {
  resource.spamFilteringCapability random(0,1000)
  node.GoodFiltering {
    fraction 0.25
    selfish 0.55
    capability spamFilteringCapability(750,1000)
  }
  node.AverageFiltering {
    fraction 0.5
    selfish 0.55
    capability spamFilteringCapability(250,750)
  }
  node.BadFiltering {
    fraction 0.25
    selfish 0.55
    capability spamFilteringCapability(0,250)
  }
  selfishness.smRational {
    actor GoodFiltering(0.9) AverageFiltering(0.9) BadFiltering(0.9)
    behaviour.bhvR {
      freeriding.on forwardMessage sendReport
    }
  }
  selfishness.smByzantine {
    actor GoodFiltering(0.1) AverageFiltering(0.1) BadFiltering(0.1)
    behaviour.bhvB {
      defection {
        probability 0.5
        on filterSpamMessage forwardMessage createReport sendReport monitorNode
      }
    }
  }
  observers.name fs.FireSpamObserver
}
```

Acting, Ben Mokhtar et al. [28]

```
seine.Acting {
  resource.bandwidth random(500,1000)
  node.Peer {
    fraction 0.7
    selfish 0.5
    capability bandwidth(0,1000)
  }
  node.Colluders {
    fraction 0.3
    capability bandwidth(0,1000)
  }
  node.exclude 0 # source
  selfishness.smP {
    actor Peer
    behaviour.bhvP {
      defection.on sendLogs
    }
  }
  selfishness.smC {
    actor Colluders
    behaviour.bhvC1 {
      collusion.on audit serveRequest
    }
    behaviour.bhvC2 {
      collusion.on proposeContents
    }
  }
  observers.name acting.ActingObserver
}
```

BitThief, Locher et al. [116]

```
seine.BitThief {
  resource.maxBandwidth
  resource.swarmSize uniform(80)
  node.BitTorrentClient {
    fraction 0.95
    selfish 0
    capability swarmSize(0,80) maxBandwidth(0,1000000)
  }
  node.BitThiefClient {
    fraction 0.05
    capability swarmSize(0,500) maxBandwidth(0,1000000)
  }
  node.exclude 0
  selfishness.smBT {
    actor BitThiefClient
    behaviour.bhv {
      defection.on sendFilePiece
      misreport.on sendHaveFileMessage notifyPeersetSize
      other.on requestRarestPiece
    }
  }
  observers.name bt.BitThiefObserver
}
```

Samsara, Cox and Noble [47]

```
seine.Samsara {
  resource.spaceUtilisation random(0,100)
  node.client { selfish 0.1 }
  selfishness.sm {
    actor client
    behaviour.bhv {
      defection {
        probability 0.5
        on storePartnerData storePartnerClaim
      }
    }
  }
  observers.name samsara.SamsaraObserver
}
```

Gnutella, Hughes et al. [87]

```
seine.Gnutella {
  resource.sharedFiles random(0,1000)
  node.BigAltruistic { fraction 0.021 selfish 0 }
  node.LittleAltruistic {
    fraction 0.129
    selfish 0
    capability sharedFiles(0,10)
  }
  node.Freerider {
    fraction 0.85
    capability sharedFiles(0,0)
  }
  selfishness.smF {
    actor Freerider
    behaviour.bhvF {
      defection.on uploadFilePieces notifySharedFiles
    }
  }
  observers.name gnutella.GnutellaObserver
}
```

Lifting, Guerraoui et al. [72]

```
seine.GossipLiveStreaming {
  resource.uploadBandwidth
  node.peer { selfish 0.25 capability uploadBandwidth(0,125000) }
  node.exclude 0  # source
  selfishness.smF {
    actor peer(0.9)
    behaviour.bhvF {
      freeriding { degree 0.3 on serveChunkRequest selectPartner }
      misreport { degree 0.3 on proposeChunks }
    }
  }
  selfishness.colluders {
    actor peer(0.1)
    behaviour.bhvC {
      collusion { probability 0.15 on selectPartner }
    }
  }
  observers.name lifting.LiftingObserver
}
```

Contracts, Piatek et al. [148]

```
seine.Contracts {
  resource.bwCapacity uniform(65000) # the median
  node.PPLiveClient { selfish 0.5 capability bwCapacity(0,150000) }
  node.exclude 0 # source
  selfishness.smS {
    actor PPLiveClient(0.6)
    behaviour.bhvS {
      freeriding.on serveUpdates
    }
  }
  selfishness.colluding {
    actor PPLiveClient(0.4)
    behaviour.collusive {
      collusion.on selectPartner
    }
  }
  observers.name contracts.ContractsObserver
}
```

[1] BitTorrent Live. URL https://btlive.tv/.

[2] eMule client for the eDonkey and the Kat networks. URL https://sourceforge.net/projects/emule/.

[3] The Network Simulator – ns-2. URL http://www.isi.edu/nsnam/ns/.

[4] Omnet++ - discrete event simulator. URL https://omnetpp.org/.

[5] PPLive. URL http://www.pptv.com/.

[6] The Racoon Framework. URL https://github.com/glenacota/racoon.

[7] The SETI@Home Problem, 2000. URL http://archive.is/PMK7g.

[8] SETI@home Frequently Asked Questions, 2002. URL http://archive.is/ovek2.

[9] BOINC stats, 2009. URL https://boincstats.com/en/forum/10/4597.

[10] PrimeGrid online forum: “Cheaters’ credits rescinded", 2013. URL http://archive.is/F5ZJc.

[11] World Cummunity Grid online forum: “The PrimeGrid Cheating Scandal", 2013. URL http://archive.is/vefXK.

[12] BOINC user survey results, 2016. URL http://boinc.berkeley.edu/poll/poll_results.php.

[13] Ittai Abraham, Danny Dolev, Rica Gonen, and Joe Halpern. Distributed computing meets game theory: robust mechanisms for rational secret sharing and multiparty computation. In *Proceedings of the 25th annual ACM symposium on Principles of distributed computing*, pages 53–62. ACM, 2006.

[14] Paarijaat Aditya, Mingchen Zhao, Yin Lin, Andreas Haeberlen, Peter Druschel, Bruce Maggs, and Bill Wishon. Reliable client accounting for p2p-infrastructure hybrids. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 8–8. USENIX Association, 2012.

[15] Amitanand S Aiyer, Lorenzo Alvisi, Allen Clement, Mike Dahlin, Jean-Philippe Martin, and Carl Porth. Bar fault tolerance for cooperative services. In *ACM SIGOPS operating systems review*, volume 39, pages 45–58. ACM, 2005.

[16] Jamal N Al-Karaki and Ahmed E Kamal. Routing techniques in wireless sensor networks: a survey. *IEEE wireless communications*, 11(6):6–28, 2004.

[17] Luzi Anderegg and Stephan Eidenbenz. Ad hoc-vcg: a truthful and cost-efficient routing protocol for mobile ad hoc networks with selfish agents. In *Proceedings of the 9th annual international conference on Mobile computing and networking*. ACM, 2003.

[18] David P Anderson. Boinc: A system for public-resource computing and storage. In *Grid Computing, 2004. Proceedings. 5th IEEE/ACM International Workshop on*, pages 4–10. IEEE, 2004.

[19] David P Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@ home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.

[20] Antonio Fernández Anta, Chryssis Georgiou, and Miguel A Mosteiro. Algorithmic mechanisms for internet-based master-worker computing with untrusted and selfish workers. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–11. IEEE, 2010.

[21] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33, 2004.

[22] Sorav Bansal and Mary Baker. Observation-based cooperation enforcement in ad hoc networks. *arXiv preprint cs/0307012*, 2003.

[23] Salman A Baset and Henning Schulzrinne. An analysis of the skype peer-to-peer internet telephony protocol. *arXiv preprint cs/0412017*, 2004.

[24] Anirban Basu, Simon Fleming, James Stanier, Stephen Naicken, Ian Wakeman, and Vijay K Gurbani. The state of Peer-to-Peer network simulators. *ACM Computing Surveys (CSUR)*, 45(4), 2013.

[25] Mira Belenkiy, Melissa Chase, C Chris Erway, John Jannotti, Alptekin Küpçü, Anna Lysyanskaya, and Eric Rachlin. Making p2p accountable without losing privacy. In *Proceedings of the 2007 ACM workshop on Privacy in electronic society*, pages 31–40. ACM, 2007.

[26] Sonia Ben Mokhtar, Alessio Pace, and Vivien Quema. Firespam: Spam resilient gossiping in the bar model. In *Reliable Distributed Systems, 2010 29th IEEE Symposium on*, pages 225–234. IEEE, 2010.

[27] Sonia Ben Mokhtar, Gautier Berthou, Amadou Diarra, Vivien Quéma, and Ali Shoker. Rac: a freerider-resilient, scalable, anonymous communication protocol. In *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*, pages 520–529. IEEE, 2013.

[28] Sonia Ben Mokhtar, Jérémie Decouchant, and Vivien Quéma. Acting: Accurate freerider tracking in gossip. In *2014 IEEE 33rd International Symposium on Reliable Distributed Systems*, pages 291–300. IEEE, 2014.

[29] Naouel Ben Salem, Levente Buttyán, Jean-Pierre Hubaux, and Markus Jakobsson. Node cooperation in hybrid ad hoc networks. *Mobile Computing, IEEE Transactions on*, 5(4), 2006.

[30] Radu Mihai Berciu. *Designing incentives in P2P systems.* PhD thesis, 2013.

[31] Alberto Blanc, Yi-Kai Liu, and Amin Vahdat. Designing incentives for peer-to-peer routing. In *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, volume 1, pages 374–385. IEEE, 2005.

[32] Tomasz Buchert, Cristian Ruiz, Lucas Nussbaum, and Olivier Richard. A survey of general-purpose experiment management tools for distributed systems. *Future Generation Computer Systems*, 45:1–12, 2015.

[33] Levente Buttyán and Jean-Pierre Hubaux. Nuglets: a virtual currency to stimulate cooperation in self-organized mobile ad hoc networks. Technical report, 2001.

[34] Levente Buttyán and Jean-Pierre Hubaux. Stimulating cooperation in self-organizing mobile ad hoc networks. *Mobile Networks and Applications*, 8(5):579–592, 2003.

[35] Levente Buttyán and Jean-Pierre Hubaux. *Security and cooperation in wireless networks: thwarting malicious and selfish behavior in the age of ubiquitous computing*. Cambridge University Press, 2007.

[36] Levente Buttyán, László Dóra, Márk Félegyházi, and István Vajda. Barter trade improves message delivery in opportunistic networks. *Ad Hoc Networks*, 8(1):1–14, 2010.

[37] Franck Cappello, Eddy Caron, Michel Dayde, Frédéric Desprez, Yvon Jégou, Pascale Primet, Emmanuel Jeannot, Stéphane Lanteri, Julien Leduc, Nouredine Melab, et al. Grid'5000: a large scale and highly reconfigurable grid experimental testbed. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, pages 99–106. IEEE Computer Society, 2005.

[38] Christopher D Carothers, Ryan LaFortune, William D Smith, and Mark Gilder. A case study in modeling large-scale peer-to-peer file-sharing networks using discrete-event simulation. In *Proceedings of the 2006 European of Modeling and Simulation Symposium which is part of the I3M Multiconference), Barcelona, Spain*, 2006.

[39] Antonio Carzaniga, Alessandra Gorla, and Mauro Pezzè. Handling software faults with redundancy. In *Architecting Dependable Systems VI*, pages 148–171. Springer, 2009.

[40] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[41] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. Planetlab: an overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review*, 33(3):3–12, 2003.

[42] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 189–204. ACM, 2007.

[43] Gianluca Ciccarelli and Renato Lo Cigno. Collusion in peer-to-peer systems. *Computer Networks*, 55 (15):3517–3532, 2011.

[44] Cisco Visual Networking Index Cisco. White paper: Cisco vni forecast and methodology, 2015-2020, 2016.

[45] Bram Cohen. Incentives build robustness in bittorrent. In *Workshop on Economics of Peer-to-Peer systems*, volume 6, pages 68–72, 2003.

[46] Thomas H.. Cormen, Charles Eric Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*, volume 6. MIT press Cambridge, 2001.

[47] Landon P Cox and Brian D Noble. Samsara: Honor among thieves in peer-to-peer storage. *ACM SIGOPS Operating Systems Review*, 37(5):120–132, 2003.

[48] Landon P Cox, Christopher D Murray, and Brian D Noble. Pastiche: Making backup cheap and easy. *ACM SIGOPS Operating Systems Review*, 36(SI):285–298, 2002.

[49] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged configuration using feature models. In *Software Product Lines*, pages 266–283. Springer, 2004.

[50] Peter J Denning. Acm president's letter: What is experimental computer science? *Communications of the ACM*, 23(10):543–544, 1980.

[51] Peter J Denning. Acm president's letter: performance analysis: experimental computer science as its best. *Communications of the ACM*, 24(11):725–727, 1981.

[52] Amadou Diarra, Sonia Ben Mokhtar, Pierre-Louis Aublin, and Vivien Quéma. Fullreview: Practical accountability in presence of selfish nodes - technical report. URL https://goo.gl/GSA11A.

[53] Amadou Diarra, Sonia Ben Mokhtar, Pierre-Louis Aublin, and Vivien Quéma. Fullreview: Practical accountability in presence of selfish nodes. In *2014 IEEE 33rd International Symposium on Reliable Distributed Systems*, pages 271–280. IEEE, 2014.

[54] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. Technical report, DTIC Document, 2004.

[55] John R Douceur. The sybil attack. In *Peer-to-peer Systems*. Springer, 2002.

[56] Kevin Fall. A delay-tolerant network architecture for challenged internets. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 27–34. ACM, 2003.

[57] Joan Feigenbaum and Scott Shenker. *Distributed algorithmic mechanism design: Recent results and future directions*. September, 2002.

[58] Jerome A Feldman and William R Sutherland. Rejuvenating experimental computer science: a report to the national science foundation and others. *Communications of the ACM*, 22(9):497–502, 1979.

[59] Michal Feldman and John Chuang. Overcoming free-riding behavior in peer-to-peer systems. *ACM SIGecom Exchanges*, 5(4), 2005.

[60] Michal Feldman, Christos Papadimitriou, John Chuang, and Ion Stoica. Free-riding and whitewashing in peer-to-peer systems. *Selected Areas in Communications, IEEE Journal on*, 24(5), 2006.

[61] Robert Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit. *Aspect-Oriented software development*. Addison-Wesley Professional, 2004.

[62] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International journal of high performance computing applications*, 15(3):200–222, 2001.

[63] Julien Freudiger, Mohammad Hossein Manshaei, Jean-Pierre Hubaux, and David C Parkes. On non-cooperative location privacy: a game-theoretic analysis. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 324–337. ACM, 2009.

[64] Roy Friedman, Alexander Libov, and Ymir Vigfusson. Molstream: A modular rapid development and evaluation framework for live p2p streaming. In *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, pages 278–287. IEEE, 2014.

[65] Roy Friedman, Alexander Libov, and Ymir Vigfussony. Distilling the ingredients of p2p live streaming systems. In *Peer-to-Peer Computing (P2P), 2015 IEEE International Conference on*, pages 1–10. IEEE, 2015.

[66] Wojciech Galuba, Karl Aberer, Zoran Despotovic, and Wolfgang Kellerer. Protopeer: a p2p toolkit bridging the gap between simulation and live deployement. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, page 60. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2009.

[67] Flavio D Garcia and Jaap-Henk Hoepman. Off-line karma: A decentralized currency for peer-to-peer and grid applications. In *International Conference on Applied Cryptography and Network Security*, pages 364–377. Springer, 2005.

[68] Gabriele Gianini, Ernesto Damiani, Tobias R Mayer, David Coquil, Harald Kosch, and Lionel Brunie. Many-player inspection games in networked environments. In *2013 7th IEEE International Conference on Digital Ecosystems and Technologies (DEST)*, pages 1–6. IEEE, 2013.

[69] Gabriele Gianini, Marco Cremonini, Andrea Rainini, Guido Lena Cota, and Leopold Ghemmogne Fossi. A game theoretic approach to vulnerability patching. In *Information and Communication Technology Research (ICTRC), International Conference on*, pages 88–91. IEEE, 2015.

[70] David Goldschlag, Michael Reed, and Paul Syverson. Onion routing. *Communications of the ACM*, 42(2):39–41, 1999.

[71] Marco Gramaglia, Manuel Urueña, and Isaias Martinez-Yelmo. Off-line incentive mechanism for long-term p2p backup storage. *Computer Communications*, 35(12):1516–1526, 2012.

[72] Rachid Guerraoui, Kévin Huguenin, Anne-Marie Kermarrec, Maxime Monod, and Swagatika Prusty. Lifting: lightweight freerider-tracking in gossip. In *Proceedings of the ACM/IFIP/USENIX 11th International Conference on Middleware*, pages 313–333. Springer-Verlag, 2010.

[73] Rohit Gupta and Arun K Somani. Game theory as a tool to strategize as well as predict nodes' behavior in peer-to-peer networks. In *Parallel and Distributed Systems, 2005. Proceedings. 11th International Conference on*, volume 1. IEEE, 2005.

[74] Jens Gustedt, Emmanuel Jeannot, and Martin Quinson. Experimental validation in large-scale systems: a survey of methodologies. *Parallel Processing Letters*, 2009.

[75] Fatima Lamia Haddi and Mahfoud Benchaïba. A survey of incentive mechanisms in static and mobile p2p systems. *Journal of Network and Computer Applications*, 58:108–118, 2015.

[76] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. Peerreview: Practical accountability for distributed systems. In *ACM SIGOPS operating systems review*, volume 41, pages 175–188. ACM, 2007.

[77] Andreas Haeberlen, Paarijaat Aditya, Rodrigo Rodrigues, and Peter Druschel. Accountable virtual machines. In *OSDI*, pages 119–134, 2010.

[78] Sidath B Handurukande, A-M Kermarrec, Fabrice Le Fessant, Laurent Massoulié, and Simon Patarin. *Peer sharing behaviour in the edonkey network, and implications for the design of server-less file sharing systems*, volume 40. ACM, 2006.

[79] Omar Hasan, Lionel Brunie, Elisa Bertino, and Ning Shang. A decentralized privacy preserving reputation protocol for the malicious adversarial model. *IEEE Transactions on Information Forensics and Security*, 8(6):949–962, 2013.

[80] Thomas R Henderson, Mathieu Lacage, George F Riley, C Dowell, and J Kopena. Network simulations with the ns-3 simulator. *SIGCOMM demonstration*, 15:17, 2008.

[81] Victor Heorhiadi, Shriram Rajagopalan, Hani Jamjoom, Michael K Reiter, and Vyas Sekar. Gremlin: systematic resilience testing of microservices. In *Distributed Computing Systems (ICDCS), 2016 IEEE 36th International Conference on*, pages 57–66. IEEE, 2016.

[82] Enrique Hernandez-Orallo, Manuel David Serrat Olmos, Juan-Carlos Cano, Carlos T Calafate, and Pietro Manzoni. Cocowa: a collaborative contact-based watchdog for detecting selfish nodes. *IEEE Transactions on Mobile Computing*, 14(6):1162–1175, 2015.

[83] Kevin Hoffman, David Zage, and Cristina Nita-Rotaru. A survey of attack and defense techniques for reputation systems. *ACM Computing Surveys (CSUR)*, 42(1), 2009.

[84] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. Introduction to automata theory, languages, and computation. *ACM SIGACT News*, 32(1):60–65, 2001.

[85] Yusuo Hu, Dafan Dong, Jiang Li, and Feng Wu. Efficient and incentive-compatible resource allocation mechanism for p2p-assisted content delivery systems. *Future Generation Computer Systems*, 29 (6):1611–1620, 2013.

[86] Chen Hua, Yang Mao, Han Jinqiang, Deng Haiqing, and Li Xiaoming. Maze: a social peer-to-peer network. In *E-Commerce Technology for Dynamic E-Business, 2004. IEEE International Conference on*, pages 290–293. IEEE, 2004.

[87] Daniel Hughes, Geoff Coulson, and James Walkerdine. Free riding on gnutella revisited: the bell tolls? *IEEE distributed systems online*, 6(6), 2005.

[88] Nwokedi Idika and Aditya P Mathur. A survey of malware detection techniques. *Purdue University*, 48, 2007.

[89] Rob Jansen, Nicholas Hopper, and Yongdae Kim. Recruiting new tor relays with braids. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 319–328. ACM, 2010.

[90] Hamed Janzadeh, Kaveh Fayazbakhsh, Mehdi Dehghan, and Mehran S Fallah. A secure credit-based cooperation stimulating mechanism for manets using hash chains. *Future Generation Computer Systems*, 25(8), 2009.

[91] Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten Van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems (TOCS)*, 25(3):8, 2007.

[92] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. A modular paradigm for building self-organizing peer-to-peer applications. In *Engineering Self-Organizing Systems Workshop (ESOA)*, pages 265–282, 2009.

[93] Seung Jun and Mustaque Ahamad. Incentives in bittorrent induce free riding. In *Proceedings of the 2005 ACM SIGCOMM workshop on Economics of peer-to-peer systems*, pages 116–121. ACM, 2005.

[94] Sepandar D Kamvar, Mario T Schlosser, and Hector Garcia-Molina. The eigentrust algorithm for reputation management in p2p networks. In *Proceedings of the 12th international conference on World Wide Web*. ACM, 2003.

[95] Jonathan Katz. Bridging game theory and cryptography: Recent results and future directions. In *Theory of Cryptography Conference*, pages 251–272. Springer, 2008.

[96] A-M Kermarrec, Laurent Massoulié, and Ayalvadi J. Ganesh. Probabilistic reliable dissemination in large-scale systems. *IEEE Transactions on Parallel and Distributed systems*, 14(3):248–258, 2003.

[97] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. An overview of AspectJ. In *European Conf. on Object-Oriented Programming*. Springer, 2001.

[98] Charles Edwin Killian, James W Anderson, Ryan Braud, Ranjit Jhala, and Amin M Vahdat. Mace: language support for building distributed systems. In *ACM SIGPLAN Notices*, volume 42, pages 179–188. ACM, 2007.

[99] Daphne Koller, Nimrod Megiddo, and Bernhard Von Stengel. Fast algorithms for finding randomized strategies in game trees. In *Proceedings of the 26th annual ACM symposium on Theory of computing*, pages 750–759. ACM, 1994.

[100] Zhen Kong and Yu-Kwong Kwok. Efficient wireless packet scheduling in a non-cooperative environment: Game theoretic analysis and algorithms. *Journal of Parallel and Distributed Computing*, 70 (8):790–799, 2010.

[101] Hartmut König. *Protocol Engineering*. Springer, 2012.

[102] Eleni Koutrouli and Aphrodite Tsalgatidou. Taxonomy of attacks and defense mechanisms in p2p reputation systems—lessons for reputation system designers. *Computer Science Review*, 6(2):47–70, 2012.

[103] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, et al. Oceanstore: An architecture for global-scale persistent storage. *ACM Sigplan Notices*, 35(11):190–201, 2000.

[104] Yu-Kwong Kwok, Kai Hwang, and ShanShan Song. Selfish grids: Game-theoretic modeling and nas/psa benchmark evaluation. *IEEE Transactions on Parallel and Distributed Systems*, 18(5):621–636, 2007.

[105] Butler W Lampson. Computer security in the real world. *Computer*, 37(6):37–46, 2004.

[106] Guido Lena Cota, Sonia Ben Mokhtar, Julia Lawall, Gilles Muller, Gabriele Gianini, Ernesto Damiani, and Lionel Brunie. A framework for the design configuration of accountable selfish-resilient peer-to-peer systems. In *Reliable Distributed Systems (SRDS), IEEE 34th Symposium on*, pages 276–285. IEEE, 2015.

[107] Jan Karel Lenstra. *Local search in combinatorial optimization*. Princeton University Press, 2003.

[108] Lorenzo Leonini, Étienne Rivière, and Pascal Felber. Splay: Distributed systems evaluation made simple (or how to turn ideas into live systems in a breeze). In *NSDI*, volume 9, pages 185–198, 2009.

[109] Dave Levin, John R Douceur, Jacob R Lorch, and Thomas Moscibroda. Trinc: Small trusted hardware for large distributed systems. In *NSDI*, volume 9, pages 1–14, 2009.

[110] Brian Neil Levine, Clay Shields, and N Boris Margolin. A survey of solutions to the sybil attack. *University of Massachusetts Amherst, Amherst, MA*, 7, 2006.

[111] Harry C Li, Allen Clement, Edmund L Wong, Jeff Napper, Indrajit Roy, Lorenzo Alvisi, and Michael Dahlin. Bar gossip. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 191–204. USENIX Association, 2006.

[112] Harry C Li, Allen Clement, Mirco Marchetti, Manos Kapritsos, Luke Robison, Lorenzo Alvisi, and Mike Dahlin. Flightpath: Obedience vs. choice in cooperative services. In *OSDI*, volume 8, 2008.

[113] Qinghua Li, Sencun Zhu, and Guohong Cao. Routing in socially selfish delay tolerant networks. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9. IEEE, 2010.

[114] Qiao Lian, Zheng Zhang, Mao Yang, Ben Y Zhao, Yafei Dai, and Xiaoming Li. An empirical study of collusion behavior in the maze p2p file-sharing system. In *27th International Conference on Distributed Computing Systems (ICDCS'07)*, pages 56–56. IEEE, 2007.

[115] Jian Liang, Rakesh Kumar, Yongjian Xi, and Keith W Ross. Pollution in p2p file sharing systems. In *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, volume 2, pages 1174–1185. IEEE, 2005.

[116] Thomas Locher, Patrick Moor, Stefan Schmid, and Roger Wattenhofer. Free riding in bittorrent is cheap. In *Proc. Workshop on Hot Topics in Networks (HotNets)*, pages 85–90. Citeseer, 2006.

[117] Nik Looker, Malcolm Munro, and Jie Xu. Ws-fit: A tool for dependability analysis of web services. 2004.

[118] Ratul Mahajan, Maya Rodrig, David Wetherall, and John Zahorjan. Experiences applying game theory to system design. In *Proceedings of the ACM SIGCOMM workshop on Practice and theory of incentives in networked systems*, pages 183–190. ACM, 2004.

[119] Mohamed MEA Mahmoud and Xuemin Shen. A secure payment scheme with low communication and processing overhead for multihop wireless networks. *Parallel and Distributed Systems, IEEE Transactions on*, 24(2), 2013.

[120] George J Mailath. Do people play nash equilibrium? lessons from evolutionary game theory. *Journal of Economic Literature*, 36(3):1347–1374, 1998.

[121] Sergio Marti and Hector Garcia-Molina. Taxonomy of trust: Categorizing p2p reputation systems. *Computer Networks*, 50(4):472–484, 2006.

[122] Tobias R Mayer, David Coquil, Christian Schoernich, and Harald Kosch. Rcourse: a robustness benchmarking suite for publish/subscribe overlay simulations with peersim. In *Proceedings of the First Workshop on P2P and Dependability*, page 3. ACM, 2012.

[123] Jim McCoy. Mojo nation responds. *URL http://www. openp2p. com/pub/a/p2p/2001/01/11/mojoht. html*, 2001.

[124] Alessandro Mei and Julinda Stefa. Give2get: Forwarding in social mobile wireless networks of selfish individuals. *IEEE Transactions on Dependable and Secure Computing*, 9(4):569–582, 2012.

[125] Jingwei Miao, Omar Hasan, Sonia Ben Mokhtar, Lionel Brunie, and Kangbin Yim. An investigation on the unwillingness of nodes to participate in mobile delay tolerant network routing. *International Journal of Information Management*, 33(2):252–262, 2013.

[126] Nikolaos Michalakis, Robert Soulé, and Robert Grimm. Ensuring content integrity for untrusted peer-to-peer content distribution networks. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, pages 11–11. USENIX Association, 2007.

[127] John C Mitchell and Vanessa Teague. Autonomous nodes and distributed mechanisms. In *Software Security—Theories and Systems*, pages 58–83. Springer, 2003.

[128] Alberto Montresor and Márk Jelasity. Peersim: A scalable p2p simulator. In *IEEE 9th International Conference on Peer-to-Peer Computing*, pages 99–100. IEEE, 2009.

[129] Hayam Mousa, Sonia Ben Mokhtar, Omar Hasan, Osama Younes, Mohiy Hadhoud, and Lionel Brunie. Trust management and reputation systems in mobile participatory sensing applications: A survey. *Computer Networks*, 90:49–73, 2015.

[130] Roger B Myerson. *Game theory*. Harvard university press, 2013.

[131] Stephen Naicken, Anirban Basu, Barnaby Livingston, and Sethalat Rodhetbhai. A survey of peer-to-peer network simulators. In *Proceedings of The 7th Annual Postgraduate Symposium, Liverpool, UK*, volume 2, 2006.

[132] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[133] Animesh Nandi, Tsuen-Wan Johnny Ngan, Atul Singh, Peter Druschel, and Dan S Wallach. Scrivener: Providing incentives in cooperative content distribution systems. In *Proceedings of the ACM/IFIP/USENIX 2005 International Conference on Middleware*, pages 270–291. Springer-Verlag New York, Inc., 2005.

[134] T-W.J. Ngan, D.S. Wallach, and P. Druschel. Incentives-compatible peer-to-peer multicast. 2004.

[135] Tsuen-Wan Ngan, Roger Dingledine, and Dan S Wallach. Building incentives into tor. In *International Conference on Financial Cryptography and Data Security*, pages 238–256. Springer, 2010.

[136] Tsuen-Wan Johnny Ngan, Animesh Nandi, Atul Singh, Dan S Wallach, and Peter Druschel. On designing incentives-compatible peer-to-peer systems. 2004.

[137] Seth James Nielson, Scott A Crosby, and Dan S Wallach. A taxonomy of rational attacks. In *International Workshop on Peer-to-Peer Systems*, pages 36–46. Springer, 2005.

[138] Noam Nisan and Amir Ronen. Algorithmic mechanism design. In *Proceedings of the 31st annual ACM symposium on Theory of computing*, pages 129–140. ACM, 1999.

[139] Noam Nisan, Tim Roughgarden, Eva Tardos, and Vijay V Vazirani. *Algorithmic game theory*, volume 1. Cambridge University Press Cambridge, 2007.

[140] Philipp Obreiter, Birgitta König-Ries, and Michael Klein. Stimulating cooperative behavior of autonomous devices: An analysis of requirements and existing approaches. 2003.

[141] Joao FA Oliveira, Ítalo Cunha, Eliseu C Miguel, Marcus VM Rocha, Alex B Vieira, and Sérgio VA Campos. Can peer-to-peer live streaming systems coexist with free riders? In *IEEE P2P 2013 Proceedings*, pages 1–5. IEEE, 2013.

[142] Esther Palomar, Almudena Alcaide, Arturo Ribagorda, and Yan Zhang. The peer's dilemma: A general framework to examine cooperation in pure peer-to-peer systems. *Computer Networks*, 56 (17), 2012.

[143] Christos Papadimitriou. Algorithms, games, and the internet. In *Proceedings of the 33rd annual ACM symposium on Theory of computing*, pages 749–753. ACM, 2001.

[144] Thanasis G Papaioannou and George D Stamoulis. Reputation-based policies that provide the right incentives in peer-to-peer environments. *Computer Networks*, 50(4):563–578, 2006.

[145] Jaeok Park and Mihaela van der Schaar. Pricing and incentives in peer-to-peer networks, 2010.

[146] Terence Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[147] Andrea Passarella. A survey on content-centric technologies for the current internet: Cdn and p2p solutions. *Computer Communications*, 35(1):1–32, 2012.

[148] Michael Piatek, Arvind Krishnamurthy, Arun Venkataramani, Yang Richard Yang, David Zhang, and Alexander Jaffe. Contracts: Practical contribution incentives for p2p live streaming. In *NSDI*, pages 81–94, 2010.

[149] Dongyu Qiu and Rayadurgam Srikant. Modeling and performance analysis of bittorrent-like peer-to-peer networks. In *ACM SIGCOMM computer communication review*, volume 34, pages 367–378. ACM, 2004.

[150] Rameez Rahman, Tamás Vinkó, David Hales, Johan Pouwelse, and Henk Sips. Design space analysis for modeling incentives in distributed systems. In *ACM SIGCOMM Computer Communication Review*, volume 41. ACM, 2011.

[151] Rodrigo Rodrigues and Peter Druschel. Peer-to-peer systems. *Communications of the ACM*, 53(10): 72–82, 2010.

[152] Giancarlo Ruffo and Rossano Schifanella. Fairpeers: Efficient profit sharing in fair peer-to-peer market places. *Journal of Network and Systems Management*, 15(3):355–382, 2007.

[153] Normalia Samian, Zuriati Ahmad Zukarnain, Winston KG Seah, Azizol Abdullah, and Zurina Mohd Hanapi. Cooperation stimulation mechanisms for wireless multihop networks: A survey. *Journal of Network and Computer Applications*, 54, 2015.

[154] Ravi Sandhu and Xinwen Zhang. Peer-to-peer access control architecture using trusted computing technology. In *Proceedings of the 10th ACM symposium on Access control models and technologies*, pages 147–158. ACM, 2005.

[155] Amit Sangroya, Damian Serrano, and Sara Bouchenak. Benchmarking dependability of mapreduce systems. In *Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium on*, pages 21–30. IEEE, 2012.

[156] Karen Scarfone and Peter Mell. Guide to intrusion detection and prevention systems (idps). *NIST special publication*, 800(2007):94, 2007.

[157] Alexander Schaub, Rémi Bazin, Omar Hasan, and Lionel Brunie. A trustless privacy-preserving reputation system. In *IFIP International Information Security and Privacy Conference*, pages 398–411. Springer, 2016.

[158] Bruce Schneier and John Kelsey. Cryptographic support for secure logs on untrusted machines. In *USENIX Security*, 1998.

[159] Haiying Shen and Ze Li. Arm: An account-based hierarchical reputation management system for wireless ad hoc networks. In *Distributed Computing Systems Workshops, 2008. ICDCS'08. 28th International Conference on*. IEEE, 2008.

[160] Jeffrey Shneidman and David C Parkes. Rationality and self-interest in peer to peer networks. In *International Workshop on Peer-to-Peer Systems*, pages 139–148. Springer, 2003.

[161] Jeffrey Shneidman and David C Parkes. Specification faithfulness in networks with rational nodes. In *Proceedings of the 23rd annual ACM symposium on Principles of distributed computing*. ACM, 2004.

[162] Christos Siaterlis, Andres Perez Garcia, and Béla Genge. On the use of emulab testbeds for scientifically rigorous experiments. *IEEE Communications Surveys & Tutorials*, 15(2):929–942, 2013.

[163] Thomas Silverston, Olivier Fourmaux, and Jon Crowcroft. Towards an incentive mechanism for peer-to-peer multimedia live streaming systems. In *8th International Conference on Peer-to-Peer Computing*, pages 125–128. IEEE, 2008.

[164] Atul Singh, Tsuen-wan Ngan, Peter Druschel, and Dan S. Wallach. Eclipse attacks on overlay networks: Threats and defenses. In *In IEEE INFOCOM*. Citeseer, 2006.

[165] Michael Sirivianos, Jong Han Park, Xiaowei Yang, and Stanislaw Jarecki. Dandelion: Cooperative content distribution with robust incentives. In *USENIX Annual Technical Conference*, volume 7, 2007.

[166] Stefano Traverso, Luca Abeni, Robert Birke, Csaba Kiraly, Emilio Leonardi, R Lo Cigno, and Marco Mellia. Experimental comparison of neighborhood filtering strategies in unstructured p2p-tv systems. In *IEEE 12th International Conference on Peer-to-Peer Computing (P2P)*, pages 13–24. IEEE, 2012.

[167] Ramona Trestian, Olga Ormond, and Gabriel-Miro Muntean. Game theory-based network selection: Solutions and challenges. *IEEE Communications surveys & tutorials*, 14(4):1212–1231, 2012.

[168] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, and David Becker. Scalability and accuracy in a large-scale network emulator. *ACM SIGOPS Operating Systems Review*, 36(SI):271–284, 2002.

[169] Athanasios V Vasilakos, Yan Zhang, and Thrasyvoulos Spyropoulos. *Delay tolerant networks: Protocols and applications*. CRC press, 2016.

[170] Vivek Vishnumurthy, Sangeeth Chandrakumar, and Emin Gun Sirer. Karma: A secure economic framework for peer-to-peer resource sharing. In *Workshop on Economics of Peer-to-Peer Systems*, volume 35, 2003.

[171] Yufeng Wang, Akihiro Nakao, Athanasios V Vasilakos, and Jianhua Ma. P2p soft security: On evolutionary dynamics of p2p incentive mechanism. *Computer Communications*, 34(3):241–249, 2011.

[172] Jörgen W Weibull. *Evolutionary game theory*. MIT press, 1997.

[173] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. *ACM SIGOPS Operating Systems Review*, 36(SI):255–270, 2002.

[174] Zhifeng Xiao and Yang Xiao. Peerreview re–evaluation for accountability in distributed systems or networks. *International Journal of Security and Networks*, 7(1):40–58, 2012.

[175] Mao Yang, Zheng Zhang, Xiaoming Li, and Yafei Dai. An empirical study of free-riding behavior in the maze p2p file-sharing system. In *International Workshop on Peer-to-Peer Systems*, pages 182–192. Springer, 2005.

[176] Younghwan Yoo and Dharma P Agrawal. Why does it pay to be selfish in a manet? *IEEE Wireless Communications*, 13(6):87–97, 2006.

[177] Aydan R Yumerefendi and Jeffrey S Chase. Trust but verify: accountability for network services. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, page 37. ACM, 2004.

[178] Aydan R Yumerefendi and Jeffrey S Chase. The role of accountability in dependable distributed systems. In *Proceedings of HotDep*, volume 5, pages 3–3. Citeseer, 2005.

[179] Aydan R Yumerefendi and Jeffrey S Chase. Strong accountability for network storage. *ACM Transactions on Storage (TOS)*, 3(3):11, 2007.

[180] Matthew Yurkewych, Brian N Levine, and Arnold L Rosenberg. On the cost-ineffectiveness of redundancy in commercial p2p computing. In *Proceedings of the 12th ACM conference on Computer and communications security*. ACM, 2005.

[181] Manaf Zghaibeh and Fotios C Harmantzis. Revisiting free riding and the tit-for-tat in bittorrent: A measurement study. *Peer-to-Peer Networking and Applications*, 1(2):162–173, 2008.

[182] Manaf Zghaibeh, Kostas G Anagnostakis, and Fotios C Harmantzis. The behavior of free riders in bit torrent networks. In *Handbook of Peer-to-Peer Networking*, pages 1207–1230. Springer, 2010.

[183] Bridge Qiao Zhao, John CS Lui, and Dah-Ming Chiu. A mathematical framework for analyzing adaptive incentive protocols in p2p networks. *IEEE/ACM Transactions on Networking*, 20(2):367–380, 2012.

[184] Sheng Zhong, Jiang Chen, and Yang Richard Yang. Sprite: A simple, cheat-proof, credit-based system for mobile ad-hoc networks. In *INFOCOM 2003. 22nd Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, volume 3, pages 1987–1997. IEEE, 2003.

[185] Runfang Zhou and Kai Hwang. Powertrust: A robust and scalable reputation system for trusted peer-to-peer computing. *Parallel and Distributed Systems, IEEE Transactions on*, 18(4), 2007.

[186] Runfang Zhou, Kai Hwang, and Min Cai. Gossiptrust for fast reputation aggregation in peer-to-peer networks. *Knowledge and Data Engineering, IEEE Transactions on*, 20(9), 2008.

[187] Haojin Zhu, Xiaodong Lin, Rongxing Lu, and Xuemin Shen. A secure incentive scheme for delay tolerant networks. In *Communications and Networking in China, ChinaCom 2008. 3rd International Conference on*, pages 23–28. IEEE, 2008.

[188] Haojin Zhu, Xiaodong Lin, Rongxing Lu, Yanfei Fan, and Xuemin Shen. Smart: A secure multilayer credit-based incentive scheme for delay-tolerant networks. *IEEE Transactions on Vehicular Technology*, 58(8):4628–4639, 2009.