A Framework for the Design Configuration of Accountable Selfish-Resilient Peer-to-Peer Systems

Guido Lena Cota*, Sonia Ben Mokhtar†, Julia Lawall‡, Gilles Muller‡,
Gabriele Gianini*, Ernesto Damiani*, Lionel Brunie†
*Università degli Studi di Milano, †LIRIS-CNRS-INSA Lyon, ‡Inria/LIP6-Whisper

***Abstract*—A challenge in designing a peer-to-peer (P2P) system is to ensure that the system is able to tolerate selfish nodes that strategically deviate from their specification whenever doing so is convenient. In this paper, we propose *RACOON*, a framework for the design of P2P systems that are resilient to selfish behaviours. While most existing solutions target specific systems or types of selfishness, *RACOON* proposes a generic and semi-automatic approach that achieves robust and reusable results. Also, *RACOON* supports the system designer in the performance-oriented tuning of the system, by proposing a novel approach that combines Game Theory and simulations. We illustrate the benefits of using *RACOON* by designing two P2P systems: a live streaming and an anonymous communication system. In simulations and a real deployment of the two applications on a testbed comprising 100 nodes, the systems designed using *RACOON* achieve both resilience to selfish nodes and high performance.**

I. Introduction

Today, peer-to-peer (P2P) systems, such as file sharing (e.g., BitTorrent, eDonkey), live streaming (e.g., PPLIVE, Popcorn Time), and instant messaging and voice over IP (e.g., Skype), are among those generating the most Internet traffic [7], [8], [9]. The success of these systems mainly resides in their high scalability and robustness to failure, without requiring costly dedicated servers. Common to all P2P systems, is the assumption that nodes are willing to share their communication and computational resources with others. However, in practice [11], [14], [20], real systems suffer from selfish nodes that decide whether to cooperate depending on the behaviour that increases their utility. For example, a selfish node may choose to receive video data without sharing their data with others, in order to save bandwidth.

A number of solutions have been proposed to deal with selfishness in P2P systems [1], [2], [3], [17], [24], [25]. Most of these solutions rely on Game Theory for modelling and reasoning about selfish behaviours. The classical process for designing selfish-resilient P2P systems first requires that the system designer exhaustively list the set of deviations that can be performed by selfish nodes. Then, the designer has to carefully provide an incentive/countermeasure for each deviation he has identified. Finally, the designer uses game theoretic arguments to prove that the resulting system is a *Nash Equilibrium* [27]. This proof guarantees that the best strategy for selfish nodes, with respect to a known utility function, is to conform to the system specification. However, carrying out this process is complex, time consuming, and error prone, especially for designers that are not experts in Game Theory.

An alternative solution to the above *static* approach, is to employ recent accountability mechanisms (e.g., FullReview [13], Lifting [17], and PeerReview [19]), with the goal of *dynamically* forcing nodes to be responsible for their actions. Noteworthy among these mechanisms is FullReview, which appears to be the only one specifically designed for dealing with selfish nodes. In an accountable system, each node maintains a *secure log* to record its interactions with other nodes. Each node is further associated with a set of *monitor* nodes, which periodically check whether the log entries correspond to a correct execution of the underlying protocol. If any deviation is detected, then the monitors build a proof of misbehaviour that can be verified by any correct node, and a *punishment* is inflicted on the misbehaving one. However, while making nodes accountable for their actions may constitute an effective incentive for selfish nodes to be cooperative, configuring accountability mechanisms for building a selfish-resilient P2P system is a challenging task.

The configuration of accountability mechanisms requires that the system designer select values for a number of parameters (e.g., number of monitors, frequency of audit, degree of punishment) that directly affect the system performance (e.g., bandwidth utilization and delay). In the literature [13], [18], [19], no indication is provided for the setting of these parameters. Indeed, the calibration of accountability mechanisms should satisfy conflicting requirements: on the one hand, it should provide the desired resilience to selfish behaviours without wrongly penalizing correct nodes, and on the other hand, it should impose minimal overhead. Resolving these constraints requires the systematic analysis of a large number of experiments/simulations, exploring 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 deviations, which is not currently supported by existing simulators (e.g., PeerSim,[1] NS-3[2]).

In this paper, we propose *RACOON*, a design and simulation framework to stimulate cooperation in a P2P system with selfish nodes, while achieving desired performance objectives. Specifically, *RACOON* provides a semi-automatic methodology, along with a set of internal and external tools, that offers an end-to-end and easy-to-use support for system designers.

The integrated framework is composed of two parts. The

[1]PeerSim: http://peersim.sourceforge.net/
[2]NS-3: https://www.nsnam.org/

design part first allows the system designer to specify the communication protocols of a desired P2P system, along with a set of performance and selfish-resilience objectives. Then, this part integrates into the specification two cooperation enforcement mechanisms, namely, an accountability system for detecting misbehaviours, and a reputation system to assign rewards and punishments. The design part proceeds automatically and extends the specification of the communication protocols with selfish deviations, assuming a common utility function for all nodes. Similarly to previous work [1], [25], the shape of this function accounts for the costs and benefits that a node derives from sharing resources with others. Finally, the design part includes an automatic tool to transform the extended specifications of communication protocols into a set of games (one per protocol), which provide the mathematical framework to reason about the behaviour of a selfish node.

The game models are the input of the second part of the framework, which automatically configures the cooperation enforcement mechanisms. To this end, *RACOON* includes a simulation module to explore the configuration space, using the game models to drive the behaviour of selfish nodes. The simulator returns a configuration file, such that, when applied, the resulting system can meet the objectives specified in the design part. Based on this result, the system designer can implement the P2P system, as well as its actual integration with the cooperation enforcement mechanisms.

We demonstrate the simplicity and effectiveness of using *RACOON* by designing two selfish-resilient P2P systems: a live streaming system and an anonymous communication system based on the onion routing protocol [16]. Simulations, as well as complementary performance evaluations involving 100 clients on a cluster of real machines, show that the live streaming system configuration chosen by *RACOON* allows correct nodes to visualize a stream of good quality in the presence of selfish nodes. Also, this configuration allows to meet a set of performance requirements set by the designer, like limiting the bandwidth overhead to a fixed threshold.

In summary, our work makes the following contributions:

- *RACOON* proposes an automatic method to generate selfish deviations from any communication protocol specified using the framework;
- *RACOON* is able to perform simulations involving game theory to reason on the dynamics of selfish behaviours;
- *RACOON* proposes an automatic and reasonably fast configuration method for an accountability and reputation mechanism in a P2P system. Such configuration takes less than 20 minutes to meet the selfish-resilient and performance objectives set by a designer on a P2P system;
- *RACOON* simulations are accurate compared to the performance of the corresponding real system, with a maximum difference confined to less than 1%.

The rest of the paper is organized as follows. Section II presents background on accountability mechanisms and their configuration. Section III presents an overview of *RACOON*, followed by a detailed description of its two essential components: the design part (Section IV), and the game-based simulation part (Section V). Section VI presents a performance evaluation of *RACOON*. Finally, Section VII presents related work, and the paper concludes in Section VIII.

II. Background

Recent successful solutions for enforcing accountability in distributed systems (e.g., [13], [17], [18], [19]) rely on secure logging and monitoring mechanisms. We focus on FullReview [13], which enables accountability in the presence of selfish nodes. FullReview applies to a set of nodes N executing a set of protocols P, defined as deterministic state machines. As part of P, each node i interacts with a set of nodes referred to as i's *partners*. When deploying FullReview [13], each node i maintains a *secure log* that is tamper-evident and append-only, in which i records all its interactions with its partners. Further, each node i is assigned a set of *monitor* nodes that periodically verify whether i sticks to the specification of P. If any deviation is detected, i's monitors inflict a *punishment* on i, which could vary from the eviction of i to the reduction of its reputation value, if the system is coupled with a reputation management system.

While enforcing accountability can constitute an effective incentive for selfish nodes to behave cooperatively, configuring the accountability mechanisms in order to discourage selfish behaviours, without excessively degrading performance, is a challenging task. Among the parameters to set are:

- The **number of monitors** associated to each node. More monitors implies more computation and communication overhead;
- The **audit period**: the period between two log audits initiated by a node's monitors;
- The **probability of audit**: the probability that a monitor audits its monitored nodes at the end of each audit period;
- The **reward/punishment function**: the way in which nodes are rewarded (resp. punished) in the case of a successful (resp. unsuccessful) audit.

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

To illustrate some of the crucial design decisions that the system designer might deal with when setting up these parameters, we performed an experiment, involving a gossip-based live streaming protocol monitored by FullReview. In this protocol, a source node disseminates a set of video chunks to a subset of nodes over an unreliable network. Periodically each node sends the video chunks it received to a set of randomly chosen partners, and asks them for any video chunks it is missing. In our experiment, we assume that the system designer aims at designing a selfish-resilient live streaming protocol in which: (1) correct nodes do not experience more than 3% jitter despite the presence of up to 50% selfish nodes; (2) correct nodes are not wrongfully expelled from the system even if the network suffers from up to 5% of message loss and

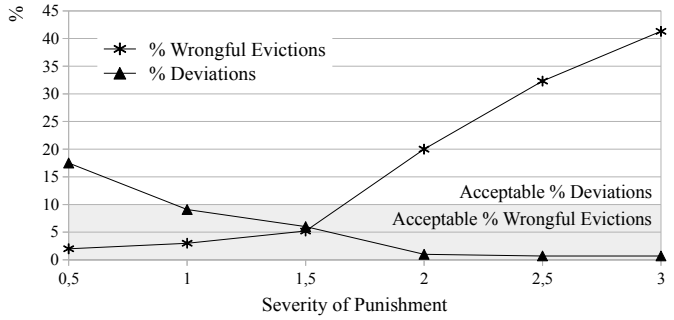

Fig. 1: Impact of the punishment values.

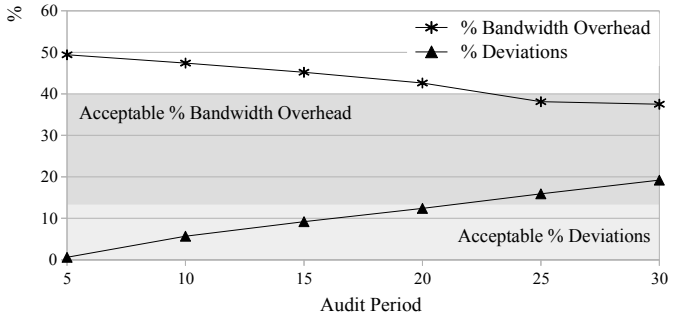

Fig. 2: Impact of the audit period.

(3) the average bandwidth consumption per node including both the video stream (which already consumes 600Kbps) and the accountability mechanisms does not exceed 1Mbps. To reach this objective, we start with the FullReview default configuration (i.e., the audit period being 10 seconds, and the probability of audit being 1) and vary the degree of punishment inflicted on nodes by the accountability mechanisms.

Fig. 1 shows the percentage of correct nodes wrongly evicted by FullReview and the percentage of selfish deviations observed in the system for various values. In our experiment, less than 10% selfish deviations for the selfish nodes translates into an experienced jitter lower than 3%. The results 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. 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 Fig. 2, show that, on the one hand, increasing the audit period decreases the overhead, because logs are requested and audits are made less often by monitors; on the other hand, the longer the audit period, the slower deviations are deterred, thereby increasing the percentage of selfish deviations. However, none of the tested values achieves a bandwidth consumption that meets the third requirement set by the designer. The designer has thus to continue the manual calibration process by testing other pairs of parameter values and running further experiments.

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

III. *RACOON* Overview

To help a system designer build a selfish-resilient P2P system, we propose the design and simulation framework *RACOON*. As depicted in Fig. 3, *RACOON* is composed of two main parts: the *design part* and the *game-based simulation part*. We give an overview of these parts here, and then provide more detail in Sections IV and V, respectively.

The input of the design part is a system specification provided by the system designer. This specification, referred to as P in Fig. 3, contains a set of deterministic state machines, describing communication protocols composing the P2P system under consideration as well as a set of performance and selfish-resilience objectives. Then, *RACOON* integrates into the provided specification accountability and reputation mechanisms configured with a default configuration (Step (1) in the figure). These mechanisms are used to detect selfish deviations and to reward/punish nodes in case of positive/negative audits, respectively. *RACOON* then automatically extends the state machines with new transitions that represent selfish deviations, according to a selfishness model provided by *RACOON* (Step (2) in the figure). Finally, *RACOON* transforms the extended state machines into games (Step (3) in the figure).

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

IV. *RACOON* Design Part

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

A. *RACOON Specification Model*

RACOON includes a specification model to assist the system designer in correctly describing all the information required by the framework. This model allows describing the *functional specification* of the P2P system, the *list of objectives* that the system must fulfill, and the *initial configuration* of the

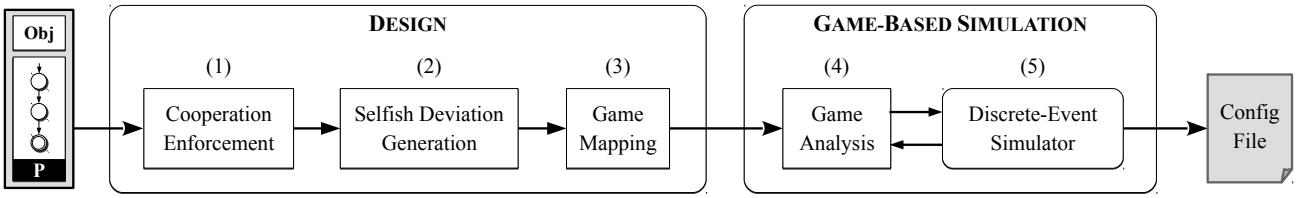

Fig. 3: *RACOON* Overview.

cooperation enforcement mechanisms. To illustrate our model and in the rest of this paper, we use the communication protocol *R&R* (*Request & Response*) shown in Fig. 4. In this protocol, a node r_0 sends a request message g_0 to a group of nodes collectively named R_1 (the capital letter denotes a set of unique nodes), and upon receiving g_0, each node in R_1 replies with a response message g_1.

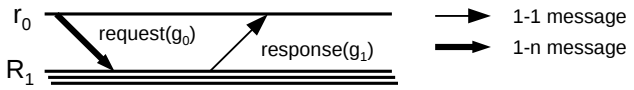

Fig. 4: The *R&R* protocol between nodes r_0 and R_1.

1) Functional Specification: The functional specification P of a P2P system is provided by means of communication protocols: a set of rules of interaction that define what actions (i.e., methods) each node can take at each step. *RACOON* describes each communication protocol in P as a deterministic finite state machine, called a *Protocol Automaton*. A Protocol Automaton is a tuple $\langle R, S, s_0, F, T, Meth, G, C, V \rangle$, where:

- $R \neq \emptyset$, is the set of *Roles* that a node may undertake in the protocol. A role can represent either a node or a group of unique nodes. For example, in the protocol *R&R*, there are two roles: r_0 and R_1, where R_1 corresponds to the set of recipients of the message g_0. The cardinality of a role denotes the number of nodes that it represents. More formally, a role $r \in R$ is a tuple $\langle id, cardinality \rangle$;
- $S \neq \emptyset$, is the set of *States* that the system goes through when implementing the protocol. Some special states are: the start state s_0, and the non-empty set of final states $F \subseteq S$, in which the protocol terminates.
- $T \neq \emptyset$, is the set of state *Transitions*. A transition $t \in T$ is a tuple $\langle id, state1, state2, method \rangle$, where $state1$ and $state2 \in S$ are the source and target states, and $method \in Meth$ is the method that triggers t;
- $Meth \neq \emptyset$, is the set of *Methods*. There are two types of methods: a communication method represents the delivery of a message from one role to another; a computation method refers to local computations. For instance, in Fig. 4, *request* is a communication method that sends a message g_0 to R_1. Formally, a method $m \in Meth$ is a tuple $\langle id, invokerRole, message \rangle$, where $message$ is defined only for communication methods;
- G is the finite set of *Messages* conveyed by communication methods. A message $g \in G$ is a tuple $\langle id, recipientRole, content \rangle$, where $content$ is the content carried by the message;
- C is the set of *Contents* delivered by the messages. A content $c \in C$ is either a single data-unit (e.g., a binary file), or a collection (e.g., a list of integers). The specification model defines c as the tuple $\langle id, ctype, size, collection \rangle$, where $ctype$ is the data type,[3] $size$ is the memory size of a single data-unit in c (given in bytes), and $collection$ is a boolean value (i.e., true if c is a set of data-units);
- V is the set of content *Constraints*. A constraint prescribes a relationship that has to be fulfilled by two contents. A constraint $v \in V$ is a tuple $\langle id, c1, c2, vtype \rangle$, where the two contents $c1$ and $c2 \in C$ are subject to the relationship defined in $vtype$. Specifically, $vtype$ can identify either an ordering relation (e.g., $=, <, >$) or a set operation (e.g., subset, equal).

A Protocol Automaton can be represented by means of a state diagram. Fig. 5 shows the state diagram of the *R&R* protocol. The label on a transition provides information about the method that triggers the transition, and about the message that might be sent. For example, the label between states s_1 and s_2, indicates that role R_1 invokes the communication method $response$, which conveys the message g_1 to role r_0.

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

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

3) Cooperation Enforcement Mechanisms: *RACOON* automatically integrates a set of accountability and reputation mechanisms into the system specification. Specifically, in addition to the P protocols described by the designer, *RACOON* applies to each node the FullReview accountability protocols, and the reputation system used by *RACOON*. We refer to the accountability and reputation protocols as the M protocol. FullReview contains a set of verification protocols including an audit protocol and an evidence-transfer protocol. These protocols are periodically executed by nodes to monitor their partners and exchange information about their partners' status.

[3]Defined by the XML Schema type system.

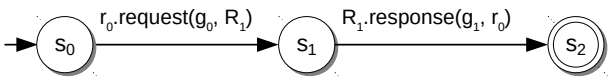

Fig. 5: The Protocol Automaton of the *R&R* protocol.

The reputation system, on the other hand, only contains computational methods that are triggered by monitors to update the reputation value of a node after an audit. The state machines corresponding to both kinds of protocols are described in detail in the companion technical report [23].

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

B. Generating Selfish Deviations

The utility that a node obtains from participating in a P2P system is given by: (1) the benefits, i.e., the positive value obtained by consuming resources; and (2) the costs, i.e., the negative value obtained due to the cost of sharing resources. The utility function is a mathematical model that evaluates the choices facing a node in terms of these quantities.

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

In a communication protocol, the bandwidth consumption depends on the number and size of the messages that are exchanged between nodes. *RACOON* automatically generates three types of communication-related deviations: (1) *timeout* deviation: the node does not perform the prescribed method within the time limit; (2) *subset* deviation: the node sends a subset of the correct message content; and (3) *multicast* deviation: the node sends a message to a subset of the legitimate recipients. Alg. 1 shows the pseudo-code for generating the deviations listed above. The algorithm, called *CDG*, takes a PA as input, and extends it with new elements (states, transitions, roles, etc.) representing deviations. Hereafter, we describe the pseudo-code in more detail.

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

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

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

```
Data: A Protocol Automaton PA.
Algorithm CDG(PA)
1   foreach non-final state s do
2       r := s.activeRole
3       CreateTimeoutDev(s)
4       OT := outgoing transitions of s
5       foreach transition t ∈ OT s.t. t.method.message ≠ null do
6           c := t.method.message.content
7           if c.collection then
8               CreateSubsetDev(t, c)
9           if t.state2.activeRole.cardinality > 1 then
10              CreateMulticastDev(t)
Procedure GenTimeoutDev(s)
11  s' := ⟨new_sId, s.activeRole, s.audit⟩
12  m' := ⟨"timeout", s.activeRole, 0⟩
13  t' := ⟨new_tId, s, s', m'⟩
14  add s', m', and t' to PA
Procedure GenSubsetDev(t,c)
15  s' := ⟨new_sId, t.state2.activeRole, t.state2.audit⟩
16  c' := ⟨new_cId, c.ctype, c.size, false⟩
17  updateConstraints(c')
18  m := t.method; g := m.message
19  g' := ⟨new_gId, g.recipientRole, c'⟩'
20  m' := ⟨new_mId, m.invokerRole, g'⟩
21  t' := ⟨new_tId, t.state1, s', m'⟩
22  add s', c', g', m', and t' to PA
23  copyOutgoingTransitions(s',t.state1)
Procedure GenMulticastDev(t)
24  r' := ⟨new_rId, 1⟩
25  s' := ⟨new_sId, u', t.state2.audit⟩
26  m := t.method; g := m.message
27  g' := ⟨new_gId, u', g.content⟩'
28  m' := ⟨new_mId, m.invokerRole, g'⟩
29  t' := ⟨new_tId, t.state1, s', m'⟩
30  add r', s', g', m', and t' to PA
31  copyOutgoingTransitions(s',t.state1)
```

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

Fig. 6 shows the result of executing the *CDG* algorithm on the Protocol Automaton PA of Fig. 5. In the correct execution of PA, in the initial state s_0, the role r_0 sends a message (g_0) to R_1. However, if r_0 is played by a selfish node, he may also (from top to bottom in Fig. 6): timeout the protocol, send a message with a smaller payload (g_0'), or send g_0 to a subset of recipients (R_1'').

C. Game Mapping

The Protocol Automaton extended in the previous step describes possible behaviours of selfish nodes, but gives no indication of the likelihood of any particular behaviour. *RACOON* uses Game Theory [27] to address this issue. Specifically, the framework provides an automatic tool to translate the PA into a game, referred to as the *Protocol Game* PG. This game provides the necessary structure to support the next step of

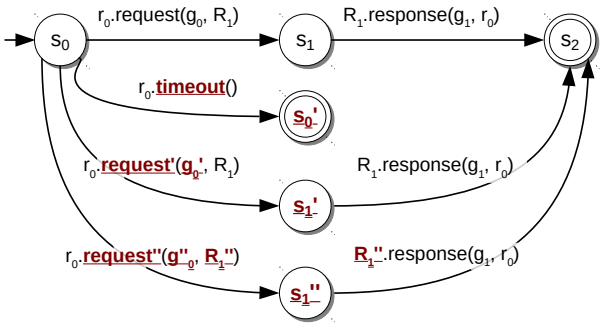

Fig. 6: The Protocol Automaton with selfish deviations.

RACOON, i.e., the game-based simulations. The process by which this game is generated is described below.

Game type and Players. The Protocol Automaton PA is modelled as a non-cooperative sequential game among self-interested players [27]. Each player is assigned to exactly one role $r \in R$. The game is non-cooperative because each player competes against the others for maximizing his own utility. At the same time, the game is sequential because players have a specific order of actions to follow, as specified by PA.

A sequential game is usually represented as a tree, also called *extensive form representation* [27]. A node in PG is derived from a state in PA, and is labelled with the player who has to move. Each leaf in PG translates to a final state in PA, while each edge in PG corresponds to a transition in PA. Edges in PG are labelled with *action*s, which correspond to methods m defined in PA. The sequence of actions that a player p_i might choose in a play constitutes his *strategy*. A *strategy profile* is a vector specifying a strategy for every player. Fig. 7 shows the extensive form representation of the game derived from the *R&R* protocol.

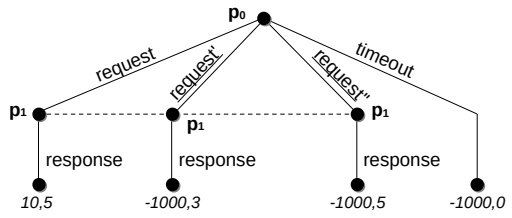

Fig. 7: The Protocol Game derived from the *R&R* protocol.

Utility function and payoffs. The utility function of a player assigns a *payoff* to every possible game outcome (i.e., the result obtained in correspondence to a strategy profile). The utility function implemented by *RACOON* has two terms: the cost k of sharing resources, and the incentives provided by the cooperation enforcement mechanisms.[4] Because in this paper we only consider communication-related selfishness, k is calculated as the bandwidth necessary to implement a certain player's strategy. Furthermore, we calculate incentives as a function ρ of the reputation variation ΔR determined by an audit. Note that ΔR also depends on the current reputation of the node, for the definition of the reputation update function.

[4]We assume that all players obtain the same benefit from playing the game.

Formally, we write the utility u_i for player p_i when implementing his strategy σ_i as follows:

$$u_i(\sigma_i) = -k(\sigma_i) + \rho(\Delta R)$$

The definition of $\rho(\Delta R)$ takes into account the following considerations. The selfish nature of players encourages them to deviate from the protocol as much as possible (to save resources), as long as this can be accomplished without causing their eviction from the system. Therefore, if a player's reputation increases at a point in time ($\Delta R > 0$), then he is more likely to deviate in the future, as he will be further from the eviction threshold than before. In this case, the function ρ assumes a positive value, which corresponds to a payoff increment. Likewise, on the contrary, ρ assumes a negative value if a player's reputation decreases ($\Delta R < 0$). A particular case is if the reputation value goes below the eviction threshold, in which case the player is evicted from the system. This negative event by itself corresponds to a cost, conventionally set at a very high nominal value.

In Fig. 7, the pairs of numbers (one for each player) below each leaf are the payoffs, which express the utility of playing the strategy that terminates in that leaf. In the following, we briefly discuss how some of the payoff values in Fig. 7 are obtained. Let us consider the following setting of the *R&R* protocol. The role r_0 has to send a request message g_0 to role R_1, with R_1 having cardinality 5. The request message conveys a content c_0: a list of $n = 10$ boolean values, each of size 1 byte. Thus, for instance, the communication cost k for player p_0 when playing the strategy σ_1 (i.e., sending the correct request message) is:

$$k(\sigma_1) = R_1.cardinality \times (c_0.size \times n) = 50$$

Let us assume that the execution of M returns: $\Delta R = 1$ when p_0 chooses the correct strategy σ_1, and $\Delta R = -1$ otherwise (i.e., selfish deviations). Fig. 7 illustrates a possible payoff structure that can be obtained from the above setting, in which $\rho(1)$ determines a payoff increment of 60, and $\rho(-1)$ determines a payoff decrement of -1000.

V. *RACOON* Game-Based Simulation

We now describe how *RACOON* computes a configuration for the accountability and reputation mechanisms that ensures the enforcement of the system objectives set by the system designer. This is realized in two interleaving phases: simulation and game analysis.

The simulation phase explores the space of the parameters presented in Section II, searching for a setting that reaches the objectives defined by the system designer. The automatic approach adopted by *RACOON* is to simulate the target system in different regions of the parameter space, using game-theoretic analysis to drive the behaviour of selfish nodes. The exploration ends when a configuration satisfying the system designer's objectives is found.

The *RACOON* framework includes a discrete-event simulator for P2P overlay networks, which uses as input the specifi-

cation P of the system (extended with selfish deviations), and the objective requirements to achieve. This simulator supports a cycle-based simulation model. Specifically, at each cycle, every node executes P in turn. Correct nodes will never deviate from the correct implementation of P, while the behaviour of selfish nodes can change from one cycle to another according to the action that maximizes their utility.

To simulate the behaviour of selfish nodes, the *RACOON* simulator (*R-sim*) interacts with the game-theoretic tool (*GT-tool*) included in the framework. At each simulation cycle, and for each Protocol Automaton $PA \in P$, *R-sim* and *GT-tool* interact as follows: (1) *GT-tool* translates PA into a Protocol Game PG, and configures it with the current reputation of the interacting nodes (provided by *R-sim*); (2) *GT-tool* conducts a game analysis to identify the best strategy of each node; (3) *GT-tool* returns the obtained strategies to *R-sim*, so that they can be executed in the simulation.

The game-theoretic analysis performed at step (2) determines the possible steady states of PG, which are the equilibrium points. *RACOON* uses the Sequential Equilibrium (SE) solution [27], a refinement of the Nash Equilibrium for sequential game with imperfect information. To find the SE of PG, *GT-tool* uses Gambit,[5] an open-source library of tools for solving non-cooperative games. Specifically, Gambit implements the algorithm by Koller, Megiddo and von Stengel [22], using linear programming. In the Protocol Game of Fig. 7, the SE found by Gambit is the strategy profile *(request, response)*, which indicates that the expected behaviour of players p_0 and p_1 is to execute the methods named *request* and *response*, respectively. The *RACOON* simulator uses this information to simulate simulates the players' behaviour accordingly. For example, if at a given turn the equilibrium strategy of a selfish node is to perform a timeout deviation, then the simulator will skip any execution of the communication protocol of that node in that turn.

A single simulation allows verifying the selfish-resilience and performance guarantees offered by a given configuration of the accountability and reputation mechanisms. To find a configuration that meets all the objectives set by the designer, *RACOON* explores the space of configuration parameters using a greedy algorithm optimized with a set of heuristics. For instance, if when simulating a given configuration the overhead is already above the threshold fixed by the designer, *RACOON* will not increase the number of monitors, the probability of audit or the audit period in the next configuration to be explored as this would further increase the overhead. We show in the following section that thanks to our heuristics, *RACOON* manages to converge in a reasonable time (18 minutes on average). We plan to investigate more optimized exploration algorithms (e.g., simulated annealing) in future work.

The outcome of the simulator is a configuration of the cooperation enforcement mechanisms that reaches all the objectives of the system designer. If no configuration is found (which may happen if the specified objectives are contradictory), the simulator asks the designer to relax some of his objectives. Once a configuration is found, the system designer proceeds with the implementation of the P2P system, in which he integrates the FullReview API calls configured using the configuration file provided by *RACOON*. In our future work, we further aim at automatizing the implementation step, by extending *RACOON* with a new module for the generation of executable code. There already exists good solutions for generating code (e.g., MACE [21]), which we plan to integrate in the future version of our framework.

[5]Gambit: http://sourceforge.net/projects/gambit/

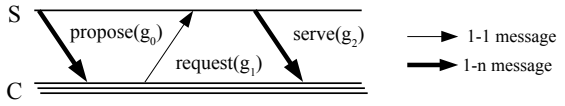

Fig. 8: The sequence diagram of the $3P$ gossip protocol.

VI. Evaluation

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

A. RACOON Design Effort

We show in this section, the effort necessary for the system designer to describe a live streaming system using *RACOON*. The functional specification of this system, briefly introduced in Section II, defines the three-phase ($3P$) gossip-based protocol studied in [17] and depicted in Fig. 8. This protocol involves two roles: the supplier S proposes the set of chunks it has received to a set of consumers C, which in turn request any chunks they need. The protocol ends when S sends to C the requested chunks. This protocol is executed periodically by the set of nodes participating in a live streaming session. Each chunk is associated with an expiration time (i.e., the *play-out delay*). A node can only playback the chunks that have not yet expired. Fig. 9 illustrates the Protocol Automaton of the $3P$ gossip protocol. The full *RACOON* specification of this protocol is found in the companion technical report [23].

In our experiment, we assume that the system designer wants to set the objectives introduced in Section II:

Obj.1 A deviation rate lower than 10%;

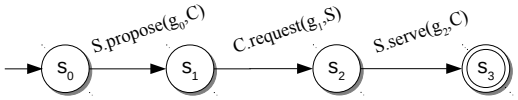

Fig. 9: The Protocol Automaton of the $3P$ gossip protocol.

Parameter	Simulation	Experiment
Network size (nodes)	1000	100
Broadcast bandwidth (Kbps)	600	600
Partner set size (nodes)	7	7
Play-out delay (rounds)	6	6
Bandwidth capacity (Kbps)	1000	1000

TABLE I: Simulation and Real Deployment Parameters.

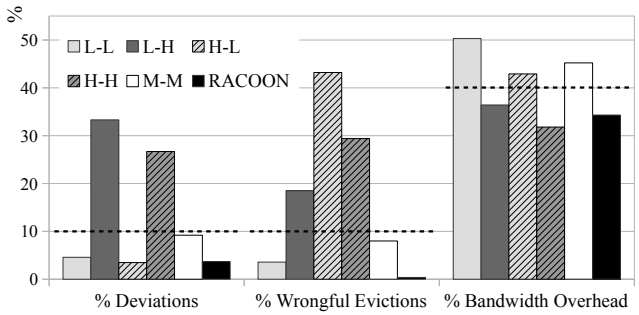

Fig. 10: *RACOON* vs FullReview Configurations.

Obj.2 A bandwidth overhead lower than 40%;

Obj.3 A wrongful eviction rate lower than 10%.

Overall, the XML specification of this protocol contains 48 lines. In addition to writing this specification, the system designer has to develop a new module for the *RACOON* simulator, which implements the P2P live streaming system model. The implementation classes of this module contain 500 lines of code (LOC). This includes the implementation of the $3P$ gossip protocol and custom monitors to measure live-streaming metrics (e.g., the number of chunks transmitted/received). The overall simulation code of this application further uses 3200 LOC provided by *RACOON* libraries (e.g., the simulation engine, the exploration algorithm, FullReview, and the reputation system).

B. Meeting System Objectives Using RACOON Simulations

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

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

Fig. 10 shows the simulation results of the six configurations. The *RACOON* configuration is the only one that fulfills all the design objectives, which are depicted as horizontal dotted lines in the figure. Furthermore, this configuration provides up to 33% fewer deviations, 42% fewer wrongful evictions and 17% lower overhead than the others.

	L-L	L-H	H-L	H-H	M-M	*RACOON*
Audit Period	5	5	30	30	15	5
Punishment	0.5	3.0	0.5	3.0	1.5	1.0
Prob. of Audit	1.0	1.0	1.0	1.0	1.0	0.5

TABLE II: FullReview Configurations

C. 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 II. Then, we deployed the prototype on a cluster of real machines.[6] Specifically, we run 100 clients on 10 eight-core physical machines. Each machine is clocked at 2.5GHz with 32GB of RAM, and is interconnected with the others via a Gigabit switch. We performed our experiment in this environment in order to measure the impact of selfish nodes on the observed jitter without the risk of fluctuating networking conditions. Otherwise, it would be difficult to assess whether a deteriorated stream quality comes from selfish nodes or from the transient network.

The third column of Table I describes our experimental settings. Note that the only difference with the simulation settings is the number of nodes in the network. In this experiment, we measure the jitter experienced by correct nodes as a function of the fraction of selfish nodes in the system.

Fig. 11 presents the results of our evaluation. This figure contains a curve showing the impact of selfish nodes on traditional Gossip (i.e., without any accountability mechanisms) as well as the two curves for the system designed using *RACOON*. The "SIM - *RACOON*" curve is obtained using *RACOON* simulations, whereas the "G5K - *RACOON*" curve is obtained using the real deployment. From this figure, we observe that without accountability mechanisms, correct nodes experience 10% jitter with only 10% of nodes behaving selfishly, which prevents them from watching the video stream. Further this figure shows that the simulated curve and the real one, overlap up to the inclusion of 50% of selfish nodes in the system. Above this value the curves still exhibit a comparable shape. Finally, this figure shows that despite 90% of nodes becoming selfish, the configuration found by *RACOON* allows

[6]Grid'5000: http://www.grid5000.fr

correct nodes to watch the video stream with a jitter lower than 3%, which reflects the effectiveness of *RACOON*.

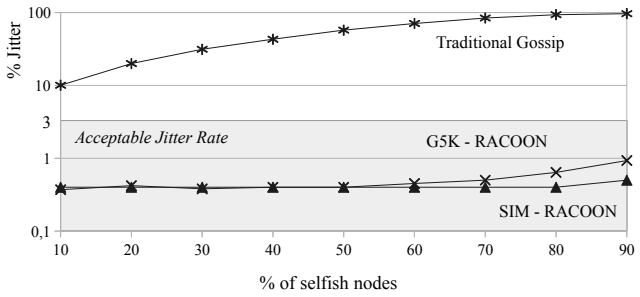

Fig. 11: Simulation vs real deployment (logarithmic scale).

D. RACOON Execution Time

To evaluate the time necessary for *RACOON* to find a satisfactory configuration, we performed the following experiment. First, we defined a set of 30 different scenarios in the live streaming application. Each scenario is a unique combination of the following elements: system objectives, simulation settings (e.g., number of nodes, bandwidth capacity, play-out delay), percentage of selfish nodes in the system, and message loss rate. Second, we measured the number of configurations that *RACOON* explores in each case before finding a satisfactory solution. Finally, we average these numbers over the total number of scenarios that have been considered. The results show that, for each scenario, an average of 26 configurations are explored by *RACOON* before finding a satisfactory one. Considering that each configuration corresponds to one executed simulation, and that each simulation lasts approximately 42 seconds,[7] 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.

E. RACOON Expressiveness

We conclude this section by illustrating the generality of our framework. To this end, we use *RACOON* to design an anonymous communication protocol based on the Onion Routing protocol [16]. In Onion Routing, when a source node wants to send a message to a destination node, it builds a circuit of *relay* nodes. The source node changes the circuit periodically, and relays can support many circuits simultaneously. To achieve anonymity, the source uses the public key of each relay along the circuit to successively encrypt the message, which constitutes an *onion*. Fig.12a illustrates the protocol enabling the forwarding of onions. In this protocol, each relay R: (i) receives onion messages from their predecessor in the circuit (PR); (ii) decrypts the external layer of each onion; (iii) forwards the resulting onions to their respective successor NR.

To design a selfish-resilient onion forwarding protocol using *RACOON*, the system designer follows the same steps we have seen to design the live streaming system. First, he provides the *RACOON* specification of the system. Due to space limitations, we provide the full specification in the companion technical report [23]. This specification describes three roles, which are the relay R, and its previous (PR) and next (NR) hops. Note that PR and NR have cardinality $n \geq 1$, because they represent a set of relay nodes. The Protocol Automaton of the forwarding protocol can be modelled as in Fig. 12b. The protocol has a communication method and a computation method. The communication method *relay* sends messages that carry the single onion to forward. The second method, called *unwrap*, is a decryption operation (i.e., local computation). A selfish relay R that aims at saving bandwidth, strategically drops onions that are not intended for it.

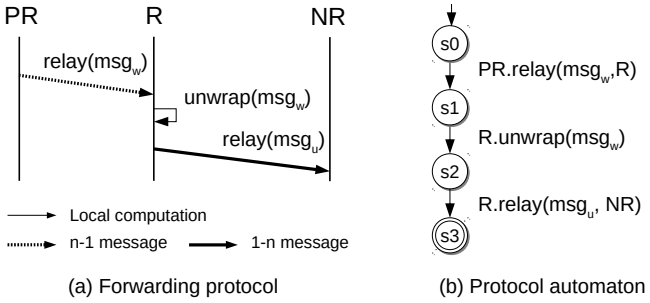

Fig. 12: Onion Forwarding Protocol.

Designing this system using *RACOON* only required writing 44 lines of XML specification and 350 LOC for the implementation of the simulation module. Notice that the system designer can reuse more than 42% of the implementation code written for the P2P live streaming module, and only requires to implement the methods of the Protocol Automaton depicted in Fig. 12. The overall simulation code further uses the same 3200 LOC used in the P2P live streaming protocol and provided by *RACOON* libraries. Simulations of this use case realized using *RACOON* can be found in [23].

VII. Related work

Game Theoretic approaches: Much work on the potential of Game Theory as tool for system designers has been carried out in the context of content-disseminating applications [12], [24], [25], wireless networking [10], [28], exchange protocols [5], and anonymity and privacy mechanisms [2], [15]. The objective of these approaches is to make cooperation the best choice for all nodes, i.e. a Nash Equilibrium. Although promising results have been achieved, most of the reported solutions are tailored to a specific system or are too difficult to adapt to a changing environment. A notable example is the BAR Model of Aiyer et al. [1], which is a general architecture for building cooperative distributed systems that are robust to selfish and Byzantine nodes. However, the design of a BAR-tolerant protocol is a complex task that has to be performed manually [1], [3], [25]. Moreover, this approach suffers from poor maintainability and reusability: every change in the system parameters requires a full revision of the solution.

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

Non-Game theoretic approaches: Incentive-based mechanisms discourage selfish behaviors by making cooperation more attractive for selfish nodes. For example, in credit-based systems [6], [30], cooperating nodes gain credits, which are the virtual currency to spend for using the system. These approaches require a trusted central authority, which may not be consistent with the system under design (e.g., in ad hoc mobile networks). On the contrary, the *RACOON* framework adopts a generic distributed solution, i.e., the cooperation enforcement mechanisms are deployed on each node. Another popular form of incentive mechanism is reputation [26], which is, in the context of our work, a measure of a node's cooperation. However, a selfish node can cheat the system by misreporting reputation information. In the reputation system included in *RACOON* such cheating become detectable and punishable, because the reputation value depends on the (provable) auditing results of an accountability mechanism.

Accountability approaches such as [13], [17], [18], [19] provide another strategy for dealing with selfish nodes. An accountability system discourages deviations by exposing a misbehaving node to the risk of punishment. Despite the high generality and the proven effectiveness of these approaches, their application incurs a non-negligible cost to the system (e.g., message overhead, intensive use of cryptography). Moreover, they often rely on strong assumptions (e.g., the presence of a quorum of correct nodes, no message loss [13], [19]). *RACOON* proposes a semi-automatic approach to configure an accountability mechanism (i.e., FullReview [13]) in such a way as to achieve good performance and selfish-resilience.

Finally, several frameworks [29] and Domain-Specific Languages [4], [21] have been proposed to ease the task of designing and maintaining secure distributed system. Although these solutions yield good results in terms of system performance and designer effort, none of them address the specific threat of selfish deviations in cooperative distributed systems.

VIII. Conclusion

In this paper we presented *RACOON*, a novel framework for designing and configuring cooperative P2P systems that are resilient to selfish nodes. *RACOON* relies on accountability and reputation mechanisms to enforce cooperation among selfish nodes. Using a combination of simulation and Game Theory, RACOON automatically configures these mechanisms in a way that meets a set of selfish-resilience and performance objectives specified by the system designer. We illustrated the benefits of using *RACOON* by designing a P2P live streaming system and an anonymous communication system. The evaluation of the P2P live streaming system performed using both simulations and a real deployment shows that this system achieves selfish-resilience and high performance.

Our future work include the integration of a domain specific language (e.g., MACE [21]) into *RACOON* to automatically generate executable code. This could be done by defining transformation rules between the *RACOON* specification model and the MACE language.

References

[1] A.S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J-P. Martin, and C. Porth. "BAR fault tolerance for cooperative services." In *SOSP*, 2005.

[2] S. Ben Mokhtar *et al.*. "RAC: a freerider-resilient, scalable, anonymous communication protocol." In *ICDCS*, 2013.

[3] S. Ben Mokhtar *et al.* "FireSpam: Spam resilient gossiping in the BAR model." In *Proc. of SRDS*. IEEE Computer Society, 2010.

[4] M. Biely *et al.*. "Distal: A framework for implementing fault-tolerant distributed algorithms." In *DSN*, 2013.

[5] L. Buttyán and J-P. Hubaux. "Rational exchange-a formal model based on game theory." In *Electronic Commerce*. Springer, 2001.

[6] L. Buttyán and J-P. Hubaux. "Stimulating cooperation in self-organizing mobile ad hoc networks." *Mobile Networks and Applications*, 8(5), 2003.

[7] Z. Shen *et al.* "Peer-to-peer media streaming: Insights and new developments." *Proc. of the IEEE*, 99(12), 2011.

[8] T. Ban, S. Guo, Z. Zhang, R. Ando, and Y. Kadobayashi. "Practical network traffic analysis in P2P environment." In *IWCMC*, 2011.

[9] Cisco Systems. "Cisco Visual Networking Index: Forecast and Methodology." A Cisco White Paper, 2015.

[10] M. Cagalj, S. Ganeriwal, I. Aad, and J-P. Hubaux. "On selfish behavior in CSMA/CA networks." In *INFOCOM*, 2005.

[11] I. Cunha *et al.* "Can peer-to-peer live streaming systems coexist with free riders?" In *P2P*, 2013.

[12] J. Decouchant, S. Ben Mokhtar, and V. Quéma. "AcTinG: Accurate Freerider Tracking in Gossip." In *SRDS*, 2014.

[13] A. Diarra, S. Ben Mokhtar, P-L. Aublin, and V. Quéma. "FullReview: Practical accountability in presence of selfish nodes." In *Proc. of SRDS*, 2014.

[14] M. Feldman *et al.* "Free-riding and whitewashing in peer-to-peer systems." *IEEE J. Sel. Areas Commun.*, 24(5), 2006.

[15] J. Freudiger *et al.* "On non-cooperative location privacy: a game-theoretic analysis." In *CCS*, 2009.

[16] D. Goldschlag, M. Reed, and P. Syverson. "Onion routing." *Commun. of the ACM*, 42(2):39–41, 1999.

[17] R. Guerraoui *et al.*"LiFTinG: Lightweight freerider-tracking in gossip." In *Middleware*, 2010.

[18] A. Haeberlen, P. Aditya, R. Rodrigues, and P. Druschel. "Accountable virtual machines." In *OSDI*, 2010.

[19] A. Haeberlen, P. Kouznetsov, and P. Druschel. "PeerReview: Practical accountability for distributed systems." *SOSP*, 2007.

[20] D. Hughes, G. Coulson, and J. Walkerdine. "Free riding on Gnutella revisited: the bell tolls?" *Distributed Systems Online, IEEE*, 6(6), 2005.

[21] C-E. Killian, J-W. Anderson, R. Braud, R. Jhala, and A. Vahdat. "Mace: Language support for building distributed systems." In *PLDI*, 2007.

[22] D. Koller, N. Megiddo, and B. von Stengel. "Fast algorithms for finding randomized strategies in game trees." In *STOC*, 1994.

[23] G. Lena Cota *et al.* "Racoon: Technical report." Available: https://sites.google.com/site/soniabm/.

[24] H.C. Li *et al.*"FlightPath: Obedience vs. choice in cooperative services." In *OSDI*, 2008.

[25] H.C. Li, A. Clement, E-L. Wong, J. Napper, I. Roy, L. Alvisi, and M. Dahlin. "BAR gossip." In *OSDI*, 2006.

[26] S. Marti and H. Garcia-Molina. "Taxonomy of trust: Categorizing p2p reputation systems." *Computer Networks*, 50(4), 2006.

[27] M.J. Osborne *et al.* "A course in game theory." *MIT press*, 1994.

[28] V. Srivastava *et al.* "Using game theory to analyze wireless ad hoc networks." *IEEE Commun. Surveys and Tutorials*, 7(1-4), 2005.

[29] P. Urbán, X. Défago, and A. Schiper. "Neko: A single environment to simulate and prototype distributed algorithms." In *ICOIN*, 2001.

[30] S. Zhong *et al.* "Sprite: A simple, cheat-proof, credit-based system for mobile ad-hoc networks." In *INFOCOM*, 2003.

Acknowledgment

The presented work was developed within the EEXCESS project funded by the EU FP7 (Grant n. 600601). This work was also partly supported by the program CMIRA2014 Coopera (Grant n. 14.007051) of the Region Rhone-Alpes, France, and by the project Vinci of the Univ. Franco-Italienne (Grant n. C4-9).