This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2706286, IEEE Transactions on Dependable and Secure Computing

RACOON++: A Semi-Automatic Framework for the Selfishness-aware Design of Cooperative Systems

Guido Lena Cota[1], Sonia Ben Mokhtar[2], Gabriele Gianini[1], Ernesto Damiani[1,4], Julia Lawall[3], Gilles Muller[3], and Lionel Brunie[2]

[1]Università degli Studi di Milano, [2]LIRIS-CNRS-INSA Lyon, [3]Sorbonne Universités, Inria, CNRS, UPMC, LIP6
[4]EBTIC/Khalifa University, Abu Dhabi, UAE

A challenge in designing cooperative distributed systems is to develop feasible and cost-effective mechanisms to foster cooperation among selfish nodes, i.e., nodes that strategically deviate from the intended specification to increase their individual utility. Finding a satisfactory solution to this challenge may be complicated by the intrinsic characteristics of each system, as well as by the particular objectives set by the system designer. Our previous work addressed this challenge by proposing RACOON, a general and semi-automatic framework for designing selfishness-resilient cooperative systems. RACOON relies on classical game theory and a custom built simulator to predict the impact of a fixed set of selfish behaviours on the designer's objectives. In this paper, we present RACOON++, which extends the previous framework with a declarative model for defining the utility function and the static behaviour of selfish nodes, along with a new model for reasoning on the dynamic interactions of nodes, based on evolutionary game theory. We illustrate the benefits of using RACOON++ by designing three cooperative systems: a peer-to-peer live streaming system, a load balancing protocol, and an anonymous communication system. Extensive experimental results using the state-of-the-art PeerSim simulator verify that the systems designed using RACOON++ achieve both selfishness-resilience and high performance.

I. Introduction

In recent years, the importance of cooperative systems such as peer-to-peer (P2P) networks and collaborative computing has rapidly grown, driven by multiple factors. First, the ever-increasing demand for video content [1] poses serious challenges to the operational and economic sustainability of traditional content delivery networks [2], paving the way for more scalable, robust and cost-effective P2P-assisted solutions [3]. Second, cooperative systems are the key enablers of new and emerging technologies, including the blockchain ecosystem [4] and the Internet of Things [5]. Finally, the decentralised nature of cooperative systems can address the increasing privacy concerns of their users [6], by avoiding control and potential misuse of sensitive data by a centralised server.

Crucial to the success of cooperative systems is that nodes are willing to collaborate with each other by sharing part of their resources — e.g., network bandwidth, storage space, CPU time. However, in practice [7]–[9], real systems often suffer from selfish nodes that strategically withdraw from cooperation to satisfy their individual interests at the expense of the system reliability and efficiency. In fact, several studies have shown that 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 [13], [23], [25]. For example, Guerraoui et al. [14] observed experimentally that if 25% of nodes 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.

Different solutions have been proposed to deal with selfishness in cooperative systems [22]–[28]. Most of these solutions rely on Game Theory (GT), a theoretical framework to model and study selfish behaviours [31]. The typical approach to design selfishness-resilient systems using GT requires first creating an analytical model of the system (the game) and then proving mathematically that the cooperative behaviour is the best strategy for selfish nodes (a Nash Equilibrium), with respect to a known utility function. However, carrying out this process is complex, error-prone, and time-consuming [30].

Detecting and punishing selfish behaviours at runtime is an alternative, more practical approach. Diarra et al. [13] showed that making nodes accountable for their actions can be a strong incentive for selfish nodes to cooperate. In an accountable system, each node maintains a *secure log* to record its interactions with other nodes. Also, each node is associated with a set of *witness* nodes that periodically check whether the log entries correspond to a correct execution of the system. 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. Although accountability mechanisms have been successfully applied to cooperative systems [12]–[14], the additional work required at each node (e.g., cryptographic operations, log auditing) can significantly increase computation, bandwidth, and storage requirements. Moreover, the fine tuning of these mechanisms for building a selfishness-resilient and cost-effective cooperative system could be a challenging task [19].

Configuring accountability mechanisms requires that a system designer select values for a number of parameters (e.g., number of witnesses, audit frequency) that directly affect the system performance (e.g., bandwidth usage, delay). In the literature [12]–[14], no indication is provided for the setting of these parameters, leaving entirely to designers to find a configuration that achieves the desired level of resilience to selfish behaviours while imposing minimal overhead. Finding this critical trade-off involves the systematic evaluation of a large number of experiments, to investigate the impact of the value of each parameter on the system performance. Moreover,

Corresponding author: G. Lena Cota (email: guido.lena@unimi.it).

such experiments require the ability to create and inject selfish behaviours, which is not supported by state-of-the-art experimental environments, such as Splay [41], NS-3 [39], and PeerSim [10].

To address the design challenges discussed above, our previous work [19] proposed *RACOON*, a *general* framework for designing efficient P2P systems resilient to selfish nodes in a *semi-automatic* manner. To begin, the designer provides the functional specification of the system (i.e., communication protocols) and a set of performance objectives. *RACOON* uses this information to mostly automate the following steps: (i) enforce practical mechanisms to foster cooperation (i.e., distributed accountability and reputation mechanisms), (ii) identify possible selfish deviations from the functional specification; (iii) develop a behavioural model of the system participants as a non-cooperative game [31], to predict the strategic choices of selfish nodes; and (iv) tune the accountability and reputation parameters to meet the designer's objectives, using GT-based simulations. Each step is carried out by a distinct module of the framework, which can be replaced or extended with a new definition (e.g., different models for selfishness). *RACOON* results in a complete design of the system, which includes finely tuned mechanisms to meet selfishness-resilience and performance objectives. This output serves as a reference to developers for the eventual implementation of the system.

In this paper, we describe *RACOON++*, which extends the previous version of our framework by addressing a number of limitations and introducing new features. First, we provide the designers 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 shapes the utility function of a node by assigning costs and benefits to specific actions of the communication protocols, and parametrises some aspects of selfish behaviours (*who* deviates, from *which action*, with *what type* of deviation). Second, we model the behaviour of selfish nodes using Evolutionary Game Theory (EGT) [32] instead of the classical GT used in *RACOON*. Using EGT, we can relax the assumption of perfect rationality of the nodes, and consider them as active learners who adjust their strategy over time in response to repeated observations of their own and others' utilities. Such learning and adaptation processes better reflect with the computational and decisional capabilities of real nodes [28], [29]. Furthermore, as noted by Palomar et al. [27], an evolutionary approach is more appropriate for modelling the dynamic behaviour of cooperative systems. Third, we integrate the *RACOON++* functionalities with the P2P simulator PeerSim [10]. To the best of our knowledge, the simulator we developed in *RACOON* was the first tool able to dynamically simulate selfish strategic behaviours. However, like all custom built simulators, it had neither the maturity nor the acceptance of state-of-the-art tools like PeerSim [39].

In summary, we present the following contributions:

- **Selfishness-aware design of cooperative systems**. We define simple declarative models for specifying cooperative protocols as well as for describing nodes' selfishness.
- **Automatic (evolutionary) game-theoretic reasoning**. We define the system under design as an evolutionary game, in order to describe how successful behaviours spread in a population of selfish individuals. We also provide an automatic methodology to generate the game using the information contained in the declarative models. Finally, we extend the PeerSim simulator with the ability to conduct EGT analysis to simulate selfish behaviours.
- **Objective-oriented configuration**. We propose an automatic configuration method for an accountability and reputation mechanism in a cooperative system, which can meet the resilience and performance objectives set by a system designer in a reasonable time for a design-time activity (18 minutes on average).
- **Generality, simplicity and performance**. We assess the design effort and effectiveness of using *RACOON++* on three use cases: a P2P live streaming system [14], a P2P load balancing protocol [10], and an anonymous communication system based on Onion Routing [15].

The rest of the paper is organised as follows. Section II reviews the related work. Section III presents an overview of *RACOON++*, followed by a detailed description of its two phases: the design phase (Section IV) and the tuning phase (Section V). Section VI summarises the operation of the framework from the designer's point of view. Section VII presents a performance evaluation of *RACOON++*. Finally, the paper concludes in Section VIII.

II. Related work

Game Theory (GT) is a mathematical framework to model and predict the interactions among selfish and strategic individuals [31]. Much work on GT as a tool for system designers has been carried out in the context of content dissemination [24]–[27], wireless and mobile networking [20], [21], cryptography, anonymity and privacy mechanisms [16]–[18], [23]. The objective of these works is to make cooperation the best choice for all nodes, i.e. a Nash Equilibrium. Most of the GT solutions are not readily applicable to cooperative systems [30], mainly due to simplifying assumptions to make the model tractable, e.g., assuming that nodes have perfect rationality [18]–[20] or are risk averse [23]–[25]. Evolutionary Game Theory (EGT) relaxes these assumptions by considering nodes with limited rationality that adapt their behaviours dynamically, by learning from experience [32]. However, most applications of EGT to system design are only suitable for toy systems [27]–[29], because of the difficulty of modelling a complex system in a formal way. *RACOON++* provides means for transforming models familiar to system designers (state machines) into games, thus making the power of EGT reasoning accessible to non-game theory experts.

Another common limitation of GT models is that they are tailored to a specific system problem and are difficult to adapt to a changing environment. A notable example is the BAR Model for designing systems robust to selfish and Byzantine participants [22]. Besides the difficulties in the manual design of a BAR-tolerant system [22]–[25], the resulting solution suffers from poor flexibility and maintainability. Every change to the system parameters requires a full revision of the design, hindering the reuse of a successful solution in other systems.

On the contrary, the general approach of *RACOON++*, as well as its application-independent mechanisms to enforce cooperation, are reusable by construction. Furthermore, *RACOON++* supports a semi-automatic design flow that greatly facilitates the refinement of system requirements and specification.

Yumerefendi and Chase [34] advocate accountability as a viable solution for dealing with non-cooperative behaviours. Distributed accountability mechanisms [12]–[14], notably FullReview [13], have been proven effective in systems populated by selfish nodes, making them an ideal and general component for supporting cooperation in *RACOON++*. However, enforcing accountability incurs a substantial cost on the system, mainly due to the high message overhead and the intensive use of cryptography. This poses a significant configuration problem, requiring designers to carefully look for a trade-off between performance and selfishness-resilience. Since no guidelines are given in the studies cited above, tuning the accountability mechanisms is manual and time-consuming. In contrast, *RACOON++* mostly automates this task.

Accountability systems usually address selfishness by isolating or evicting selfish nodes [13], [14]. A complementary approach is to introduce incentives to make cooperation more profitable for selfish nodes. The vast body of literature on incentives for cooperative systems can broadly be divided into trust-based and trade-based incentive schemes. A trust-based scheme associates each node with a level of trust, which can serve as a guide for distributing incentives. For example, nodes with a high trust level can benefit from a higher quality of service. Reputation is the principal mechanism to evaluate and maintain trust in dynamic large-scale environments like cooperative systems [37]. Reputation mechanisms offer high flexibility and scalability, and can be implemented in a fully decentralised manner. Because of these features, *RACOON++* uses a distributed reputation mechanism to foster cooperation, which complements the trust-enabling approach of its accountability system. Specifically, the reputation of nodes is updated based on verifiable evidence and linked to a unique and permanent identity, thereby inhibiting the dissemination of false information (e.g., bad mouthing, unfair praise) [38].[1]

In trade-based incentive schemes, nodes pay for obtaining services or resources (as consumers) and get paid for sharing (as providers). In schemes such as barter and tit-for-tat [11], [20], [25], the trade is direct and symmetric: each unit of resource is reciprocated with a unit of resource. Although very robust and easy to implement, these schemes require that trading nodes need something from each other (a condition known as double coincidence of wants) and that they establish long duration relationships to ensure adequate opportunities for reciprocation. These requirements can be too restrictive or inefficient in some cooperative systems, such as opportunistic networks [20] and real-time constrained applications [24]. To overcome this limitation, credit-based mechanisms [35], [36] use virtual currency as the commodity for trading resources and allowing its later expenditure. On the downside, these approaches introduce economic issues in the system (e.g., price negotiation, inflation, deflation) [36], and may require a trusted authority (bank) to issue and certify the currency [35]. By contrast, *RACOON++* uses fully distributed mechanisms that are not affected by economic factors.

Several frameworks and domain-specific languages have been proposed to ease the task of designing and evaluating dependable distributed systems (e.g., [40], [41]). Although these solutions yield good results in terms of system performance and designer effort, none of them addresses the specific threat of selfish deviations in cooperative distributed systems.

III. *RACOON++*: Overview

RACOON++ is a design and simulation framework aimed at supporting system designers in building a selfishness-resilient cooperative system that meets desired performance objectives. As depicted in Fig. 1, the operation of *RACOON++* consists of two phases: the assisted *design* of the system and the objective-oriented *tuning* of its parameters. The dark boxes in Fig. 1 are the input provided by the designer. We give an overview of these phases here, and more details in Sections IV and V.

The design phase is initiated by the system designer (hereafter "Designer", for brevity), who provides a state-machine specification of the communication protocols composing the cooperative system. In Step (1) of Fig 1, *RACOON++* integrates the system specification with mechanisms to encourage nodes to cooperate. Specifically, *RACOON++* uses two general and configurable Cooperation Enforcement Mechanisms (CEM): an 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)). For a better control over the process, the Designer can describe the preferences and capabilities of selfish nodes using the *Selfishness Model*. The result is an *Extended Specification* of the cooperative system, which includes selfish behaviours and cooperation enforcement mechanisms.

The goal of the tuning phase is to find a configuration setting for the CEM that makes the Extended Specification meet a list of *Design Objectives* set by the Designer. Tuning is an iterative refinement process consisting of a sequence of two steps: game-based evaluation and configuration exploration (Steps (3) and (4) in Fig. 1). The evaluation is done using game theory-driven simulations, carried out automatically by our framework. More precisely, *RACOON++* transforms the Extended Specification into a game model, which it uses to simulate the strategic behaviour of selfish nodes given an implementation of the system specification by the Designer. The framework uses the results of the evaluation to traverse the configuration space and evaluate new configuration candidates for the CEM. The output of *RACOON++* is a new specification of the cooperative system that includes finely tuned accountability and reputation mechanisms to achieve the selfishness-resilience and performance objectives set by the Designer. This output provides a reference guide for developers to use when implementing the system.

[1] Although out of the scope of our present work, it is worth noting that strong identities are the prerequisite for preventing other strategic misbehaviours against reputation systems, such as whitewashing and Sybil attacks [34], [38].

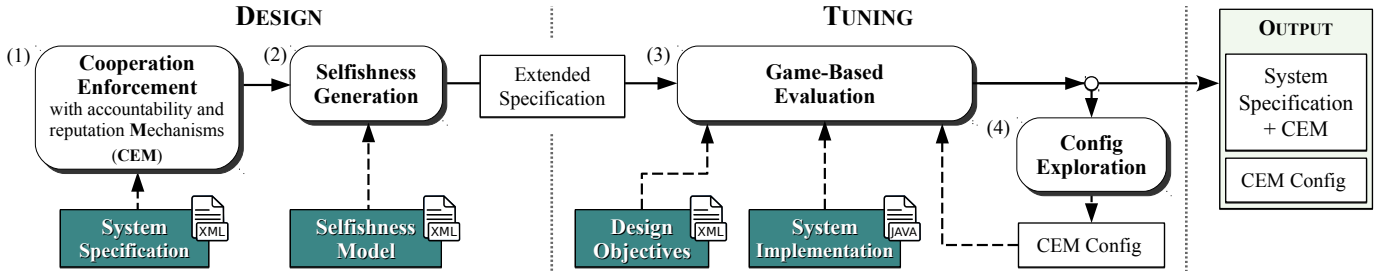


Fig. 1: Overview of the *RACOON++* framework.

IV. *RACOON++* Design phase

The design phase helps the Designer in specifying a cooperative system that embeds mechanisms for fostering cooperation as well as in defining a behavioural model of the system participants. The output is a new artefact called the *Extended Specification* of the system.

In this section, we introduce the inputs of the phase, we describe the accountability and reputation mechanisms used in *RACOON++*, and, finally, we present the algorithm used to generate selfish deviations. To support the description of the framework, we use the simple communication protocol *Search, Request & Response* (S-R-R) shown in Fig. 2. 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 set 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).

A. Input of the Design phase

The inputs of the design phase are the *functional specification* of the protocols of the system that should be made resilient to selfish behaviours, and the *selfishness model* adopted by selfish nodes.

1) Functional specification

The functional specification describes the correct, cooperative behaviour of nodes by means of communication protocols. Like many other approaches [12], [22], [40], notably the accountability system [13] that we plan to adapt for our framework, each communication protocol is specified using a notation based on deterministic finite state machines, called a *Protocol Automaton*. A Protocol Automaton PA is a tuple $\langle R, S, T, M, G, C\rangle$, with each component described below.

Roles (R): the parties involved in the protocol execution. A role determines the responsibilities of a party (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 requester of a resource or service, and the provider. Other types are also possible (e.g., brokers, auditors, recommenders). For example, the S-R-R protocol has two roles: the requester r_0 and the set of potential providers R_1. Formally, a role $r \in R$ is a tuple $\langle rId, cardinality, rType\rangle$, where $cardinality$ denotes the number of nodes represented by r, and $rType$ is either requester, provider, or other.

States (S): the set of states that the system goes through when implementing the communication protocol. A state $s \in S$ is a tuple $\langle sId, roleId, sType\rangle$, where $roleId$ identifies the role $r \in R$ that triggers a change of state or terminates the protocol execution, and $sType$ is either initial, final, or intermediate.

Transitions (T): a transition represents a protocol step, i.e., the set of method calls that determine the next protocol state. The PA supports three types of transition: 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. The remaining transition types allow to define the (communication or computation) method that triggers the transition. Formally, a transition $t \in T$ is a tuple $\langle tId, state1Id, state2Id, methodId\rangle$, where $state1Id$ and $state2Id$ identify the source and target states in S, and $methodId$ identifies the method executed in t (*null*, for abstract transitions). In the S-R-R protocol, the transitions are *search* (abstract), *request* and *response* (communication).

Methods (M): the set of actions that can trigger a protocol transition. A communication method represents the delivery of a message from one role to another, whereas a computation method performs local computations. A method $m \in M$ is a tuple $\langle mId, messageId\rangle$, where $messageId$ is defined only for communication methods, and *null* otherwise. For instance, *request* is the communication method of the S-R-R protocol that sends a message g_2 to R_1. Note that the methods called during an abstract transition (e.g., *search*) are not in M.

Messages (G): a message $g \in G$ sent by a communication method is a tuple $\langle gId, senderId, receiverId, contentId\rangle$, where $senderId$ and $receiverId$ identify the interacting roles in R, and $contentId$ refers to the content $c \in C$ carried by g.

Contents (C): a content $c \in C$ is a collection of data units (e.g., integers, binary files), formalised as the tuple $\langle cId, cType, cLength\rangle$, which defines the data type $cType$[2] and the number $cLength$ of data units comprising the content.

Fig. 3 shows the state diagram of the S-R-R protocol. The labels on each transition indicate the role and the method that trigger the transition, along with the message sent (if any). For example, the label between states s_1 and s_2 indicates that role r_0 invokes the communication method *request* to send the message g_2 to R_1. The label of an abstract transition indicates the role that executes the first method encapsulated in it.

[2]Defined by the XML Schema type system.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2706286, IEEE Transactions on Dependable and Secure Computing

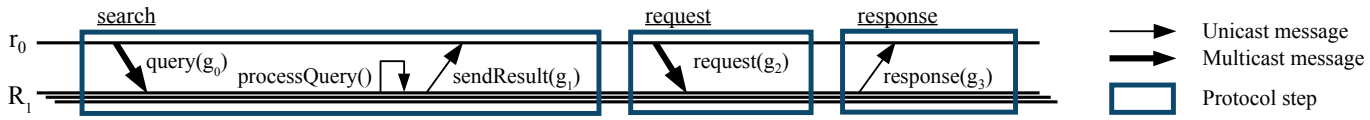


Fig. 2: The S-R-R protocol between nodes r_0 and R_1.

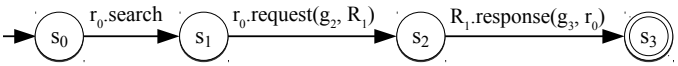


Fig. 3: The Protocol Automaton of the S-R-R protocol.

2) Selfishness Model

The selfishness model carries the information about the economic drivers of a party, by specifying the utility that a node obtains in participating in the system. Also, it defines the possible deviations from the functional specification. Formally, a selfishness model is a tuple $\langle V, D \rangle$, detailed below.

Valuations (V): the set of contributions to the overall utility of a certain behaviour. The utility that a node receives by participating in the system is given by the benefit obtained by consuming resources and the cost of sharing resources. A valuation $v \in V$ specifies this information at the granularity of transitions and messages of a Protocol Automaton. Formally, v is a tuple $\langle vId, vScope, roleId, benefit, cost \rangle$, where $vScope$ is the identifier of the PA element (transition or message) that brings some $benefit$ and $cost$ (numeric values) to the role in R identified by $roleId$.

If the $vScope$ of a valuation v_t refers to a transition $t \in T$, then v_t defines the utility that the role with identifier $v_t.roleId$ obtains by executing t. We denote by $\upsilon(v_t)$ the function to evaluate the contribution of v_t to the overall utility, and we define it as: $\upsilon(v_t) = v_t.benefit - v_t.cost$. 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 $\langle v_0, search, r_0, 10, 1 \rangle$, which results in a contribution to the utility of $\upsilon(v_0) = 9$. Note that another system designer may value the same transition differently, according to her expertise and knowledge of the system.

Conversely, if the $vScope$ of a valuation v_g refers to a message $g \in G$, then v_g defines the utility obtained by the role identified in v_g when g is sent or received. The contribution of v_g to the overall utility accounts for the cardinality of the receiver role of the message as well as the number of data units comprising the delivered content. This is based on the observation that the costs and benefits of a message are typically 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 a message g_2 to role R_1. Let c_2 be the content transmitted by g_2, and let $\langle v_1, g_2, r_0, 5, 1 \rangle$ be the valuation associated with g_2. In this case, the contribution that v_1 makes to the utility of the node playing the role of r_0 is given by: $\upsilon(v_1) = (5-1) \cdot c_2.cLength \cdot R_1.cardinality$. 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, $v_2 = \langle v_2, g_2, R_1, 1, 0 \rangle$.

Selfish Deviations (D): the set of deviations from the correct execution of the system, made by selfish nodes to increase their utility. In the context of a cooperative system, a selfish node can increase its utility by reducing the cost of sharing resources. Concretely, a deviation can reduce the bandwidth consumption by sending fewer and shorter messages [7], [8], [14], [23], or interrupt resource contribution by refusing to execute some methods [9], [13], [23]. Based on the study of these and other examples from the literature, we have selected the three generic types of selfish deviation supported by *RACOON++*, namely: (1) *timeout* deviation: a node does not implement the prescribed transition within the time limit; (2) *subset* deviation: a node sends a subset of the correct message content; and (3) *multicast* deviation: a node sends a message to a random subset of the legitimate recipients. Some other types of selfishness, notably collusion and provision of false or misleading information, have been investigated in our recent work [42].

Formally, a selfish deviation $d \in D$ from a transition $t \in T$ is a tuple $\langle dId, dScope, dType, degree \rangle$, where $dScope$ identifies t, $dType$ indicates whether d is a timeout, subset or multicast deviation, and $degree \in [0, 1]$ specifies the intensity of the deviation. Note that the *timeout* deviation can affect all types of transitions, whereas the *subset* and *multicast* deviations affect only communication transitions, as they interfere with the delivery of a message. For instance, $\langle d_0, response, timeout, 1 \rangle$ describes selfish nodes that never reply to a request. Note that *timeout* deviations only occur to the maximum degree. 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, where the wildcard value "*" indicates that all communication transitions in the PA are subject to d_1.

B. 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 accountability and reputation protocols 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. To this end, the CEM 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. 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 advantage of such a flexible incentive scheme is twofold. First, and with respect to rigid punishment strategies such as direct eviction [13], it alleviates the impact of false-positive detection of selfish nodes [19]. Second, all nodes are evaluated based on the quality of their contribution (cooperative or selfish) rather than on the quantity of the shared resources, so as not to penalise nodes suffering from persistent resource shortage (e.g., battery-powered devices).

The CEM is a key component for the Designer, as it provides general and off-the-shelf mechanisms for fostering cooperation in a wide range of settings, without the need to devise ad-hoc solutions for the particular system at hand.

Hereafter, we discuss the CEM used in *RACOON++*.

1) Accountability Mechanism

RACOON++ uses accountability techniques for detecting misbehaviours and assigning nodes non-repudiable responsibility for their actions. Specifically, we propose the *R-acc* mechanism, based on the FullReview [13] protocols and architecture. *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 on other assumptions (e.g., nodes are not risk averse).

RACOON++ can automatically integrate *R-acc* into the functional specification provided by the Designer. To begin, *R-acc* requires each node to maintain a tamper-evident record of all its observable actions (i.e., message exchanges). Further, each node is assigned to a set of other nodes, called its witness set. A witness is in charge of auditing the log of its monitored nodes, generating provable evidence of their behaviours and assigning punishments or rewards accordingly. Such operations are defined by the protocols described below.

Commitment protocol: ensures that the sender and the receiver of a message have provable evidence that the other party has logged the exchange. Fig. 4 shows the integration between the PA of the S-R-R protocol and the commitment protocol. Consider for example the node with role r_0 in state s_1. Before sending the request message g_2 to R_1, r_0 records the action in its local log, creating a new entry e_w. Then, r_0 generates a signed statement $\alpha_w^{r_0}$, called an authenticator, indicating that it has logged e_w. Next, r_0 sends $\alpha_w^{r_0}$ to R_1 along with the message. Upon receiving the message (state f_0 in Fig. 4), each node in R_1 logs this event in a new log entry e_z, and generates the corresponding authenticator $\alpha_z^{R_1}$. Finally, R_1 sends this authenticator to r_0 to acknowledge the reception of g_2.

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. Fig. 5 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 requests and obtains a portion of the log of r_m (messages g_{a1} and g_{a2}). Then, r_w verifies if r_m's log conforms to the correct behaviour making up the functional specification of the system (transition *audit* in Fig. 5). The witness sends the audit result back to the monitored node (message g_{a3}). Finally, r_m checks the correctness of its audit by forwarding the collected results to the witness set of each of its witnesses (indicated as $w(r_w)$ in the figure). If the witness does not receive the requested log from r_m (state f_7 in Fig. 5), then it will address the issue by using the challenge/response protocol.

Consistency protocol: ensures that each node maintains a single and consistent linear log [13].

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*, and creates a challenge for it. In FullReview, 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.

R-acc does not include the evidence transfer protocol used in FullReview [13]. 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.

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. *RACOON++* includes a PA 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.

2) Reputation Mechanism

The reputation of a node is the summary of its history of behaviours, which is used to assist nodes in choosing a cooperative partner with which to interact. Cooperation leads to a good reputation, whereas selfish behaviours lead to a bad reputation. To provide this information, the CEM includes a distributed reputation mechanism (*R-rep*) to form, aggregate, and disseminate reputation values.

In order to reduce design complexity and to reuse available knowledge, *R-rep* shares some features with the *R-acc* accountability mechanism described above. First, a witness node plays the role of recommender in *R-rep*, as it can form an opinion of a monitored node based on the audit result. This solution keeps the computational overhead of the CEM under control, as it avoids performing the same 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) [38]. Second, *R-rep* relies on *R-acc* for storing the reputation data in a reliable manner. More precisely, nodes store their own reputation locally. To prevent manipulations, only witnesses — in their role of recommenders — can update the reputation value. Also, the update must be recorded in the *R-acc* secure log, so that any tampering can be detected.

In *R-rep*, the reputation ρ of a node is an integer value

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2706286, IEEE Transactions on Dependable and Secure Computing

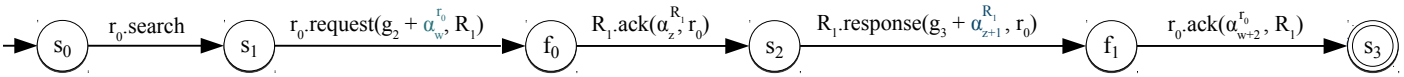


Fig. 4: The integration between the commitment protocol of *R-acc* with the S-R-R protocol shown in Fig. 3.

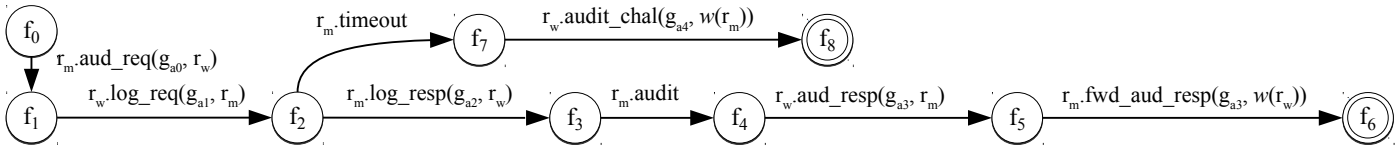


Fig. 5: The Protocol Automaton of the *R-Acc* audit protocol.

between 0 and an upper limit ρ_{max}. The value of ρ is estimated after every audit, and can be calculated as:

$$\rho = \begin{cases} \max\{\rho_{old} - f_p(\rho_{old}, d_p, d_d), 0\}, & \text{if positive audit} \\ \min\{\rho_{old} + f_r(\rho_{old}, d_r), \rho_{max}\}, & \text{if negative audit} \end{cases}$$

where ρ_{old} is the old reputation value, f_p and f_d are the update functions in case of punishment or rewards, d_p and d_r are two *R-rep* parameters that control the degree of punishment and of reward, and d_d is the degree of the deviations detected by the audit. A punishment comes in the form of a reputation decrease. The decrease value is proportional to d_p and to the degree d_d of the detected deviation, and indirectly proportional to the old reputation value ρ_{old}, in such a way as to punish with greater severity nodes that already have a bad reputation, in order to inhibit recidivism. In the case of a negative audit, function f_r rewards the cooperative node by assigning a reputation increase, which is proportional to the degree of reward d_r and to the old reputation value.

Consider for example the following setting of *R-rep*: ρ_{max} is 10, the d_r is 0.2, and d_p is 2. Also, let a currently cooperative node have the reputation value 5. After being audited, the node's reputation value will be $\rho = \min\{5+(5\cdot 0.2), 10\} = 6$. Given the same *R-rep* setting, consider a selfish node that has deviated with degree 1 from the correct specification of the protocol. Assuming the current reputation of the node be 6, then its new reputation after the audit will be: $\rho = \max\{6-2\cdot 1\cdot(10-6), 0\} = 0$. Note that the set up of the *R-rep* parameters can yield different results, with varying effects on the nodes' behaviour. In Section V-D, we will show how the tuning phase of *RACOON++* can support the automatic configuration of these parameters to induce the desired behaviour.

C. Selfishness Generation

The last step of the design phase is the automatic generation of selfish deviations from both the functional specification of the system and the CEM. This is implemented by the *Selfish Deviation Generation* (SDG) algorithm given in Alg. 1. The algorithm takes as input a Protocol Automaton and the Selfishness Model SM. Then, it extends the PA with new elements (states, transitions, roles, etc.) representing deviations. Note that the *SDG* algorithm can generate the deviation types introduced in Section IV-A2, namely, *timeout*, *subset*, and *multicast* deviations. For brevity, in the pseudo-code we use the notation $\boldsymbol{get}(elementId)$ to refer to the element of the PA to which the *elementId* identifier is associated.

Alg. 1: The *Selfish Deviation Generation* algorithm.

Data: A *Protocol Automaton* PA, the *selfishness model* SM.

Algorithm `SDG(PA,SM)`

1 $origT := T$ `// original transitions in PA`
2 **foreach** $t \in origT$ **do**
3 **if** $\exists\, d \in D \mid d.dScope = \{t.tId, \text{“*”}\}$ **then**
4 **if** $d.dType = \text{“}timeout\text{”}$ **then**
5 InjectTimeoutDev(t)
`/* only for communication transitions */`
6 **if** $\boldsymbol{get}(t.methodId).messageId \neq null$ **then**
7 $c := \boldsymbol{get}(t.methodId.messageId.contentId)$
8 **if** $d.dType = \text{“}subset\text{”}$ **and** $c.cLength > 1$ **then**
9 InjectSubsetDev(t, c, d)
10 $r := \boldsymbol{get}(t.state2Id.roleId)$ `// recipient role`
11 **if** $d.dType = \text{“}multicast\text{”}$ **and** $r.cardinality > 1$ **then**
12 InjectMulticastDev(t, r, d)

Procedure `InjectTimeoutDev(t)`

13 $s' := \langle new\_sId, null, final\rangle$
14 $sourceState := \boldsymbol{get}(t.state1Id)$
15 $t' := \langle new\_tId, sourceState.sId, s'.sId, null\rangle$
16 **add** s' and t' to PA

Procedure `InjectSubsetDev(t, c, d)`

17 $length' := \lfloor c.cLength(1 - d.degree)\rfloor$
18 $c' := \langle new\_cId, c.cType, length'\rangle$
19 $g := \boldsymbol{get}(t.methodId.messageId)$
20 $g' := \langle new\_gId, g.senderId, g.receiverId, c'.cId\rangle$
21 $m' := \langle new\_mId, g'.gId\rangle$
22 $targetState := \boldsymbol{get}(t.state2Id)$
23 $s' := \langle new\_sId, targetState.roleId, targetState.sType\rangle$
24 $t' := \langle new\_tId, t.state1Id, s'.sId, m'.mId\rangle$
25 **add** c', g', m', s', and t' to PA
26 **foreach** $ot \in T \mid ot.state1Id = targetState.sId$ **do**
27 $ot' := \langle new\_otId, s', ot.state2Id, ot.methodId\rangle$;
28 **add** ot' to PA;

Procedure `InjectMulticastDev(t, r, d)`

29 $cardinality' := \lfloor r.cardinality(1 - d.degree)\rfloor$
30 $r' := \langle new\_rId, cardinality'\rangle$
31 $s' := \langle new\_sId, r'.rId, s.sType\rangle$
32 $message := \boldsymbol{get}(t.methodId.messageId)$
33 $g' := \langle new\_gId, message.contentId\rangle$
34 $m' := \langle new\_mId, g'.gId\rangle$
35 $t' := \langle new\_tId, t.state1Id, s'.sId, m'.mId\rangle$
36 **add** r', s', g', m', and t' to PA
37 **add** out-transitions of s' ▷ *as in lines 26-28*

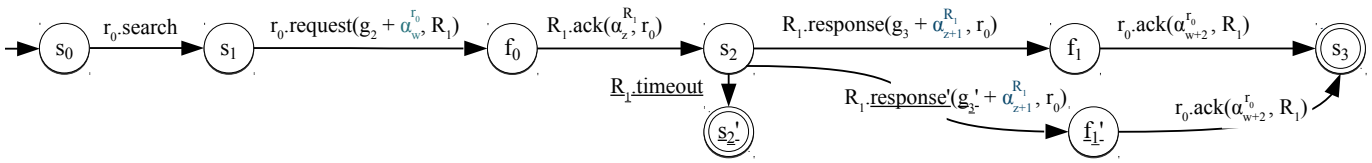


Fig. 6: The Protocol Automaton of the S-R-R protocol, extended with selfish deviations.

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 *SDG* algorithm first checks if the SM contains a selfish deviation d that affects t (line 3 in Alg. 1). Then, it 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. 1). 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, *SDG* checks if the message content c is a collection of data units (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 using $d.degree$ (line 17).

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). If so, line 12 calls the procedure *InjectMulticastDev* to create the role r' (line 30) with a smaller cardinality than the correct one (calculated in line 29).

Fig. 6 shows the result of executing the *SDG* algorithm on the Protocol Automaton of Fig. 3. 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').

V. *RACOON++* Tuning phase

The tuning phase of *RACOON++* aims at configuring the accountability and reputation mechanisms according to a list of design objectives provided by the Designer. Tuning involves an iterative two-step refinement process, which alternates evaluation with the tuning of the configuration parameters. The evaluation involves EGT analysis to study the system dynamics in a given configuration setting. This task is performed by the *R-sim* simulator integrated into the framework. We have chosen EGT simulations as a modelling tool because in many practical settings the populations of individuals participating in a system evolve towards states of statistical equilibrium. After the evaluation, an exploration algorithm uses the evaluation results to optimise the parameters of the CEM. The tuning process ends after a number of iterations, or when a configuration that satisfies the Designer's objectives is found.

A. Input of the Tuning phase

RACOON++ provides a set of selfish-resilience and performance objectives for the cooperative systems designed within its framework. Each objective defines a predicate over a system metric, which can be evaluated by the *RACOON++* evaluation tool, i.e., the *R-sim* simulator. The possible predicates are *at most* and *at least*. Hereafter, we list some of the application-independent objectives natively supported by *RACOON++*.

- *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;
- *Audit recall*: the number of correct positive audits divided by the number of audits that should have been positive;
- *CEM bandwidth overhead*: the additional bandwidth consumed by the accountability and reputation mechanisms;
- *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). For each custom objective, the Designer needs to implement the methods to collect and evaluate the related metrics in the evaluation tool.

The second input of the tuning phase is an implementation of the functional specification for the *R-sim* simulator.

B. Evolutionary game model

EGT models how strategic individuals evolve their behaviours by learning and imitating [32]. Similarly to several recent works [27]–[29], *RACOON++* applies this theoretical framework to model the dynamic behaviour of selfish nodes in a P2P system. The components of an evolutionary game are: (i) a static representation of the system interactions, i.e., the 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.

1) 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 using the *extensive form* (or *game tree*) [31]. Fig. 7 shows the game tree of the stage game derived from the S-R-R protocol illustrated in Fig. 6. *RACOON++* provides an automatic tool to create the SG using the information contained in the Extended Specification resulting from the design phase. Specifically, the tool translates the PA included

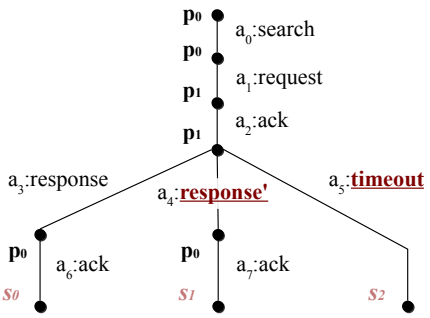


Fig. 7: The SG derived from the S-R-R protocol in Fig. 4. The label besides each decision node indicates the player that takes action at that node. The label on each edge denotes an action along with its corresponding method in the PA. The labels beside each leaf denote the strategy profile of that play.

Player	Strategy
p_0	$\{a_0$*:search, a_1:request, a_6:ack*$\}$
p_1	$\{a_2$*:ack, a_3:response*$\}$

TABLE I: The strategies of the strategy profile s_0, implementing the cooperative execution of the stage game in Fig. 7.

in the Extended Specification into the elements of a stage game as described hereafter.

Players. A *player* p represents a role of the PA. For example, players p_0 and p_1 in Fig. 7 map to roles r_0 and R_1 of the S-R-R protocol, respectively. For ease of notation, let $p_k.type$ refer to the $rType$ of the role mapped by player p_k.

Nodes. A *node* 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 Fig. 7, each leaf k is labelled with the corresponding outcome o_k.

Actions. An *action* is a move of the player in the SG, and is derived from a method in the PA. Note that an edge of the game tree in Fig. 7 corresponds to a transition in the PA.

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 Fig. 7, which represents the cooperative execution of the S-R-R protocol: Table I reports the strategies of players p_0 and p_1 to implement it.

2) 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 [27] and [28], 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.

Player	Strategy profile	Strategy
p_0	s_1	$\{a_0$*:search, a_1:request, a_6:ack*$\}$
p_1	s_2	$\{a_2$*:ack, a_5:timeout*$\}$

TABLE II: The strategies implemented in the SG of Figure 7 when players p_0 and p_1 are from sub-populations ω_1 and ω_3.

Thus, for example, and with reference to Fig. 7, the strategy profile s_0 represents the behaviour of cooperative nodes and includes the strategies presented in Table I.

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_i participates in the system by repeatedly playing what specified by her strategy profile, regardless of the outcome of the play. However, a member of ω_i can join another sub-population ω_j if she expects to increase his 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 across the sub-populations can vary. We call this information the *population state* of the system.

3) Utility function

The utility function of a player assigns a value (i.e., the utility) to each outcome of a game. An outcome 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 Fig. 7, and let players p_0 and p_1 be played by members of sub-population ω_1 and ω_3, respectively. Table II 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 Fig. 7, which leads to the outcome induced by the strategy profile s_2. The outcomes of a stage game describe the interaction between every possible combination of players from different sub-populations.

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 IV-A2). Let us illustrate the procedure to evaluate the protocol payoff γ_0 in the stage game of Fig. 7, 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: $\langle v_0, search, r_0, 10, 1\rangle$ and $\langle v_1, g_3, r_0, 3, 1\rangle$, which refer to the a_0*:search* and a_3*:response* edges in Fig. 7, respectively. Let the content $c \in C$ carried by message g_3 be a list of 10 data units. Then, the protocol payoff of player p_0 is:

$$\gamma_0(p_0) = \upsilon(v_0) + \upsilon(v_1) = 9 + 2 \cdot r_0.cardinality \cdot c.cLength = 29.$$

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 IV-B). 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:

$$u_j(p_i) = \begin{cases} \gamma_j(p_i) \cdot \varrho(p_i) & \text{if } p_i.type = \text{“}requester\text{”} \\ \gamma_j(p_i) & \text{otherwise} \end{cases}$$

where the function $\varrho : \mathcal{P} \rightarrow [0, 1]$ 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:

$$u_0(p_0) = \gamma_0(p_0) \cdot \rho(p_0) = 29 \cdot 0.6 \simeq 17.4 \ .$$

4) Evolutionary dynamics

A common assumption in classical 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 [31]. However, as other works have highlighted [27]–[29], this assumption places a heavy burden on nodes' computational and communication capabilities, which is infeasible in most real-world cooperative systems. On the contrary, EGT assumes that players are neither perfectly rational nor fully informed about all the possible strategies, but tend to implement the most remunerative strategy through learning and imitation [32].

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 [29]. This probability determines the evolution rate: the smaller its value, the slower the fittest sub-population in the system increases.

C. Game-based evaluation

The game-based evaluation step evaluates a configuration setting for the CEM. To this end, *RACOON++* first creates an evolutionary game model of the system and then it simulates the game dynamics using the *RACOON++* evaluation tool *R-sim*. The simulation results indicate whether the evaluated CEM configuration has satisfied the list of design objectives set by the Designer or not.

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. The networking environment of *R-sim* consists of independent nodes that are able to send a message to any other node, provided that the address of the target node is known. Communication is assumed to be subject to arbitrary message loss, controlled by a probability parameter. Nodes can leave and join the network at any time. The simulation engine of *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 V-B. During the simulation *R-sim* collects statistics about such dynamics, to evaluate the design objectives.

In contrast with *RACOON*, which includes a custom-built simulator for cooperative systems [19], *RACOON++* relies on the state-of-the-art PeerSim simulator [10], thereby improving the usability, accuracy and performance of the framework. We have chosen PeerSim among other simulation tools (see [39] for a comprehensive review) because: (1) it meets the requirements of scalability and dynamicity imposed by the evolutionary model; (2) it supports integration with *RACOON++* thanks to its modular architecture; (3) it is an active project, with a good developer community and support. *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. Also, *R-sim* includes a reference implementation of the accountability and reputation systems used by *RACOON++*, along with an intuitive API to simulate their calls. The Designer can use these facilities to implement the functional specification of his system for PeerSim. 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.

Other important *R-sim* parameters are listed below:

- *Network*: the network size; the message loss rate.
- *Evolutionary game model*: the initial population state (e.g., equal-sized sub-populations, majority of cooperative nodes); the probability to join a fitter sub-population.
- *Monitoring*: the duration of a simulation; the frequency and the types of statistics to collect (e.g., nodes' payoffs, amount of messages exchanged, audit results).

D. Design Space Exploration

The output of the *RACOON++* framework is the design and configuration of a cooperative system that achieves the design objectives set by the Designer. Thus far, we have described how *RACOON++* fosters cooperation using accountability and reputation mechanisms (Section IV-B), and how it evaluates the system performance using *EGT* and simulation (Section V-C). The last step of the framework relies on the evaluation results to tune the configuration parameters of the CEM, aiming to achieve the desired design objectives. A configuration candidate is an assignment of the *R-acc* and *R-rep* parameters, i.e., the size of the witness set, the audit period, the audit probability, the degree of punishment, and the degree of reward.

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. If no feasible solution is found after a pre-defined number of iterations (e.g., because the objectives were contradictory or too demanding), the framework stops the search, asking the Designer to improve the design manually or to relax the design objectives.

RACOON++ explores the configuration space using a greedy constraint-satisfaction algorithm, which is guided by a set of observations derived from an empirical analysis of the CEM parameters and their impact on the design objectives natively supported by *RACOON++*.[3] For instance, we observed 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 observed that the shorter the audit period, the higher the cooperation level, because selfish nodes are detected earlier and punished more often. The exploration algorithm relies on these observations to generate the next configuration candidate. For instance, if the evaluation of a given configuration results in a bandwidth overhead larger than what required by a design objective, the exploration algorithm will not generate configuration candidates with a greater number of witnesses. If no guidelines are available for updating a particular configuration, the exploration algorithm will create a random configuration candidate. In order to avoid the re-exploration of the regions of the configuration space, the algorithm records the previously generated candidates.

VI. Using the *RACOON++* framework

RACOON++ is provided as a Java program, which is released under a free software licence and is publicly available [33]. In the previous sections, we described the main steps and building blocks of the framework. Now we turn our attention to how *RACOON++* is used by the Designer.

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 IV-A2. 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 [12], [13].

Then, the Designer inputs the functional specification, along with the selfishness model to study (Section IV-A2) and the design objectives to achieve (Section V-A), to the framework. In *RACOON++*, these specifications are encoded in an XML-based format, and are provided as a single XML document.[4]

To evaluate a configuration setting for the CEM, *RACOON++* simulates the system behaviour using the integrated simulation framework *R-sim*, based on the PeerSim simulator. To this end, the Designer has to produce a Java implementation of the cooperative system, notably of its functional specification. *R-sim* facilitates this task by providing a set of ready-to-use components and an intuitive API for interfacing a standard PeerSim protocol with the *RACOON++* models and functionalities. In particular, the framework includes an implementation of the CEM, the algorithms to simulate the behaviour of selfish nodes, and 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.

Once all the inputs have been defined, the Designer can run the *RACOON++* framework and wait for the result of its design and tuning phases (Fig. 1).

VII. Evaluation

In this section, we demonstrate the benefits of using *RACOON++* to design selfishness-resilient cooperative systems. First, we introduce the three use cases considered in the evaluation, namely, a live-streaming protocol, a load balancing protocol, and an anonymous communication system. Second, we assess the effort required by a Designer to specify and implement the use cases. Third, 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. Then, 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.

The implementation of the use cases, as well as the configuration files related to the experiments reported in this section, can be downloaded from the project website [33].

A. Use cases

We consider the following use cases.

Live Streaming. A P2P live streaming 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 gossip-based live streaming protocol studied by Guerraoui et al. [14].

Load Balancing. The heterogeneity of nodes and the high dynamics of P2P systems can lead to a load imbalance.[5] 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

[3]The analysis involved the systematic evaluation of 250 configuration candidates in three cooperative systems (i.e., the ones considered for evaluating our work) for a total of 750 experiments.

[4]The XML Schema for this document can be found in [33].

[5]The load can be measured in terms of different metrics, such as the number of queries received per time unit.

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. [10].

Anonymous Communication. This system is based on a simplified version of the Onion Routing protocol for communication channel anonymity [15]. 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 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.

B. 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.

To begin, the Designer specifies the communication protocols (i.e., Protocol Automata) to be included in the functional specification of the system. Fig.s 8-10 illustrate the Protocol Automata defined for our use cases. The live streaming protocol (see Fig. 8) 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. In the load balancing protocol (see Fig. 9) each node starts with a certain amount of load. Periodically, each node r_0 is allowed to transfer all or a portion of its load to one of its neighbours R_1, after a negotiation step. The negotiation is based on locally available information, obtained from past interactions or sample observations [10]. Lastly, 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 Fig. 10).

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.

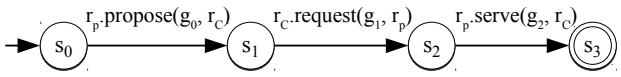


Fig. 8: The PA of the live streaming protocol [14].

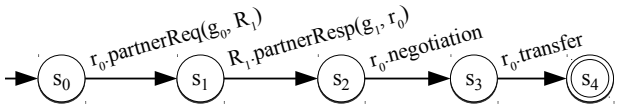


Fig. 9: The PA of the load balancing protocol [10].

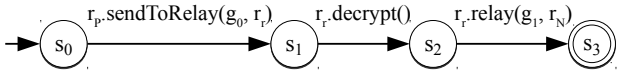


Fig. 10: The PA of the anonymous communication protocol.

	Specification	R-sim Program [a]		
		Std	RS	TOT
Live Streaming protocol	51	384	28	444
Load Balancing protocol	48	232	28	290
Anonymous Comm. protocol	48	212	23	289

[a] Std = standard operation, RS = *R-sim* functionalities, TOT = Std + RS.

TABLE III: Lines of Code needed for the use cases.

Finally, the Designer provides *RACOON++* with a list of design objectives that the system must satisfy. Recall from Section V-C 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, (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 "Specification" column of Table III 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 *RACOON++* framework requires the Designer to implement the functional specification of the system in the *R-sim* simulator. The "R-sim Program" columns of Table III shows the LoC of the use cases' implementations, distinguishing the LoC needed to implement the standard operation ("Std" column) from those introduced to invoke the *R-sim* functionalities ("RS" column). The results show that the software facilities provided by *R-sim* allow adapting a system implementation to be used in *RACOON++* without significant coding effort. More precisely, the RS LoC are in the range 6.3%-9.6% of the total implementation code, which appears reasonable as it corresponds to only 28 additional LoC, at most.

C. 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 combination of design objectives, system parameters (e.g., number of nodes, message loss rate), application-specific parameters (e.g., playback deadline, length of a circuit of relays, initial

distribution of loads), and fraction of selfish nodes in the system.[6] Second, we used *RACOON++* to find a satisfactory configuration for each scenario, while measuring the number of tested configurations and the duration of the process. In the experiment, *RACOON++* tests an average of 7 configurations before finding a satisfactory one (median 4, range 1–56). The process takes less than 18 min on average to complete (median 6 min, range 10 s–208 min),[7] which we consider reasonable, as *RACOON++* runs offline at design time.

Overall, the tuning process failed to meet all the design objectives in only three scenarios out of 90, which we consider as an acceptable result. The failures were due to too hard constraints on the efficiency and effectiveness of the CEM, 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 in terms of the design objectives. If not satisfied with this outcome, the Designer can either relax the performance requirements or optimise some application-specific operation or parameter.

D. RACOON++ effectiveness

In this section, we show that the cooperative systems designed using *RACOON++* can effectively achieve cooperation as well as application-specific objectives in the presence of an increasing proportion of selfish nodes. To this end, we evaluated 3 scenarios per use case, which were randomly selected from the scenarios generated for the previous experiment.

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. For each use case, we calculated the median result from the 3 scenarios. Results in Fig. 11(a) show that the CEM succeeds in making the nodes behave cooperatively in all use cases. Even the worst result (in the live streaming use case) shows a dramatic increase of the cooperation level, from 0.1 to 0.94.

We now focus on the correlation between cooperation level and application-specific performance. Fig.s 11(b-c-d) 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*), and another curve for the results obtained when using *RACOON++* (curve CEM). For example, Fig. 11(d) 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 leads 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 Fig. 11(b). 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 Fig. 11(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 withstand the impact of large populations of selfish nodes.

E. RACOON++ vs FullReview

In this section, we present the benefits of using the *RACOON++* CEM instead of the original FullReview protocols [13]. The main differences between these mechanisms, already discussed in Section IV-B, 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, the CEM was automatically configured by the *RACOON++* tuning phase, whereas FullReview was tuned manually.

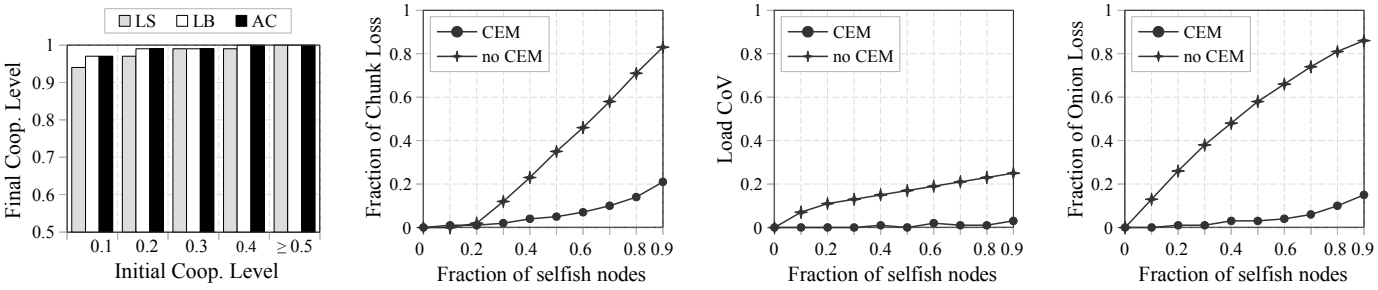
The first important benefit of using CEM is shown in Fig. 12(a), which represents the fraction of nodes that are participating in the cooperative system at the end of the simulation. This figure readily illustrates the opposite approaches adopted by *RACOON++* and FullReview to deal with selfishness: *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, Fig. 12(b) shows that the CEM of *RACOON++* can decrease the bandwidth overhead in the tested system, notably by 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, *R-acc* allows probabilistic audits, which further reduces the traffic and computation overhead associated with the audit activities.

As shown in earlier work [19], FullReview is very sensitive to message loss, which can significantly increase the number of suspect nodes, and might even lead to the wrongful

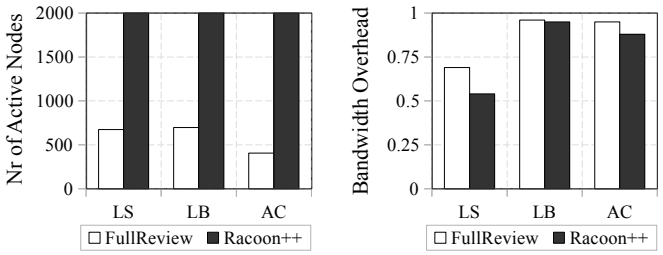
[6]For reasons of space, the full setting for this and the following experiments is not reported here, but have been made available on the project website [33].

[7]Measures made on a 2.8 GHz machine with 8 GB of RAM.



(a) Cooperation Level (b) Chunk Loss in LS (c) Load CoV in LB (d) Onion Loss in AC

Fig. 11: Cooperation levels (a) and application-specific performance of the Live Streaming (LS) (b), Load Balancing (LB) (c), and Anonymous Communication (AC) (c) use cases, when varying the initial fraction of selfish nodes.



(a) Active Nodes (b) Bandwidth Overhead

Fig. 12: Performance comparisons between FullReview and *RACOON++* CEM in the Live Streaming (LS), Load Balancing (LB), and Anonymous Communication (AC) use cases.

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. Fig.s 13(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 IV-B). Notice that the FullReview curves in Fig. 13(a) confirm what already discussed for Fig. 12(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 Fig.s 13(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 Fig.s 13(b-c-d) demonstrate that this simple strategy is enough to guarantee resilience from selfish nodes while being tolerant to message loss.

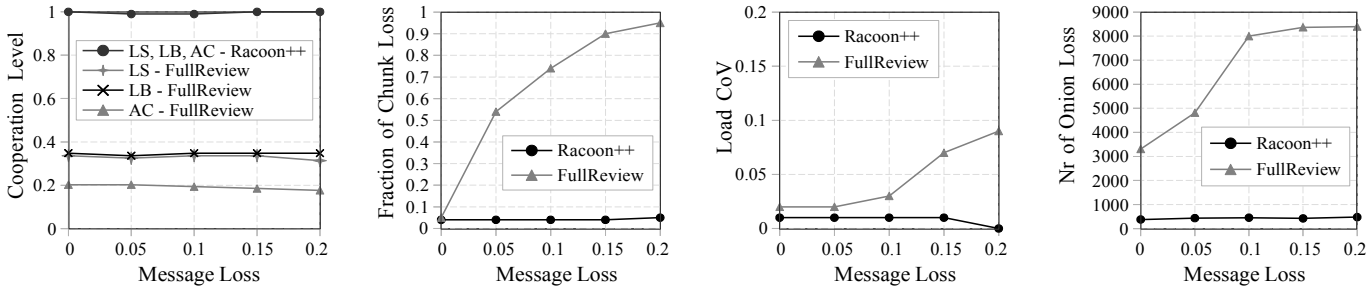
VIII. Conclusions

In this paper we presented *RACOON++*, a model-based framework for designing, configuring, and testing cooperative systems that are resilient to selfish nodes. *RACOON++* relies on accountability and reputation mechanisms to enforce cooperation among selfish nodes. Using a combination of simulation and Evolutionary Game Theory, *RACOON++* automatically configures these mechanisms in a way that meets a set of design objectives specified by the system designer. We illustrated the benefits of using *RACOON++* by designing a P2P live streaming system, a load balancing protocol, and an anonymous communication system. The evaluation of the use cases, performed using the state-of-the-art simulator PeerSim, shows that the cooperative systems designed using *RACOON++* achieve selfishness-resilience and high performance. The *RACOON++* framework is provided as a Java program, and is freely available for download [33].

Our future work includes the integration of a domain-specific language into *RACOON++* to specify more complex selfish behaviours, such as the one we proposed in [42], and the investigation of other mechanisms to foster cooperation (e.g., decentralised credit-based systems).

References

[1] Cisco Systems. "Cisco Visual Networking Index: Forecast and Methodology, 2015-2020." A Cisco White Paper, 2016.
[2] X. Liu *et al.* "A case for a coordinated internet video control plane." In *SIGCOMM*, 2012.
[3] N. Anjum, D. Karamshuk, M. Shikh-Bahaei, N. Sastry. "Survey on Peer-assisted Content Delivery Networks." *Computer Networks*, 2017.
[4] M. Swan. "Blockchain: Blueprint for a new economy." O'Reilly Media, Inc., 2015.
[5] R. Want, B. N. Schilit, S. Jenson. "Enabling the internet of things." In *IEEE Computer*, 48(1), 2015.
[6] G. Ziccardi. "Resistance, liberation technology and human rights in the digital age." *Springer Science & Business Media*, 2012.
[7] I. Cunha, E. C. Miguel, M. V. Rocha, *et al.* "Can peer-to-peer live streaming systems coexist with free riders?" In *P2P*, 2013.
[8] M. Zghaibeh, H. C. Fotios. "Revisiting free riding and the Tit-for-Tat in BitTorrent: A measurement study." *Peer-to-Peer Networking and Applications*, 1(2), 2008.
[9] D. Hughes, G. Coulson, J. Walkerdine. "Free riding on Gnutella revisited: the bell tolls?" *Distributed Systems Online, IEEE*, 6(6), 2005.
[10] M. Jelasity, A. Montresor, O. Babaoglu. "A modular paradigm for building self-organizing peer-to-peer applications." In *Internation Workshop on Engineering Self-Organizing Applications*, 2003.
[11] B. Cohen. "Incentives build robustness in BitTorrent." Workshop on Economics of Peer-to-Peer systems, 2003.



(a) Cooperation Level (b) Chunk Loss in LS (c) CoV of load in LB (d) Onion Loss in AC

Fig. 13: Experiment results with different proportions of message loss.

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

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

[14] R. Guerraoui, K. Huguenin, A-M. Kermarrec, *et al.* "LiFTinG: Lightweight freerider-tracking in gossip." In *Middleware*, 2010.

[15] D. Goldschlag, M. Reed, P. Syverson. "Onion routing." *Commun. of the ACM*, 42(2), 1999.

[16] I. Abraham, D. Dolev, R. Gonen, J. Halpern. "Distributed computing meets game theory: robust mechanisms for rational secret sharing and multiparty computation." In *PODC*, 2006.

[17] J. Kats. "Bridging game theory and cryptography: Recent results and future directions." In *Theory of Cryptography*, Springer, 2008.

[18] J. Freudiger, M. H. Manshaei, J-P. Hubaux, D. C. Parkes. "On non-cooperative location privacy: a game-theoretic analysis." In *CCS*, 2009.

[19] 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 *SRDS*, 2015.

[20] L. Buttyán, *et al.* "Barter trade improves message delivery in opportunistic networks." *Ad Hoc Networks*, 8(1), 2010.

[21] R. Trestian, O. Ormond, G.M. Muntean. "Game theory-based network selection: solutions and challenges." *IEEE Commun. Surveys and Tutorials*, 14(4), 2012.

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

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

[24] H. C. Li, A. Clement, M. Marchetti, *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, M. Dahlin. "BAR gossip." In *OSDI*, 2006.

[26] R. T. Ma, S. Lee, J. Lui, D. K. Yau. "Incentive and service differentiation in P2P networks: a game theoretic approach." IEEE/ACM Transactions on Networking, 14(5), 2006.

[27] E. Palomar, A. Alcaide, A. Ribagorda, Y. Zhang. "The peers dilemma: A general framework to examine cooperation in pure peer-to-peer systems." *Computer Networks*, 56(17), 2012.

[28] Y. Wang, *et al.* "P2P soft security: On evolutionary dynamics of P2P incentive mechanism." *Computer Communications*, 34(3), 2011.

[29] B. Q. Zhao, J. Lui, D-M. Chiu. "A mathematical framework for analyzing adaptive incentive protocols in P2P networks." In *Networking, IEEE/ACM Transactions on*, 20(2), 2012.

[30] R. Mahajan, M. Rodrig, D. Wetherall, J. Zahorjan. "Experiences applying game theory to system design." In *SIGCOMM*, 2004.

[31] R. B. Myerson. "Game theory." *Harvard University press*, 2013.

[32] J. Weibull. "Evolutionary game theory." MIT press, 1997.

[33] "The RACOON++ Framework." "https://github.com/glenacota/racoon"

[34] A. Yumerefendi, J. S. Chase. "The role of accountability in dependable distributed systems." In *HotDep*, 2005.

[35] M. Belenkiy, *et al.* "Making P2P accountable without losing privacy." *Proc. of the ACM Workshop on Privacy in Electronic Society*, 2007.

[36] F. D. Garcia, J. H. Hoepman. "Off-line karma: a decentralized currency for peer-to-peer and grid applications." *ACNS*, 2005.

[37] S. Marti, H. Garcia-Molina. "Taxonomy of trust: Categorizing P2P reputation systems." *Computer Networks*, 50(4), 2006.

[38] K. Hoffman, D. Zage, C. Nita-Rotaru. "A survey of attack and defense techniques for reputation systems." *ACM Computing Surveys*, 42(1), 2009.

[39] S. Naicken, B. Livingston, A. Basu, *et al.* "The state of peer-to-peer simulators and simulations." In *SIGCOMM*, 2007.

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

[41] L. Leonini, É. Rivière, P. Felber. "SPLAY: Distributed Systems Evaluation Made Simple (or How to Turn Ideas into Live Systems in a Breeze)." In *NSDI*, 2009.

[42] 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 *DSN*, 2017.

Guido Lena Cota Guido Lena Cota is a Ph.D. in computer science, co-supervised by Università degli Studi di Milano (Italy) and INSA Lyon (France). His research interests are in the areas of dependable and selfishness-resilient cooperative distributed systems. Lena Cota is member of the IRIXYS Int. Research and Innovation Center.

Sonia Ben Mokhtar Sonia Ben Mokhtar is a CNRS researcher at the LIRIS lab, in the DRIM group, since October 2009. Before that, she was a research associate at University College London (UCL) for two years, working with Licia Capra. She received her Ph.D. in 2007 from University Pierre et Marie Curie (Paris 6), which she did under the supervision of Valrie Issarny and Nikolaos Georgantas in the INRIA ARLES project-team. Her research interests include Reliable Distributed Systems and Middleware for Mobile Environments. Ben Mokhtar is member of the IRIXYS Int. Research and Innovation Center.

Gabriele Gianini Gabriele Gianini is an assistant professor in the Department of Computer Science of Università degli Studi di Milano. His research interests include game theoretic applications to networking and security, quantitative modeling of processes, and statistical and soft computing techniques. Gianini received a PhD in physics from Università degli Studi di Pavia. Gianini is a coordinator of the IRIXYS Int. Research and Innovation Center.

Ernesto Damiani Ernesto Damiani is a full Professor at Università degli Studi di Milano and the Director of the Information Security Research Center at Khalifa University, Abu Dhabi. His research interests include cloud assurance, Web services and business process security, and Big Data processing. Damiani received a PhD in Computer Science from Università degli Studi di Milano, and a doctorate honoris causa from INSA Lyon. Damiani is a leading scientist of the IRIXYS Int. Research and Innovation Center.

Julia Lawall Julia Lawall is a Senior Researcher at Inria. Her research interests include program analysis and language design, applied to systems code. She received a PhD in Coputer Science at Indiana University and was previously on the faculty of the University of Copenhagen.

Gilles Muller Gilles Muller received the Ph.D. degree in 1988 from the University of Rennes I, and the Habilitation a Diriger des Recherches degree in 1997 from the University of Rennes I. He is currently a senior research scientist at Inria Paris and the head of the Whisper group. His research interests include the development of methodologies based on domain-specific languages for the structuring of infrastructure software.

Lionel Brunie Lionel Brunie is a Professor at the National Institute of Applied Sciences of Lyon, France. His main research interests include data management and resilience in distributed systems, and information privacy and security. Brunie is a leading scientist of the IRIXYS Int. Research and Innovation Center.