A Semi-Automatic Framework for the Design of Rational Resilient Collaborative Systems

Guido Lena Cota  Pierre-Louis Aublin  Sonia Ben Mokhtar  Gabriele Gianini  Ernesto Damiani  Lionel Brunie
Univ. of Milano  INSA Lyon  LIRIS - CNRS  Univ. of Milano  Univ. of Milano  INSA Lyon

Abstract—Collaborative systems (e.g., P2P instant messaging, file sharing, live streaming applications) constitute the largest traffic of today’s Internet. Common to all these systems is the assumption that, in return to the service offered by the collaborative system, users are willing to participate by sharing their resources with others. However, in practice, these systems suffer from rational users, i.e., users that benefit from the system without contributing their fair share to it. A number of solutions have been devised in the literature to deal with the problem of rational users in collaborative systems. However, most of these solutions are tailored to specific systems and thus lack flexibility and re-usability. In this paper, we propose RACOON, the first framework for RAtional resilient COLlabOrative system design. RACOON relies on an extensible model that allows a system designer to specify the protocol steps and the different types of rational users he wants to consider. Furthermore, RACOON relies on game theory to reason on the behaviour of rational users. Finally, RACOON includes a simulation module that allows performance-oriented tuning of the system. Throughout the paper, we show how we used RACOON for the design of a rational-resilient, collaborative live-streaming application. Performance evaluation realised on one hundred real machines shows that the configuration proposed by RACOON allows all users to visualise a quality stream even in presence of rational users.

I. INTRODUCTION

According to numerous studies (e.g., [9], [20], [5]), collaborative systems, also called peer-to-peer systems, account for the largest traffic of today’s Internet. These systems include P2P instant messaging and voice over IP (e.g., Skype), P2P file sharing (e.g., Ares, BitTorrent, eDonkey), P2P live streaming (e.g., P2PTV, PPLIVE). The success of these systems mainly resides in their attractive properties among which the robustness to failures, the scalability to millions of users and the unnecessary to maintain costly dedicated servers as cost is shared among the participating users.

Common to all these systems is the assumption that, in return to the service offered by the collaborative system, users commit to sharing their (communication and computational) resources with others. However, in practice, these systems suffer from rational users, i.e., users that benefit from the system without contributing their fair share to it.

A number of solutions have been devised in the literature to deal with the problem of rational users in collaborative systems. Most of these systems rely on game theory by including incentives for rational users to collaborate with other users. Examples of such systems include rational-resilient live streaming [12], [6], rational-resilient anonymous communication [23], [3], rational-resilient spam filtering [2]. To design these systems, experts perform a manual analysis of all the possible rational deviations. Then, they augment the original protocol with incentives in such a way that it is not in the interest of rational users to deviate from the system specification. However, these systems suffer from a number of limitations. First, the analysis of rational deviations and the integration of the corresponding incentives are performed manually by the system designers, which is complex and error prone. Second, these systems are hardly extensible. Indeed, any modification in the original system requires to rethink the system as a whole, as modifications may introduce new rational deviations. Third, these solutions often lack flexibility as designers often assume that all rational users follow the same rationality model (i.e., they all behave similarly in the same conditions). Finally, in order to force rational users to always stick to the protocol specification, most of existing solutions heavily rely on cryptographic-based fault-detection mechanisms, which results in poor performance.

In this context, the challenge is to devise a framework to assist system designers in the performance-oriented design of robust and flexible rational resilient distributed systems.

In this paper, we embrace this challenge and propose RACOON, the first framework for RAtional resilient COLlabOrative system design. To reach this objective, RACOON is composed of three parts. First, RACOON includes an extensible model that allows a system designer to describe the interaction patterns between users using state machines, and the different types of rational users he wants to consider. This model also allows a system designer to plug inspection mechanisms at specific points of the protocol in order to verify whether users participating to the protocol effectively stick to its specification or not. Second, RACOON includes a game theoretic module that allows reasoning on the behaviour of rational users when the latter are confronted to the inspection mechanisms. Third, RACOON includes a simulation module that uses the game analysis to provide the designer with performance-oriented guidelines for the fine tuning of its system (e.g., the penalty/rewards that users obtain when their behaviour appears as faulty/correct after inspection).

Throughout the paper, we show how we used RACOON for
the design of a rational-resilient, collaborative live-streaming application. Performance evaluation realised on one hundred real machines shows that the augmented system proposed by RACOON allows all users to visualise a quality stream in presence of rational users.

The remaining of the paper is organized as follows. First, Section II presents the problem statement and the goals and non-goals of our framework. Then, Section III presents the overview of RACOON while Section IV presents its detailed description. Further, Section V presents the performance evaluation of RACOON. Finally, Section VI discusses related work, and Section VII concludes this paper.

II. PROBLEM STATEMENT, GOALS AND NON-GOALS

Collaborative systems are systems in which users cooperate to realise a mutually beneficial service (e.g., overlay routing [8], cooperative backup [14], file sharing [24], live streaming [12], [6]). In addition to classical failures (e.g., crash of machines, malicious users), collaborative systems are subject to rational behaviour, i.e. users that benefit from the system without contributing their fair share to it. The rationality of a user is often defined by a utility function, i.e. a mathematical representation of its preferences (e.g., minimizing its bandwidth consumption) over a set of behavioural choices (e.g., replying or not replying to a given request).

In this context, our aim is to propose a framework to assist system designers in the design and evolution of collaborative systems that are robust to rational users.

To better illustrate the objectives of our framework, let us consider a simple protocol P1 in which a user u0 sends a request message m0 to a user u1. Upon receiving m0, u1 performs some local computation and replies to u0 by sending a message m1. In P1, the rationality of users can take different forms. For instance, a rational user u1 can omit to send a response to m0 to save bandwidth (i.e. communication-related rationality). Alternatively, to avoid being harassed by u0 which might retransmit its request, u1 can skip the computations necessary to compute m1 and send instead a wrong response to m0 (i.e. computation-related rationality). These two different types of rationalities require the system designer to include different types of incentives/counter measures. Furthermore, while it is often assumed in the literature that all rational users have the same rationality, the reality is different as many different types of rational users can co-exist in a collaborative system.

The first objective of RACOON is thus to allow a system designer to model different types of rationalities and to automatically propose incentives/counter measures adapted to each of them.

A classical solution to reason on rational behaviours is to rely on game theory and specifically on the concept of Nash equilibrium [17]. This concept defines a strategy for each participant in the game (i.e. each user in the collaborative system) from which there is no benefit in deviating. Many collaborative systems are thus designed (from scratch) to be a Nash equilibrium (e.g., [12], [2], [3], [13]) by embedding incentives on each protocol step in such a way that it is not in the interest of rational users to behave rationally. However, these systems are hardly extensible as any modification in the original protocol may introduce new rational deviations and thus break the Nash equilibrium.

The second objective of RACOON is thus to introduce automation in the process of devising the rational resilient system. This enables more robustness and the easy evolution of the resulting collaborative system.

Among the classical incentives to force rational nodes to stick to a protocol is to integrate inspection mechanisms (e.g., [7], [1]). For instance, in the protocol P1 introduced above, each user could be associated with a secure log in which it writes all its interactions with other users. Periodically, each user can inspect the log of its partners to verify that their behaviours correspond to a correct execution of the protocol. However, these mechanisms are very costly especially if inspections are performed regularly (e.g., to deter faults faster). Moreover, a realistic collaborative system should accept the presence of some uncooperative behaviours. For example, users may deviate in good faith, because of limited resources or inexperience. In these situations, it may be beneficial to relax the security properties of the system.

The third objective of RACOON is thus to allow the system designer to set up inspection mechanisms according to the rationality models he wants to consider on the one hand and to the performance he wants to reach on the other hand.

Finally, we also list some non-goals in order to help position our contribution. First, we do not aim at outperforming established context-specific solutions for dealing with rational users. Instead, we aim at providing software designers with a framework to easily design and maintain a rational resilient collaborative system. Second, RACOON does not generate executable code. Third, RACOON does not enable on-line analysis of software systems. All the analysis performed by RACOON are done offline at design time.

III. RACOON OVERVIEW

RACOON reaches the objectives specified above by relying on classical building blocks for the design of rational-resilient collaborative systems. Specifically, RACOON specifies the communication protocols among users using finite state machines. It predicts rational behaviours using game theory. Further, it makes deviations unprofitable using inspection mechanisms. Finally, it tests the achievement of performance and security requirements using simulations.

Fig. 1 shows the four phases that a system designer follows when using our framework. The first phase, i.e. the specification phase, is manually performed by the system designer. In this phase, the designer specifies the system following the model presented in Section IV-A. This model allows the specification of the interaction patterns between users, the types of rationalities investigated, and the configuration of the security mechanisms. Once the first phase has been completed, then RACOON proceeds automatically. The second phase depicted in the figure is the augmentation phase. In this phase,
the specification produced by the designer is automatically augmented with rational deviations. The automatic generation of deviations applies the model of rationality defined by the designer in the specification phase (see Section IV-B1). The original specification is also augmented with the security modules, which are: (i) inspection methods to expose rational deviations, and (ii) an incentive system to reward users when they behave cooperatively, and punish them when they do not. We present these modules in Section IV-B2. Then, the framework proceeds with the third phase, i.e. the gamification phase, which aims at predicting the behavioural choice of a user in a certain system set-up. In other words, RACOON estimates the most favourable behaviour among all the possible candidates from the augmented protocol. We achieve this with a game theoretic approach. Game theory is indeed the most appropriate tool to predict the interactions among rational decision-makers. The gamification phase carries out the game theoretic analysis by transforming the augmented protocol into a suitable game representation (see Section IV-C1). The analysis leads to a Nash Equilibrium [17], which we use to predict rational behaviours. Finally, the last phase, i.e. the simulation phase, tests whether the rational-resilience and performance requirements are satisfied. The simulator uses the augmented protocol and the game theoretic predictions to produce a set of measures. The designer compares these measures to determine if the system specification is worth implementing or not. If not, RACOON provides insights to help the system designer to find the right parameters to meet its performance requirements.

IV. RACOON DETAILED DESCRIPTION

In this section we present RACOON in detail, following the sequencing of phases introduced in the previous section. To better illustrate the various parts of RACOON with concrete examples, we start by introducing a case study, which is the design of a P2P live-streaming application.

Case Study: P2P Live Streaming

The P2P live-streaming system consists of a stream source and an audience of \( n \) users (peers). The source of the streaming session encodes the video, cuts it into pieces (chunks) and periodically broadcasts \( c \) of them to a set of users. At each period of time, the broadcast reaches a fraction \( a \) of the users in the network. We assume the source is cooperative, while a proportion \( r \) of users can be rational. Each user must receive chunks within a deadline \( d \) to be played out (play-out delay). An expired chunk is equivalent to a lost chunk, because it can not be reproduced. In order to support the media distribution process, users share chunks with each other using a content dissemination protocol. The upload bandwidth of users is limited to \( b \) chunks/round. We consider the three-phase gossip protocol used in [6]. Every \( p \) rounds (which is the period), users establish a partnership with \( f \) others selected uniformly at random. Then, they propose their not expired chunks to all of them. Fig. 2(a) shows the propose phase, where user \( u_0 \) sends a list of chunk’s identifiers \( (c_0, c_7, c_9) \) to its 3 partners \( (u_1, u_2, u_3) \). As soon as the propose has been received, the partners request the chunks they miss (request message). Eventually, the proposing user sends the payload of the requested chunks (serve message). Fig. 2(b) illustrates the state machine of the above protocol, where user \( u_1 \) requests the chunks 7 and 9 from \( u_0 \). The label of each transition defines the user’s ID, the name of the method invoked and the ID of the message sent.

A. Specification

In this phase, the designer defines the collaborative system. The framework provides a specification model to guide this task. The output is an instance of the specification model characterizing the target collaborative system.

The designer specifies its collaborative system following the UML diagram depicted in Fig. 3. Each element in this diagram describes characteristics or requirements of the system. We classify them in four logical areas: the User domain, the Rationality domain, the Security domain, and the System domain. The white elements in Fig. 3 are application dependent, whereas the coloured ones can be re-employed in different contexts. This approach improves the reusability, and enables the automation of the next phases of the framework. The specifications are encoded as a machine readable XML-based format. The XML Schema of each element can be found in the companion technical report [11]. In the following, we detail the four specification domains.

User domain defines the user and the interaction protocols
RACOON Specification Model of a collaborative system.

in which it participates. The Role element describes the part
that a user may undertake in a protocol. For instance, in
our case study, one can participate to the gossip protocol
as a sender or as a receiver. The interaction among users is
represented as deterministic finite State Machine, like the one
shown in Fig 2(b). We assume no loops in the state machine.
A State represents a stage of the protocol, while Transitions
are the operations (e.g., communications or computations)
that enable its evolution. An operation corresponds to the
invocation of a set of methods. The Interface element collects
all the methods that a transition must implement. The same
method can describe the functionality that a user provides
or uses, according to the role that it plays in the protocol.
For example, the method propose in Fig. 2(b) is provided
by $u_0$ (sender) and used by $u_1$ (receiver). The designer also
declares possible relationships among contents, and provides
their XML Schema type.

Rationality domain The rationality of a user is defined
by its utility function, that is specified in the Utility Profile
element. Note that the same user may have several Utility
Profiles, one for each role it undertakes in a certain protocol. In
order to simplify the definition of a utility function, we propose
to build it by enabling Rationality Model elements, which
identify specific types of rational behaviours. This encourages
the reuse of already defined elements, and simplifies the
analysis of the system under new rationality assumptions. The
Utility Profile and the Rationality Models are independent
from the specific application. Finally, in the Rationality Conf
element the designer specifies the models of rationality to
activate, along with the types of deviation to generate. For
our case study, we have implemented the Communication
Rationality model, a specialization of the Rationality Model
element. We discuss it in Section IV-B1.

Security domain It specifies the modules that provide
resilience from rational behaviours. They are defined as imple-
mentation of the abstract Security Layer element, so that they
can be easily replaced with different modules. In this paper,
we use the Inspection module to detect deviations, and the
Reputation module to enforce cooperation. The Security Conf
element allows to configure both. We defer a description of
the security modules and their parameters in Section IV-B2.

System domain It specifies system specific information,
which are necessary to set-up the simulations. In particular,
the designer specifies the proportion of rational users in the
system, possibly all of them. In our use case, some example
of these are the number of users in the system, the length
of the stream, and the probability of message loss in each
communication. The code in Listing 1 presents an extract of
the User domain specification of our case study. The complete

B. Augmentation

The goal of this phase is to include rational behaviours
and security mechanisms into the protocol’s representation.
The result, called augmented protocol, details how the
collaborative system changes in presence of rational users.

1) Rational Deviations: A rational deviation is a violation
of the protocol that aims at maximizing a utility function.
In RACOON, utility functions are defined by activating a
Rationality Model created by the designer. This is a set of
algorithms that realize a specific model of rationality (e.g.,
saving bandwidth consumption, reducing computational costs).
Each algorithm takes as input the User domain specification
of the system, and describes the procedure to derive rational
deviations. RACOON provides an automatic tool that executes
the algorithms of the Rationality Models enabled by the
designer. This process results in the generation of new states
and transitions, which the automated tool adds to the original
state machine. Notice that each new transition is created along
with a new interface, which contains at least one deviation. The
resulting state machine is called augmented automaton, where
every execution path (i.e. the path that connects the initial state
to a final state) represents a possible user’s behaviour.
Listing 1: User domain specification of the state machine shown in Fig. 2(b).

```xml
<userDomain>
  <roles>
    <role name="sender" />
    <role name="receiver" />
  </roles>
  <states>
    <state name="s0"> <initial /> </state>
    <state name="s1" />
    <state name="s2" />
    <state name="s3"> <final /> </state>
  </states>
  <transitions>
    <transition name="t0" interface="i0">
      <fromState="s0" toState="s1">
        <fromRole>sender</fromRole>
        <toRole>receiver</toRole>
      </transition>
    </transitions>
  </interfaces>
</userDomain>
```

We provide as part of the RACOON framework the specification of the Communication rationality model. Its utility function aims at minimizing the communication costs for rational users. We define communication costs in terms of bandwidth consumption, which depends on the length and on the type of the transmitted contents. From the three-phase gossip protocol of the case study, we can identify three rational deviations consistent with this model: (d1) decreasing the number of partners, so that a sender receives a lower number of requests, (d2) proposing less chunks than what a sender holds, and (d3) sending only a subset of the requested chunks. Based on these insights, we have designed three rational deviations intended to reduce communication costs:

i) Timeout Deviation. The user does not perform the prescribed operation within the necessary time frame. For example, it does not send any chunks proposal, or does not reply to a request. The algorithm: for each non-final state \( s \) of the protocol, checks whether it has at least one outgoing transition that executes a legal timeout. If so, skip to the next state. Otherwise, create a new final state \( s_{to} \), and connect it with \( s \) by a timeout deviation.

ii) Subset Deviation. The user sends only a subset of the legal content. This behaviour covers the deviations (d1) and (d2) described above. The algorithm: for each non-final state \( s \) of the protocol, consider the set of its outgoing transitions \( T \). If a transition \( t \in T \) transmits a content with a complex type (according to the XML Schema type definition), then create a new state \( s_{sub} \) and connect it with \( s \) by a subset deviation \( t_{sub} \). The content transmitted in \( t_{sub} \) must be a subset of the legit one. Fig. 4 illustrates the procedure. The illegal states are represented with filled circles, and labels of deviations are in italics. Note that the message sent by a subset deviation is indicated by an exclamation mark (!) before its ID.

iii) Multicast Deviation. The user does not perform a certain operation with all the prescribed partners, but only with a subset of them. For instance, a user can decrease the size of its partnership, like described in (d3). The algorithm: for each non-final state \( s \) of the protocol, consider the set of its outgoing transitions \( T \). If a transition \( t \in T \) sends a message to more than one recipient, then create a new state \( s_{mul} \) and connect it with \( s \) by a multicast deviation \( t_{mul} \). The recipients of \( t_{mul} \) must be less than the legit ones. The procedure is shown in Fig. 5. For reason of clarity, we write the recipient of a transition at the end of each label, after a semicolon (;). The legit set of recipients is indicated with \( R \), while the non-legal one with \( !R \).

Notice that once an illegal state is created, RACOON updates its connections with the other states of the automaton. The generality of the algorithms makes the Communication Rationality model applicable to any collaborative systems. Fig. 6 shows the three-phase gossip protocol augmented with timeout and subset deviations. For clarity we do not represent multicast deviations. Consider for instance the state \( s_2 \). In the correct execution of the protocol, the user \( u_0 \) sends a list with the payload of the requested chunks \( m_2 \) to user \( u_1 \). However, a rational user may also timeout the request, or serve a list with less chunks \( m_2 \). In both cases, the protocol terminates in an illegal state \( (s_2 \) and \( s_2 \), respectively), and \( u_1 \) does not obtain what it requested.

2) Security Layer: RACOON forces collaboration by making rational users accountable for their actions. This is realized by employing a security layer, which seamlessly extends the logic of the original protocol. The security layer implements an inspection module to detect rational deviations, and a reputation module to assign positive or negative feedback on the reputation of users. The deployment of the security mechanisms on an augmented automaton results in an augmented protocol, which is also the outcome of this phase of
In Section IV-C1 we discuss how to integrate such mechanisms into the utility function. The designer can configure the reputation module by setting the range of values for the reputation.

C. Gamification

The augmented protocol is transformed into a game. A game theoretic analysis is then performed to predict the behaviour of rational users. Such prediction is the output of this phase.

We use game theory to assess the likelihood of rational deviations from the augmented protocol. In fact, game theory is the natural framework to model the interplay of rational users in competitive systems [19]. RACOON provides an automatic tool to transform any augmented protocol into a game. The tool also performs the game theoretic analysis, that reveals the expected behaviour of the rational users.

1) Game Mapping: We model a collaborative protocol as a non-cooperative game with complete information [17]. A non-cooperative game describes the interactions among rational decision-makers (players). A player represents a role played by a user in the protocol, and is associated with a set of possible actions. Each action corresponds to a method of the augmented protocol, possibly a deviation. As the augmented protocol defines a certain order of execution of methods, we map it into a game with sequential moves. When a player performs an action, it does not know if it is responding to a deviation unless it performs an inspection. The information available to a player at a given point in the game form an information set. The sequence of actions that a player executes in a game is called a strategy, while a set of joint strategies (one for each player) is called a strategy profile. A strategy profile is called a Nash Equilibrium (NE) if no player, when finding himself in that strategy profile, has incentive in deviating unilaterally from its strategy: a NE conventionally characterizes in game theory the expected behaviour of the rational users.

The augmented protocol is transformed into a game. A game theoretic analysis is then performed to predict the behaviour of rational users. Such prediction is the output of this phase.

We use game theory to assess the likelihood of rational deviations from the augmented protocol. In fact, game theory is the natural framework to model the interplay of rational users in competitive systems [19]. RACOON provides an automatic tool to transform any augmented protocol into a game. The tool also performs the game theoretic analysis, that reveals the expected behaviour of the rational users.

1) Game Mapping: We model a collaborative protocol as a non-cooperative game with complete information [17]. A non-cooperative game describes the interactions among rational decision-makers (players). A player represents a role played by a user in the protocol, and is associated with a set of possible actions. Each action corresponds to a method of the augmented protocol, possibly a deviation. As the augmented protocol defines a certain order of execution of methods, we map it into a game with sequential moves. When a player performs an action, it does not know if it is responding to a deviation unless it performs an inspection. The information available to a player at a given point in the game form an information set. The sequence of actions that a player executes in a game is called a strategy, while a set of joint strategies (one for each player) is called a strategy profile. A strategy profile is called a Nash Equilibrium (NE) if no player, when finding himself in that strategy profile, has incentive in deviating unilaterally from its strategy: a NE conventionally characterizes in game theory the expected behaviour of the rational players [15]. Finite games (finite number of players, finite number of strategies) are granted to always have a NE [17]. The goal of a player is to follow the strategy that maximizes its utility function. The assumption of rationality requires that such choice takes also into account the potential strategies of other players. Indeed, all the factors of the game (i.e. players, order of moves, possible strategies, utility function) are common knowledge.

All these conditions result in a sequential game with complete but imperfect information, which is usually represented by its extensive-form, also called game tree [19]. RACOON derives the game elements from the augmented protocol specification. Table I summarizes the rules to map the latter into a game tree. For instance, a player in the game tree corresponds to a role played by a rational user in the augmented protocol. The utility \( \bar{u}_i \) for player \( p_i \) while playing the strategy \( \sigma_i \) is a function of the operational costs incurred to implement the strategy and the incentives received from the security mechanisms. It can be expressed as:

\[
\bar{u}_i(\sigma_i) = \sum \text{cost}_i(\sigma_i) + \gamma(\text{oldRep}_i, \text{newRep}_i),
\]
where \( \gamma \) integrates the reputation values into the utility function. The intuition behind \( \gamma \) is that the 'shadow of the future' affects the perception of today’s utility. The function \( \gamma \) formulates such effect in terms of additional costs or discounts. More specifically, when a player expects a punishment (the new reputation \( \text{newRep} \) is lower than \( \text{oldRep} \)), \( \gamma \) identifies two types of cost:

- Eviction cost: the cost for being expelled from the system, which is paid only if the new reputation is below a given threshold. It is conventionally set at a very high nominal value, since an eviction is the worst utility a player may have.
- Redemption cost: the cost for regaining the old reputation. It is the product between the profit margin of a deviation (i.e. the cost saved by performing a certain deviation) and the redemption time (i.e. the expected time during which the player must behave correctly).

In contrast, if a player expects a reward (\( \text{newRep} \) is greater than \( \text{oldRep} \)), then the function \( \gamma \) identifies a cost discount. This is because the convenience of a future deviation becomes more important as the reputation value increases. In other words, the benefit of the profit margin of tomorrow is higher than the cost of possible punishments. Fig. 7 illustrates the result of the game mapping of the augmented protocol shown in Fig. 6. Each leaf of the game tree is associated with a pair of payoffs, one for each player. For reason of space, they are not reported in the figure. Moreover, details on the algorithm that augments the protocol with rational deviations are omitted but can be found in the companion technical report [11].

2) Game Analysis: The gamification tool of RACOON can integrate several solution concepts for the game theoretic analysis [18]. For the sake of simplicity, and without loss of generality, the solution concept we use for this paper is the Subgame-Perfect equilibrium (SPE) in pure strategies [19]. It is a refinement of NE for extensive-form games. Subgame perfection ensures that no player has an incentive to change the equilibrium strategy as the game progresses. We assume that if a game has a SPE, then all rational players will follow the equilibrium strategy. If no equilibrium in pure strategies is found, then we assume that rational players will withdraw from the protocol (worst case). To find the SPE of the protocols gamified in the previous step, RACOON uses Gambit\(^1\), an open-source library of tools for game-theoretic analysis. The algorithm to find the SPE is described in [18].

D. Simulations

We test whether the augmented protocol achieves the designer’s performance requirements. To this end, we simulate the system using the predictions of the gamification phase.

The solution of the gamified protocol represents a "snapshot" of the system evolution. In fact, it holds only for a particular configuration of factors (e.g., the security setting, the current reputation of players). Therefore, the predictive power of a single game solution is too narrow to give a full picture of the users’ dynamics. The last phase of RACOON framework addresses this issue by adopting a simulation approach. Simulations provide a practical evaluation of the security configuration, by reproducing how users behave within a given system set-up. Another direct advantage of such pragmatic approach is that it enables to study how different degrees of tolerance to rational deviations affect the final outcome.

A prerequisite for the simulation phase is to have a simulation environment for the system under consideration. With this in mind, we have developed a Java-based simulator that supports cycle-based applications over a generic P2P overlay network. The simulation workflow proceeds in four steps:

1) Parse the system specifications and configure the simulation setup accordingly.
2) Partition the system dynamics into behavioural units. Each behavioural unit corresponds to the particular reputation setting in which two or more users may interact. Behavioural units are the building blocks to generate every possible evolution of the system.
3) Gamification of the behavioural units as independent games, which results in a broad range of game solutions. These represent the predictions of selfish users behaviours, that tell the simulator how to progress.
4) Run the simulation, while collecting the information that is relevant to the performance evaluation.

The simulation’s result determines whether the configuration of the security modules is satisfactory or not. A satisfac-

---

\(^1\)GAMBIT: http://sourceforge.net/projects/gambit/
tory configuration must guarantee good performance even in presence of rational users. If not, then the designer can set different security settings and restart the simulation. If instead the system achieves the desired performance, then the designer can implement the collaborative system that has been designed and tested using RACOON.

V. EVALUATION

We present in this section the evaluation of our RACOON framework. This evaluation divides in three parts. In order to better understand the scope of the problem posed by the presence of rational users, the first part, presented in Section V-A, shows the impact of these users in the collaborative live streaming application introduced in Section IV. Then, the second part, presented in Section V-B, focuses on the effort required by a software developer to make the above live streaming application resilient to rational users using our framework. Furthermore, in this part, we show how RACOON helps the software developer to easily update its application if he wants to consider a new rationality model or change some protocol steps.

Finally, the third part, presented in Section V-C, shows how the simulation part of our framework helps the software developer in choosing the right configuration for its live streaming application considering a list of performance requirements. Also, in this part we show that the configuration chosen by the software developer after the simulation part, implemented and deployed on 100 users running on real machines, exhibits the expected performance.

A. Impact of Rational Users on the Live Streaming Use Case

To measure the impact of rational users on the live streaming use case presented in Section IV we implemented this application and deployed it on a Grid’5000 cluster\textsuperscript{2} of 10 eight-core physical machines. Each machine is clocked at 2.5GHz with 32GB of RAM, and is interconnected with the others via a Gigabit switch. We then deployed 100 users, and we studied different proportions of rational users ranging from 0 to 100%. These users have two possible rationalities, according to the role they play in the protocol. A receiver performs only subset deviations, whereas a sender also performs timeout and multicast deviations. In fact, senders carry most of the cost of collaboration, but only the receivers benefit from it. We measure the percentage of video chunks lost by users on average. According to study performed in [21], if this value is lower than 3%, users do not notice any degradation in the quality of the received video. Fig. 8 presents the results of our experiment. We observe in this figure that, as expected, the percentage of lost chunks without any rational-resiliency mechanism (i.e. curve G5K w/o RACOON) increases as the percentage of rational users increases. Further, this curve shows that in presence of even 10% of rational users, the quality of the stream is unvisualisable for all users. In addition to this curve, Fig. 8 shows the performance of our rational-resilient live streaming application as simulated by RACOON (i.e. the SIM w/ RACOON curve), as well as a real implementation of it (i.e. the G5K w/ RACOON curve). These two curves show that, thanks to RACOON, we were able to build a rational resilient live streaming application in which even in presence of 100% rational users all users are able to visualise a quality stream. In the remaining of this section, we show how RACOON was used to reach this objective.

B. Using RACOON for the Design and Evolution of the Live Streaming Application

We now describe the effort required by a software developer to design, implement and maintain the rational resilient live streaming application corresponding to our case study using RACOON. Table II shows the size of the XML description, the size of the simulation code and the size of the real implementation that the software developer has to write to describe and implement its application. For each of these three parts, we distinguish between the number of lines that are specific to the use case application and those that are generic and thus reusable from one application to another. From this table, we observe that the amount of reusable code is at least of 40% (for the XML description) and can be as high as 83% (for the simulation part). This high degree of reusability is enabled thanks to the modularity of RACOON.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
Lines of code & XML description & Simulator & Prototype \\
\hline
All & 140 & 2500 & 3064 \\
Specific & 83 & 410 & 1118 \\
Reusable & 57 & 2090 & 1946 \\
\hline
\end{tabular}
\caption{Number of lines of code and percentage of reusable code in the XML description, the simulator and the prototype.}
\end{table}

We now focus on the evaluation of the ability of RACOON in enabling software evolution. Let us consider that the software developer, which has initially designed its software by assuming a proportion of up to 50% rational users, wants to consider a proportion of 80% instead. Furthermore, let us consider that the designer, which has initially activated timeout deviations only for the role of sender, wants now to consider these deviations also for the role of receiver.

Using RACOON, performing the first modification requires changing one single line in the XML description of the

\textsuperscript{2}Grid’5000: http://www.grid5000.fr
application. In particular, the designer has only to set the rationalFraction property to 0.8 in the following line:

```xml
<property name="rationalFraction">0.8</property>
```

Furthermore, adding the timeout deviation for the receiver role requires adding one line in the Rationality Domain part of the XML description. In the following, the bolder line is the only modification required:

```xml
<deviation name="timeout">
  <role>sender</role>
  <role>receiver</role>
</deviation>
```

After these two modifications performed on the XML description, RACOON proceeds automatically by updating the augmented protocol and the game analysis, which allows to automatically change the results of the simulation.

This shows that using RACOON, the software developer can very easily make its system evolve to adapt to changes into the environment (e.g., updating the rationality model, changing the protocol steps).

C. Building a Rational Resilient Live Streaming Application using RACOON

We show in this section how RACOON helps the software developer in choosing the right configuration for its application using the simulation module.

Let us consider that the software developer has fixed the following performance requirements for his rational resilient live streaming application:

(Req1): Correct nodes are not wrongly evicted by the system, even in presence of (up to 5% of) message loss due to disrupted networking conditions.

(Req2): The overhead of the protocol in terms of bandwidth consumption does not exceed 10% of the bandwidth consumed by the live streaming.

(Req3): Correct nodes do not lose more than 3% of video chunks, which ensures that they watch a good quality stream.

There are two parameters in the application that the developer can tune to get the expected performance. These parameters are: (1) the penalty, which is the value by which the reputation of a user is decreased if its log is inspected and detected as faulty; and (2) the percentage of audits that a receiver performs on a sender with whom it interacts.

To fix these parameters the software developer performs the following simulations, for which the results are depicted in Fig. 9. First, it starts by studying the impact of the penalty on the percentage of eviction of correct users in presence of 5% message loss. From Fig. 9(a), we can observe that the higher the penalty the higher the proportion of evicted correct nodes. This is due to the fact that with a high penalty, it is enough to be subject to the loss of one or two messages in order to be wrongly considered as behaving rationally. From this experiment, the software developer can decide to fix the value of penalty to 8 or lower as this value limits the percentage of evicted correct nodes to less than 1%, which matches its first performance requirement.

Using this value of penalty, the software developer can also notice from Fig. 9(b) that the percentage of evicted rational nodes is at least of 65% (if the probability to audit users is equal to 10%), which is satisfactory. From this curve, the developer can also notice that the higher the probability to audit users, the higher the percentage of evicted rational users. However, intuitively, the designer may be interested in the overhead of performing more inspections (in terms of bandwidth consumption). To help fixing the probability to audit users, the designer can refer to the results of Fig. 9(c), which show the overhead in terms of network traffic with respect to the probability to audit. From this figure the designer may decide to fix this value to less than 20% in order to meet its second requirement.

The last experiment (Fig. 9(d)) shows the quality of the stream perceived by correct users as a function of the percentage of rational users for different values of audit probability. From this experiment, the software developer can finally decide to fix the audit probability to 15%. Indeed, this value is enough to get all correct nodes receive a stream with less than 3% of missed chunks.

Summarising, using the simulations enabled by RACOON, the software developer decides to fix the value of the penalty to 8 and the audit probability to 15%, which allows to satisfy all its performance requirements. As discussed at the beginning of this section and shown in Fig. 8, the performance of the live streaming application when implemented with the above parameters and deployed in a real setting validate the choice prescribed by RACOON.

VI. RELATED WORK

There is a vast amount of literature on designing distributed systems. We approach this overview with our research objectives in mind: resilience to rational behaviours, support to system evolution, and good performance. Table III summarizes these objectives, along with the contributions of the works closest to ours. As we can observe, there is no existing solution that meets all the challenges. The RACOON framework does.

<table>
<thead>
<tr>
<th>Game Theory</th>
<th>Rationality</th>
<th>Evolution</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accountability</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>DSL</td>
<td>-</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>RACOON</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

TABLE III: Summary of the state-of-the-art solutions available to design rational-resilient collaborative systems.

Game theory is recognized as being a good framework to address rational behaviours [12], [2], [3], [13]. The aim of game theoretic approaches is to make cooperation the best rational choice, i.e. a Nash Equilibrium. Unfortunately, their solutions are tailored to specific applications, thus are extremely sensitive to changes in the system conditions. Another well-known criticism of game theoretic solutions concerns
Fig. 9: RACOON results.

their performance, because of the strict requirements imposed by the equilibrium assumptions. For example, [12] provides good rational resilience (along with fault tolerance) in live-streaming application, but it suffers from high bandwidth consumption and low scalability [13]. The semi-automatic procedures adopted by RACOON overcome these limitations.

A rational deviation is a particular type of malicious behaviour. For this reason, accountability systems like [7] seem to be an acceptable solution for the rationality issue. Furthermore, the generality of their specification also meets the challenge of supporting the system evolution. The major pitfall of these approaches is that maintaining accountability incurs a non-negligible cost in terms of performance (i.e. high communication and computational costs of the monitoring system). On the contrary, RACOON enables to take performance requirements into account, finding the right compromise between them and rational resilience. Other shortcomings of accountability methods come from some strong assumptions about the system. For example, they require a quorum of cooperative users to be effective [7]. Further, they are not tolerant to message loss. Using RACOON, the designer can find proper solutions without relying on these assumptions.

Finally, several frameworks [22] and Domain-Specific Languages [10], [4] have been proposed in the literature to ease the task of designing and maintaining a distributed system. It is well known that domain-specific approaches can yield to high performance results. Nevertheless, to the best of our knowledge, none of these solutions considers the problem of rational behaviours. The game theoretic approach of our framework meets this challenge.

VII. CONCLUSION

In this paper we have presented RACOON, a framework for semi-automatic game-theoretic analysis of collaborative systems. RACOON’s objective is to allow the designer to automatically design a rational-resilient version of a collaborative system. For this, RACOON augments a given protocol specification with possible rational deviations using a modular and extensible approach. Then, it relies on game theory to predict the behaviour of rational users. Finally, RACOON provides a simulator to rapidly assess the behaviour and performance of the system. Evaluation, performed using simulations and a real prototype with 100 users, shows that the configuration proposed by RACOON allows all users to visualise a quality stream even in presence of rational users.

Experiments presented in this paper were carried out using the Grid’5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

REFERENCES

APPENDIX A
SPECIFICATION OF THE CASE STUDY

In the following, we present the XML Schema for the RACOON Specification Model. Figure 10 shows a diagram of the schema.

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="systemSpecification">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="userDomain">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="roles">
                                <xs:complexType>
                                    <xs:sequence>
                                        <xs:element name="role" maxOccurs="unbounded" minOccurs="1">
                                            <xs:complexType>
                                                <xs:simpleContent>
                                                    <xs:extension base="xs:string">
                                                        <xs:attribute type="xs:string" name="name" use="required"/>
                                                    </xs:extension>
                                                </xs:simpleContent>
                                            </xs:complexType>
                                        </xs:element>
                                    </xs:sequence>
                                </xs:complexType>
                            </xs:element>
                            <xs:element name="states">
                                <xs:complexType>
                                    <xs:sequence>
                                        <xs:element name="state" maxOccurs="unbounded" minOccurs="1">
                                            <xs:complexType mixed="true">
                                                <xs:sequence>
                                                    <xs:element type="xs:string" name="initial" minOccurs="0" maxOccurs="1"/>
                                                    <xs:element type="xs:string" name="final" minOccurs="0" maxOccurs="1"/>
                                                </xs:sequence>
                                                <xs:attribute type="xs:string" name="name" use="required"/>
                                            </xs:complexType>
                                        </xs:element>
                                    </xs:sequence>
                                </xs:complexType>
                            </xs:element>
                            <xs:element name="transitions">
                                <xs:complexType>
                                    <xs:sequence>
                                        <xs:element name="transition" maxOccurs="unbounded" minOccurs="0">
                                            <xs:complexType>
                                                <xs:sequence>
                                                    <xs:element type="xs:string" name="fromRole"/>
                                                    <xs:element name="toRoles">
                                                        <xs:complexType>
                                                            <xs:sequence>
                                                                <xs:element type="xs:string" name="toRole"/>
                                                            </xs:sequence>
                                                        </xs:complexType>
                                                    </xs:element>
                                                </xs:sequence>
                                                <xs:attribute type="xs:string" name="name" use="required"/>
                                                <xs:attribute type="xs:string" name="interface" use="required"/>
                                                <xs:attribute type="xs:string" name="fromState" use="required"/>
                                                <xs:attribute type="xs:string" name="toState" use="required"/>
                                            </xs:complexType>
                                        </xs:element>
                                    </xs:sequence>
                                </xs:complexType>
                            </xs:element>
                            <xs:element name="interfaces">
                                <xs:complexType>
                                    <xs:sequence>
                                        <xs:element name="interface" maxOccurs="unbounded" minOccurs="1">
                                            <xs:complexType>
                                                <xs:sequence>
                                                    <xs:element name="methods">
                                                        <xs:complexType>
                                                            <xs:sequence>
                                                                <xs:element name="method">
                                                                    <xs:complexType>
                                                                        <xs:sequence>
                                                                            <xs:element type="xs:string" name="name" use="required"/>
                                                                            <xs:element type="xs:string" name="type" use="required"/>
                                                                        </xs:sequence>
                                                                        <xs:attribute type="xs:string" name="name" use="required"/>
                                                                    </xs:complexType>
                                                                </xs:element>
                                                            </xs:sequence>
                                                        </xs:complexType>
                                                    </xs:element>
                                                </xs:sequence>
                                            </xs:complexType>
                                        </xs:element>
                                    </xs:sequence>
                                </xs:complexType>
                            </xs:element>
                            <xs:element name="rationalityDomain">
                                <xs:complexType>
                                    <xs:sequence>
                                        <xs:element name="rationalityModels">
                                            <xs:complexType>
                                                <xs:sequence>
                                                    <xs:element name="rationalityModel">
                                                        <xs:complexType>
                                                            <xs:sequence>
                                                                <xs:element name="deviations">
                                                                    <xs:complexType>
                                                                        <xs:sequence>
                                                                            <xs:element name="deviation" maxOccurs="unbounded" minOccurs="0">
                                                                                <xs:complexType>
                                                                                    <xs:sequence>
                                                                                        <xs:element name="roles"/>
                                                                                        <xs:element name="interface" use="required"/>
                                                                                        <xs:element name="fromRole" use="required"/>
                                                                                        <xs:element name="toState" use="required"/>
                                                                                    </xs:sequence>
                                                                                </xs:complexType>
                                                                            </xs:element>
                                                                        </xs:sequence>
                                                                    </xs:complexType>
                                                                </xs:element>
                                                            </xs:sequence>
                                                        </xs:complexType>
                                                    </xs:element>
                                                </xs:sequence>
                                            </xs:complexType>
                                        </xs:element>
                                    </xs:sequence>
                                </xs:complexType>
                            </xs:element>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```
Fig. 10: The diagram of the XML Schema for the RACOON Specification Model.
APPENDIX B

SPECIFICATION OF THE CASE STUDY

In the following, we present the XML encoding of our case study specification.

```xml
<systemSpecification name="P2PliveStreaming">
  <userDomain>
    <roles>
      <role name="sender" />
      <role name="receiver" />
    </roles>
    <states>
      <state name="s0"> <initial /> </state>
      <state name="s1" />
      <state name="s2" />
      <state name="s3"> <final /> </state>
    </states>
    <transitions>
      <transition name="t0" fromState="s0" toState="s1">
        <fromRole>sender</fromRole>
        <toRole>receiver</toRole>
      </transition>
      <transition name="t1" fromState="s1" toState="s2">
        <fromRole>receiver</fromRole>
        <toRole>sender</toRole>
      </transition>
      <transition name="t2" fromState="s2" toState="s3">
        <fromRole>sender</fromRole>
        <toRole>receiver</toRole>
      </transition>
    </transitions>
  </userDomain>
</systemSpecification>
```

In the following, we present the XML encoding of our case study specification.
APPENDIX C
AUGMENTATION ALGORITHM

In the following, we present the algorithm to generate the three Communication Rationality’s deviations for reducing communication costs: timeout, subset, and multicast deviations. These deviations augment the original User domain representation of the protocol, and produce as output the augmented protocol.

A. Timeout Deviations

The algorithm checks for each non-final state $s$ of the protocol whether $s$ has no outgoing transition that executes a legal timeout. If so, calls the procedure CreateTimeoutDeviation. This creates a new final state illegalState and connects it with $s$ by a timeout deviation (i.e., a new transition dev and the related interface devIntf).

B. Subset Deviations

The algorithm considers all the outgoing transition of each non-final state $s$. If a transition $t$ in this set implements a method $m$ that transmits a complex-type content, then calls the procedure CreateSubsetDeviation. The procedure creates a new state (illegalState), a new transition (dev) and a new interface (devIntf) to represent the deviation. The interface defines a method devMethod that is a copy of $m$ except for the content, which is a subset of the legit one ($m$.content).

C. Multicast Deviations

If an outgoing transition $t$ of a non-final state $s$ targets more than one recipient, then the algorithm launches the procedure CreateMulticastDeviation. A new state (illegalState) is then generated, and it is connected to $s$ with a new transition dev. Such transition is equal to $t$ but for the list of recipients, which is a subset of the one of $t$ (t.toUsers).

D. UpdateConnections

The algorithm ends by calling the procedure UpdateConnections, which generates the transitions between the new illegal states and the other states of the automaton. For example, in Fig. 6, the transition between $s_0$ and $s_2$ is created by the UpdateConnections procedure.

```plaintext
Data: The data structure P parsed from the User domain specification

Algorithm CreateCommDeviations(P)
1   NFS := non-final states in P
2   foreach s ∈ NFS do
3      OT := set of the outgoing transitions of s
4         if OT has not legal timeout then
5            CreateTimeoutDeviation(s)
6         foreach transition t ∈ OT do
7            st := target state of t
8               M := set of methods in t.interface
9               if m ∈ M | m.content is a complex type then
10                  CreateSubsetDeviation(st, t, m)
11         if t.toRoles.size > 1 then
12            CreateMulticastDeviation(st, t)
13   UpdateConnections(P)

Procedure CreateTimeoutDeviation(s)
14   illegalState := new final state
15   dev := new transition from s to illegalState
16   devIntf := new interface
17   devIntf.method := "new method with name "timeout"
18   add illegalState, dev and devIntf to P

Procedure CreateSubsetDeviation(st,t,m)
19   illegalState := new state
20   dev := new transition from st to illegalState
21   copy t.fromUser and t.toUsers into dev
22   devMethod := m
23   devMethod.content := subset of m.content
24   devIntf := t.interface
25   replace m with devMethod into devIntf
26   add illegalState, dev and devIntf to P

Procedure CreateMulticastDeviation(st,t)
27   illegalState := new state
28   dev := new transition from st to illegalState
29   copy t.fromUser into dev
30   dev.toUsers := subset of t.toUsers
31   devIntf := t.interface
32   add illegalState, dev and devIntf to P
```