Institutions
Markets
Technologies
IMT
INSTITUTE
FOR ADVANCED
STUDIES
LUCCA

#05
2013

IMT LUCCA CSA TECHNICAL
REPORT SERIES 05
March 2013

RA **Computer Science and Applications**

Network-aware Evaluation Environment for Reputation Systems

Alessandro Celestini
Rocco De Nicola
Francesco Tiezzi

IMT LUCCA CSA TECHNICAL REPORT SERIES #05/2013

Piazza San Ponziano 6, 55100 Lucca

Research Area
Computer Science and Applications

Network-aware Evaluation Environment for Reputation Systems

Alessandro Celestini
IMT Institute for Advanced Studies Lucca

Rocco De Nicola
IMT Institute for Advanced Studies Lucca

Francesco Tiezzi
IMT Institute for Advanced Studies Lucca

Network-aware Evaluation Environment for Reputation Systems

Alessandro Celestini[1], Rocco De Nicola[1], and Francesco Tiezzi[1]

IMT Institute for Advanced Studies Lucca, Italy
{alessandro.celestini,rocco.denicola,francesco.tiezzi}@imtlucca.it

Abstract. Parties of reputation systems rate each other and use ratings to compute reputation scores that drive their interactions. When deciding which reputation model to deploy in a network environment, it is important to find the most suitable model and to determine its right initial configuration. This calls for an engineering approach for describing, implementing and evaluating reputation systems while taking into account specific aspects of both the reputation systems and the networked environment where they will run. We present a software tool (NEVER) for network-aware evaluation of reputation systems and their rapid prototyping through experiments performed according to user-specified parameters. To demonstrate effectiveness of NEVER, we analyse reputation models based on the beta distribution and the maximum likelihood estimation.

Keywords: Reputation systems, Network-awareness, Evaluation tool

1 Introduction

In recent years, we have seen an increasing use of reputation systems in different areas of ICT, from e-commerce to different forms of open computer networking, such as P2P, ad-hoc, or sensor networks. This phenomenon is likely to continue, due to the success of networked applications (like social networks or other Web 2.0 technologies) and to the need, in such environments, of instruments to build up relationships of trust among the interacting parties. In order to establish such trust relationships, parties in a reputation system are free to interact and rate each other after any interaction, such ratings are then used to derive parties' reputation scores. The computed *reputation* score is a collective measure of parties' trustworthiness and it is used by parties in selecting the party to interact with.

Parties in a reputation system can exchange ratings and interact by relying on a network infrastructure. We consider in this work a centralised architecture (widely used for networked trust infrastructures) graphically depicted in Figure 1. In this general infrastructure, a *rating server* collects ratings from system's parties and makes them publicly available, while a *search server* allows parties to find resource providers in the system. Every party can play the role of a client, of a provider, or both, and may offer different kinds of resources (services, computational and storage resources, etc.). Whenever a party needs a resource, first it queries the search server to get the list of parties providing it, and then retrieves from the rating server the ratings of each provider in the list. Thus, to choose a provider, it computes the reputation scores of each of them and selects the

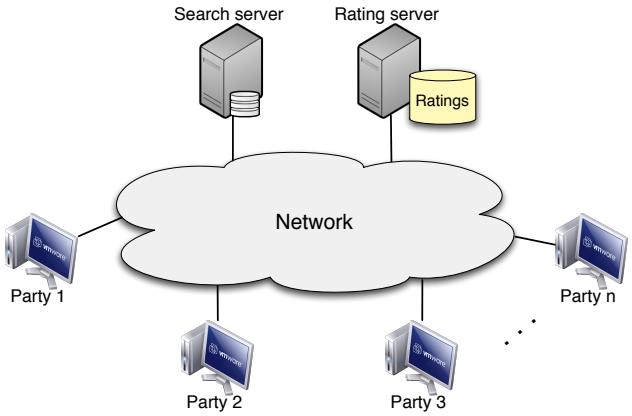

Fig. 1: General infrastructure of a reputation system

one with the highest reputation score. Finally, after the interaction, it rates the provider according to the quality of the provided resource. On top of the general infrastructure just described, different kinds of reputation system can be layered, which mainly differ for the model they use to aggregate ratings when computing reputation scores.

Several models have been proposed and once a reputation system has to be deployed in a network environment, we might ask which reputation model is more suitable for the given environment and how the reputation system should be configured in order to meet the desired behaviour. This calls for an engineering approach for describing, implementing, evaluating reputation systems while taking into account real-word implementation details of such systems and of the network environment where they have to be deployed.

In this paper, we address this issue by proposing a software tool for network-aware evaluation of reputation systems. On the one hand, we provide a framework for rapidly developing Java-based implementations of reputation system models and for easily configuring different networked execution environments on top of which the systems will run. On the other hand, we developed a tool that automatically performs experiments on the reputation system implementations according to user-specified parameters; this enables the study of their behaviour while executing on given network infrastructures. The main novelty of our approach, with respect to other proposals in the literature with a similar aim, is that we allow the evaluation of implemented reputation systems through experiments on real networks, rather than performing simulation of models of reputation systems that abstract from many details. In this way, given a specific network environment, we can study the system behaviour to find the configuration that better meets the system requirements by tuning its parameters (reputation model, response timeouts, resource quality evaluation, ratings aging, etc.). Moreover, since we consider reputation systems at implementation level, the analysed systems could be then directly used in the corresponding end-user applications (we will come back on this point in Section 6).

To demonstrate the effectiveness of our proposal, we analyse models of reputation systems based on the beta distribution [14] and the maximum likelihood estimation [9]. To carry out experiments on network infrastructures involving many nodes, we have also exploited a Cloud IaaS platform, namely Zimory Enterprise Cloud [12].

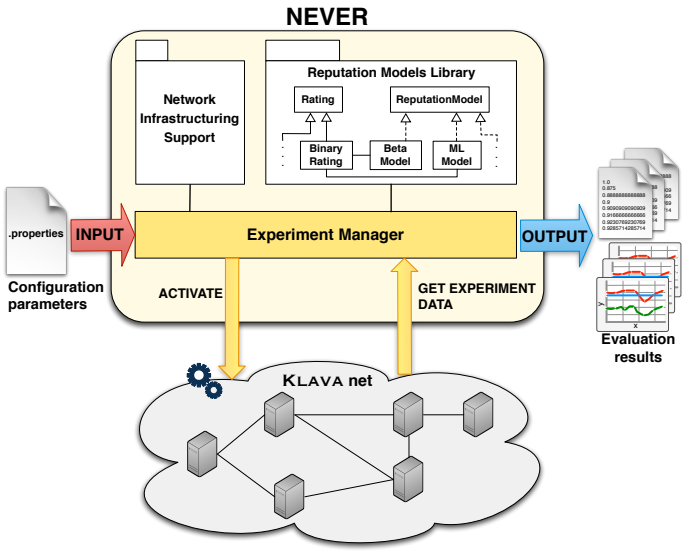

Fig. 2: NEVER architecture and workflow

The rest of the paper is organized as follows. Section 2 describes the architecture and functional principles of our tool. Section 3 provides a brief overview of the tool component dealing with networking aspects. Section 4 presents the theoretical and implementation aspects of the reputation models currently considered in this work. Section 5 reports on the analysis of a few reputation systems. Finally, Section 6 concludes the paper by also reviewing some of the related work and suggesting directions for future work.

2 The NEVER tool

In this section, we present the architecture and the workflow of NEVER (Network-aware EValuation Environment for Reputation systems), graphically depicted in Figure 2. The NEVER tool consists of three main components: (1) the experiment manager, (2) the network infrastructuring (3) the reputation models library.

The *experiment manager* is the components playing the main role, because it is in charge of managing the execution of each experiment. An experiment consists of a user-specified number of runs, each run performed with the same configuration. The number of runs and their duration, together with other experiments characteristics, are defined by users through configuration parameters.

The *network infrastructuring support* provides the libraries (i.e., classes and interfaces) required to create and set up a KLAVA *net* (see Section 3.1) implementing the general infrastructure graphically depicted in Figure 1. Each element of the infrastructure is a node hosted by a (possibly remote and/or virtual) machine. The NEVER tool takes as input the addresses of the hosting machines and automatically activates the nodes

forming the wanted network infrastructure. We refer to Section 3 for further details on the network infrastructure library supporting our experiments.

The *reputation models library* acts as a framework allowing the user to define the trust and reputation models under evaluation. The library is a Java package containing a number of abstract classes and interfaces necessary to implement the models. In this way, the NEVER tool is customizable and extendible by the user. More specifically, a reputation model is defined by a class implementing the `ReputationModel` interface and, possibly, a class extending the abstract class `Rating`. The former class defines how reputation scores are computed, which rating values are used by the system and how parties in the system evaluate the interactions. The latter class defines the kind of rating values and how to manage them. Thus, the addition of new reputation models to NEVER can be achieved by implementing `ReputationModel` and, if necessary, by extending `Rating`. We refer to Section 4 for further details on the reputation models library and on the models already available in NEVER.

We describe now the NEVER workflow, by lingering on the main features of the experiment manager component. The tool takes as input a set of *configuration parameters*, written in a *.properties* file as pairs of the form *key* = *value*. Such parameters are used by the experiment manager to instantiate and carry out an experiment. First, the manager creates the network on top of which will be run the experiment. A node is created for each of the two servers and for each party in the system. Once the network is set up, the reputation system (configured according to user's parameters) is deployed on the network and the experiment starts, i.e. network components are enabled and system parties interact and rate each other. During the activity of the network, data about interactions are stored in appropriate files for a later analysis. Experiment runs are repeated in order to reach the desired precision; thus, the manager starts and stops runs till the last run is accomplished. Afterwards, data are analysed and provided as output, both in form of textual files and charts[1]. We refer to Section 5 for a discussion about different experiments carried out with our tool.

We conclude this section by commenting on the relevant configuration parameters. Through such parameters it is possible to specify the number of parties in the system and the addresses of the machines where parties have to run. For each party, a new KLAVA node is automatically created and deployed in the associated hosting machine. The tool also supports a 'local only' modality, where all KLAVA nodes are deployed in the same machine running the tool. Such modality can be useful to compare reputation systems in presence or absence of networking aspects affecting the evaluation.

A specific configuration parameter is used to set the main reputation model, which is used during the experiment to drive the interactions among parties. In fact, when a party is looking for a provider of a specific resource, it computes the providers' reputations and selects for the interaction the most trusted one, i.e. the party (or one of the parties) with the highest reputation value. Besides the main model, it is possible to give a list of trust and reputation models to be compared during the experiment: each party's reputation is computed according to all models specified in such list. Values of party's reputation are returned for each run and, at the end of the experiment, as a mean value

[1] The tool automatically generates charts by exploiting the Java library JFreeChart (freely available at `http://www.jfree.org/jfreechart/`).

over all runs. Moreover, the user can require to randomly select the providers, by thus ignoring the choice of the providers based on the main reputation model. Such modality is indeed often used in our experiments, because it gives the opportunity of evaluating models performances by comparing party reputations on the basis of approximately the same amount of ratings for each party.

A group of configuration parameters regulates the parties' behaviour. The user specifies a set of possible party behaviours and the percentage of parties with each given behaviour. By means of such information, the experiment manager assigns a behaviour to each party. Moreover, it is possible to set the initial reputation of parties by specifying the values and the number of their initial ratings. Such ratings determine the initial parties reputation computed by the system. In the default case, parties' behaviour are assumed to be fixed, but a changeable behaviour can be configured. In such a case, the user sets when the variation has to happen and the magnitude of the variation. Currently, the variation implemented is negative, i.e. party's behaviour gets worse after the variation. Several studies (see, e.g., [14, 19, 22]) use similar approaches for the evaluation of reputation models.

Finally, the configuration parameters allow the user to set two threshold values: the maximum delay and the maximum waiting time. The first parameter sets the maximum delay after which a resource is considered unsatisfactory, i.e. once the party receives the resource it checks if the arrival time exceeds the maximum delay and, in such a case, a negative rating is given to the provider no matter the quality of the resource. The second parameter sets the maximum time that a party will wait for a resource; expired this time a new provider is selected by the party and no rating value is given. In this way, a party will not wait indefinitely for a resource.

The NEVER tool is developed in Java, by exploiting freely available third-party libraries. Source and binary files of NEVER can be found at `http://sysma.lab.imtlucca.it/tools/never/`.

3 Network infrastructuring support

The network infrastructuring support of NEVER provides an API that allows the experiment manager to create different networks underlying the reputation systems to be evaluated. To this aim, this tool component exploits the KLAVA library. In this section, we briefly introduce KLAVA and discuss the structure of the Java package providing the network infrastructuring support. In particular, we present the functionalities of each component of the package and describe its implementation in KLAVA. We illustrate how network components interact by showing pieces of code.

3.1 KLAVA

KLAVA [4] is a Java library providing the run-time support for KLAIM actions within Java code. KLAIM [7] is a formal coordination language specifically designed for modelling mobile and distributed applications and their interactions, which run in a network environment. KLAIM provides communication primitives enabling tuple-based interaction à la Linda [11], which decouples the communicating processes both in space and time.

Exchanged data are sequences of values, i.e. *tuples*. Communication is achieved via distributed multisets of tuples, called *tuple-spaces*, where processes insert, read and withdraw tuples. The data retrieving mechanism uses associative pattern-matching to find the required data in the tuple-space. The KLAIM communication primitives, written as KLAVA methods, used for implementing the network infrastructuring package are as follows:

- **public void** out(Tuple t, Locality l)
- **public void** in(Tuple t, Locality l)
- **public void** read(Tuple t, Locality l)
- **public boolean** read_t(Tuple t, Locality l, **long** timeOut)
- **public boolean** in_t(Tuple t, Locality l, **long** timeOut)
- **public boolean** in_nb(Tuple t, Locality l)
- **public boolean** read_nb(Tuple t, Locality l)

These methods take as parameters a tuple and the (either logical or physical) locality, i.e. the *address*, of the destination node. If the action refers to the current execution site (through the reserved logical locality **self**), it is simply redirected to the local tuple-space, otherwise a message will be sent to the (possibly remote) destination node. In particular, the method call out(t,l) adds the tuple resulting from the evaluation of t to the tuple-space of the target node identified by l. Method calls in(t,l) and read(t,l) implement retrieval actions that permit to withdraw/read data tuples from the tuple-space hosted at locality l. They look for a tuple matching the result of the evaluation of t; if no matching tuple is found the execution is suspended until one is available, otherwise one of the matching tuple is non-deterministically chosen and its values are assigned to the corresponding formal[2] fields of t. Calls in_t(t,l,time) and read_t(t,l,time) permit specifying upper bounds to the waiting time, i.e. a *time-out*, expressed in milliseconds. This is useful to deal with high network latency or absence of matching tuples. Finally, calls in_nb(t,l) and read_nb(t,l) implement non-blocking versions of the retrieval actions: if a matching tuple is found, they act similarly to in and read, and additionally return the value **true**, otherwise they return the value **false** and the execution does not block. They can be used, e.g., to test whether a tuple is present in a tuple-space. We refer the interested reader to [2] for a complete account of the methods and classes provided by the KLAVA library.

The KLAVA library relies on the IMC framework [3], which provides recurrent mechanisms for network applications and, hence, can be used as a middleware for the implementation of different formal languages. Specifically, KLAVA provides classes to be instantiated to create a net, and the nodes that can be connected to the net to build the desired network environment. An abstract class is then provided to create processes to be placed on to the nodes, by means of instantiation of subclasses specialized through inheritance and method overriding.

[2] A formal field is an item of a tuple subjects to substitution in case the match is satisfied; formal fields are distinguished from actual fields because they are created with the *default* constructor (i.e., the constructor with no parameters).

3.2 The network infrastructuring package

The network infrastructuring package specifies three different kinds of nodes that take part in the KLAVA net: a *rating server* node, a *search server* node and a *user* node. Each of these nodes implements a component of the infrastructure graphically depicted in Figure 1.

The rating server node serves as public database for collecting parties' ratings and executes the process RatingServerProcess. This process is in charge of collecting data produced by each experiment run.

The search server node assists parties while seeking a resource provider. The process SearchServerProcess (Listing 1.1) runs on this node. It waits for search requests sent by parties (line 5). Parties send requests, i.e. tuples tagged by the search_request string, stating the type of the resource they want from the provider. Then, SearchServerProcess looks in the local tuple space for available providers offering such resource (lines 11 and 18): for each provider matching the request, the process sends its address to the requesting party (line 16). The set of tuples sent to the party forms a list of provider addresses. When all providers have been checked[3], the SearchServerProcess closes the list by sending its length to the requesting party (line 22).

Listing 1.1: SearchServerProcess

```
1   // Wait for a new search request
2   KInteger n_providers = new KInteger( 0 ) ;
3   Locality loc_requester = new PhysicalLocality();
4   KString res_type = new KString() ;
5   in( new Tuple(new KString("search_request"),loc_requester,res_type), self);
6
7   // Scan the list of parties providing resources of type 'res_type'
8   Locality loc_provider = new PhysicalLocality();
9   Tuple templRead_nb = new Tuple(res_type, loc_provider);
10  templRead_nb.setHandleRetrieved(true);
11  KBoolean forallExpressionArgument = new KBoolean( read_nb(templRead_nb, self ) ) ;
12  while ( forallExpressionArgument.booleanValue() ) {
13          // Increase the counter of providers
14          n_providers = new KInteger( n_providers.intValue() + 1 );
15          // Send the provider's address
16          out(new Tuple(new KString( "list" ),n_providers,loc_provider),loc_requester);
17          templRead_nb.resetOriginalTemplate();
18          forallExpressionArgument = new KBoolean( read_nb(templRead_nb, self) );
19  }
20
21  // Send the length of the list of providers
22  out(new Tuple(new KString( "list_length" ), n_providers), loc_requester) ;
```

[3] To read all matching tuples only once in this kind of loops, KLAVA provides specific built-in mechanisms that prevent matching twice the same tuple. In particular, method setHandleRetrieved() allows the tuple templRead_nb to store all the tuples that it has matched (line 10), while method resetOriginalTemplate() is used to reinitialize to empty values the formal fields of templRead_nb (line 17) in order to use this template to retrieve another tuple in the next read_nb action (line 18).

The user node implements a generic party; nodes of this kind interact to ask and provide resources and, after any interaction, rate each other. Two processes run on the user node[4]: the ProviderProcess (Listing 1.2) and the ClientProcess (Listing 1.3). The former process implements the functionalities of a provider: when a new resource request coming from a client is received (line 4), the resource is selected (line 7) and sent to the client (line 10). The resource selection consists of determining its quality according to the provider's behaviour; in fact, the actual provision of the resource is not relevant for our studies.

Listing 1.2: ProviderProcess

```
1   // Wait for a resource request
2   Locality requesterLoc = new PhysicalLocality();
3   KString res_type = new KString();
4   in(new Tuple(new KString("resource_request"), requesterLoc, res_type), self);
5
6   // Get resource quality according to provider's behaviour
7   KDouble res_quality = model.getResourceQuality(new_behaviour,rand);
8
9   // Send the resource
10  out(new Tuple(new KString("resource"), res_type, res_quality), requesterLoc);
```

The ClientProcess seeks providers for the resource it is looking for, and selects the most trusted one for the next interaction. It first sets the variable most_trusted_user to NO_ONE (lines 2-3) denoting that no provider has been selected. Then, it determines the resource type it wants to request (lines 6-7). The ClientProcess asks the search server to find a provider for a given resource type (line 10) and selects, among the providers returned by the search server, the most trusted one, i.e. a provider with highest reputation score (line 16). Then, it checks if the reputation of such provider is higher than the minimum reputation value defined in the configuration file (line 20). If this check is positive, the process sends a request for the resource to the selected provider (lines 23-24), otherwise it starts again the procedure from the beginning. The waiting time of a requested resource is bounded by a time-out, request_time_out, specified in the configuration file (lines 30-31). When the resource is received the process computes a rating value for the provider (line 35) and sends it to the rating server (lines 41-44).

Listing 1.3: ClientProcess

```
1   // Initialize the "most trusted user" tuple
2   out(new Tuple(new KString("most_trusted_user"), getPhysical(self),
3                 new KDouble(NO_ONE), new KInteger(0)), self);
4
5   // Resource type is randomly selected
6   int res_num = (int) ((rand.Fran2() * (number_of_resource_types-1))+1);
7   KString res_type = new KString("type_"+res_num);
8
9   // Send the request to the search server and determine the most trusted user
```

[4] Depending on the processes running in its node, a party can play the role of a client, of a provider, or both. We consider here the latter case, which is the most general.

```
10      searchProvider(res_type);
11
12      // Get the data of the most trusted user
13      Locality trusted_loc = new PhysicalLocality();
14      KDouble reputation_most_trusted = new KDouble();
15      KInteger number_of_ratings = new KInteger();
16      in(new Tuple(new KString("most_trusted_user"), trusted_loc,
17                      reputation_most_trusted, number_of_ratings), self);
18
19      // Check if provider's reputation is higher than the minimal reputation
20      if (min_reputation <= reputation_most_trusted.doubleValue()) {
21
22              // Send a resource request to the provider
23              out(new Tuple(new KString("resource_request"), getPhysical(self),
24                              res_type), trusted_loc);
25
26              // Wait the resource
27              long time_of_request=System.currentTimeMillis();
28              KDouble quality = new KDouble();
29              Rating provider_rating = model.createNonInitializedRate();
30              if ( in_t(new Tuple(new KString("resource"), res_type, quality),
31                      self, request_time_out)) {
32                long time_of_service = System.currentTimeMillis();
33                long response_time = time_of_service-time_of_request;
34                // Check the quality of the obtained resource and rate the provider
35                try { provider_rating = model.rateProvider(quality,response_time); }
36                catch (MalformedRateValueException e) {
37                        e.printStackTrace();
38                        System.exit(1);
39                }
40                long ratingTime = System.currentTimeMillis();
41                out(new Tuple(getPhysical(self), trusted_loc,
42                                provider_rating.getValue(),
43                                new KString(Long.toString(ratingTime)) ),
44                    UserNode.rating_serverLogLoc);
45              }
46      }
```

4 Trust and reputation system models

In this section, we provide some details about trust and reputation models already implemented in NEVER; this would also serve as a guide for using the framework to implement new models to be evaluated. Specifically, first we briefly introduce a theoretical formalization of trust and reputation systems needed for the presentation of the models, then we discuss their implementation.

4.1 Models

Parties in a trust and reputation system are free to interact and rate each other. After each interaction, a *rater* assigns a score to a *ratee*. We denote by $\mathcal{R} = \{r_1, ..., r_m\}$ a finite, non-empty set of rating values. We focus on probabilistic trust and reputation models [10, 9] where party's behaviour is modelled by a probability distribution on $\mathcal{R}$. Let $\mathcal{P}$ be a set of party identities, the *behaviour* of each party $p \in \mathcal{P}$ is characterised by a distribution parameter $\theta_p \in \Theta$, with Θ denoting the set of possible parameters of a given distribution $p(\cdot \mid \theta_p)$. Let $r \in \mathcal{R}$ be a rating value, this probability distribution returns the value $p(r \mid \theta_p)$ denoting the probability of observing a rating value r after an interaction with a party whose behaviour is (determined by) θ_p.

The goal of a reputation system is to predict parties' behaviour in future interactions, given the rating values of past interactions. Thus, a reputation system has to provide the *reputation* of each party p, i.e. an estimation $\tilde{\theta}_p$ of the party's behaviour θ_p. In the models we consider, the sequence of rating values $r_n^p = r_1, \ldots, r_n$, assigned by parties after each interaction with party p, is assumed to be a realization of a sequence of independent, identically distributed random variables $R_n^p = R_1, \ldots, R_n$. Moreover, the set of rating values $\mathcal{R}$ is simply the binary set $\{0, 1\}$, with values 0 and 1 denoting 'unsatisfactory' or 'satisfactory' interactions, respectively. Random variables R_i are assumed to be distributed according to a Bernoulli distribution with success probability θ_p. It is thus assumed that when interacting with a party p, whose behaviour is θ_p, the probability that the next interaction is 'satisfactory' is $p(1 \mid \theta_p) = \theta_p$, and 'unsatisfactory' is $p(0 \mid \theta_p) = 1 - \theta_p$. Below, we show how such framework is instantiated to capture two of the most used reputation models, namely the Beta model and the Maximum Likelihood (ML) model.

The *Beta model* takes as input both the number of past satisfactory interactions and the number of past unsatisfactory interactions. These numbers are determined by taking into account both party's own experience and the reports from other parties without considering the origin of the rating values. The Beta model estimates the probability that a next interaction with the party will be satisfactory by using only these two values. The name of this model is due to the fact that it uses the beta distribution to estimate the posterior probabilities of binary events. Party's reputation score $\tilde{\theta}_p$ is given by the expected value of the beta distribution $Beta(\alpha + 1, \beta + 1)$ with $\alpha \geq 0$ and $\beta \geq 0$:

$$\tilde{\theta}_p = \mathrm{E}[Beta(\alpha + 1, \beta + 1)] = \frac{\alpha + 1}{\alpha + 1 + \beta + 1} \tag{1}$$

Parameter α represents the number of satisfactory past interactions with party p and β represents the unsatisfactory interactions with p.

The *ML model* takes as input the number of satisfactory and unsatisfactory past interactions, as in the case of the Beta model. Also in this model there is no difference between own and other experience. The ML model uses a different approach for the estimation of the probability that a next interaction with the party will be satisfactory. Indeed, it tries to find the parameter that maximises the likelihood:

$$L(\theta \mid R_n^p) = Pr(R_n^p \mid \theta) = \prod_{i=1}^{n} Pr(R_i = r_i \mid \theta)$$

In case of binary events, given $\alpha \geq 0$ and $\beta \geq 0$ representing the number of satisfactory and unsatisfactory past interactions respectively, the party's reputation score $\tilde{\theta}_p$ is as follows:

$$\tilde{\theta}_p = \frac{\alpha}{\alpha + \beta} \tag{2}$$

Other components can be added to these two models to take into account other aspects of parties' behaviour. In particular, for the Beta model it is possible to specify a parameter called *forgetting factor* [14], which is a value in the interval $[0, 1]$. The aim of this parameter is to give a different weight to each rating based on its age. We can order all the available ratings for a party p from the newest to the oldest, denoting with $r_n^p = r_0, \ldots, r_n$ the ordered sequence of rating values and with λ the forgetting factor. The weight associated to rating r_i is defined as λ^i, i.e. older ratings will be gradually forgotten. Thus, value $\lambda = 1$ is equivalent to absence of the forgetting factor, while value $\lambda = 0$ results in taking into account only the last rating (with the convention that $0^0 = 1$). The other possible values for λ approximate such extreme behaviours. In this work, we have used the forgetting factor also to parameterize the ML model.

Moreover, we have found it useful to analyze the reputation models by using just a part of the knowledge acquired by past interactions. Thus, when the reputation of a party is computed, we consider only a subset of the available rating values. We denote with the term *window* the number of the used ratings. Among all available ratings, only the latest ones are selected, e.g. a window set to 20 means that only the last 20 ratings are used for the computation of party's reputation. The window parameter can be set for both ML and Beta model, obtaining a different model for each value of the window. To the best of our knowledge seems that such parameter is not present in any proposal of reputation models, or at least not explicitly mentioned or discussed. In Section 5, we show how and how much different window size are effective for the estimation of parties' reputation.

4.2 Implementation

As we have shortly discussed in Section 2, it is possible to implement trust and reputation models in NEVER through the reputation models library. The first step is to create a class implementing the ReputationModel interface, whose main methods are the following:

- **public void** setWindow(**int** size);
- **public void** setForgettingFactor(**double** value);
- **public** KDouble evaluateReputation(Vector<Rating> ratings, Vector<Locality> raters);
- **public** KDouble getResourceQuality(**int** behaviour, Ran2 rand);

The methods setWindow and setForgettingFactor are used to set the corresponding parameters (see Section 4.1). The method evaluateReputation is the basis of any reputation model. It takes as input a list of rating values (Vector<Rating> ratings) and the corresponding list of raters (Vector<Locality> raters), and returns as output the reputation score of the ratee. Ratings in the vector are sorted from the newest to the oldest; such organization is useful for models that discriminate ratings depending on their age. Finally,

method getResourceQuality is used to determine the quality of the resource to be provided to the client. This value corresponds to the outcome of the interaction, and depends on the model implemented and on the set of rating values in use.

Now, we show how the models outlined in Section 4.1 are implemented in NEVER, by focussing on the code of evaluateReputation method that specifies how rating values are used to compute party's reputation in the models. We start from the implementation of the ML model (Listing 1.4).

Listing 1.4: evaluateReputation (MLModel.java)

```
1   double num_of_ratings = ratings.size();
2   double num_positive_ratings = 0;
3   if ( num_of_ratings == 0){
4           return new KDouble();
5   }
6   if ( (WINDOW != 0) && (num_of_ratings > WINDOW)){
7           num_of_ratings = WINDOW;
8   }
9   for (int i = 0; i < num_of_ratings; i++) {
10          KInteger rating_value = (KInteger) ratings.get(i).getValue();
11          if ( rating_value.intValue() == POSITIVE_RATE_VALUE ){
12                  num_positive_ratings++;
13          }
14  }
15  double ml_reputation = num_positive_ratings/num_of_ratings ;
16  return new KDouble(ml_reputation);
```

This method first checks the number of available ratings (line 3): if there are no ratings the computation does not take place and the default reputation value is used by the system, i.e. an empty KDouble() object is returned and the system uses as party's reputation the value set in the configuration file (parameter no_rating_reputation). A second check (line 6) is done on the window's size: if the number of rating values is bigger than the window's size, only the newer ratings are used. The last part of the code (lines 9-15) computes the party's reputation. In case of binary ratings, in the ML model this means to simply divide the number of satisfactory interactions by the total number of interactions (see equation (2)). Finally, the computed reputation value is returned as a result by the method (line 16).

We now examine the code implementing the Beta model (Listing 1.5). In the Beta model, party's reputation is computed as the expected value of beta distribution (see equation (1)). We show only the last two lines, since the first part of the code, where it is checked the number of available ratings and the window's size, is common among all considered models.

Listing 1.5: evaluateReputation (BetaModel.java)

```
1  double beta_reputation = (num_positive_ratings+1)/(num_of_ratings+2) ;
2  return new KDouble(beta_reputation);
```

The last kind of models we implemented makes use of the forgetting factor. The code shown in Listing 1.6 is the final part of the method evaluateReputation implemented by the class BetaModelForgetting; the implementation of the same method within

class MLModelForgetting is similar. Each rating value here is weighted according to its age (lines 4 and 6). The weight of each rating is given by the value λ^i, where λ (i.e. LAMBDA, in the code) is the forgetting factor and i denotes rating's age.

Listing 1.6: evaluateReputation (BetaModelForgetting.java)

```
1  for (int i = 0; i < num_of_ratings; i++) {
2      KInteger rating_value = (KInteger) ratings.get(i).getValue();
3      if ( rating_value.intValue() == POSITIVE_RATE_VALUE ){
4              num_positive_ratings = num_positive_ratings + Math.pow(LAMBDA, (i));
5      } else {
6              num_negative_ratings = num_negative_ratings + Math.pow(LAMBDA, (i));
7      }
8  }
9  double beta_rep = (num_positive_ratings+1)/(num_negative_ratings+num_positive_ratings+2) ;
10 return new KDouble(beta_rep);
```

To implement a reputation model, it is also needed to provide the implementation of rating values. The models currently available in NEVER use *binary ratings*, i.e. each interaction can be rated either 'unsatisfactory' or 'satisfactory'. In order to define another kind of rating, a new class must be created as a subclass of the abstract class Rating (Listing 1.7).

Listing 1.7: Rating.java

```
public abstract class Rating implements Comparable<Rating>{
2     protected TupleItem value;
3     protected long timestamp;
4
5     public abstract void setValue (TupleItem value) throws MalformedRateValueException;
6
7     public TupleItem getValue(){
8         return this.value;
9     }
10    public void setTime(KString time){
11        this.timestamp = Long.parseLong(time.toString());
12    }
13    public long getTime(){
14        return timestamp;
15    }
16    ...
17 }
```

This class defines four methods. Methods setTime and getTime are used to set and retrieve ratings' timestamp. Such values are used for sorting the vector containing rating values. Methods getValue and setValue respectively return and set the value of the rating. In particular, a new rating class has to implement the method setValue that is declared abstract; its implementation should check the format of the rating value.

5 NEVER at work

In this section we show which data NEVER returns as output and in which formats the output is provided. For illustration purpose, we run an experiment with the following parameters: four parties are active in the system, data are averaged over fifteen runs, each run lasts twenty minutes. Two possible behaviours are defined in the system, both modelled as Bernoulli distributions of parameter θ. One behaviour fixes the value of θ to 0.9, while the other to 0.6. Among the four parties two have a behaviour $\theta = 0.9$ and two a behaviour $\theta = 0.6$. Parties' behaviours are set to change after twenty five interactions, specifically each party update its behaviour to $\theta = \theta - 0.4$, i.e. parties assume a worst behaviour than the initial one.

No initial reputation is set for parties: when a party is found to have no ratings is assumed to have a reputation score of 0.5 (parameter no_rating_reputations). No minimum reputation values is set for interacting with a party (parameter min_reputation), i.e. to interact with a party no limits are fixed about its reputation score. The main reputation model is defined to be a ML model without forgetting factor and window. Instead the list of model to compare contains five models:

- ML model without window and forgetting factor;
- ML model with window=20 and without forgetting factor;
- ML model with window=30 and without forgetting factor;
- ML model without window and with forgetting factor=0.9;
- Beta model without window and forgetting factor.

At the end of the experiment, NEVER returns as output both a graphical representation of data and a textual list of data. Textual output is provided in order to permit data manipulation without re-executing experiments. Different graphical representations can be used for data analysis.

The graph in Figure 3 is given as output by NEVER for the experiment just described. It shows the reputation trends of the four parties calculated using the main model. The preset behaviour of each party is denoted by an horizontal line. The vertical dashed line denotes the time when parties change their behaviour. The horizontal dashed lines denote new parties' behaviours after the change. The trend of party's reputation is denoted by a polygonal line and each party is identified by its ip address and port number. Dashed lines are present only if a changeable scenario is set in the configuration file (parameter changeable_behaviour). Through this graph is possible to analyse the evolution of parties' reputation in relation with the number of available ratings. The reputation values shown are computed for a fixed trust model and averaged over all runs. When changeable behaviour is set, it is also possible to analyse the reaction of the model when party's behaviour change in time. From Figure 3 we see that the ML model quite rapidly detects party's behaviour, with no significant error in reputation score assignation. Such model slowly adapts to party's new behaviour, and (according to other experiments) it is even worst when the change happens after a bigger number of interactions. Indeed values about past interactions influence more the estimation of the actual behaviour.

Besides this, party reputation is computed for each party with respect to all models specified in the model list within the configuration file. Figure 4 considers the case of

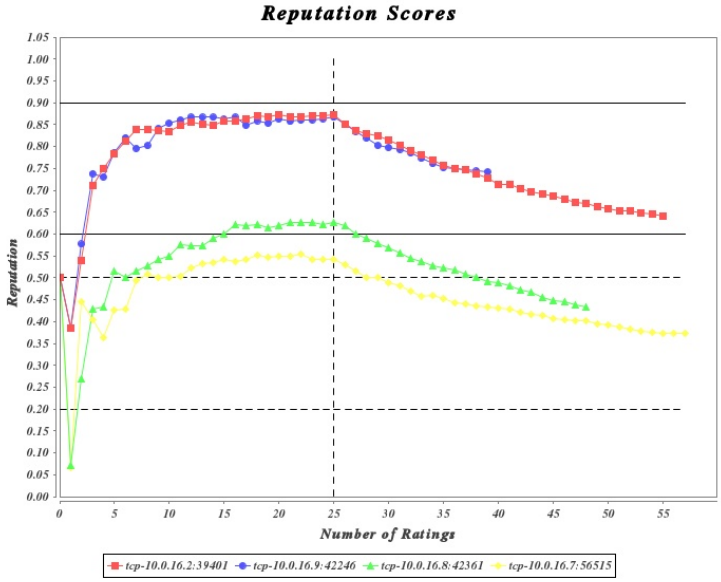

Fig. 3: Reputation trend of the four parties for a fixed trust model (ML model, no window, no forgetting factor)

a party with behaviour $\theta = 0.9$. Vertical and horizontal lines have the same meaning as in the previous figure. For each trust model is drawn a line denoting party's reputation. Through this graph is possible to analyse how different trust model evaluate party's reputation and how they react in case of changeable behaviour. From this kind of comparison it is possible to determine which is the best strategy for a given scenario. We can imagine scenario where some kind of parties' behaviours are more probable than others, e.g. the majority of parties behave badly or goodly. Scenario where the user is interested in preventing specific behaviours, e.g. parties try to reach a good reputation and then they start to behave badly. From Figure 4 it is possible to see how the five models react when party's behaviour changes from a behaviour $\theta = 0.9$ to a behaviour $\theta = 0.5$. The models using window parameter or forgetting factor are able to detect more rapidly new party's behaviour. Specifically, the ML model with a window size of 20 adapts more rapidly than the one with a window size of 30, but both models have a wors performance than the ML model using a forgettin factor of 0.9. Moreover, the Beta model has wors performance than the ML model and all its variations. This happens both when the behaviour is fixed and when the party suddenly assumes a new behaviour.

When a system with several parties is set up, it becomes hard to read the first kind of graph reported in Figure 3. Figure 5 shows an example with twenty parties[5]. In order to manage systems with several parties, NEVER returns as output four different graphs

[5] Given a limited number of physical machines at our disposal, to perform experiments with this number of parties, we have deployed KLAVA nodes on virtual machines running on the cloud IaaS platform Zimory Enterprise Cloud [12].

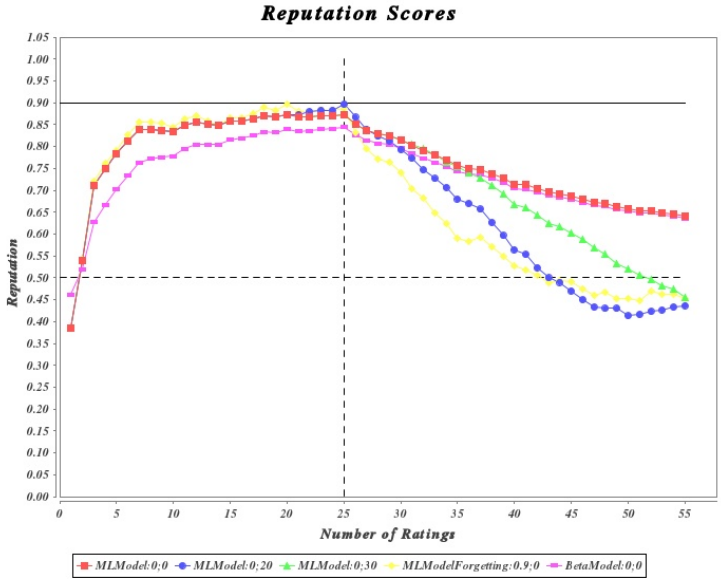

Fig. 4: Reputation trend of one party respect to all trust models specified in the model list as configuration parameters

where data is aggregated and parties are grouped depending on their behaviour. In Figure 6 is shown one of these graphs, where data about parties with a behaviour between $\theta = 0.5$ and $\theta = 0.75$ is aggregated. The horizontal line denotes the true behaviour of the group; such value is obtained through a weighted mean among parties' behaviour belonging to the group, i.e. the weight of each behaviour in the group is given by the number of parties having such behaviour. Group's reputation is computed following the same approach. For each model in the list a line denotes group's reputation.

Finally, in Figure 7, we show a graph where are graphically depicted two risk functions, the bayes and worst risk (see [5] for a risk analysis of trust and reputation systems based on such functions). These two functions evaluate the system as a whole. The *bayes risk* can be seen as the average risk in the system, where risk means the possibility of inferring party's behaviour wrongly. The *worst risk* is instead the risk incurred in the worst case, i.e. the maximum risk for a given behaviour. The Figure 7 reports bayes and worst risk trends for a system with twenty parties. In this case, parties' behaviour is supposed to change after twenty-five interactions. From the graph, it is possible to notice such change in the behaviour.

6 Concluding remarks

In this paper we presented NEVER, a network-aware tool for evaluating trust and reputation systems. The design of NEVER is based on the KLAIM formal specification of trust and reputation system presented in [6]. We used the Java library KLAVA for implementing the models specified in KLAIM. NEVER allows the user to rapid prototyping and testing reputation system models in a real network environment, thus realizing a

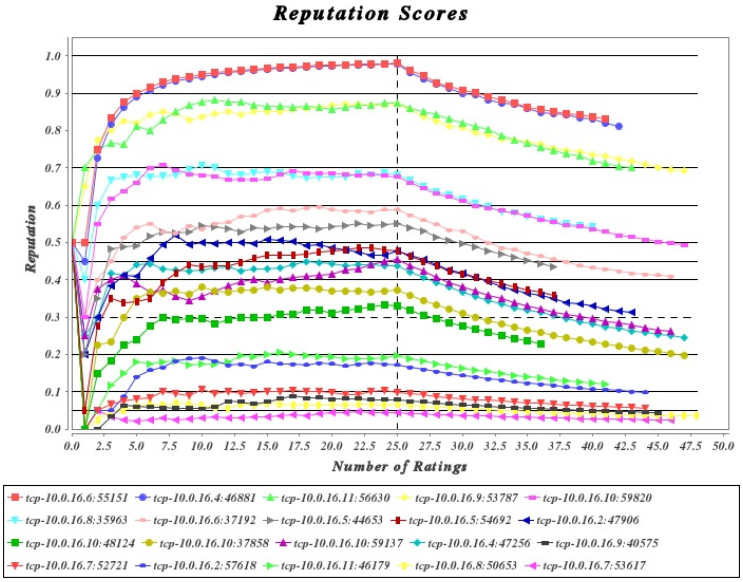

Fig. 5: Reputation trend of twenty parties for a fixed trust model (ML model, no window, no forgetting factor)

generic testbed for evaluating trust and reputation systems, thus realizing a generic testbed for evaluating trust and reputation systems. We discussed the architecture of NEVER showing the logical structure and short part of its implementation. We showed how NEVER works by means of experimental data obtained through the evaluation of some implemented models.

Related work. Trust and reputation are often used as synonyms in the literature. According to [15] trust is a subjective perception of reliability of a party, mainly derived from private knowledge (e.g. direct interactions with the party). Instead reputation is an objective measure of party's trustworthiness derived from referrals or ratings provided by other parties. In our work we comply with this distinction.

Among the many works in the literature whose goal is the evaluation and comparison of reputation systems, to the best of our knowledge, our contribution is the first effective tool allowing the evaluation of such systems in a real networked execution environment. Several works base their evaluation solely on a 'pen-and-paper' mathematical study of the models, without taking into account how they will be implemented and executed over distributed systems. For example, a formal framework for the comparison of probabilistic trust models, based on KL-divergence, is proposed in [21]. In this work KL-divergence is used as a measure of the quality of reputation functions. With the same purpose we exploit the notions of bayes and worst risk presented in [5]. NEVER computes the empirical value of such risk functions for the model set in the configuration file. Results of such computation are returned as output and are used for models evaluation.

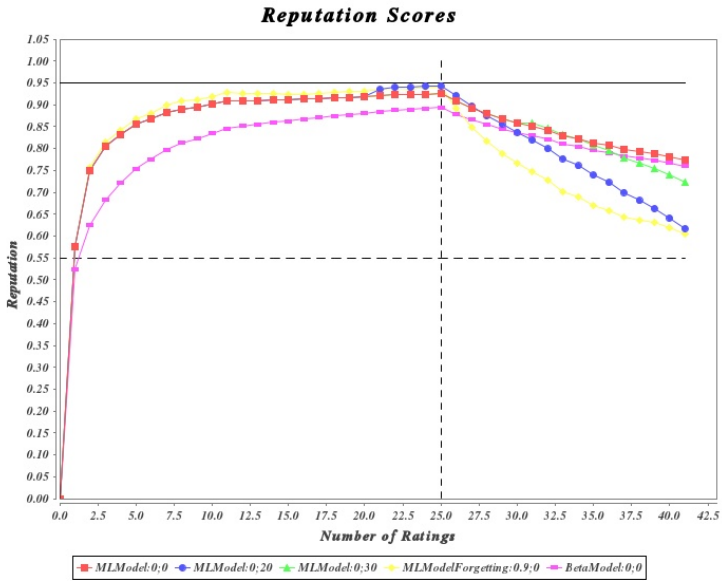

Fig. 6: Reputation trend for a group of parties

Other works use simulation techniques for the evaluation of trust and reputation systems. For example in [17], a simulator implemented in Java is proposed as testbed (the ART testbed) enabling a competition forum for evaluating trust systems. In this case, no networking or other real world aspects are taken into account. Other examples of testbed are TREET [16] and the one proposed in [13]. The latter testbed is used for the evaluation of robustness of reputation systems. Specifically, this proposal focuses on robustness against unfair ratings, i.e. against parties that release score that intentionally under-estimate interaction outcome. The TREET testbed is proposed as an alternative to ART in order to overcome the limitations of ART. Indeed, the authors claim that ART is not well-suited for general-purpose experimentation of reputation systems, it has indeed agents evaluation as its main purpose. Instead TREET is designed specifically to support general-purpose experimentation and evaluation.

All these proposals are simulators or designs of testbeds that focus on marketplace applications. Our proposal, instead, does not fix a specific environment in which parties interact, but we use interactions as an abstraction of any parties relation. Moreover, we explicitly focus on probabilistic trust and reputation systems and on how they are evaluated. Our work aims at filling the gap between simulation and implementation of reputation systems, where networking aspects may play an important role when choosing and tuning trust and reputation systems. Indeed such aspects must be considered when implementing these systems. Specifically, problems such as how to rate parties when interactions are affected by network delays, or how to rate parties that are sporadically connected, have to be addressed. For this reason, reputation systems in NEVER are specified so that such problems can be taken into account by users when evaluating the systems. Indeed, they can be tuned on the basis of the features of the underlying network infrastructure exploited by NEVER for the execution.

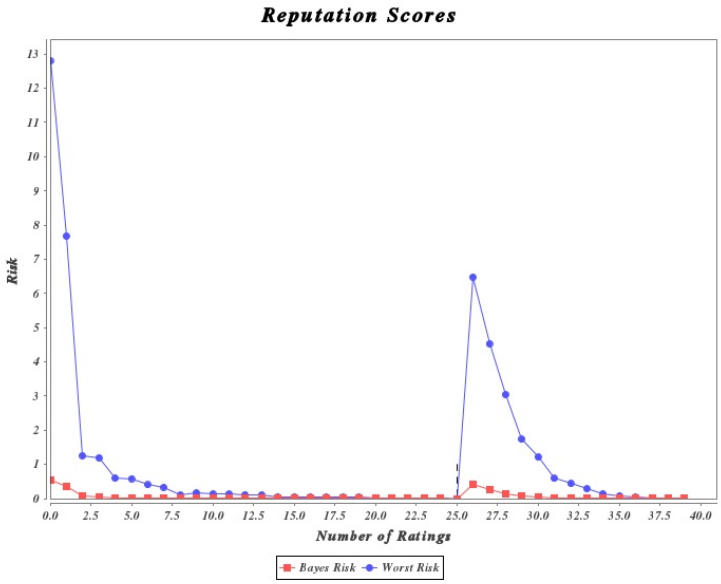

Fig. 7: Bayes and worst risk trends for the overall system

Future work. We intend to continue our analysis programme by considering other reputation models proposed in the literature. Some of the models that we plan to consider in the near future are those surveyed in [20, 15].

Apart from considering richer reputation models, we intend to extend our investigation to reputation systems over network architectures that rely on distributed rating servers, rather than a single centralised one. Examples of such systems can be found in literature; many authors have proposed adaptations of trust models for decentralised architectures. A reputation model adapted to ad-hoc networks for enhancing collaborations is proposed in [18]. For evaluating the relationships among devices in pervasive computing environments, a trust management scheme is introduced in [8], while [1] presents data structures and algorithms for assessing trust in a peer-to-peer environment. In particular, we intend to study how different underlying network architectures affects the performances of a given reputation model.

It is our intention to extend the tool to process real data from applications. The tool would be embedded in real applications and used to evaluate reputations systems in such environments. Applications could use reputation models in two different modalities: active or passive. In the active case, parties would compute reputation scores and use them to drive their interactions. In this modality the behaviour of an application would be modified by the deployed reputation system. In the passive case, the tool would collect rating values, compute reputation scores and just store them, without using such data to drive parties' interactions. The computed information would thus be used only for evaluating reputation systems. The passive modality would be useful in case of applications already deployed and in production. In this case it is important to understand how the application's behaviour would change before altering it. The passive modality could be also used for monitoring applications relying on existing reputation

systems and contrast their reputation models with respect to the models implemented in our tool.

References

1. K. Aberer and Z. Despotovic. Managing trust in a peer-2-peer information system. In *CIKM*, pages 310–317. ACM, 2001.
2. L. Bettini. KLAVA: a Java package for distributed and mobile applications. Reference manual Version 2. Technical report, Università di Firenze, March 2011.
3. L. Bettini, R. De Nicola, D. Falassi, M. Lacoste, and M. Loreti. A Flexible and Modular Framework for Implementing Infrastructures for Global Computing. In *DAIS*, volume 3543 of *LNCS*, pages 181–193. Springer, 2005.
4. L. Bettini, R. De Nicola, and R. Pugliese. Klava: a Java Package for Distributed and Mobile Applications. *Software - Practice and Experience*, 32(14):1365–1394, 2002.
5. M. Boreale and A. Celestini. Asymptotic Risk Analysis for Trust and Reputation Systems. In *SOFSEM*, volume 7741 of *LNCS*, pages 169–181. Springer, 2013.
6. A. Celestini, R. De Nicola, and F. Tiezzi. Specifying and Analysing Reputation Systems with a Coordination Language. In *SAC*. ACM, 2013. To appear.
7. R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: A Kernel Language for Agents Interaction and Mobility. *Transactions on Software Engineering*, 24(5):315–330, 1998.
8. M.K. Deno and T. Sun. Probabilistic trust management in pervasive computing. In *EUC*, volume 2, pages 610–615. IEEE Computer Society, 2008.
9. Z. Despotovic and K.Aberer. A Probabilistic Approach to Predict Peers' Performance in P2P Networks. In *CIA*, volume 3191 of *LNCS*, pages 62–76. Springer, 2004.
10. D. Gambetta. *Trust: Making and Breaking Cooperative Relations*, chapter 13: Can We Trust Trust?, pages 213–237. Basil Blackwell, 1988.
11. D. Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
12. Zimory GmbH. Zimory Enterprise Cloud, 2012. Web site: `http://www.zimory.de`.
13. A. A. Irissappane, S. Jiang, and Jie Zhang. Towards a comprehensive testbed to evaluate the robustness of reputation systems against unfair rating attack. In *UMAP Workshops'12*, 2012.
14. A. Jøsang and R. Ismail. The beta reputation system. In *Bled Conference on Electronic Commerce*, 2002.
15. A. Jøsang, R. Ismail, and C. Boyd. A survey of trust and reputation systems for online service provision. *Decision Support Systems*, 43(2):618–644, 2007.
16. R. Kerr and R. Cohen. TREET: the Trust and Reputation Experimentation and Evaluation Testbed. *Electronic Commerce Research*, 10:271–290, 2010.
17. K.K. Fullam et al. A specification of the Agent Reputation and Trust (ART) testbed: experimentation and competition for trust in agent societies. In *AAMAS*, pages 512–518. ACM, 2005.
18. C.T. Nguyen, O. Camp, and S. Loiseau. A bayesian network based trust model for improving collaboration in mobile ad hoc networks. In *RIVF*, pages 144 –151. IEEE, 2007.
19. J. Sabater and C. Sierra. Regret: reputation in gregarious societies. In *AGENTS*, pages 194–195. ACM, 2001.
20. J. Sabater and C. Sierra. Review on computational trust and reputation models. *Artif. Intell. Rev.*, 24:3360, 2005.
21. V. Sassone, K. Krukow, and M. Nielsen. Towards a formal framework for computational trust. In *FMCO*, volume 4709 of *LNCS*, pages 175–184. Springer, 2006.
22. G. Zacharia and P. Maes. Trust management through reputation mechanisms. *Applied Artificial Intelligence*, 14(9):881–907, 2000.

INSTITUTE FOR ADVANCED STUDIES LUCCA