Network-aware Evaluations of 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. Reputation systems are nowadays widely used to support decision making in networked systems. Parties in such systems rate each other and use shared ratings to compute reputation scores that drive their interactions. The existence of many reputation systems with remarkable differences calls for software frameworks for describing, implementing and comparing their performances while taking into account the architecture of the network where the systems have to be deployed. We tackle this problem through a software framework for network-aware evaluations of reputation systems based on the notion of probabilistic trust. Specifically, we describe a tool for rapid prototyping and evaluation of reputation system models, which takes explicitly into account the networked execution environment. To implement specific models we just enrich KLAVA (a network-aware extension of Java) with additional classes. To assess performances of the resulting reputation systems, their implementations are analysed by a control manager that performs experiments according to user-specified parameters. The developed framework relies on the formal foundations of KLAIM, a network-aware coordination language, and its Java implementation KLAVA. Feasibility and effectiveness of our proposal is demonstrated by reporting on the analysis of two simple models of reputation systems.

Keywords: Reputation systems, Network-awereness, Formal methods

1 Introduction

The growth witnessed in the e-commerce industry in the last years is impressive and is likely to continue. The online provision of services through open computer networks involves an increasing number of customers, that have to deal with new issues, not encountered in traditional forms of commerce. Nowdays, there is no physical contact with providers and it is relatively easy and cheap to build an online shop or put items on sale in existing mediator sites. This has further reduced the information in possession of customers about providers, and the goods and services they offer. To mitigate this inconvenience, *reputation systems* are more and more used to support decision making in commercial online applications. The success of reputation systems in this area has stimulated their application also in other contexts, e.g ad-hoc networks, sensor networks, P2P networks, where the parties are likely to be disconnected from their preferred security infrastructures and have to rely on other means to build up confidence in their partners.

In a reputation system the involved parties rate each other, e.g. after completing an interaction, and use the aggregated ratings about a given party to derive a *reputation*

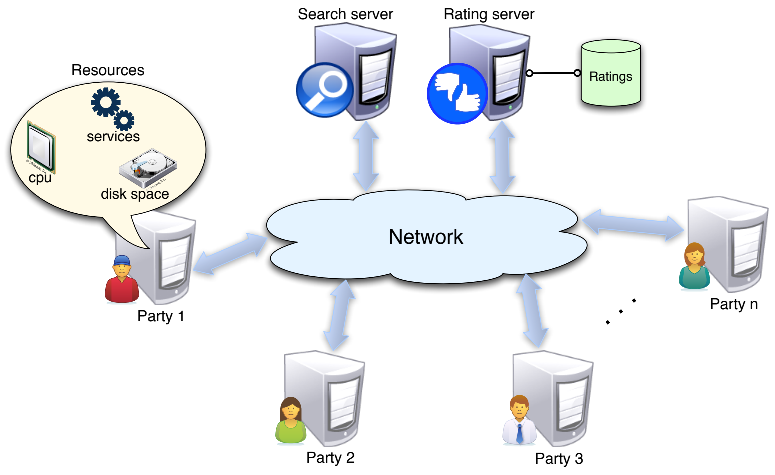


Fig. 1: General infrastructure of a reputation system

score, i.e. a collective measure of trustworthiness based on the ratings from the members of a community. Such computed reputation is used to assist the other parties in deciding whether to interact with a specific partner.

A *networked trust infrastructure* allows parties of a reputation system to interact and to exchange ratings. In this paper, we consider a general infrastructure, graphically depicted in Figure 1, where a *rating server* collects all 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 client, provider, or both, and may offer different kinds of resources. Therefore, when a party needs a resource, it queries the search server to get the list of parties providing it, and retrieves the corresponding ratings of each of these parties. Then, it selects one of the providers with the highest reputation score and, after the interaction, rates it according to the quality of the provided resource.

Due to the widespread diffusion of reputation systems, research work on them is intensifying and many different systems have been (and are still) proposed, often developed from scratch without considering existing approaches. In fact, on top of the general networked infrastructure mentioned above, many different kinds of reputation systems can be layered, which mainly differ for the model they use to aggregate ratings when computing reputation scores. This calls for a methodological approach for describing, implementing and comparing different reputation systems while taking into account the network architecture where they have to be deployed. In this paper, we tackle this issue by proposing a software framework for network-aware evaluations of reputation systems. To model the behaviour of the parties involved in these systems, we rely on the probabilistic approach to trust [10, 8], according to which the parties' behaviour is rendered through a probability distribution. More specifically, we present a tool for rapid prototyping of Java-based implementations of reputation system models, devised to run in a networked execution environment. To assess performances of the resulting reputation systems and comparing them among each other, the obtained implementations are handled by a control manager that performs experiments according to user-specified parameters.

The proposed framework takes advantage of the coordination language KLAIM [6], specifically designed for network-aware programming of distributed applications and for reasoning about them. For the implementation, we rely on the Java library KLAVA [4], which provides a run-time support for KLAIM actions within Java code. In particular, to implement specific reputation models we enrich KLAVA with additional classes implementing a given interface. In this way, our framework enables a dynamic analysis, based on experiments carried out in a real network environment rather than through simulation of mathematical models as for most of the proposals in the literature. The formal foundations underlying the framework paves the way for the use of formal tools and techniques already developed for KLAIM (see, e.g., [15]) to support the verification of reputation system models. Moreover, the prototypical implementations used here for evaluation purposes could be exploited as the basis of a real world Java implementations of reputation systems.

To demonstrate the feasibility and effectiveness of our proposal we have analysed two simple models of reputation systems, namely the Beta model [10] and a model based on the maximum likelihood estimation [8], and we have shown that the reputation scores in the latter model converge more rapidly to the right estimations of parties' behaviour than in the former one.

The rest of the paper is organized as follows. Section 2 provides a brief overview of KLAIM and KLAVA. Section 3 presents the KLAIM model of reputation system underlying our framework. Section 4 describes the architecture and functional principles of the framework, while Section 5 reports on the analysis of two models of reputation systems. Finally, Section 6 concludes the paper by also reviewing some of the related work and suggesting directions for future work.

2 The formal language KLAIM and the Java library KLAVA

In this section we informally present KLAIM[1], a coordination language specifically designed for modelling mobile and distributed applications and their interactions, which run in a network environment. Then, we provide a brief overview of KLAVA, a Java library implementing the run-time support for KLAIM actions. We refer the interested reader to [6] and [4] for a more complete account of KLAIM and KLAVA, respectively, and to [5] for a survey on research work based on KLAIM.

KLAIM specifications consist of *nets*, namely finite plain collections of nodes where *components*, i.e. processes and data tuples, can be allocated. Nodes are composed by means of the parallel composition operator $\_ \| \_$. At net level, it is possible to restrict the visibility scope of a name s by using the operator $(\nu s)\_$: in a net of the form $N_1 \| (\nu s)N_2$, the effect of the operator is to make s invisible from within the subnet N_1.

[1] We consider in this paper a version of KLAIM enriched with standard control flow constructs (i.e., assignment, if-then-else, sequence, etc.), which simplify the modelling task. Although such constructs were not included in the original presentation of the language [6], they can be easily rendered in KLAIM and are directly supported by related tools.

Nodes have the form $s\ ::_\rho\ C$, where s is a unique *locality name* (i.e., a network address), ρ is an allocation environment, and C is a set of hosted components. An *allocation environment* provides a name resolution mechanism by mapping *locality variables* l (i.e., aliases for addresses), occurring in the processes hosted in the corresponding node, into localities s. The distinguished locality variable **self** is used by processes to refer to the address of their current hosting node. In the rest of this section, we will use ℓ to range over locality names and variables.

Processes are the KLAIM active computational units and may be executed concurrently either at the same locality or at different localities. They are built up from the process **nil**, which does nothing, and from basic actions by means of sequential composition $\_;\_$, parallel composition $\_\,|\,\_$, conditional choice **if** (e) **then** $\{\_\}$ **else** $\{\_\}$, the iterative constructs **for** $i\ =\ n$ **to** $m\ \{\_\}$ and **while** $(e)\ \{\_\}$, and process definition $A(f_1,\ldots,f_n) \triangleq \_$, where A is a *process identifier* and the formal parameters f_i are pairwise distinct. Notably, e ranges over *expressions*, which are formed from basic values (booleans, integers, strings, floats, etc.) and variables by using standard operators on values and the non-blocking retrieval actions **inp** and **readp** (explained below).

During their execution, processes perform some basic *actions*. Actions **in**$(T)@\ell$ and **read**$(T)@\ell$ are retrieval actions and permit to withdraw/read *data tuples* (i.e. sequences of values) from the tuple space hosted at the (possibly remote) locality ℓ: if a matching tuple is found, one is non-deterministically chosen, otherwise the process is blocked. These actions exploit templates as patterns to select tuples in shared tuple spaces. *Templates* are sequences of actual and formal fields, where the latter are written $!\,x$ or $!\,l$ and are used to bind variables to values or locality names, respectively. Actions **inp**$(T)@\ell$ and **readp**$(T)@\ell$ are non-blocking versions of the retrieval actions: namely, during their execution processes are never blocked. Indeed, if a matching tuple is found, **inp** and **readp** act similarly to **in** and **read**, and additionally return the value *true*; otherwise, they return the value *false* and the executing process does not block. Actions **inp**$(T)@\ell$ and **readp**$(T)@\ell$ can be used where either a boolean expression or an action is expected (in the latter case, the returned value is simply ignored). Action **out**$(t)@\ell$ adds the tuple resulting from the evaluation of tuple t (which may contain expressions) to the tuple space of the target node identified by ℓ, while action **eval**$(P)@\ell$ sends the process P for execution to the (possibly remote) node identified by ℓ. Actions **out** and **eval** are both non-blocking. Finally, action **newloc** creates new network nodes, while action $x\ :=\ e$ assigns the value of e to x. These latter two actions, differently from all the others, are not indexed with an address because they always acts locally.

The tuple-based communication model underlying the KLAIM language is particularly suitable for implementing distributed applications that run over a heterogeneous network. Therefore, a Java package, called KLAVA, has been developed to programme such applications according to the KLAIM coordination paradigm. This package 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 nodes that can be connected to the net in order to build the desired network environment. An abstract class is then provided to create processes to be added to the nodes, by means of instantiation of subclasses specialized through inheritance and method overriding.

3 Klaim model of reputation system

In this section, we present the Klaim model of reputation system underlying our evaluation framework. For the sake of readability, we report the Klaim specification rather than the corresponding Klava code; we refer to [2] for the complete Java source and binary code of the framework.

We consider here the general infrastructure of a reputation system described in Section 1 and graphically depicted in Figure 1. It can be rendered in Klaim as follows:

$$\begin{array}{l} s_{search\_server} \ ::_{\rho_{search\_server}} \ \langle \text{“}type\_1\text{”}, s_{party\_j}\rangle \ | \ \ldots \ | \ \langle \text{“}type\_m\text{”}, s_{party\_h}\rangle \ | \ A_{search} \\ \| \ s_{rating\_server} \ ::_{\rho_{rating\_server}} \ \langle \text{“}query\_lock\text{”}, s_{party\_1}\rangle \ | \ \ldots \ | \ \langle \text{“}query\_lock\text{”}, s_{party\_n}\rangle \\ \| \ s_{party\_1} \ ::_{\rho_{party\_1}} \ A_{party\_1} \ \| \ \ldots \ \| \ s_{party\_n} \ ::_{\rho_{party\_n}} \ A_{party\_n} \end{array}$$

where $\rho_{search\_server} = \{\textbf{self} \mapsto s_{search\_server}\}$, $\rho_{rating\_server} = \{\textbf{self} \mapsto s_{rating\_server}\}$, and $\rho_{party\_i} = \{\textbf{self} \mapsto s_{party\_i},\ l_{rating} \mapsto s_{rating\_server},\ l_{search} \mapsto s_{search\_server}\}$, with $i = 1..n$.

The tuple space of the search server contains tuples of the form $\langle \text{“}type\_i\text{”}, s_{party\_k}\rangle$ stating that the party s_{party_k} provides resources of type i. In the considered setting, a party can provide infinitely many times a given resource (i.e. we consider resources such as files, services, etc.). The model, as well as the framework, can be easily extended to settings dealing with non-persistent resources (e.g., by representing them as tuples).

The tuple space of the rating server contains, instead, a tuple of the form $\langle \text{“}query\_lock\text{”}, s_{party\_i}\rangle$ for each party s_{party_i} belonging to the net. This tuple acts as a lock that must be acquired before querying the rating server for retrieving the ratings about the party s_{party_i}. Notably, such tuples are used just in the Klaim specification.

It is worth noticing that tuples $\langle \text{“}type\_j\text{”}, s_{party\_i}\rangle$ and $\langle \text{“}query\_lock\text{”}, s_{party\_i}\rangle$ are inserted into the tuple spaces of the two servers when the party node s_{party_i} connects to the net (this step is not described in the Klaim model). Moreover, for the sake of simplicity, in the Klaim specification, once party nodes and servers are connected to the net they will not disconnect. The Klava implementation, instead, deals with (unexpected or programmed) node disconnections, by setting timeouts through the *timed* versions of the blocking Klaim actions (see [4]).

The rating server does not execute any process (i.e. it simply acts as a store of tuples), while the following recursive process runs on the search server:[2]

```
$A_{search\_server} \triangleq$
  $n_{providers} := 0$;
  // get a search request to be processed
  $\textbf{in}(\text{“}search\_request\text{”}, !l_{requester}, !res\_type)@\textbf{self}$;
  // read (and consume) all the ‘resource type-provider’ tuples for the given resource type
  $\textbf{while}\ (\textbf{inp}(res\_type, !l)@\textbf{self})$ {
      // increase the counter of providers
      $n_{providers} := n_{providers} + 1$;
      // add the current provider to the list sent to the requester
      $\textbf{out}(\text{“}list\text{”}, n_{providers}, l)@l_{requester}$;
      // store the current tuple in a temporary local tuple
```

[2] The specification code reported in this section is made self-explanatory through the use of comments (the string // indicates that the rest of the line is a comment).

```
        out("$tmp$", $res\_type$, $l$)@self
    };
    // send the length of the list to the requester
    out("$list\_length$", $n_{providers}$)@$l_{requester}$;
    // recreate the consumed ‘resource type-provider’ tuples
    while (inp("$tmp$", !$res\_type$, !$l$)@self) {
        out($res\_type$, $l$)@self
    }; $A_{search\_server}$
```

The search server processes one request at a time; for each requested type of resource it determines a list of providers by exploring the information in its tuple space. This list is rendered in Klaim as a set of tuple of the form $\langle$“*list*”, i, $s_{party\_j}\rangle$ indicating that the i-th element of the list is the provider s_{party_j}. Notably, in the Klaim specification above, to read all matching tuples only once in a request processing, the ‘resource type-provider’ tuples are first consumed and then reinserted into the tuple space. Instead, the corresponding Klava implementation of this process performs the same task in an equivalent, but more efficient, way by exploiting a built-in mechanism that prevents matching twice the same tuple in this kind of loops.

Depending on the processes running in the party nodes, each party s_{party_i} can play two roles: it can provide resources, require resources, or both. We consider here the latter case, the more complete, where A_{party_i} is defined as follows:

$$A_{party\_i} \triangleq A_{provider\_i} \mid A_{client\_i}$$

The process for the provider role is defined as follows:

```
$A_{provider\_i}$ $\triangleq$
    // wait for a new resource request
    in("$request$", !$l_{requester}$, !$resource\_type$)@self;
    // get the quality of the resource to be provided according to its behaviour
    $quality$ := $getResourceQuality(party\_i\_behaviour, resource\_type)$;
    // provide the resource to the requester
    out("$resource$", $resource\_type$, $quality$)@$l_{requester}$;
    $A_{provider\_i}$
```

The provider specified above serves one client at a time, but the specification could be easily modified to serve multiple requests. The processing of a request is based on the function *getResourceQuality*, that takes into account the party behaviour and the type of the resource. The definition of such function and that of resource quality may vary from a reputation system to another. Therefore, these are some of the parameters defining a specific reputation system that must be specified in order to implement it and evaluate its performances.

The process for the client role, instead, is defined as follows:

```
$A_{client\_i}$ $\triangleq$
    // randomly select a resource type
    $res\_type$ := $getResourceType()$;
    // query the search server for obtaining a list of parties providing a resource of type $res\_type$
    out("$search\_request$", self, $res\_type$)@$l_{search}$;
    // read (and consume) the number of providers
    **in**(“list_length”, !m)@**self**;
    // initialize the tuple containing the locality of the most trusted party and its reputation value:
    // the reputation value NO_ONE indicates that there is no provider for the requested resource
    **out**(“most_trusted_party”, **self**, NO_ONE)@**self**;
    // read (and consume) the list sent by the search server
    **for** $j = 1$ **to** $m$ {
        // get an element of the list
        **in**(“list”, $j$, $!l_{provider}$)@**self**;
        // compute the reputation value of $j$-th party
        $A_{evaluate\_reputation}(l_{provider})$
    };
    // request the resource to the most trusted party
    $A_{request}(res\_type)$;
    $A_{client\_i}$
```

A client cyclically chooses a type of resource to request, retrieves a list of parties providing this type of resource from the search server, determines the most trustworthy parties (see process $A_{evaluate\_reputation}$ below) and requests the resource to one of them (see process $A_{request}$ below). Notably, to select a resource type, the above process exploits the function *getResourceType*, which returns a random type from “*type*_1” and “*type*_*m*” according to a uniform distribution. The constant NO_ONE is defined within our framework to indicate that there exists no provider for a requested resource.

The process $A_{evaluate\_reputation}$ is defined as

```
$A_{evaluate\_reputation}(l) \triangleq$
    // acquire the lock for querying the rating server about party $l$
    **in**(“query_lock”, $l$)@$l_{server}$;
    // read (and consume) the rating values of party $l$
    **while** (**inp**($!l_{rater}$, $l$, $!rate$)@$l_{server}$) {
        // store a local copy of the rating value
        **out**($l_{rater}$, $rate$)@**self**
        // store a temporary copy of the rating tuple to repopulate the rating server’s tuple space
        **out**($l_{rater}$, $l$, $rate$)@**self**;
    };
    // recreate the rating values of party $l$ in the rating server
    **while** (**inp**($!l_{rater}$, $l$, $!rate$)@**self**) {
        **out**($l_{rater}$, $l$, $rate$)@$l_{server}$
    };
    // release the lock
    **out**(“query_lock”, $l$)@$l_{server}$;
    // check if at least one rate exists for party $l$
    **if** (**readp**($!l_{rater}$, $!rate$)@**self**) **then**{
        // compute the reputation of party $l$ by exploiting (and consuming) the local rating tuples
        $reputation$ := $evaluateReputation$(**self**, $l$)
    } **else** {
        // the reputation is set to a constant value
        $reputation$ := NO_RATINGS
    };
    // update the most trusted party
    in(“$most\_trusted\_party$”, $!l_{trusted}$, $!reputationMostTrusted$)@self;
    if ($reputationMostTrusted$ < reputation) then{
        out(“$most\_trusted\_party$”, $l$, $reputation$)@self
    } else {
        out(“$most\_trusted\_party$”, $l_{trusted}$, $reputationMostTrusted$)@self
    }
```

To properly retrieve all rating values about the party l, the process must acquire a lock from the rating server, which will be released only after all consumed rating tuples will be reinserted into the server's tuple space. Like for process A_{search_server}, the corresponding KLAVA code implements this task more efficiently. The most relevant part of the above process is the computation of the party reputation through the function *evaluateReputation*, which differs from a reputation system to another and, hence, is one of the parameters that must be specified to implement a reputation system in our framework. Similarly, the constant NO_RATINGS is a parameter of the system and specifies the reputation assigned by default to providers without rating values (e.g. 0 means that providers without rating values are not trusted, while 0.5 means that they have an ‘average' reputation).

Finally, the process $A_{request}$ is defined as

```
$A_{request}(type)$ $\triangleq$
    // get the most trustworthy provider
    in(“$most\_trusted\_party$”, $!l_{trusted}$, $!reputationMostTrusted$)@self;
    // check if the most trustworthy provider has enough reputation
    if (MIN_REPUTATION $\leqslant$ $reputationMostTrusted$) then{
        // send the resource request to the most trustworthy provider
        out(“$request$”, self, $type$)@$l_{trusted}$;
        // receive the resource
        in(“$resource$”, $type$, $!quality$)@self;
        // check the quality of the resource and rate the provider
        $rate$ := $rateProvider(quality)$;
        // send the rate to the rating server
        out(self, $l_{trusted}$, $rate$)@$l_{server}$
    }
```

Again, the constant MIN_REPUTATION and the function *rateProvider* are parameters of the system. The constant MIN_REPUTATION specifies the minimum reputation required by the client for an interaction with a provider, while the function *rateProvider* returns a rating value for the invoked provider by possibly taking into account runtime measures (e.g. response time).

4 The evaluation framework

In this section, we present the architecture of our framework, describe how an evaluation is carried out and explain how to use our tool for prototyping reputation system models.

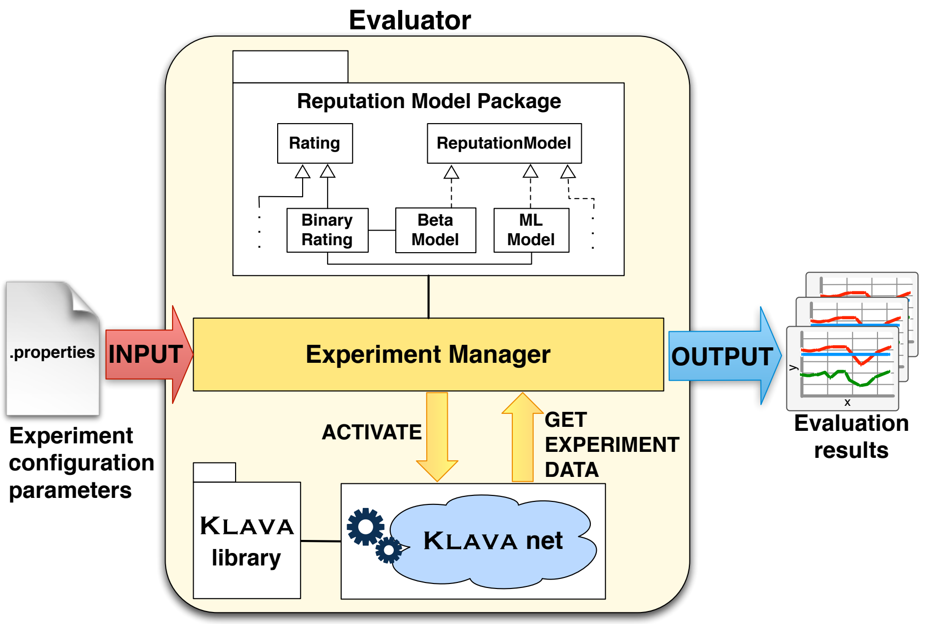


Fig. 2: Framework workflow

The framework[3] is composed by three main parts: (1) the reputation model package, (2) the experiment manager, and (3) the KLAVA net. The reputation model package contains the Java classes implementing the reputation system models. The experiment manager is a Java class that manages the execution of each experiment. Finally, the KLAVA net exploits the KLAVA library to implement the network where reputation systems are deployed. The architecture of the framework, as well as its workflow, are graphically depicted in Figure 2.

In the following we report what goes on from the user setting of configuration parameters to the output of evaluation results. First, the user sets the configuration parameters for each evaluation, writing them on a configuration file. The configuration file is a Java *properties* file that is read by the tool after the launching. Some of the most significative configuration parameters are the number of parties in the system, the number of different resource types that parties can provide, the set of parties' behaviours, the trust model to use, and the possibility of randomly selecting providers, instead of choosing one of those with the best reputation score. The latter parameter allows a client party to choose a provider independently from the providers' reputation scores. In this way, each provider in the system will have the same number of rating values on the average. This may turn out to be useful to compare the trends of reputation scores associated to parties having a 'bad' behaviour. In fact, without random choice, after a while such parties would not be chosen and rated by clients (see Section 5, Figure 5).

The tool then uses the input parameters to create the network: a network node is created for each of the two servers and for each party in the system. Once the network

[3] This is a free software; it can be redistributed and/or modified under the terms of the GNU General Public License as published by the Free Software Foundation.

is completely set up, the reputation system (configured according to user's parameters) is deployed on it and the experiment starts, i.e. network components are enabled so that system parties can interact and rate each other. During the activity of the network, data about interactions are stored for a later analysis. The user sets the duration of each experiment and the number of experiments to execute through configuration parameters. Experiments are repeated in order to reach the desired precision; thus the procedure of starting and stopping experiments is executed till the last experiment is accomplished. Afterwards, data are analysed and provided as output, also in form of charts[4], by the tool. We refer to Section 5 for a deeper description of data analysis.

For what concerns extending our tool to other reputation system models, it is possible to implement them through the specification of few Java classes: for each new model, the user has to create a class implementing the `ReputationModel` interface and, possibly, a class extending the abstract class `Rating`. The class implementing the interface defines how reputation scores are computed, which rating values are used by the system and how parties in the system evaluate the interactions. The class extending `Rating` defines the set of rating values used in the system and how to store them.

The framework includes the implementation of two reputation system models: the first based on the Beta model [10], and the second based on maximum likelihood estimation [8]. Both reputation systems use *binary ratings*, i.e. system parties in both models rate each other in a binary way: an interaction can be either 'satisfactory' or 'unsatisfactory'. Therefore, a class has been created for each model and a single class has been created for binary ratings. The latter class is used by both models, but each of them specifies how to specifically use the rating values inside.

In the following we describe the two models in more details. Let $\mathcal{P}$ be a set of party identities, the behaviour of each party $p \in \mathcal{P}$ is assumed to be probabilistic, in the sense that there is a fixed probability $\theta_p \in [0, 1]$ that an interaction with the party p will be satisfactory. In a reputation system, the goal is to predict parties' behaviour in future interactions, given the rating values about past interactions, i.e. determining an estimation $\tilde{\theta}_p$ of the unknown parameters θ_p. The sequence of rating values $x_p^n = x_{1p}, \ldots, x_{np}$, with $x_{ip} \in \{0, 1\}$, about past interactions with party p is considered as realization of a sequence of independent, identically distributed (*i.i.d.*) random variables $X_p^n = X_{1p}, \ldots, X_{np}$.

The implemented models assume that random variables X_{ip} are distributed according to a Bernoulli distribution with success probability θ_p. This means that, when interacting with a party p, the probability that the i-th interaction is satisfactory, given θ_p the behaviour of party p, is

$$Pr(\text{satisfactory} \mid \theta_p) = \theta_p$$

The reputation system based on the Beta model seeks to estimate the a posterior distribution for the value θ_p, given the results of past interactions with party p. The model uses a conjugate prior distribution, specifically a *beta prior*. Hence, the a posterior distribution results in a beta distribution. 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 $\beta \geq 0$, where

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

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

The reputation system based on maximum likelihood estimation[5] seeks to find a value $\tilde{\theta}_p$ which maximises the following likelihood expression $L(\theta)$:

$$L(\theta) = Pr(X_p^n \mid \theta) = \prod_{i=1}^{n} Pr(X_{ip} = x_{ip} \mid \theta)$$

The resulting θ is the party's reputation score $\tilde{\theta}_p$.

5 The framework at work

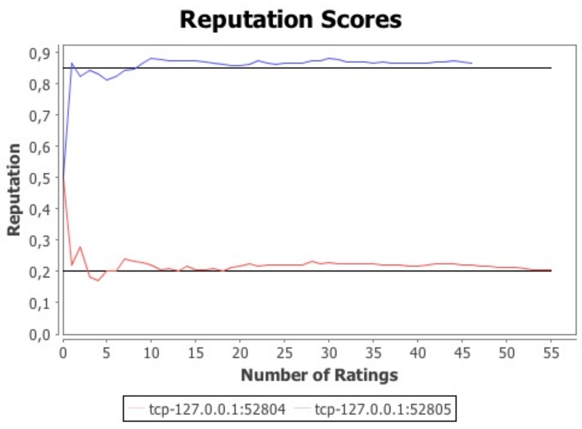
In this section we illustrate how reputation systems are evaluated by using our framework through two exemplifying experiments, whose results are reported in Figures 3 and 4. The charts have been obtained by specifying the following configuration parameters. The considered reputation systems have two parties, both acting as client and provider; one party behaves according to a Bernoulli distribution with success probability $\theta_1 = 0.85$, while the other with success probability $\theta_2 = 0.20$. The Beta model and ML model are used for calculating reputation scores. Only one resource is supplied by the providers and required by clients. Providers of a given resource are chosen randomly. Figure 5 shows what happens when providers selection is not random, i.e. the party with a better behaviour is chosen more often than the other one. Random choice gives the possibility of analysing the trend of party's reputation scores also in the case of a bad behaving party that would have a very low number of rating values. Moreover, the random choice shows what happens when reputation systems are not used, i.e. when reputation scores are not considered for selecting a provider.

The charts in Figure 3a and Figure 3b present the trend of each single party's reputation in the system. On the x-axis we find the numbers of rating values used to compute reputation scores, on the y-axis the reputation scores. The behaviours of the two parties, denoted respectively by $\theta_1 = 0.85$ and $\theta_2 = 0.20$, are thus represented in the charts as black horizontal lines. Instead, the trends of their reputation scores are indicated by the blue and red lines, respectively. The charts show that the ML model converges more rapidly to the right estimation of party's behaviour than the Beta model. In the case of the Beta model, the convergence is slower but smoother. The Beta model is less reactive in the presence of few rating values than the ML model, while it is more robust than the ML model, i.e. party's reputation scores change more brusquely in the ML model.

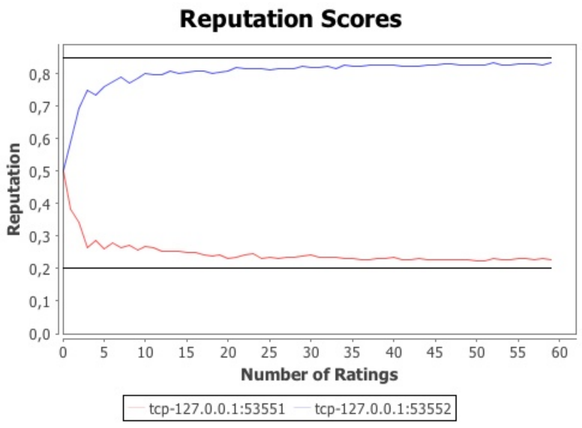
The same comments apply looking at the charts in Figure 4a and Figure 4b that show the error trend between parties' reputation scores and parties' behaviour. On the x-axis we find the numbers of rating values, on the y-axis the estimation errors. Such errors are assessed by computing the Kullback-Leibler divergence [14] between each party's reputation and the corresponding party's behaviour, as proposed in [17].

For each experiment, the tool creates some log files containing data used for evaluation purposes and additional information about the experiment setting. Such data are

[5] In the rest of the paper, we refer to this model of reputation system as *ML model*.



(a) ML model



(b) Beta model

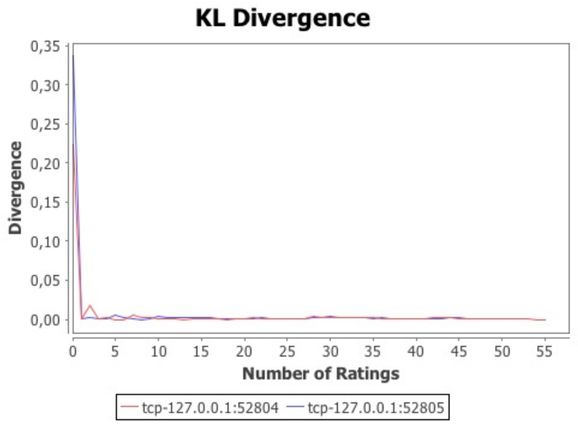
Fig. 3: Evaluation results: parties' reputation

then exploited to automatically generate party's reputation scores charts and Kullback-Leibler divergence charts for each single experiment run and for the aggregate data when all runs are completed.

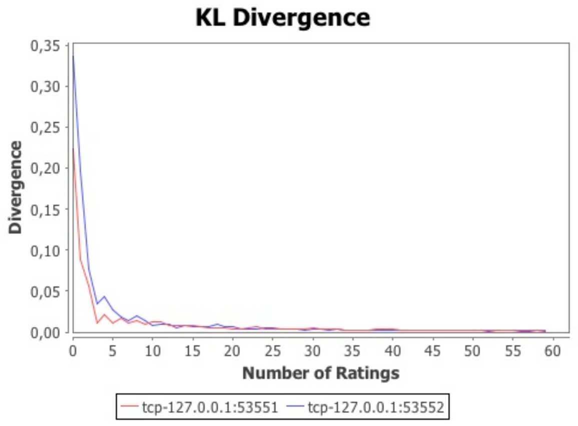
6 Concluding remarks

In this paper, we provide a formal-based framework for evaluation of reputation system models, which permits to rapidly implement them and analyse their executions in a networked running environment. To illustrate our approach, we have shown the results of the analysis of two well-known models of reputation systems.

Related work. The terminology used in the literature to describe the kind of systems considered in this paper is sometimes quite confusing, due to the usage of the term *trust* in different contexts with a variety of meanings. In fact, trust and reputation are often used as synonyms. The difference between the two concepts is clarified in [11]. Accord-



(a) ML model



(b) Beta model

Fig. 4: Evaluation results: estimation errors

ing to this survey, trust is based on a subjective measure of reliability of a given party, derived from some private knowledge (e.g. past direct interactions). Instead, reputation relies on an objective measure derived from referrals or ratings provided by other parties. Therefore, by adopting such distinction, our work focusses on reputation.

There are many works in the literature whose goal is the evaluation and comparison of reputation systems. However, to the best of our knowledge, ours is the first effective framework allowing the evaluation of reputation systems in a real networked execution environment. In fact, most of the existing 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 in distributed systems. For example, [17] propose a measure, based on the Kullback-Leibler divergence, for a mathematical comparisons of probabilistic computational trust systems. Such measure is then applied in [13] to compare trust systems in ubiquitous computing. We have exploited the same measure for calculating the estimation errors shown in Figure 4.

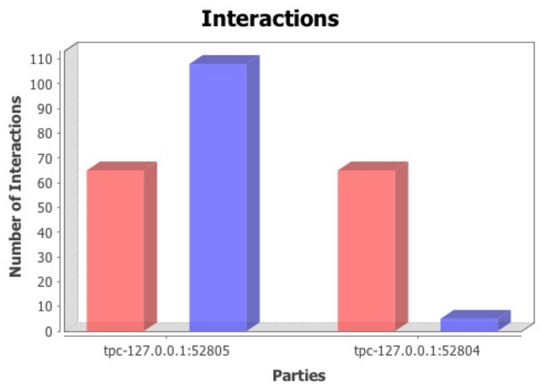


Fig. 5: Providers selection

Other works exploit computational software programs [20] or software for multi-agent modelling [19], for running simulation of their reputation models. As an example, in [18] the RePast simulator [1] is used to conduct various studies on reputation systems in a multi-agent context. In particular, simulations with increasing amount of hostile agents in the system have been performed in order to find the critical point, where the system fails to compute dependable reputations. The aim of such studies is to simulate systems under extreme conditions, while we are more interested on the real world behaviour, which also takes into account the networking aspects. However, similar experiments could be also carried out in our framework. In the same way, we could perform analyses similar to those described in [9] aiming at evaluating the robustness of reputation systems. We leave this for future investigation. Another simulation-based approach is described in [12], where a simulator implemented in Java is proposed as testbed enabling a competition forum for evaluating trust systems. As in the cases above, no networking or other real world aspects are taken into account.

Future work. In this paper, we have shown our framework at work on two simple reputation system models, i.e. the Beta model and a model based on the maximum likelihood estimation. We intend to extend this programme to evaluate reputation models already defined in the literature, through experiments in a networked environments. Some models that we plan to take into account in the near future are those surveyed in [16, 11].

Apart from considering other models, we also intend to extend our investigation to reputation systems deployed on network architectures relying on distributed rating servers, rather than a centralised one. As an example of such decentralised architectures, we can consider a case (like, e.g., in P2P networks) where each party stores the rates about interactions and provides them to other parties. Thus, before interacting with a given party, one has to retrieve ratings about him from as many parties as possible. In particular, we intend to study how different underlying network architectures affects the performances of a given reputation model.

As a further work on our evaluation tool, we want to make it more usable by developing a graphical interface, exploiting Java libraries such as Swing or GWT, in order to guide users in setting and executing experiments.

Another line of research we want to explore is to enlarge our evaluation approach of reputation systems by applying other forms of analysis that still rely on the KLAIM model presented in Section 3. The KLAIM model, used here as a formal basis to develop Java implementations based on the KLAVA library, can also be used for analysis purposes by means of existing tools already developed for KLAIM. In particular, we believe that the use of the stochastic logic introduced in [7] for formalising the desired system's properties and the use of the corresponding analysis tool [15] could allow us to obtain results that would integrate those obtained by using our framework.

References

1. RePast. Web site: `http://repast.sourceforge.net`.
2. Source and binary code of the evaluation framework, 2012. Available at `http://cse.lab.imtlucca.it/rep_sys_eval/`.
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. L. Bettini et al. The Klaim Project: Theory and Practice. In *Global Computing*, volume 2874 of *LNCS*, pages 88–150. Springer, 2003.
6. 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.
7. R. De Nicola, J. Katoen, D. Latella, M. Loreti, and M. Massink. Model checking mobile stochastic logic. *Theor. Comput. Sci.*, 382(1):42–70, 2007.
8. 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.
9. A. Jøsang and J. Golbeck. Challenges for robust of trust and reputation systems. In *STM*, 2009.
10. A. Jøsang and R. Ismail. The beta reputation system. In *Bled Conference on Electronic Commerce*, 2002.
11. 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.
12. 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.
13. K. Krukow, M. Nielsen, and V. Sassone. Trust models in ubiquitous computing. *Phil. Trans. R. Soc.*, 366:37813793, 2008.
14. S. Kullback and R.A. Leibler. On information and sufficiency. *Annals of Mathematical Statistics*, 22(1):7986, 1951.
15. M. Loreti. SAM: Stochastic Analyser for Mobility, 2010. Available at `http://rap.dsi.unifi.it/SAM/`.
16. J. Sabater and C. Sierra. Review on computational trust and reputation models. *Artif. Intell. Rev.*, 24:3360, 2005.
17. 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.
18. A. Schlosser, M. Voss, and L. Brückner. Comparing and Evaluating Metrics for Reputation Systems by Simulation. In *Workshop on Reputation in Agent Societies*, 2004.
19. Y. Wang and J. Vassileva. Trust and reputation model in peer-to-peer networks. In *P2P*, pages 150–157. IEEE, 2003.
20. L. Xiong and L. Liu. A reputation-based trust model for peer-to-peer e-commerce communities. In *CEC*, pages 275–284. IEEE, 2003.