F. Moyano, C. Fernandez-Gago, and J. Lopez, "A Framework for Enabling Trust Requirements in Social Cloud Applications", *Requirements Engineering*, vol. 18, pp. 321-341, 2013.
http://doi.org/10.1007/s00766-013-0171-x
NICS Lab. Publications: https://www.nics.uma.es/publications

# A Framework for Enabling Trust Requirements in Social Cloud Applications

Francisco Moyano, Carmen Fernandez-Gago and Javier Lopez
Network, Information and Computer Security Lab
University of Malaga, 29071 Malaga, Spain
{moyano,mcgago,jlm}@lcc.uma.es

September 20, 2013

**Abstract**

Cloud applications entail the provision of a huge amount of heterogeneous, geographically-distributed resources managed and shared by many different stakeholders who often do not know each other beforehand. This raises numerous security concerns that, if not addressed carefully, might hinder the adoption of this promising computational model. Appropriately dealing with these threats gains special relevance in the social cloud context, where computational resources are provided by the users themselves. We argue that taking trust and reputation requirements into account can leverage security in these scenarios by incorporating the notions of trust relationships and reputation into them. For this reason, we propose a development framework onto which developers can implement trust-aware social cloud applications. Developers can also adapt the framework in order to accommodate their application-specific needs.

## 1 Introduction

Cloud computing has been gaining increasing attention over the years. This model comes from integrating and leveraging past, distributed computing models such as utility computing, Peer-to-Peer computing or volunteer computing, turning computational resources (i.e. CPU, bandwidth, storage, software, etc) into commodities that can be delivered at different abstraction layers [36]. One of the key benefits is that users can access services based on their needs without regard to where the services are hosted or how they are delivered [4]. If these services are software, the delivery model is called *Software as a Service*. Other benefits of cloud computing, such as the elasticity, scalability and the platform independence, to name just a few, are pushing its adoption and use forward.

However, the unique features of this model also raise security and privacy challenges [35]. Users are oblivious regarding the location of the services, and they may share the same resources with other users without even knowing their identity. This calls for mechanisms to ensure trust in the cloud infrastructure. This is even more important when the cloud providers are, in turn, users who are willing to share their own resources within a community, leading to the so-called *social cloud* [8], where trust and reputation may play a crucial role in order to make this scenario feasible and secure.

Although there is not any standard trust definition, it is often agreed that it can leverage security by assisting in decision-making processes. There is a huge amount of trust and reputation models in the literature, each one very context-dependent and targeting different domains. These models are usually built on top of an existing application in an ad-hoc way in order to match the specific needs of the application and its environment, limiting the models re-usability. Furthermore, most models do not distinguish explicitly between trust and reputation, nor do they provide guidelines to combine these notions to yield more solid results.

We believe that this approach is not adequate, and that a holistic approach where trust and reputation requirements are considered from the very beginning is required. This is especially true in the social cloud scenario, where there might be thousands of users and where trust and reputation requirements may change over time.

Another approach, especially used in social online applications and web sites, is to hard-code the reputation process in the application itself. However, as stated by Farmer and Glass [11], this might lead to poor, unmanageable solutions. Fixing bugs or mitigating abuse become impossible unless reputation remains an isolated module.

In order to overcome these problems, we propose a trust framework, the aim of which is to help developers to build different types of trust and reputation models as part of applications. We aim to provide developers with a systematic way to include almost any type of trust and reputation model in the services and applications they are implementing, making the difference between the notions of trust and reputation explicit. The suggested approach is a *callable framework*[1] that can be adapted by developers to their application-specific needs.

It is important to remark that our goal is not proposing a new trust or reputation model, but providing developers with mechanisms to implement existing (or new) models. Therefore, the framework is not concerned about the goodness of the

[1] A callable or called framework is composed of passive entities that can be called by other parts of the application, as opposed to a calling framework, where the framework takes over the main loop of the application and calls the pieces of code written by developers.

model inners. For example, the framework does not address that a malicious user attacks an implemented reputation model by providing false ratings. It should be the model itself that takes care of these problems, and thus, the final responsibility lies on the developer that implements the model.

The structure of the paper is as follows. Section 2 describes some work related to our contribution. A discussion on the concept of social cloud and a scenario example is presented in Section 3. Background information on trust and reputation is provided in Section 4 whereas a domain analysis on these areas is discussed in Section 5. The framework architecture is explained in Section 6, which is applied to the aforementioned social cloud scenario in Section 7. Finally, Section 8 outlines some directions for future research and concludes the paper.

## 2 Related Work

The idea of using trust and reputation in order to leverage cloud security can be found in several works. Usually, trust and reputation are used to help cloud stakeholders to make a decision about other stakeholders and services they have to interact with. For instance, Habib, Ries and Mühlhäuser [14] explores how these concepts can support consumers in selecting trustworthy cloud providers, whereas Liman and Boutaba [22] propose a reputation system in order to improve the process of selecting external services for integration in development projects. A similar goal is pursued by Abawajy [1], who suggests using a trust-based reputation system to determine service trustworthiness in Intercloud computing environments, and Xiao et al. [39] propose a reputation-based QoS (Quality of Service) provisioning model.

To the best of our knowledge, none of the proposals aims to build a unified framework that allows developers to implement existing or new trust models in the social cloud scenario. However, there are other frameworks that aim to provide, to a certain extent, development platforms to build trust models in other types of applications. Differences with our work include the approach used, the main focus, the covered domain and the comprehensiveness of the solution. Furthermore, none of them makes explicit the difference between the concepts of trust and reputation. Further details are given next.

Suryanarayana, Diallo and Taylor [33] propose the *4C* framework, where they describe a trust model as a composition of four sub-models, namely the content sub-model, the communication sub-model, the computation sub-model and the counteraction sub-model. For each sub-model, they identify the main building blocks that are present in existing reputation models. In the end, using an Java-based editor and following a personalized XML schema, they create a XML document where a trust model is described according to these building blocks. Then, they can use the PACE Support Generator to create software components that integrate within the PACE architectural style, which is further described in the work by Suryanarayana et al. [34]. In this latter work, the authors study the feasibility of an event-based architectural style in order to provide architects with guidelines on how to include trust models into decentralized applications.

Although their final goal is the same as ours, that is, making possible to include different trust models as part of applications, the authors follow a different approach. First, as they put the focus on decentralized applications, messages formats exchanged among peers and communication protocols play a crucial role. Therefore, their high-level interaction model is based on request-response among peers. Our approach is centralized, following a callable framework approach where the high-level interaction model is based on request-response to and from the framework. Second, in their approach, they follow a two-steps process where the trust model is explicitly described first and then it is integrated in the application following a particular architectural style. Instead, in our framework, the model is implicitly described by implementing certain components, and we do not assume any particular architectural style for the overall application.

Kiefhaber et al. [19] present the Trust-Enabling Middleware (TEM), which enhances the OC$\mu$ middleware, a message-based, service-oriented organic middleware. The TEM allows both the middleware and the applications built upon it to save experiences between interaction partners, which may be used to derive trust information. The TEM uses some middleware built-in functions to measure the reliability of nodes by considering packet losses, but it also allows defining customized transformers. Although rather complete, it requires applications to be built upon the OC$\mu$ middleware, which might be inappropriate and overkill for many applications.

Windley, Tew and Daley [37] presents the *Pythia* framework. Unlike the framework that we propose, which targets developers who are building applications from the ground-up, this framework is oriented towards the actual owners and users of already-implemented web sites, such as blog sites. The framework follows a plug-in architecture, where plug-ins provide the instantiation required for a concrete domain (i.e. a concrete web site). Therefore, a user who wants to use a *Pythia*-based system should first download the plug-in for that specific system. An important component of the framework is the rules engine, which is edited by users to decide how reputation is to be evaluated. Although this user-oriented approach is interesting, it presents a couple of shortcomings. First, as the system is targeted at non-expert users, the rules operators must be kept simple, which makes tough to define more sophisticated computation engines. Second, it is not clear whether the plug-in approach is powerful enough to use the framework in more general environments others than blogs or internet sites. Furthermore, plug-ins for every new environment have to be externally provided, as users cannot be expected to know how to create them.

Lee and Winslett present TrustBuilder 2 [20], where they propose an extensible framework that supports the adoption of different negotiation-based trust models. The purpose of this work is similar to ours, but the framework domain is different: whereas they focus on trust decision models, our targets are trust evaluation models, as we discuss later in Section 4.3.

SECURE platform [6] seeks a formal basis for reasoning about trust and risk. The trust model relies on a cost-benefit analysis represented by combined cost-PDFs (Probability Density Functions). The idea is, for every possible outcome of an action, to consider all possible costs and benefits the user might incur in. If the final combined cost-PDF shows that the benefits outweigh the other outcomes' costs, then the action proceeds. Otherwise, further interaction is arranged. The authors propose a software framework that allows building applications on top of this trust model. Although the trust model is interesting, we do not find the framework general enough to implement other types of trust or reputation models found in the literature.

Huynh [16] proposes the Personalized Trust Framework (PTF) that aims to replicate the trust evaluation process carried out by humans in a computational setting. Humans are capable of determining the context under which a trust evaluation is to be performed, but this is challenging for a computer. In order to overcome this challenge, the author proposes a rule-based system that uses semantic technologies (e.g. ontologies) to capture knowledge about different contexts and to apply the most suitable trust model according to these contexts. Even though it is an interesting contribution, its focus is on solving the context dependency of trust, lacking a framework-oriented approach as it does not provide guidelines nor application programming interfaces to create existing or new trust models.

Har Yew [15] presents a computational trust model and a middleware called SCOUT, made up of three services that implement the model: the evidence gathering service, the belief formation service and the emotional trust service. Regardless being a comprehensive trust model, which considers many facets of human trust, it is not designed as an extensible framework and it is not clear, if possible at all, how a developer could implement existing trust models.

The contributions by Pavlidis, Mouratidis and Islam [29] extends the Secure Tropos [26] modeling language in order to include trust concepts. Also, Pavlidis et al. [30] present a trust meta-model that allows developers to reason about trust and to model trust-based concepts. The authors of these contributions argue that it is required to take trust concepts into account while modeling security in order to achieve trustworthy systems. These contributions are relevant because the make explicit important trust concepts, but their approach is more oriented towards the requirements phase of the Software Development Life Cycle (SDLC), whereas our framework belongs in the design and implementation phases.

# 3 Exploiting Trust in the Cloud: towards the Social Cloud

Trust and reputation are key issues in order to enhance the cloud model within collaborative web environments, such as social networks. Chard et al. [8] propose an architecture for what they call a *social cloud*, which basically leverages traditional social networks in order to create a trusted cloud environment where users (i.e. friends) can seamlessly share storage resources. They implement a prototype by using the Application Programming Interface (API) provided by Facebook[2], which allows using the pre-established trust relationships between friends.

In our opinion, this approach, even though interesting and novel, has a shortcoming: it assumes that trust relationships are fully captured by friend relationships, which is often far from being a fact. One reason is that users tend to accept friendship from almost anyone who requests for it. Also, when some friend of a user starts acting annoyingly, the user tends to not delete this relationship as it can be considered a disproportionate or impolite reaction.

On the other hand, the authors do not consider reputation nor changes in trust relationships as a consequence of the state and quality of the provided services.

For the aforementioned reasons, we advocate that a more holistic approach, where both trust and reputation requirements are taken into account from the beginning, is required, and that a framework to assist developers in building this sort of environments from the ground-up would lead to more trusted systems. In Section 3.1, we explain the scenario used to validate our framework, whereas Section 3.2 discusses some trust and reputation requirements in this kind of scenario.

## 3.1 Scenario Description

The scenario that will be used as benchmark to validate our framework is the following. A developer needs to implement a social website for cloud providers. The site's purpose and workings are briefly explained next in order to shape a real scenario where trust and reputation considerations are desirable.

Cloud providers can register in the site. Once registered, they can publish web services on the site by posting a full description of the service, including the API calls (e.g. by using the Web Service Description Language (WSDL)). Cloud providers can also look up a web service according to their needs, and use the service in order to create a larger, composed web service. When one cloud provider consumes a service from another provider, the latter can charge the former according to the type or complexity of the service. Thus, the site acts as a *software as a service* market between cloud providers.

Eventually, each cloud provider will use its own infrastructure to provide the resulting services to their customers, although this is out of the scope of the scenario.

[2] http://developers.facebook.com

### 3.2 Trust and Reputation Requirements

The underlying framework must enforce trust and reputation requirements in order to try to prevent risks for the cloud providers and to foster trust in the site.

There are two basic reputable entities in this scenario: cloud providers and web services. Each of them might have a reputation that can be derived from the personal opinions and feedbacks of other providers in the site. For example, if a provider uses a service and notices that the service is not running appropriately, it could rate negatively the service, which in turn could negatively affect the service provider's reputation.

In addition to reputation, cloud providers can establish trust relationships among themselves. As it is discussed in Section 4.2, even though there are important relationships between trust and reputation, they are different concepts, and both of them should be implemented on the website. Figure 1 depicts the scenario and its trust and reputation requirements.

The way trust and reputation is computed depends on the models implemented, and the developer of the website should be provided with mechanisms to decide which model to use at design-time. These mechanisms are provided by the trust and reputation framework described in Section 6.

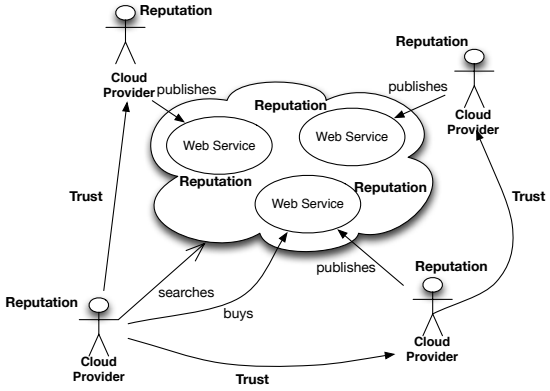


Figure 1: Scenario

## 4 Trust Foundations

The aim of this section is to set up the foundational concepts that surround the notions of trust and reputation. In Section 4.1, we analyse existing trust definitions and highlight some of the most important trust-related concepts. The relationship between trust and reputation is discussed in Section 4.2, whereas Section 4.3 elaborates on the concept of trust model and provides a classification.

### 4.1 Trust Definitions and Concepts

In this section, we review the most often cited trust definitions and also some recent ones. From these definitions (among others), we can extract some relevant concepts that often surround trust and reputation.

There has been a huge amount of different definitions of trust along the years. This is due to mainly two factors: first, trust is very context dependent, and each context has its own particularities. Second, trust spans across many disciplines, including psychology, economics and law, and have different connotations in each of them. Finally, there are many factors that influence on trust, and it is not straightforward to identify them all.

The singularity about trust is that everyone intuitively understands its underlying implications, but an agreed definition has not been proposed yet due to the difficulties in putting them down in words. The vagueness of this term is well represented by the statement made by Miller [25]: "trust is less confident than know, but also more confident than hope". Some definitions of trust in the computer science context are given next.

Gambetta [12] defines trust as "a particular level of the subjective probability with which an agent will perform a particular action [...] in a context in which it affects our own action". McKnight and Chervany [24] explain that trust is "the extent to which one party is willing to depend on the other party in a given situation with a feeling of relative security, even though negative consequences are possible". For Olmedilla et al. [28], "trust of a party A to a party B for a service X is the measurable belief of A in that B behaves dependably for a specified period within a specified context (in relation to service X)". Ruohomaa and Kutvonen [32] state that trust is "the extent to which one party is willing to participate in a given action with a given partner, considering the risks and incentives involved". Finally, Har Yew [15] defines trust as "a particular level of subjective assessment of whether a trustee will exhibit characteristics consistent with the role of the trustee, both before

the trustor can monitor such characteristics (or independently of the trustor's capacity ever to be able to monitor it) and in a context in which it affects the trustor's own behavior".

In an earlier work [27], we built the concept cloud depicted in Figure 2 in order to highlight the most important concepts present in the previous (and other) definitions. We can draw some conclusions from this cloud of concepts. First, trust would not make sense unless there were entities that trusted each other. Also, trust heavily depends on the context, is subjective and can be assessed. Trust often involves belief or expectation with regards to another entity's action. Trust is strongly related to security, although it goes beyond as security is just one dimension of trust. Finally, trust is beneficial in the presence of uncertainty and risk during the interaction of entities, which must be willing to collaborate and to depend on each other.


Figure 2: Concepts Cloud for Trust Definitions

We propose the following definition of trust: *trust is the personal, unique and temporal expectation that a trustor places on a trustee regarding the outcome of an interaction between them.* This interaction usually comes in terms of a task that the trustee must perform and that can (negatively) influence the trustor. The expectation is personal and unique because it is subjective, and is temporal because it may change over time.

An important remark at this point is that the framework that we propose aims to build trust-aware systems. By trust-aware, we mean systems that implement trust and reputation models and that use trust and reputation information in order to make decisions. However, we do not aim to build trustworthy systems, as these systems require a holistic approach to security from the very beginning of the SDLC, including a design for assurance methodology [5].

## 4.2 Trust and Reputation

We advocate that there exists a bidirectional relationship between trust and reputation, in the sense that each one may build upon the other one.

According to the Concise Oxford dictionary, reputation is "what is generally said or believed about a person or the character or standing of a thing". This definition, and more concretely, the word *generally*, implies that reputation is formed by an accumulation of opinions. This accumulation of information makes reputation a more objective concept than trust.

A good approximation to the relationship between trust and reputation was suggested by Jøsang [18], who made the following two statements: 'I trust you because of your good reputation' and 'I trust you despite your bad reputation'. In this sense, reputation can be considered as a building block or indicator to determine trust but, as stated by the second statement, reputation has not the final say. One could either trust someone with low reputation or could distrust someone with high reputation, because there are other factors that may have a bigger influence on trust determination, such as the trustor's (the entity which places trust) disposition to believe in the trustee (the entity onto which trust is placed), the trustor's feelings, or above all, the trustor's personal experiences with the trustee.

However, it is important to note that trust has an influence on reputation to a similar extent as reputation influences on trust. In the literature, it is often said that reputation is a more objective concept than trust, but this higher objectivity comes from the fact that a lot of subjective impressions can lead to a more objective one. As an example, think of Alice having had a bad experience with a service provider. This bad experience has led her to distrust this service provider, because she is basing her trust judgement on her personal experience. However, the service provider could have had just a 'bad day' with her, whereas it provides an excellent service to other thousands of people who openly claim their delight with the provider. In that case, the accumulation of positive personal experiences are making more objective the fact that the service provider is actually good, in spite of Alice's bad experience. That is, the service provider has a good reputation because lots of people trust it, regardless Alice does not. In this case, (aggregated) trust is having an influence on reputation too.

## 4.3 Trust Model Definition and Classification

The concept and implications of trust are embodied in the so-called trust models, which define the rules to process trust in an automatic or semi-automatic way in a computational setting. There are different types of trust models, each one considering trust in different ways and for different purposes. The origins of trust management date back to the nineties, when Marsh

[23] proposed the first comprehensive computational model of trust based on social and psychological factors. Two years later, Blaze [3] coined the term *trust management* as a way to simplify the two-step authentication and authorization process into a single trust decision.

These two seminal contributions reveal the two main branches or categories of trust models that have been followed until today, and which we classified in our previous work [27]. On the one hand, and following Marsh's approach, we find evaluation models, where factors that have an influence on trust are identified, quantified and then aggregated into a final trust score. Uncertainty and evaluation play an important role in these models, as one entity is never completely sure whether it should trust another entity, and a quantification process is required to evaluate the extent to which one entity trusts another one.

On the other hand and following Blaze's approach, we find decision models, which are tightly related to the authorization problem. An entity holds credentials and a policy verify whether these credentials are enough to grant access to certain resources. Here, trust evaluation is not so important in the sense that there are no degrees of trust (and as a consequence, there is not such a big range of uncertainty), and the outcome of the process is often a binary answer: access granted or access denied.

Both categories, evaluation and decision models evolved leading to ever-complex models. On the one hand, in the case of decision models, new concerns arose regarding the privacy of the entities that publicly reveal the credentials and policies. As a consequence, negotiation models appeared [38], in which a negotiation strategy is used in order to disclose only the required credentials and policies in a controlled manner. On the other hand, one of the branches of evaluation models with higher impact have been reputation models, in which a reputation score about a given target is derived from other entities' opinions about it.

# 5 Domain Analysis

Prior to building a framework, it is of paramount importance to gather as much knowledge as possible about the framework domain [10]. In our case, this domain is trust and reputation, and concretely, evaluation models.

The purpose of this section is to provide a domain analysis, which is partially based on the one performed in our earlier work [27]. First, in Section 5.1, general concepts and their relationships are identified, whereas specific concepts for evaluation models and web reputation models are discussed in Section 5.2 and Section 5.3, respectively. These analysis help during the elicitation of the high-level requirements of the framework, which are discussed in Section 5.4.

## 5.1 General Trust Models Concepts

Figure 3 provides a conceptual map for common concepts and relationships that are present in both evaluation and decision trust models. Trust models are very heterogeneous because they are defined for different contexts, under different assumptions and for several application domains, such as for example, P2P or multi-agent systems. This section summarizes important concepts that can be found in most trust models, as well as their relationships.

In any trust setting, there are entities and each entity plays a role, or even several ones. At least, each entity is a trustor and/or a trustee. An entity can play also the role of a witness that gives its opinion about another entity. Depending on the application domain, further specialization of these roles is possible, such as service requesters, service providers and trusted third parties that issue credentials or gather feedbacks. Once we have a trustor trusting a trustee, we can say that a trust relationship has been established.

Every trust relationship has a purpose. According to Jøsang et al. [18], a trust purpose is an instantiation of any of the following trust classes identified by Grandison and Sloman [13]: access trust, provision trust, identity trust, and infrastructure trust (considering delegation a sub-class of provision trust).

Access trust is used to determine whether a service requester can be trusted, whereas provision trust determines trust in the service provider. Identity trust refers to using the identity or attributes of an entity to determine whether it can be trusted. Infrastructure trust is the trust that an entity places on the hardware and, in general, the infrastructure that supports the entity's activities.

Trust can also be conceived as a strong belief about a given property of the trustee. From a theoretical perspective, there would be no purpose under this trust conception. Yet we are interested in trust models from a more pragmatic perspective. Thus, trust in a given property would eventually assist in making a decision for some purpose. For instance, if an entity believes that another entity is competent to encrypt files, it would select the latter among other candidates less qualified (according to the entity's belief). In this example, the purpose will therefore be the provision of an encryption service (i.e. provision trust).

Trust is context-dependent, one of the most important properties of trust, since it influences all the other concepts, such as the purpose, the type of entities and the role that they can play. Note also that, as mentioned earlier, some efforts have been made to build context-independent trust models by using semantic technologies [16].

There are other factors, in addition to the context, that influence trust [40]. For instance, the trustee's/trustor's subjective properties such as honesty, confidence, feelings, willingness or belief, and the trustor's/trustee's objective properties, which may include observed behaviour, security, reliability, ability, a given set of standards or reputation.

A trust model also makes assumptions, such as 'entities will provide only fair ratings', 'initial trust values are assumed to exist' or 'reputation scores are low for newcomers', and might follow different modeling methods, including mathematical, linguistic and graphical.

## 5.2 Evaluation Models Concepts

Figure 4 provides a conceptual map for concepts that are specific to evaluation models. We distinguish three types of evaluation models: reputation models, propagation models and behaviour models. Reputation models are those where a score is derived from the aggregation of opinions of other entities in the community. This score is made public and can be looked up by any entity before making a trust decision. Regarding propagation models, also known as flow models, they assume that some trust relationships already exist and aim to use them in order to create new relationships, usually by exploiting trust transitivity. Finally, behaviour models are those where the trustor evaluates its trust in the trustee by personally observing its behaviour and interacting with it, or by building a mental state. Even though there exists pure reputation models [31], propagation models [2] and behaviour models [7, 23], this classification should not be considered disjoint, as we might find features from each of them in several models. For example, Advocato [21] is a reputation model that allows users of the community to provide a ranking for other users. However, it is also a propagation model, since it allows computing a reputation flow through a network where members are nodes and edges are referrals between nodes.

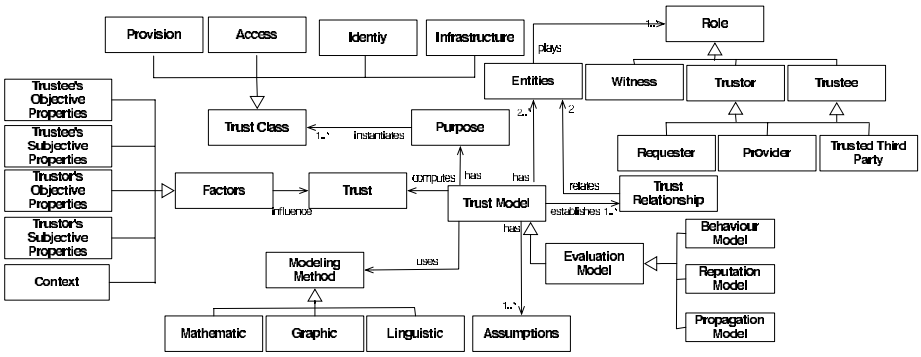


Figure 3: Common Concepts for Trust Models

In evaluation models, each trust relationship is tagged with a trust value that indicates to what extent the trustor trusts the trustee. This value has a dimension, which indicates whether it is a single value or a tuple of values. Also, a trust value has semantics, which can be represented by two dimensions: objectivity and scope. The former refers to whether the measure comes from an entity's subjective judgement or from assessing the trusted party against some formal criteria. The latter specifies whether the measure is done against one factor or against an average of factors.

Evaluation models often follow a lifecycle with two stages. First, a bootstrapping phase might be required to assign initial trust values to the entities of the system and to every newcomer. Trust propensity is a concept related to the bootstrapping phase and it refers to the propensity of the model towards high or low trust values in the beginning. Second, a trust assessment process is performed in order to assign trust values to entities according to certain variables. This process usually involves some monitoring in order to feed these variables with accurate data.

The concept of trust assessment, and all the concepts related to it, is likely the most important one in evaluation models, as it may become the signature of the models, what it makes them different from others. In order to carry out this process, trust metrics are used. Trust metrics use variables, such as risk, utility, past experience or observed behaviour, and combine them in order to yield a final score for the measured attribute(s). General attributes that are measured are trust and reputation. Yet depending on the application domain, more specialized attributes can be measured, such as 'reliability of the seller' in an e-Commerce scenario. Trust metrics use computation engines, which may be simple summations, average, continuous, and discrete engines, or more sophisticated ones like belief, Bayesian, fuzzy or flow engines. Jøsang [18] provides a good summary of their features.

Related to the trust assessment process, the sources of information that feed the metric might come from direct experience (either direct interaction or direct observation), sociological factors, psychological state and reputation. Sociological factors capture the fact that trust can be influenced by the roles played by an entity or the group the entity belongs to. Psychological or mental state includes trustor's subjective factors that can influence trust determination, such as the trustor's prejudice or its belief in certain trustee's properties.

In the case of reputation models, the main source of information is the reputation score, which represents an aggregate opinion from other entities. Reputation models can be centralized, where there is an entity in charge of collecting and distributing reputation information; or distributed, where there is no such an entity and each one has to maintain a record of trust values for other entities, and send this information to the rest of entities.

Regardless of which information source is used to compute the trust value, the model might consider how certain or

reliable this information is (e.g. credibility of witnesses), and might also consider the concept of time (e.g. how fresh the information is).

Most evaluation models follow a game-theoretic approach, where the trustor determines trust in a trustee by examining the outcomes after each interaction. Trust is usually defined in these models as a subjective expectation about the outcome of an interaction with another entity. Fewer evaluation models follow a cognitive approach, where trust is primarily determined by the mental state of the entities, which usually comprises a set of beliefs on other entities.

As mentioned earlier, propagation models often assume that several trust relationships have already been established and quantified, and exploit the trust transitivity in order to disseminate trust information [3]. New trust values are often computed by means of operators, and in several models, we find two of them: a concatenator and an aggregator. The former is used to compute trust along a trust path or chain, whereas the latter aggregates the trust values computed for each path into a final trust value. For example, Agudo, Fernadez-Gago and Lopez [2] use a sequential and a parallel operator in order to compute trust along a path. Subjective logic [17] uses a discounting operator to compute opinions along different trust paths, and a consensus operator to combine them into a final opinion.

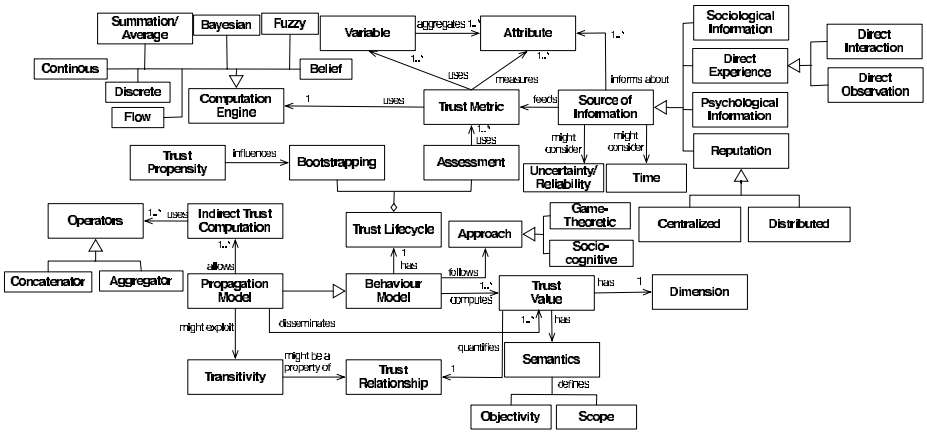


Figure 4: Concepts for Evaluation Models

## 5.3 A Closer Look at Web Reputation Models

Given that our intent is to support trust and reputation in the social cloud, which actually is comprised by a community of users, it is worth deepening into some concepts that are present in web reputation models. In such models, reputation can be divided into two terms [11]:

- Reputation: information used to make a value judgement about an object of a person.
- Karma: the reputation(s) for a user.

Web Reputation models bring a lot of value to online community sites. Users are presented with higher quality information as this is filtered by the reputation of its creator. Also, there is a cost reduction, as automatic actions can be performed in response to high or low reputation. For instance, if a user comment or review is rated with a very low reputation, it might be automatically deleted as it could be spam. This saves the cost of having people taking care of these issues.

### 5.3.1 Reputation Statement

The core concept behind reputation is a *reputation statement*, which can be defined as a claim stated by a source regarding a target. As an example, if Alice says: 'The film Titanic has a good photography', the source is *Alice*, the target is the *film Titanic*, and the claim is *has a good photography*. Of course claims can also be made regarding another user; for instance, Alice may say: 'Bob is a great fantasy writer'. In this case, the source is Alice, the target is another user, Bob, and the claim is *is a great fantasy writer*.

A source can be any entity with a unique identifier that makes a reputation claim. There are three types of sources:

- Non-human source: includes anti-spam filters, input from other reputation models, log crawlers or recommendation engines.
- Human source: users are often the source of reputation statements.

[3] Note however that transitivity is not, in general, considered as a property that holds for trust [9].

- Aggregate source: it represents a set of sources within a system, where the set is meaningful and does not require to know the identity of each individual source. A final reputation store for an entity is computed by using the claims of an aggregate source.

A claim is the assessment made by the source to the target. A claim may be simple or compound. A simple claim represents a value for a single property of the target, whereas a compound claim defines a value for more than one property of the target. For instance, a reputation model may allow users to give a global evaluation for a film (e.g. number of stars), or it may allow them to evaluate several properties of the films separately (e.g. actors, photography, soundtrack, etc). A simple claim and each dimension of a compound claim are of a type. A *type* is a tuple with the following elements:

- Representation format: claims can be quantitative or qualitative. Quantitative claims include accumulators, votes, segmented enumerations (such as stars), averages and rankings. Qualitative claims can be blocks of text (e.g. comments), videos, or author attributes.
- Scale: it defines the minimum and the maximum bounds for the value represented by the claim.
- Display format: it specifies the way that the value represented by the claim is displayed to the users.

Regarding the last element of a reputation statement, that is, the target, it is worth pointing out that reputation statements can be themselves the target of a claim. For instance, a user Alice might claim the review performed by Bob regarding the film Titanic was useful. In this case, the target is *Bob's review about Titanic*, that is, a reputation statement where the source Bob expressed its opinion (claim) about the target Titanic.

### 5.3.2 Context, Events and Computation

Reputation always takes place in a context. Certainly, a reputation statement made in one context may influence another context. If a flight company has a good reputation, it is likely that the company's aircrafts or customer service have also a good reputation. However, in general, reputation only makes sense in the particular context where it is being evaluated. For example, having a good reputation as a film reviewer does not influence on the reputation as a dentist.

Claims are introduced in a reputation model by triggering events. These events depend on the application. Some clear example of events are: the user rates a film, the user clicks on *I like* button or the user clicks on *It was useful* button. However, there are other events that, even though more subtle, may be more informative: the fact that a user has read a review to the end may show that the review is interesting. If the user has posted a comment on a review, it may indicate that the topic covered by the review is appealing. If a user has seen another user's profile, it may show that the former likes how the latter performs.

These events trigger a set of processes in order to compute a reputation score. The way this reputation score is computed determines the type of model, as well as the factors taken into account during the computation.

## 5.4 High-Level Requirements

This section discusses the coarse-grained requirements that a trust framework should provide developers with. It also builds upon the concepts and relationships explained in the previous sections.

At the highest level, the framework must support the implementation of three types of evaluation models, namely reputation models, behaviour models and propagation models. This requirement boils down to the following requirements:

- Entity management: the framework must be able to give a unique identifier to each entity in the system and to retrieve trust and/or reputation information from an entity.
- Trust relationship management: trust relationships might change over time. New trust relationships might be created (e.g. by propagation models), other relationships might be deleted, and trust values attached to these relationships may change. Trust relationships can be affected by reputation, but also by subjective properties of the trustor and/or trustee. The framework should support the definition and management of these other factors too.
- Computation engines definition: computation engines are in charge of computing a trust or reputation score, depending on the model. Although the framework can provide some default built-in metrics implementations, it is important to let developers to define their own trust metrics, as they are the core concept in evaluation models.
- Events definition: events that occur in the application trigger the communication with the framework. It is required to configure the framework to respond to these events accordingly.
- Claim management: the type and value of a claim may determine a trust or reputation score. It should be possible to configure claims in order to support application-specific needs.
- Trust dissemination: trust information can be propagated by means of operators along trust chains. The developer should be empowered to define his own operators, or to use some built-in ones.

- Time and Uncertainty: these factors may play an important role when computing a trust or reputation score. The framework should provide the developer with mechanisms to include them as part of the computation process.
- Trust and Reputation separation: the framework should allow developers to consider trust and reputation as different concepts. However, given their strong relationship, it should be possible to take each other into consideration when computing a trust/reputation score.

The next section describes the architecture that supports these requirements. Given that the framework is still under development, and also for simplicity sake, we focus on reputation and behaviour models, and lay aside propagation models, although a complete trust framework should address them as well.

# 6 Trust and Reputation Framework

The social cloud, where thousands of users might interact and make many important decisions, can greatly benefit from having an executing platform that takes trust and reputation into account as separate and interrelated concepts.

At a high-level of abstraction, the framework is a middle-tier server that mediates between the client application and the database tiers. The application requests for trust and reputation information updates and retrieves trust and reputation information from the databases. In the simplest case, where the developer may use the pre-defined functionalities of the framework, just two interfaces are needed: one to signal that a new event has occurred, and another one to apply for trust information, as depicted in Figure 5. However, as a framework, it is also required to provide some mechanisms to adapt the framework functionality to the developers' needs at design-time.

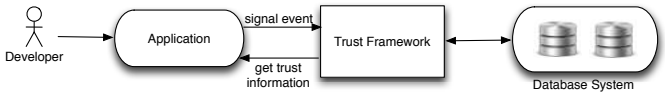


Figure 5: The Framework in Context

The kind of API that the framework exposes depends on the implementation details, and could range from Remote Procedure Calls (RPC) to a REST- or SOAP-based API. Even though the framework is still under an early stage of development, some implementation guidelines are given in Section 6.2.

One of the goals of the framework design is to allow making explicit that trust and reputation are two different notions, even though they are strongly related. The architecture is presented in Section 6.1, whereas some implementation discussions are provided in Section 6.2.

## 6.1 Architecture: Structural and Behavioural Views

The most architecturally significant components of the framework are shown in Figure 6. The framework API provides three interfaces: *write event*, to explicitly signal that an event has occurred, *get trust information*, to retrieve trust-related information, such as the reputation of a given entity or existing trust relationships. Additionally, the API may provide mechanisms to configure the trust database attributes and tables.

There are three components that represent queues: the *Event Queue*, which stores events, the *Reputation Statement Queue*, which stores reputation statements and the *Trust Statement Queue*, which stores trust statements.

The *Event Handler* component is in charge of reading events from the *Event Queue*, creating reputation statements according to the event type and writing these events into the *Reputation Statement Queue*. The *Engine Dispatcher* component reads reputation statements and creates engines that perform computation on these statements, producing trust statements. The *Data Manager* component prevents the rest of components from needing to know the internal details of the database schema and technology used. It basically transforms queries and write actions into technology-dependent statements (e.g. SQL statements). The *Trust Data Store* component represents the RDBMS (Relational Database Management System) that provides persistence to the framework.

Figure 7 depicts a coarse-grained sequence diagram that describes the steps triggered by an event signalled by the client application.

Note that in Figure 7 there are two architectural elements that are not shown in the component diagram, but have been added here for a clearer understanding. The *Reputation Statement* entity represents a reputation statement, that is, a tuple $< source, claim, target >$. The other entity is *Engine Factory*, which is used by the *Engine Dispatcher* in order to create an appropriate engine according to the reputation statement type.

Another issue to note from Figure 7 is the area of the bottom part, where the steps carried out if the target of a reputation statement is, in turn, a reputation statement, are described. In this case, the original reputation statement is retrieved from the database through the data manager component and a new reputation statement will be written in the *Reputation Statement Queue* for later processing. In a real scenario, this typically happens when a user states that the

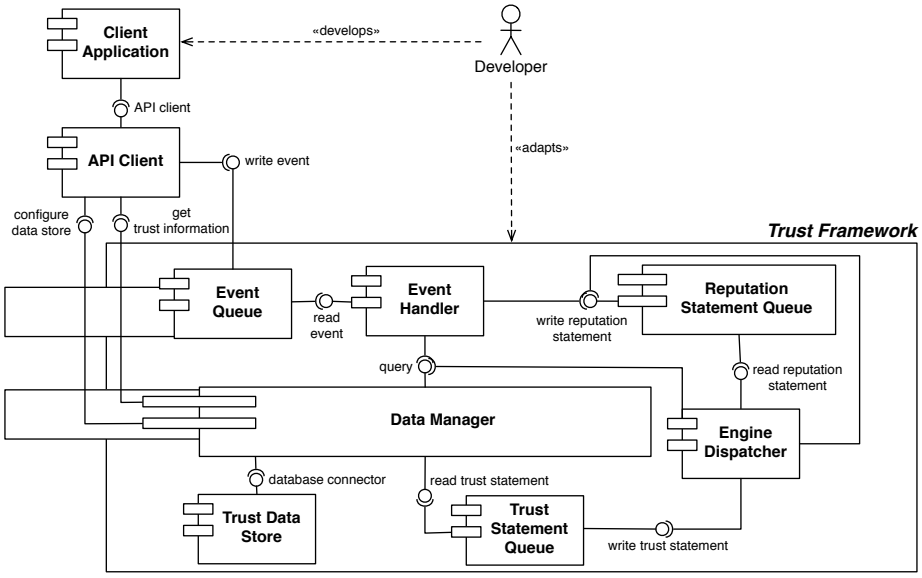


Figure 6: Framework Architecture: Component Diagram

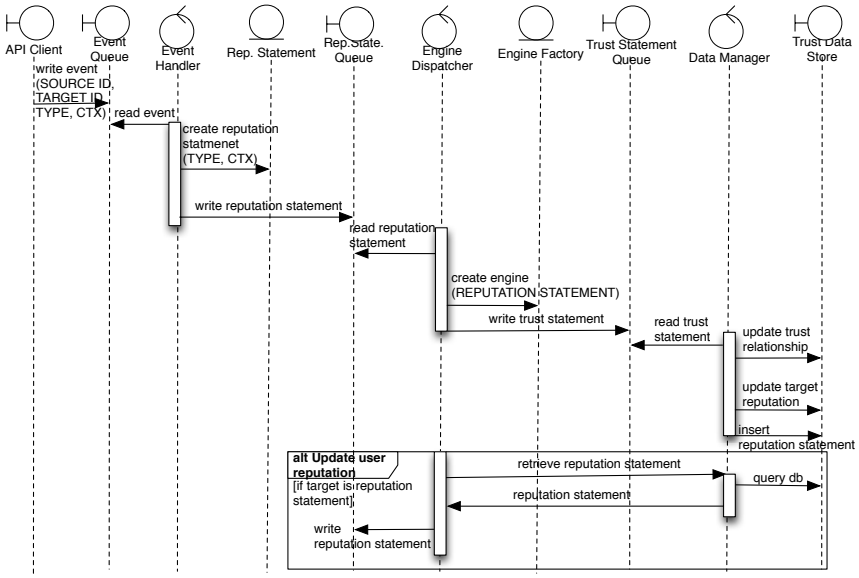


Figure 7: Framework Architecture: Sequence Diagram

information provided by another user (that is, a reputation statement), has been helpful. In this case, the reputation of the reputation statement would be updated, but the reputation of the user who provided the information could be updated as well. This behaviour could be configured by the developer, for instance, by an optional parameter passed as an argument.

### 6.1.1 Components and Modules

A component decomposition into modules for the *Engine Dispatcher*, the *Event Handler* and the *Data Manager* component is shown in Figure 8, Figure 9 and Figure 10 respectively.

The *Event Handler* uses an *Event Reader* in order to retrieve events from an event queue, and a *Claim Factory* in order to create a *Claim*. The factory inspects a *Claim Manager* to determine which type of claim to make. The manager creates a *Reputation Statement*, which uses a *Claim* and a *Reputation Statement Writer* in order to forward a reputation statement to a queue.

In the case of the *Engine Dispatcher*, the *Engine Manager* module uses the *Reputation Statement Reader* module in order to retrieve a reputation statement. It also uses an *Engine Factory*, which accesses a *Computation Manager* in order to figure out how to make two engines: a *Reputation Engine* and a *Trust Engine*. The manager creates a *Trust Statement* and uses a *Trust Statement Writer* that knows to which queue to forward a trust statement. Finally, the manager allows

querying information and writing reputation statements into queues (in case the target of the reputation statement is itself a reputation statement).

As for the *Data Manager*, it has a *API Translator* module that uses the *Trust Statement Reader* in order to retrieve a trust statement and translates the native API call (e.g. Java) into a technology-dependent statement (e.g. SQL) through a database connector. The API translator may use of data structures that will be sent back to other components or the client application.

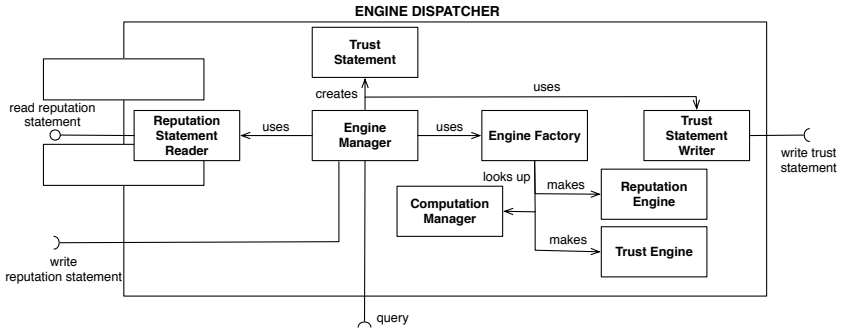


Figure 8: Engine Dispatcher Component

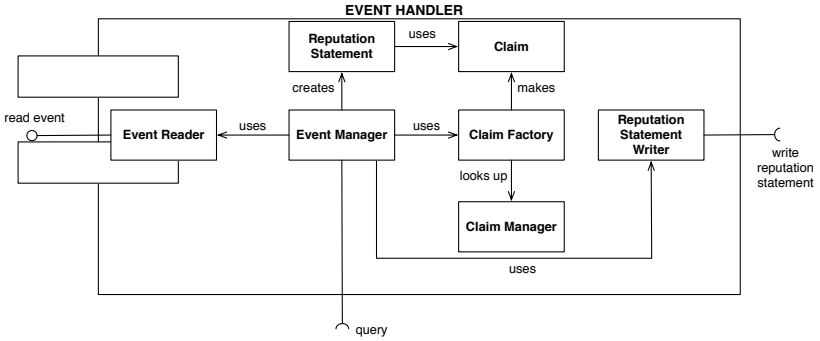


Figure 9: Event Handler Component

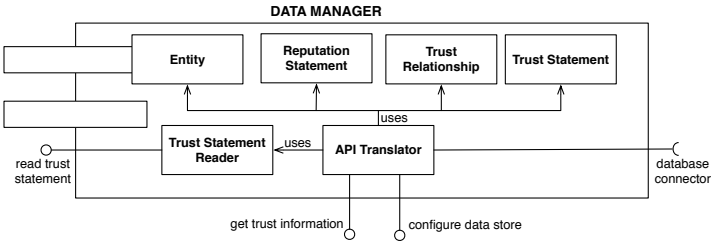


Figure 10: Data Manager Component

Next we describe some data structures that are of special relevance.

### 6.1.2 Data Structures

Even though many more modules and structures might arise in a further detailed design, there are four data structures that are specially relevant as they encapsulate crucial information that flow between components. If we consider them from an Object-Oriented (OO) perspective, they could be refined as classes, which are shown in Figure 11. For each of these structures, only the most important attributes and applicable functions (i.e. methods in OO design) are shown.

The *Event* structure represents an event by means of a name, a context, a source of the event, and a target of the event. Several event types can be pre-defined, but new events can be created.

A *Claim* represents the assessment made by an entity. The type of event that is triggered determines the type of claim that is made. A claim has a scale (minimum and/or maximum boundaries) and a value, which might be numeric or qualitative.

A claim can be normalized (resp. denormalized) from its range scale (resp. interval [0..1]) to the interval [0..1] (resp. its range scale).

A *Reputation Statement*, as stated previously, contains a source, a claim and a target. Source and target are entities. In order to allow developers to take time into account, a reputation statement holds a time stamp, which indicates when the reputation statement was made. Also, as a reputation statement is made in a context, the latter is considered as part of the statement.

A *Trust Statement* (which is a new notion that has not been mentioned previously) contains a reputation statement, a reputation value and a trust value. This structure allows the framework to convey trust and reputation information separately, fostering the idea that trust and reputation are two different concepts. A trust statement is the structure that is passed onto the data manager in order to update the different database tables.

A *Trust Relationship* represents the trust value that a trustor (the source *Entity*) places on a trustee (the target *Entity*). This relationship is established in a given context at a given time.

An *Entity* represents any object that can be evaluated. It has a unique name, a type (Human, Non-Human and Reputation Statement), a reputation score and the context under which the reputation score is assigned. A *HumanEntity* is an entity which, additionally to the previous fields, also holds a list of trust relationships, a rating bias and a set of beliefs. Each belief has a name (e.g. capability, honesty), a value, and a target (i.e. another human entity). These last fields are further detailed next.

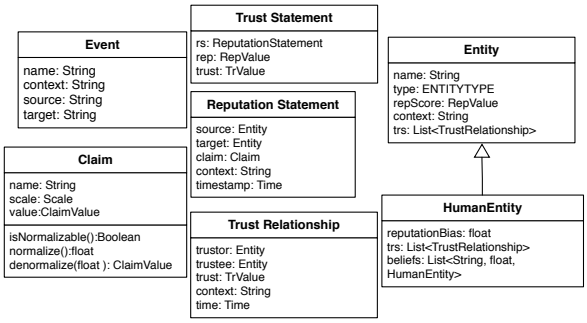


Figure 11: Important Data Structures

### 6.1.3 Enabling Trust

When a user entity rates another user entity, their trust relationship may change. As explained in Section 4.1, there are other factors that influence trust beyond reputation, such as the trustor's subjective properties. The framework provides developers with support to include the following properties:

- Rating bias: this property indicates the usual disposition of the user to high or low rates. If a user has always rated other users with the maximum value in the past, the fact that now the user rates a user with a high value does not give much information. However, it would give a lot of information if the user rated another user with a low value. This can be implemented as the standard deviation of all the claims made by the user over a period of time.
- Beliefs: they indicate how much a user believes in the capability, honesty, etc of a target user. This information could come from several events: number of visits to the target's profile or ratings of the target's claims (e.g. the user thinks that the review of the target is useful).

The following section discusses some implementation issues for the architectural elements and design concepts that have been described up to now.

## 6.2 Implementation Guidelines

Once the framework is configured according to the developer needs, it can be deployed as a JavaEE application that constitutes a runtime platform onto which to develop trust-aware applications. The decision of implementing the framework in Java is two-fold: since Java is a quite popular development language, it may be easier for developers to familiarize with the framework and with the mechanisms to adapt it to their needs. Furthermore, we achieve portability, as the framework can be executed on any platform and operating system.

As an example, Figure 12 shows the steps that a client application would perform in order to query trust-related information from the trust database. The client application would need to look up an instance of the trust server from the JNDI (Java Naming and Directory Interface) in order to invoke the query API call, implemented by EJBs (Enterprise Java Beans) that connect to a SQL server by means of a JDBC (Java Database Connectivity) connector.

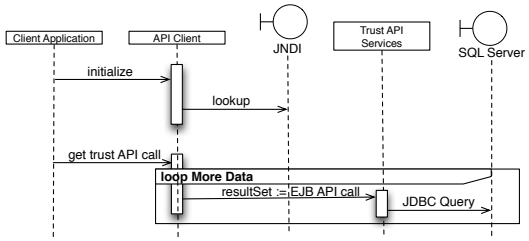


Figure 12: Query API Call Sequence Diagram

In the next sections, we elaborate on some implementation ideas we are working on for different architectural elements and concepts.

### 6.2.1 Context

As we mentioned in Section 5, the context is very important in the trust and reputation domains. Every trust or reputation score makes sense in a single context and cannot (usually) be transferred directly to another context. In order to take into account the context, whenever an event is triggered, the developer introduces a string that represents this context. It will then be stored together with all the rest of trust or reputation information, as explained in the next section.

### 6.2.2 Database Tables

A RDBMS (Relational Database Management System) such as SQL can be the implementation choice in order to persistently stores all the trust-related information. The tables design is of paramount important for the efficiency and correct behaviour of the framework, as they may support more or less easily the implementation of the concepts discussed above. As an example, we propose Table 1, Table 2, Table 3 and Table 4. For each one, we explain the main attributes they should hold and their meaning.

Table 1: Entity Table

| Attribute | Description |
|---|---|
| ID | Unique identifier |
| Type | Human, Non-Human or Reputation Statement |
| Reputation | Reputation score |
| Context | Context where this information applies |
| Claim | Type of the claim that corresponds to this evaluation |
| Time | Indicates when this reputation score was first created |
| LastTime | Indicates the last time this entity's reputation was changed |
| Number of evaluations | Number of evaluations made on this entity |

Table 2: Trust Relationship Table

| Attribute | Description |
|---|---|
| ID | Unique trust relationship identifier |
| ID Trustor | The unique identifier of the entity placing trust |
| ID Trustee | Unique identifier of the entity onto which trust is placed |
| Trust value | Trust value placed by the trustor on the trustee |
| Context | Context where this trust relationship holds |
| Number of updates | Times that the value of this relationship has changed |
| Time | Indicates when this trust relationship was first established |
| Last Time | Indicates when this trust relationship was last updated |

Primary keys could be a made up of the ID and the Context, since the same entity could hold different trust or reputation values for different contexts. Unique identities could be the foreign keys used in order to relate tables among each other.

The *Data Manager* component must provide the interface required to set and obtain most of this information. For this purpose, it uses a JDBC (Java Database Connectivity) connector in order to translate from Java method calls to SELECT,

Table 3: Reputation Statement Table

| Attribute | Description |
|---|---|
| ID | Unique identifier |
| Source | Source entity's ID |
| Target | Target entity's ID |
| Claim | Name of the claim |
| Claim value | The value of the claim |
| Context | Context where the statement is applicable |
| Time | Time when the claim was made |

Table 4: Beliefs Table

| Attribute | Description |
|---|---|
| ID | The unique identifier of the belief (e.g. capability, honesty, ...) |
| Source | Source entity's unique ID |
| Target | Target entity's unique ID |
| Value | Belief value |
| Context | Context where this belief is applicable |

INSERT and UPDATE SQL statements. Given that this can be complex, a suitable, more maintainable design approach would be to create different objects to manage each table. The *API translator* would be split into a mediator object that delegates the queries to these specialized objects. Thus, one object would not need to know how to interact with all the database tables.

Note that these tables support taking the trustor's subjective properties into account. In order to compute the *rating bias*, the *Data Manager* would first retrieve all the claims made by the user, normalize them (in order to consider different types of claims with different scales), and then compute the average and finally the standard deviation. This can be achieved by looking up the *Reputation Statament* table.

Another implementation choice would be using an OODBMS (Object-Oriented Database Management System), which provides higher flexibility and avoids the tedious mapping between two representation models (i.e. from objects to relational tables), as they allow storing objects directly. This simplifies greatly the implementation of the *Data Manager*, which would basically become a direct mediator between an API call and the database system, without requiring the translation process. However, RDBMS are often more efficient, above all when considering simple objects and relationships.

#### 6.2.3 Messaging Infrastructure

The main components of the architecture communicate with each other using asynchronous, optimistic queue-based messaging system, implemented onto the JMS (Java Message Service). This decision is made because of the performance requirements in social cloud contexts, where thousands of users may concurrently perform operations on the trust server.

Whenever a new event is triggered by the developer, the client application does not wait for a response from the server, but it continues doing other tasks. The application knows that the trust server will eventually update the trust relationships and reputation scores, but does not matter when.

The same happens with the *Event Handler*, *Engine Dispatcher* and *Data Manager* components. They are listening to their specific queues. As soon as a new piece of information arrives, they take it out from the queue, process it, and place it in another queue for further processing. This way, the server can adjust, on-demand, the number of instances of the same component that is listening to a queue, providing higher performance and scalability. This is especially true for the *Data Manager*, which can receive queries and writes requests from different components: the *Trust Statement Queue*, the *Engine Dispatcher*, the *Event Handler*, and even from the client application. Therefore, many instances could be concurrently listening to queries from these different sources.

#### 6.2.4 Engines

When an *Engine Dispatcher* instance reads a reputation statement, it uses an *Engine Factory* to create the appropriate *Engine* to deal with such statement. The *Engine Factory* creates an engine by inspecting computation rules, which define which type of engine to create under which circumstances. This can be implemented as a XML file that the factory reads. This file includes a set of conditions and an effect, which states the engine type to build. A very simple example is shown next:

```
<CompRule RuleId=‘‘CompRule CounterUp’’ Engine=‘‘CounterUp’’>
        <Context>Film Review</Context>
        <Claim>Positive Vote</Claim>
        <Source>
                <SourceType>Human</SourceType>
        </Source>
        <Target>
                <TargetType>Non-Human</TargetType>
        </Target>
</CompRule>
```

This file specifies that if the context of the reputation statement is ‘Film Review’, the claim is ‘Positive Vote’, the type of the source of the reputation statement is ‘Human’, and the type of the target of the reputation statement is ‘Non-Human’, then the engine to apply is ‘CounterUp’. The *Engine Factory* would read the file (through another XML reader object), compare the conditions against the reputation statement fields, and create an instance of this type of engine.

The following code snippet shows how the class *EngineManager* would work:

```
public class EngineManager {

        //...more stuff

        //This method is called after a Reputation Statement Reader instance
        //signals that a new reputation statement has arrived.
        public void onNewReputationStatement(ReputationStatement rs) {
                Engine[] e = EngineFactory.getEngine(rs);
                //e[0] holds an instance of the reputation engine
                //e[1] holds an instance of the trust engine
                if (e[0] != null) {
                        reputation = e[0].compute(rs);
                }
                if (e[1] != null) {
                        trust = e[1].compute(ts);
                }
                //tsw is an instance of a trust statement writer, a JMS client
                //which actually knows how to send data to which queue
                tsw.write(new TrustStatement(rs,e[0],e[1]));

                //Optionally, if the target is a reputation statement (say rs2)
                 //it could retrieve this statement and write a new reputation
                 //statement (rs3) with the source and claims of rs, and the target
                 //of rs2
        }
}
```

### 6.2.5 Deployment

The decision on how deployment is done is of paramount importance when pursuing high performance behaviour in environments with thousands of users. There are several choices, such as:

- Everything on the same machine: this is the simplest deployment option. It does not scale and does not allow for failover capabilities in case of electrical problems.
- The application server on one machine, the trust server and the DBMS server on other machine: an intermediate solution where the trust server can be replicated on-demand, offering a higher scalability.
- Everything on different machines: this is the most flexible choice, although it is more vulnerable to network and bandwidths problems.

Figure 13 shows the first and the third deployment options discussed above.

The next section ties together all the concepts discussed here by showing how the framework can be applied in a social cloud scenario.

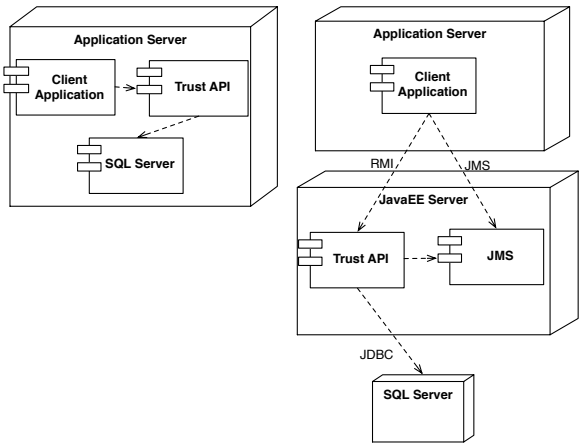


Figure 13: Two Deployment Configurations

# 7 Implementing a Social Cloud Application

We revisit the scenario described in Section 3, but now, we put ourselves in the developer's shoes, who is in charge of implementing it. Even though the framework should provide some built-in functionality, we are going to assume that the developer has to define everything from the ground-up.

The following list describes a sub-set of trust and reputation requirements that may involve the social cloud application:

1. Cloud providers can rate web services with one, two or three stars. When cloud providers rate a web service, this affects the reputation of the web service and the trust relationship between the evaluator and the web service creator.
2. Cloud providers can rate each other by using a 'I like' and 'I don't like' statement. When cloud providers rate other providers, this affects only the trust relationship between them, but not the reputation.
3. Reading a cloud provider profile increases the capability belief of the reader.
4. Reputation scores in the context 'WebServiceForOffice' must be normalized to the range [0..1] prior to being written in the trust database, and must be denormalized to the original range of the model prior to being sent to the application.[4]

The realization of the first requirement would be as follows. As the framework updates trust information after an event has occurred, the first step for the developer is deciding the name of the event. Then, he must decide the claim type that is associated to this event. In this case, the event name could be 'WebServiceRating'. The following code snippet creates a claim:

```
//BoundedClaim is a claim that is (de)normalizable according to
//a linear transformation. This claim would be offered by default
public class BoundedClaim extends Claim {

        int value, minimum, maximum;
        String name;

        BoundedClaim(String n, int v, int min, int max) {
                name = n;
                value = v;
                minimum = min;
                maximum = max;
                isNormalizable = true;
        }

        public float normalize() {
                //Applies a linear transformation
                //[minimum,maximum] -> [0,1]
        }
```

[4]We assume that the reputation engine correctly implements the model, and that the developer knows the model and, therefore, knows the model range.

```
        public int denormalize() {
                //Applies a linear transformation
                //[0,1] -> [minimum,maximum]
        }
}
```

Then, it is required to associate the claim to the event. This can be done by configuring an XML file as shown next.

```
<EventClaim evId=‘‘EV Example’’ Engine=‘‘CounterUp’’>
        <Event>
                <Name>WebServiceRating </Name>
        </Event>
        <Claim>
                <Name>WebServiceRating</Name>
                <Class>BoundedClaim</Class>
                <Min>1<Min>
                <Max>3<Max>
                <Value>EVENTVALUE</Value>
        </Claim>
 </EventClaim>
```

The developer is specifying that when an event with name WebServiceRating is triggered, a BoundedClaim object must be created with the parameters name, minimum, maximum and value. The value parameter is given by the value parameter in the event object (EVENTVALUE).

The next step is creating the trust and reputation engines. This is done by extending the *Engine* abstract class and implementing the *compute* method, as shown in the next code snippet:

```
public class RepExampleEngine extends Engine {

        //It computes the target's reputation by multiplying the
        //claim value by the reputation of the source
        public float compute(ReputationStatement rs) {
                return rs.getClaim().getValue() * rs.getSource().getReputation();
        }
}

public class TrustExampleEngine extends Engine {

        //It computes the trust value between the trustor and the trustee
        //by multiplying the trustor's belief in the target's capability by
        //the claim value. Note that the result is not a number, but
        //a string: TRUSTWORTHY or UNTRUSTWORTHY
        public String compute(ReputationStatement rs) {
                List<beliefs> lb = rs.getSource().getBeliefs();
                float capBelief = retrieve(lb, "capability", target);
                float aux = capBelief * rs.getClaim().getValue();

                if (aux > THRESHOLD) {
                        return 'TRUSTWORTHY';
                } else {
                        return 'UNTRUSTWORTHY';
                }
        }

        private float retrieve(List<beliefs> lb, String name, HumanEntity target) {
                //This method retrieves the value of the belief with name 'name'
                //about an entity 'target' from the list of beliefs 'lb'
        }
}
```

Finally, the developer would need to configure the computation rules by XML, as shown next:

```
<CompRule RuleId=‘‘CompRule Example’’ RepEngine=‘‘RepExampleEngine’’

        <Context>
                <Name>any<Name>
        </Context>
        <Claim>
                <Name>WebServiceRating</Name>
        </Claim>
</CompRule>
```

The developer is specifying that, no matter which the context is, if the name of a claim is 'WebServiceRating', then apply 'RepExampleEngine' and 'TrustExampleEngine' as reputation and trust engines respectively.

Once all this is configured, the developer only needs to retrieve a trust server instance from a JNDI directory and make the corresponding API call, where *eventName* should be 'WebServiceRating':

```
public void sendEvent(String eventName, String ctx, String source, String target)
```

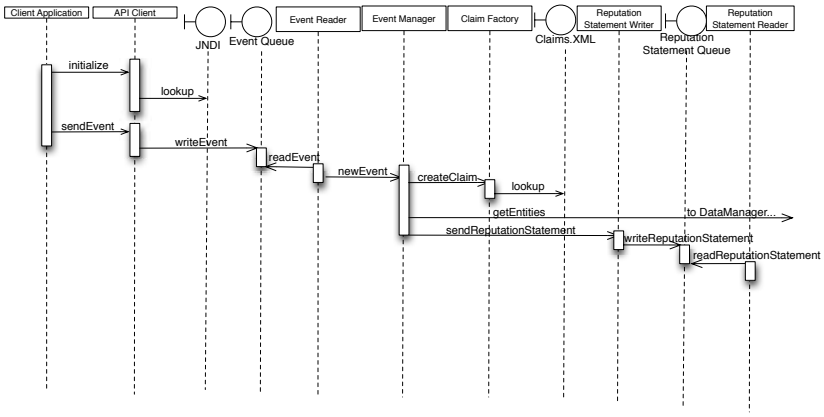


Figure 14: Detailed Sequence Diagram of *sendEvent* API Call

A detailed sequence diagram of this use case is depicted in Figure 14 and Figure 15. The server, upon receiving the call, creates the *Event* and sends it to a the *Events Queue*. From that moment, the client application can continue its execution. An *Event Manager* instance is asynchronously notified by an *Event Reader* that a new *Event* is available for processing, and it creates a *Claim* according to the rules specified in the XML file. The *Event Manager* also retrieves the complete information about the entities using a *Data Manager* instance, and with all this information it creates a *Reputation Statement*, which is sent to the *Reputation Statement Writer*, which in turn writes it in the *Reputation Statement Queue*.

The *Engine Manager* is asynchronously notified by a *Reputation Statement Reader* instance that a new *Reputation Statement* is in the queue ready for processing, and it creates an *Engine* using the computation rules codified in the XML file. The values returned by the reputation and trust engines are encapsulated in a *Trust Statement* instance by the *Engine Manager*, and sent to a *Trust Statement Writer* instance, which writes it into the *Trust Statement Queue*.

Finally, the *Data Manager* is notified by a *Trust Statement Reader* instance upon the arrival of the new *Trust Statement*. Using the JDBC connector, it translates the *Trust Statement* into the appropriate INSERT/UPDATE statements in the *Entity*, *Reputation Statement* and *Trust Relationship* tables.

An instantiation of this use case could be as follows. Let us assume that a cloud provider CP1 rates the web service created by another provider CP2 with 3 stars. The resulting tables after this event occurs are shown in Table 5, Table 6 and Table 7.

The second requirement is basically the same as the previous one, therefore the steps are pretty much the same. However, there are two major differences. The first one is that the value of the new claim is set to *null* (there is no value in a 'I Like' or 'I don't Like' statement) and therefore it is not possible to normalize it. The second one is that the reputation engine is set to *null* in the computation rules XML file, since no reputation update is required. The developer, upon detecting that a a cloud provider has rated another provider, would call the *sendEvent* call with the new event name provided by himself.

Regarding the third requirement, updating trust-related information (such as beliefs) does not require the developer to trigger an event, but only to use the API call that requests the *Data Manager* to update this information. Therefore, upon detecting that a cloud provider has seen other provider's profile, it would perform a similar call as the one presented next:

```
public void increaseBelief(String beliefName, String ctx,
                String source, String target, float quantity)
```

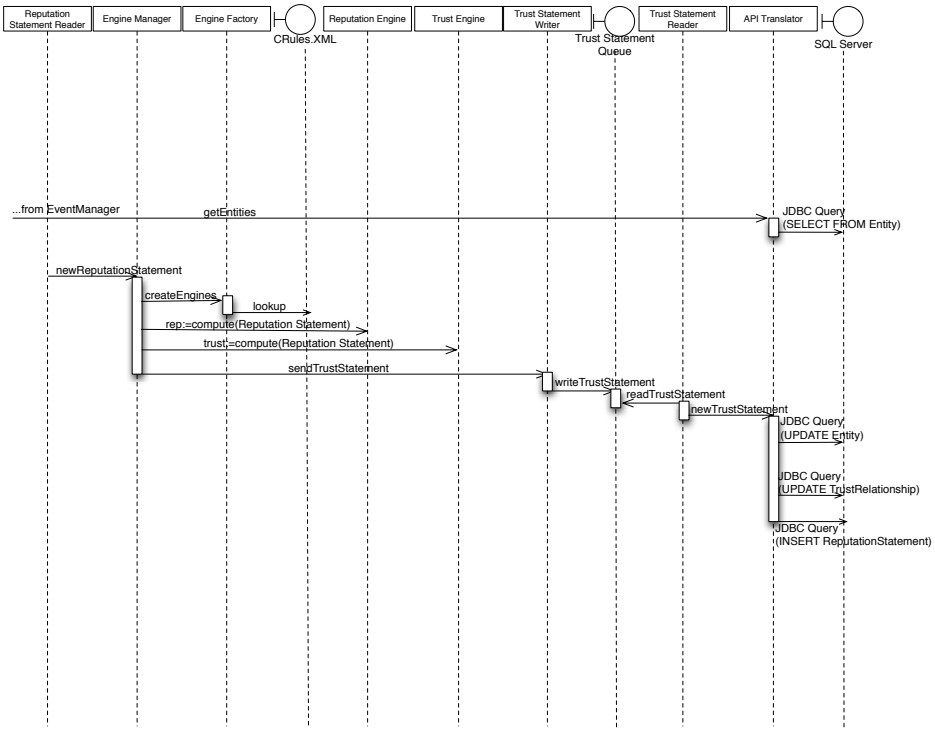


Figure 15: Detailed Sequence Diagram of *sendEvent* API Call (cont.)

Table 5: Entity Table Example

| Attribute | CP1 | WebService | RepStatement |
|---|---|---|---|
| ID | CP1-ID | CP2-WebService0123 | RepStatementCP1-CP2 |
| Type | Human | Non-Human | RepStatement |
| Reputation | 0,7 | 2,1 | - |
| Context | WebServiceRatingCtx | WebServiceRatingCtx | WebServiceRatingCtx |
| Claim | AnotherClaim | BoundedClaim | - |
| Time | 10-05-Mon25Sep2012 | 14-30-Tue05Dec2011 | - |
| LastTime | 10-05-Mon25Sep2012 | 09-24-Wed24Jan2012 | - |
| Number of evaluations | 1 | 5 | 0 |

Table 6: Trust Relationship Table Example

| Attribute | CP1 - CP2 |
|---|---|
| ID | CP1-CP2-RelID |
| ID Trustor | CP1-ID |
| ID Trustee | CP2-ID |
| Trust value | TRUSTWORTHY |
| Context | WebServiceRatingCtx |
| Number of evaluations | 1 |
| Time | 09-24-Wed24Jan2012 |
| Last Time | 09-24-Wed24Jan2012 |

As for the last requirement, the framework should offer hot spots or hooks in order to provide the developers with extension points for application-specific needs. The methods of the abstract class *InfoFilter*[5] play this role.

```
abstract class InfoFilter {

        //This method is called right before an engine receives a
```

[5]In order to keep the architecture cleaner, and also because this class may belong to a more detailed design, we haven't mentioned it earlier.

Table 7: Reputation Statement Table Example

| Attribute | RepStatement |
|---|---|
| ID | RepStatementCP1-CP2 |
| Source | CP1 |
| Target | CP2-WebService0123 |
| Claim | BoundedClaim |
| Claim value | 3 |
| Context | WebServiceRating |
| Time | 09-24-Wed24Jan2012 |

```
//reputation statement
public ReputationStatement beforeComputation(ReputationStatement rs);

//This method is called right after an engine computes a
//trust statement
public TrustStatement afterComputation(TrustStatement ts);

//This method is called right after retrieving some trust information
//and right before sending this information to the client application
public RepValue afterRetrieval(RepValue rv);
```

The developer would need to extend this class and implement the methods according to his needs. In the case of the last requirement, a reputation score in the context 'WebServiceForOffice' must be normalized in the range [0...1], and denormalized to the original range prior to being sent back to the application. Therefore, the developer would need to place the normalization code in the *afterComputation* method, and denormalization code in the *afterRetrieval* method.

# 8 Conclusion and Future Work

We have presented a trust framework that may assist developers to implement applications that need to take into account trust and reputation requirements. This kind of applications are steadily emerging, responding to an increasing demand of users eager to participate in collaborative environments. We have seen this trend with the success of blogs and social networks.

Social cloud applications go one step further by turning users into service providers, which raises lots of security and trust concerns. This calls for holistic approaches that address these concerns. We argue that trust and reputation requirements become specially crucial elements in order to leverage security and foster the adoption of this kind of applications.

In spite of their importance, trust and reputation are often discussed in the literature from a theoretic perspective: hundreds of trust and reputation models can be found, but few works address them from a more pragmatic point of view. Moreover, the concepts of trust and reputation are often confused and mixed up, leading to models that do not address appropriately the trust requirements.

In order to overcome these shortcomings, we have proposed a trust framework, which focuses on helping developers to build trust- and reputation-aware applications. We believe that counting on this kind of solutions can greatly simplify the task of implementing successful social cloud applications that users can trust and are willing to use.

We are also aware that learning a framework may be a daunting task. For this reason, we have tried to keep a simple, event-based design. We also advocate that the hard work of learning a new framework can significantly pay off the burden of hard-coding trust and reputation solutions from scratch every time, in addition to enable the maintainability of the whole application.

There are, however, several issues that require further attention and that we plan to explore as future work. Up to now, the framework design does not support the implementation of propagation models. Therefore, no trust information can be transferred between entities. For the inclusion of these models, it is first required to add algorithms that generate trust chains (i.e. graphs) from the trust relationships stored by the trust server. The developer can then define rules to transfer trust between the entities following the trust paths. The output would be new trust relationships between entities that have not had a personal direct encounter.

Also, a validation on a real scenario is required to measure the performance of the framework. At this point, some relevant research questions arise. For example, is the trust framework actually improving the security of the application and the decision-making processes?; or, is it easier for developers to use the framework rather than to hard-code the trust and reputation functionality into the application?

In systems including trust and reputation models, there are usually human users behind the decision-making processes and trust/reputation dynamics. Users usually provide feedback after certain operations and trigger events that update trust relationships and reputation values. It would be interesting to reflect on how the notions of trust and reputation can be effectively tailored for software components and physical devices, without any kind of human interaction. In this direction, we should decide in which terms a software component can trust another component. Even more interestingly, we could ask ourselves whether trust and reputation information can be used to make autonomic reconfiguration decisions on the software architecture, or whether software developers can be provided with relatively easy tools to achieve this.

## Acknowledgements

This work has been partially funded by the European Commission through the FP7/2007-2013 project NESSoS (www.nessos-project.eu) under grant agreement number 256980, and by the Junta de Andalucia through the project FISICCO (P11-TIC-07223). The first author is funded by the Spanish Ministry of Education through the National F.P.U. Program.

## References

[1] Jemal Abawajy. Determining Service Trustworthiness in Intercloud computing environments. In *Proceedings of the 2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks*, ISPAN '09, pages 784–788, Washington, DC, USA, 2009. IEEE Computer Society.

[2] Isaac Agudo, Carmen Fernandez-Gago, and Javier Lopez. A Model for Trust Metrics Analysis. In *5th International Conference on Trust, Privacy and Security in Digital Business (TrustBus'08)*, volume 5185 of *LNCS*, pages 28–37. Springer, 2008.

[3] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized Trust Management. In *IEEE Symposium on Security and Privacy*, pages 164–173, 1996.

[4] Buyya, Rajkumar and Yeo, Chee Shin and Venugopal, Srikumar and Broberg, James and Brandic, Ivona. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Gener. Comput. Syst.*, 25(6):599–616, June 2009.

[5] Scott Cadzow. Making better security standards: A review of the security update to mbs and a new etsi deliverable. Technical report, ETSI TISPAN, 2008.

[6] Vinny Cahill, Elizabeth Gray, Jean-Marc Seigneur, Christian D. Jensen, Yong Chen, Brian Shand, Nathan Dimmock, Andy Twigg, Jean Bacon, Colin English, Waleed Wagealla, Sotirios Terzis, Paddy Nixon, Giovanna di Marzo Serugendo, Ciaran Bryce, Marco Carbone, Karl Krukow, and Mogens Nielsen. Using Trust for Secure Collaboration in Uncertain Environments. *IEEE Pervasive Computing*, 2(3):52–61, July 2003.

[7] Christiano Castelfranchi and Rino Falcone. *Trust Theory: A Socio-Cognitive and Computational Model*. Wiley Series in Agent Technology, Jun 2010.

[8] Kyle Chard, Simon Caton, Omer Rana, and Kris Bubendorfer. Social Cloud: Cloud Computing in Social Networks. In *Proceedings of the 3rd International Conference on Cloud Computing IEEE Cloud 2010*, 2010.

[9] Bruce Christianson and William S. Harbison. Why Isn't Trust Transitive? In *Proceedings of the International Workshop on Security Protocols*, pages 171–176, London, UK, UK, 1997. Springer-Verlag.

[10] Mohamed E.Fayad, Douglas C.Schmidt, and Ralph E.Johnson. *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. Wiley, Septembre 1999.

[11] Randy Farmer and Bryce Glass. *Building Web Reputation Systems*. Yahoo! Press, USA, 1st edition, 2010.

[12] Diego Gambetta. Can We Trust Trust? In *Trust: Making and Breaking Cooperative Relations*, pages 213–237. Basil Blackwell, 1988.

[13] Tyrone Grandison and Morris Sloman. A Survey of Trust in Internet Applications. *Communications Surveys & Tutorials, IEEE*, 3(4):2–16, 2000.

[14] Sheikh Mahbub Habib, Sebastian Ries, and Max Muhlhauser. Cloud Computing Landscape and Research Challenges Regarding Trust and Reputation. In *Proceedings of the 2010 Symposia and Workshops on Ubiquitous, Autonomic and Trusted Computing*, UIC-ATC '10, pages 410–415, Washington, DC, USA, 2010. IEEE Computer Society.

[15] Chern Har Yew. *Architecture Supporting Computational Trust Formation.* PhD thesis, University of Western Ontario, London, Ontario, 2011.

[16] TD Huynh. A Personalized Framework for Trust Assessment. *ACM Symposioum on Applied Computing - Trust, Reputation, Evidence and other Collaboration Know-how Track*, 2:1302–1307, December 2008.

[17] Audun Jøsang. A Logic for Uncertain Probabilities. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 9(3):279–311, June 2001.

[18] Audun Jøsang, Roslan Ismail, and Colin Boyd. A Survey of Trust and Reputation Systems for Online Service Provision. *Decision Support Systems*, 43(2):618–644, March 2007.

[19] Rolf Kiefhaber, Florian Siefert, Gerrit Anders, Theo Ungerer, and Wolfgang Reif. The Trust-Enabling Middleware: Introduction and Application. Technical Report 2011-10, Universittsbibliothek der Universitt Augsburg, Universittsstr. 22, 86159 Augsburg, 2011. http://opus.bibliothek.uni-augsburg.de/volltexte/2011/1733/.

[20] Adam J. Lee, Marianne Winslett, and Kenneth J. Perano. TrustBuilder2: A Reconfigurable Framework for Trust Negotiation. In Elena Ferrari, Ninghui Li, Elisa Bertino, and Ycel Karabulut, editors, *IFIPTM*, volume 300 of *IFIP Conference Proceedings*, pages 176–195. Springer, 2009.

[21] Raph Levien. *Attack Resistant Trust Metrics.* PhD thesis, University of California at Berkeley, 2004.

[22] Noura Limam and Raouf Boutaba. Assessing Software Service Quality and Trustworthiness at Selection Time. *IEEE Trans. Softw. Eng.*, 36(4):559–574, July 2010.

[23] Stephen Marsh. *Formalising Trust as a Computational Concept.* PhD thesis, University of Stirling, April 1994.

[24] D. Harrison McKnight and Norman L. Chervany. The Meanings of Trust. Technical report, University of Minnesota, Management Information Systems Research Center, 1996.

[25] Keith W Miller, Jeffrey Voas, and Phil Laplante. In Trust We Trust. *Computer*, 43:85–87, 2010.

[26] Haralambos Mouratidis and Paolo Giorgini. Secure tropos: a security-oriented extension of the tropos methodology. *International Journal of Software Engineering and Knowledge Engineering*, 17(2):285–309, 2007.

[27] Francisco Moyano, Carmen Fernandez-Gago, and Javier Lopez. A conceptual framework for trust models. In Simone Fischer-Hübner, Sokratis Katsikas, and Gerald Quirchmayr, editors, *9th International Conference on Trust, Privacy & Security in Digital Business (TrustBus 2012)*, volume 7449 of *Lectures Notes in Computer Science*, pages 93–104, Vienna, Sep 2012 2012. Springer Verlag, Springer Verlag.

[28] D. Olmedilla, O.F. Rana, B. Matthews, and W. Nejdl. Security and Trust Issues in Semantic Grids. In *Proceedings of the Dagsthul Seminar, Semantic Grid: The Convergence of Technologies*, volume 5271, 2005.

[29] Michalis Pavlidis, Haralambos Mouratidis, and Shareeful Islam. Modelling security using trust based concepts. *IJSSE*, 3(2):36–53, 2012.

[30] Michalis Pavlidis, Haralambos Mouratidis, Shareeful Islam, and Paul Kearney. Dealing with trust and control: A meta-model for trustworthy information systems development. In *Sixth International Conference on Research Challenges in Information Science*, pages 1–9. IEEE, 2012.

[31] Paul Resnick and Richard Zeckhauser. Trust Among Strangers in Internet Transactions: Empirical Analysis of eBay's Reputation System. In Michael R. Baye, editor, *The Economics of the Internet and E-Commerce*, volume 11 of *Advances in Applied Microeconomics*, pages 127–157. Elsevier Science, 2002.

[32] Sini Ruohomaa and Lea Kutvonen. Trust management survey. In *Proceedings of the Third international conference on Trust Management*, iTrust'05, pages 77–92, Berlin, Heidelberg, 2005. Springer-Verlag.

[33] Girish Suryanarayana, Mamadou Diallo, and Richard N. Taylor. A Generic Framework for Modeling Decentralized Reputation-based Trust Models. In *The Fourteenth ACM SigSoft Symposium on Foundations of Software Engineering*, 2006.

[34] Girish Suryanarayana, Mamadou H. Diallo, Justin R. Erenkrantz, and Richard N. Taylor. Architectural Support for Trust Models in Decentralized Applications. In *Proceeding of the 28th international conference*, pages 52–61, New York, New York, USA, 2006. ACM Press.

[35] Hassan Takabi, James B. D. Joshi, and Gail-Joon Ahn. Security and Privacy Challenges in Cloud Computing Environments. *IEEE Security and Privacy*, 8(6):24–31, November 2010.

[36] Aaron Weiss. Computing in the clouds. *netWorker*, 11(4):16–25, December 2007.

[37] Phillip J. Windley, Kevin Tew, and Devlin Daley. A Framework for Building Reputation Systems. `http://www.windley.com/essays/2006/dim2006/framework_for_building_reputation_systems`, 2006.

[38] Marianne Winslett, Ting Yu, Kent E. Seamons, Adam Hess, Jared Jacobson, Ryan Jarvis, Bryan Smith, and Lina Yu. Negotiating Trust on the Web. *IEEE Internet Computing*, 6(6):30–37, November 2002.

[39] Yanping Xiao, Chuang Lin, Yixin Jiang, Xiaowen Chu, and Xuemin Shen. Reputation-based QoS Provisioning in Cloud Computing via Dirichlet Multinomial Model. In *IEEE International Conference on Communications*, pages 1–5. IEEE, 2010.

[40] Zheng Yan and Silke Holtmanns. Trust Modeling and Management: from Social Trust to Digital Trust. *Computer Security, Privacy and Politics: Current Issues, Challenges and Solutions*, January 2008.