A Framework for Building Reputation Systems

Phillip J. Windley, Ph.D.
xri.net/=windley
www.windley.com

Kevin Tew
tewk {at} tewk.com
blog.tewk.com

Devlin Daley
devlin.daley {at} gmail.com

Dept. of Computer Science
Brigham Young University

Abstract

This paper introduces a set of principles for governing the design and operation of online reputation systems. We also introduce the design, architecture, and implementation of a flexible, general-purpose framework. called Pythia, for building reputation systems and show an example application of the framework.

Keywords

identity, reputation, framework, blogging

Introduction

Most online interactions are devoid of many of the cues that people use in the physical world to made judgments about the character, stability, reliability, etc. of people, systems, and other entities. Yet these cues are critical to sophisticated interactions and transactions. Online reputation systems attempt to remedy this situation.

A reputation system is an online application for providing such cues in an algorithmic way. The results from a reputation system are used by other systems that need to assess past performance of a person or other entity in making decisions about their likely future performance or their intent.

Online reputation can be accessed informally or formally. Informal assessments are made by people from their experience on the Internet. For example, I may not trust email I get from the domain `hotmail.com` because I've received substantial Spam from addresses in the domain in the past. Formal systems collect and process online information about specific identifiers in an attempt to calculate reputation scores in a more systematic manner.

This paper presents a set of principles that we believe should govern the operation of general-purpose, online reputation systems. The paper also presents the design and implementation of a reputation framework that is consistent with these principles and that can be specialized to particular tasks. The framework, called Pythia, has been designed and implemented using a rules engine and plug-in architecture to make it flexible for use in a variety of contexts.

Reputation Principles

We believe there are several important principles that apply to reputation systems and should govern their design and operation. The following principles are the ones we have identified. These are based on our own research as well as a group discussion from the Berkman Identity Mashup in June 2006 [Wind06a].

Reputation is one of the factors upon which trust is based. The is much confusion between trust and reputation. We consider reputation a building block for trust. We are not concerned in this paper with what other factors go into trust, how trust is built, or how trust is exchanged.

Reputation is someone else's story about me. I can't control what you say about me although I may be able to affect the factors you based your story on. Every person should be able to have their own story about me.

Reputation is based on identity. Reputation, as someone else's story, isn't part of your identity, but is based on an identity or set of identities.

Reputation exists in the context of community. Any given context will have specific factors for what is important in determining reputation. This is different than saying "communities have a reputation about someone." Communities do not have beliefs, only people have beliefs including beliefs about what others believe.

Reputation is a currency. While you can't change reputation directly, reputation can be used as a resource. For example, Paul Resnick *et. al.* has shown the value of a positive eBay reputation [Res00a].

Reputation is narrative. Put another way, reputation varies with time. Reputation is dynamic because the factors that affect it are always changing. Reputation may require weaving together plot lines from different contexts.

Reputation is based on claims (verified or not), transactions, ratings, and endorsements.. How these factors are used in determining reputation is up to each individual. Individuals may use various evidence in making claims or proposing a certain rating or endorsement. The penalty for making false claims or giving false endorsements varies from context to context.

Reputation is multi-level. A reputation isn't just based on facts, but is also based on other's beliefs about the target of the reputation. These beliefs are signaled to others in various ways depending on the context.

Multiple people holding the same opinion increases the weight of that opinion. Reputation systems should have some way of weighting scores. As a related issue, repeat behavior is another way of weighting reputation.

These reputation principles guided our design as we built the reputation system described in this paper. We view these principles as universal while recognizing that there is still room for discussion on the particulars. We invite feedback and discussion about these principles.

Design Philosophy

Beyond the principles discussed in the last section, we also determined a design philosophy--a collection of design decisions that were important for what we wanted to accomplish.

Our design philosophy is central to our system. Changing this philosophy would fundamentally change the nature of the system we built. Unlike the principles we discussed earlier, the philosophy is mutable and could be changed depending on the end goal of the system.

Reputation is about people. This decision is sure to be controversial, but during our research, we found it difficult to come up with abstractions and nomenclature that worked equally well for human and non-human, especially inanimate, entities. This we concluded that we would design the system assuming that it would be used to determine the reputation of humans. What we have produced could be bent to provide reputation in other circumstances, but it was not designed to be so used.

In the discussions that follow, we will refer to two groups of entities. The first is the group we call "users." Users are the targets of a reputation request. Our system aggregates certain kinds of information about users and uses that information in processing reputation requests.

The second group is a subgroup of the users that we call "relying parties." A relying party is a user making a reputation request of about another user. Any user can be a relying party.

Users are identified by one or more identifiers. Multiple identifiers (email addresses, URLs, phone numbers, etc.) can be associated with a single user. The identifiers are validated where possible.

Reputation is calculated. For our purposes, reputation can be represented as function:

$$R_u = F_{rp}(I_u, Tx_{u,u'}, E_u)$$

where

rp is the relying party's identifier
u is the user id
u' is all users
I_u is a vector of verified identifiers for u
$Tx_{u,u'}$ is a vector of transactions between u and every other user in the system
E_u is a vector of ratings and endorsements for u

While reputation is calculated, it doesn't necessarily follow that the result is a single number. It's possible that the function could return a vector of values if that was appropriate.

Transactions are jointly owned by the parties to the transaction. The exact nature of the transactions in the system depends on the particular use to which it's being put. Transactions always contain an identifier for each party to the transactions and a timestamp in addition to the facts pertinent to the transactions as defined by the use. Ownership implies that all parties to a transaction can see it.

Transactions are persistent and immutable. Once the system has recorded a transaction, it cannot be deleted by any of the parties to the transaction. A later transaction may record that it was nullified, but cannot remove it.

Transparency. Our system is designed to favor transparency wherever possible. Consequently,

- Users can see any information the system knows about them (e.g. transactions they are party to)
- Users can see any reputation queries about them, what the results were, and how the result was arrived at.
- Users who are *not* party to a transaction cannot see it, but can use it to calculate reputation about any of the parties to the transaction.

Online resources should be consulted whenever possible to garner reputation information. Claims can be validated in various ways. For example, an email address can be validated by sending a unique URL to the address that the user clicks on to validate receipt. URLs can be validated by having the user embed a unique code in the HTML for the page. Data stores such as the `whois` database and Netcraft can be used to gather information about domains and Web sites respectively. These online sources of information provide facts that can be used as evidence in a reputation computation.

Architecture

The architecture of Pythia is based on the foregoing principles.

- Pythia is identity system neutral and stores multiple identifiers for any user.
- Pythia uses a rules engine to allow reputation calculations to be customized by each relying party according to their individual needs.
- Pythia implements a plugin architecture to customize the context for reputation computation.

Figure 1 shows the high-level block diagram architecture for Pythia.

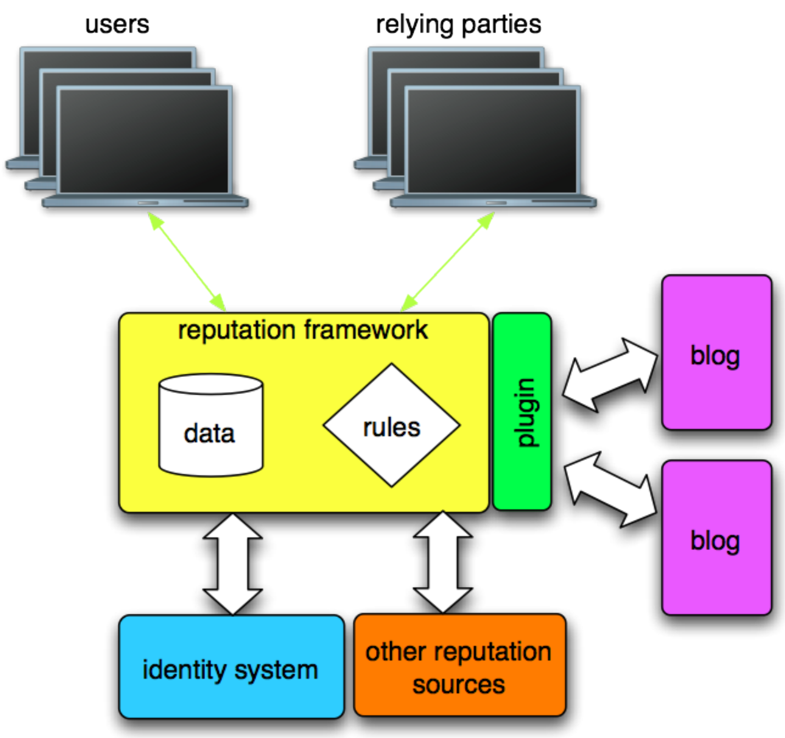


Figure 1: Reputation Framework Architecture

The framework is identity system neutral. Pythia is not an identity system, but is meant to rely on one or more existing identity systems for authentication and user IDs. As implemented, Pythia uses OpenID as an authentication mechanism, but other authentication systems could also be used.

The framework allows users to claim and verify identifiers. At present, these identifiers are limited to email addresses and URLs, but this is only an implementation limitation. Email addresses are verified by sending the email address a message asking the user to click on a URL to claim the email address in Pythia. URLs are verified using MicroIDs [Mic06], a standard for claiming online resources.

Relying parties can use the built-in rules engine to create rule sets for calculating reputation based on information in the system as well as other reputation information from the Internet. A global set of system-defined rules can be used as-is or customized by relying parties for their particular purpose.

The plugin architecture is the device for allowing the framework to be specialized for a particular task. The plug-in defines domain specific transactions that are to be recorded, the API that relying parties will use to record transactions, and any functions that are added to the rules-engine specific to the domain.

Implementation

The Pythia framework is implemented using Ruby on Rails. The system implements user, relying party, and administrator Web interfaces. The user interface allows users to claim and validate identifiers, view transactions to which they were a party, and view any reputation requests that were made about them.

The relying party interface is available to any user. Using this interface, relying parties can create and manage rule sets that are used to compute reputation.

Authentication for the framework is provided using OpenID. The Ruby libraries for OpenID are from openidenabled.com.

Plugins

As noted earlier, the framework was designed to calculate and store reputation information for a variety of applications. Although the framework itself is abstract, specific communication and transaction messages are necessary for each integrating system. The framework is shown how to communicate with these systems through the use of plugin extensions.

Plugins define a transaction format, allowing the framework to specialize the reputation computation to data appropriate for a specific application. For example, an online forum might want to report comments being posted as belonging to "veteran" members, as this might be an important part of the reputation for the forum. This same transaction attribute would not be reported by a blog system.

Transactions are transferred as XML through the relying party interface, a RESTful API. Plugins define the data types and the message contents for transactions in the API. Plugins are implemented as XML files that are loaded by the system at startup, allowing plugins to be easily installed, managed and transferred to other installations. Transactions enumerate attributes that include a name, a type (text, numeric, boolean) and a regular expression for determining valid values before type conversion.

Plugins may also include default rule sets. These rule sets can be cloned by users of the system and subsequently modified. Rule sets are executed by the rules engine to determine a reputation score. Future work will include adding other reputation sources also by way of plugins.

Data Model

To simplify development, the Sqlite3 database was used for data storage, but there are no specific Sqlite3 dependencies in the code. Database tables and relationships are implemented as Ruby objects using the ActiveRecord ORM libraries. Plugins are stored after being read from XML files into tables. Transactions are also stored in the database along with the rule sets which operate on them.

Rules Engine

The underlying rules engine used by Pythia is Rools (see http://rools.rubyforge.org/). Rule sets can be defined by plugins or through the Web interface by relying parties.

Rule sets are a named grouping of rules. Rules specify filters, conditions and actions. Filters only permit the desired transactions to participate in the reputation calculation. Some example filters would be, "transactions where the customer satisfaction score is greater than five", or "transactions where a blog comment was rejected".

Conditions are simple boolean expressions which apply aggregate functions to the filtered transactions. Implemented aggregate functions are count, max, min, sum and average. Using the filters above, an example condition would be, "if the number (count) of transactions where a blog comment was rejected is less than ten". Permitted boolean expressions are <, >, ==, <=, >=.

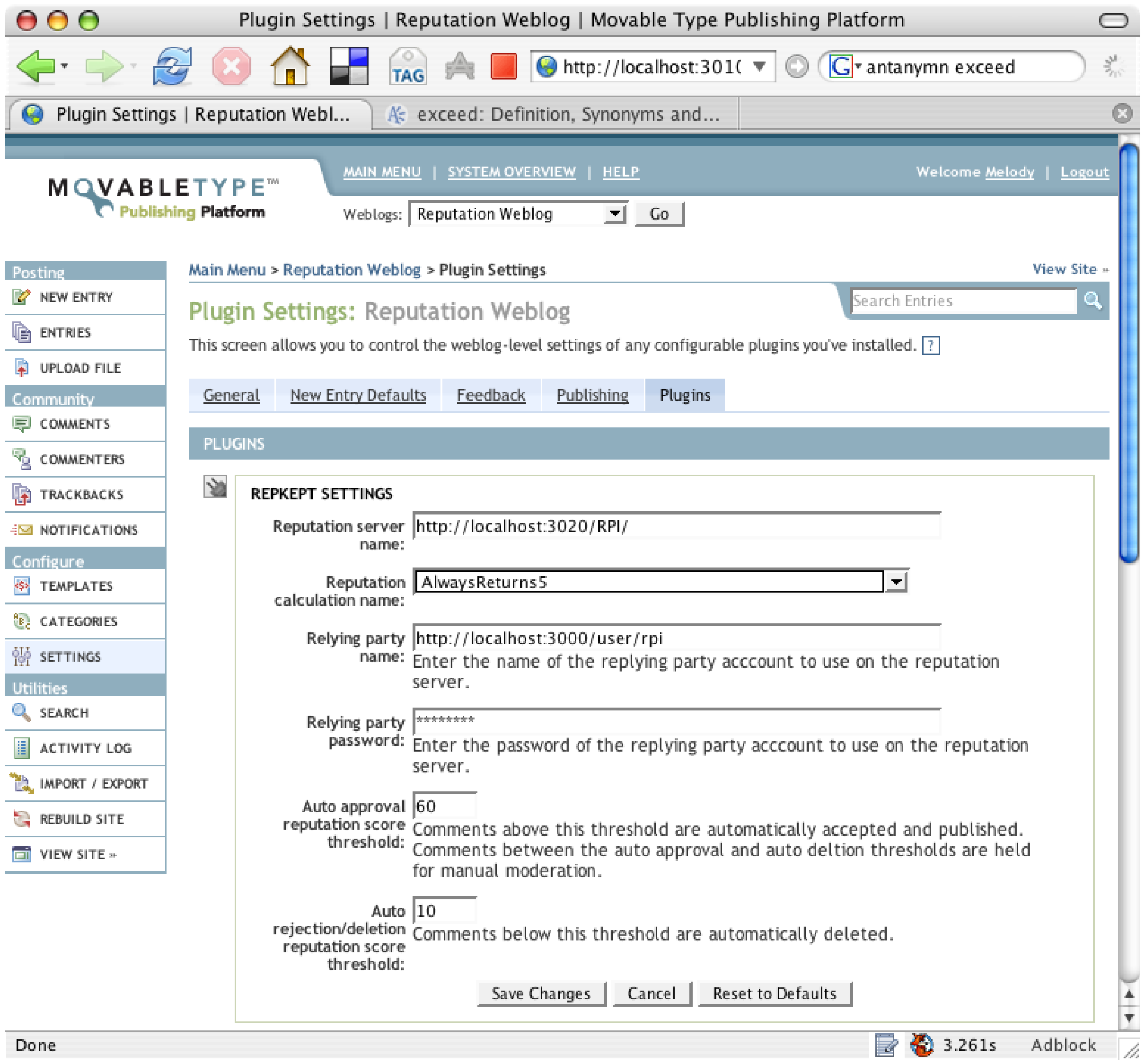
Actions modify the reputation score when rule conditions are satisfied. Presently, an action may add, subtract or multiply any amount to the current reputation score. If multiple rule conditions are satisfied, all corresponding rule actions will be executed.

An Example Application: Blog Comments

As part of the development and testing, we created a plug-in that allows Pythia to be used for blog comment moderation activities. A relying party's blog system can query the reputation of people leaving comments. Any actions taken by the relying party with respect to the comment (e.g. deleting it, approving it, etc.) are recorded as transactions. For purposes of the demo, we chose the MovableType blogging software, but any blogging system could be used.

Customizing the general purpose framework for comment moderation required two additions: a plugin for the reputation framework and a MovableType integration module. The reputation framework plugin defines its own schema for legal transaction types and transaction values specific to blog comments. The framework uses the plugin's schema definition to receive, store, and log blog comment transactions.

Figure 2 shows the configuration screen in MovableType for the reputation-based comment moderation system. The MovableType plugin we built to integrate with Pythia is called RepKept. The blog owner, acting as the relying party, specifies the URL of the reputation server and gives authentication information for the reputation server. The relying party also selects which rule set they want to use to calculate the reputation score and gives thresholds for various actions. The rule sets in the drop-down box are populated from Pythia and represent rule sets the relying party has created.



Plugin Settings | Reputation Weblog | Movable Type Publishing Platform

Plugin Settings: Reputation Weblog

This screen allows you to control the weblog-level settings of any configurable plugins you've installed. [?]

REPKEPT SETTINGS

Reputation server name: http://localhost:3020/RPI/

Reputation calculation name: AlwaysReturns5

Relying party name: http://localhost:3000/user/rpi
Enter the name of the replying party acccount to use on the reputation server.

Relying party password: ********
Enter the password of the replying party acccount to use on the reputation server.

Auto approval reputation score threshold: 60
Comments above this threshold are automatically accepted and published. Comments between the auto approval and auto deltion thresholds are held for manual moderation.

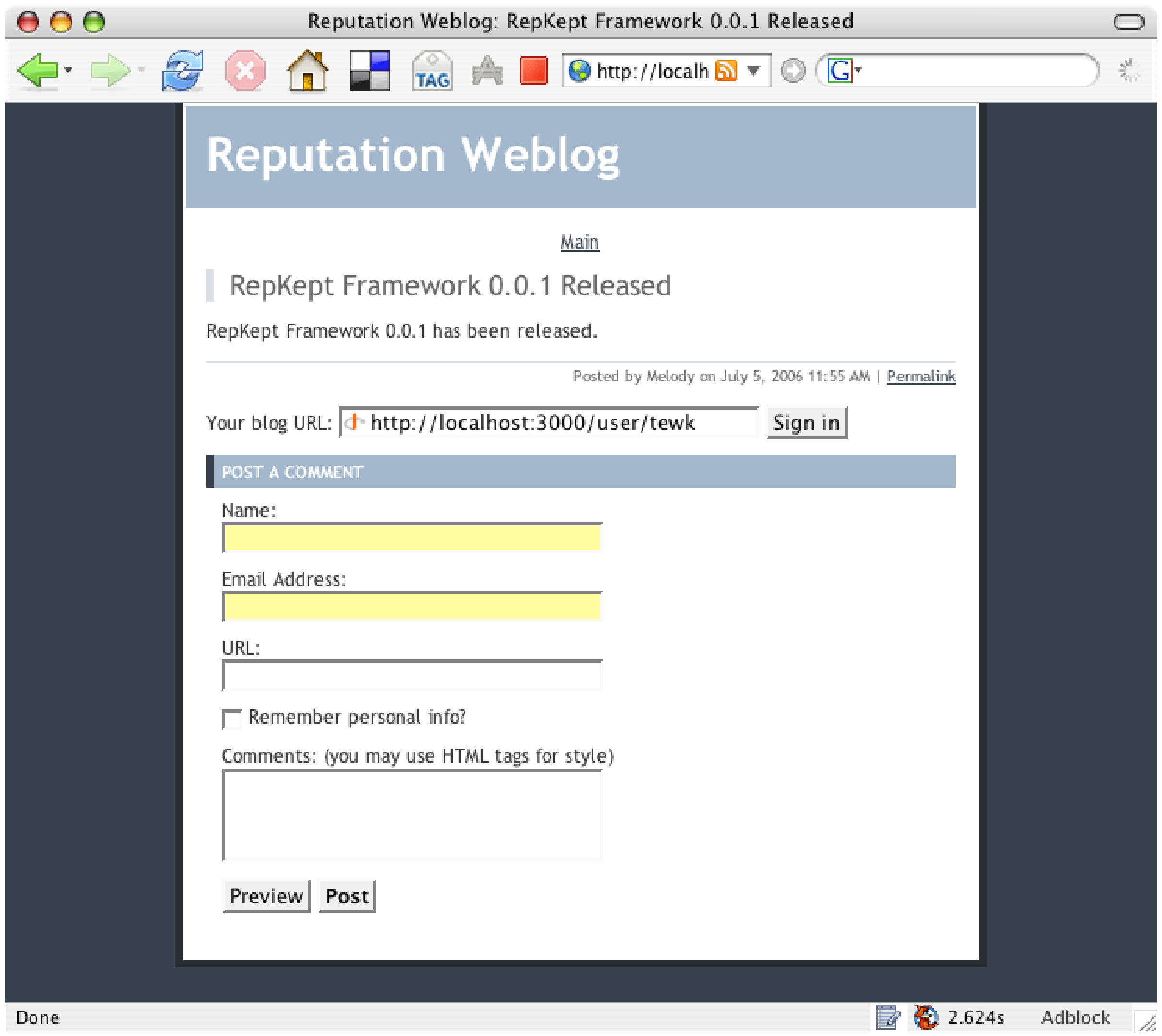
Auto rejection/deletion reputation score threshold: 10
Comments below this threshold are automatically deleted.

Figure 2: RepKept Configuration Screen

From the relying party's point of view, in this case the blog or blog owner, there are two types of interactions with Pythia. The first is a reputation request where the blog asks the reputation server for the reputation of a particular identity. The second is a feedback event where the relying party sends Pythia information relating to a comment it has processed. These feedback events are stored in Pythia as a transaction between the commenter and the relying party.

A reputation request consists of a digital identifier representing the relying party making the request, the digital identifier representing the subject of the request, and a rule set identifier. To complete the request, the reputation server looks up the rule set requested by the relying party and executes that rule set for subject of the request. The result of the reputation calculation is then returned for the relying party to act upon.

As shown in Figure 3, a commenter supplies his digital identifier by entering an OpenID-enabled URL into the "Your blog URL:" input box. When the commenter clicks the "Sign In" button, the relying party's blog contacts the commenter's OpenID server and authenticates the commenter's identity. The blog then returns the comment page showing that OpenID authentication has occurred.


Reputation Weblog: RepKept Framework 0.0.1 Released

http://localh

Reputation Weblog

Main

RepKept Framework 0.0.1 Released

RepKept Framework 0.0.1 has been released.

Posted by Melody on July 5, 2006 11:55 AM | Permalink

Your blog URL: http://localhost:3000/user/tewk Sign in

POST A COMMENT

Name:

Email Address:

URL:

Remember personal info?

Comments: (you may use HTML tags for style)

Preview Post

Done 2.624s Adblock

Figure 3: Commenter's OpenID login prompt

Once the commenter has authenticated, the blog has a valid identifier to use for requesting reputation and later submitting feedback. Note that authentication alone, particularly with lightweight systems like OpenID, does not provided any verifiable attributes about the commenter other than they have a valid OpenID somewhere on the Internet.

The reputation request is made and a reputation is computed according to the rule set the relying party has selected in the blog configuration. Depending on the value returned and the threshold values set by the relying party, the blog either automatically publishes the comment, holds it for moderation, or deletes it. Each of these represent a feedback event that generates a transaction that Pythia will log and use for future reputation computations about the relying party and the user.

When the reputation for a subject exceeds the auto-publish threshold of the relying party the comment immediately published and a `AutoPublished` feedback event is sent to the reputation server. In the case where a subject's reputation falls bellow the relying party's auto-rejection threshold the comment is immediately deleted and a `AutoDeleted` feedback event is sent. If a subject's reputation lands between the auto-reject and auto-publish thresholds the comment is placed in a queue for manual moderation. When a human takes action on a comment in the manual moderation queue either a `ManualPublish` or a `ManualDelete` feedback event is sent to the reputation server.

Much like the reputation request, feedback events contain the digital identifier of the relying party sending the feedback event and the digital identifier of the subject of the feedback event. The payload of a feedback event is simply a list of transaction records represented by a key/value pair. The key in this case is the transaction type. The value is a valid transaction value for the specified transaction type as defined by the plugin schema.

In the blog comment case a transaction type could be the `Commenter's IP address` and the corresponding transaction value could be `192.168.0.1`. A transaction can also be represented as the presence of an enumerated value. In this case the transaction type contains the enumerated value and the transaction value is empty. The feedback transactions in the blog comment example are predominately enumerated values. These values represent actions taken by the relying party given the reputation of the subject. The blog comment example has four types of feedback, each represented by a enumerated value style transaction. In this particular example the feedback transaction types are mutually exclusive. In general however this need not be the case.

Related Work

There has been significant past work on reputation systems and our work builds on and draws from that work. This section highlights a few of the most important papers in the development of the philosophy and design of Pythia.

In *Reputations Systems: Facilitating Trust in Internet Interactions* [Res00a], Paul Resnick *et. al.* define a reputation system as a system that "collects, distributes, and aggregates feedback about participants' past behavior" Their paper states two factors for how trust grows naturally in long-term relationships.

1. Interacting with a person over time, a history of past interactions characterizes the individual's character, abilities, and disposition.
2. Expectation of future reciprocity or retaliation creates an incentive for good behavior in the present.

Also included in the Resnick paper are three requisite characteristics of a successful reputation system.

1. Entities in the system must be longed lived in order to ensure a expectation of future interactions.
2. Feedback concerning interactions is recorded and distributed to entities in the future.
3. Recorded feedback on past interactions must have influence on the actions of entities in the future. Entities must pay some attention to reputations in the system during their decision making process.

Resnick *et. al.* also explain the challenges reputation systems face. There is a human reluctance to give negative feedback. In order to mitigate losses and failures, individuals turn to negotiation and arbitration in place of giving negative feedback. In some systems, fear of retaliation leads to an acceptance of mediocrity. Finally, disgust and desires to avoid further confrontation may cause individuals to leave bad service in the past and move on without leaving negative feedback.

Pythia incorporates many of these ideas in its design philosophy and the architecture.

Sierra is the reference implementation of OpenPrivacy.org's Reputation Management System (RMS). Reptile, a Syndicated Content Directory Server providing a personalized news and information portal with privacy and reputation accumulation, was the principle application built upon the Sierra RMS [Lab01a]. Reptile eventually became http://www.newsmonster.org/.

Sierra is composed of three services: Nym Service, Bias Management, and a Reputation Calculation Engine (RCE). The Nym Service, nym being an abbreviation of pseudonym, is simply a digital identity provider. The Pythia reputation framework externalizes OpenPrivacy's concept of a nym service to external digital identity providers. Using existing digital identity solutions narrows the focus of the Pythia and extends its influence beyond a single digital identity implementation.

A collection of opinions in Sierra is called a bias. Sierra's opinion objects are roughly comparable to the transactions in Pythia. "OpenPrivacy's reputation management system can assemble a set of related opinions into a bias. ... Often, a bias may consist of Opinions from multiple nyms." [Lab01b] Biases are paralleled by rule sets in the Pythia system.

The Reputation Calculation Engine (RCE) component of Sierra amalgamates local opinions and biases to produce its own opinions (i.e. reputation responses). RCE's in Sierra are objects that have bindings to particular opinions and biases which they use to produce their opinions. Pythia's RCE is a specialized runtime environment in which rule sets are executed. Transactions submitted as feedback are Pythia's input data and the reputations Pythia produces are its output.

In *A Reputation and Trust Management Broker Framework for Web Applications*, Lin *e. at.* affirm that trust should be based on verifiable identification of the subject entity, qualification of the entity to perform the requested service, and the subject entity's consistency of performance. [Lin05]

In their system reputation is a consistency of performance metric based on reports from a service's peers. A service's reputation thus becomes a component of its customers trust calculation. In the framework design presented by Kwei-Jay Lin's team, a reputation broker first attempts to answer a users reputation requests based on the broker's local reputation database. If a broker lacks sufficient local feedback to make a recommendation it contacts peer brokers for reputation information. If peer brokers lack sufficient feedback to return a reputation, a broker can contact institutional reputation authorities before resulting to untrusted sources. Kwei-Jay Lin's hierarchy of diminishing trust is complex.

user -> broker -> peer brokers -> reputation authority

Pythia's model is simpler, having only users and relying parties both interacting with a reputation authority. The Pythia server answers reputation requests based only on the feedback that it has collected. Pythia's collected feedback may be thousands of transactions, ten or twenty transactions, or none at all. Examining Kwei-Jay Lin's work leads to two conclusions:

1. Reputation response should have an accompanying audit trail of external brokers and services used to calculate the reputation.
2. Confidence measures such as the quantity of feedback transactions used during a reputation calculation should also be part of a reputation framework response.

Future Work

The framework we have described is still under development. While most of the key ideas are represented in the system, there is much to be done to complete our vision for them.

One part of our design that has yet to be implemented is allowing users to create ratings and endorsements of other users. For this to be useful some way of recording interuser trust is necessary to measure the confidence that should be placed in a particular rating or endorsement. Otherwise, there's no way for the system to know whose ratings to use and whose to ignore.

Interuser trust can be defined explicitly by users, but might also be calculated or otherwise inferred. One way to infer interuser trust is by tapping into social cues that are already on the Web. For example, allowing users to claim OPML files from a site like Bloglines would give an indication of the URLs of other users whose blogs they read. FOAF data or other data from social networking sites could also be used.

Another avenue for future research is using the system as an identifier exchange service. The system already validates claims to identifiers and associates those identifiers together. Pythia could authoritatively state that a particular URL is controlled by the same person who controls a particular email address, for example. Identifier exchange services are useful in aggregating Web services where a client supplies one kind of identifier, but the server requires a different kind.

Conclusion

The emergence of identity systems such as OpenID, LID, and i-names that are independent of any particular site, provides an opportunity to build reputation systems that are based on those identifiers and have broad application. This paper has described such a system that also employs a rules engine and plug-in architecture to make it malleable to a variety of tasks.

The flexibility of our framework makes it a good platform for further studies in reputation since the effect of various policies in various settings can be more easily explored than when the entire system must be built from scratch.

Acknowledgments

This work grew out of a class project at Brigham Young University in the Winter of 2006. Class participants were Devlin Daley, Jared Laney, Harshwardhan Nagaonkar, Roger Pack, Sterling Porter, Jason Read, Michael Reynolds, Nathan Stocks, Kevin Tew, and Terry Wilcox.

This paper was written in HTML and rendered as PDF using Prince XML.

References

[Lab01a] Labalme, Fen and Burton, Kevin *OpenPrivacy.org* , http://www.openprivacy.org/

[Lab01b] Labalme, Fen and Burton, Kevin *Enhancing the Internet with Reputations* , An OpenPrivacy white paper (March 2001) Version: 0.7 http://www.openprivacy.org/papers/200103-white.html

[Lin05] Lin, Kwei-Jay, Lu, Haiyin, Yu, Tao, and Tai, Chia-en, *A Reputation and Trust Management Broker Framework for Web Applications.* , 2005 IEEE International Conference on e-Technology, e-Commerce, and e-Services (EEE 2005) 29 March - 1 April 2005, Hong Kong, China Pages 262-269 http://doi.ieeecomputersociety.org/10.1109/EEE.2005.14

[Mic06] Miller, Jeremie. *MicroID - Small Decentralized Verifiable Identity*, http://microid.org/

[Res00a] Resnick, Paul, Kuwabara, Ko, Zeckhauser, Richard, and Friedman, Eric, *Reputation Systems* , Communications of the ACM, Volume 43 , Issue 12 (December 2000), ppg. 45-48.

[Wind06a] Windley, Phillip J., *Principles of Reputation*, Phil Windley's Technometria http://www.windley.com/archives/2006/06/principles_of_r