Portable Reputations with EgoSphere

Keith Bonawitz Chaitra Chandrasekhar Rui Viana
bonawitz@mit.edu chaitra@mit.edu ruilov@mit.edu

ABSTRACT

Many online services require some form of trust between users – trust that a seller will deliver goods as advertised, trust that an author's thoughts are worth the time spent on reading them. To accommodate an internet community where users are constantly interacting with strangers, online services often construct proprietary reputation management systems for their community, with the side effect of locking users into that service if they wish to maintain their reputation. In contrast, this paper outlines EgoSphere, a system for *portable Internet reputations*, so that reputations built on one service can be used elsewhere. EgoSphere hinges on the use of correlational statistics to automatically project reputations from one service to other similar services. To achieve these goals, EgoSphere must gather webservices' reputation data. EgoSphere avoids the unreasonable expectation that all webservices will publish their reputation databases, while also avoiding the use of a webcrawling robot (a violation of many webservices' robots.txt restrictions and source of incurring additional website load) by gathering its reputation data using a *distributed passive robot* system. This system simulates the function of a standard web-crawling robot by using webproxies on users' computers to analyze the responses to standard webservice requests. This paper outlines the design and proof-of-concept implementation of EgoSphere, targeted specifically at providing portable reputations for bulletin-board style internet services.

1. INTRODUCTION

Reputation Systems

With over 900 million people [COM] interacting on the web, Internet users are regularly finding themselves in situations where they must choose to trust strangers – trust them to faithfully complete a commercial transaction or to provide information that is worth the time to read.

To facilitate trust among strangers, reputation systems have been successfully applied in various settings on the Internet. These systems provide summaries of users' pasts, with the rationale that a user who has acted trustworthily in the past is likely to continue to do so. Such summaries must be concise – if a user's reputation summary is too lengthy, reading it will cost more than the informational gain it provides. In many cases, these concise summaries take the form of a single number, with the benefit that software can also use these numerical summaries to automatically organize and filter information.

The most notable reputation system currently deployed is Google's PageRank algorithm for computing the expected relevance of a hit from a web search (e.g., how much the user should trust the hit to provide useful information). High PageRank indicates high expected relevance, and results from the page having a large number of other pages linking to it. The implicit assumption is that links to a webpage are evidence that someone finds that webpage relevant.

Amazon zShops and eBay, two of the most successful online marketplaces, deploy typical reputation systems for sites based on commercial transactions. When users complete transactions, the system allows the users to rate each other how smoothly the transaction was executed. Most of this information is made public, so that anyone can easily understand the past performance of a specific user. In order to demonstrate the relevance of reputation systems, some research has already been made [DEL01] on how e-commerce reputation services can affect the online markets for the goods offered through these web-sites. Intuitively, if a user holds a higher reputation rank, he or she is able to sell products for a higher price.

Other services, such as Slashdot or infoAnarchy, use reputations to assign trust to certain pieces of information. At Slashdot, users can rate the comments posted by other users. Although Slashdot only publishes per-comment rating summaries rather than per-user rating summaries, it internally uses per-user summaries (called "Karma" in Slashdot terms) to make highly-reputed users' comments more visible by synthetically boosting their per-comment reputation scores. This in turn results in earlier placement of the comment on the web page. Although not published, a user's "Karma" can be inferred by reviewing a history of that user's per-comment ratings.

The current design of EgoSphere focuses on information sharing websites like Slashdot. It is important to make clear what an overall reputation value would reflect on such sites. If a user receives good feedback for one of his comments on the *Apache* section of Slashdot, it could mean that the user is a skilled writer on technical issues or s/he has a great knowledge about the Apache server. It is our desire that the reputation value reported by EgoSphere will account for both of these components.

The Portable Reputation Vision

A critical shortcoming of current reputation services is that they are generally bound to a specific website. A user that has built a good reputation at Slashdot is not able to take advantage of that reputation at infoAnarchy, even though these two websites offer highly similar services.

Amazon, for example, realized the advantages of portable reputations early and it used to allow its users to import their eBay ratings. However, it did not take long for eBay to complain about this scheme claiming that its reputation algorithms were proprietary [RES00][1].

[1] Perhaps the true motivation was that eBay was getting nothing in return for assisting their customers in switching to Amazon; since EgoSphere shares reputations symmetrically, we hope the perceived cost will be minimized.

EgoSphere proposes to integrate different reputations services into one global system facilitating the transfer of reputations between services. Since EgoSphere gathers data from feedback provided from many different web-sites, it has more information available to its algorithms than any single service does. Thus, EgoSphere can compute reputation rankings that are more informed than the ones computed by any single service. Naturally, the values provided by each service are still available to the user, but the ability to gather information from other websites allows EgoSphere to provide a more complete reputation profile. EgoSphere particularly shines in its ability to "fill in" reputations when users are new to a service, until the user can build up a local reputation.

Isolated reputation systems leave no option to the user other than to perform most of his or her transactions at the dominant websites. A good reputation built at a smaller online service is not nearly as valuable as one built at the larger websites. Hence, it is extremely difficult for smaller or new services to compete against the larger ones. However, if the reputations of a small service constantly agree with those of a larger service, then there seems to be no reason why the users of the larger website should not trust the reputation values from the smaller as indicators of trust

Thus, portable reputations not only allow for more informed reputation reports, but it also facilitates competition between services, which should have positive impact on the quality of the services offered.

EgoSphere goals

There are five major goals that EgoSphere intends to achieve.

First and foremost, EgoSphere should provide portable reputations as described above. A user that has good reputation at Slashdot should be able to open a new infoAnarchy account and instantaneously enjoy his old reputation with his new account. Ideally, any transaction performed by a user at any service should have an impact on his or her reputations everywhere else.

Second, EgoSphere should use the diverse reputation evidence it gathers to provide more informed reputation rankings than any single service is able to. Note that since EgoSphere is a global system, it must be able to differentiate between the types of reputation a user can build up.

Third, EgoSphere should not require the help of online services to do its own job. It would be much easier to design EgoSphere if the major web-sites were willing to cooperate by offering complete reputation data to EgoSphere. However, as explained before, given that EgoSphere should facilitate competition against these web-sites, it is unlikely that any help will be provided.

Fourth, users should not have to perform any complex tasks in order to prove to EgoSphere that they own the accounts and usernames that they claim they own. A user will probably want to register a large number of accounts with EgoSphere, and if registration consists of a complicated procedure it is unlikely that users will adopt the system.

Finally, EgoSphere should provide enough information about a user's past so that other users can make decisions about whether to trust this user, but no more information than necessary. The concern is that the diversity of reputation information EgoSphere maintains about its users may become a privacy concern. In particular, EgoSphere should avoid publishing cross-webservice user correspondences.

2. DESIGN

Criteria

A successful design for EgoSphere must perform the following functions:

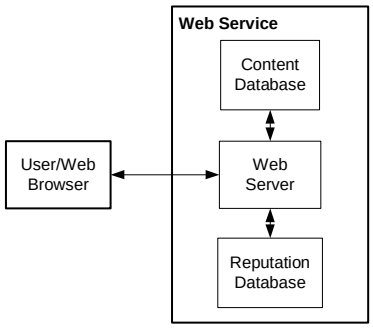

Figure 1: Traditional web services only expose their reputation database indirectly by servicing HTML requests.

1. Obtain raw reputation evidence from webservices, and compute per-user numerical reputation summaries if these are not provided
2. Identify which usernames on different webservices correspond to the same user
3. Estimate to what extent reputations on site X are relevant to reputations on site Y
4. Combine reputations that originate from multiple sites
5. Present the results to the user

Rationale

Existing web services typically incorporate reputation as shown in Figure 1. Users of the website use their browser to submit requests for content to the web server. The web server then queries a content database and a reputation database, using the database responses to assemble the HTML document and return it to the user's browser. The user can also submit requests for the web server to update the reputation database based on the user's opinion of reputation-bearing content.

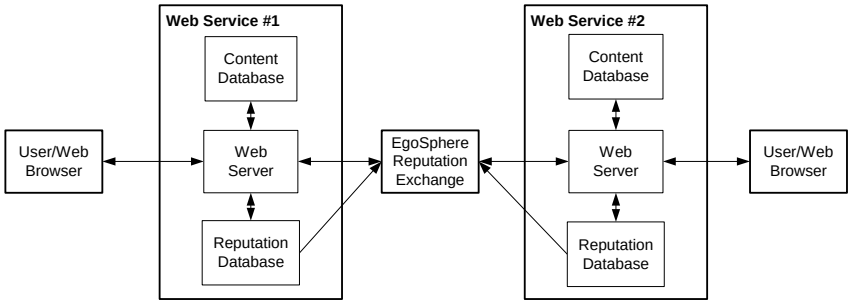

Figure 2: Web services could cooperate with EgoSphere when assembling HTML responses to user queries, as well as updating EgoSphere with modifications to the reputation database.

Given this architecture, the most direct design for EgoSphere is shown in Figure 2. In this design, web services cooperate with EgoSphere. When the web server for a web-site handles a user's request, it requests reputation transfer information from the EgoSphere Reputation Exchange, and uses this information when it assembles its HTML response. The web service also is responsible for updating EgoSphere when there are changes to its reputation database.

Unfortunately, this design has a very high barrier to entry. We would have to convince many websites to spend money and resources to integrate EgoSphere into their core services. This is impractical, both for this project's time limitations and in general, due to the fact that web services are not likely to cooperate and because EgoSphere only shows its true potential once many web services are participating.

Design Overview

To eliminate the entry barrier, we opt for a *zero impact* approach that supports *incremental adoption* from web services, allowing EgoSphere to function and grow to a critical mass of supported web sites without any site modifying its code or expending any additional resources. In order to achieve the incremental adoption goal, EgoSphere will only be able to interact with existing web services through the standard web server interface they supply – issuing web requests and interpreting the HTML responses. This suggests a robot style design, similar to the webcrawler robots used by search engines to index the web. However, in order to comply with our zero impact design goal, as well as with the robot restrictions enacted by many of our target web-sites, we cannot simply have a robot issue a large number of queries to the web server. Instead, we design EgoSphere to use a *distributed passive robot* scheme, which simulates the data a robot would gather, without actually crawling the website. Instead, the distributed passive robot observes requests made by users in their normal interaction with the website, and extracts and collates from the site's responses the information that would have been gathered by a traditional robot. In EgoSphere, the distributed passive robot scheme is implemented by the *Webproxy* and the *Reputation Database*, as described below. Figure 3 highlights how this design simulates the presence of the hypothetical access channels shown in Figure 2 using only the access channels actual available in Figure 1.

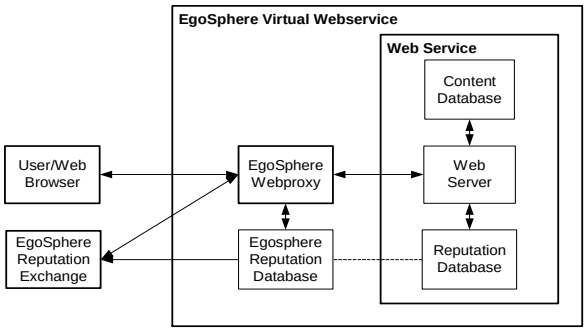

Figure 3: EgoSphere uses a webproxy on the client-side to annotate HTML responses from the web server with EgoSphere information, and to gather reputation evidence.

The EgoSphere Webproxy

The *EgoSphere Webproxy* serves as the system's eyes, ears, and mouth. Every EgoSphere user runs an instance of the *EgoSphere Webproxy* locally on their computer, and configures their web browser to use the webproxy for all requests. Whenever the user requests a webpage from an EgoSphere-supported service, the webproxy first fetches the webpage from the web server. It then analyzes the HTML, searching for EgoSphere annotatable content, such as usernames. The webproxy requests annotations for those usernames from the EgoSphere Reputation Exchange, and inserts the annotations into the HTML at the appropriate places (i.e., beside the corresponding username), before returning the annotated HTML document to the user's browser.

In addition, the webproxy also analyzes the HTML it got from the web server for reputation evidence. This evidence may take different forms on different sites: for example, on Slashdot, each comment is accompanied by a rating given to that comment by other users on the site. The webproxy sends this information to the Reputation Database for this web service.

The EgoSphere Reputation Database

The *EgoSphere Reputation Database* collates the evidence gathered from many users' webproxies into a unified view of the reputations of users on a system. Together with the webproxy, these two EgoSphere components form the *distributed passive robot* subsystem.

The EgoSphere Reputation Database will receive reputation evidence from many webproxy sources, each of which will have an incomplete view of the website users' reputations. For example, most pages a user requests from the website will not have reputation evidence about all users of the system: on Slashdot, reputation evidence is only provided for those users which have commented on the current article. Furthermore, the evidence may evolve over time: as a Slashdot comment gathers more ratings, its rating total will change. Finally, a single evidence view may not be sufficient to determine the user's reputation: at Slashdot, a user's reputation would be more adequately described by the average rating of all their comments, then by the rating assigned to just one of their comments. The EgoSphere Reputation Database is responsible for receiving and managing reputation evidence in order to compute a reputation estimate and a reputation uncertainty factor for each user (e.g., the more reputation evidence has been gathered, the more certain the reputation estimate is), as well as for updating the EgoSphere Reputation Exchange with this information.

The EgoSphere Reputation Exchange

The *EgoSphere Reputation Exchange* is responsible for storing the reputation of each EgoSphere user on each web service, and for computing how much reputation should transfer from one service to another. The Exchange tackles the problem of transferring reputation using a simple linear regression model with correlation analysis. Regression data between services is periodically calculated and stored in a database. Services with correlation values exceeding a threshold level of certainty are considered to be correlated web services. The regression estimate is used to calculate the transferred reputation. This is sent to the webproxy, which displays it in a format suitable for the user. For privacy reasons, *Exchange* will return the transferred reputation from another service (with the service name) but not the actual username on that service.

Specifically, when the Reputation Exchange is asked to transfer the reputation of user *X* from website A to website B, it forms a vector U of all the users with reputations on *both* of these sites, as well as vectors A(U) and B(U) containing the reputation estimates of those users on each site. By computing the correlation between A(U) and B(U), EgoSphere can estimate how well reputation transfers between domain A and domain B. Finally, the Reputation Exchange can compute a regression for the A(U) and B(U) vectors; using this regression, it can predict B(*X*) given A(*X*).

Both the vectors A(U) and B(U) are subject to measurement error so the kind of regression used is with consideration that there are errors in both estimates. Let x=A(*X*) be the independent variable which can used to predict y=B(*X*), the dependent variable. Egosphere uses a simple linear regression model using the least squared error method

$$Y = aX + b$$

The fitted line is obtained by minimizing the sum of squared residuals; i.e. finding *a* and *b* so that $(Y_1 - a - bX_1)^2 + \ldots. (Y_n - a - bX_n)^2$ is as small as possible. The transfer of reputations is done using the *a* and *b* values. Y is the predicted reputation of the user in Service A given his reputation X in service B.

To test for the strength of the correlation, the Pearson correlation coefficient r is used. The formula for r is:

$$r = \frac{n \sum_{i=1}^{n} x_{ik} x_{im} - \left(\sum_{i=1}^{n} x_{ik}\right)\left(\sum_{i=1}^{n} x_{im}\right)}{\sqrt{\left[n \sum_{i=1}^{n} x_{ik}^2 - \left(\sum_{i=1}^{n} x_{ik}\right)^2\right]\left[n \sum_{i=1}^{n} x_{im}^2 - \left(\sum_{i=1}^{n} x_{im}\right)^2\right]}}$$

Correlation is perfect when $r = \pm 1$, strong when r is greater than 0.8 in size and weak when r is less than 0.5 in size.

The Pearson r measures precision of the relationship and not accuracy. The r-squared

value (square of Pearson r) gives an estimate of the gain in accuracy between using the model and just guessing. So a value of r=0.5 implies a 25% gain in accuracy due to the model. The accuracy of the prediction model is determined using the standard error of the estimate. This is measured using the Pearson chi-square test. The formula for X^2 is

$$\chi^2(a,b) = \sum_{i=1}^{N} \frac{(y_i - a - bx_i)^2}{\sigma_{y\,i}^2 + b^2\sigma_{x\,i}^2}$$

where $\sigma_{x\,i}$ and $\sigma_{y\,i}$ are, respectively, the x and y standard deviations for the ith point.

The square root of the chi-square allows for evaluation of the goodness of fit. It gives the estimated error in the Y values. EgoSphere in particular tolerates an error of up to 1. Since the reputation is always a small integral value, this tolerance is rational.

Reputation Contexts

In the introduction to this paper, we observed that a reputation at one service may have several components (such as raw writing skill and expert knowledge about a particular domain) and that only some of those components may be transferable to any given target webservice. Seemingly at odds to this, we have just described reputation correlation and regression calculations which group all users from a particular service into a single reputation context. We reconcile these positions by considering the (implicit) set of reputation contexts C_1, C_2, … representing the potentially transferable components of a reputation (eg, C_1="writing skill", C_2="expert on Apache", etc). The reputation context on the Apache section of Slashdot can then be thought of as a compound context, for example $C_{slashdot.apache} = C_1 + C_2 + C_{65}$. Our correlational statistics will uncover some predictive power between, say, $C_{service\ X} = C_1 + C_3$ due to the shared C_1 component context. However, the correlation will be much stronger for $C_{service\ Y} = C_1 + C_2$, since more the $C_{service\ Y}$ is shared with $C_{slashdot.apache}$. Thus, by favoring highly correlated services, we implicitly access these context components.

Merging Transferred Reputations

Egosphere transfers reputation information from multiple source webservices individually using the regression model. In order to compute a single most-informed estimate of reputation at the target service, EgoSphere must merge these estimates into a single overall value. Each estimate is accompanied by an uncertainty value, based on:

1. The uncertainty of the reputation summary at the source webservice
2. The uncertainty of the regression-transfer.

EgoSphere computes the merged reputation using a weighted average of the estimates from all sources, where the weights are inversely related to the uncertainty values.

Solving The User Correspondence Problem

A critical component of EgoSphere's Reputation Exchange process is determining the correspondence between users on site A and users on site B. A straightforward approach would be to require the use of identical IDs across websites, which could be in the form of a username, e-mail address, etc. Unfortunately, usernames are unreliable since a majority of the users do not use the same name across services. Also, email addresses are usually obfuscated in most services and Egosphere (in the current version) will not be privy to this information. Even if EgoSphere could expect identical IDs, it would still need to confirm their authenticity.

To solve this ID correspondence problem, EgoSphere requests that users demonstrate control of particular usernames. Users log onto the EgoSphere website and claim to be a particular user on a particular webservice. EgoSphere will then ask them to post a specific random verification string in a user-controlled portion of the web service – for example, in a comment on Slashdot. When an EgoSphere webproxy observes a verification string, it notifies EgoSphere that a particular string appeared in a particular user-controlled area. If the string was posted by the correct user, then the EgoSphere user is verified to have control of that webservice account. This should work well for the bulletin-board style reputation problem that we are addressing.

Solving this problem also helps us deal with malicious attacks. Either a service could be malicious and trick EgoSphere into believing it has a high correlation with a trusted service or a set of users could try to launch a similar

attack. A malicious service (X) could manipulate Egosphere into thinking that X correlates well with a trusted service only if it could have a lot of common users between the two services. It does not have access to the Egosphere IDs of most of the users on the trusted service so it would have to set up fake Egosphere accounts that make the webservice correlate highly. Even if this happens, the users on service X would not have high reputations on the trusted service and hence would not benefit by correlating with a trusted service. Similarly, a set of malicious users cannot benefit by using this attack.

Managing Load

Each of the logical services (the Egosphere Webproxy, Reputation Database, and Reputation Exchange) are implemented as independent servers. In the anticipated use of the EgoSphere system, every user would run his own Webproxy, so this computational effort is essentially free. This scheme also affords the user privacy: no centralized service monitors the user's complete web usage, which might include privacy oriented sites such as online banking or webmail; this information never leaves the users' own machine. A separate Reputation Database can be run in correspondence with each webservice whose reputations are being accumulated. Because the Database is only charged with taking a significant sampling of the reputation data for its associated website, it may freely choose to ignore reputation evidence messages from users' Webproxies if the Database is getting an overload of messages. Finally, several instances of the Exchange server can be run: one of them as the master which accepts updates from the various Databases and runs the statistical computations, the rest as slaves which are updated by the master. The slave Exchange servers can service a large number of client Webproxies through a load-balancing configuration. Because it is not critical that clients see completely up-to-date information, load on the Exchange master can be managed by having Databases only periodically update the Exchange master with new information, and by allowing the Exchange master to compute the statistics offline before passing the results on to the Exchange slaves.

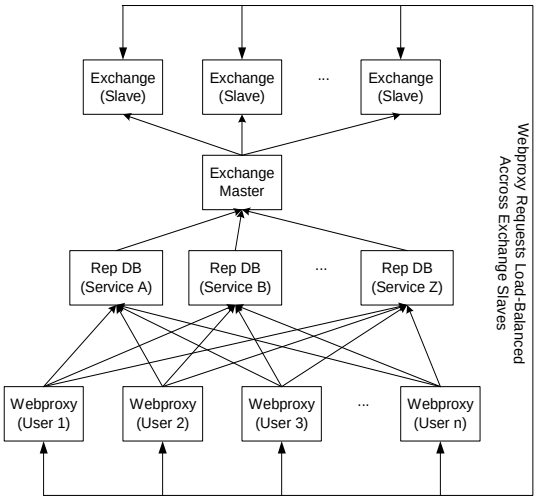

Figure 4: Managing request load on a production-level Egosphere system.

3. IMPLEMENTATION

Overview

For our proof-of-concept implementation, we built each module on top of the libasync framework [MAZ, LIB]. Interservice communication is implemented with Sun's RPC/XDR specification [SUNa, SUNb]. The RPC client and server implementation is built using the `rpcc` utility, modeling off of the sample system available at [6824]. RPC is a stronger commitment than necessary for certain communication channels – for example, it is acceptable for calls from a Webproxy to a Reputation Database to go unacknowledged and even to be lost under heavy load. To keep our implementation straightforward, however, we have used RPC for all communication and reserve performance-tuning of the communication protocol as a future enhancement.

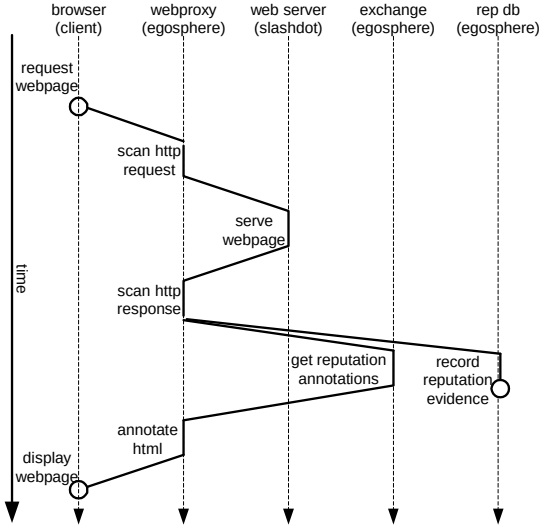

Figure 5: The flow of control as Egosphere services a client's request.

Webproxy

The Egosphere Webproxy is a straightforward web proxy implementation. It determines whether each http request it processes is intended for an EgoSphere-supported webservice by examining the requested URL. For such requests, the Webproxy requests the document from the webserver and buffers the server's response. The response HTML is then scraped using a set of regular expressions to extract information such as usernames, webservice supplied reputation evidence, and byte-offsets into the response where Egosphere annotations should be inserted. Note that every webservice that is supported by EgoSphere requires the webproxy to be given a customized set of regular expressions that are tailored to scrape the appropriate content from that site's pages.

The Webproxy sends the Exchange server the list of usernames found along with the name of the webservice. The Exchange server replies with a list of user-annotations, each of which is inserted at the proper location in the buffered HTML before the Webproxy returns the annotated response to the client. While the Webproxy is waiting for a response from the Exchange server, it also reports the reputation evidence it gathered to the Egosphere Reputation database responsible for the current webservice. If any Egosphere account verification codes are detected, these are reported as well. Verification codes have an easily detectible pattern (eg, "#ego20329"), so a simple regular expression is sufficient for detection.

The annotations returned by the Exchange server for each user are sets of tuples of the form (W, r_W, u_W), indicating that the Exchange server used evidence from webservice W to estimate a reputation r_W for this request's webservice, and that this estimate is uncertain within the range $r_W \pm u_W$. The Webproxy reports this information directly, but for the user's convenience, it first reports a merged reputation r_m computed using evidence from all webservices and weighted by uncertainty:

$$r_m = 1 / u_{sum} * SUM_w \; [(1 / u_w) * r_w]$$
$$\text{where } u_{sum} = SUM_w \; (1 / u_w)$$

Following [POL] and ignoring uncertainty about the uncertainty estimates themselves, we can estimate u_m, the range of uncertainty of around r_m:

$$u_m = 1 / u_{sum} * SQRT(SUM_w \; 1)$$

With this information, a rendered annotation looks something like:

$$(r_m \pm u_m \parallel W_1: r_W \pm u_W \parallel W_2: r_W \pm u_W)$$

Accumulator (EgoSphere Reputation DB)

The accumulator serves two main functions in the EgoSphere system:

1) It acts as the central repository of the distributed passive robot. The Accumulator receives all the data obtained by the webproxies, and must figure out what to do with this data.
2) It computes reputation and uncertainty values for each *(service, user)* pair and sends these values to the exchange server.

The first of three major issues in implementing these functions is that the Accumulator must deal with a large amount of information. Since every user must run a proxy, and every proxy is constantly sending data to the Accumulator, the Accumulator can easily get overloaded depending on the number of users of EgoSphere. The existing Accumulator does not use any special methods to deal with the large amount of data coming in. It simply processes one request at a time. As we expect very few people to be running webproxies in this implementation, this solution is satisfactory. However, in a full system, the workload will probably be disk intensive as the Accumulator must keep a large database of users and comments. Thus, an asynchronous server would probably be best suited for this job.

Second, malicious users will certainly be able to build fake webproxies that attempt to corrupt the reputation values reported to EgoSphere. Note, however, that the data sent by valid webproxies do not have to agree all the time, as for example, if the reputation of a user actually changes. Thus, the Accumulator must be able to differentiate between valid but conflicting information and corrupted data that conflict with legitimate data. The current implementation uses the value with the latest *time* tag. A more complete implementation

would keep a count of how many times it has seen each value and at what times, and only believe that a reputation value has changed after enough webproxies have reported it.

Finally, for each user in each service, the Accumulator receives feedback about many of the user's comments. It must then calculate unique reputation and uncertainty values for that user and send these to the Exchange server. However, this sort of computation is likely to be expensive when performed for a large base of users, and so the Accumulator must be able to do it at convenient times. Again, the current implementation takes the simple approach. It computes the average of all of one user's ratings as a piece of information about that user comes in, and it uses this number as the reputation value. The standard deviation, representing uncertainty, is also computed on-the-fly. The complete implementation would probably log the data it receives when it is under heavy load, and only actually compute those values when it can.

Exchange

The Exchange consists of one database with three related components

1. Services database: For each webservice, this database maps local userids on that service to other information about the user including EgoSphere ID
2. User database: Maps EgoSphere IDs to a list of (webservice, local userid, valcode) tuples which gives the userid of the EgoSphere user and the status of validation (whether the EgoSphere user can be trusted to have that id.
3. Correlational database: Maps pairs of webservices to [correlation measure, a, b] tuples (where a and b are the coefficients of the least-square minimization line).

These databases are stored dynamically as hash structures and statically in a directory structure. The regression data is calculated and stored in the correlation database. The correlation value is measured in a range from +1 to –1. The closer to +1 or -1, the stronger the relationship. The closer to zero, the weaker the relationship. The other two values that are stored in each tuple of the correlation database are used to calculate the transferred reputation. a and b are the coefficients in the line of best fit which gives the prediction of the reputation.

The Exchange responds to three main queries

i. setReputations(*W, Accumulator Reputation Database*): Updates the reputations of users in webservice W with the details given in the Database
ii. getAnnotations(*W, L*): Returns a list of Annotations A which has the mappings of users specified in the list L (who are part of webservice W) to a list of annotation tuples (*SourceWebservice, Estimate, Uncertainty*). SourceWebservices is a subset of those that the user is a member of and there is a high correlation between W and these SourceWebservices,
iii. observeValidationCode(*W, U, ValCode*): Updates the User Database and sets the validation code (ValCode) of user U in webservice W.

4. RESULTS

The results of this project include a proof-of-concept implementation of EgoSphere in 5000 lines of C++ code and XDR specifications. Figures 6 and 7 demonstrate a user's experience while viewing Slashdot without and with EgoSphere reputations, respectively.

5. FUTURE WORK

The EgoSphere framework serves as the basis for future research. First, experiments should be conducted investigating how well EgoSphere's division of labor allows the system to cope with large numbers of users, and tailoring the communication protocols to work around bottlenecks. A first step would be to move away from RPC for messages which do not require acknowledgements. A more complicated system may be needed if clients are putting too much load on a popular Reputation Database; it may be necessary to coordinate the load across various database replicas and/or implement a feedback channel which allows the Reputation Database to throttle Webproxy input.

Future research should also investigate more robust algorithms for inferring reputation values from the information supplied to the Reputation Database. For example, one could

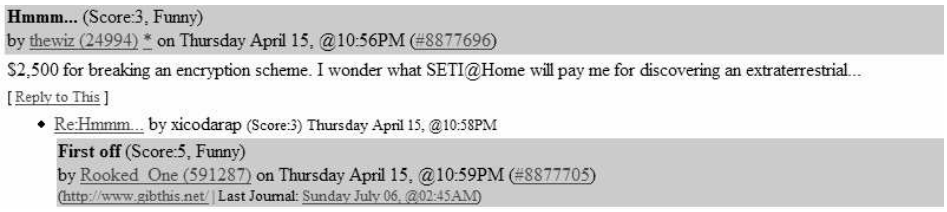

Figure 6: Slashdot without EgoSphere annotations

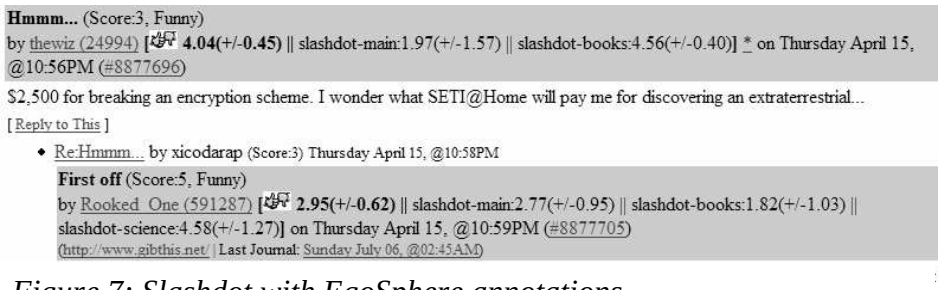

Figure 7: Slashdot with EgoSphere annotations

imagine modeling the inference as a partially observable Markov decision process, where hidden variables include the actual per-comment rating and whether or not the reporter is faulty/malicious. The per-comment rating would be modeled as having a small probability of change between observations. Reputation observations would be based on both hidden variables, such that faulty/malicious users report random values. Standard statistical inference techniques should then be able to estimate the most likely values of these hidden variables at each point in time.

Finally, alternate methods of regression should be pursued. Our model of reputation exchange uses a relatively simple linear regression, but a more sophisticated model may be more accurate.

REFERENCES

[6824] "6.824 Airline reservation rpcc demo." `http://pdos.lcs.mit.edu/6.824-2002/lecnotes/{rx.x, rs.C, rc.C}`.

[COM] Computer Industry Almanac. `http://www.clickz.com/stats/big_picture/geographics/article.php/5911_151151`

[DEL01] Chrysanthos Dellarocas. "Analyzing the Economic Efficiency of eBay-like Online Reputation Reporting Mechanisms". MIT Sloan Working Paper No. 4181-01. October 2001.

[DEL03] Crysanthos Dellarocas and Paul Resnick. "Online Reputation Mechanisms: A Roadmap for Future Research." First Interdisciplinary Symposium on Online Reputation Mechanisms, 2003. `http://www.si.umich.edu/~presnick/reputation/symposium/ReportDraft1.doc`

[LAB01] Fen Labalme and Kevin Burton. "Enhancing the Internet with Reputations: An OpenPrivacy white paper." 2001. `http://www.openprivacy.org/papers/200103-white.html`

[LIB] "Libasync tutorial." `http://www.pdos.lcs.mit.edu/6.824/async/index.html`

[MAZ] David Mazieres, Frank Dabek, Eric Peterson, and Thomer Gil. "Using libasync." `http://pdos.lcs.mit.edu/6.824/doc/libasync.ps`

[POL] Polson, James. "Uncertainty Propagation and Linear Least-Squares Fitting" `http://www.upei.ca/~physics/polson/course/P211/lab/error.pdf`

[RES00] Paul Resnick, Richard Zeckhauser, Eric Friedman, and Ko Kuwabara. "Reputation Systems: Facilitating Trust in Internet Interactions." *Communications of the ACM*, *43*(12), December 2000, pages 45-48.

[SUNa] Sun Microsystems. "ONC+ Developer's Guide: RPC Protocol and Language Specification." 2002. `http://docs.sun.com/db/doc/816-1435/6m7rrfn9f?a=view`

[SUNb] Sun Microsystems. "ONC+ Developer's Guide: XDR Specification." 2002. `http://docs.sun.com/db/doc/816-1435/6m7rrfn9l?a=view`