Design and Implementation of Davis Social Links OSN Kernel

Thomas Tran, Kelcey Chan, Shaozhi Ye, Prantik Bhattacharyya, Ankush Garg, Xiaoming Lu, and S. Felix Wu

Department of Computer Science
University of California, Davis
{ttran,kchan,sye,pbhattacharyya,garg,lu,sfwu}@ucdavis.edu

Abstract. Social network popularity continues to rise as they broaden out to more users. Hidden away within these social networks is a valuable set of data that outlines everyone's relationships. Networks have created APIs such as the Facebook Development Platform and OpenSocial that allow developers to create applications that can leverage user information. However, at the current stage, the social network support for these new applications is fairly limited in its functionality. Most, if not all, of the existing internet applications such as email, BitTorrent, and Skype cannot benefit from the valuable social network among their own users. In this paper, we present an architecture that couples two different communication layers together: the end2end communication layer and the social context layer, under the Davis Social Links (DSL) project. Our proposed architecture attempts to preserve the original application semantics (i.e., we can use Thunderbird or Outlook, unmodified, to read our SMTP emails) and provides the communicating parties (email sender and receivers) a social context for control and management. For instance, the receiver can set trust policy rules based on the social context between the pair, to determine how a particular email in question should be prioritized for delivery to the SMTP layer. Furthermore, as our architecture includes two coupling layers, it is then possible, as an option, to shift some of the services from the original applications into the social context layer. In the context of email, for example, our architecture allows users to choose operations, such as reply, reply-all, and forward, to be realized in either the application layer or the social network layer. And, the realization of these operations under the social network layer offers powerful features unavailable in the original applications. To validate our coupling architecture, we have implemented a DSL kernel prototype as a Facebook application called CyrusDSL (currently about 40 local users) and a simple communication application combined into the DSL kernel but is unaware of Facebook's API.

1 Introduction and Motivation

The rising growth in popularity of online social networks (OSNs) has been phenomenal in the last few years. Millions of people connect with one another and maintain friendships using the available OSNs. As a result, there is a plethora of rich and interesting user data spread across the networks. The user data is not limited to professional and personal information. It also contains status updates, wall posts (e.g. in Facebook), scraps (e.g. in Orkut), location information, etc. With the growth of OSNs, API's to let developers access this data and build applications have been created.

The scope of applications include social games (e.g. Lexulous, various trivia quizzes), displaying one's musical tastes, etc. However, under our architecture, we can greatly expand the set of applications using social networks by providing digested information gathered and calculated from these OSNs that we believe have never been utilized before. Software classes such as email, search engines, and online telephony can be easily modified to gain the advantages that come the social knowledge in these networks. Furthermore, we present methods with which applications can easily utilize a robust system of trust and reputation inherent to Davis Social Links (DSL) [1].

The graph set present in OSNs due to the interconnection of users is a reflection of the social human network in the digital format. The hypothesis that social networks are small world networks (with property of small diameters) such that everyone can connect to everyone else using a short path length motivates us to exploit the presence of this rich user information set to build communication protocols based on trust and reputation of the users and are robust and secure in nature. In this paper, we present an architecture that attempts to leverage the rich user set represented in the form of social graph to build communication protocols. Our architecture thus attempts to bring the 'social context' in message exchange. The system uses the available OSN API's to build the social graph and facilitate the introduction of social context in the communication layer. So far, a friendship, from the OSN's perspective, has been binary. Two users either are friends, or they are not. However, we hope to present a better model of friendship by realizing that not all friendships are equal. This distinguishes DSL from a normal online social network. It is common for users to trust some friends better than others. Therefore, our DSL architecture captures, analyzes, and presents this information using a robust and easy to use API.

The DSL kernel architecture we have designed and implemented gives standard applications the ability to leverage social network data without requiring the user of the application to access his or her social network site. The only extra step the user needs to handle is authentication to the network. Our architecture also attempts to require little to no modification on pre-existing applications that wish to leverage social network data. Another feature of our architecture is that the application does not break when a social network changes its web layout which is a problem when having a bot crawl web pages for data.

To establish the effectiveness of the architecture, we are currently building an application which uses a Thunderbird or Outlook Express client to send an email using a social path. The social path is computed using the connectivity information imported from an OSN (we currently use Facebook). In this paper, we will delve into how our architecture builds off of DSL to allow software developers unprecedented access to the rich social graphs to add an element of trust and reputation in user-to-user interactions. Furthermore, we will explain how our robust and flexible design allows developers to modify their existing applications with very little effort. By giving some background on DSL in section 2, we can present our architecture in section 3. Next, in section 4, we examine some ways that our architecture can improve on currently existing communication paradigms by examining the modifications we have made to email. We further explain the status of our project and some performance data in section 5. We then compare our work with some related work in section 6 and we conclude our paper in section 7 while also examining some of the current shortcomings and potential research areas opened up by our architecture.

2 Overview of Davis Social Links (DSL)

Since our system utilizes the DSL protocol, it would be useful to first give a quick summary of DSL and how it aims to reduce spam while increasing connectivity amongst its users. To better understand how DSL and our OSN kernel work as well as why we believe it is effective, we will reference an example: Consider the simplest form of communication. Alice wants to send Bob an email. However, Alice does not know Bob's email address nor does she know him well enough to ask him using a different medium. With DSL, she can rely on her network of friends to help find and convey her message to him through a system where he can quantify her trustworthiness. In our example, Alice is friends with Carol, who in turn is friends with Bob. Ignoring the decimal values for each friendship link, refer to the figure below:

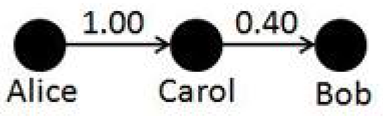


Fig. 1. An example social graph

2.1 Social Routing

We see that in the human society, people can communicate with each other if they can develop a social route amongst themselves. While Alice does not know Bob well, Alice can depend on her friends, namely Carol, to introduce her to Bob. As a result, many of Alice and Bob's initial social interactions are founded on the reputation of their mutual friend. If Alice turns out to be a scammer, then Bob is less likely to trust Carol in the future. On the other hand, if Alice and Bob get along very well, they are both more likely to trust Carol. In the digital world, the current internet model uses routable identities (e.g. email address) as the mechanism for people to interact. Unfortunately, these routable identities are also used by spammers to send unwanted messages with little fear of repercussion. The DSL model of a community based social network model tries to incorporate human behavior of using social routes for message transmission. Here, we use online social networks to define friendship, which can grow stronger or weaker as various users interact with each other. In other words, DSL reaps the benefits of 'social context' existing in OSNs to build communication protocols with reduced spam and higher controllability to message receivers. Each user (or node in the social graph) sets up Profile Attributes ($PAtt$) [1] which are then propagated in the network to allow other people to contact them. Each Profile Attribute k (for node v) is propagated to other users according to the policy associated with it:

$$\forall k \in K_v^{PAtt},\ \exists Policy(k) = [D, T, C]$$

Keywords received from other nodes are termed as Friendly Attributes ($FAtt$). Only those nodes can contact v with the keyword k which satisfy the above policy i.e., the node must be within D hops, all the links on the social path must have the minimum trust level T and all the nodes on the path must have all the keywords in C in their profile attributes. Thus, the receiver gets a large amount of control on who can contact him/her. The keywords help a node to route messages by deciding the next node in the social path. Previous researches like [2] [3] [4] have also used profile information to route messages or search queries in small world networks. In DSL, the information that nodes use to route messages are based on the keywords that they have. Thus, keywords serve as lose identities for nodes in place of global identifiers. In the following section we discuss the trust model.

2.2 Trust Management

DSL utilizes KarmaNet to manage trust so that bad nodes are removed from the network and good nodes' communication are not affected by bad nodes. In KarmaNet [5], bad nodes are nodes who either send unwanted messages or those who utilizes network resources but do not contribute to the network. Good interaction result will be propagated from destination to source and the nodes on the social route will be rewarded up to the sender. Bad interaction result will cause the social path to be punished from destination to source. If a social link is below a certain threshold, the message sent along that link maybe dropped with probability proportional to trust. In Fig. 1, we show values marking how much each person trusts the previous hop. For example, Carol completely trusts Alice and therefore Alice's trustworthiness, as judged by Carol, is 1.00. On the other hand, Carol is not very well trusted by Bob and her trustworthiness is only 0.40. Note that each person in the relationship may judge the other differently. In this instance, while Carol completely trusts Alice, Alice may not reciprocate. In fact, Carol's trustworthiness, as judged by Alice, may only be 0.30, for example.

KarmaNet is a fully distributed trust management protocol which can be used both in centralized and decentralized system. Therefore we use it for manage the trust of our system.

3 Companion Architecture

The purpose of the architecture is to provide an easy way for many applications to gain meaningful social context. By leveraging this data, applications such as email, Skype, online search, and multiplayer gaming can benefit by adding trust and route discovery to the system. We have purposefully designed our architecture in order to minimize changes to any third party application source code, especially at the client side. For example, we have successfully implemented a DSL-compatible version of email that only requires changing the SMTP server but not the client-side email application. This flexibility allows developers to add DSL functionality without forcing their clients to update anything. In fact, it is possible that the end users may not have to change anything at all. In this section, we will describe the system and how it's architecture facilitates this:

All applications that wish to take advantage of our architecture must have a way to remotely communicate with the architecture. If the application cannot communicate remotely, then a plug-in must be written for it or the source code must be modified and recompiled. Remember, we want to only perform minimal or no changes on pre-existing applications. So our architecture only requires that the user change which SMTP server she will be sending the email too. The SMTP server that the user is sending her email to will be the server that is connected to the underlying architecture, diagramed below, that allows social network data to be used for the application.

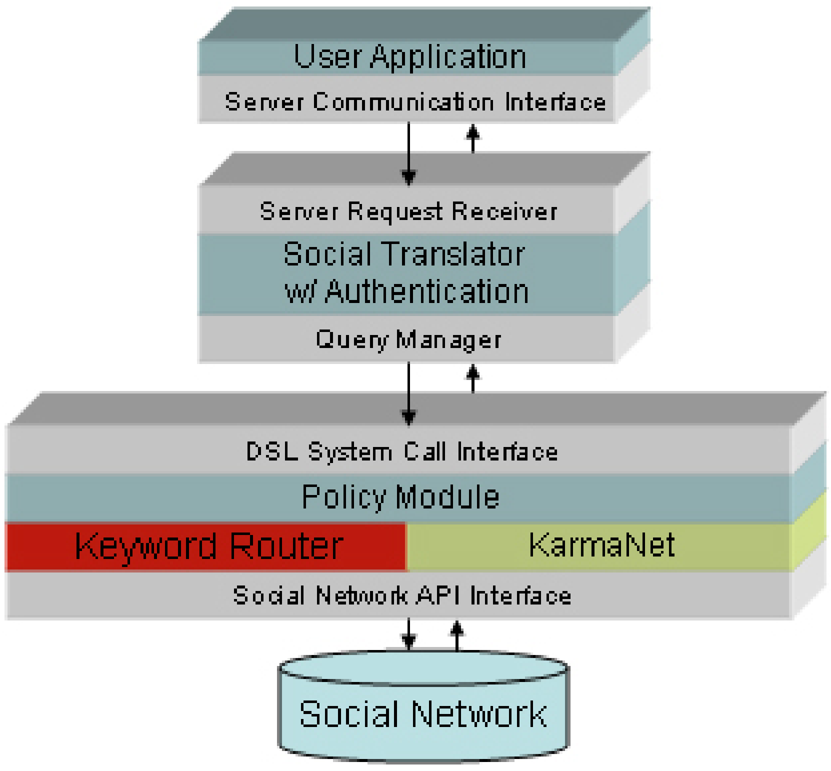

User Application
Server Communication Interface
Server Request Receiver
Social Translator w/ Authentication
Query Manager
DSL System Call Interface
Policy Module
Keyword Router
KarmaNet
Social Network API Interface
Social Network

Fig. 2. Davis Social Links Architecture

3.1 Request Receiver

Once a user sends an email, our architecture's request receiver, or the SMTP server in this case, receives the email. The role of the request receiver is essentially to sit and wait for data from the user. The request receiver will typically expect some form of content and the necessary identification which varies between applications. In this case, the request receiver is an SMTP server that waits for the sender and receiver email addresses as well as the email content. Once the request receiver receives data, it then sends the data to the next component of the architecture, the translator.

Returning to our old example, let us say that Alice is using Thunderbird and wants to send an email to Bob along a social path. She first must configure Thunderbird and change the SMTP server address to the address of the SMTP server inside our architecture. This SMTP server is the request receiver of our architecture. Sendmail is setup on the SMTP server to run all email through a milter. This milter is a program that is given every email that Sendmail receives. The milter extracts the source and destination email addresses as well as the content.

3.2 Social Translator

The social translator is an application specific module that translates between the IDs provided by the user (in this case, the sender's and the recipient's email addresses) to the corresponding social network ID. If the application cannot provide a global unique ID, then it is impossible for the translator to perform any lookup. As a result, our architecture cannot be used if global unique IDs cannot be provided by the application. After translating the IDs to the corresponding social IDs, the translator then returns the social IDs for the sender and recipient to our request receiver, which sends the IDs and the message content to the Query Manager.

3.3 Query Manager

The QM's purpose is to retrieve the social network data requested as well as provide the data back to the application requesting the data. When the QM receives the email content as well as the social network IDs, it queries the social router. The QM can either request the optimal social path or a set of paths so that the user can choose the one he prefers. Once the QM receives the social paths between the two users from the DSL layer, it has to perform some action with the social network data. One option would be to allow the user to choose the path he prefers the message to take, although in the interest of making DSL unintrusive, the Query Manager can simply choose a best path according to some preset conditions.

In our email implementation, a user such as Alice can choose a path before actually sending the message. Giving Alice an interface to choose a social path can be accomplished in many ways. We chose to create a web interface via a Facebook application. In this interface, Alice can view all ”send pending” message and choose a path to send the email along to Bob. We realize this would be a tedious process to do for each email a user wanted to send. Thus, we have added configurability options. Alice can choose to always send a message through a certain path (for emails sent to Bob after the initial one) thus eliminating the need to deal with the interface every time she wants to send an email to Bob. Once Alice chooses a path, the email will be marked as sent. Alternatively, Alice can simply allow the OSN kernel to choose a path for her.

3.4 Social Router

Given the user's ID and the intended destination ID, the social router will attempt to either find an optimal path between them using the decentralized algorithm [5] or return a set of routes for the user to choose from. If the router is successful, it will return the path(s) along with the likelihood that the message will be delivered successfully (which is a function of the trust between the nodes in the path(s)). If no path from the sender and the recipient can be found, the social router returns an error. Alice, according to her relationships, will discover that the path to Bob through Carol is her best bet on reaching Bob.

Once a social path has been selected, the router will examine each link along the path. If the trust value, τ of any link is less than a preset threshold, then the router will randomly drop the message at that link with probability equal to $1 - \tau$. This is done as punishment to weaker links since social links who's trust value is below the threshold are deemed untrustworthy and may be connected to a malicious user. The social router will then inform the query manager of it's decision and the query manager can then send out the email or drop it accordingly.

If the message is dropped, then the social router will automatically update the trust values along the path, punishing the nodes and the sender. Otherwise, if the message reaches it's destination, then the social router will also update the trust values along the path, rewarding all nodes along the path for delivering the message successfully.

3.5 Policy Module

The policy module is an optional module that can be utilized for path discovery. The policy module allows users to find recipients based on keywords, as discussed in the DSL paper. The user application provides the policy module with a list of keywords along with the sender's ID and the policy module will return a list of potential recipients that also have the keyword along with a few other contraining characteristics.

3.6 The Recipient's Experience

The receiver, Bob, can also use a web interface to choose whether he will accept the email or not. Bob is shown who the sender is (Alice) and which path (including trust values) she chose to send the email along. However, Bob is not shown the content of the email. If Bob rejects the email, then the application makes a request to the DSL layer to penalize the social path. On the other hand, Bob can accept the message. If Bob does so, the email is then sent via SMTP to Bob's inbox where he can open up Thunderbird to view it. Bob also has the same configurability options that Alice did, but for receiving. He can always choose to accept a message from Alice along a certain social path or any path. Bob still has an opportunity to penalize the path after receiving the email to his inbox if the content of the email is considered spam.

4 Extended Features

In addition to creating a social context for messaging, we have redesigned a few key concepts in messaging by incorporating our DSL system to increase controllability of the reply, reply-all, and forwarding functions.

4.1 Reply

Currently, replying to an email simply means that the user sends the original sender an email with the body included for reference. At the system level, there is no clear distinction between a reply and a new email. In our system, we have decided to implement our own reply functionality in order to incorporate social context along with recipient controllability. Due to the nature of keyword routing, the recipient of a message may or may not know what keyword to use in order to send a message back to the original sender. Furthermore, it may be impossible to actually find a social path if the original sender set up his keywords to be restrictive. Returning to the Alice and Bob example, even though Alice can find a path to Bob using keyword K_a, there is no guarentee that Bob can send a message back using the same keyword. In fact, it is possible that there is no such keyword which could allow Bob to communicate to Alice. This seemed to be a crucial feature in communication and we could not consider our architecture to be complete without it. As a result, we have implemented a system-level version of the reply functionality.

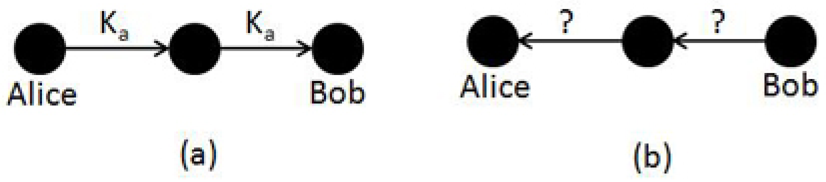


Fig. 3. (a) Alice finds and communicates to Bob using keyword K_a. (b) However, Bob will have trouble responding to Alice if he cannot find a keyword that will return a path to Alice.

When Alice is composing her message, she is given the option of granting reply tokens to her recipient, Bob. Each token allows Bob to reply to Alice once through the social path that Alice used to reach Bob. As a result, Bob does not have to find a social path on his own. After Bob has used up all the reply tokens, if he wishes to contact Alice again, he must find a new social path. By restricting the number of reply tokens granted, Alice can prevent Bob from spamming her.

If, between Alice sending the message and Bob replying, a user along the social path removes himself from DSL, then Bob will simply get a ”path not found” error message and will then have to search for a new path.

4.2 Reply-All

Similar to reply, we have implemented reply-all functionality using tokens. When a user (let us use Alice again for this example) wants to send a message to multiple users, she can grant a number of reply-all tokens (in addition to reply-tokens). Note that if Alice grants x tokens, then each recipient will receive x tokens. If Bob is a recipient of her message and wishes to respond back to everyone, he can use up one of his tokens. Bob's message then travels back to Alice, where it is automatically sent by Alice to all of the original recipients. If one of the recipients decides that Bob's message is spam, DSL punishes the social path from this recipient to Alice and also the path from Alice back to Bob.

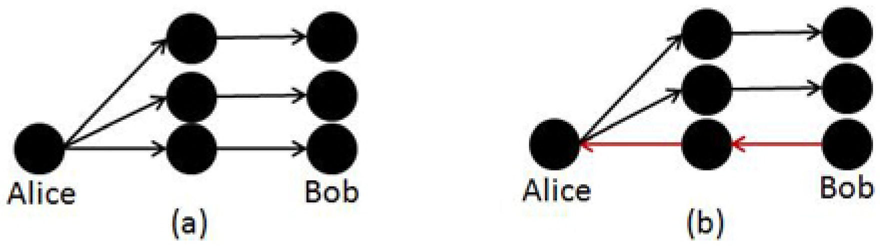


Fig. 4. (a) Alice sends a message to multiple recipients. (b) Bob wishes to reply to all the recipients, which he does so by first sending the message to Alice and having her forward the message to everyone else.

We are currently considering the situation in which one node is part of the path from Alice to Bob along with being part of the path from Alice to some other recipient. If Bob does a reply all, this node will be affected twice. For example, if the recipient marks Bob's message as spam, the node will be punished as the outcome traverses from the recipient to Alice and again when the outcome traverses from Alice to Bob. Similarly, the node can be rewarded twice if the message is marked as being good.

4.3 Forward

Let's assume that Alice and Bob are professors at a university. Alice is going to be giving a presentation on a subject that she thinks Bob's students may be interested in hearing. Alice then sends a message to Bob asking him to forward the message to his students on her behalf. We assume that Alice does not know who all of Bob's students are and therefore are unable to contact them directly. Alice, when composing the email, has the option of granting Bob a forward token, which means that Alice is willing to accept some of the risks that Bob would take by forwarding the message. As a result, if Bob's students decide that the message is spam, Alice would receive some of the punishment, as would Bob. Similarly, if the students really liked the notice about the talk, both Alice and Bob would be rewarded. We have designed a prototype of the forward functionality and we are currently experimenting with how much punishment or reward Alice and Bob should each receive.

One important thing we want Alice to be able to control is the integrity of the message. While Alice most likely trusts Bob, since Alice's reputation can be adversely affected if Bob makes bad changes to the message, we are currently preventing the recipient from modifying the body of the original message before forwarding it. They are, however, allowed to add a new note intended for recipients of the forwarded message.

Another situation that we are currently experimenting with is if Alice knew exactly who the forwarded message's recipient is. For example, let us assume that Alice wishes to send her resume and job application to a hiring manager but she knows that her application would receive more weight if it was forwarded by her friend Eve, who knows the hiring manager very well. We are currently testing out a system where Alice can specify the final recipient (the hiring manager) and Eve will automatically forward the message to the hiring manager. As a result, DSL will find a path from Alice to Eve and then from Eve to the hiring manager.

There are some challenges we need to solve for this to be successful, however. While having the message be automatically forwarded makes things much easier for Eve, Eve is not able to add a personal note before the message is forwarded to the hiring manager.

4.4 Other Application Support

Email works quite well with our architecture. However, the question is how well are other applications supported with our architecture? The main requirement our architecture has for applications is that they must have globally unique IDs within the application's scope. The relationship between the application's IDs and the underlying OSN's IDs is crucial. Without these application IDs, it is nearly impossible to obtain a user's social network information since the kernel cannot translate between the two sets of IDs. In this subsection we will examine how authentication at the application level affects the effectiveness of our OSN kernel architecture.

Another type of communication besides email is Skype[6], which allows users to make telephone calls over the internet. While Skype users do have global unique IDs, they're encrypted and cannot be easily obtained by the user or the architecture. Though it may seem it would be impossible for Skype to work well with the architecture, there is a solution that satisfies the architecture's requirement. The solution involves Skype creating a server that acts as the middleman in the communication between the user and architecture. The user can choose to notify the Skype servers that the user wishes to utilize the architecture which will have Skype decrypt the IDs and send them over to the architecture. This would allow an application ID to social network ID translation. After this, Skype's usage of the architecture can be similar to what we have done with email.

A unique way to utilize the architecture for non-communication purposes would be to associate search words in Google with friends. The idea is that when the user submits a search query, the architecture can find out what friends are associated with particular words in the search terms. The user can simply be presented with what friends are associated with what words in the search terms as a result. On the other hand, an interesting use would be utilizing the friends associated with the search terms to influence Google's search results. How much influence a friend has on the search terms could be fine tuned by the user.

5 Status

Currently, we have adapted email to work with our companion architecture as a proof of concept. To analyze it's performance, we assume that the cost of the sender sending the email to the architecture and the architecture sending the email to the receiver is

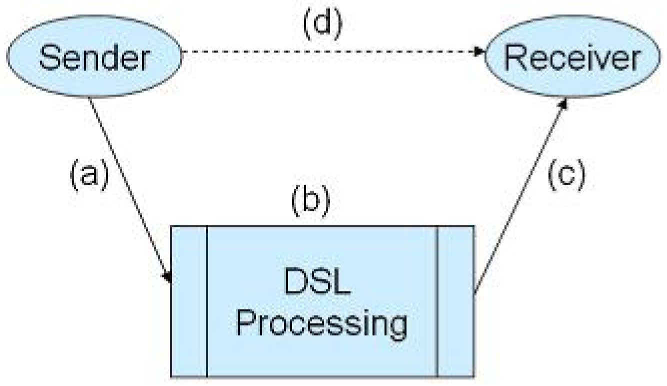


Fig. 5. A high level overview of the architecture. (a) First email sent to DSL. (b) DSL processes the email. (c) DSL sends email to receiver. (d) The original path email follows without DSL.

double the cost of sending email regularly. Thus, we only measure the overhead of the architecture processing. Utilizing the architecture for email can sometimes require human interaction as previously outlined. We chose to measure the scenarios that don't require human interaction. In these scenarios, the emails match some criteria (default path, trust threshold) that allows the emails to simply be sent to the receiver without the need for the receiver or sender to select any options for the message.

The process of sending a message through our system can be broken up into 3 parts. The first part is when the sender sends the email to the supporting architecture. The next part is the architecture applying the required work on the message prior to actually sending out the message to the receiver. The final part is sending the message to the receiver. For this analysis, we assume that the cost of the sender sending the email to the architecture and the architecture sending the email to the receiver is double the cost of sending email regularly. Thus, we only measure the overhead of the architecture processing.

Our tests were ran on a system with an Intel Xeon E5345 (2.33GHz) processor with 8GB of RAM. We measured the overhead that would be presented by the extra architecture processing for 1,000, 10,000, and 100,000 consecutive emails. We ran three tests to confirm that our results could converge to a similar result. Each message was 1000 characters long. Processing each of the 1,000 emails was executed on a single thread and the processing of the next email would only start after the current email finished. The overhead per email in all of those scenarios converges to approximately .00212 seconds. The overhead is quite small in our implementation that still has room for code optimization. We can conclude that the overhead required for our architecture is small enough to warrant the benefits of utilizing the architecture.

6 Related Work

Some work has been done on utilizing social networks in email systems, such as TrustMail [7] and RE [8]. Given a sender and a recipient, TrustMail finds a social path between them and further computes a trust score for the sender, thus the recipient can decide whether to accept the emails from the sender. Different recipients may get different trust scores for the same sender, according to the social paths between them. RE allows users to propagate their whitelists on the social network such that a recipient can decide whether to whitelist a sender based on the social relationship between the recipient and the users who have whitelisted the sender. These work couples their applications with social networks tightly and describes briefly how applications and social networks interact, while in this paper, we dedicate a separate layer for general applications and discuss in detail how to implement such interfaces. We believe that with DSL architecture, it will be easier to build new applications or connect existing applications to social networks.

There is also effort to consolidate existing social networks from industry. Notable examples include OpenID and OpenSocial. Developed by Google along with a number of social networks, OpenSocial provides a set of common APIs for social network applications, which can serve as the communication layer between DSL and existing social networks. OpenSocial, however, does not provide the high level features proposed in DSL such as reputation systems and social routing. Industry may have developed similar ideas as DSL internally, such as some social network projects presented in Microsoft Research TechFest 2009, while to our best knowledge, no details has been published. Presenting DSL architecture here, we wish this paper may initiate further discussions in software system level on how to connect existing social networks to future OSN applications.

7 Conclusion and Future Work

There are three sets of standard applications that we have to accommodate with the architecture. The first is applications that have already been written that dont allow plug-ins to be written for them. These applications are clearly the toughest to satisfy as we have to find ways for them to communicate remotely with a system they were never designed to communicate with. Beyond modifying the source code, we believe there isnt much room for improvement for our architecture to support this set of applications.

On the other hand, the set of applications that support plug-ins can typically be programmed to remotely communicate with any system the programmer desires. Similarly, the third set of applications being applications written from scratch can always be programmed to remotely communicate with any system. Thus, we hope to develop another API layer above the whole architecture presented in this paper to support these two sets of applications. With this API, the architecture is completely hidden from the application as the developer (with sufficient permissions) can read and write data through the API.

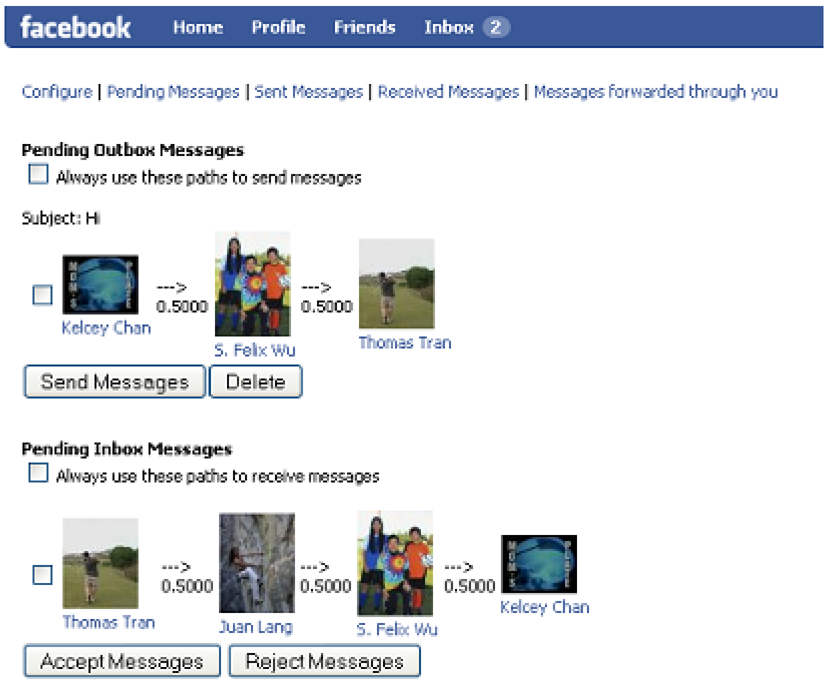
Acknowledgements

We would first like to thank Ken Gribble for his tireless help with setting up the milter and configuring it to our specificiations. We would also like to thank Juan Lang, Lerone Banks, and Matthew Spears for their help and guidance on this paper. This research is funded in part by National Science Foundation under CNS-0832202, Intel, and an MURI grant from Army Research Office under the ARSENAL project and we would like to thank them for continued support.

References

1. Banks, L., Bhattacharyya, P., Wu, S.F.: Davis social links: A social network based approach to future internet routing. In: FIST 2009: The Workshop on Trust and Security in the Future Internet (July 2009)
2. Kleinberg, J.: The small-world phenomenon: An algorithm perspective. In: STOC 2000: Proceedings of the 32nd annual ACM symposium on Theory of computing, pp. 163–170. ACM, New York (2000)
3. Milgram, S.: The small world problem. Psychology Today 61, 60–67 (1967)
4. Sandberg, O.: The Structure and Dynamics of Navigable Networks. PhD thesis, Chalmers University (2007)
5. Spear, M., Lang, J., Lu, X., Wu, S.F., Matloff, N.: KarmaNet: Using social behavior to reduce malicious activity in networks (2008), `http://www.cs.ucdavis.edu/research/tech-reports/2008/CSE-2008-2.pdf`
6. OpenSocial, `http://www.skype.com/`
7. Golbeck, J., Hendler, J.: Reputation network analysis for email filtering. In: Proceedings of the 1st Conference on Email and Anti-Spam (CEAS) (2004)
8. Garriss, S., Kaminsky, M., Freedman, M.J., Karp, B., Mazieres, D., Yu, H.: Re: Reliable email. In: Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI), pp. 297–310 (2006)

Appendix: Screenshot of the Application


facebook
Home
Profile
Friends
Inbox 2
Configure | Pending Messages | Sent Messages | Received Messages | Messages forwarded through you
Send Messages
Delete
Accept Messages
Reject Messages

Pending Outbox Messages

☐ Always use these paths to send messages

Subject: Hi

☐ Kelcey Chan ---> 0.5000 S. Felix Wu ---> 0.5000 Thomas Tran

Pending Inbox Messages

☐ Always use these paths to receive messages

☐ Thomas Tran ---> 0.5000 Juan Lang ---> 0.5000 S. Felix Wu ---> 0.5000 Kelcey Chan

Fig. 6. A user's pending inbox and outbox. Here, the user can manage messages to be sent out along with ones that they are about to receive.