

Leveraging Social Network Data for Messaging Applications

By

Kelcey Chan

B.S. (University of California, Davis) 2006

M.S. (University of California, Davis) 2009

THESIS

MASTER OF SCIENCE

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Dr. S. Felix Wu

Dr. Kenneth I. Joy

Dr. Chen-Nee Chuah

Committee in Charge

2009

Abstract

Social network popularity continues to rise as social networks broaden out to more users. Hidden away within these social networks is a valuable set of data that outlines relationships amongst the users. Networks have created APIs such as Facebook Development Platform for developers to build applications that can leverage the social network's data. Unfortunately, in most if not all social networks, the data can only be accessed within a web page. Many applications that could benefit from social graphs such as Thunderbird and Skype, cannot. In this thesis, I 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. The proposed architecture attempts to preserve the original application semantics and provides the communicating parties a social context for control and management. As the architecture includes two coupling layers, it is then possible to shift some of the services from the original application into the social context layer. To validate the architecture, I have implemented a DSL kernel prototype as a Facebook application called CyrusDSL and a simple communication application combined into the DSL kernel that is unaware the Facebook API.

Chapter 1: Introduction

The rising growth in popularity of online social networks (OSNs) has been phenomenal in the past few years. There are millions of people connecting with one another and maintaining relationships through these networks. As a result, there is a rich social graph that exists for each user that outlines his or her relationships with others in the network. This social graph can be used to enhance many applications.

OSNs have developed APIs for developers that allow them to utilize the social network's data to enhance their applications. Whether it means sacrificing 10 friends for a Whopper or building a Fluff Friend with more decorations than one's friends, social network applications really can use the social graph in unique and powerful ways. I take a look at how I can enhance communication applications such as email through Davis Social Links (DSL) [1].

The graph present in OSNs is virtually a digitized format of a user's social life. The hypothesis that social networks are small world networks (with property of small diameters) where everyone can connect to everyone else using a short path length motivated me to exploit the presence of this rich user information set to build communication protocols based on trust and reputation of the users. In this thesis, I present an architecture that attempts to leverage the rich user set represented in the form of a social graph to build communication protocols. Thus, the architecture attempts to bring 'social context' to communication.

I present an architecture that attempts to give communication applications the ability to leverage DSL OSN data without requiring the user of the application to log into his or her social network. DSL up until now hasn't actually been used in email. Given

that approximately 75% of all email is considered spam [10], it was imperative that we get DSL working with email to see how well it can really combat spam. The architecture attempts to require little to no source code modification for pre-existing applications that wish to leverage social network data. Another goal of the architecture is that the application doesn't break when a social network changes its HTML which is a problem when having a bot crawl web pages for data. As with any application that works with social network data, a user must be able to be identified to determine what access he or she has to certain data. Thus, the system only works with applications that can provide global unique identifiers within the application's domain for users. For example, an email address is a unique identifier that can be used to identify a user to his or her social network data.

To establish the effectiveness of the architecture, I am currently building a prototype that 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 thesis, I will delve into how the architecture builds off of DSL to provide communication applications access to the rich social graphs to add an element of trust and reputation in user-to-user interactions. In the next chapter, I will discuss some work that's related to what I'm currently doing. By giving some background on DSL in chapter 3, I can present the architecture in chapter 4. Next, in chapter 5, I present how my prototype for email works with the outline of the architecture. I examine how I can enhance certain features in email via the architecture such as replying and unifying multiple OSN social graphs in chapter 6. An analysis of

the architecture is presented in chapter 7. In chapter 8, I discuss the limitations and usability issues with the architecture. I finally conclude the thesis in section 9.

Chapter 2: Related Work

Some work has been done on utilizing social networks in email systems to reduce spam. Similar work relies on reputation within an email system. The difference between each system is how reputation is calculated and presented to a user. Furthermore, each system uses different ideas to incorporate their reputation schemes into current email protocol. I take a brief look at two systems that attach some type of reputation to users inside a network in order to reduce spam.

2.1 TrustMail

Golbeck [6] has developed a prototype email interface, TrustMail, which attempts to build a social network designed to reduce spam. In TrustMail, two people are directly connected if at least one person has scored the other. The two users can score the other differently. We can see how a social network is built when assigning a score to user also creates an edge to that user as well. The single requirement of this system is that a user needs some interface to assign a score to another user as no such interface exists in web-based or desktop application email. Golbeck states that a good starting place to create the social network is through email service providers such as Yahoo!, MSN, and Google to implement such a system since they already have many users. Another benefit of using service providers is that they also provide IM services. Thus, the reputation system can be universal across all of a provider's communication applications.

Of course the problem with this is that the system would be restricted to each service provider. What would happen if a user with a Yahoo! email account wanted to rate Gmail user? Clearly the solution is creating a unified social network that supports all users. Once a reputation network is created, a metric must be applied to the reputation

scores. Once a user sends a message to another user, the system finds a path between the two and presents a reputation score along the path that the recipient can use to determine if the sender is trustworthy or not.

Their current implementation is a mail program similar to applications such as Outlook Express that displays trust next to each email in a user's inbox. Incorporating the exact same functionality inside popular applications such as Thunderbird would require code source code modification for every email application. Though, as I will present later, TrustMail's concept can be realized through the architecture in this thesis without requiring source code modification.

2.2 Reliable Email

RE [7] 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. RE is actually an application that utilizes other already used email technologies. An important note is that the tools that RE uses didn't need any code modification. However, RE requires every email domain to run attestation servers in order for RE to work. It seems like that in order for this to really be useful, RE must be incorporated into actual email protocol as there are numerous email domains. Similar to TrustMail, RE's whitelist functionality can also be implemented via my architecture.

Chapter 3: Davis Social Links

DSL uses the available OSN APIs to build the social graph and facilitate the introduction of social context in the communication layer. Thus far, a friendship, in the OSN perspective, has been binary. Two users either are friends or they are not. However, I 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 people to trust some friends more than others. .

The DSL kernel architecture I have designed and implemented gives communication applications the ability to leverage social graph data without requiring the user of the application to access his or her social network site. I can actually authenticate a user to his or her social graph data without utilizing the user's OSN username and password.

DSL revolves around the idea of utilizing social paths to send messages along as opposed to routable identities (e.g. email). Routable identities, while convenient, are easily abusable. The owner of the routable identity has little to no control over controlling who can and who can't contact the owner once his or her routable identity is known. If a spammer obtains the user's routable identity, it's quite trivial to continually spam the user. The user can whitelist or blacklist users, but spammers can always obtain new email addresses from different origins. DSL's key contribution is to mitigate the vulnerabilities from routable identities by being built atop a social network where messages propagate along social links.

DSL's main purpose is to eliminate spam as it introduces a reputation system for users within a social graph. To illustrate how DSL can do so, I will reference a small

example. Assume we have two users, Alice and Bob. 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 the example, Alice is friends with Carol, who in turn is friends with Bob as seen in Figure 1.

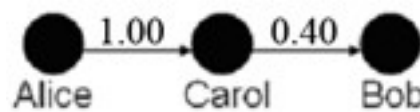


Figure 1. An example social graph where Alice can reach Bob through Carol.

3.1 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's and Bob's initial social interactions are founded on the reputation of their mutual friends. 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.

The DSL model of a community based social network model tries to incorporate human behavior of using social routes for message transmission. Here, I 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.

Attributes and Policy

Each user (or node in the social graph) sets up *Profile Attributes* (PAtt) [1]. Profile Attributes are user-defined/maintained keywords along with their associated policies. Profile Attributes are essentially a loose identity for a user within the network. Note that a user isn't always tied to his or her profile attributes like he or she would be with an email address. The user can modify keywords however he or she wants to. Profile Attributes are propagated in the network to allow other people to contact the owner of the attributes. 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 that 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 research [2] [3] [4] has also used profile information to route messages or search queries in small world networks. In DSL, the information that nodes use to route messages is based on the keywords that they have. Thus, keywords serve as loose identities for nodes in place of global identifiers.

Receiver Controllability

As stated in the previous section, a user can modify keywords however he or she pleases. Beyond that, a user can modify the depth of the keywords within the social graph to either expand or limit incoming messages. Furthermore, a user can tweak threshold for trust. All of these settings can be used to extend a user's openness in the graph or confine the user's openness. If a user is receiving bad messages through a particular keyword, the user need not find a "new keyword" as would be the case with emails. The user can simply get rid of the keyword or tweak settings such as depth and trust. This kind of controllability works well for handling groups of bad messages as opposed to bad messages from a single individual.

A user also has controllability over individuals that continually spam the user. In our example, if Bob continually receives bad messages from Alice, Bob can punish the social path to reduce the amount of messages Alice can send through the social path. This is explored more in the following section.

3.2 Trust Management

DSL utilizes KarmaNet [5] to manage trust so that bad nodes are removed from the network and good communication is not affected by bad nodes. In KarmaNet, bad nodes are nodes that either send unwanted messages or utilize network resources but do not contribute to the network. Good interaction results will be propagated from destination to source and the nodes on the social route will be rewarded up to the sender. Bad interaction 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 may be dropped with probability proportional to trust.

In Figure 1, I 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 feel similarly. 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, it is used to manage the trust inside of DSL.

Trust Structure

KarmaNet's trust structure utilizes the same structure that has been used and studied in [8] [9]. The structure is $T = \langle g_{uz}, b_{uz} \rangle$ where g_{uz} is the number of good interactions and b_{uz} is the number of bad interactions user u has had with user z . b_{uz} and g_{uz} can be affected in a couple ways. First, a user can supply feedback on interactions with his or her direct friends which produces a reputation value, R_{uz} . R_{uz} represents u 's reputation from z 's view. As noted previously, good interaction and bad interaction with indirect friends affects reputation along the path.

A trust structure exists for three actions: *rt*, *fwd*, and *init*. *rt(route)* denotes the ability a node has to route messages. *fwd(forward)* denotes the ability a node has to forward communication. *init(initiate)* describes the ability of a node to initiate communication. All three of the actions are considered to be in a set A. Thus, the new notation for a trust structure is $T_a = \langle g_{uz}, b_{uz} \rangle$ where a is an element of the set A of actions.

By tracking a trust structure for each action, we can prevent a bad node from cutting a good node off from the network. Imagine a bad node that has the ability to send a massive amount of spam through another node (the victim node). Doing so lowers the victim's forwarding value. However, the victim's initiating value is left untouched. As a result, the ability that the victim has to initiate a message and forward messages through direct friends is unaffected by the bad node. What is affected however is the willingness for a direct friend to forward a message that was initiated from the bad node and forwarded through the victim node prior to reaching the direct friend.

The probability that the next interaction will be beneficial for a node is given by the equation:

$$\sigma_{\text{basic}}(m, n) = \frac{m + 1}{m + n + 2}$$

where m is the number of good interactions and n is the number of bad interactions. The equation is also described in [8] [9]. We can see as time increases, a bad node's probability of success decreases quickly. A node can also redeem itself and increase via good interactions.

As for forwarding, when a forwarder, u , receives a message from v , u will forward the message based on u 's trust respect to v , $\sigma(T_{\text{init}}(u, v))$. Thus, the probability a node, u will forward a message from v is given by:

$$p_{\text{forward}} = \begin{cases} 1 & \text{If } \sigma(T_a(u, v)) > \tau \\ \sigma(T_a(u, v)) & \text{Otherwise} \end{cases}$$

which means if the expectation that the next outcome is larger than Γ , then the message will always be forwarded. If that is not the case, then the message will be dropped probabilistically.

A forwarding node must come up with a single value to represent trust to use in σ . KarmaNet encourages nodes to be active and good “citizens” within the network. As a result, KarmaNet chooses to use the *minimum* function in obtaining a single trust value. Utilizing the minimum function, KarmaNet forces a node to be active in all three of the actions. A node must actively initiate, forward, and route wanted messages. When a node u obtains a message originating from node v , then $a = \min(T_{init}(u,v), T_{rt}(u,v))$ for the $p_{forward}$ equation. Alternatively, if u receives a message for forwarding from v , then KarmaNet uses $a = \min(T_{fwd}(u,v), T_{rt}(u,v))$.

Spam Interpretation

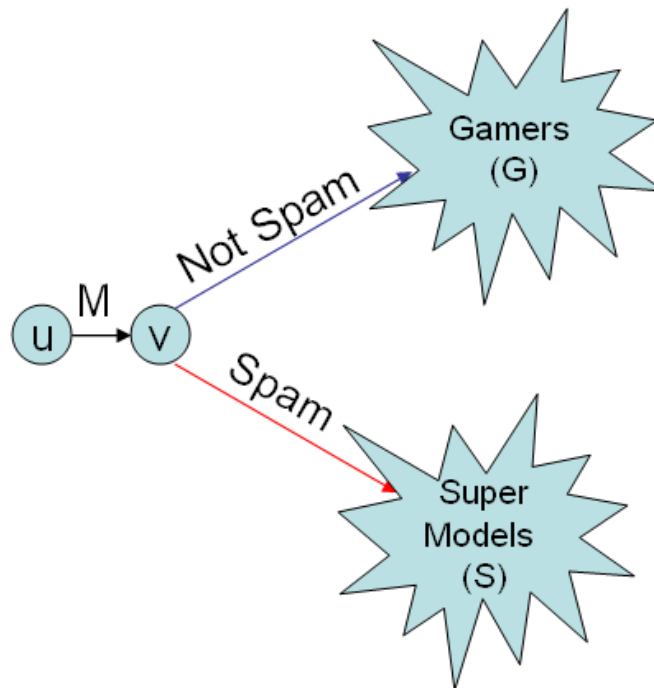


Figure 2. A message forwarded through v that the Gamers group doesn't consider spam while the Super Models group considers it spam.

What may be spam to one person may not be spam to another. Consider a group of gamers, G and a group of super models, S . Then consider a node, u that wants to send a message, M , about a video game, Madden 2009, to G through another node, v . The message initially arrives to group G as well as group S as seen in Figure 2. Assume that a group of super models aren't interested whatsoever in Madden 2009. Thus, the group S views the message forwarded through v as spam. The group S will punish v 's forwarding value while group B will reward v 's message. Thus, v 's forwarding value gains 1 bad interaction and gains 1 good interaction. This result isn't quite ideal since the message was only "sort of" spam since one group actually preferred the message.

We can alleviate this situation and provide a mechanism where continual messages about gaming will gain trust from the gamer group and lose trust from the super model group. Quite simply, we can partition each action, a , in A into the two groups, G and S . By giving each action groups, then certain groups can drop messages that are considered spam to them while other groups can continually accept messages that aren't spam to them.

3.3 Obtaining the Social Graph

There is problem of keeping the social graph up to date and be in perfect synchronization with the OSN social graph. There is no way for DSL to know *exactly* when two users become friends. I explore two methods that try to alleviate the situation.

The first method, which is quite popular, is to crawl the OSN. Since most users allow anyone to view their friends, an application can simply parse the HTML page that

contains the user's friend list. This process can be run each time a user requests his or her social graph inside an application. By doing so, the user's social graph will *appear* to be tracked in real time. Furthermore, the user never has to explicitly inform the system to update the social graph as it happens without the user knowing.

The major downfall of this method is the possibility of OSNs changing their HTML code. If the crawling code isn't up to date, social graph retrieval will either crash or simply fail. Moreover, it's highly likely that this process will run numerous times without the majority of users needing to have their social graphs updated. Finally, we can't make the assumption that *every* user has his or her friend list public.

The second option to obtain a user's social graph is by constructing it within a webpage that utilizes the OSN API. Friend lists are available via OSN API calls. This method can be invoked in a few ways. The first is obviously by having DSL update the social graph when a user logs into DSL. Another way to maintain the social graph is by creating an external webpage that allows the user to authenticate to the OSN to update the graph. Finally, by embedding a DSL widget on the user's page, the social graph can update each time the user's page is visited. Unless there is an issue with the OSN API, this method can't cause crashes or data unavailability.

The downside of this method is that a web page has to be visited in order for the graph to be updated. This means that it requires *some* user to initiate the update process. The architecture could store the user's OSN login information and crawl to the web page that will initiate the update. However, the process would run into crawling issues if the OSN changes the HTML on its pages similarly to the first method. Furthermore, user

login information can become out of sync if a user changes his or her password which would lead to data unavailability.

In the end, I chose the second option of relying on the native API calls to be the best choice. Inserting a component into the architecture that can crash or become unavailable at any given time without an actual coding error in the architecture is simply unacceptable. The best way to invoke the updating process would be embedding an application widget on the user's OSN profile. It's likely that when the user becomes friends with someone, that someone will visit the user's profile, thus initiating an update to the social graph. This is the ideal situation as we'd like to update the social graph each time a user gains a friend. If a user wishes to explicitly update the graph, then the user needs to visit the DSL management system which is discussed in Section 5.4.

3.4 Other Uses of DSL

DSL can be used for a lot more than punishing spammers. It actually can be used in a myriad of applications. Here I take a look at couple interesting applications that can utilize DSL in powerful ways.

File-Sharing

Consider a problem with file-sharing applications such as Bit Torrent and Gnutella. It was reported that 70% of Gnutella users were freeloaders [11]. While BitTorrent requires a peer to upload if the peer is going to download, BitTorrent can't do anything about a peer not sharing the file once the peer has completely downloaded the file. DSL can be inserted as a reputation system for peers within the network where those who don't upload much will have their reputations punished. Perhaps a low reputation

can result in low download speeds. However, a user can increase his or her reputation by sharing more.

Certain BitTorrent networks give the ability to block a user if his or her upload percentage falls under a certain threshold. Unfortunately that cuts the user off completely and forces the user to upload files the user already has. This can be problematic if there is nobody requesting files that the user already has. The user is essentially cut off from the particular BitTorrent network. DSL can give the user a chance to redeem his or her self. DSL allows users to keep their BitTorrent Network IDs even if they're initially freeloaders. When sharing, a user may upload more to direct friends since DSL is built on top of a social network.

Online Gaming

There is currently an enormous problem with online gaming. Individuals that game online are often racist, vulgar, and simply bad sports. Of course there are gamers that are pleasant and others that just enjoy the thrill of competition. Unfortunately, the only way to filter out "bad" gamers is if they either violate software protection laws or are extremely vulgar. Even then, ridding a bad gamer out of the network may not be the solution to the problem. Some "bad" gamers may also want to play with other "bad" gamers.

With the DSL feedback system, a "bad" gamer can have a low reputation. However, that doesn't mean that this gamer won't have anyone to play with. As I stated, some may like to game with a vulgar gamer. DSL can be used to help filter search criteria for gamers. If the majority of gamers don't want to play with a vulgar gamer,

then the majority of gamers just have to specify that they don't want to play with a gamer that has a low reputation.

Furthermore, since DSL is built on a social network, searching for someone to play with can first look for friends, then look for friends of friends and so on. However, that is simply an option as often times it is nice to play with a completely random user.

Chapter 4: Architecture

The purpose of the architecture is to provide an easy way for applications to gain meaningful social context. By leveraging this data, applications such as email and Skype can benefit by adding trust and route discovery to the system with extremely minimal source code modification, if necessary at all. For example, I have successfully implemented a DSL-compatible version of email that only requires the user to change the user's outgoing SMTP server. Though quite obviously, this type of situation isn't always possible.

This flexibility allows developers to add DSL functionality without forcing their clients to update anything. In this section, I will describe the system and how the architecture facilitates this as I follow the flow of information from the application to the DSL kernel.

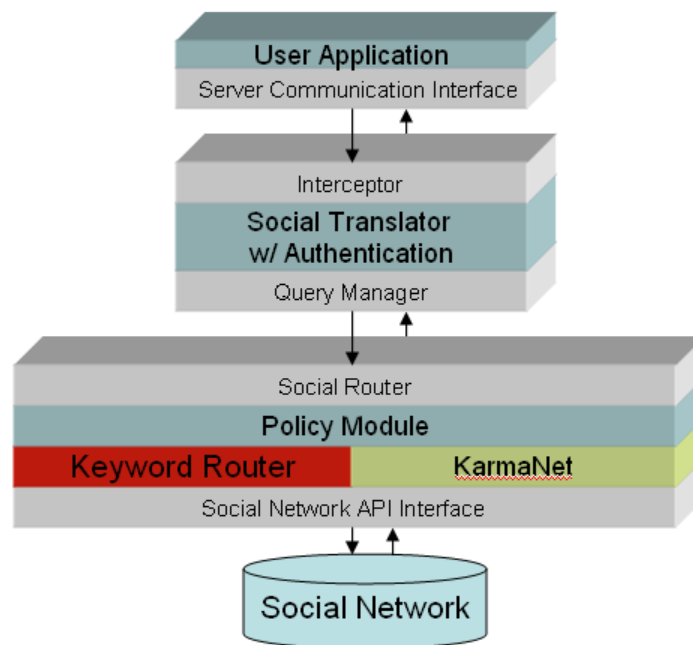


Figure 3. A diagram of the architecture components and how data flows through them.

4.1 Authentication

The first thing any user must do in order to gain access to the OSN data is authenticate to the architecture. I explore a couple ways the user can authenticate to the architecture. The first method I examine is token authentication followed by transitive authentication.

Token Authentication

The idea of token authentication is to give the user a unique token that they can insert into a message to the architecture that proves that the user is in fact who the user claims to be. Let's consider the example of Bob wanting to authenticate to the architecture to access his DSL information with email. Bob will enter in his email address into the Facebook DSL application which will email him a unique key. Bob will need to insert this key at the top of each email he sends to prove to the architecture that he was the Facebook user that registered his email to DSL. Now when the architecture receives the email from Bob, it knows that he is the Facebook user associated with the unique key. Furthermore, the architecture knows that Bob actually owns the email address registered with DSL since he couldn't have the key if he didn't own the address. This is a very powerful feature of token authentication. Note how the architecture can actually authenticate Bob to the email address he registered with DSL (gmail, hotmail, yahoo, or anything). This means the architecture wouldn't have to actually authenticate his email address to any other SMTP server.

Of course this type of authentication isn't very user friendly. Requiring the user to embed a unique key in each request he or she sends to the architecture is cumbersome and harms application usability.

Transitive Authentication

I stated earlier that a user doesn't need to authenticate to any OSN network to utilize the architecture. I realize this feature through *transitive authentication*. If a user can prove to the architecture that he or she is the owner of a global application identifier, then the user is authenticated to both the application and the architecture when he or she is authenticated via his or her global application identifier.

The question is how can the user prove that he or she owns a certain global application identifier to the architecture? Considering the fact that communication, and various mobile applications to name a few usually require and offer the ability to be logged in via a global identifier, we can utilize the ability to message the user on any specific application network. For example, consider a user Dave that wishes to couple his email address authentication with his Facebook authentication. All Dave has to do is login to the DSL application within Facebook and enter his email address in a form. Then the DSL system will send him an email with a unique URL to visit. This URL is tied to the Facebook ID that logged in to initiate the email request. In order for Dave to visit this unique URL, Dave has to log into his email account and click on the link. Then, to access the page link, he also has to authenticate to Facebook. As far as the architecture is concerned, Dave has proven that he is the owner of the email address since he clicked a link that only someone who owned the email address could see. Thus, we can assume if a user is authenticated to Dave's email address, then that user is also safely authenticated to the architecture under Dave's Facebook ID.

While transitive authentication may have to be forced upon the user by requiring the user to create accounts for applications that otherwise wouldn't need them, I feel that

this is the best method for authentication. Since users are automatically logged into most of their communication application via saved passwords, transitive authentication can actually authenticate the user to his or her social graph data without the user manually logging into any network. Furthermore, there is a fairly large set of applications that require a global application ID with the ability to receive messages. The usability gains and low amount of work involved pushed this design choice into the architecture.

Authentication Location

Since I have decided to use transitive authentication, it would make sense that if a user authenticates in the application space, then the architecture doesn't need to perform *any* authentication. Unfortunately there are a couple issues with this idea. First, the architecture has no way of knowing if the user is actually authenticated to the application. Even if the architecture held a list of trusted applications, we run into a similar problem of not always knowing what application is utilizing the architecture. For example, let's say we wanted to allow Outlook Express to obtain DSL data. We can approach this in two ways. The first way would be to set up an SMTP server that simply allows all email to go through. There is no way we can differentiate what SMTP request is from Outlook Express or from just a hacked up application a programmer wrote.

The second way to handle this is by modifying Outlook Express' source code and giving Microsoft a unique key to pass to the architecture when accessing DSL data. This would most certainly work. However, one of the aims of the architecture is to minimize client-side source code modification to zero if possible. So we'd like to be able to set up an SMTP server to take in email to work with DSL without having the developer modify source code. This would allow any mail program to work with DSL. If we do this and

let the authentication happen on the application side, then the server has no idea which application the request is coming from.

As a result of one of the architecture's goals, I allow authentication plug-ins to be written for the architecture. For example, I can write an SMTP authentication plug-in that simply just checks if the user can login successfully to the correct SMTP server. On the other hand, a Skype plug-in can be written with the Skype API.

4.2 Interceptor

This component of the architecture is named the *Interceptor* because it stalls the application's regular flow to perform DSL operations. Really, the Interceptor is typically a remote machine that handles all requests to utilize DSL. Since not all applications will provide exactly the same data, the Interceptor has a look up table that describes exactly what incoming data is required and optional for any application. For example, while email and Skype both have global identifiers (email address, Skype ID), email has a subject field while Skype doesn't. The calling application can communicate with the Interceptor via well known communication protocols such as SMTP.

4.3 Social Translator

The Social Translator's purpose is to map from a global user ID within the application to a DSL ID. This step is clearly necessary as we have to move from application space to the OSN space in order to retrieve a user's social graph. The architecture simply couldn't figure out what user was accessing the data if no ID was given in the first place. Having this necessary step certainly implies that without a global user ID, an application can't utilize the architecture. This is quite true. While this may

seem limiting, most applications that can benefit from DSL data probably have IDs for the user. Once again, email is a prime example of this as it can be tied not only to email, but many other applications that utilize email as login credentials.

As I stated earlier in Section 4.1, the authentication must reside inside of the architecture. The authentication takes place inside the Social Translator prior to actually obtaining the user's DSL ID. After successfully translating the user's DSL ID, the translator may need to translate more IDs depending on how many are given to it (e.g. email requiring two email addresses). Though, only the user's ID needs to be authenticated since a user should only access his or her social graph. The translator will not retrieve OSN data for a user ID that wasn't authenticated. It, however, can apply a global ID to DSL ID translation without authentication merely to retrieve a user's OSN ID.

Once translating all necessary IDs, the translator will pass the IDs to the Query Manager.

4.4 Query Manager

The Query Manager's purpose is to request and perform some action with the DSL data. It organizes the IDs into the correct sequences to obtain the needed social paths. Once it has properly organized the IDs, it passes them to the *Social Router* which will be described in the following section.

When the Query Manager retrieves all the data from the Social Router, it needs to perform some operation on the data. The Query Manager checks DSL user configurations, which will be discussed in Section 5.4, prior to actually acting on the

returned data. Once running through the user's configuration options, the paths get put into a database that can be accessed from the DSL Management System.

4.5 Social Router

The Social Router expects two DSL IDs at any given time to retrieve paths between the two corresponding users. 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. If the router is successful, it will return the path(s) along with the trust values per link in the path. If no path from the sender and the recipient can be found, the Social Router returns an error. Consider the Alice example again. Alice, according to her relationships, will discover that the path to Bob through Carol is her best bet on reaching Bob assuming that every other possible link has a lower trust value.

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 the link with probability equal to $1 - \tau$. This is done as punishment to weaker links since social links whose trust values are below the threshold are deemed untrustworthy and may be connected to a malicious user. The Social Router will then inform the Query Manager of its decision and the Query Manager can then choose what it wants to do.

4.6 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 constraining characteristics. I explore using keywords within the architecture in chapter 6.

Chapter 5: Implementation

I have developed an early prototype that follows the guidelines of the architecture outlined in the previous section. The prototype attempts to use email on top of DSL. To illustrate how the prototype works from a user perspective, first assume that all users involved in the example are users of DSL. The sender that wishes to send an email on top of DSL will first send the email to another user, the recipient through an email client such as Thunderbird. In order to do so, the sender must change his or her outgoing SMTP server to the address of the DSL interceptor. Before the email actually arrives to the recipient, the sender must choose a social path to send the email along. Once the sender chooses the path, the receiver must choose to accept the message after viewing the path the sender sent the email through. The receiver will receive the message to his or her email account after choosing to accept it. Only after the receiver chooses to accept the message can the receiver view the contents of the message. The receiver can then choose to punish the sender if the contents of the message are considered spam which will lower trust values along the path used for the message. The prototype is fully configurable so the sender and receiver don't have to manage paths for every message. This prototype requires no source code modification to any client-side applications. Currently, DSL only resides on Facebook.

5.1 SMTP Authentication

As discussed in Section 4.1, I chose to rely on transitive authentication. I have developed two methods that can achieve transitive authentication. The first is by actually placing the interceptor on the client machine as a milter. A milter is a plug-in written for sendmail that is a program that can run analysis on each email that goes through

sendmail. This method assumes that the user has a mail account tied to the user's local machine. Since the user must authenticate to the machine to be logged in, the architecture can safely assume that the user has authenticated his or her email address and provide access to the user's social graph. How a user links his or her email to the DSL system is covered in Section 5.4.

Clearly there are problems with this implementation. It requires a user to install a plug-in, use a UNIX machine with sendmail, and have their login account tied to an email address. Clearly this isn't feasible in the real world and I developed it to realize the information flow. However, I have come up with a method that alleviates all these problems

In the second method, the interceptor lives on a remote machine and acts as an SMTP server. Similar to the first method, the interceptor is a milter for sendmail. However, instead of processing outgoing email, the milter processes incoming email. The only thing the user needs to do in order to use email on top of DSL is change the outgoing SMTP server address to the address of the machine that hosts the milter. While this is a nice way to lift the annoyances off of the sender, there is still the problem of authenticating the user. Authentication is done on the server-side after the milter has collected all the email information.

I currently haven't implemented the authentication portion of this method. However, it can be realized by recording the user's real outgoing SMTP server through the DSL management system. Then I can use an SMTP library to see if the user's credentials can successfully authenticate to the SMTP server I have recorded for the user. As an alternative, I can keep a mapping of popular email addresses to their corresponding

outgoing SMTP servers (e.g. gmail corresponding to smtp.gmail.com). This can stop users from posing as others via fake SMTP servers as well as not require the user to enter his or her SMTP server into DSL. Unfortunately there are so many email domains that it'd be a tough list to maintain. The interceptor will pass the pair of email addresses to the Social Translator where the actual authentication can be performed.

Milter

As stated in the last section, I wrote a milter to capture the sender email address, receiver email address, subject, and body of incoming email. A milter is a program written in C that can be installed for Sendmail. Each time Sendmail receives or is about to send an email, it runs the email through all milters installed for that specific instance of Sendmail. The milter is essentially a series of callback functions I had to implement. Each callback function would provide pieces of the email that I needed. The initial flow of the email can be seen in Figure 4.

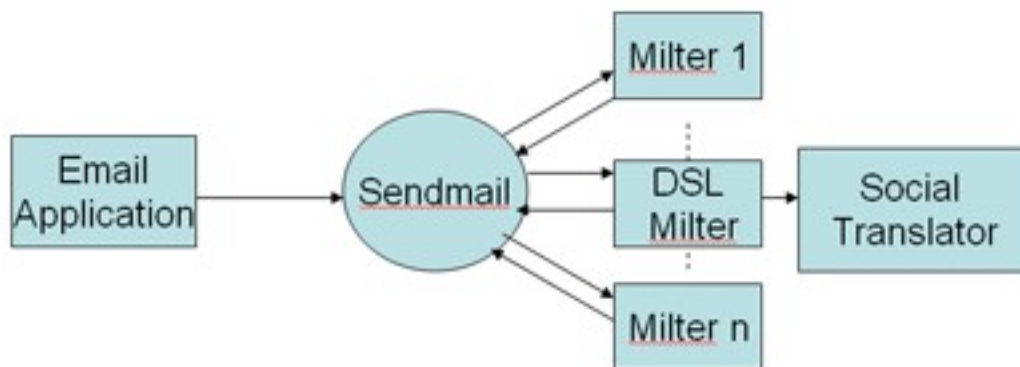


Figure 4. A diagram of the initial flow of an email through the architecture.

The actual prototype differs slightly from the figure above. In our setup, the DSL milter is the only milter that exists. An important note about the DSL milter is that it has to actually reject the email. If the DSL milter doesn't reject the email, then the email will

follow through and arrive at the destination email address. We only want the email to arrive to the receiver after the sender has chosen a path to send the email along and the receiver has chosen to accept the email along that path. The milter collects the data it needs and then sends it through to the social translator. It then rejects the email so the email won't arrive to the receiver. Though, in the event that the DSL milter is installed on an instance of Sendmail that already has milters such as spam filters, the DSL milter *must* be the last milter to see the email due to the fact that it has to reject the email. If the DSL milter is run before any of the other milters, then the other milters will not be able to see the email after DSL rejects it.

The milter uses the following struct to store the necessary email data:

```
struct msgData
{
char* senderEmail;
char* receiverEmail;
char* subject;
char* message;
};
```

Due to the fact that multiple threads can be processing different emails at once, each thread's data must be stored and accessed in some fashion to prevent callback functions from accessing incorrect data for a particular email. The callback functions that need to be implemented do not run on a thread per email. Rather, the callback functions are executed with no notion of "belonging to a thread of execution". The two functions that were used to save and retrieve data specific to an email were:

```
struct msgData * smfi_setpriv(SMFICTX *ctx, void *privateData) // set data
struct msgData * smfi_getpriv(SMFICTX *ctx) // get data
```

where `ctx` is used to identify which email data we want to set or get. We can look at `ctx` as an identifier that specifies what email is currently being processed. Thus a callback function should look similar to:

```
sfsistat mlfi_envrcpt(ctx, argv)
    SMFICTX *ctx;
    char **argv;
{
    // get the data for this particular thread
    struct msgData *mData = smfi_getpriv(ctx);
    // extract some email data and store in mData
    // store the data
    smfi_setpriv(ctx, mData);

    return SMFIS_CONTINUE;
}
```

where at the beginning of the function, a call to `smfi_getpriv()` must take place in order to work on a specific e-mail's thread data. Afterwards, the function should extract some email information and store it in `mData`. Then before the function ends, a call to `smfi_setpriv()` must be called to store the data that has the new piece of email information.

Thus, consider the event when `mlfi_envrcpt()` is called for an email, `e1`, and is instantly called again for another email, `e2`. In the first call, `ctx` is used to retrieve the struct, `msgData`, for the email, `e1`. Thereafter, the same `ctx` is used to store the `msgData` for `e1`. Similarly, in the second call, `ctx` is used to retrieve `msgData` for the email, `e2`, as well as store the data.

5.2 Social Translating

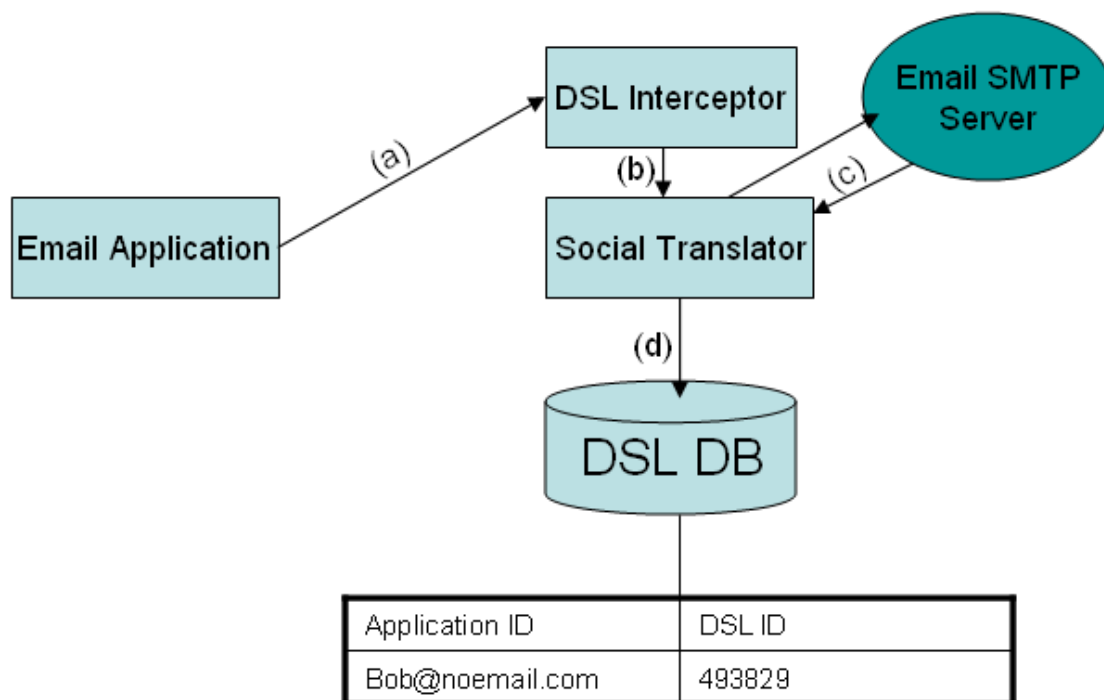


Figure 5. (a) Application sends email to the Interceptor. (b) Interceptor parses email information and passes it to the Social Translator. (c) The Social Translator performs SMTP authentication for the email. (d) If authentication was successful, the translator looks up the user's DSL ID with the user's authenticated email address.

The Social Translator is written in Python which is called from the milter after the milter retrieves the necessary data from the email. The translator will expect two email addresses. One is the sender's email address and the other is the receiver's. The translator will then authenticate the sender's email address to an SMTP server as seen in Figure 5. If the sender's email address successfully authenticates, then the translator will translate each email to their corresponding Facebook IDs. It will then pass the pair of IDs to the Query Manager to handle the data request.

There are two ways the process can fail. The first is when the user can't authenticate to the correct SMTP server. In this event, we can reject the email and notify the calling application that authentication failed. The other way it can fail is when either

of the two emails can't be translated to their corresponding Facebook IDs. In the event that either ID can't be translated, we can notify the user of the error via emailing the user similar to how a sender is notified when an email fails to send.

Due to the fact that DSL's API was written in Python, the social translator is actually embedded Python within the DSL milter. It's embedded Python because starting the Python interpreter for each email turned out to be quite costly. With embedded Python, the interpreter is started once at the beginning of the milter's execution.

5.3 Query Manager

The Query Manager's purpose for this scenario is to retrieve paths between the sender and receiver. Currently it only retrieves the optimal path between the two users. Though I have added the ability to tweak the amount of paths returned. Simply choosing a path based on trust isn't always the optimal path.

For example, consider a professor, Xavier, teaching a class on mutation. Now let's say there's a student, Logan, that wishes to contact Professor Xavier because Logan is having trouble with the homework. So when Logan uses DSL to contact Xavier, he retrieves the optimal path based on trust which is through Xavier's niece. This is the only path that Logan can see. In reality there could be another path through the class' TA to Xavier which holds much more weight in this context.

Once the Query Manager retrieves the data from the Social Router, it runs checks against the possible user configurations. Earlier I stated that it'd be quite cumbersome for a user to have to manage path choosing for every message. Thus I have developed some configuration options that allow the user to have the system automatically perform actions for the user. The configurations will be explained in the next section. After

running through the configurations, the Query Manager will insert the paths into a database which can be accessed via the DSL management system.

Similar to the translator, the Query Manager is also a piece of embedded Python code. It utilizes the DSL API function *findPath(senderFBID,receiverFBID)*, to retrieve the optimal path between the two users identified by their Facebook IDs. The Query Manager then runs a series of SQL queries that checks all of the user's configuration options. If the sender's configuration is set up such that the email will be sent without requiring the user to choose a path, the Query Manager issues another query to see if the receiver's configurations will allow the email to be sent to the receiver without the receiver choosing to accept the email.

5.4 DSL Management System

I have mentioned this management system throughout the thesis as a lot of pieces depend on it. The management system is a web interface that is currently a Facebook application. The system's use isn't constrained to email. It's actually general as most applications that can use DSL will require similar features. First, I will explain the configuration options that a user has in regards to DSL.

Configuration

The screenshot shows the Facebook configuration interface. At the top is a dark blue navigation bar with the Facebook logo and links for Home, Profile, Friends, and Inbox. Below this is a breadcrumb trail: Configure | Pending Messages | Sent Messages | Received Messages | Messages forwarded through you. The main content area is titled 'Account Configuration' and contains two sections. The first section, 'Your registered emails', has a sub-header 'Delete Email Address' and a list of email addresses with checkboxes. One address, 'chankt@cyrus.cs.ucdavis.edu', is selected. Below the list is a 'Delete Selected' button. The second section, 'Add an Email Address', features an empty text input field and an 'Add' button. The third section, 'Basic Settings', includes a checkbox for 'Always use optimal path:' which is unchecked, and a 'Trust Threshold (0 means it won't be used):' field with the value '0'. A 'Change Settings' button is located at the bottom of this section.

Figure 6. The configuration page is where a user can manage his or her emails, enter a trust threshold, and specify to always send an email along the optimal path.

A user can choose to always communicate through a path that meets a default trust threshold as seen in Figure 6. This applies to both sending and receiving. A user can also always choose to send a message through a specific path so future messages sent to the same user will automatically use the default path chosen if the path still exists. The management system is also the location where the user can register multiple emails to use with DSL where the system will send an email with a link for the user to confirm his or her email. This is how the architecture can link a user's Facebook ID to the user's email address. Finally, the last configuration option is to always use the optimal path when sending a message.

Path Choosing and Receiving Confirmation

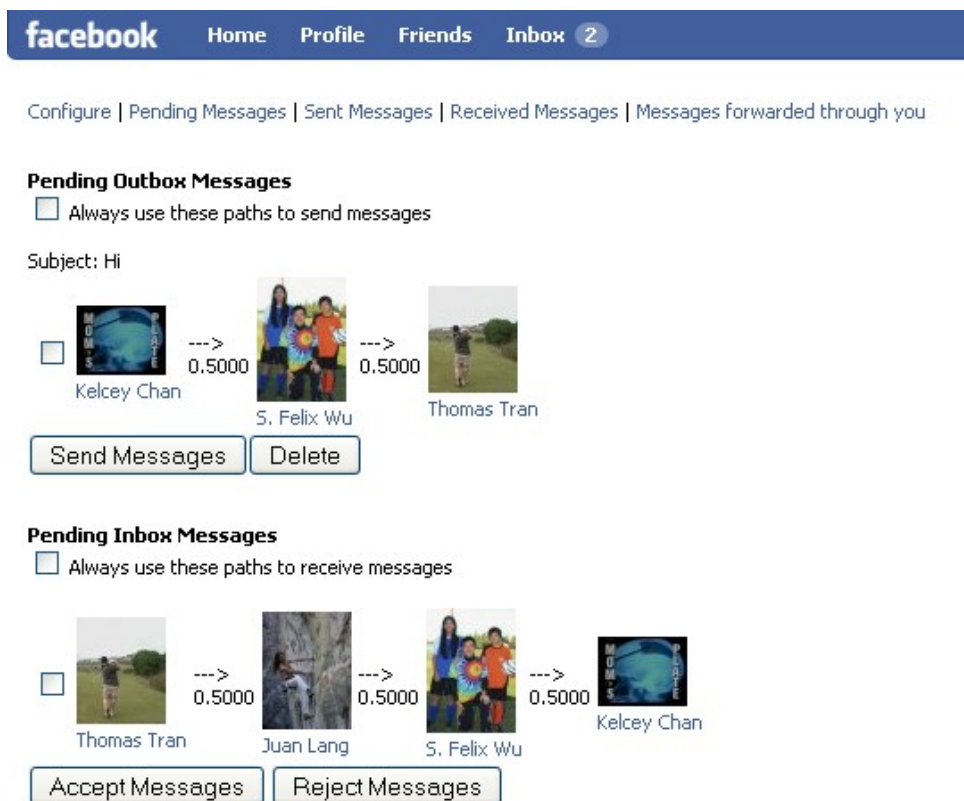


Figure 7. The page that lists all pending inbox and outbox messages.

The main piece of the management system is path choosing as seen in Figure 7. Once a message goes through, a user has to choose a path prior to the message actually going out. The user must choose a path through the management system. The sender can choose to always use a certain path to send a message along to the receiver. The receiver can either choose to accept or reject the message. If the receiver rejects the message, then the path's trust is punished.

History

The ability to view one's history is absolutely necessary given the way the architecture was designed. One major issue with the design is that once the sender sends

the email, the sender has no way of knowing whether or not the receiver chose to accept the email unless the receiver notifies the sender. By allowing a user to view his or her history, the user can see the messages that are still waiting to be either confirmed or rejected by the receiver. Furthermore, the sender can view messages that have been accepted and messages that have been rejected as seen in Figure 8.

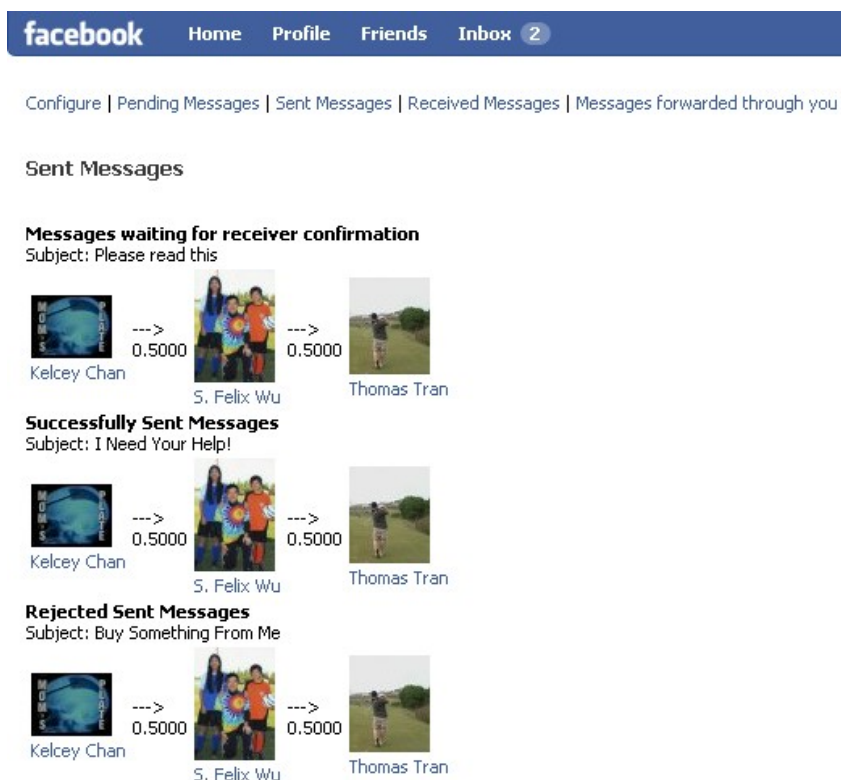


Figure 8. The DSL Management page that lists all the status of messages sent by the user.

A user can view messages he or she has received as well. This page is necessary in the event that the user actually receives spam from another user. This is where the user can go back and penalize another user for sending out spam. Finally, a user can view messages that involved the user where the user wasn't either the sender or the receiver.

Chapter 6: 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. I explain how DSL can add further meaning to these common functions. This functionality can be moved from application space to OSN space via the architecture to enhance the application. I also take a look at how to unify all of a user's social graphs.

6.1 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 DSL, 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 or her keywords to be restrictive. Returning to the Alice and Bob example, even though Alice can find a path to Bob using keyword Ka, there is no guarantee 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 DSL to be complete without it. As a result, we have implemented a system-level version of the reply functionality.

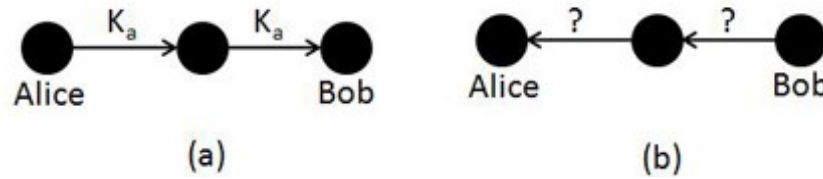


Figure 9. (a) Alice finds and communicates to Bob using keyword K_a . (b) However, Bob will have trouble responding to Alice if he cannot find a keyword that will return a path to Alice.

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.

6.2 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 for the 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.

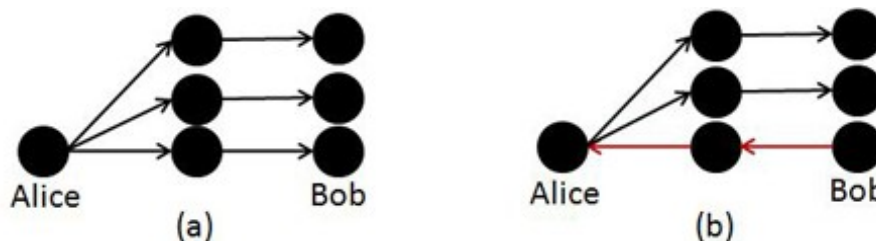


Figure 10. (a) Alice sends a message to multiple recipients. (b) Bob wishes to reply to all the recipients, which he does so by first sending the message to Alice and having her forward the message to everyone else.

We are currently considering the situation where 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.

6.3 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. Assume that Alice does not know who all of Bob's students are and therefore is 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.

One important thing we want is for Alice to be able to control 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.

However, there are some challenges we need to solve for this to be successful. While having the message 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.

6.4 Implementing the Features

Automatically detecting whether a user is forwarding, replying, or "reply-alling" is difficult inside the DSL militer. The best guess the system has is by reading the email subject since replying and forwarding email subjects contain "re:" or "fwd:". Thus, in order to be completely accurate, I once again rely on the DSL Management System and configuration options.

A user can specify that an email beginning with “re:” should always be treated as a reply email if the system detects that it’s a possibility. A similar configuration could be available for forward. Reply-all can work similarly to reply except that the system must detect if there is more than one email address the reply is being sent to.

Aside from configuration, a user can mark any given email as forward, reply, or reply-all when choosing the path for the email. Tokens can be granted when choosing a path to send the email along. Similarly, utilizing tokens can be done when choosing a path as well.

Keyword Implementation

Currently, the architecture isn’t using the keyword functionality of DSL. Utilizing keywords doesn’t make a whole lot of sense when the sender knows the receiver’s routable identity already which is the case with email. Though, we can apply somewhat of a twist to DSL keywords to make them useful. As opposed to simply specifying keywords and a message, we can actually look for keywords *inside* the message. The receiver can keep his or her keywords set up as normal and can choose to accept or reject messages based on the policy of attributes described in Section 3.1. The filter has access to the plain text body of the email as well as the DSL keywords database. So it’d be rather easy to implement keywords into the architecture. Of course keywords would only be set up for emails sent through DSL.

Beyond that, we can actually attempt to apply natural language processing to try and understand *what* the email is about. Rather than keywords simply being text strings, they can be subjects that a particular user is interested in.

6.5 Support for Multiple Networks

It would be a shame to limit the architecture to one network. After all, most people are on multiple networks. To obtain a unified social graph, we must first obtain every social graph for each network the user belongs to. This is quite straightforward by actually using external OSN APIs within one web page to obtain a user's social graphs. This can be placed inside the DSL management page. Since the external APIs don't require the user to actually be inside the OSN's site, we can have the user initiate social graph updates to all networks within one page. After obtaining the graphs, we have to merge them all into one unified graph.

The goal behind designing the method in merging social graphs is to minimize user correction. We would like to match overlapping friends in multiple OSNs so a user doesn't see the same user (from two OSNs) twice. From the social networks, we can gather a user's OSN ID, first name, last name, and display name. This data can be retrieved at the same time a user's social graph is formed. The architecture's job would be easier if it were able to obtain a user's email address. Unfortunately this is not the case for obvious reasons. Clearly the user's ID is different between networks and a user's display name can be different across networks. However, matching on first and last name can produce great results. It's very rare that a single user will have two friends with the exact same first and last names. Though, the situation is still possible. The architecture lets the user correct errors where the merging algorithm has in fact incorrectly matched two users with the same name. The architecture also gives the ability to merge two friends that didn't match on name in the event that a nickname was used in one network and not the other for example. These rare cases can also be rectified

automatically based on users linking their OSN accounts. As we can see, there may be need for a user to manage his or her social graph. The user can fix his or her graph inside the DSL management system.

Chapter 6: Analysis

The process of sending a message through the system can be broken up into 3 parts as illustrated in Figure 11. 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. Utilizing the architecture for email can sometimes require human interaction as previously outlined. I chose to measure the scenarios that don't require human interaction as we know we can just tack on the time it takes a user to login to Facebook and click a couple buttons. In these scenarios, the emails match some criteria (default path, trust threshold) that allow the emails to simply be sent to the receiver without the need for the receiver or sender to select any options for the message. For my first analysis, I compare how long it takes to send email through the DSL architecture to how long it takes to send an email regularly.

Complete Email Comparison

This test was run on a DSL Server with an Intel Xeon E5345 (2.33GHz) processor with 8GB of RAM. Unfortunately I couldn't perform extensive runs with this test without being scolded by network administrators, thus I simply measured how long it took to send 3 emails both through regular email and through the DSL architecture. For "regular" email, I ran a script that sent out 3 emails as quickly as possible straight to the destination. For the DSL architecture test, I ran a similar script that sent out 3 emails as fast as possible, but using the DSL architecture. For this test, the DSL system was set up to automatically send the email without a user choosing a path. I also assumed the worst case scenario in the DSL architecture where the Query Manager is forced to check all the

configuration options for the user. Each email for both tests consisted of 1000 character messages and a 3 character subject. The destination of the email was targeted to a Gmail address.

The regular email system took approximately 3 seconds to send and receive the 3 emails. On the DSL server, the system took approximately 6.5 seconds to complete all email transactions. The DSL server takes over twice the amount of time of regular email. This makes quite a bit of sense due to the fact that using the DSL server requires sending email twice plus the added overhead of processing the email through the DSL architecture. Of course there are things I couldn't account for such as network congestion and Google server performance. However, if we assume that sending an email twice utilizing the DSL architecture is twice the cost of sending a regular email, then we can say the overhead of processing in the architecture for 3 emails took approximately half a second. That would bring the cost of overhead per email (in DSL processing) at .16 seconds. However, once again, the test did only contain 3 emails which is a very small amount. So, the unknown variables in the test could possibly throw off the test slightly since it's such a small task. I attempt to get a better look at the results in the following test.

Measuring Processing Overhead

For this analysis, I 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. I make this assumption to compare these results with the results of the previous section. Thus, I only measure the overhead of the architecture processing.

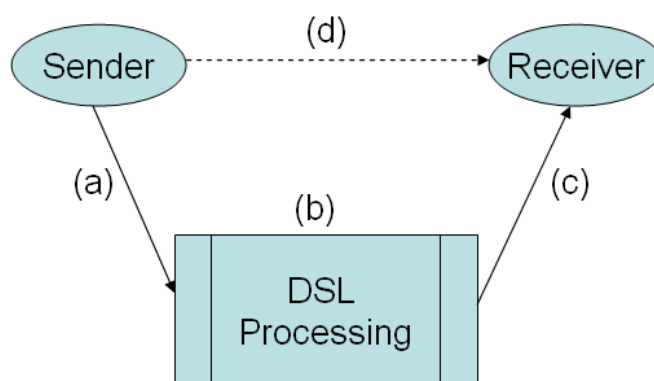


Figure 11. (a) First email sent to DSL. (b) DSL processes the email. (c) DSL sends email to receiver. (d) The original path email follows without DSL.

The test was run in the exact same environment as the first test. Email messages contained 1000 character with a 3 character subject. The DSL processing was assumed to be the worst case where the Query Manager had to run all configuration checks on each email. I measured the overhead that would be presented by the extra architecture processing for 1,000, 10,000, and 100,000 consecutive emails separately. I ran three tests to confirm that our results could converge to a similar result. Processing each of the emails was executed on a single thread and the processing of the next email would only commence after the current email was finished.

Number of Emails	Time in Seconds (Approx.)
1,000	2
10,000	21
100,000	212

Figure 12. Overhead analysis results for multiple batches of emails.

As seen in Figure 12, the overhead per email converges to approximately .00212 seconds. The overhead is quite small in the implementation that still has room for code optimization. Now comparing this data back to the first test yields some interesting

results. There is a discrepancy between the two tests if sending the email to the DSL architecture is exactly the same cost as sending the email directly to the recipient. The first test yielded that the overhead of the DSL processing was .16 seconds while this test gave us .00212 seconds. There's clearly a difference between the DSL server and the reception server (Google's). Furthermore, the server's could be running different mail transfer agents. There are enough variables to explain the difference in results. Regardless, both tests demonstrate that at least the overhead of the processing is quite minimal. Though, sending an email through DSL will take slightly longer than twice as long as regular email.

Chapter 8: Future Work

The architecture is still in its infancy. There are plenty of improvements that can be made to the system in regards to usability. Moreover, expanding the architecture or creating a new one based off the current design to encapsulate more types of applications could be done.

8.1 Usability

There are clearly some usability issues with the system. Consider our prototype implementation where I require the user to go into a browser, login to Facebook and the DSL application, and finally choose a path to send a message along per contact can be quite cumbersome. On that note, when a path is no longer valid, the users will have to re-establish their configuration options. Similarly, there is the problem of keeping the social graph up to date. The management system needs to be more streamlined into the application experience.

To temper these issues, a native application can be built that pops up whenever a user sends/receives an email through the DSL system. This can be accomplished by adding a few lines of code into key places of the mail application to simply invoke the pop up application that handles the entire configuration. This pop up application can be designed similarly to the DSL management system. Rather than requiring the user to login to the OSN and open a browser, the user can instantly handle the management from what would appear to be in the email application. The design of the pop-up application would really have to require *just* a few lines of code to utilize. If it required anymore lines of code, utilizing the architecture the way described in the thesis may not be worth it.

8.2 Generality

Quite obviously, the system works well with email applications such as Thunderbird without requiring client-side source code modification. Similarly, the architecture setup can work well with an application such as Skype requiring minimal source code modification. The architecture has mainly been designed to favor communication applications. So embedding the use of DSL into something like Google Maps Mobile requires significantly more code modification should really just be handled by a DSL API. Perhaps the architecture can only cater to communication applications. A challenge would be extending the architecture to work well with other types of applications. Though that is quite a challenge and perhaps may not even be possible.

Chapter 9: Conclusion

The architecture design works extremely well with email as demonstrated. Bringing social context to email only requires recording a user's "real" SMTP server and having the user change his or her SMTP server in the client email application. The architecture works with any email client that supports SMTP as well. The usability issues can easily be improved. While the architecture can't always be integrated into applications without application source code modification, the foundation is set up to really require little code modification and maintain application semantics. Furthermore, I have shown how we can improve typical email functions such as reply, reply-all, and forward by utilizing social context.

Though, the real question is how easily the architecture can adapt with non-communication applications. With the main goal of the architecture being to minimize client-side source code modification, there exists a distinct separation of whether it's worthwhile to use the architecture or simply use a DSL API library for a non-communication application.

References

1. Banks, L., Bhattacharyya, P., Wu, S.F.: Davis Social Links: A Social Network Based Approach to Future Internet Routing," to appear in FIST'09 (The Workshop on Trust and Security in the Future Internet), Seattle, WA, July 2009.
2. Kleinberg, J.: The Small-world Phenomenon: An Algorithm Perspective. In: STOC '00: Proceedings of the 32nd Annual ACM symposium on Theory of Computing, ACM (2000) 163–170
3. Milgram, S.: The Small World Problem. *Psychology Today* 61 (May 1967) 60 – 67
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) Available at <http://www.cs.ucdavis.edu/research/tech-reports/2008/CSE-2008-2.pdf>.
6. Golbeck, J., Hendler, J.: Reputation Network Analysis for Email Filtering. In: Proceedings of the 1st Conference on Email and Anti-Spam (CEAS). (2004)
7. 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). (2006) 297–310
8. R. Ismail and A. Josang. The Beta Reputation System. BLED 2002 Proceedings, page 41, 2002.
9. M. Nielsen, K. Krukow, and V. Sassone. A Bayesian Model for Event-based Trust. *Electron. Notes Theor. Comput. Sci.*, 172:499–521, 2007.
10. Symantec. The State of Spam - A Monthly Report - January 2008. http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/Symantec_Spam_Report_-_January_2008.pdf.
11. E. Adar and B. A. Huberman. Free Riding on Gnutella. *First Monday*, 9 2000.