Authenticated Out-of-Band Communication Over Social Links

Anirudh Ramachandran and Nick Feamster
School of Computer Science, Georgia Tech
{avr,feamster}@cc.gatech.edu

ABSTRACT

Many existing host-based applications rely on their own authentication mechanisms and peer discovery services. Although social networking sites already provide mechanisms for users both to discover other users (*e.g.*, by logging on to the social network Web site) and to communicate securely with each other (*e.g.*, using instant messages within the social networking site), today's applications have no way to exploit the relationships and trust that are inherent in these networks. This paper proposes *Authenticatr*, a framework that allows applications to use the authentication and peer discovery mechanisms inherent in social networking sites to bootstrap their own authenticated communication channels. We describe motivating applications, detail the interface that *Authenticatr* exposes to applications, and discuss practical considerations and security threats.

Categories and Subject Descriptors

C.2.0 [**Computer Communication Networks**]: Security and protection; C.2.4 [**Computer Communication Networks**]: Distributed Applications

General Terms

Design, Security, Flexibility

Keywords

social networks, authentication, security

1. INTRODUCTION

Social networking is experiencing explosive growth, both in terms of communities targeted by these networks and overall user participation in each of these networks [1, 2]. Online communities such as Usenet and IRC have existed for decades, but the recent wave of social networking Web sites are notable for requiring stricter authentication procedures for registering and creating social connections: Most networks do not allow automated registrations, and creating new social links (*e.g.*, adding another user to your list of acquaintances) requires authorization from the other party. Many newer social networking sites such as Facebook [10] have even stricter requirements for joining a social circle (*e.g.*, to be part of the Georgia Tech group, the user must possess a `gatech.edu` email address).

Despite the rise in popularity of social networks and the inherent information they contain about trust between their users, these networks remain balkanized and largely separate from many applications that could benefit from richer social context. For example, consider a user who wishes to share files with a set of trusted friends who are members of the user's network at a secure social networking site (*e.g.*, Facebook). The filesharing application, however, is unaware of this relationship, and must treat all friends as untrusted users on the Internet; thus, the user must either manually set up and distribute passwords to all friends, or settle for an unsecured service. The filesharing application would benefit from the ability to utilize the secure social link to "bootstrap" its own authentication service. Many web-based applications can also benefit from a common social context: users today must create isolated social networks for each Web-based application (*e.g.*, a user who has accounts on a bookmark sharing application and a Web-based feed reader cannot unify his friends on both applications); if Web applications could tap into a common "social substrate" to find friend relationships, collaboration and sharing could be made seamless [1].

This paper presents *Authenticatr*, a system that uses social links between users—which already exist within the contexts of various social networking Web sites—as a substrate for establishing authenticated out-of-band communication between applications that these users run (*i.e.*, for applications unrelated to the social network). Much like Web services that allow users to authenticate themselves using credentials from other social networking sites [3] or universal login mechanisms such OpenID [12], *Authenticatr* provides applications a generalized view of authentication interfaces; applications, therefore, can treat all social network instances in the same way. *Authenticatr* also enables these applications to communicate using the social network's secure messaging capabilities, which applications can use to

---

[1]Google's Social Graph project [7] addresses a similar goal, using social annotations in hyperlinks as the underlying social network. This service, however, uses publicly available information and cannot be used for secure communication.

perform challenge-response or establish cryptographic keys before initiating direct communication[2].

This paper makes the following contributions. First, we present the design of *Authenticatr*, a framework that leverages the relationships inherent in today's social networks to build authenticated out-of-band communication channels for other networked applications. Second, we describe the *Authenticatr* API, which applications can use to build such authenticated channels. Finally, we describe how developers can use this API to build applications for performing a variety of tasks ranging from file-sharing to network troubleshooting. We also outline various practical considerations for *Authenticatr* and discuss an agenda for future research.

The rest of the paper is organized as follows. Section 2 describes motivating applications for *Authenticatr*. Section 3 presents *Authenticatr*'s design and API, and Section 4 explains how applications can use *Authenticatr* to construct authenticated channels. Section 5 describes various practical considerations and potential security threats. Section 6 outlines related work, and Section 7 concludes with a summary and research agenda.

2. MOTIVATING APPLICATIONS

Authenticatr provides an authenticated out-of-band communication channel for many other applications, including overlay routing, troubleshooting, root-cause analysis for networked applications, censorship avoidance, and authentication for content sharing (*e.g.*, blogs, photo albums, posted news items). Current authentication and access control for these types of applications are either application-specific or too complex for typical users to establish and configure.

In contrast, social links are easy for users to establish and verify. Social links also embody exactly the type of authentication that is required by many applications: To share a set of photos, a news posting, or some other content only with friends, a user can specify access control in terms of existing relationships on social networks (*e.g.*, "my friends on Facebook"). *Authenticatr* extends the same authentication mechanisms to host-based applications, allowing the user to, for example, configure her computer to "Allow my friends on Facebook to route traffic through me". In this section, we detail a few example applications that can benefit from *Authenticatr*'s authentication mechanisms.

Secure email. Although strong end-to-end encryption schemes such as PGP have existed for over a decade, the complexity of public key cryptography has made this approach unattractive to most email users. On the other hand, people already understand and use social networking services for business and leisure, and many social networks support secure authentication and messaging (*i.e.*, HTTP over SSL); *Authenticatr* can exploit the authentication and security already inherent in social networks for securing host-based email clients.

File-sharing between friends. Currently, many users lack the means or the technical expertise to share files only with their friends. Complete security requires setting up and managing servers, which is relatively difficult for most users. A file-sharing client can use *Authenticatr* to select a social

network and a friend (or group of friends) to share files with; *Authenticatr* will exchange session keys with peer programs, establish ports, verify logins, etc. using the social network.

Backup routing and censorship avoidance. Routing failures often cause some parts of the Internet to become disconnected; BGP convergence delays cause these disruptions to affect availability. Censorship poses reachability problems of a different sort: Users may be denied access to certain Web sites on the Internet. *Authenticatr* allows these users to locate a friend's host (using the social network) and route through it to avoid the failure or the censor (previous work has proposed a similar mechanism that applies only to censorship [9]). Routing through a trusted friend's computer would also be preferable to using an anonymous proxy of unknown provenance.

Collaborative measurement and troubleshooting. Collecting measurements from end-users is cumbersome because most users are unwilling (or unable) to operate measurement software; moreover, there is no way a researcher can signal a set of end-user machines to begin measurement (due to DHCP reallocation, machines being turned off, etc.). Such applications can also benefit from *Authenticatr*: whenever a researcher requires a measurement, she can issue a request to all end-user measurement applications (authorized through a social channel, such as all subscribers of a certain Facebook application) to conduct a measurement (*e.g.*, "traceroute to `slashdot.org`").

3. DESIGN

This section presents the design of *Authenticatr*. We outline design goals, present *Authenticatr*'s design and API, and describe how multiple processes can communicate using the same social authentication channel. We are in the process of implementing *Authenticatr*.

3.1 Design Goals

Authenticatr attempts to satisfy the following goals:

- *Simplicity.* The Application Programming Interface (API) that *Authenticatr* presents is similar to the Berkeley sockets API, both in terms of the functions and the data types handled by the API. Simple API calls are easier to compose and build upon. As a result, the primitives provided by *Authenticatr* are minimal (*e.g.*, `send`, `recv`, etc.), but applications or libraries can compose these primitives to perform more complex tasks.

- *Flexibility.* The design of *Authenticatr* is not tied to the functions offered by existing applications or to existing social network structures, which makes it compatible across many of future applications and social networks. The API handles only basic operations (*i.e.*, authentication, enumerating a user's friends, and sending and receiving data), leaving functions that are specific to social networks to lower layers, and application-specific functionality to higher layers.

3.2 The Middleware Layer

Figure 1 presents the high-level design of *Authenticatr*. The middleware layer provides applications with various social-network functions. Its API includes functions for authenticating a user to the social network and sending messages to a user's friends, but is independent of functions
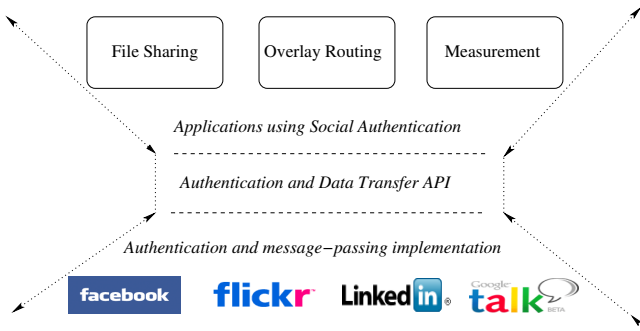

Figure 1: High-level design of *Authenticatr*.

that are specific to a particular application or a particular social-network communication protocol. Because the middleware provides only basic functionality, applications can use libraries built on top of the middleware message-passing API for commonly-used functionality, such as enumerating friends-of-friends of a user, negotiating a cryptographic session key between two users using private messaging, etc. Applications may build upon intermediate libraries on top of the middleware API, or deal directly with the message-passing API for new functionality (*e.g.*, friend "suggestions", or users who are connected via many common friends). The thin middleware layer is analogous to the hourglass design of the TCP/IP stack, which facilitates a wide range of applications to use it on higher layers, and a wide range of lower layer (physical or link-level) protocols and media to interact with the TCP/IP layer from below.

Below the middleware layer, *Authenticatr* implements various types of social network communication and authentication schemes. The implementation includes the communication protocol (*e.g.*, XML-RPC for Web-based social networking sites, Jabber for instant messaging applications, etc.) and the functionality to send and receive private messages to friends within each network. If the social network offers other functionality through its API (*e.g.*, obtain the list of a user's friends or a list of users in a group), these may also be implemented and exported through the middleware to applications.

3.3   Authenticatr API

Table 1 presents the API that *Authenticatr* exposes to applications. The API consists of two functions for authentication and two for data transfer. The authentication functions are: 1) `login`, to log on to a social network such as Facebook or Google Talk [11] using a credential (such as a username and password combination or a private key), and 2) `set_auth_level`, to set the authentication level of a session handle obtained using `login` to a specified value. The data transfer functions, analogous to Berkeley sockets functions, are `send` and `recv`. These functions require a session handle (obtained using `login`) and the identifier of a friend in the social network with whom to communicate with. A successful `send` call places the message in the Inbox (message storage provided by the social networking site) of friend `f`. The friend, `f`, can then call `recv` to retrieve the message.

The messages sent using the data transfer functions are opaque. The middleware layer performs two basic operations: 1) Given a handle to a social network ("`network *n`" in the `login` API), *Authenticatr* locates the appropriate so-

cial network implementation and obtains a session handle using the credentials provided by the application; 2) It maintains a mapping between `session` tokens (returned by the social network implementation on a successful `login` call) and the corresponding social network sessions (*e.g.*, using HTTP cookies). On a `send` or `recv` call, it forwards the messages to the implementations to send and receive messages on that social network. Applications that use the middleware API can send messages for various purposes, such as negotiating a cryptographic key, or exchanging IP addresses of the machines the applications are running on to initiate direct (peer-to-peer) connection (similar to STUN [8]).

Auxiliary functions. Because the middleware passes the friend identifier `f` unchanged to the corresponding social network, the application using a `send` or `recv` may need to understand the semantics of mapping a user's real identity to that user's corresponding social network identity (*i.e.*, on one network, a user may be uniquely accessible using her email address while on another, the unique identifier may be a number). To tackle this problem, the middleware also exports functions such as `get_friends`, which retrieves a list of a user's friends from the corresponding social network. Because application-specific details are encapsulated within the `friend` structure, applications can continue to treat users on all social networks equally. Other examples of auxiliary functions include `get_groups` (which retrieves a list of communities a user is a member of) and `get_profile_data` (which retrieves the profile data of a user).

Using the data transfer and auxiliary functions provided by the middleware, developers can construct more complex message types for specific applications to use, such as "private message" (sent to a single friend), "FoF message" (sent to all friends and friends-of-friends), "group message" (sent to all members of a group), or "broadcast message" (sent to all friends).

3.4   Establishing the Channel

For each *session* token, the middleware stores the following information[3]:

1. The access details of the social network, such as the IP address and port.

2. The current authentication level of the session token.

3. The pointer to the set of functions implementing authentication and data transfer functions for this social network (registered as a callback function), with at least one implementation for each data transfer function (`send` and `recv`), as well as the auxiliary functions. Unsupported functions point to NULL.

4. A set of process identifiers that have rights to invoke operations (*i.e.*, data transfer or authentication functions) on the session token.

When middleware allocates a `session` token upon a successful `login` call to a social network, the process that presented the credentials for login receives ownership to the token. Session tokens are valid across all processes on a host. If a process wants to allow other processes (*e.g.*, child

---

[3]Because the middleware must store information regarding open `session` handles of all processes, we expect that it will be implemented as a daemon that handles all such function calls.

Prototype	Purpose
`session* login (network *n, credential *cred)`	Attempts to login to network `n` with credentials `cred`, returning a `session` handle
`set_auth_level (session *s, auth_level *level)`	Sets the authentication level for the social network session `s` to `level`
Communication API	
`send (session *s, friend *f, message *msg)`	Sends the opaque message `msg` to friend `f` through the social network session `s`
`recv (session *s, friend *f, message **msg)`	Receives the next message from friend `f` through social network session `s` into the opaque buffer `msg`
Auxiliary Functions	
`get_friends (session *s, friend *f, friend **l)`	Get the list of friends of a user `f` as the list `l` using the social network session `s`

Table 1: API for *Authenticatr*'s middleware layer, both for authentication to a social network, and for data transfer through the social network.

processes) to perform calls using this token, it must ensure that such processes do not perform unauthorized operations, because the same token is usable across all processes on the host. Thus, *Authenticatr* provides authentication levels (set using a `set_auth_level` call) to control which processes on a given host can use each `session` token, and how each process can use each token.

In this model, a user could then provide credentials to a trusted application that performs the login, creates a `session` token, adds read and write permissions for the untrusted application's process ID, and passes the token to the untrusted application process. Similarly, authentication levels allow trusted applications to impose access restrictions, such as read-only or write-only restrictions, on other processes, even though the underlying social network does not provide this feature.

The set of authentication levels must be applicable to most social networks while still allowing expressive access control policies. For each `session` token, the middleware maintains *a pair* of 2-bit masks each for two permission settings: 1) *read* permission allows reading from the social network session (*i.e.*, reading content through the social network, such as messages on a user's Facebook Inbox); and 2) *write* permission allows writing to the session (*i.e.*, sending a message using the social network to a friend). The authentication level is represented as the concatenation of two such bitmasks. The first bitmask is for processes that are in the *owner* group, and the second is for processes that are in the *group* group. The middleware also maintains, for each `session` token, two lists of process IDs that belong to each group (*i.e.*, processes that are members of *owner* and processes that are members of *group*). Modifying and enforcing the authentication levels work much like UNIX file permissions: only process IDs that are members of the *owner* group can change the authentication level bitmasks or add other processes to the *owner* or *group* lists.

4. USING AUTHENTICATR

Authenticatr assumes that the social network implementations allow users to exchange private messages over an authenticated social link. Host-based applications can make *automated* use of *Authenticatr* either for sending secure messages between peers through these links or use the social link as an out-of-band signaling channel to negotiate or set up parameters for direct communication. *Authenticatr* is most

useful to external applications as an authenticated out-of-band signaling mechanism for small messages. These messages can allow host applications to perform important functions such as discovering the IP address and port of a service operated by a friend, and establishing a cryptographic session key. We describe how *Authenticatr* can be used to perform these functions in the context of a peer-to-peer filesharing application.

4.1 Secure Peer-to-Peer Filesharing

Consider a user who wishes to securely share a file with a group of friends using a peer-to-peer filesharing application (*e.g.*, BitTorrent). The application must first provide a way for the set of friends to discover and join the P2P network; in BitTorrent, this process involves using a third-party "tracker". In addition to performing authentication at the discovery service, the application must also secure the communication between peers. Thus, the filesharing application requires two steps: (1) A mechanism to securely disseminate the location and authentication parameters of the discovery service to the desired group of friends, and (2) A way to set up per-peer keys for encrypting communication between peers. These tasks are akin to creating an independent social network, where step (1) is equivalent to logging in securely at a well-known social network URL (*e.g.*, www.facebook.com), and step (2) can be accomplished using private messaging between friends on the social networking site. *Authenticatr* allows the P2P application to use existing social networking services to perform these tasks. We discuss these two steps below.

Step 1: Address and service discovery. Many Internet users are highly mobile or have access only to dynamic IP addresses, and locating friends' computers (for filesharing, voice or video chat, etc.) can be difficult. Social networking Web sites and instant messaging services—many of which already track all logged-in users—can be used by friends to "meet in the middle", and exchange information in order to initiate direct connectivity (*e.g.*, the publicly accessible port numbers of the hosts and ports where inbound access is allowed). Existing techniques such as STUN [8] or UDP hole punching also discover this information; the social network acts as a point for securely exchanging the information.

Implementation. Each host that wants to advertise a service first discovers its public IP address, port, etc. using services such as STUN. It then logs into one or more social

networks, uses `get_friends` to enumerate a list of friends on each network, and uses `send` to send the details of the service as private messages to one or more friends. Applications run by users who receive the private message periodically call `recv` and directly connect to the IP hosting the service.

Step 2: Negotiating cryptographic keys. Although address discovery can use the security of the social network, the actual data exchange in a filesharing application must use the open Internet. Fortunately, peers can use private messaging within the social network to establish pairwise cryptographic keys, which are in turn used to encrypt their direct communication. Private messaging is ideal for key exchange because message sizes are typically small, and only a few round trips are required to complete the exchange. The established secret can be used to secure direct communication between the peers.

Exchanging keys using secure private messaging services provided by a social network entails additional benefits: typical key exchange mechanisms such as Diffie-Hellman are susceptible to man-in-the-middle attacks because not all routers through which the messages pass are trustworthy. Using a secure social channel implies that the user only has to trust the social network provider, which greatly reduces the chances of man-in-the-middle attacks.

Implementation. Each pair of hosts that want to establish a shared secret first log on to the same social network. In an initial series of "hello" messages, the hosts establish the prime number and base for the Diffie-Hellman key exchange. The key exchange completes using two more messages in one more round trip. Note that if the prime and base are well known, the hosts do not need to wait for each other to compute the secrets (*i.e.*, the hosts do not need to log in to the social network at the same time).

4.2 Additional Uses of Authenticatr

Collaborative measurement and monitoring. Troubleshooting network problems and identifying root causes requires measurements from many vantage points. End-users are usually the quickest to notice a problem (*e.g.*, no route to a host, high packet loss or latency, etc.), but they usually lack a method to conduct measurements from multiple vantage points. *Authenticatr* could allow a group of users to run measurement services (*e.g.*, traceroute, ping, bandwidth measurement, etc.) on their machines to which their friends on the social network can make queries.

Implementation. Because many network failures are transient, in the case of network measurements, the user initiating the measurement first enumerates all of his *online* friends that have the measurement application installed (*e.g.*, using a `get_profile_data` query that lists the available *Authenticatr*-based applications for each user). If the measurement queries and responses are small (*e.g.*, traceroute), they may be exchanged directly using the social network messaging facility. Otherwise, *Authenticatr* first establishes a session between each pair of hosts (as above), and communication proceeds directly between the measurement applications.

"Bridging" users. Most existing social networks lack a method to verify user's identity, which can create a threat of impersonation. For two users who are unsure of the other's real identity, having a friend in common could allow that friend to act as the "bridge" and vouch for each user to the other. The same idea can also be extended across social networks: A user who is registered on two social networks (say A and B) can voluntarily act as a bridge between a friend on network A and a friend on network B. Through the bridge user, the two distrusting users may conduct socially authenticated communication (*e.g.*, set up session keys), even though they are not friends of each other (or even on the same network).

5. PRACTICAL CONSIDERATIONS

This section discusses various practical and security considerations with *Authenticatr*. In particular, we discuss how applications need to change to use *Authenticatr* and how *Authenticatr* might multiplex multiple communication channels over a single login session. We also describe how *Authenticatr* might be used as a vector for malware, and outline possible defenses.

5.1 Changes to Host Applications

The change required to existing applications is very minimal, and only the process of obtaining authentication information is different. For example, consider a password-protected P2P directory. To incorporate social authentication, the P2P application needs to: (1) Handle user input of credentials to log into a social network. This includes the name of the social network in question, and the username and password; (2) Get a list of friends of the user, and disseminate the login and password for the P2P share (using `send`) to the group of friends selected by the user. The rest of the application can function normally, allowing access only to remote hosts that present the correct password. On the remote end, the clients that connect to the P2P share must similarly incorporate functionality to search a user's Inbox for credentials. Thus, instead of the remote user typing in the password of the P2P share, the P2P application automatically retrieves the password by virtue of the user's social connection to the user hosting the P2P share.

5.2 Session Multiplexing

Most social networks allow only one login session per credential from a given host. In *Authenticatr*, however, multiple host processes may have to simultaneously use the same social link to communicate with different remote processes on another host. However, the message store (the Inbox) in a social network behaves like a *single* message pool: without any serialization, a scenario with multiple readers and writers may result in messages being retrieved by the wrong host processes.

To tackle this problem, we suggest that these processes use a library built upon the middleware layer to "broker" their requests and responses. The library is responsible for appending identifications to each outgoing message based on the process that sent the message, and mapping identifications on incoming messages back to the processes that are supposed to receive the message. These extra fields are attached to the message body and require no support from the middleware. Because the middleware treats messages as opaque objects, applications (or libraries) can impose arbitrary structure on message bodies to achieve diverse end-to-end functionality (*e.g.*, complex authentication schemes, remote procedure call functionality, etc.).

5.3 Malware

Authenticatr securely connects host-based applications using social links. These connections could be used to spread malware infections, especially if a participating host is compromised. With the basic assumption that the middleware is not compromised (*i.e.*, if it is part of the trusted code base of the operating system), we can show that the risk of malware propagating across hosts is low.

Consider the case where the compromised process on a host is *not* the application that is communicating using a social link. In this case, unless the application explicitly grants rights to the malware process to read/write through the social communication channel, the malware cannot know the parameters required to connect to a remote host (*e.g.*, IP address, password, etc.), and hence cannot copy itself to the remote host. Now suppose that the application— say a peer-to-peer file sharing application—communicating with a remote host using a social link becomes compromised. In the worst possible scenario, the malware will propagate to the remote host. Unless the remote host executes the malware, it will not propagate further. Even if the remote host executes the malware, because the social link is used already used by the file transfer application is already, the malware cannot exploit the channel to gain access to a new host.

6. RELATED WORK

Many products and Web-based services have begun offering functionality beyond merely Web-based communication. Instant messaging networks such as Google Talk [11], Yahoo IM [6], and MSN [5] have supported file-sharing between friends for a long time. Pownce [14], a recently set-up social networking and micro-blogging site, supports sharing files with groups of friends. Researchers have also created host-based applications for file-sharing and backup [4] using contacts downloaded from sites such as Facebook. While all these applications are similar in spirit to *Authenticatr*, they are all programmed to accomplish a task on a single social network. Following the *Authenticatr* framework allows an application to use *any* underlying social network to accomplish a task, and even lets applications *bridge* between two social networks.

Previous work has attempted to unify access for social networks; a prominent effort on this front is Google's OpenSocial framework [13]. The main difference between OpenSocial and *Authenticatr* is the scope: OpenSocial attempts to unify *web-based* application frameworks for various social networks, while *Authenticatr* attempts to unify the basic social network access primitives (*e.g.*, logging in, get a list of friends, send a private message to a friend, etc.) for use by both host-based and web-based applications. Thus, we believe *Authenticatr* is complementary to OpenSocial's direction: OpenSocial may benefit by including the *Authenticatr* API, but *Authenticatr* can function on a social network that does not even have a web-based application framework. Moreover, OpenSocial does not attempt to unify other networks with similar properties, such as IM networks. By providing a unifying framework, *Authenticatr* can facilitate writing application code, because applications do not have to worry about specific access protocols (*e.g.*, HTTP, XML-RPC, XMPP).

SybilGuard [15] is a research effort to defend against sybil attacks (*i.e.*, attacker using multiple fake identities in or-

der to get more resources) using properties of social links between honest users. Although *Authenticatr* tackles a different problem, it relies on social links having significant authentication value.

7. CONCLUSION

As both social networks and the applications that rely on them (and other types of trust or authentication) gain popularity, we believe that the time is ripe to design a framework using these social networks (*e.g.*, Facebook) and communications tools that have social networks embedded in them (*e.g.*, Google Talk) as a substrate for authenticating communication in other applications.

Applications and social networks have evolved largely in parallel, resulting in a situation where each application must reinvent the wheel to authenticate users, leaving social networks—and more importantly, the trust associated with the social links—largely untapped. We aim to rectify this situation by presenting *Authenticatr*, which uses the trust that is inherent in the links in social networks to construct authenticated, out-of-band communication channels between corresponding users. We have described the design for *Authenticatr*, including an API, and we have described various applications that might be built on top of *Authenticatr*. In our ongoing work, we are implementing *Authenticatr* to test its suitability for our motivating applications.

Acknowledgments

REFERENCES

[1] Top 10 social networking sites see 47 percent growth. http://socialsoftware.weblogsinc.com/2006/05/17/top-10-social-networking-sites-see-47-growth/, 2006.

[2] Social networking goes global. http://www.comscore.com/press/release.asp?press=1555, 2007.

[3] Billmonk: Easily Split Bills With Roommates. http://www.billmonk.com/, 2008.

[4] Friendstore: Enjoy Free and easy online backup with your friends. http://www.news.cs.nyu.edu/friendstore/, 2008.

[5] MSN Messenger. http://webmessenger.msn.com, 2008.

[6] Yahoo! Messenger. http://im.yahoo.com, 2008.

[7] Google, Inc. Google Social Graph API. http://code.google.com/apis/socialgraph/.

[8] J. Rosenberg and J. WeinBerger and C. Huitema and R. Mahy. *STUN - Simple Traversal of UDP Through Network Address Translators*. Internet Engineering Task Force, Mar. 2003. RFC 3489.

[9] Y. Sovran, A. Libonati, and J. Li. Pass it on: Social Networks stymie censors. In *7th International Workshop on Peer-to-Peer Systems (IPTPS 08)*, Feb. 2008.

[10] Facebook. http://www.facebook.com, 2008.

[11] Google Talk: A Google Approach To Instant Communication. http://talk.google.com/, 2008.

[12] OpenID.net. http://openid.net, 2008.

[13] OpenSocial - Google Code. http://code.google.com/apis/opensocial/, 2008.

[14] Pownce: Send stuff to your friends. http://www.pownce.com, 2008.

[15] H. Yu, M. Kaminsky, P. B. Gibbons, and A. Flaxman. SybilGuard: Defending Against Sybil Attacks via Social Networks. In *Proc. ACM SIGCOMM*, Pisa, Italy, Aug. 2006.