The Architecture and Implementation of a Decentralized Social Networking Platform

Seok-Won Seong Matthew Nasielski Jiwon Seo Debangsu Sengupta Sudheendra Hangal
Seng Keat Teh Ruven Chu Ben Dodson Monica S. Lam

Computer Systems Laboratory
Stanford University
Stanford, CA 94305

Advertisement-supported social networking portals generally aim to lock in users' data and exploit personal information for ad targeting and other marketing purposes. Because of the network effect, it is not hard to envision a situation where the information of a very large population can end up in the hands of an oligopolistic group or even a sole monopolistic actor. Beyond the obvious privacy concerns, this outcome would clearly pose problems for healthy competition, ultimately harming end-users.

Our overall research goal is to create an open standard and API so that social applications can work on anybody's data, regardless of where the data is stored and where the application is running. Furthermore, we believe that making it easy for users to store their data in a personal "safe haven" will create an environment for exciting new developments in social computing.

This paper presents PrPl, a decentralized infrastructure aimed at letting individuals participate in online social networking without giving up ownership of their data. Our proposal is a person-centric architecture, where a service we call the Personal-Cloud Butler indexes and shares each individual's data while enforcing fine-grained access controls. Due to difficulties that moving to a decentralized model introduces for application developers, it is important that we make decentralized social applications easier to develop in order for our platform to be widely accessible to developers. To this end, we propose a location-agnostic social database language called SociaLite that hides the complexity of distribution and access control while still allowing expressive queries.

1. Introduction

1.1 Ad-Supported Social Networks

To be commercially viable, an advertisement-supported social networking portal must attract as many targeted ad impressions as possible. This means that this type of service typically aims to encourage a network effect, in order to gather as many people's data as possible. Once this is achieved, it is in their best interest to allow users very little control of their data, instead locking this data in to restrict mobility, assuming ownership of it, and monetizing it by selling it to marketers. This social networking paradigm is deeply flawed because it is not designed with the interests of individuals users in mind. Beyond the obvious erosion of personal privacy that this type of service entails, there are also many other deficiencies.

At the societal level, data lock-in has the tendency of creating an oligopoly, or even a monopoly. Social networking is particularly "sticky", because we are compelled to remain in a network to interact with our friends. It is inconvenient to have to rebuild our social graph and upload our personal content for every application provider, limiting our willingness to move to a better provider as one would expect in a competitive market. When there is a lack of competition, it goes without saying that the consumers suffer, and it is clear that proprietary and closed platforms give the owners the right to limit competition. For example, Apple has strict regulations on the kinds of applications that can be run on the iPhone, and has used these to justify locking out potential competitors. Even if we do manage to use more than one provider, it is harder to manage our data which is now scattered in different data silos owned by different application portals. For example, we may put our pictures up at Flickr to share and Shutterfly to print. Furthermore, the centralization of data and computation requires a heavy infrastructure investment, creating a barrier to entry and again limiting user choice in a market with few competitors.

It is alarming how much intimately personal information some people (particularly those belonging to generations that have grown up using online services) are willing to divulge on these portals. Beyond basic privacy issues, the difficulty of turning down friends and the lack of good access controls are some of the biggest causes of concern. For example, multiple incidents of job loss as a result of employers gaining access to private information shared on social networks have been reported. Even ignoring the potential for this type of accidental sharing, it is hard to ignore the fact that today's social networking portals either

claim full ownership of all user data through their seldom-read end user license agreements (EULA), or stipulate that they reserve the right to change their current EULA without any notice to the users (in effect, meaning that they could retroactively claim ownership of the data at any time in the future). Given these facts, it borders on absurd that we leave the stewardship of all this personal data to an enormous and unaccountable company; public outcry would be to no avail were such a "big brother" company to fail and need to sell its data assets. By amassing large amounts of private data in one place, we are not only running the risks already mentioned, but we are also creating an opportunity for large-scale fraud. Like any large collection of valuable information, it would be the target of hackers, crooked employees and malicious organizations.

1.2 Decentralized, Open, and Trustworthy (DOT) Social Networking

Our overall aim is to create an environment where everybody can participate in online social networking without reservations. Today, many are not participating or participating to the fullest extent because of privacy issues. Specifically, we wish to create a social networking platform that has the attributes described below.

Decentralized across different administration domains. This allows users who keep data in different administrative domains to interact with each other. Users have a *choice* in services that offer different levels of *privacy*. They may choose to store their data in personal servers they own and keep in their homes, they could keep their data with stored vendors, or they may choose to use free ad-supported services. Or, they can keep data in a variety of these places.

Open API for distributed applications. We aim to create an API that allows a social application to run across different administrative domains. For example, we can write an application that allows users in two different social networking sites to interact with each other. The current model in which only users belonging to a common site may interact is just as unacceptable as disallowing users on different cellular networks to call each other.

Trustworthy interactions with real friends. It is of absolute importance that there is a safe haven where we are comfortable with keeping the most private of information, such as our history of GPS locations or the list of our closest friends. Such a safe haven would enable the development of many more personal utility applications as well as social networking applications. It is important that we can share this information with our friends, in a easy and natural manner. True social networking means that we share personal information with each other, more information with close friends, and less with acquaintances. The information should be gathered, *in situ*, as they are generated. For example, we can keep track of what songs we like the best by keeping statistics of the songs we play.

1.3 Contributions of this Paper

We have created an architecture called PrPl, short for Private-Public, as a prototype of a DOT social networking system.

Personal-Cloud Butlers. We propose the notion of a Personal-Cloud Butler, which is a personal service that we can trust to keep our personal information, it organizes our data and shares them with our friends based on our private preferences on access control.

A Federated Safe Haven for Personal Data The Butler provides a safe haven for all our personal data, from emails to credit card purchases. To take advantage of freely available data storage on the web, the Butler lets the user store (part of) their data, possibly encrypted with different storage vendors. It provides a unified index of the data to facilitate browsing and searching and hands out certificates that enable our devices and friends to retrieve the data directly from storage.

Decentralized ID and Server Management. Our system allows users to use their established personas by supporting OpenID, a decentralized ID management system. We propose extending the OpenID system so that the OpenID provider supports the lookup of a designated Butler with an OpenID. The OpenID provider thus becomes the root of trust for the authentication of the Butlers.

A Social Multi-Database. The Butlers form a network of a decentralized, social multi-database, with each Butler in a separate administrative domain. A query from a friend may trigger a cascade of queries through our friends' Butlers. To support interoperability, we represent our data in a standard format based on RDF and standard ontologies whenever they are available.

The SociaLite Language. We have designed and implemented a declarative database language called SociaLite for trusted queries into the PrPl social multi-database. Supporting composition and recursion, this language is expressive enough so many of the social applications can easily be written by adding a GUI to the result of a SociaLite query. Details about network communication and authentication are hidden from the application writer.

Access Control to Social Data. SociaLite makes it possible to express intricate access controls of personal data in an extensible way. These access controls are automatically enforced on friends' queries using a rule-rewriting system. This simplifies the implementation of the queries, prevents attacks, and also protects against innocent mistakes.

Experimental Results. We have implemented a fully working prototype of the SociaLite language and PrPl infrastructure as proposed in this paper. Applications were developed along side of the infrastructure so as to drive the design. At this point, we have created quite a rich set of applications on the PrPl platform. We have developed an email mining application called Dunbar that uncovers many interesting personal information. For example, it creates an ordered contact list according to the strength of tie inferred

from the email communication. This information can be exploited by any PrPl application to give users a better experience. We have developed a PPLES client on the Android phone that lets us browse selected friends GPS locations and photos. We have also created a music client called Jinzora Mobile for both Android and iPhone that is used by members of the group daily. This client streams music from our own and our friends' personally hosted music servers and lets users share playlists stored in their respective Butlers. PrPl applications are relatively easy to develop because they consist mainly of GUI code wrapped around SociaLite queries. We have also performed some preliminary measurement of our prototype. In a simulation of 100 Butlers on the PlanetLab, first results of distributed queries arrive within a couple of seconds. Given that the prototype is unoptimized, the results suggest that this approach is technically viable.

1.4 Outline of the Paper

We first describe an overview of the design and rationale of the system in Section 2. We then describe the various components of the infrastructure: the federated identify management subsystem, the PrPl index and data management, and finally SociaLite language and access control in Sections 3, 4, and 5, respectively. Section 6 describes our experimental results: the implementation of the infrastructure, the experience of the social networking applications based on the infrastructure, and finally measurements from running queries over about 100 PCs. We describe related work in Section 7 and conclude in Section 8.

2. System Design and Rationale

This section describes the user experience that we wish to provide and the high-level design rationale of the system.

2.1 Safe Haven and Cloud Services of Personal Data

Rationale 1. Allow users to host their own private cloud services for themselves and their friends with home Butler services.

Even though a lot of data are being put up in the cloud, fundamentally most people are still storing their data on the hard drives of their personal computers at home. The advent of smart personal phones provides another major impetus for people to put their data up in the cloud. Fast as broadband is, our demand for storage grows even faster, but fortunately, storage is highly affordable. Having the home server at home is economical and desirable. Keeping the data at home also has the benefit that it exploits locality of reference. Most of the the time, you will be browsing your photos in high resolution on your big-screen TV. Finally, having the physical device at home provides the ultimate privacy. Even if we host our data encrypted with a storage vendor, we need to decrypt the data to index.

The Personal Cloud Butler should be an appliance that does not need end-user maintenance and upkeep. We imagine that an ISP may offer the Butler service along with its broadband gateway, for a few more dollars a month. An ISP may also keep a fully encrypted, thus passive, copy of your data as a backup for additional fees. Examples include the Tivo or the game consoles. There also exist more recent low-power products like the Pogoplug that allows users to host their services.

Note that the Butler is not just making the data available, but also providing cloud services. Having our music indexed is not that useful, however, streaming our music to our mobile devices is of great value. Just like we have PCs today, we imagine the average household will have a personal server (PS) running a Butler and associated services. We imagine that there will an appstore that we can go to buy services we wish to host, just like there is an appstore for mobile devices today.

2.2 Leveraging Existing IDs and Storage

Rationale 2. Leverage existing personas and storage in the cloud by creating a searchable index to the federation of storage on the web.

We see PrPl as an alternative for storing and offering the private information we have. It does not replace many of the services out there on the Internet right now. We may choose to create and publish our public persona on Facebook or MySpace, which is no different from University researchers having a public web site at their institutions. It is important that the Butler knows our private and public data as well as our personas and our friends' personas. We have adopted OpenID as a way of managing our identities; not only is our Butler an OpenID provider itself, it also accepts log-ins from our friends via their OpenID and Facebook accounts, thus they do not have to create yet another account just to interact with our system.

Similarly, we wish to leverage the many free or low-cost data storage and services available by incorporating the contents of the information in our Butler services, without copying all the data over. Our solution is to create a federated data service and an index, called the *PrPl Index*, to the data stored with the Butler and outside. This index is like a cross between a data base and a semantic file system. This includes personal information such as our relations, devices, calendar, contact information, etc., as well as meta-data associated with large data types, such as photos and documents. The meta-data, kept in an RDF (Resource Description Framework) format, includes enough information to answer typical queries about the data, and location of the body of the larger data types, known as blobs. Blobs can be distributed in remote storage, and possibly encrypted. Data are identified by a PrPl URI (Uniform Resource Identifier). Each storage device in the PrPl federated data service runs a Data Steward (DS). The DS serves blobs to applications directly when presented a valid blob ticket and updates the PrPl semantic index if blobs held locally change.

2.3 Data-Centric Rather Than Application-Centric

Rationale 3. A safe haven of private data lets us pool all our data, including the very private ones, in one place logically, which we can leverage to create new types of programs.

The mobile device is going to make available even more personal information, such as daily GPS traces, all our phone logs, SMS, email, credit card purchases, and more. Having all the data together lets us answer questions that would otherwise not be possible had the data been independently stored in different silos offered by various applications service providers.

2.4 Open API for a Social Multi-Database

Rationale 4. An open API that makes it easy to develop trustworthy social applications across Social Multi-Databases can encourage independent software vendors to create many attractive social applications.

While our personal data may be interesting in itself, adding a social context exponentiates the types of applications we can build. The social functionality comes as an extension to our database language, allowing developers to build queries that run across friends' databases to achieve interesting results. The following are some interesting social queries.

"Are any of my friends near where I am right now?" We wish to eliminate the need to browse more data than necessary to get to the answer we are interested in. In this case, there is no need to first find out where everybody is and then filter it down to the people we are interested in. Similarly, suppose we have a GPS trace and photos that only have time stamps on them, we can easily find pictures taken at a certain location.

"Show me pictures of the retreat as they were taken." We wish to treat data from different parties as a collective, sort them, and display them chronologically etc.

"Suggest a restaurant choice based on the preferences of my closest friends who are available and their current locations". We can compute an answer based on a combination of different kinds of data, which is not possible had information like locations and calendars been trapped separately with different applications service providers.

Many existing social applications are just a matter of combining the result of a simple query with a GUI. By supporting a query interface, we can enable many more interesting applications. We have developed a distributed query language called SociaLite. The language is based on Datalog, a declarative database language that supports function composition and recursion. The example queries above can be written succinctly in just a few Datalog rules. The following two lines defines the recursive friends of friends (FStar) relationship.

```
FStar(?u,?p) :- Friend(?u,?p).
FStar(?u.?p) :- FStar(?u,?x), Friend(?x,?p).
```

Each Datalog rule defines a predicate in terms of a conjunction of predicates. Here, p is a FStar of u if p is a friend (the first line), and p is a FStar of u if x is u's FStar and p is a friend of x (the second line).

We use an ACCORDING-TO operator, denoted by "[ ]", to allow the query writer to treat the multiple databases in different Butlers as one database. For example, the query

```
FStar(?u,?p) :- FStar(?u,?x), Friend[?x](?x,?p).
```

says that a person p is u's FStar if x is a Fstar and, "according to" x, p is a friend of x. The implementation of SociaLite automatically contacts the friend's Butler for his friends' list of friends.

2.5 Access Control to Social Data

Rationale 5. Provide a flexible and general way to express access controls over one's social data and enforce it without relying on the cooperation of third-party software.

Ad-supported sites, motivated primarily to increase their user base, pay little attention to access control today. Facebook, for instance, only provides a small number of canned access control settings to the user. It is also an application platform provider, but all applications we run have the same privilege as the account holder. In other words, no one who has chosen to restrict access to their data in any way should run any third-party application. Access control ideas presented here are relevant to all social networking services, be they centralized or decentralized, and especially if the services sport an application platform.

Access control over social data is not just a matter of letting users or groups read or execute a file, as in the case of Unix file systems. One may even need to transform the data before releasing it. For example, Loopt, a company that allows users to share GPS locations, found that some users demanded the ability to not share their GPS locations if they happened to be in a certain neighborhood, and even to lie about their current location. The latter was found to be necessary because wives in an abusive relationship may not have the choice to turn off their GPS.

Our SociaLite language makes available to the user the full power of Datalog for expressing access control policies. These access control rules are then composed with external queries to create an efficient query against the database. Moreover, we have developed a rewrite system that automatically enforces access control without relying on the cooperation of third-party software vendors. This is necessary because even signed applications from trusted vendors may contain errors that render them vulnerable, as demonstrated by SQL injection attacks on web applications.

SociaLite makes possible subtle and powerful access control that lets a query operate on raw data while controlling the information eventually returned. For example, suppose we wish to share our GPS locations only with our family if we happen to return to our home town in Korea. This query:

```
Currloc(?l, $r) :- CurrLoc(?l), $IsInKorea(?l),
                   ($r prpl:memberOf Family).
```

says that requester r can get access to the current location, l, if the current location is in Korea, and r is a family member.

Beside expressibility, extensibility, and automatic enforcement of access control, usability of the access control features is also extremely important. However, usability is beyond the scope of this paper. Our approach is to first provide a general and powerful framework to facilitate experimentation. We intend to create intuitive user interfaces for common use cases as they emerge.

2.6 Economics, Efficiency and Scalability

The study of decentralized social networking is worthwhile even if it does not become widely adopted because such research brings awareness to the underlying issues. By providing alternatives, it may challenge centralized providers to respond with equivalent features. Some of these technologies we discuss, such as access control, are directly applicable to centralized implementations as well. Indeed, major web service providers are themselves finding the need to decentralize as they scale out from one site to many.

So ultimately, is decentralized social networking just a pipe dream? Is it financially feasible? We are hopeful that the answer is yes, for the following reasons.

First, the super-giant star topology in large portals dictates an expensive infrastructure. For example, Credit Suisse estimated that YouTube may be losing over $300 million per year[1]. Especially for personal information that is shared between a small number of individuals (such as the numerous baby videos shared on YouTube) a distributed topology is more scalable. Throughput drives the design of centralized web servers. In a distributed context, individuals with PCs can easily afford the computation and networking cost for personal services. Distributed processing certainly takes longer, but in the social context, we only have to compete with how long social interactions usually take. Furthermore, it has been postulated by the social science community that people can maintain a relatively small number of stable social relationships. The limit, known as the Dunbar's number, is commonly believed to be approximately 150.

Second, while decentralized social networking does not seem to support advertisement-based models at first glance, it may eventually provide an even better marketing opportunity, allowing the data owner full participation in terms of financial rewards while preserving the privacy of the most sensitive information. Our safe haven of personal information is a marketer's dream because it has all the information about the user's interests. For example, it may contain not just the purchase history from a single site, but history across all stores, online and offline. We advocate a model where advertisers run applications on users' machines. With our access control policy enforcement, application may access the personal information during the computation but only export information they are explicitly allowed to. For example, a department store may broadcast all the sales items, while an application running on a cell phone can determine which sales items are most appropriate and display those to the end user, without sending personal information such as whose birthday presents the user is buying.

Finally, we are encouraged by the history of how the closed, walled garden of AOL failed to compete against the forces of the open Internet. The need for people to interact and share is so fundamental that we remain hopeful that there will eventually be an open infrastructure, where people can interact freely with whomever regardless of the vendor they choose.

3. Federated Identity Management

The PrPl system utilizes federated, decentralized identity management that enables secure logins, single sign-on, and communication among applications in which Butlers belong to different principals in the system. The overall goal is to enable PrPl users to reuse existing credentials from multiple providers and avoid unnecessary ID proliferation. Requirements for our identity management include authenticating users to Butlers, registering Butlers with the Directory Service, third-party service authentication, and authentication between Butlers and applications. To this end, we chose OpenID due to its position as an open standard (in contrast to Facebook Connect [2]), extensive library support, availability of accounts, and the ability to extend the protocol easily for PrPl's needs.

Conceptually, an OpenID handshake or login consists of the following steps: a) Requester enters his OpenID identifier at a Relying Party (RP)'s web page. b) RP performs YADIS/XRI discovery protocol [3] discovery on the identifier, fetches an XRDS file [18] that encodes his OpenID Provider (OP)'s, and redirects the user to an acceptable OP. c) User successfully enters credentials at the OP, which verifies and redirects the user back to the RP along with a signed success message in the HTTP headers. d) RP verifies the result with the OP and welcomes the requester.

3.1 Distributed Butler Directory Service

Rather than relying on the commonly-used centralized directory services of the OpenID network, we have implemented a distributed Butler directory service for added robustness. We do so by extending the Butler OP to include a pointer to the user's Butler. This can be done by exposing a "user-butler" element in a user's XRDS file.

In order to register a Butler with its Directory Service, the owner authenticates himself to the Directory Service using OpenID as described above. Post-authentication, the owner submits the registration package he gets from the Butler. The package contains the Butler's public key, a mapping from the owner ID to unique Butler ID, a mapping from the Butler ID to the Butler's URL, and a HMAC of the mappings for verification.

We propose that the OP associated with a Butler also serve as its Certificate Authority (CA). The OP provides a digital certificate for the Butler's public key. The Butler can

now present this certificate as proof that it is the registered Butler for the associated OpenID during inter-butler communications.

The discovery process works as follows. Anyone who knows an owner's e-mail address can identify their OpenID identifier using a "well-known location" [4] or using a discovery protocol. Then, a YADIS/XRI discovery is followed to discover an owner's XRDS document, an XML document enumerating his identity providers (IdP), profile information, and extensions like public key and Butler address. By obtaining the Butler's address, we can reach the Butler. This is no different from allowing anybody knowing your e-mail address to reach you. The distributed butler directory service provides one level of indirection for the owner. This means that one can change the hosting of one's Butler by updating the information with their OP.

In our current prototype, since common OpenID providers are not providing the Directory Service yet, we have created our own Global Butler Directory Service for the purposes of testing and bootstrapping. This is a centralized service running at a well known location.

3.2 User Authentication at the Butler

Our recommendation is to configure the Butler to allow anybody to view public information and to leave a message. However, the Butler offers additional services only upon authenticating the guest's credentials. In current architectures, a guest needs to register an account for each service (in different administrative domains) before using it. We leverage OpenID's single sign-on (SSO) properties to obviate this tedious step. The Butler acts as an OpenID Relying Party (RP), and authenticates the guest by contacting his OP. The guest typically does not even have to explicitly enter credentials at each Butler due to the common practice of staying signed in to popular web e-mail services that are OpenID providers. Subsequent accesses to the same Butler are entirely seamless due to the use of HTTP cookies.

While the above mechanism works well for web applications, we also support native guest applications. The native application contacts the Butler to obtain a temporary authorization token, which it passes on to the login screen of the default system browser. This browser can be embedded into the application itself for better performance, with no loss of security. After the OpenID handshake, the Butler creates a session ticket and maps it to the authorization token. The application reverts to native mode and exchanges the authorization token for a session ticket, which it uses for subsequent requests.

3.3 Authentication with Third Parties

In the decentralized PrPl architecture, a distributed query means that the Butler needs to contact other Butlers on behalf of a user, and it is necessary to present the user's authentication known to the initiating Butler to the peer Butlers. To support single sign-on, our system uses a PrPl Session Ticket.

For inter-Butler communication, the Butler's certificate can be used to identify the owner; it is not necessary for the owner to present his OpenID credentials. Authenticated users of a Butler can also initiate queries. A query may be propagated through zero or more Butlers to gather the requested information. The PrPl Session Ticket always identifies the last Butler that is sending the request. The receiving Butler can verify the identity of the Butler through the signature also included in the session ticket. The ticket also includes a requester ID field that specifies the originator of the message and all the intermediate Butlers involved. (Note that this information is just advisory and not verifiable. This is not unlike the social phenomenon of someone seeking medical advice for his condition claiming that he is asking the information for a "friend".)

Specifically, a PrPl Session Ticket consists of a tuple $\{$*issuer ID, requester ID, session ID, expiration time, issuer's signature*$\}$. A ticket can be renewed before its expiration time. Upon a request for renewal, the issuing Butler will issue a new ticket with the same session ID by updating the expiration time and signature.

4. The PrPl Semantic Index

In this section, we describe the details of the PrPl semantic index presented in Rationale #2. The Butler keeps the index, which is built with the cooperation of Data Stewards; it presents a programmable interface to applications. It also includes a basic management console so the user can administer and access his data over the web.

4.1 Semantic Index API

The PrPl semantic index contains all the personal information as well as meta-data associated with large data types, such as photos and documents. The meta-data includes enough information to answer typical queries about the data, and location of the body of the larger data types, known as blobs. Blobs may be distributed across remote data stores and possibly encrypted. A unit of data in the system is known as a resource. A resource conceptually is a collection of RDF (Resource Description Framework) subject-predicate-object triples with the same subject. Resources have a globally unique URI (Universal Resource Identifier). These resources contain much the same information that one would find maintained by a traditional filesystem such as name, creation time and modification time in addition to keyword and type (e.g. photo, message etc.). Blob resources contain type and size information about the blob and a pointer to the Data Steward that physically hosts the file.

We use standard ontologies whenever possible. The ontology, types of resources and related properties. are described in OWL, the Web Ontology Language[5]. The system has two OWL specifications: the default ontology that describes default user and system resources, and the second, a user-extensible ontology that describes basic data types

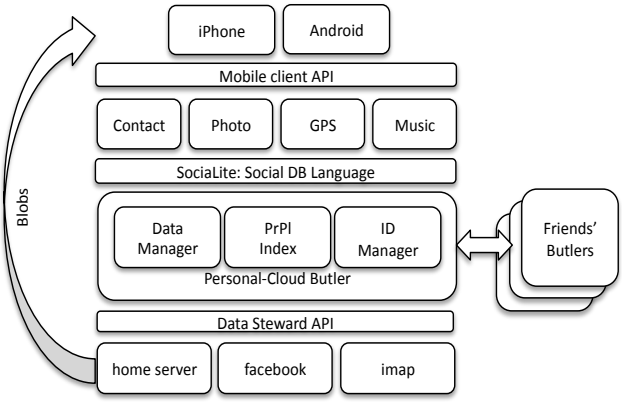

Figure 1. PrPl Data Subsystem

such as Address, Calendar, Person, or Music. The ontology can be used to enforce restrictions on resource properties (e.g. single *last name* property for Person resource) and to generate inferred information from given facts.

We are standardizing on the use of RDF just as an API. We currently store data in RDF triples, but we intend to optimize the implementation by representing important relations in a more efficient representation.

4.2 Data Stewards on Storage Devices

Each federated store that hosts blob data runs a Data Steward, operating on behalf of the user. It provides a ticket-based interface to PrPl applications and hides the specifics about how the blob is actually stored.

At configuration/startup time, the Steward registers itself with the owner's Butler. For existing data sources, the Steward checks any changes since the last communication with the Butler. It periodically sends heartbeats to the Butler with updated device access information, such as when it is running on a portable device with changing IP addresses. For data sources like file systems that may be updated externally, the Steward monitors the resource and sends notifications to the index with its heartbeats. For all resources located in its store, the Steward tracks where the blobs are located. Specifically, it maps a virtual PrPl resource URI to a physical URI, such as one beginning with `file://`.

The Steward services blob access requests from applications directly, to avoid making the Butler a bottleneck. An application is required to first obtain a ticket from the Butler owning the resource. The ticket certifies that the requesting application has access control rights to perform specific operations. The ticket contains the URI of the requested resource, PrPl user, ticket expiration time, list of authorized operations, and one or more locations of the Data Stewards that are hosting the blob. Our current implementation does not support revocation of tickets; however, the ticket is not renewable and a new one must be acquired after it expires.

4.3 Butler Management Console

Each Butler provides an interactive web-based management tool where a user can login to administer and access his personal cloud information. The user can create and remove identities and manage group memberships, view registered devices and services, and run simple queries directly. It also provides a generic resource browser, where a user can edit meta attributes, download blobs, and specify access control policies.

5. SociaLite: a Language for a Social Multi-Database

The following first introduces Datalog, the language on which SociaLite is based, then the five extensions we made: access to the RDF triples in the PrPl index, function extension, the ACCORDING-TO operator for distributed queries, access control in Datalog rules, and a rewrite system for enforcing access control.

5.1 Datalog

Datalog is a query and rule language for deductive databases that syntactically is a subset of Prolog[20]. Deductions are expressed in terms of rules. For example, the Datalog rule

D(w,z) :- A(w,x), B(x,y), C(y,z).

says that "D(w,z) is true if A(w,x), B(x,y), and C(y,z) are all true." Variables in the predicates can be replaced with constants, which are surrounded by double-quotes, or don't-cares, which are signified by underscores. Predicates on the right side of the rules can be inverted. Lastly, a query can be issued like

?- Friend(Alice, ?x)

The results are all the tuples that satisfy the relationship, which in this case is a list of pairs, Alice and a friend of Alice.

Datalog is more powerful than SQL, which is based on relational calculus, because Datalog predicates can be recursively defined[20]. If none of the predicates in a Datalog program are inverted, then there is a guaranteed minimal solution consisting of relations with the least number of tuples. Conversely, programs with inverted predicates may not have a unique minimal solution. The SociaLite language accepts a subset of Datalog programs, known as *stratified* programs[9], for which minimal solutions always exist. Informally, rules in such programs can be grouped into strata, each with a unique minimal solution, that can be solved in sequence.

We choose Datalog as the basis of our language for accessing the PrPl Social Multi-database for the following reasons. First, many of the social applications can be written as a database query with some GUI. Datalog supports composition and recursion, both of which are useful for building up more complex queries, and recursive ones as well as we gather information about, say, our friends of friends. Being

a high-level programming language with clean semantics, Datalog programs are easier to write and understand. More importantly, it avoids over-specification typical of imperative programming languages. As a result, the intent of the query is more apparent and easily exploited for optimizations and approximations. For example, it is more important that we return information about some friends quickly instead of taking all the time to find all friends. The same iterative fixed-point calculation in a Datalog implementation can be easily adapted to provide incremental results early. This same mechanism is useful for implementing standing queries where more and more results are generated as inputs come in. In addition, it has a uniform syntax for referring to the data in the database (extensional predicates) or the programs in terms of rules (intensional predicates). This provides the language implementor a simple way to cache the results of a program, save that in the database and reuse them later. Finally, its simple syntax makes it conducive to the implementation of rewrite systems and the like to extend its functions. We shall show in Sections 5.5 how we take advantage this to enforce access control.

5.2 RDF-Based Database

The database in our Butler is unstructured, meaning that relation schemas need not be predefined. This allows us to add new relationships easily. SociaLite provides syntactic sugar for RDF by allowing RDF triples be included as predicates in the body of a rule. For example, we can say that a contact in the PrPl database is a friend:

Friend(?u) :- (?u a prpl:Identity).

(?u a prpl:Identity) is the RDF syntax for saying that u has type "Identity" in our PrPl database. Had we defined a predicate called InPrPlIndex and rewrote (?u a prpl:Identity) as InPrPlIndex(?u a prpl:Identity), the result would have been in proper Datalog predicate.

5.3 Function Extension

In some cases, a social application is simply a matter of presenting the result of a SociaLite query graphically. Often times, however, additional processing is necessary beyond relational algebra, as supported in Datalog. In the general case, we have need a full-blown programming language, and SociaLite is used just as an interface to the database. We can greatly enhance the expressiveness of SociaLite by allowing users to define pure functions as predicates in the body of the rules. This enables more computations to be written in SociaLite so as to take advantage of the features of the language, such as distributed computations.

Consider, for example, Google's Latitude which collects location histories of one's friends and filters them by proximity to one's current location. With function extension, we can write the query like this:

FriendsLocationNearMe(?f, ?l) :-
  FriendsLocation(?f, ?l), Location(?*myl*), $IsNear(?$l$, ?*myl*).

where the predicate $IsNear is a user-defined function, indicating whether the two input locations are close to each other.

Function names are preceded by the dollar ($) sign, and can currently be written in either Java or Python. It has a signature indicating the number of output variables, and the number of input variables that follow, the sum of which gives the length of the arguments in the predicate. For $IsNear, there are zero output variables and two input variables. A user-defined function accepts as input an array, each representing an element in the input tuple; the function may return zero or more arrays, with each array representing the results matching the input tuple. Suppose we are interested in finding the square root of a number. Then, $root(?$r$, ?$s$) returns null if $s < 0$, [0] if $s = 0$, and [2][-2] if $s = 4$. Similarly, $InNear(l1,l2) returns an empty array if l1 and l2 are near, and null otherwise.

Naturally, one needs to be careful when using this type of function within a group of recursive predicates as, due to fixed point semantics, the query may never terminate.

5.4 Remote Queries

The ACCORDING-TO operator introduced in Section 2.4 makes it possible to perform a query across the entire social multi-database as a whole. When used together, recursion and the ACCORDING-TO operator allow one to traverse the distributed directed graph embedded in the social database. Suppose we are interested in collecting all the pictures taken at a Halloween party among our friends. The SociaLite query may look like:

FStar-Halloween(?p,?f) :- FStar(?p), Halloween[?p](?f).
Halloween(?f) :- (?f a prpl:Photo),
  (?f prpl:tag 'Halloween09').

This query gathers together pictures with the same "Halloween09" tag that are in our friends' respective Butlers. Any variable appearing in the ACCORDING-TO operator must be bound in another non-negated predicate on the right hand side of the implication operator. This restriction is necessary because one cannot enumerate all of the Butlers taking part in the system.

5.5 Access Control

We now describe how SociaLite allows the users to specify access control and enforce it automatically as summarized in Rationale 5. SociaLite has two kinds of predicates: *controlled predicates* and *uncontrolled predicates*. All the predicates we have described so far are uncontrolled. Controlled predicates has access control enforced; they are predicates with an extra parameter, $$r$, which is reserved to identify the requester. Friends and third-party software can only define and refer to controlled predicates directly. Different levels of trust can be associated with third-party software. For example, signed queries from your bank may be treated as more trusted requestors. Queries downloaded from the web are treated like they are strangers. The owner can express

controlled predicates in terms of uncontrolled, thus allowing the privilege to be escalated in a controlled manner.

5.5.1 Controlled Predicates

First let us discuss the basic use of Datalog to protect access to the contents of the PrPl index. All uncontrolled accesses to the PrPl index are expressed in terms of subject-predicate-object $(?s, ?p, ?o)$ triples. Accesses are protected for adding a fourth element to the triple that represents the requester. The basic level of protection is given by the SociaLite rule:

(?s ?p ?o \$$r$) :- (?$s$?p ?o), AC(?s \$$r$), AC(?$p$ \$$r$), AC(?$o$ \$$r$).

This says that \$$r$ can access the tuple (s, p, o) only if \$$r$ can independently have access to s, p, and o.

We can implement many different kinds of access control using this general mechanism. Let us illustrate the system with a basic tagging scheme. Suppose we wish to allow r access only if it has tags with the subject, predicate, or object in question. This can be expressed as follows:

AC(?x, \$$r$) :- (?$x$ prpl:tag ?t), (\$$r$ prpl:tag ?t).

For the most part, we expect most users to have access to the predicates, since access control can separately be used to protect the subject or the object. There may be exceptionally sensitive relationships like social security numbers that we may restrict general access.

As discussed in Section 2.5, we may wish to grant users access to higher level functions and not the lower level details. The query discussed earlier

CurrLoc(?l, \$$r$) :- CurrLoc(?$l$), \$IsInKorea(?l),
(r prpl:memberOf Family).

illustrates this point. Note that while CurrLoc appears both in the head and the body of the rule, it is not a recursive call. They share the same name because they serve the same function, and only differ in access control. The special \$$r$ parameter in the former indicates it is a controlled predicate, and the latter is uncontrolled. This rule says that r can access l if the requester r is a family member and the current location is in Korea.

As illustrated above, especially with function extensions, Datalog rules can encode very expressive access controls.

5.5.2 Access Control Enforcement

Access control has to be enforced automatically, we cannot leave it to our friends or third-party software writer. Even if they are not malicious they may make unintended mistakes that can be exploited. For example, Moinmoin is a popular wiki web platform, and hundreds of plug-ins have been written. Third-party plug-ins are supposed to follow the access control convention such as first authenticating the user before executing any operations on the Wiki database. However, it has been found that many of these plug-ins do not follow the convention hence making the system vulnerable to attacks.

It is not sufficient to just ask friends to submit only queries that refer to the controlled predicates in the system. We have to enforce it. We do so with a rewrite system that automatically converts external queries into a controlled form. It also has the side effect of making SociaLite programs easier to write and read. An additional \$$r$ parameter is automatically inserted into every external query's predicates, thus subjecting them all to access control.

That is, a user may request ?- CurrLoc(?l), our rewriter will automatically turn that into ?- CurrLoc(?l,\$$r$), and binding the \$$r$ parameter to the identify of the requester. Thus the SociaLite statement

Halloween(?f) :- (?f a prpl:Photo),
(?f prpl:tag 'Halloween09').

is rewritten as:

Halloween(?f,\$$r$) :- (?$f$ a prpl:photo \$$r$),
(?f prpl:photo 'Halloween09' \$$r$).

thus injecting control into every statement.

In other words, we create a controlled analog for every uncontrolled predicate defined, such that the controlled version is a conjunction of the controlled analog of all the subterms in the body of the rule. We refer to this as the default propagation of control from head to the body.

Owners are allowed to define controlled predicates in terms of uncontrolled predicates, as discussed above. If the user does not explicitly specify a rule for a controlled predicate, one is generated from the uncontrolled predicate by default to save the user from having to define the same predicate twice. The default is to propagate the control of the head to each of the terms in the body as discussed above.

6. Experimental Results

To understand the issues in building a decentralized, open and trustworthy social network, we have been developing applications as our infrastructure has evolved. We have written many applications and have learned from each one, including those we eventually discarded. We now describe the implementation of our infrastructure, our experience with the applications we built, and then some measurements obtained by running a network of about 100 Butlers.

6.1 Implementation of the PrPl Infrastructure

A block diagram of our prototype implementation is shown in Figure 1. Our implementation of the SociaLite language includes a number of optimizations such as semi-naive evaluation to speed up convergence, caching on remote sites to minimize recomputation of the same data, and query pipelining so that partial results can be returned without waiting for all the results to be generated. The details of these optimizations are described elsewhere and are beyond the scope of this paper. The SociaLite implementation makes use of the Jena/ARQ libraries for iterating over triples and XStream for serializing and streaming query re-

sults. The PrPl index is implemented using HP's Jena semantic web framework and a JDBM B+Tree persistence engine.

To integrate with OpenID and implement PKI for authentication, SSL, and tickets, we use the OpenID4Java library and built-in security and cryptography libraries from Sun. We use Java's keytool and OpenSSL for generating, signing, and managing certificates and keys. We use Jetty as a web server and Apache XML-RPC for RPC. Services communicate with each other over HTTP(S) and make requests via RPC or custom protocols for efficiently fetching blobs or streaming query results. The Butler Management Console is written using JSP and Struts.

We have implemented client API libraries for Java, Android, and the iPhone to hide low level details like RPC. As an optimization, the meta-data of resources is cached at the first read attempt and refreshed upon the request of the application, eliminating expensive remote/system calls. Write requests first get committed at the owner Butler before updating the client cache. The client API also supports basic atomic updates and batch operations at a resource level.

The data stewards we have developed include services for generic file systems on a PC, file systems and location data on Android and iPhone, IMAP contacts and attachments, contacts and photos on Facebook, and Google contacts available with the GData interface. Most of these data stewards are only a few hundred lines of code.

Finally, to provide developers a run-time platform for testing their applications without worrying about setting their own PrPl service, we provide a PrPl hosting service. Test developers can register on the web to get an instance of their own PrPl service in minutes.

The PrPl infrastructure is written in Java and has approximately 34,000 source lines of code (SLOC) including 8,800 SLOC for SociaLite. The major infrastructure components include Personal Cloud Butler, Data Steward, Directory, and Client API for building applications.

6.2 Application Experience

At this point, we have created a reasonably rich environment that shows off various strengths of this architecture. Our Butler provides a safe haven for all of our personal data, from the most private such as email, to those we commonly share with friends like photos and music playlists. Not separated in different silos, all our personal data can be used together to enhance our online social activities.

Because of the personal nature of Butlers, users should have no reservations allowing their personal Butler to import all of their contacts in their email accounts. All friends with OpenID-supported email accounts, such as Gmail and Yahoo Mail, can log into a user's Butler and gain access to whichever data the user allows them to see. In addition, we have developed a data-mining system called Dunbar (described elsewhere) that automatically analyzes our email patterns and determines the strength of our ties with individual friends. Most users would not want to show this data to their friends because of its private nature. This further illustrates the usefulness of having a private haven for our data. We found that not only is this information curious on its own, it can be combined with other information to create a more pleasant user experience. For example, when the Butler contacts our friends to run queries, it can contact friends with whom we are closer first.

The PrPl system enables us to make our personal information, such as contacts, GPS locations and photos, available over the web and on our smart phones. There is a unified contact list, which can be used for sharing all kinds of data. This is much more convenient than the current practice of inviting friends for every different website we use to share data. Because of the high-level abstractions the infrastructure provides, we were able to quickly develop a number of applications on various platforms (over the web, and natively on smart phones like Android and iPhone) for accessing our data and that of our friends on different Butlers. These applications are mostly SociaLite queries dressed up in a graphical user interface. The data is protected with authenticated user IDs and Butlers, in addition to allowing users to specify their own access control.

6.2.1 Weighted Social Graphs

By aggregating all of our data in a safe haven, PrPl makes it possible to do *in situ* social networking that takes advantage of information deduced from our behavior and data patterns. The data integration afforded by PrPl lets us automatically estimate the strength of our ties with individuals in our social circle using a metric based on patterns of email communication. In particular, we use the number of sent messages, as a score for the strength of the association between the user and another individual. Though we are looking into other metrics that incorporate factors like recency, longevity and intensity of communication, this remarkably simple metric is effective in identifying our closest ties. Moreover, it can be updated automatically over a sliding time window as relationships change.

A query to the weighted social graph based on the strength of ties can be expressed by the following SociaLite query:

```
CloseFriend(?f, ?t):- (owner, prpl:friend, ?f), (?f, prpl:tie, ?t),
                      $GreaterThan(?t, S).
CloseFOAF(?p, ?t):- CloseFriend(?p, ?t).
CloseFOAF(?p, ?t):- CloseFOAF(?x, ?t1), CloseFriend[?x](?p, ?t2),
                      $Multi(?t, ?t1, ?t2), $Greater(?t, S).
```

This query says that any friend whose strength of tie is greater than a threshold S is a close friend. And the close friend of a close friend may also be considered our close friend if the product of the tie strengths is still greater than S. This reflects real world practice where we would rather ask a friend of a close friend for a favor instead of asking

one of our acquaintances. These weights in a social network may also be used to route SociaLite queries.

6.2.2 Sharing Personal Information

We now describe our experience in building PPLES, an Android interface to the Butler service. Essentially, PPLES enables users to submit SociaLite queries to their Butler service via a graphical user interface. PPLES queries a user's Butler for his list of friends, ranks them by order of tie strength and shows them to the user. The user can select a subset of these friends, and ask to see their shared photos or GPS locations. PPLES sends to the Butler a distributed SociaLite query that uses the According-To operator to retrieve the data of interest. The Butler handles all the communication with the respective Butlers and returns the answers to PPLES. Note that only URIs of the photos are passed around, as the Android application fetches the photos directly from the blob servers. The low-level data retrieval operations are all handled by the PrPl client running on the phone, requiring no effort on the part of the application developer. Note that all queries are subjected to access controls by the respective Butlers according to their owners' specifications.

Our PPLES application presents a user interface organized into UI tabs, each of which represents a different section of the application that is catered to a specific task. The Friends tab displays a user's unified list of social contacts with which to make selections for further shared data queries. The results of the distributed query are displayed as a unified view of photo collections or GPS locations under the Photos and Map tabs respectively. Finally, the Settings tab lets a user gain access to his Butler by specifying his PrPl login credentials, or an existing social networking persona that PrPl supports, such as OpenID or Facebook.

In developing PPLES, most of our focus was on writing application UI code. It took us about 5 days to build a functional version of the application. Significant development time was saved as PrPl and SociaLite dealt with the intricacies of the networking and distributed programming that made distributed queries possible. Out of PPLES's approximately 3028 lines of source code, about 332 lines or 11% of the code dealt with executing SociaLite queries and transforming their results for application usage. Ease of distributed application development is thus another key advantage of our system.

6.2.3 Jinzora: Free Your Media

With the goal of trying to get real users, we have also experimented in creating a mobile social music experience by leveraging a popular open-source music-streaming web application called Jinzora. By integrating into the PrPl infrastructure, users can stream personally hosted music to themselves and their select friends, and to share playlists together. This design gives users the freedom to acquire their music anywhere, while enjoying the accessibility typical of hosted online music services. Not that Jinzora is not designed to be used for mass distribution of music content

We have built a client called Jinzora Mobile (JM) for both the Android and iPhone that connects to a Jinzora music service, lets users browse music by artist, genre, and album, etc., and performs as a standard playback music streamer. A screenshot of the JM client is shown in Figure 2. JM allows a user to switch between Jinzora services, hosted personally by the user or a friend. JM looks up the location of the Jinzora server directly from the PrPl Butler Directory service. Users can login to the service using their OpenID credentials. Friends can share their playlists with each other and discover music together. Users' playlists are saved on their respective Butlers. To access the shared playlists, the JM client needs only to issue a distributed SociaLite query to the user's Butler; it automatically contacts all the friends' Butlers, collates the information and sends it back to JM.

We found that the PrPl platform made it relatively easy to enable service and playlist sharing in JM. The coding effort is reasonable as it involves mainly just creating a GUI over 6 SociaLite queries.

This application is reasonably polished such that a number of the authors use it on a daily basis. We plan to release this software publicly so we can perform large scale experiments in the wild on our infrastructure.

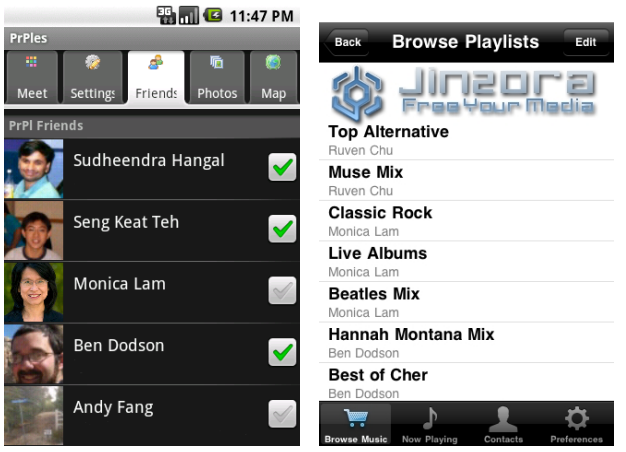

Figure 2. (a) The PPLES application, running on Android, and (b) Jinzora's playlist sharing on the iPhone.

6.3 Performance Measurements

The current PrPl system is designed with the explicit goal of providing a foundation for further research; to this end, we have deliberately chosen general representation schema like RDF and powerful and flexible language support like Datalog. We expect that many of these design aspects will be refined and optimized in the future. We measured our current prototype primarily with the goal to see if the proposed architecture is technically viable. We performed two tests: (1) measuring the performance of a single Butler to evaluate the overhead of SociaLite, and (2) measuring the

performance of SociaLite on a network of Butlers running on Planet Lab.

6.3.1 Performance of a Single Query

We estimate that users will have a collection of a few ten to hundred thousands of music, photos, videos, and documents, each with approximately 5-10 properties. Thus, our first experiment is to evaluate the performance of SociaLite on four PrPl indices, ranging from 50,000 to 500,000 triples. The experiment is run with both the client application and Butler running on an Intel Core 2 Duo 2.4 GHz CPU with 4GB of memory.

We ran two queries and measured its first response time and completion time (Table 1) . The first is a simple triple filter and the second requires joins.

# of Triples	JDBM (MB)	*Filter Query*		*Join Query*	
		1st/Lst (sec)	# of Answr	1st/Lst (sec)	# of Answr
50,000	8	0.9 / 1.2	1,024	0.6 / 0.6	53
100,000	22	0.9 / 2.0	2,095	0.8 / 0.9	91
250,000	118	1.0 / 3.3	5,045	1.0 / 1.3	264
500,000	628	1.4 / 6.9	10,019	1.5 / 2.4	516

Table 1. Performance of SociaLite on 1 Butler

Our SociaLite implementation supports query pipelining because we realize that users often times would rather get incomplete but prompt answers. This means we can start showing the results of the query without having to wait for all the results. With pipelining, this means that the user can start seeing the first result in less than a second. The completion time depends greatly on the size of the indices, from 1 to 7 seconds.

6.3.2 Performance on a Network of Butlers

To simulate a social networking environment with distributed servers, we have deployed Personal Cloud Bulters on over 100 PlanetLab nodes and evaluated three queries:

TAGGED-PHOTOS : Find all "wedding", "party", or "trip" photos among my friends

COMMON-FRIENDS : Find people that you and your friends have in common (contact intersection).

CLOSE-FRIENDS : Get all friends measured to have a social tie greater than a threshold (0.85).

We estimate that about 95 to 100 servers were up at a time on average. We geographically distributed the Butlers and used the 9:1:1 ratio to allocate them in US, Europe, and Asia respectively. The latency varies substantially depending on the location. The round trip times to the same or neighboring states were found to be as small or smaller than 10 ms while to Asia and Europe were as slow as 200-300 ms.

We harvested about 5,000 Facebook user's data and used about 20,000 pictures for this experiment. Each batch of 10 Butlers were given 10 friends, 20 friends, etc, with the last batch connected to 100 friends. Each Butler was given a random number of photos between 50 and 350. We ran the experiment with the query requester client and Butler connected over wifi in our lab for the sake of consistency. We ran five experiments, simulating five users, A, B, C, D, E, each with 10, 20, 30, 50, and 100 friends, respectively. Table 2 presents the characteristics of the queries and the experimental results. The table shows the number of RDF triples that each requester has and the total number of triples owned by the requester's friends. The table shows how long it takes to get the results for each of the queries. Because machines in the Planet Lab have wildly varying performances, depending also on the load, we are reporting the best times measured to indicate how fast a result can be returned. Please note that the mileage will vary in practice with personally hosted Butlers. But as we show below, we can deliver a reasonable user experience by reporting results as soon as they start showing up.

The queries take different amounts of time, with TAGGED-PHOTOS taking the least, and COMMON-FRIENDS taking the longest. The results are mainly a function of the amount of data returned and the number of connections made. For example, CLOSE-FRIENDS makes only 15% of remote queries that COMMON-FRIENDS makes.

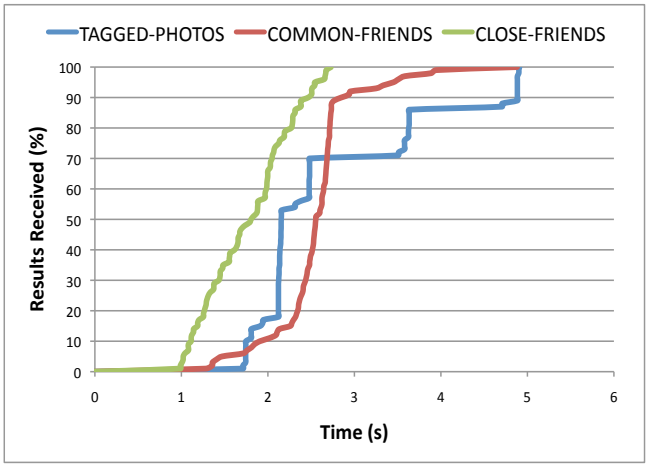

Figure 3. Query Results for User D

There is a large variance on the response times between different Butlers; the first response arrived as quickly as 0.6 ms and most of them arrived within 2 seconds while some queries took as long as 12 seconds. To delve into the detail further, we show in Figure 3 the response time to the three queries in the case the requester has 50 friends. We show that 75% of the responses came in within 3 seconds with outliers taking significantly longer.

In one instance, the CLOSE-FRIENDS query took longer than 9 seconds to complete even though more than 55% of the results came in 4 seconds (User B). We found that one of the close friends who accounted for 60% of the photos was running a Butler on a server that had an extremely high CPU load. Had that tie not been there, the same query would

User	Friends	Own Triples	Friends' Triples	TAGGED-PHOTOS 1st/Med/Last Time	TAGGED-PHOTOS # of Results	COMMON-FRIENDS 1st/Med/Last Time	COMMON-FRIENDS # of Results	CLOSE-FRIENDS 1st/Med/Last Time	CLOSE-FRIENDS # of Results
A	10	4,050	28,130	0.7 / 1.4 / 1.5 s	76	0.6 / 0.7 / 1.6 s	40	0.8 / 1.5 / 2.1 s	302
B	20	2,500	98,910	1.1 / 1.7 / 2.2 s	72	1.2 / 1.6 / 2.6 s	162	0.7 / 3.8 / 9.9 s	514
C	30	2,250	106,810	1.1 / 1.1 / 3.3 s	105	1.2 / 2.1 / 3.2 s	493	1.0 / 1.7 / 2.4 s	482
D	50	4,500	179,551	1.7 / 2.2 / 4.9 s	141	1.3 / 2.6 / 4.9 s	1083	1.0 / 1.8 / 2.7 s	451
E	100	4,085	341400	1.8 / 3.0 / 6.9 s	420	1.9 / 6.6 / 12.3 s	4477	1.6 / 2.8 / 3.5 s	1,130

Table 2. Characteristics and Measurements of Distributed Queries

have taken 1.2 seconds to complete. We observed the same trend for the other cases with more or less friends, while the response variation worsened with an increasing number of friends. Figure 4 shows the response times for all the cases on the same TAGGED-PHOTOS query.

Query pipelining is effective especially in a distributed environment with high variance, but this also shows off the importance of a powerful programming language. We can get to the end results in single SociaLite query rather than having to send multiple rounds of subqueries.

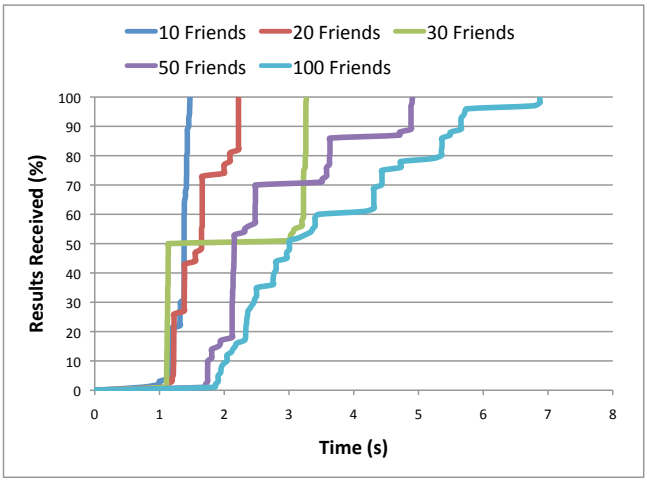

Figure 4. TAGGED-PHOTOS Query Results

7. Related Work

OpenSocial [6] provides API's that makes user information available from a single social network. One can embed widgets on their web page, and access information about a) people and relationships, b) activities feeds and c) simple key-value persistence. OpenSocial's is not an inter social networking API; it does not help users to interact across multiple social networks. Furthermore, access control is weak. In contrast, we allow users to perform deeper integration of their data by running distributed queries in the Socialite language. Users are able to traverse administrative domains while accessing data and services across multiple social networks.

Facebook Connect and the Facebook Platform [7] provide a popular API into a closed social network. It remains the exemplar of a walled garden; inter-operability across administrative domains is not possible. Users are limited to Facebook's changing terms of services and suffer weak access control. By adding an application, users unintentionally share wide-ranging access to their profile information. In contrast, we embrace open platforms/API's such as OpenID, which enable us to extend APIs, perform deeper integration, and most importantly, offer flexible access control.

Homeviews [12] describes a P2P middleware solution that enables users to share personal data based on pre-created views based on a SQL-like query language. Access is managed using capabilities. which are cumbersome for a client to carry, can be accidentally shared and broadcasted, and are harder to revoke. In contrast, PrPl uses a federated identity management system (OpenID) that eases account management overhead, introduces automatic account creation and usage.

Jim et al. explore different evaluation methods for distributed Datalog [14]. Their approach centers on allowing a remote database to respond with a set of rules rather than a table of answers. Loo et al. give an example distributed Datalog system that is used to simulate routing algorithms [15]. They use a pipelined evaluation methodology that is similar to the one implemented in SociaLite. Unlike SociaLite, many domain-specific optimizations and restrictions are incorporated in their language and implementation.

Ensemblue is a distributed file system for consumer devices [16]. While Ensemblue is targeted at consumer appliances and managing media files, it lacks collaboration support, semantic relationships between data items, and a semantic index.

Desktop search applications such as Google desktop [8] create an index over personal data including documents, emails and contacts. While desktop searches allow users to quickly search over several data types, they are limited not only to personal data, but also to single devices mostly.

Mash-ups are web applications that combine data from multiple service providers to produce new data tailored to users' interest. Although mashups can provide unified view of data from multiple data sources, they tend to be shallow compared to our work. First, data sources are limited to service providers: users have to upload their data to each individual service provider. Second, their API's generally create restrictions on usage: PrPl provides a very flexible

API that enables users to implement deep data and service integration, or create deep mash-ups.

Personal information managers: The Haystack project developed a semantically indexed personal information manager [17]. IRIS [10] and Gnowsis are single-user semantic desktops while social semantic desktop [11] and its implementations [19][13] envision connecting semantic desktops for collaboration. PrPl differs from such work by permitting social networking applications involving data from multiple users across different social networking services. We have built a distributed social networking infrastructure that include ordinary users whereas the social semantic desktops only focus on collaboration among knowledge workers.

8. Conclusions

This paper argues that a decentralized, open, trustworthy (DOT) platform is a better alternative to the current centralized, ad-supported online social networking services. We presented the architecture of PrPl, an instance of a DOT social networking platform.

We propose personal-cloud butlers as a safe haven for the index of all personal data, which may be hosted in separate data stores. A federated identity management system, based on OpenID, is used to authenticate users and Butlers. We simplify the development of decentralized social networking application by creating the SociaLite language, a logic programming language for deductive databases, to hide the complexity of distribution from the user. In addition, SociaLite allows users to describe flexible access control policies easily; through rule composition, the access control rules are composed with incoming queries to generate efficient data queries. More importantly, our rewrite system subjects all friends' queries and third-party queries to the access control rules without the cooperation of the query initiator.

We believe that lowering the barrier to entry for distributed social application developers is key to making decentralized social networking a reality. Socialite, with its high-level programming support of distributed applications and automatic enforcement of access control, has the potential to encourage the development of many decentralized social applications, just as Google's map-reduce abstraction has promoted the creation of parallel applications. Furthermore our concepts of access control are applicable to centralized social networking services as well.

References

[1] `http://tinyurl.com/youtube300M2009`.

[2] `http://developers.facebook.com/connect.php`.

[3] `http://yadis.org`.

[4] `http://tinyurl.com/wikiopenidnet`.

[5] `http://www.w3.org/2004/OWL/`.

[6] `http://www.opensocial.org/`.

[7] `http://developers.facebook.com/`.

[8] `http://desktop.google.com`.

[9] A. Chandra and D. Harel. Horn clauses and generalizations. *Journal of Logic Programming*, 2(1):1–15, 1985.

[10] A. Cheyer, J. Park, and R. Giuli. Iris: Integrate. relate. infer. share. In *Proceedings of the Semantic Desktop Workshop at ISWC*, pages 738–753, 2005.

[11] S. Decker and M. Frank. The social semantic desktop. In *DERI Technical Report 2004-05-02*, 2004.

[12] R. Geambasu, M. Balazinska, S. D. Gribble, and H. M. Levy. Homeviews: peer-to-peer middleware for personal data sharing applications. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 235–246, New York, NY, USA, 2007. ACM.

[13] T. Groza et al. The nepomuk project - on the way to the social semantic desktop. In *Proceedings of I-Semantics' 07*, pages 201–211, 2007.

[14] T. Jim and D. Suciu. Dynamically distributed query evaluation. In *Proceedings of the Twentieth ACM Symposium on Principles of Database Systems*, pages 28–39, 2001.

[15] B. T. Loo et al. Declarative networking: language, execution and optimization. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 97–108, 2006.

[16] D. Peek and J. Flinn. Ensemblue: integrating distributed storage and consumer electronics. In *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 16–16, 2006.

[17] D. Quan, D. Huynh, and D. R. Karger. Haystack: a platform for authoring end user semantic web applications. In *Proceedings of the ISWC*, pages 738–753, 2003.

[18] D. Reed, L. Chasen, and W. Tan. Openid identity discovery with xri and xrds. In *IDtrust '08: Proceedings of the 7th Symposium on Identity and Trust on the Internet*, pages 19–25, 2008.

[19] J. Richter, M. Volkel, and H. Haller. Deepamehta - a semantic desktop. In *1st Workshop on The Semantic Desktop*, 2005.

[20] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, Rockville, Md., volume II edition, 1989.