PrPl: A Decentralized Social Networking Infrastructure

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

Computer Science and Electrical Engineering Departments
Stanford University
Stanford, CA 94305

ABSTRACT

This paper presents PrPl, a decentralized infrastructure that lets users participate in online social networking without loss of data ownership. PrPl, short for private-public, has a person-centric architecture–each individual uses a *Personal-Cloud Butler* service that provides a safe haven for one's personal digital assets and supports sharing with fine-grain access control. A user can choose to run the Butler on a home server, or use a paid or ad-supported vendor of his choice. Each Butler provides a federation of data storage; it keeps a semantic index to data that can reside, possibly encrypted, in other storage services. It uses the standard, decentralized OpenID management system, so users can use their established personas in accessing the data.

One pre-requisite to the success of a platform is the availability of applications, which means that ease of application development is essential. We have developed a language called SociaLite, based on Datalog, that allows developers to use a simple declarative database query to access the large collection of private data served up by the Butlers in our social circle running under different administrative domains.

We have developed a prototype of the PrPl infrastructure and implemented a number of simple social applications on the system. We found that the applications can be written in a small number of lines of code using SociaLite. Preliminary experimental results suggest that it is viable to enable sharing of private social data between close friends with a decentralized architecture.

1. INTRODUCTION

1.1 Decentralized, Open, and Trustworthy Social Networking

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. It is in their best interest to encourage users to share all their data publicly, lock this data in to restrict mobility, assume ownership of it, and monetize it by selling such data to marketers. Social networking portals often 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. In addition, a number of factors such as data lock-in and the exorbitant cost of running large-scale centralized services all point to the likely establishment of an oligopoly, or even a monopoly. It is unsettling that we leave the stewardship of all this personal data to a for-profit company. Public outcry would be to no avail were such a company to fail and sell its data assets.

Our goal is to develop the technology that makes possible a decentralized, open and trustworthy social networking infrastructure. This enables people worried about privacy issues to participate in social networking without reservations.

- **Decentralized** across different administrative 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, amongst other metrics of quality. They may choose to store their data in personal servers they own and keep in their homes, or entrust their data to paid or free ad-supported services. Furthermore, they may wish to take advantage of a myriad of storage services available by keeping their data, possibly encrypted, in various locations. 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.

- **Open** API for distributed applications. We aim to create an API that allows a social application to run across different administrative domains. Note that the OpenSocial API [1] is much less ambitious; its goal is to allow the same application to be run on different web sites separately; that is, each application just operates on data wholly owned by a web portal.

- **Trustworthy** interactions with real friends. We wish to create a safe haven for individuals to keep all of their

Research supported in part by the NSF POMI (Programmable Open Mobile Internet) 2020 Expedition Grant 0832820, NSF Grant TRUST CCF-0424422, an Amazon Web Services research grant, and a Samsung scholarship.

ACM Workshop on Mobile Cloud Computing & Services: Social Networks and Beyond (MCS) June 15, 2010, San Francisco, CA, USA
Copyright 2010 ACM 978-1-4503-0155-8/10/06 ...$10.00.

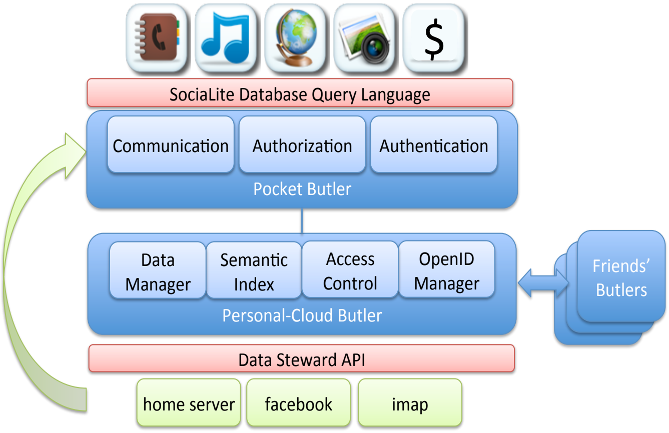


Figure 1: The PrPl data subsystem

data without reservation and to share selected items with different friends. This safe haven will enable new applications since all the personal data are available in one place, and is more convenient for users because they do not have to upload them to different web sites.

1.2 Contributions of this Paper

We have created an architecture called PrPl, short for Private-Public, as a prototype of a decentralized, open and trustworthy social networking system. Having such an option available may also put pressure on ad-supported social networking sites to provide more guarantees on data privacy and ownership.

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 high-level overview of the Butler architecture is shown in Figure 1.

- **Semantic index of personal data**. It provides a unified index of the data to facilitate browsing and searching of all personal information. To support interoperability, we represent our data in a standard format based on RDF and standard ontologies whenever they are available.

- **Federated storage system**. To take advantage of freely available data storage on the web, the Butler lets the user store their data, possibly encrypted, with different storage vendors if they wish. The Butler hands out certificates that enable our devices and devices of friends to retrieve the data directly from storage.

- **Decentralized ID 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.

- **Common client proxy**. To support single sign-on for applications running on the client device, a Pocket Butler handles the underlying authentication and communications with the Personal-Cloud Butlers. It also provides common interfaces to specify authorization and caching so that resources can be shared between multiple applications.

- **Butler services**. Besides providing an open API to the data, described in more detail below, the Butler also provides a web-based service that allows friends to log in with their OpenID to enjoy data and services they are entitled to.

How would Personal-Cloud Butlers be deployed? One possibility is to host such services on set-top boxes and game consoles, which are already in many consumers' homes. These devices already provide users a cost-effective and convenient way to store their large-scale data such as full resolution photos.

SociaLite Language. We have designed and implemented a database query language called SociaLite to allow easy access to the large amount of data stored in a distributed network of Butler services. SociaLite is an extension of Datalog, a declarative deductive database language. Supporting composition and recursion, this language is expressive enough that many social applications can easily be written by adding a GUI to the result of a SociaLite query. This language hides the complexities in distributing a query to the friends' Butlers.

Experimental Results. We have implemented a fully working prototype of the SociaLite language and PrPl infrastructure as proposed in this paper. A rich set of applications was developed alongside the infrastructure so as to drive the design. PrPl applications are relatively easy to develop because they consist mainly of GUI code wrapped around SociaLite queries. Measurements of our prototype running on a testbed consisting of 100 Butlers on Amazon EC2 suggest that this approach is technically viable. Even though our prototype is not optimized, the first query results arrive within a couple of seconds.

2. FEDERATED ID MANAGEMENT

The PrPl system utilizes federated, decentralized identity management that enables secure logins, single sign-on, and communication among applications in an environment where Butlers belong to different administrative domains. We wish 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.

An OpenID login consists of the following steps:

1. Requester enters his OpenID identifier at a Relying Party (RP)'s web page.

2. RP performs the YADIS/XRI discovery protocol [3] on the identifier, fetches an XRDS file [17] that encodes his OpenID Providers (OP), and redirects the user to an acceptable OP.

3. User successfully enters credentials at the OP, which verifies and redirects the user back to the RP along with a signed success message.

4. RP verifies the result with the OP and welcomes the requester.

2.1 The Butler Directory Service

We envision that in the future a Butler service will be associated with each OpenID. The Butler service is to be registered with the OpenID provider, which also serves as its Certificate Authority (CA). The OP provides a digital certificate for the Butler's public key, which can be used as proof that it is the registered Butler for the associated OpenID during inter-butler communications. 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, which 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.

2.2 User Authentication at the Butler

Given an OpenID, anybody can look up the associated Butler service and view the information made available to the public. However, the Butler offers additional services only upon authenticating the guest's credentials. A guest needs to sign on to each Butler service, possibly running in different administrative domains, before using it. We leverage OpenID's single sign-on 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.

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. 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.

2.3 Authentication between Butlers

In the decentralized PrPl architecture, a query presented to a Butler may require the Butler to contact friends' Butlers on behalf of the user. In general, a query may be propagated through a chain of Butlers. To support single sign-on, we use a PrPl Session Ticket which is a tuple <issuer ID, requester ID, session ID, expiration time, issuer's signature>. The issuer is the last Butler issuing the request, whose identification can be verified by its signature. The requester ID specifies the originator of the message and all the intermediate Butlers involved. 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.

3. THE PRPL SEMANTIC INDEX

The Butler keeps an index of personal data and relations which is built with the cooperation of Data Stewards. It enforces access control and presents a programmable interface to applications. It also includes a personal homepage with the personalized services and management console so the user can administer and access his data over the web.

3.1 Semantic Index API

The PrPl semantic index contains all the personal information (e.g. contacts and locations) as well as the 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) [4] 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.

The PrPl semantic index is schema-less and is able to store generic and possibly unknown information types. Whenever possible, we use standard ontologies (for types such as Address, Contact, Calender, and Music). The ontology, types of resources and related properties, are described in OWL, the Web Ontology Language[5]. The common ontology helps evaluate queries over multiple schema-less indices in the network of independently-administered Butlers. The ontology also can be used to enforce restrictions on resource properties (e.g., each Person resource has a single *last name*) and to generate inferred information from given facts.

3.2 Data Stewards on Storage Devices

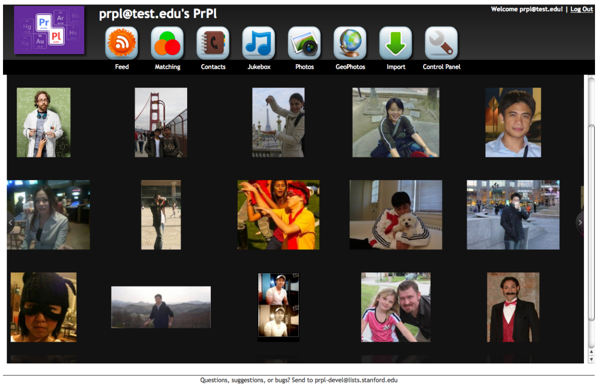
Each federated store that hosts blob data runs a Data Steward, operating on behalf of the user. It provides a ticket-based interface to PrPl applications and hides the specifics about how the blob is actually stored.

At configuration/startup time, the Steward registers itself with the owner's Butler. For existing data resources, the Steward checks for any changes since the last communication with the Butler. It periodically sends heartbeats to the Butler with updated device access information, such as a change in IP address when it is running on a portable device. 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 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. An application can cache the ticket until its expiration time. 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.

3.3 Butler Home and Management Console

A Butler's owner can add services to his Butler, examples of which include active feeds of friends, photo browsing, music streaming service, etc. These services are accessible via the Butler's web-based homepage. This homepage also includes a management console with which the owner can administer and access his personal cloud information. He can manage individual accounts as well as group memberships. He can control read/write access of individual PrPl resources and access to the services provided by the Butler. He can view registered devices and services, view/revoke issued session tickets, and run simple queries directly. It also provides a generic resource browser, where the owner can edit and add meta attributes, download blobs, and specify access control policies.



Questions, suggestions, or bugs? Send to prpl-devel@lists.stanford.edu

Figure 2: The PrPl Butler homepage and photo browser application

4. SOCIALITE: A LANGUAGE FOR A SOCIAL MULTI-DATABASE

SociaLite is an expressive query language for social multi-databases. The abstractions provided by SociaLite hide low-level details of distributed communications, authentication, and access control. It is based on Datalog[9] which 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 negated.

We chose Datalog as the basis of our language for accessing the PrPl social multi-databases for the following reasons. First, Datalog supports composition and recursion, both of which are useful for building up more complex queries. Being a high-level programming language with clean semantics, Datalog programs are easier to write and understand. Also, 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.

4.1 RDF-Based Database

The database in our Butler is an unstructured semantic index, 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 to 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.

4.2 Function Extension

In SociaLite queries, user-defined functions may be used to do additional computation on retrieved data. Two types of functions are supported - *tuple-wise functions* and *relation-wise functions*.

Tuple-wise functions can be applied to each tuple of a relation, one at a time. Formally, we say that a tuple-wise function $\$F$, which takes n inputs and returns m outputs, $\$F(o_1, \ldots, o_m, i_1, \ldots, i_n)$ is true if and only if $(o_1, \ldots o_m) = F(i_1, \ldots, i_n)$.

Relation-wise functions operate on an entire set of tuples. A relation-wise function is syntactically located on the left-hand side of a query statement. Its arguments must include another predicate, but may also include other variables such as constants. The function may return either a set of tuples or a single scalar value. For instance to count the number of friends, the function Count can be used as following:

FriendsCount(Count(f)) :- Friend(f).

4.3 Remote Queries

SociaLite has an ACCORDING-TO operator "[ ]" to allow succinct expression of remote queries. The predicate "$P[x](y)$", read $P(y)$ according to x, evaluates predicate $P(y)$ on x's database. When used together with recursion, the operator allows one to traverse the distributed directed graph embedded in the social database.

With SociaLite, we can write a multi-domain distributed query in a few lines. Suppose we are interested in collecting all the pictures taken at a Halloween party among our friends. The SociaLite query may look like:

Friends-Halloween(p,f) :- Friend(p), Halloween[p](f).
Halloween(f) :- (f, a, prpl:Photo), (f, prpl:tag, "Halloween").

This query gathers together pictures with the same "Halloween" tag that are in our friends' respective Butlers by sending a sub-query to the Butlers. When a (sub-)query needs to be evaluated in a remote Butler, the SociaLite engine takes care of authentication and proper access control, freeing the programmers from low-level burdens.

5. PROTOTYPE IMPLEMENTATION

We have developed a fully functional prototype of the PrPl architecture as described in this paper. The prototype, written in Java, 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, and Pocket Butler with Client API for building applications. Because we cannot change the OpenID Providers to provide the Butler directory service, we have created a Global Butler Directory service in the meantime to support experimentation.

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 HTTPS and make requests via RPC or REST APIs. Custom protocols are used for efficiently fetching blobs or streaming query results. The Butler Management Console is written using JSP and Struts. The PrPl index is implemented using HP's Jena semantic web framework and a JDBM B+Tree persistence engine.

The SociaLite implementation also makes use of the Jena/ARQ libraries for iterating over triples and XStream is used for serializing and streaming query results. Our implementation of the SociaLite language includes optimizations such as semi-naive evaluation, caching, and query pipelining so that partial results can be returned to minimize the query response time. Details of these optimizations are beyond the scope of this paper.

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. Currently, Pocket Butler is implemented for Android using the Intent framework, supporting single sign-on.

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 up their own PrPl Butlers, we provide a PrPl hosting service. Test developers can register on the web to get an instance of their own PrPl service in minutes.

6. PRELIMINARY EVALUATION

The current PrPl prototype is designed as an experimental vehicle and has not been optimized for speed. Nonetheless, it is useful to measure the performance of queries running on this prototype to ensure that the decentralized approach is viable.

6.1 Queries to one Butler

Our first experiment is to measure the performance on a single Butler to establish the baseline of the system. 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 4 GB of memory.

The queries for this experiment return the URI of the photos that match one and two given tags, respectively. Because users like to see partial results as soon as possible, the Butler pipelines the query execution and starts returning initial results without waiting for all results to be computed. Figure 3 shows the number of results returned for each of the queries, and the time the Butler takes to report the first and last result to the client application.

		One Tag		*Two Tags*	
# of Triples	**DB (MB)**	**1st/Lst (sec)**	**# of Ans**	**1st/Lst (sec)**	**# of Ans**
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

Figure 3: Photos matching one and two tags on a single Butler

While the first query is simpler, it returns many more results. Thus, even though the second query requires two lookups and one join operation, it is uniformly faster than the first. The overhead in communicating the results to the client dominates execution time. By pipelining the query, the Butler returns the first results within 1.5 seconds regardless of the size of the database and the number of results of the query. This suggests pipelining is very important, especially since the end user cannot view the results all at once anyway. Note that the performance of the system is commensurate with the results reported for the underlying Jena library used in the PrPl prototype.

6.2 Performance on a Network of Butlers

To simulate a social network, we deployed 100 Personal-Cloud Butlers, each of which simulates a user with his or her own database, on the Amazon EC2 platform. Each Butler was randomly given 10 to 100 friends, 50 to 350 photos, up to 1,500 songs, and 1,000 location data points. In assigning music, we used a Gaussian distribution over the synthetically generated library of 100,000 songs to simulate real-world song popularity. We associate a tie strength [11] with each friend, which may be estimated for example by the number of email messages sent to the friend. For this experiment, we simply use a weight randomly generated between 0 and 1; friends with weights greater than 0.7 are considered *close* friends. We refer to the Butler initiating the query as the *issuer* and the Butler of a friend as simply a "friend". The issuer in this experiment is set up to have 30 friends. We tested the system with a set of queries typical of social applications. Characteristics of the queries are summarized in Figure 4, and their performance is shown in Figure 5.

- **Common friends**. This query finds common friends an issuer has with *each* of his friends. In this case, the query contacts 30 friends and the sum of all the friends' list returned is 1,570. A join is performed for each list with the list of friends the issuer has, resulting in 494 tuples in all.

- **Friends star**. This query computes the closure on

Query	Description	# Rules in Query	# Butlers Reached	Answers	1st/Last (sec)
Common friends	Common friends between me and my friends	2	30	494	1.0/1.9
Friends star	All the people connected through my friends	3	100	100	0.7/5.2
Close friends' pictures	URIs of photos from my close friend	3	10	1,642	1.3/3.6
Top 10 songs	Top 10 popular songs among my friends	6	30	10	4.2/4.2

Figure 4: 4 Distributed SociaLite queries

the people that one can reach by following friendship connections in the network. The issuer contacts his immediate friends, each of whom recursively computes it's own friends network. The issuer in this case is connected to all the 100 people in the simulation; all the friends can be reached within two hops, and a third hop is required to detect convergence.

- **Close friends' pictures.** This query returns the URIs of the photos of our close friends. The filter by tie strength in this query illustrates the use of user-defined functions in SociaLite. This query returns approximately 1,642 results.

- **Top 10 songs**. In the `Top10Songs` query, the issuer contacts his friends to return their most popular 50 songs and the respective play counts. With 30 friends, the issuer receives 1,500 song counts from his friends. It combines the friends' counts with his own counts and reports the 10 most popular songs to the user. This query also shows off the relation-wise function extension in SociaLite.

We first observe from Figure 4 that these queries can be expressed succinctly in 2 to 6 SociaLite rules. For all these queries, the client application contacts the issuer's Butler, which is responsible for contacting all the other Butlers. Note that the performance reported here includes the overhead of contacting the various butlers on the EC2 platform as well as authentication of the Butlers against the Butler Directory service. The queries all complete within 5 seconds. The performance of the query depends on the number of butlers contacted and the number of intermediate and final results computed. "Friends star" takes the longest because it has to reach convergence across 100 butlers. With the exception of "top 10 songs", all the queries benefit from query pipelining, with the first results returning within 1.5 seconds. Query pipelining is especially important in a distributed environment where the response time of different Butlers can vary greatly. These results suggest that the decentralized approach is viable.

7. APPLICATION EXPERIENCE

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. 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.

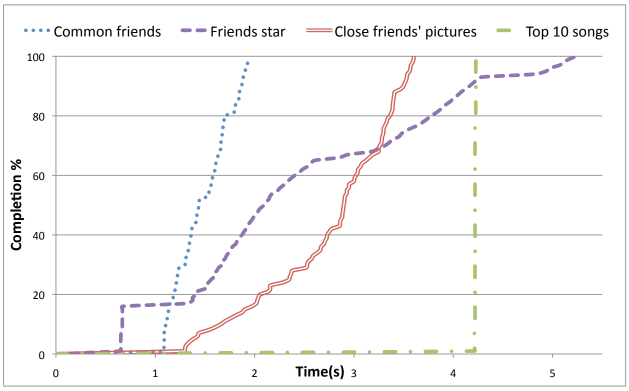


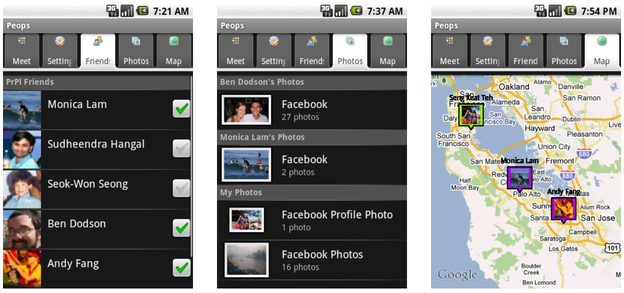
Figure 5: Rate of results returned for queries in Figure 4

7.1 Sharing Personal Information

We now describe our experience in building Peops, an Android interface to the Butler service. Essentially, Peops enables users to submit SociaLite queries to their Butler service via a graphical user interface. Peops 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. Peops 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 Peops. 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 Peops 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 Peops, 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 Peops's approximately 3,028 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.



PrPl Friends

Monica Lam ☑

Sudheendra Hangal ☐

Seok-Won Seong ☐

Ben Dodson ☑

Andy Fang ☑

Ben Dodson's Photos

Facebook
27 photos

Monica Lam's Photos

Facebook
2 photos

My Photos

Facebook Profile Photo
1 photo

Facebook Photos
16 photos

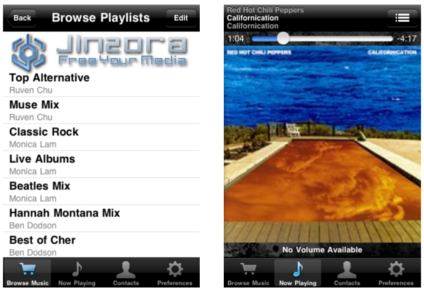
Figure 6: The Peops application, running on the Android.

7.2 Jinzora: Free Your Media

With the goal of trying to attract 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 Jinzora with the PrPl infrastructure, users can stream music to themselves and their select friends, and also share playlists. This design gives users the freedom to host their purchased music anywhere while enjoying the accessibility typical of hosted online music services. Note that Jinzora is not designed to be used for mass distribution of music content, but rather personal streaming and streaming to a select group of friends.

We have built a client called Jinzora Mobile (JM) for both the Android and iPhone platforms that connects to a Jinzora music service. Users can browse and search their music and stream the content directly to their mobile phone. JM allows a user to switch between Jinzora servers, hosted personally by the user or by 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.

8. RELATED WORK



Top Alternative
Ruven Chu

Muse Mix
Ruven Chu

Classic Rock
Monica Lam

Live Albums
Monica Lam

Beatles Mix
Monica Lam

Hannah Montana Mix
Ben Dodson

Best of Cher
Ben Dodson

Red Hot Chili Peppers
Californication
Californication

No Volume Available

Figure 7: Sharing playlists in Jinzora on the iPhone.

OpenSocial [1] provides API's that makes user information available from a single social network. One can embed widgets on their web page, and access people and activity information. OpenSocial's is not an inter social networking API; it does not help users to interact across multiple social networks. 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 [2] 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 [10] 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 the federated OpenID management system that eases account management overhead and supports automatic account creation and usage.

Lockr [19] is an access control system for online content sharing services. It decouples data management from sharing system by letting users exchange attestations which certify a social relationship between the issuer and a recipient. However, real-life social relationships are often not equally bi-directional. It also relies on the issuer and recipient to mutually protect the attestation. However, the recipient could collude with malicious attackers or share the attestation for monetary reasons.

Persona [6] is a distributed OSN (Online Social Network) that shares the goal of promoting user privacy. It uses a cryptographic mechanism to share personal data. However, Persona requires its users to generate a key-pair and distribute the public key out of band to other users. In addition, it does not provide a mechanism for handling expiring relationship keys. NOYB [13] instead obfuscates sensitive information by randomly permuting the data. Again, the user must share the secrets with friends so that friends can view the user's private information. The PrPl system takes advantage of well-established identities such as email addresses, phone numbers, and OpenIDs, and it does not require any out-of-band secret exchange.

Looet al. give an example distributed Datalog system that is used to simulate routing algorithms [14]. 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 [15]. While Ensemblue is targeted at consumer appliances and managing media files, it lacks collaboration support, semantic relationships between data items, and a semantic index.

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.

The Haystack project developed a semantically indexed personal information manager [16]. IRIS [7] and Gnowsis are single-user semantic desktops while social semantic desktop [8] and its implementations [12, 18] 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.

9. CONCLUSIONS

This paper presents the architecture of PrPl, a decentralized, open, and trustworthy 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 hide the complexity of decentralization from the application writers by creating the SociaLite language.

We have implemented a fully working prototype of the PrPl system and have developed a couple of mobile social applications on it for the Android and iPhone. We found that the applications are easier to write. Preliminary performance measurements of representative queries on a simulated network of 100 Butlers in the EC2 system suggest that it is viable to use a decentralized architecture to support sharing of private social data between one's close friends.

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, 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.

10. REFERENCES

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

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

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

[4] `http://www.w3.org/RDF/`.

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

[6] R. Baden, A. Bender, N. Spring, B. Bhattacharjee, and D. Starin. Persona: An online social network with user-defined privacy. *SIGCOMM Comput. Commun. Rev.*, 39(4):135–146, 2009.

[7] 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.

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

[9] H. Gallaire and J. Minker, editors. *Logic and Data Bases.* 1978.

[10] 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, 2007.

[11] E. Gilbert and K. Karahalios. Predicting tie strength with social media. In *CHI '09: Proceedings of the 27th International Conference on Human factors in Computing Systems*, pages 211–220, 2009.

[12] 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.

[13] S. Guha, K. Tang, and P. Francis. Noyb: Privacy in online social networks. In *WOSP '08: Proceedings of the First Workshop on Online Social Networks*, pages 49–54, 2008.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] A. Tootoonchian, S. Saroiu, Y. Ganjali, and A. Wolman. Lockr: Better privacy for social networks. In *CoNEXT '09: Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*, pages 169–180, 2009.

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