PRPL: A DECENTRALIZED SOCIAL NETWORKING INFRASTRUCTURE

A THESIS
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
ENGINEER

Seok-Won Seong
May 2010

This dissertation is online at: http://purl.stanford.edu/bv954hs0202

Approved for the department.

Monica Lam, Adviser

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost Graduate Education

This signature page was generated electronically upon submission of this thesis in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Acknowledgements

First and foremost, I would like to thank my adviser, Monica Lam. I am deeply grateful for consistent support and guidance that you provided me throughout the time. My family, your presence is the biggest motivator of my life and I can not thank you enough. And last but not least, I would like to thank the Stanford colleagues, summer students, and collaborators. This was a group effort and I was fortunate to work with all of you.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Loss of Control, Lack of Privacy

Online social networking applications have greatly improved our way of interacting with friends in the digital life. They help us to stay in touch with existing friends or to make new ones. Today, consumers have a large number of free social networking services and data storages to choose from. However, a free 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 claim full ownership of all user data. 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.

While plenty of companies are vying to store our data, majority of personal data are stored on the home computers. The uplink bandwidth has not improved to match our ability to capture photos and videos at higher and higher resolutions. Moreover, online social networks require us to explicitly upload data to their own storage to use their services. It is difficult to keep track of all data now that it is scattered across

multiple devices and websites. Moreover, much of our data are shared with only a few friends and family at a time, while the social networking companies are spending lots of resources to scale.

1.2 Decentalized, Open, and Trustworthy Platform

Instead of a centralized application-based approach to social networking, we propose a very different model. We presented PrPl, short for Private-Public, as a prototype of a decentralized, open and trustworthy social networking architecture.

PrPl allows users to keep their data in different administrative domains that they prefer but makes it possible to interact with each other. Users have a choice in services that may offer different levels of performance and privacy. For example, they could store data in personal servers at home, pay to keep it in the cloud, or host in a free ad-supported portals.

PrPl provides open APIs for building distributed applications. Applications can cross multiple administrative domains to perform queries and access shared data while the platform hides away complexities underneath.

In PrPl, users can interact with real friends unlike superficial relationships we build in the current social networks. We wish to create a safe haven for individuals to keep all of their 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.

1.3 Contributions of Thesis

We propose a design of a new decentralized social networking platform and the notion of Personal Cloud Butler. With the Personal Cloud Butler, users can safely store all of personal data yet easily share with friends.

Second, we present open application framework and APIs. Openness lowers the

bar for innovation and provides more choices to consumers. With our APIs, application developers can focus on the business logic rather than struggling with the complexities of building distributed applications.

Third, we implemented a prototype of the proposed system and demonstrate that it is a viable one. To support our claim, we have built a number of social applications on the top of the infrastructure and conducted a large scale deployment study. We report the measurements from the study and experience with the applications.

The remaining of the thesis is organized as follows. Chapter 2 presents the PrPl architecture and its key features. Chapter 3 describes the detailed implementations (of the infrastructure and applications) and programming APIs. Chapter 4 presents the evaluation of the system in a large-scale study and experience from building the applications, Chapter 5 discusses related work, and Chapter 6 concludes the thesis.

Chapter 2

PrPl Infrastructure

2.1 Architectural Components

We propose the notion of 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. Larger data such as video, photo, music are stored in federated storages called Data Stewards. Data can be stored encrypted. To be able to discover existing and new friends, Directory provides a lookup service. Pocket Butler sits in the client device, providing single sign-on and helping with communication and authentication details. A high-level overview of the PrPl Butler architecture is shown in Figure 2.1.

2.1.1 Personal Cloud Butler

In our architecture, every user runs a person-centric service called Personal Cloud Butler. It provides a unified view of user's private and shared data but does not store actual copies of all the data. Instead, it retains only the small descriptions (metadata) and builds a semantic index on which queries can be performed. Larger blob objects are stored in federated storage services called Data Stewards and a Butler maintains one or more references to them. Blobs may be replicated amongst multiple Data Stewards to improve reliability and locality.

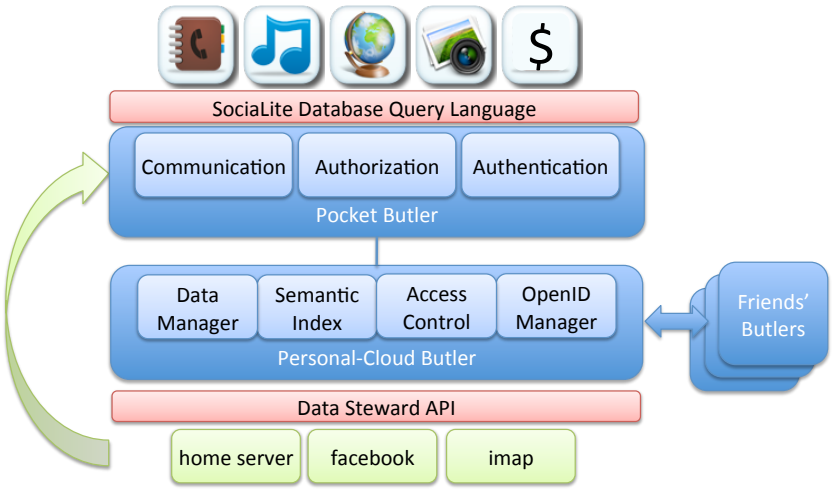

Figure 2.1: PrPl Data Subsystem

Each Butler runs in its own administrative domain and locally enforces access control. However, Butlers connect and communicate with other Butlers cooperatively to create an overlay of distributed social databases. To answer a query, a Butler can forward an inquiry to the Butlers of one's friends to search shared information. Once the user is logged in, the user session is valid with multiple Butlers as long as the user and the Butler who vouched for the user are both trusted.

Note that the Butler is not just making the data available, but also providing cloud services. There are a number of available services that runs on the Butler. We imagine that there will be an appstore that we can go to buy services we wish to host, just like there is an appstore for mobile devices today. A Butler also allows friends to access user's shared data even if these friends do not own a Butler. Each Butler has a homepage where friends can log in with their existing ID to enjoy services and data that a user shares with them.

2.1.2 Data Steward

Every user's device in his federated storage runs a Data Steward operating on user's behalf. It provides a uniform interface to the Butler and PrPl applications, hiding the specifics about how information is actually stored on the devices.

At configuration time, the Steward registers itself with the owner's Butler. It

generates the summary and updates of the blobs in the storage and pushes them into the Butler. If the blob changed since the last communication, a Butler can update its index and compute the number of replicas for a given blob. Data Steward periodically sends heartbeats to the Butler with updated access information and blobs for data sources like file systems that may be updated externally. It tracks where the blobs are located. Specifically, it maps a PrPl URI to a source URI, such as one beginning with `file://` or `http://`.

The Steward services blob access requests from applications directly to avoid making the Butler a bottleneck. An application is required to obtain a ticket from the Butler owning the resource. The ticket certifies that the application was granted the right 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.

If the number of replicas for a blob becomes lower than the threshold, the replication process takes place, initiated by the Butler. There is a primary Steward assigned for each blob resource and it is responsible for propagating the changes.

We envision in the future that there will be storage services that implement the PrPl protocol natively. To enable experimentation, however, it is useful to create interfaces for existing data sources, so they can be made part of our federated storage system. Some examples of important data sources we have incorporated are,

Personal information management tools : Calendar and contact information are particularly relevant in social networking applications.

Files in personal computers : A Butler should capture all the metadata associated with files on one's personal computers for existing photos, videos, and documents. Furthermore, application specific database files (e.g. browser and music player history) contain very rich personal meta information.

Email messages and attachments : Many people resort to mailing data to themselves so the data can be accessed from anywhere. However, it is hard to figure out where to find the correct version of a file. Meta information on email messages, contacts, and attachments, is an important data source to incorporate. Email messages from online stores such as Netflix and Amazon contain a user's movie rental history and purchase history over the years. These meta information is very relevant for PrPl.

Online social networking sites : Many people have uploaded a lot of interesting information to various networking web sites. Leveraging the APIs that some of these sites provide, it is easy to build a Data Steward for them. For example, with the Facebook API, we can not only extract user data in Facebook, we can push information from PrPl to Facebook.

2.1.3 Directory

We envision in the future that a Butler service is associated with one of more OpenIDs. Each OpenID represents different persona that user possesses (e.g. employee, students, son, father, and etc). For friends to find the user's Butler, the Butler registers itself with the Directory Service, which also is a OpenID provider. A user can request the Directory for information of a friend, given the friend's OpenID.

Directory also serves as a Certificate Authority (CA). At Butler registration, the Directory 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.

To register a Butler with its Directory Service, 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, the Butler's URL. The owner then authenticates himself to the Directory Service to prove that user controls the ID. Post authentication, Directory certifies the public key and creates mappings for lookup.

2.1.4 Pocket Butler and Applications

Pocket Butler, a proxy service running on the client device, provides single sign-on for all the applications running on the same device and handles the underlying authentication and communications with the user's Personal-Cloud Butler. In addition, a Pocket Butler provides the common authorization interfaces for selecting and sharing with personalized groups of a user.

A program that interacts with the PrPl Butler, regardless if it is user-facing or background-running, is consider a PrPl application. Based on its interactivity with the Butler, a PrPl application can be categorized into one of the three groups: read-only, write-only, or both read-&-write. Examples of read-only applications are RSS feed and photo browsers; they issue a query to the Butler, then transform the results and present back to a user. Location tracker application is an example of write-only application.

Majority of applications that we have built were either read-only or read-intensive. Pocket Butler leverages this by remembering recently read metadata and blobs. It also caches the results of the queries and whenever possible, it serves the requests locally for efficiency. Since many applications repeat same or similar requests, they benefit from having a Pocket Bulter cache.

Not only it serves as a local cache but also it provides a limited offline operability. Not every application requires an up-to-date or real-time information, and at times incomplete answers are better than no answer at all. Phonebook application, for example, can sufficiently operate with only the cached data. Additionally, a Pocket Butler supports new resource creations and modifications during the offline mode. Once the device is reconnected to the network, it pushes the local state changes to the owner Butler. Since it does not permit modification of the existing resources, there is no inconsistency issue to resolve at this time.

2.2 Key Features

2.2.1 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 administration 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 [1]), extensive library support, availability of accounts, and the ability to extend the protocol easily for PrPl's needs. An OpenID handshake or 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 [2] on the identifier, fetches an XRDS file [22] 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.

User Authentication at the Butler In the PrPl system, a user may have multiple personas, each of which may be given a different external name, similar to the collection of email addresses that we own today. Multiple of these personas are likely to map to the same Butler. However, if desired, a user may have different Butlers for different personas.

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. 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. 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.

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 create a new ticket with the same session ID by updating the expiration time and signature.

2.2.2 PrPl Semantic Index

Each Butler maintains a PrPl semantic index to manage an individual's personal cloud. The PrPl semantic index is a cross between a semantic database and a semantic file system. It contains all the smaller data objects like personal information and the meta information of the larger objects. For example, it knows when a photo is taken,

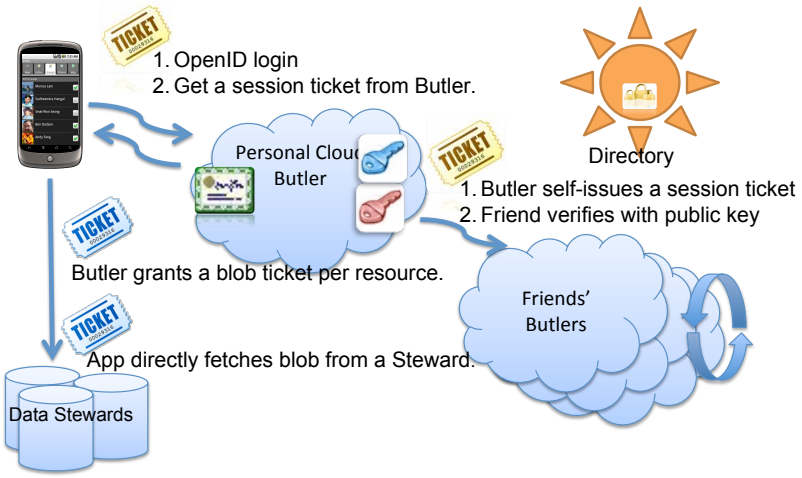

Figure 2.2: Federated ID Management and Authentication

where it is stored, and tags associated with the photo.

PrPl Resources All data objects in the PrPl semantic index are known as resources. There are two kinds of resources: large data objects where the body of the objects, known as *blobs* or binary large objects, are stored externally, and smaller data objects that are kept locally in their entirety in the PrPl semantic index. We refer to the former kind of object as a *blob resource*, and the latter as a *blobless resource*. All resources in the system have a unique PrPl URI (Universal Resource Identifier). There are special blobless resources in the semantic index:

Identity: An *identity* represents a PrPl user. Its URI has the form, `prpl://identity/#`*uname*, where *uname* is a unique name. Encoded in the unique name is the information on how to locate the user's Butler. One possible example for naming identities is OpenID [21]. Directory service maps the user ID to his Butler.

Group: A *group* is a set of identities and it is used to facilitate the specification of access control. The URI of a group resource has the form, `prpl://group/`*uname*`#`*gname*, where *gname* is the name of a group given by user *uname*.

Device: A *device* holds the user's blobs. The URI of a device resource has the form, `prpl://device/`*uname*`#`*dname*, where *dname* is the name of a device given by

user *uname*.

Service: A *service* represents a PrPl application. The URI of a service resource has the form, `prpl://service/#`*sname*, where *sname* is the name of the service.

All other resources have a generic PrPl URI of the form `prpl://resource/#` followed by a unique identifier [3]. Resources have the following metadata properties:

Name: A human-readable and/or user-given name of a resource.

Type: The name of the type of the resource. Each type may have type-specific metadata. For example, a Device resource has a *Location* property, which specifies how the device can be contacted. A Photo resource may have a resolution of the picture, camera model, geographic location where the photo was taken, etc.

Owner: The identity owning the resource.

Last modification time: The time when the information in the resource was last changed.

For blob resources, additional metadata include:

Size of the blob: Its length in bytes

Content hash: The message digest of blob content computed with the SHA-256 hash function

MIME type: The associated MIME content-type (e.g. image/jpeg)

Source: It points to the Device resource holding the blob.

Last modification time of the blob: The time when the blob was last updated.

Representing Metadata in RDF The PrPl system aims to support a wide range of users' existing data as well as new data types in the future without requiring any change to the underlying system. It is difficult, if not impossible, to pre-define what these data types might be. The specification of existing data types frequently changes.

We describe resources and metadata in a general format called RDF (Resource Description Framework) [4] as a series of subject-predicate-object triples. For instance, the fact that Alice has a friend `Bob` with a phone number `123-456-7890` and an email address `bob@bob.net` is described with the following triple:

`prpl://identity/#alice`	*foaf:knows*	`prpl://identity/#bob`
`prpl://identity/#bob`	*rdf:type*	`foaf:Person`
`prpl://identity/#bob`	*foaf:name*	`‘‘Bob’’`
`prpl://identity/#bob`	*foaf:mbox*	`‘‘bob@bob.net’’`
`prpl://identity/#bob`	*foaf:phone*	`‘‘123-456-7890’’`

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 evaluating 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. single *last name* property for a Person resource). By incorporating more domain specific and relational associations between the objects, the Butler can make use of semantic inferences.

2.2.3 SociaLite Query Language

SociaLite is an expressive query language for social multi-databases based on Datalog. The abstraction of SociaLite hides low-level details of distributed communications, authentication, and access control. In other words, SociaLite addresses issues associated with having multiple administrative domains.

Datalog[12] is a query and rule language for deductive databases that syntactically is a subset of Prolog[25]. 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. 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.

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.

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 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. When data is retrieved from RDF database, SociaLite enforces access control checking for the data. Hence, programmers need not check access control outside of SociaLite query.

Functions and Remote Queries 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 while relation-wise functions operate on entire sets of tuples.

SociaLite introduces the ACCORDING-TO operator to grant a query access to the entire social multi-database as a whole. When used together with recursion, the operator allows 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:

Friends-Halloween(f,p) :- Friends(f), Halloween[f](p).
Halloween(p) :- (p, a, prpl:Photo), (p, prpl:tag, "Halloween09").

This query gathers together pictures with the same "Halloween09" tag that are in our friends' respective Butlers by sending a sub-query to the Butlers. By simply binding the variable in the the ACCORDING-TO operator, we can write a multi-domain distributed application in a few lines. In the above query the photos may reside in the query-issuer's own Butler or they may reside in friends Butlers.

2.2.4 PrPl Filesystem Interface

While SociaLite provides an easy programming interface for a new application to be built upon, it is integral for us to leverage a large collection of application-base that currently exists and users are familiar with. To this end, the system provides a traditional filesystem interface to the PrPl semantic index via FUSE (File system in userspace) [6]. Without needing to rewrite the code, existing applications can find this familiar interface to interact with the Butler.

The PrPl semantic filesystem populates the specified mount point with all accessible PrPl resources. A user can then navigate into subfolders, as if he is to write a query to filter resources by different attributes. In a resource folder, metadata are represented in text files. If a resource has a blob, a file representing the blob shows in the folder in their expected form (*jpeg* for photo, *mp3* for music, etc). If a resource has meta relationship with other resources, a user can navigate into a related resource by simply changing the current folder. It provides a semantic filesystem view of not only a user's personal data but also friends' data that are shared with the user.

2.2.5 Butler Homepage

Each Butler provides an interactive web-based homepage where a user can add and share services on his Butler, including photo wall, feeds of friends, contact list, and jukebox. Friends of the user can login and enjoy shared data and services without having to own a Butler. The homepage also includes a management console. The owner of the Butler can administer and access his personal cloud information. On

the control panel, the user can manage user accounts and group memberships. He can control read/write access of individual PrPl resources and services provided by the Butler. He can view registered devices and services, view issued tickets, and run simple queries directly. It provides a generic resource browser where a user can edit meta attributes, download blobs, and specify access control policies.

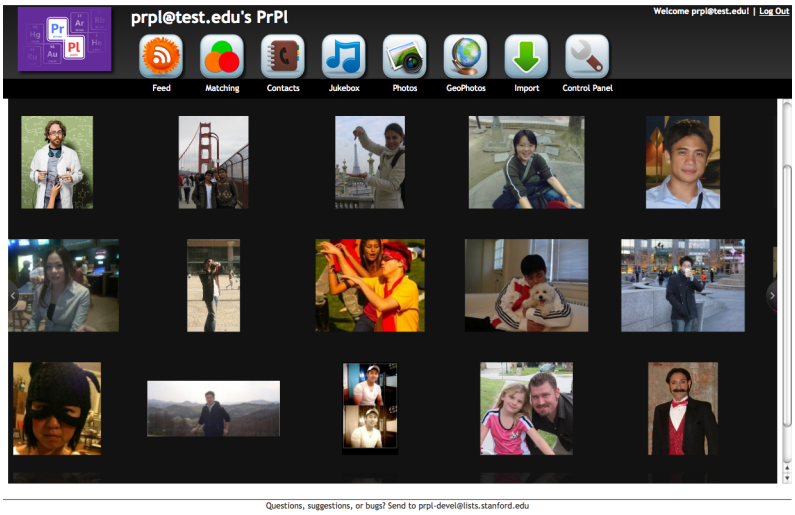

Figure 2.3: PrPl Butler Homepage with Photo Wall Application

Chapter 3

Implementation

3.1 Applications

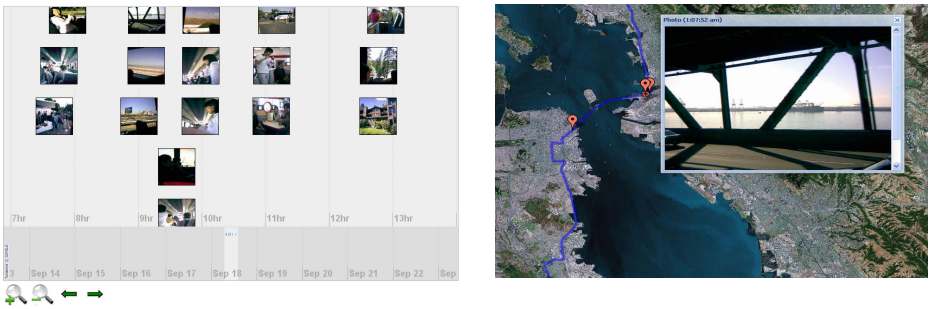

Figure 3.1: Collective Memory Application: Timeline and Map Views

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.

In the early stage of the project, our applications were built using SPARQL [7], which is a standard query language for RDF. The experience helped us to understand the limitation and motivated us to create a new query language for social multi-databases that can scale. Today, all of new PrPl applications are built using SociaLite and many of early applications are being rewritten.

The PrPl infrastructure with SociaLite facilitates creation of powerful applications. The total lines of code for the applications we have built vary from 100s to a few 1000s. In many cases, it is a few lines of SociaLite queries with the GUI code. We have built applications on a number of different platforms including the Android, iPhone, web and PCs. We describe the applications here and in Chapter 4, we report our development experience more in detail.

Smart Phonebook: It is a cumbersome experience to build a phonebook applications, as it involves importing existing contact lists from multiple sources (websites, older phones, and email accounts). Given that PrPl already unifies the information about individuals, the Smart Phonebook is a very simple application that shows the unified view of people you communicate with. It also integrates other data such as Facebook status messages and profile photos.

Collective Memory: The Collective Memory application constructs a complete picture of an individual or a group's day, week, or month, using data from many different sources. Aggregating the friends and own pictures from the same trip and visualizing this data in temporal and spatial graphical interfaces, users can truly reconstruct the experience that they shared with friends.

Social Reviews: The Social Reviews lets a user ask "what are interesting places around me that my friends recommend?". The application helps him identify what is popular among his circle of friends near his current location. This yields results that are of higher relevance due to shared, social context rather than querying generic public services such as Yelp.

Personal Feeds: Personal Feeds enable a user to subscribe to his friends activities across the web using a single feed URL. The feed reader synchronizes continuously with the user's Butler and fetches updates when a new resource is discovered or existing one is modified.

Movie Card: The Movie Card leverages a user and friends' movie preferences, location, and calendar data, along with the publicly available movie schedules and

ratings to assist tedious tasks of deciding and scheduling a movie night with friends.

PrPl Chat: We have built a decentralized chat application on PrPl. A user can host an entire chat session or each participant can contribute to the session but keeps his messages locally. The first version was written in SPARQL but later was rewritten in SociaLite with just 4 lines of a query plus the GUI interface.

Synchronous Browsing: One of the big annoyances of having multiple devices is that a user must perform the same tasks multiple times on each machine because application state is not being shared across. Especially, it is the case for browsing. A user finds a restaurant and directions on a PC, then later on his iPhone he must search for the same restaurant. With PrPl, we can easily synchronize the browser states in multiple devices.

Peops: Peops is an Android interface to the Butler service. It 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.

Jinzora Mobile: Jinzora Mobile (JM), built for both the Android and iPhone platforms, connects a user to a Jinzora music service. Users can browse and search their music and stream the content directly to their mobile phone.

Email Publisher: Email client is a de facto standard application on every PC and mobile platform. We leverage its ubiquity to make it easy for a user to post data to the PrPl Butler. For quick status updates or uploading a new picture on the road, a user can send an email to the designated email address. The Email Publisher service automatically detects a new message and updates the PrPl index, making real-time updates and sharing simple.

The Bulter homepage also hosts a number of social applications. Friends of a user, without having to own a Butler, can log in and browse shared photos (the Photo Wall application) and listen to music (the Jukebox application). With the

Figure 3.2: RSS Feed and Smart Phonebook Applications on iPhone.

PrPl Twitter application, a user can post a quick status update or record his daily activities (e.g. weight, height, daily spending, etc). The PrPl Plot application helps a user to visualize these activities by plotting them on a graphical chart and compare data with the friends'. PrPl Match is an application that finds common interests between a user and his friends.

Each application above demonstrates one or more features of PrPl as described in Chapter 2 (unification of data from disparate and federated sources, collaborative viewing/visualization, distributed query over multiple databases, and continuous updates from multiple input sources). It is important to note that the applications are mere examples and can be easily generalized.

In order to assist new PrPl application developers, we provide tutorials and sample "Hello PrPl" applications, built for Java and Android. The source code guides through a number of key PrPl API calls and a simple SociaLite query.

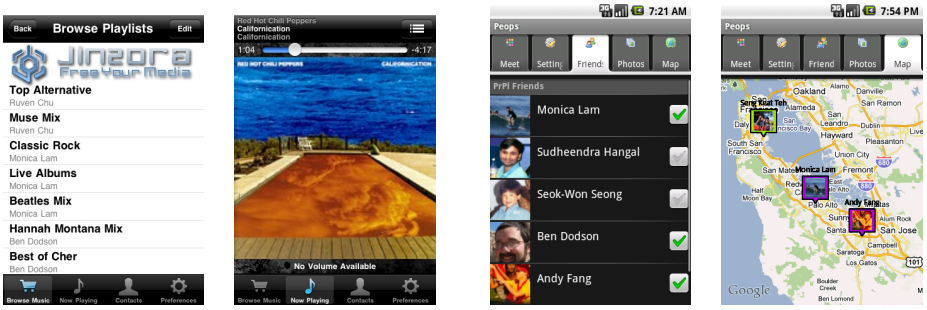

Figure 3.3: Jinzora and Peops Applications on iPhone and Android.

3.2 Infrastructure

We have developed a fully functional prototype of the PrPl architecture as described in this paper. The prototype 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, 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 Directory 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. We recently added support for REST API. 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. To minimize data loss from a crash or failure, the system periodically checkpoints the index states.

The SociaLite implementation also makes use of the Jena/ARQ libraries for iterating over triples and XStream us 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 metadata 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

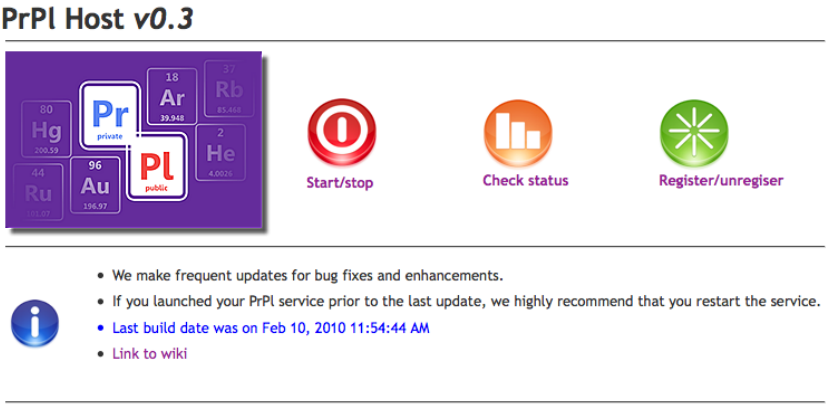

Figure 3.4: PrPl Butler Hosting Service

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 and photos available with the GData interface. Most of these Data Stewards are only a few hundred lines of code.

The PRPL semantic filesystem interface was built on top of FUSE-J, a Java API that uses JNI bindings to the FUSE [6] library. FUSE makes it possible to implement a fully functional filesystem in a userspace program. By implementing the FUSE-J file system calls with the appropriate calls to access the PRPL system.

To provide developers a run-time platform for testing their applications without worrying about setting their own PrPl Butlers, we provide a PrPl hosting service (Figure 3.4). Test developers can register on the web to get an instance of their own PrPl service in minutes. Today, we have approximately $2 \sim 30$ Butlers hosted on our sites.

3.3 APIs

We aim to create an open standard for building distributed social networking applications. By building and testing a number of applications to complement the

infrastructure design, we have gained insight and experience. We also gathered external developers' feedback to continuously refined the interface designs. The interfaces are defined for interactions between the Butler and applications, between the Butler and Data Stewards, between the Butlers, and with the Directory.

3.3.1 Application API

For application developers, we provide common interfaces to access data, manage users and groups, control sharing policies, and authenticate users. The following sections describe some of the key interfaces in the application API.

Data Access: Data access API primarily consists of calls to retrieve, create, or modify resources in the PrPl semantic index and blob data in Data Stewards. It also includes a SociaLite query interface to describe and execute more complex data retrieval that crosses multiple administrative domains.

API	Description
createResource(Metadata)	Create a new blobless or blob resource, initialize with the provided metadata, and returns the URI of the new resource.
getResource(URI)	Return the resource and all the metadata.
destroyResource(Resource)	Destroy all metadata of the resource.
getMetadata(Resource, Property)	Retrieve all property metadata of the resource.
add/removeMetadata(Resource, Property, Values)	Add/Remove property metadata, by inserting new RDF triples or by remove existing RDF triples from the index.
getResourceBlobTicket(Resource, Right)	Acquire a blob ticket to read or write a blob from the Data Steward.
get/setBlob(Resource, Ticket)	Read/Write a blob. The blob ticket provides one or more locations of Data Stewards hosting the resource blob. The Steward verifies the ticket.
runSociaLiteQuery(Query)	Run a (distributed) SociaLite query. The infrastructure handles sub-query forwarding, credential authentication, and access authorization.

Table 3.1: Data Access API

User and Group Management: User and group management APIs (shown in Table 3.2 & 3.3 respectively) provide calls to add new friends, tag and manage relationship, and organize personal groups. A user can have more than one PrPl identities, each representing a different persona. The user list and names may be be shared, however, the identity metadata (e.g. tie strength or tags) are kept private. A group can be either personal or shared. A personal group (e.g. a family group) keeps its membership private and a shared group (e.g. a research group) makes the membership list available to the group members.

API	Description
createIdentity(User ID, IsSelf)	Add a new identity representing a user's own persona or another user.
get/removeIdentity(Identity)	Retrieve/Delete the identity and all the metadata associated with it.
add/removeMetadata(Identity, Property, Value)	Add/Remove tags (e.g. "office mate", "high school friend") or other user-defined properties (e.g. "tie-strength").

Table 3.2: User API

API	Description
createGroup(Group Name, IsShared)	Add a new personal or shared group by given a name.
get/removeGroup(Group)	Retrieve/Delete the group and all the metadata associated with it.
add/removeMetadata(Group, Property, Value)	Add/Remove tags (e.g. "family", "friends", or "work") or other user-defined properties.
add/RemoveIdentityToGroup (Group, Identity)	Add/Remove a member to/from the group.
getPublicGroup()	Retrieve the built-in Public group. By default, everyone is a member of the Public group.
getFriendsGroup()	Retrieve a built-in Friends group. By default, all the user's contacts are in the Friends group.

Table 3.3: Group API

Access Control: Access control API provides calls to specify sharing policies and check user permission. Currently, the smallest unit of sharing is a resource

meaning a user can read and write all of its metadata (and a blob) of the shared resource. A *service resource* is unique as granting an access to a *service resource* implies a permission to use the application that the resource describes. When a user logs into the Butler homepage, the upper menubar displays a collection of applications that a user can run.

API	Description
share/unshareWith (Resource, Identity/Group, Right)	Share/unshare a resource with the identity or group, granting a read or write permission.
checkAccess(Resource, Identity/Group, Right)	Check if the identity or group has an access to the resource.
makePublic/Private(Resource)	Share the resource with everyone/Make the resource private.
getAllSharedWith(Identity/Group)	Retrieve all the resources that are shared with the identity or group.

Table 3.4: Access Control API

Authentication and Session Management Authentication and session management APIs (shwon in Table 3.5) handle verifying user credentials, acquiring, and renewing a session ticket. Every request to the Butler in exception of a few authentication and session API calls, must present a valid session ticket. Since the session ID is random and unique, the session ticket signature needs to checked only once.

3.3.2 Data Steward API

Data Steward coordinates with the Butler to keep the metadata consistent and provides interfaces for applications to access blobs directly. The Data Steward API (shown in Table 3.6) consists of registering the device running the Steward and performing blob replications.

For every blob, the Butler assigns a primary Data Steward. A blob may be replicated to multiple Stewards. Applications can read a blob from any of the replicas but must write to the primary Steward. At the Butler, a blob resource can be in either a READY or PENDING-WRITE state. The READY state indicates that the blob

API	Description
getAuthorizationToken()	Retrieve an authorization token and use it to login.
getSessionTicket(Authorization Token)	After a successful login, retrieve a session ticket assigned for the authorization token.
getSessionTicket(User ID, Password)	Retrieve a session ticket by directly passing in user ID and SHA1 hash of user password. This is only available if the owner is accessing own Butler.
renewSession(Session ID)	Request a renewal for the session ID prior to its expiration. The Butler issues a new session ticket with the same session ID and updated the expiration and signature.
destroySession(Session ID)	Destroy the current session.

Table 3.5: Authentication and Session APIs

is available for read or write. The PENDING-WRITE state indicates that the blob is either being written by an application or replicated. During that time, a blob can be only read from the primary Steward.

API	Description
register/unresiterDevice(Device, Updates)	Register/Unregister the Data Steward with/from the owner's Butler.
sendHeartbeat(Updates)	Send a still-alive signal with any blob updates or device status changes (e.g. IP address or storage capacity).
getBlobState(Resource)	Retrieve the blob state (READY or PENDING-WRITE).
setBlobState(Resource, State)	Set the blob state.
getDataStewards(Resource)	Retrieve currently available Stewards for the blob resource.
getReplicaCount(Resource)	Retrieve the number of blob replicas.

Table 3.6: Data Steward API

3.3.3 Directory API

Directory provides an interface to find or update a user's Butler information. A user entry in the Directory contains the user ID, user name, user's OpenID, and location of the Butler, and can be looked up, provided a user ID. Table 3.7 summaries the Directory API calls.

API	Description
getEntry(Identity)	Retrieve a user entry in the Directory.
removeEntry(Identity)	Remove the user entry in the Directory.
updateEntry(Identity, Updates)	Update the user entry with the new user/Butler information.

Table 3.7: Directory API

Chapter 4

Experiments and Experience

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.

4.1 Performance on a Single 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 4GB of memory.

The queries for this experiment return the URI of the photos that match one and two given tags, respectively. Because users would like to see partial results as soon as possible, the Butler pipelines the queries and starts reporting the results without waiting for all the data to be computed. Figure 4.1 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.

While the first query is simpler, it returns many more results. Thus, even though

# of Triples	Database Size (MB)	*One Tag*		*Two Tags*	
		1st/Last (sec)	# of Answers	1st/Last (sec)	# of Answers
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 4.1: Single Butler Queries Matching One and Two Photos Tags

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 the performance. By pipelining the query, the Butler returns the first results in 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.

4.2 Performance on a Network of Butlers

To simulate a social network, we deployed 100 Personal-Cloud Butlers (each of them 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 [14] 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. The performance of the queries is summarized in Table 4.2 and shown in Figure 4.1.

Query	Description	# Rules in Query	# Butlers Reached	Answers	1st/Last (sec)
Common friends	Common friends b/t 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

Table 4.2: Distributed SociaLite Queries

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 all together 494 tuples.

Friends star. This query computes the closure on the people that one can reach by following friendship connections in the network. The issuer contacts his immediate friends, who then recursively computes one's own 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 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 most popular 10 songs to the user. This query also shows off the relation-wise function extension in SociaLite.

We first observe from Table 4.2 that these queries can be expressed succinctly in 2 to 6 SociaLite rules. For all these queries, the client application contacts the

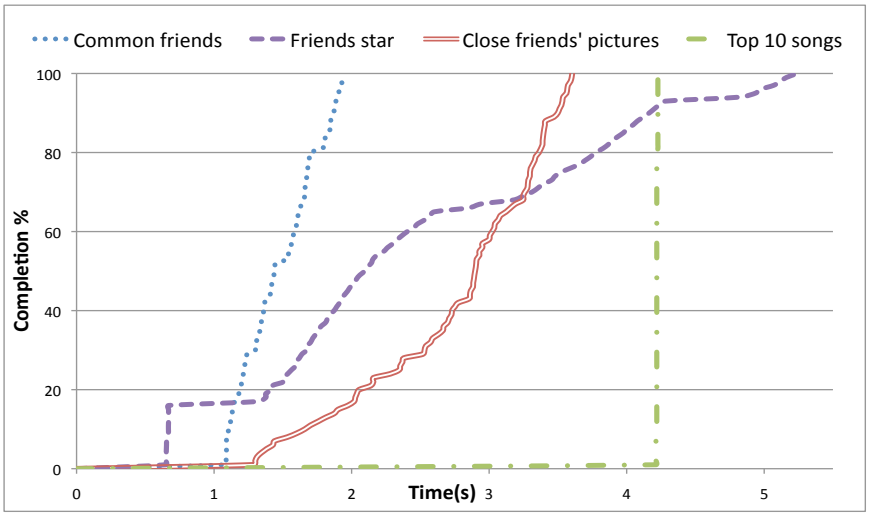

Figure 4.1: Experimental Query Results

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.

4.3 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. This is much more convenient than the current practice of inviting friends for every different website we use to share data. We were able to to quickly develop a number of applications on various platforms, using the high-level abstractions the infrastructure provides. We

now describe our experience building Peops and Jinzora Mobile.

4.3.1 Peops: Sharing Personal Information

Peops is an Android interface to the Butler. It 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.

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. Only the URIs of photos are passed around and the 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.

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.

4.3.2 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.

Chapter 5

Related Work

OpenSocial [8] 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 [1] 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 [13] 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 [24] 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 [9] 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 [16] 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.

Jim et al. explore different evaluation methods for distributed Datalog [17]. 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 [18]. 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 [19]. 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 [20]. IRIS [10] and Gnowsis are single-user semantic desktops while social semantic desktop [11] and its implementations [23][15] 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.

Chapter 6

Conclusion

The paper presents the architecture of PrPl, an instance of 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. SociaLite extends Datalog with an According-To operator for specifying distribution and user-defined functions.

We have implemented a fully working prototype of the PrPl system and have developed a number of social applications on it to demonstrate various features of PrPl. We found that the applications are easier to write with PrPl as the system provides an unification of data, a high-level distributed query language, and takes care of underlying authentication, and access control.

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.

The experience helped us to identity additional challenges and opportunities for research. While our real-life relationships are intricate with varying degrees of tie strength, current social networking sites provide only a flat relationship for everyone. We frequently share with small but dynamic subsets of friends, yet there is no easy

way to systematically organize them into logical groups. To determine who our close friends and groups are, we are looking at our daily interactions in popular communication tools (e.g. email messages, SMS, call logs, etc). Personal social topology that we mine can make sharing even simpler for users.

It is important to answer not only who we share with, but what and how we share. For instance, we want to share all our photos taken from a trip only with the friends who were on the trip or to share a current location only if a friend is nearby. We are evaluating a few approaches to easily specify and efficiently enforce these complex and dynamic real-life sharing policies.

We are working to improve SociaLite performance and to support real-time updates. We can dynamically profile queries and recompile new incoming queries for better performance. As many queries do not ask precise answers, lazy evaluation and approximation can help to return the results quicker. An unstructured RDF-based database gives flexibility and easy programmability, but performance suffers due to its generality. As we were building a number of social applications, we have seen common patterns both in user data and queries. The persistent store and query engine can take advantage of them to provide higher performance without losing the benefits.

It is ineffective and wasteful to flood the queries to everyone. By understanding what a user is searching for and who is best person to ask, we can intelligently forward the query to people who most likely have an answer. Ability to classify and infer person's expertise and interests can help further personalizing the Butler for each user. Lastly, we are interested in ways to evaluate queries by revealing little or no information to the other party. For instance, it is useful to find common interests without disclosing everything that a user is interested in.

In conclusion, we believe that lowering the barrier to entry for distributed social application developers is key to making decentralized social networking a reality. The proposed decentralized, open, and trustworthy PrPl platform 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.

Bibliography

[1] http://developers.facebook.com/.

[2] http://yadis.org.

[3] http://www.ietf.org/rfc/rfc4122.txt.

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

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

[6] http://fuse.sourceforge.net.

[7] http://www.w3.org/TR/rdf-sparql-query/.

[8] http://www.opensocial.org/.

[9] Randy Baden, Adam Bender, Neil Spring, Bobby Bhattacharjee, and Daniel Starin. Persona: An online social network with user-defined privacy. *SIGCOMM Comput. Commun. Rev.*, 39(4):135–146, 2009.

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

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

[12] Herve Gallaire and Jack Minker, editors. *Logic and Data Bases*. 1978.

[13] Roxana Geambasu, Magdalena Balazinska, Steven D. Gribble, and Henry 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.

[14] Eric Gilbert and Karrie 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.

[15] Tudor Groza et al. The NEPOMUK project - on the way to the social semantic desktop. In *Proceedings of I-Semantics' 07*, pages 201–211, 2007.

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

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

[18] 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.

[19] Daniel Peek and Jason 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.

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

[21] David Recordon and Drummond Reed. Openid 2.0: A platform for user-centric identity management. In *DIM '06: Proceedings of the Second ACM Workshop on Digital Identity Management*, pages 11–16, 2006.

[22] Drummond Reed, Les Chasen, and William 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.

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

[24] Amin Tootoonchian, Stefan Saroiu, Yashar Ganjali, and Alec 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.

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