HomeViews: Peer-to-Peer Middleware for Personal Data Sharing Applications

Roxana Geambasu, Magdalena Balazinska, Steven D. Gribble, and Henry M. Levy
Department of Computer Science and Engineering
University of Washington, Seattle, WA
Email: {roxana,magda,gribble,levy}@cs.washington.edu

ABSTRACT

This paper presents HomeViews, a peer-to-peer middleware system for building personal data management applications. HomeViews provides abstractions and services for data organization and distributed data sharing. The key innovation in HomeViews is the integration of three concepts: views and queries from databases, a capability-based protection model from operating systems, and a peer-to-peer distributed architecture. Using HomeViews, applications can (1) create views to organize files into dynamic collections, (2) share these views in a protected way across the Internet through simple exchange of capabilities, and (3) transparently integrate remote views and data into a user's local organizational structures. HomeViews operates in a purely peer-to-peer fashion, without the need for account administration or centralized data and protection management inherent in typical data-sharing systems.

We have prototyped HomeViews, deployed it on a small network of Linux machines, and used it to develop two distributed data-sharing applications: a peer-to-peer version of the Gallery photo-sharing application and a simple read-only shared file system. Using measurements, we demonstrate the practicality and performance of our approach.

Categories and Subject Descriptors

D.4 [**Operating Systems**]: File Systems Management, Security and Protection; H.3.3 [**Information Systems**]: Information Search and Retrieval; H.2.4 [**Database Management**]: Systems – Distributed Databases, Query Processing

General Terms

Design, Management, Security

Keywords

Personal information management, access control, capabilities, peer-to-peer, search

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'07, June 11–14, 2007, Beijing, China.
Copyright 2007 ACM 978-1-59593-686-8/07/0006 ...$5.00.

1. INTRODUCTION

The volume of personal data created by home users far outpaces their ability to manage it. Inexpensive storage, powerful multimedia appliances (*e.g.*, digital cameras, iPods, TIVOs), and new applications for creating and editing digital content provide home users with tools to generate enormous quantities of digital data. As a consequence, users face several challenges: they need to organize files into directories, search through large volumes of personal data to find objects of interest, and manually share their data with family, friends, and others connected through broadband networks.

These challenges have motivated applications such as *desktop search tools*, which help users to locate and organize files using queries and views [14, 38, 28]. Similarly, new peer-to-peer [3, 26] and Web-based [9, 43] *file-sharing systems* help users to share their data. However, such tools fall short for three reasons. First, they are not integrated with each other or with other applications; therefore users must often employ several independent tools to manipulate, search, organize, and share their data. Second, distribution is still visible and heavyweight in most of these tools, requiring manual uploads and downloads. Third, many sharing tools do not deal with dynamically changing data collections, forcing users to take action every time they update shared data or add files to a shared collection. Overall, using today's data organization, search, and sharing services is far from effortless for users.

Our goal is to simplify the creation of a new generation of *personal data management and data sharing applications.* To do this, we have designed and implemented *HomeViews*, a middleware layer that provides a set of powerful application services. Similarly to a DataSpace Support Platform (DSSP) [10, 15], which provides services to applications operating on a user's or organization's "dataspace," HomeViews provides services to applications that operate on a user's personal and shared data files. HomeViews' abstractions and services are thus geared toward personal data organization and sharing. Specifically, HomeViews supports:

- the creation of database-style views over a user's file repository,
- a lightweight protection mechanism for selective granting (and later revocation) of view access to remote users,
- seamless sharing and integration of local and remote data and views, and

- peer-to-peer communication between HomeViews instances on peer computers.

Using HomeViews, applications can leverage flexible organization and transparent sharing of distributed objects. For example, a photo-album application built on HomeViews enables users to create and share dynamic photo albums with their friends, and to integrate their friends' shared photos with their own. The application focuses on high-level abstractions (albums in this case), while issues such as view creation, protection, and distributed query execution are managed by HomeViews.

HomeViews' peer-to-peer structure provides direct ad-hoc data sharing between peer nodes. It requires no centralized servers or services, no user identities, and no user registration. All distribution is handled by HomeViews and is transparent to the applications. HomeViews views are *dynamic*: users can share views of changing data sets, rather than just static copies of their files.

A crucial feature of HomeViews is its simple, lightweight, and flexible protection mechanism for controlling access to shared views. Protection in HomeViews is based on capabilities, a protection model developed in the context of object-based operating systems [5, 23, 37, 42]. A capability to a view is a data structure that binds together a global view name with access rights to that view. Users grant each other access to their data simply by exchanging capabilities to their views, much like users share access to private Web pages by exchanging URLs. We show that capabilities are well matched to the goals of ad-hoc sharing in peer-to-peer environments that lack (or shun) the identities and coordinated management of common protection structures, such as user accounts or access control lists (ACLs).

To simplify application development and to support sophisticated queries, HomeViews provides a declarative query language interface based on SQL. We show that a capability-based access control model can be easily integrated into SQL, requiring only a small set of changes. The resulting language, called *SQLCapa*, enables definitions of new views atop previously defined local and remote views, and the subsequent sharing of these views without coordinated protection management. Capabilities also enable rewriting and optimization of distributed queries, leading to good query execution performance.

We have prototyped HomeViews and built two applications on top of it. Our current implementation targets read-only data sharing for medium-sized peer communities (e.g., hundreds of users). This paper presents the HomeViews design, our experience building high-level applications on HomeViews, and measurements that validate our approach.

The rest of the paper is organized as follows. Section 2 provides a more in-depth motivation for the features of HomeViews. Section 3 presents a high-level overview of the system and gives a detailed technical description of its main components: the capability-based access control, the SQL-based query language, and the distributed query execution. Section 4 describes how to build applications on top of HomeViews. We evaluate the performance of HomeViews in Section 5. Section 6 discusses previous work, and we summarize and conclude in Section 7.

2. MOTIVATION

As outlined above, personal data management needs are characterized by three key requirements. First, people need powerful but simple tools to *organize their files*. Traditional organizational structures – static, hierarchical file directories – fall short when users' data collections grow large. As a result, desktop search tools, such as Google Desktop [14] or Spotlight [38], have emerged. These tools index the user's files and support keyword or attribute-based search. Tools such as Spotlight also provide organizational help in the form of *smart folders* or *views*, which are dynamic collections of files populated by results from searches. For example, when the user creates a new file, the file appears in all the appropriate smart folders, based on its contents, extended attributes, or other metadata (such as ID3 tags for audio files).

Second, people want to *share data with friends, family, and colleagues across the Internet.* However, the ability to share selectively within a small trusted community is limited. While email remains one of the most commonly used data-sharing tools, it is inappropriate for sharing large or dynamically changing data collections. Hosting services such as Flickr [9] and YouTube [44] have become popular for photo and video sharing, respectively. But these services are centralized and users must register with them to manually upload content. For protected sharing, recipients must register to view and download content. Ultimately, users must trust the service with the storage and control of their data. This issue has become problematic in light of recent government data requests to Internet services such as Yahoo and Google [41].

An alternative to centralized services is peer-to-peer (P2P) file sharing [3, 26]. P2P systems are designed for data sharing within communities, particularly where data is published to the entire community. However, P2P systems are not usually intended for selective sharing, i.e., sharing data with restricted sets of users within the community.[1]

Finally, in a distributed data-sharing environment, people want to *seamlessly integrate shared files* (*i.e.*, files made

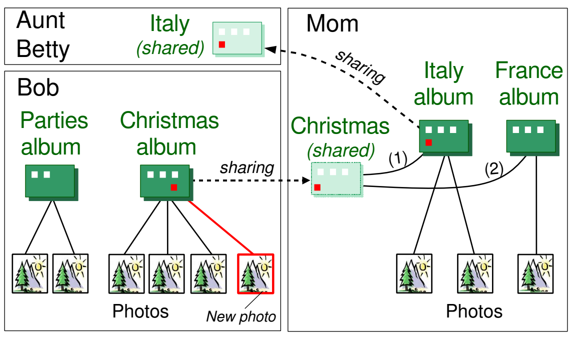

Figure 1: A simple photo organizing and sharing scenario. Bob shares his Christmas photo album with Mom. When Mom organizes her photos, some of Bob's Christmas photos end up in Mom's Italy album (1) while others go into Mom's France album (2). Sharing is dynamic: when Bob creates a new photo of Christmas in Italy, it appears automatically in all appropriate albums.

[1]The DC++ [6] system supports selective sharing with groups of users (called private hubs); however, it requires the user to create and administer a hub (including keeping track of the hub's password) for each set of files shared with a different set of people.

available by other remote users) with their own local files. That is, they want to access data in a location-independent fashion. With current systems, users must manually copy or download shared data onto their home machines in order to include them in their local organizational structures. This is cumbersome, inefficient, and static.

To make these problems concrete, consider the following simple family photo-sharing scenario. Figure 1 shows three members of a family: Bob, his Mom, and his Aunt Betty. Bob and Mom are well-organized people who label their photos as they upload them from their cameras, noting the place, the date, the occasion when the pictures were taken, and other attributes. Using their annotations, they choose to *organize* their photo collections into albums. However, Bob and Mom organize their albums differently. Bob likes to create albums based on occasion: parties, Thanksgiving, Christmas, and so on. Mom prefers to organize her albums based on where the photos were taken: *e.g.*, Home, Italy, France, etc. Bob and Mom would like these albums to be automatically populated, just like smart folders, with results from queries over the photo metadata.

Also, Bob wants to *share* his Christmas photos with Mom, while keeping his party photos private (sharing them only with his close friends). Mom would then like to *integrate* Bob's shared photos into her own photo repository, but she wants to organize her global collection (including Bob's photos) according to her own scheme – the place where the pictures were taken. So, when Mom looks at the Italy photos, she might find Bob's photos of their Christmas vacation in Italy. Similarly, her France photos will include Bob's photos from another family Christmas in France.

Mom knows that Aunt Betty loves Italy and decides to share her Italy album with her. Aunt Betty should then be able to organize all the photos (Mom's Italy photos and Betty's own photos) in whatever way she wishes and further share her albums with her own friends. Aunt Betty's new organization should include the photos she received from Mom (and via Mom, from Bob).

Finally, everyone in the family wants the photo sharing to be *dynamic* and *transparent*. For example, when Bob creates a new photo from a Christmas vacation in Italy, this photo should automatically appear in all the appropriate albums.

At a high level, then, this scenario suggests several requirements: (a) data is organized into views populated by queries over a base set of files; (b) views must be directly sharable with trusted parties, without the need to register with a service; (c) it should be possible to integrate remote data shared by others with local data; and (d) views are dynamic, their contents must change as underlying data are added or removed.

Our goal is to simplify the implementation of such dynamic personal-data-sharing applications by providing a set of common services for organization, protected sharing, and integration of data. In the next section we describe HomeViews and how it fulfills the requirements listed above.

3. ARCHITECTURE AND IMPLEMENTATION

HomeViews provides services that allow applications to *create, compose, and query dynamic, location-independent collections of files* and to *share them in a peer-to-peer environment.* After we present the high-level architecture of our system, we describe in detail our implementation of HomeViews, focusing on capability protection, query language, and query processing.

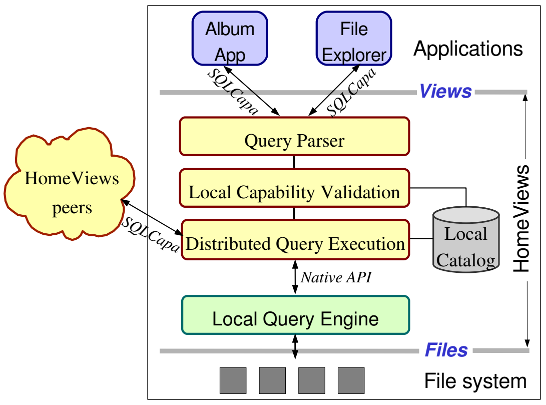

Figure 2: HomeViews system architecture.

3.1 Architectural Overview of HomeViews

Figure 2 shows the HomeViews system architecture. HomeViews sits between applications and the underlying file system. It presents applications a view-based interface to the filesystem. It executes queries over the local file system and communicates with other peers to evaluate distributed queries. More specifically, HomeViews performs the following basic functions, which correspond to its internal structure shown in the figure.

First, HomeViews exposes to applications a database view abstraction of the file system. Applications use this view abstraction to define dynamic collections of files on top of a user's local file storage. For example, an application can create a view of all party photos. To create or query views, applications issue requests to HomeViews using a flexible query language, called *SQLCapa* (described in Section 3.3). The HomeViews query parser receives requests and parses them into internal data structures.

Second, HomeViews provides a lightweight access control scheme for views that is based on capabilities [42, 27, 23, 37]. These functions are managed by the HomeViews local capability validation layer. When an application creates a view using SQLCapa, HomeViews validates the request, creates the view and a new capability for the view, registers the new view and capability in its local catalog, and returns the capability to the application. With the help of applications, users can grant each other access to views simply by exchanging capabilities. We describe HomeViews' capability-based access-control further in Section 3.2.

For every incoming query, the capability layer determines whether the query is on a local view. If so, it uses the local catalog to validate the request. Invalid queries return with an error. If the query is on a remote view, the capability layer forwards it to the distributed query execution layer.

Third, the distributed query execution layer shown in Figure 2 is responsible for executing queries. It uses the local query engine to evaluate local queries and communicates with peer HomeViews instances to validate and solve distributed queries. In this way, HomeViews offers com-

128 bits	128 bits	32 bits
global view ID	password	IP hint

Figure 3: Capability for a view.

Node-local view table (ViewTable)

global view ID	view definition	other attributes
...	...	...

Node-local capability table (CapTable)

global view ID	password	rights
...	...	...

Figure 4: Capability and view catalog tables.

plete location independence: applications access views in the same way no matter where these views have been defined. HomeViews uses a pull-based data access method, similar to that of the WWW: a user sees updates to a view only after re-evaluating the view. We present HomeViews' query execution algorithms in Section 3.4.

Below the distributed query execution layer, HomeViews sees the file system as a database with a local query engine that provides indexing and keyword or attribute-based search functions. At this layer, we leverage existing tools for desktop search. Our prototype uses Beagle [2], a keyword-based desktop search engine for Linux, which is similar to Spotlight [38] or Google Desktop [14].

Since HomeViews is a middleware layer, users do not interact directly with it. Instead, they interact with applications that hide views, capabilities, and the query language behind application-specific abstractions and graphical interfaces. For example, the photo application envisioned in our example scenario displays albums and photos to users. Underneath, it uses HomeViews views to populate these albums and capabilities to access the views.

3.2 Capability-based Access Control

This section describes capability-based access control in HomeViews. We introduce capabilities in general, describe HomeViews' capability implementation, and discuss data sharing using capabilities. Overall, we show that our protection approach is lightweight: it enables selective sharing and revocation while incurring little administrative overhead.

3.2.1 Background

Conceptually, a capability consists of a *name*, which uniquely identifies a single object in the Internet, and a set of *access rights* for that object. HomeViews capabilities protect views and enable view sharing. A capability represents a *self-authenticating permission* to access a specified object in specified ways. It is like a ticket or door key: possession of a capability is proof of the holder's rights to access an object. Without a capability for an object, a user cannot "name" or access the object.

To be self-authenticating, a capability must be *unforgeable*. That is, it must be impossible to fabricate a capability, to modify the rights bits in a capability, or to change the "name" field to gain access to a different object. Previous systems have guaranteed this property in various ways. These include encryption [40], storing capabilities in the OS kernel [42], or using hardware tag bits to prevent modification to capabilities in memory [17]. HomeViews uses a *password-capability model* [1, 5], in which the integrity of a capability is ensured through the use of sparse random numbers (called passwords) in an astronomically large space.

3.2.2 Naming and Access Control in HomeViews

A HomeViews capability has three parts (Figure 3). First, a 128-bit *global view ID* uniquely identifies an individual view in the Internet; no two views have (or will ever have) the same ID. To achieve this property, each HomeViews instance creates a global view ID by concatenating a hash of the local node's MAC address[2] with a locally unique-for-all-time view ID. Minting of new capabilities is therefore a local operation for the HomeViews instance on a node and requires no coordination with other nodes.

Second, associated with each capability is a 128-bit random password that ensures the capability's authenticity. The protection is thus probabilistic, but the probability of guessing a valid capability is vanishingly small. To forge a HomeViews capability requires guessing a 256-bit number consisting of both a valid 128-bit view ID along with its associated 128-bit password.

A 32-bit *IP hint* field in the capability contains the IP address of a node that likely contains or can locate the object addressed by the capability in the P2P network. In general, we expect that objects will not move in our network, and the IP hint will be the address of the node that created the capability and still holds its definition. If the hint fails, we fall back on a conventional distributed hash-table scheme for location [39]. In this case, IP hints serve as entry points for a new node to join the peer-to-peer network.

Figure 4 shows the per-node catalog tables that hold view and capability information. For each view created on a node, there is one entry in a local view table (*ViewTable*). The ViewTable entry contains the global view ID, the view definition, and other attributes (such as the human-readable view name). While a capability identifies only one view, multiple capabilities for the same view can exist (see Section 3.3.3).

A node's capability table (*CapTable*) contains one entry for each capability minted to a *locally known view.* The CapTable entry stores the global view ID of the named view, the password, and the access rights. Storing access rights in the system catalog rather than in the capability itself eliminates the need for encryption or other mechanisms to protect the capability's rights from being forged.

When HomeViews receives a capability, it uses the IP hint to determine whether the capability is for a local view. If the capability is local, HomeViews checks whether the <global view ID, password> pair in the capability matches a <global view ID, password> pair in CapTable. If so, the capability is valid, and HomeViews then examines the access rights in CapTable to see if the requested operation is permitted. If the capability is not found in CapTable or the operation is not permitted, the request fails. If the capability is for a remote view, HomeViews forwards the request to the appropriate node in the peer-to-peer network, which then performs the validation itself.

To revoke a capability, HomeViews simply removes an entry from the CapTable. Once a capability is revoked, all queries issued on that capability will fail. Of course, if a

[2]The MAC address is the unique identifier of the node's network card.

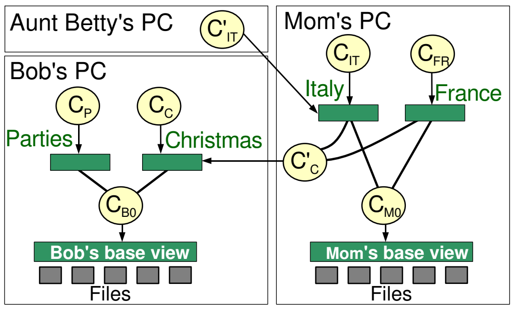

Figure 5: Solving the photo sharing scenario with views and capabilities. Rectangles denote views, *e.g.*, 'Christmas' is the view of all Bob's Christmas photos; circles denote capabilities, *e.g.*, C_C is Bob's capability to his 'Christmas' view. C'_C is a copy of C_C that Bob sent to Mom to share the 'Christmas' view with her. Mom's 'Italy' view composes two capabilities, C'_C and C_{M0}.

user with a capability has made a local copy of the shared data, revoking the capability cannot prevent them from distributing that copy. It does, however, prevent the holder from executing a query and seeing new or modified files that would result from that query.

Because a capability is independent of the person using it, HomeViews' access control scheme requires no user identities. Thus, sharing in a capability-based model requires no user accounts, no user authentication, and no centralized protection structures.

3.2.3 Sharing

Capabilities facilitate data sharing because they can be easily passed from user to user as a way to grant access. Figure 5 illustrates the use of capabilities for the dynamic album-sharing scenario we previously presented. An album corresponds to a HomeViews view; it is is accessed through a capability and populated with results from querying the view. For example, the figure shows that Bob has a capability, C_C, for his 'Christmas' album. Bob shared this album with Mom by giving her a copy of his capability (shown as C'_C) for that album. When Mom looks at her Italy album, her application uses that capability to query Bob's album, integrating the results with those from her local files.

As previously noted, a capability is a 288-bit data structure (shown in Figure 3), which names and protects a view. In this form, a capability would be difficult for users to deal with. However, applications typically store capabilities as part of their internal representation of the abstractions they support (such as albums in our example). Applications can then convert capabilities to a different form when passing them to users. This form is a human-readable text string that we call a *token*. For example, when a user requests a capability from our photo-album application, the application produces a URL as a token. The URL is simply a text string that includes the capability, encoded as a numeric string, as an embedded parameter. The user can then treat that URL as a capability, storing it, passing it to an application, or passing it to others as he wishes.

Returning to our previous scenario, to share his Christmas album, Bob obtains a token for it from his application. He then emails it to Mom, just as he would email her a URL to a Web page. Mom then presents the URL to her local photo application as proof of her right to access Bob's album. Given the URL, the application extracts the capability and executes HomeViews queries on the remote view to populate the local albums. Of course, tokens are best sent in secure email.[3]

As mentioned previously, sharing capabilities is similar to exchanging URLs to folders holding a user's private files, except that capabilities enable selective revocation and provide a greater range of access rights (as we discuss in Section 3.3). Like URLs, a user must trust his friends not to forward capabilities to people who should not have access to the data. Our protection system is intended to enable selective sharing within trusted communities, while preventing access from disconnected third parties; however, it cannot prevent cheating by trusted peers. Similar trust assumptions exist for other protection mechanisms, as well. For example, consider an access control list (ACL) scheme. When a user adds his friend to an object's ACL, nothing stops the friend from copying the object and distributing it further, or from acting as an invisible proxy for unwanted third parties.

Overall, we believe that several features of capability protection make it perfectly suited for our target environment of protected peer-to-peer data sharing. Capabilities are easily exchanged using simple channels that users are accustomed to. They are anonymous, require no identities, no global authentication, and no centralized or distributed management of protection structures. In these ways, they are consistent with the goals of a P2P system.

3.3 Query Language

To facilitate application development, HomeViews offers a flexible relational query language interface that enables sophisticated queries over files. HomeViews' query language, SQLCapa, is a modified version of SQL with integrated capability-based access control. Our SQL modifications are simple and intuitive, and the resulting language is easy to understand and enables seamless view sharing and composition across peers.

HomeViews models the file system as a single relation, called `Files`. Each tuple in the relation represents one file. All files that are indexable by a desktop search engine are included in the `Files` relation (*e.g.*, annotated files, Office documents, emails, etc.). The schema of the relation is the set of all known file attributes (*e.g.*, name, author, date, music genre, photo resolution). File contents are also included in the relation either under the 'text' or 'binary' attributes. If a file does not support an attribute, it has a `NULL` value for that attribute. Using a single relation permits files of different types to be returned as part of a single query.

On this relation, applications define views using predicates on file attributes and content. Views can also be composed with union, set difference, and intersection operators.

Table 1 summarizes SQLCapa's modifications to SQL. We now describe these modifications in more detail.

3.3.1 Specifying Capabilities With Queries

To execute queries on views, HomeViews client applications must present appropriate capabilities as proof of authority. This requirement is reflected in SQLCapa. Because a capability identifies exactly one view, capabilities are used

[3]To simplify the presentation, we consider capabilities and tokens to be identical for the remainder of the paper, referring to both forms as capabilities.

New/modified statement			Return Type	Meaning
SELECT *	FROM	*Cap* [WHERE ...]	Relation	Query
CREATE VIEW SELECT *statement*	<ViewName> [UNION/...	AS SELECT *statement*]	Capability	Create a view and a capability to the view.
SELECT *	FROM	CATALOG OF *Cap*	Catalog info	Look up capability in catalog
CREATE	BASEVIEW		Capability	Create the base view
DROP VIEW	*Cap*		Void	Drop view associated with capability
ALTER VIEW	*Cap* ...		Void	Modify view associated with capability
RESTRICT	*Cap*	RIGHTS *rights*	Capability	Create new capability to same view, possibly with different access rights
REVOKE	Cap_1	USING Cap_2	Void	Revoke capability Cap_1

Table 1: SQL modifications. *Cap*, Cap_1, and Cap_2 are capabilities; *rights* is a string that encodes access rights

directly to name views in the FROM clause of the SELECT statement. With this approach, the semantics of SELECT statements remain unaltered. Only the view naming scheme changes. Therefore, the query:

```
SELECT * FROM $C'_C$
WHERE date > '2006-01-01'
AND place = 'France'
```

returns Bob's France Christmas photos taken after Jan. 1st 2006. C'_C is Mom's capability to Bob's view (see Figure 5), date is the file creation date, and place is the attribute indicating the location where the picture was taken. This query selects all attributes, including the file content.

HomeViews also supports keyword queries with a simplified form of the CONTAINS predicate used by SQL Server [13]. In HomeViews, the predicate takes the form: CONTAINS (column, 'k_1, k_2, ..., k_n'), where column indicates the column to search, and k_1 through k_n are the keywords that must be present for the result to match the query. For example, if each photo has an attribute description, the query:

```
SELECT * FROM $C'_C$
WHERE CONTAINS (description,'snow')
```

returns all of Bob's Christmas photos with snow (or more precisely, those photos that include the keyword "snow" in their 'description' attribute).

3.3.2 Creating Views

Views are created using the standard CREATE VIEW statement. Once again, capabilities serve to name the underlying views. More importantly, the CREATE VIEW statement *returns* a capability to the newly created view. This initial capability has all rights enabled. Thus, the query:

```
CREATE VIEW Italy AS
SELECT * FROM $C_{M0}$ WHERE place = 'Italy'
UNION
SELECT * FROM $C'_C$ WHERE place = 'Italy'
$\rightarrow C_{IT}$
```

creates the 'Italy' view for Mom. The right arrow denotes the returned capability, C_{IT}, for the new 'Italy' view. This view is a seamless composition of a local view (Mom's files, specified by C_{MO}) and a remote view (Bob's shared files, specified by C'_C). Similarly, applications specify capabilities instead of view names to ALTER or DROP views.

To bootstrap the system, we add a new CREATE BASEVIEW statement that creates the first capability and view in the system for a specific user. The returned capability provides access to the view containing all files in the file system that are visible to the user. The underlying file system's access control determines this set of files. From this initial capability, which has all rights enabled, the user's applications can execute queries and create additional views. As an example, Figure 5 shows Mom's base view and her initial capability, C_{M0}, to that view.

3.3.3 Capability Restriction

To share access to a view, a client application can directly pass the capability returned by the CREATE VIEW statement to the sharee. In most cases, however, users may want to limit their friends' access rights to a view.

To support this operation, we introduce the statement RESTRICT. Given a valid capability X, RESTRICT X RIGHTS rights creates a new capability that refers to the same view as X. The RIGHTS clause enumerates all rights to be *enabled* on the view; only rights already present in X can be carried onto the restricted capability. Before Mom shares her 'Italy' view with Aunt Betty, she can ask her application to give her a restricted capability, C'_{IT}. The application creates that capability by issuing the following statement to HomeViews:

```
RESTRICT $C_{IT}$ RIGHTS SELECT $\rightarrow C'_{IT}$
```

Mom can then email C'_{IT} (instead of C_{IT}) to Betty; this gives Betty the ability to look at the Italy photos, but prevents her, for example, from looking up the definition of the view in the catalog. Currently, HomeViews supports the following rights: SELECT (read), DROP (delete the view), ALTER (modify the view definition), REVOKE (revoke capabilities defined for the view), and CATALOG_LOOKUP (look up the view definition in the catalog). File creation, removal, and updates are currently performed outside of HomeViews, through the file system. Hence, only the owner of a file can modify it.[4]

3.3.4 Capability Revocation

Applications can revoke previously created and shared capabilities with the REVOKE statement. Given two valid capabilities (C_{IT} and C'_{IT}) to the same underlying view, if C_{IT} has the REVOKE right enabled, then the statement: REVOKE C'_{IT} USING C_{IT} revokes capability C'_{IT}. Any subsequent use of C'_{IT} will fail. This REVOKE statement would revoke Aunt Betty's capability to Mom's 'Italy' view.

To revoke a capability, an application must have another capability to the same view that has the REVOKE right enabled. This ensures that arbitrary applications (and their

[4]Similarly, because we focus on read-only sharing in this paper, the HomeViews prototype currently ignores remote requests to alter or drop a view.

users) cannot revoke capabilities. By default, capabilities returned by the `CREATE VIEW` statement have all rights enabled. Thus, the 'owner' of a view (the user on whose behalf it was created) can revoke all capabilities for that view. A reasonable policy is for applications to restrict the capabilities on behalf of users before sharing them and never enabling the `REVOKE` right in these restricted capabilities. HomeViews, however, does not enforce this policy.

3.3.5 Catalog Information Lookup

Views and capabilities are stored in two catalog tables (see Section 3.2). The `CATALOG_LOOKUP` right enables the capability holder to access *only* the attributes corresponding to her capability, `X`, with a statement of the form:

```
SELECT * FROM ViewTable V, CapTable C
WHERE V.GlobalViewID = C.GlobalViewID
AND V.GlobalViewID = GlobalViewID(X)
```

where `GlobalViewID(X)` returns the global view ID of capability `X`.

To simplify catalog lookups, we introduce the shorthand notation `CATALOG OF` to refer to the results of the above query. Mom's application can look up the definition of Bob's view of 'Christmas' photos with the statement: `SELECT definition FROM CATALOG OF` C'_C.

In summary, the main change we propose to SQL – from which most other changes derive – is the use of capabilities to access and name views. Although we currently do not support joins, the query language can easily be extended to include this operator.

3.4 Query Processing

HomeViews can process queries in several ways. The choice depends in part on the `CATALOG_LOOKUP` rights of the capabilities involved in the query.

3.4.1 Query Execution Algorithms

Recursive evaluation. If capabilities do *not* have the `CATALOG_LOOKUP` right, then HomeViews evaluates the query recursively. Recursive evaluation pushes queries from peer to peer down the view definition tree, validating access on each node in the tree. Results are then returned and aggregated hop-by-hop following the same tree. Figure 6 shows the detailed algorithm for recursive query evaluation. Note that HomeViews nodes do not perform arbitrary computation on behalf of other nodes. HomeViews drops queries from remote nodes if they access views that are not locally defined. Our algorithm is best-effort: it returns as many results as are available at the time of the query execution.

Query rewrite and optimization. If capabilities include the `CATALOG_LOOKUP` right, HomeViews first fetches all view definitions by contacting the nodes where the views are defined. It then rewrites the query in terms of *base views* and executes the simplified distributed query. In this approach, each capability is validated during the catalog lookup phase.

In a typical query, different capabilities have different rights, and query evaluation is a hybrid of the above two schemes. Our general model also supports a query optimizer, although we have not yet implemented one. For optimization, the catalog lookup could return statistics in addition to the view definition. A standard cost-based query optimizer could then determine an appropriate query execution plan. The distributed plan could span nodes holding base views, but also other nodes in the system.

```
Input: A query - SELECT * FROM $C_0$ WHERE $Q$,
where $C_0$ is a capability to view $V$, and $Q$ is a selection expression
Output: Query result
1.  Determine node where $C_0$ should be evaluated (use IP hint)
2.  If $C_0$ can be evaluated locally then
3.      Look up $C_0$ in local catalog and verify validity (Section 3.2)
4.      If $C_0$ is invalid then return ERROR
6.      Look up view definition in local catalog
7.      If $V$ is a base view then
8.          Forward query to query engine and return result
9.      Else: View $V$ is defined on capabilities $C_1, C_2, ..., C_n$
10.     Foreach capability $C_i$ do
11.         Recursively evaluate $C_i$, pushing selections downwards
12.     Process results from subqueries and return result
13. Else
14.     Forward query remotely and return result
```

Figure 6: Recursive Query Evaluation Algorithm.

3.4.2 Implications of the `CATALOG_LOOKUP` right

Allowing others to look up view definitions supports query rewrite and optimization, potentially improving query execution performance (as shown in Section 5.2). There are situations, however, when a user may not want to let others look up a view definition. In our scenario, Bob's "Christmas" view is defined on a capability to his base view. If Bob lets Mom look up the definition of the Christmas view, she will gain access to Bob's base view capability. Bob may thus want to prevent Mom from looking up catalog information to protect his 'Party' photos from her.

4. APPLICATIONS

In this section, we show how HomeViews supports personal data management and sharing applications. We describe two data-sharing applications that we built on top of HomeViews and use our experience to show what features HomeViews provides to applications and what features applications need to implement themselves.

4.1 Two Data-Sharing Applications

We built two applications on top of HomeViews: ViewGallery and ViewFS. ViewGallery is a modified version of the well-known centralized photo-sharing application Gallery v.1 [11]. ViewGallery allows users to organize their photos into distributed dynamic albums and to share them in a P2P fashion. It uses HomeViews views to populate photo albums and capabilities to access and share these albums. Each photo album is associated with one view. Whenever a user opens an album, ViewGallery submits a query to HomeViews using the capability to the corresponding view. HomeViews returns a list of photos that match the query, which the application displays as the album. Since ViewGallery albums are built on location-independent views, they can be composed of local and remote albums. We support the following features of Gallery v.1: gallery and album thumb-level visualization and navigation, album sharing with other people on the Internet, album creation, and album naming. Our modifications disabled the following features: rating of photos (because we currently support only read-only sharing) and downloading an album (which is no longer needed, since the integration of files into local albums is now transparent).

ViewFS is a simple file system layer that we implemented on an early HomeViews prototype. It allows users to create distributed dynamic directories and offers read-only access to the files in these directories. ViewFS' directories are dynamically populated by results from querying HomeViews location-independent views. Users create directories to organize their own files and files shared by other users. ViewFS supports directory creation, deletion, and listing.

4.2 Application Requirements

The goal of the HomeViews middleware is to provide abstractions that reduce the complexity of building distributed personal information management applications, by decreasing the set of features that applications must implement. Table 2 shows an open list of application features, partitioned into those provided by HomeViews and those that must or may be implemented by applications built on HomeViews. We use our experience with ViewGallery and ViewFS to describe how applications can implement each feature.

First, an application must construct its own *application-level abstractions* defined on top of HomeViews' view abstraction. For example, ViewGallery exposes a dynamic album abstraction, and ViewFS offers a dynamic directory abstraction. To implement their abstractions, applications must track and store the association between their abstractions and HomeViews capabilities.

Second, applications must provide a user interface for manipulating their abstractions. ViewGallery uses the unmodified GUI from Gallery 1 to display albums. For album creation, it exposes a Web form that hides HomeViews' SQL-based query language. The form also lets users select backgrounds and other attributes for their new albums. ViewFS is simpler; it exposes a standard Unix file system interface to its users. When a user executes `ls` on a directory, ViewFS lists the files in the view associated with that directory. When a user executes `mkdir`, ViewFS creates a directory from a query on top of one or more capabilities specified in the `mkdir` command.

Third, applications may wish to store application-dependent metadata associated with their abstractions. For example, ViewGallery needs to save styles, backgrounds, and other metadata for its albums. ViewFS, which has fewer options, does not need to do that.

Fourth, applications can support sharing by helping users to pass capabilities to others. As previously noted, we use email to transmit capabilities from one user to another. ViewGallery could easily provide an "email this album" function – similar to the numerous Web sites that provide this function for URLs. Clicking on the function would open a Web form for the user to enter the email of the recipient. The form might also allow the user to specify the rights to enable in the capability that would be mailed. While we have not yet implemented this feature, it would be straightforward to provide.

On the receiving side, applications must present an interface for users to inject capabilities received via email into the application. ViewGallery exposes a form interface for this purpose. ViewFS uses a simple `mkdir` command that creates a dynamic directory atop a remote capability.

Finally, as seen in Section 3.3, HomeViews provides a mechanism for revoking the capabilities that it gives out. However, remembering capabilities that have been given out is an application-level task. To help users choose the right capability to revoke, an application could track the association between a shared capability and the recipients. This would be simplified if the application supported an "email this capability" function.

Feature	**Home-Views**	**Applications**	
		View-Gallery	**View-FS**
View operations (create, drop)	X		
Query execution	X		
Distribution	X		
Capability generation	X		
Capability revocation	X		
App-level abstractions (R)		X	X
(Graphical) User Interfaces (R)		X	X
Metadata for app-level abstractions (O)		X	
Transmit capabilities (O)			
Track capabilities (O)			
Help users annotate files (O)			

Table 2: Application features and where they are implemented. Applications provide additional features, some of them are required (R) and others are optional (O).

Previously, we have assumed that files are either annotated (*e.g.*, Bob's photos all have the location set) or that content-based search is performed. These annotations or attributes are entirely application-dependent and are not interpreted by HomeViews. Some applications may want to help users annotate their files so they can build richer queries.

In summary, applications built on top of HomeViews must focus primarily on high-level abstractions, user interfaces, and application-specific metadata issues. HomeViews handles view creation, protection, and distribution.

4.3 Building ViewGallery

We briefly describe our experience implementing ViewGallery, since it is the more complex of our two applications. Porting Gallery v.1 to HomeViews was a simple process. One developer spent roughly seven work days on the port: one day to review the source code; two days to remove functions related to user account access validation, integrity checks for album modifications outside of Gallery, etc.; and the remaining four days to implement the features mentioned in the previous section. Overall, we modified only 11 out of 787 files, added or modified 488 lines of code, and removed 91. The result of this straightforward port (from Gallery to ViewGallery on top of HomeViews) was to change a centralized application into a distributed, peer-to-peer application supporting album sharing, dynamic views, and integration of local and remote views.

5. EVALUATION

This section uses results from microbenchmarks to characterize distributed query performance in HomeViews. Our goal is to determine: (1) the impact of various system components on overall performance, and (2) whether HomeViews is sufficiently fast to organize and share data in practice.

We prototyped HomeViews on Linux using the open-source Beagle [2] desktop search engine. We implemented view creation, catalog lookups, and both capability-based query evaluation algorithms (recursive evaluation and evaluation based on view rewrite).

We ran all experiments on a collection of five Dell PCs running Fedora Core 5 and Beagle 0.2.6. At the high end

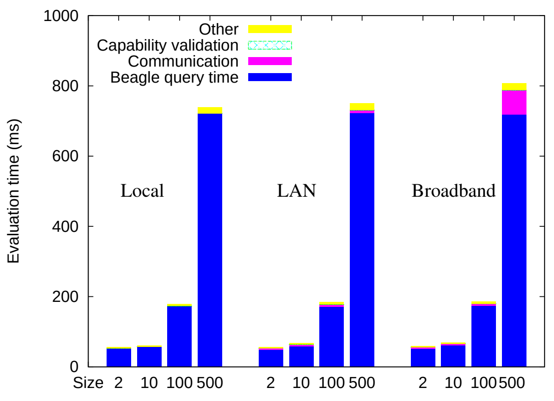

Figure 7: Query execution-time breakdown for simple queries on local and remote views and for different result sizes. The local query processing time (Beagle query time) forms the bulk of total query execution even for remote views.

were 3.2GHz Pentium-4s with 2GB of memory. From our measurements, we believe that the hardware differences in our environment had no significant impact on our results.

For our tests, we synthetically generated a file database of 38,000 music files. We chose music files because their ID3 tag attributes enable rich queries that are supported by Beagle. We controlled the query result size by appropriately setting the ID3 tags of different files. For example, to experiment with a query of size 100, we created 100 files with the album tag "Album100" and a view that selected them.

We examine both *simple* and *complex* queries. Simple queries are one level deep; that is, they involve a single view, itself defined directly over a base view. Complex queries involve views whose definitions include multiple other views composed in various ways.

5.1 Evaluation of Simple Queries

Simple queries allow us to identify and reason about the impact of different components of our system on total query execution times. To evaluate simple queries, we measure the time to execute such queries both from the local machine and remotely. Remote queries use a capability on one machine to access a view defined on another. We experiment with a 100 Mbps local-area network (LAN) and a slower 5 Mbps, 20 ms-delay network (characteristic of home-like broadband connections). Our queries return file names, *i.e.*, we evaluate queries of the form `SELECT filename FROM` *`cap`*. In our experiments, we also vary the query result size.

Figure 7 breaks down query execution time into components for simple queries on local and remote views. Each value is the average over 50 trials. For local and LAN configurations, most of the query execution time is due to Beagle. Capability validation time and other HomeViews overhead (query parsing, view definition lookup in local catalog, and caching of local query results) are negligible, although the HomeViews overhead increases slowly with result size.

As the result size increases, the network transmission time becomes noticeable over slow connections. Table 3 shows query execution times for larger-size query results. Query execution is fast for medium-size results, both for local and remote views (under two seconds for 1000 filenames). Transmission delays increase evaluation time on slow networks

Result size (# filenames)	Time (ms)			
	Beagle	Local eval	LAN	Broadband
1000	1297	1341	1349	1779
3000	3897	4009	4025	5876
5000	6465	6641	6661	11876

Table 3: Local and remote evaluation of simple queries with large-size results. Times are averages of 50 trials. As the result size increases, result transmission over broadband becomes the bottleneck.

Result size	Time (ms)			
	Spotlight	Local eval	LAN	Broadband
1000	332	376	384	814
3000	473	585	601	2452
5000	546	722	742	5957

Table 4: Expected query execution times if Spotlight were used instead of Beagle. Local, LAN, and broadband evaluation times are computed from Table 3 by replacing Beagle query time with Spotlight query time. In the simple-query benchmarks, requests are serial and the query engine time does not overlap with other HomeViews components, which makes this a good approximation of HomeViews based on Spotlight.

when the result size is large. However, techniques such as streaming the results can be employed to reduce the user-perceived response latency.

Since Beagle represents a major component of query execution time, we ran some basic tests to compare it to Spotlight (available on Mac OS X). We used out-of-the-box commands to access each tool: `beagle-query` for Beagle and `mdfind` for Spotlight. We used attribute-based queries in both cases. While Spotlight has similar performance to Beagle for queries with small results (up to 100 files), it scales much better for large result sizes (*e.g.*, for results containing 5,000 filenames `mdfind` is about 12 times faster than `beagle-query`).

Table 4 shows the *expected* local and remote HomeViews query execution time if we replaced Beagle with Spotlight. After the substitution, total query execution times remain below 6 seconds even for 5,000-filename results evaluated over broadband. Even with a fast local query engine such as Spotlight, the other HomeViews components (query parsing, capability validation, etc.) remain below 25% of the total local query execution time.

Thus, both local and remote query execution is fast for small result sizes. For large result sizes, query execution times are dominated by the query engine for LANs or network latency for broadbands. However, with a fast query engine such as Spotlight, even queries with many results over broadband can achieve good performance.

5.2 Evaluation of Complex Queries

In our system, views can be composed and distributed seamlessly. We now analyze the performance of more complex queries. Views can be composed and distributed in two ways: (1) either by applying a selection on top of another (remote) view (in which case the *depth* of the view is said to grow), or (2) by applying union, set difference, or intersection on top of other (remote) views (in which case the *breadth* may also grow). Figure 8 gives an intuition of the two dimensions in which views expand in our system. To create a view of a given depth, we initially define a view on

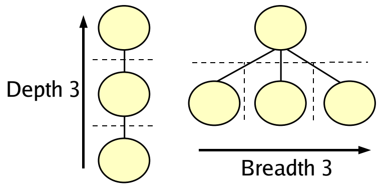

Figure 8: Depth and breadth of views. Dashed lines are machine boundaries; solid lines denote composition (via selection, union, etc.).

Result size (# filenames)	**View depth 1**	**2**	**3**	**4**	**5**
100	175	182	198	208	225
1000	1341	1353	1376	1400	1429
5000	6641	6669	6785	6788	6849

(# filenames)	**View breadth 1**	**2**	**3**	**4**
100	179	218	221	261
1000	1355	1566	1594	1616
5000	6665	7663	7749	7848

Table 5: Recursive evaluation of complex queries on a LAN. Reported times are in ms and are averages over 50 trials. View composition has little effect on recursive evaluation over fast networks.

top of the base view of a node (depth 1). A capability to that view is then given to another node that creates a new view defined on the remote one (the resulting view has depth 2), and so on until we reach the desired depth. Similarly, to create views of increasing breadth, a node creates views defined as unions over increasingly many remote views.

Table 5 shows the results of *recursive* query evaluation over deep or broad views on a LAN. Because transmission costs are small, increases in execution time mainly show HomeViews' overhead. As shown in the table, increasing the depth from one to five nodes leads to an increase in query execution time of 28% on average for the recursive evaluation and a 100-file result. For a 5000-file result the same increase is only 3%. Similarly, a 4-level increase in view breadth results in a 45% execution time increase when each query returns 100 file names. When each query returns 5000 filenames (so 20,000 filenames are gathered at the root), the penalty of the 4-level breadth increase is only 17%.

We see that as query result size increases, the overhead due to the large depth or breadth becomes small compared to the total cost (which is dominated by Beagle). The increase for broad views is larger than that for deep views, because more file names are gathered at the root node in the former case. For small query results, the increase is proportionally higher primarily because all query execution times are already short. Hence, HomeViews scales well with the depth and breadth of views distributed over a fast network.

Figure 9 shows the increase in query evaluation time as the depth of a view increases over a network with limited bandwidth (5Mbps, 20ms delay). The results show both the recursive and query rewrite techniques. The recursive evaluation of large-size queries is now greatly affected by depth, because large network transfers occur from hop to hop back on the recursive path (*e.g.*, when the depth increases from one to five nodes, query execution time increases by 80% for a query returning 3000 filenames).

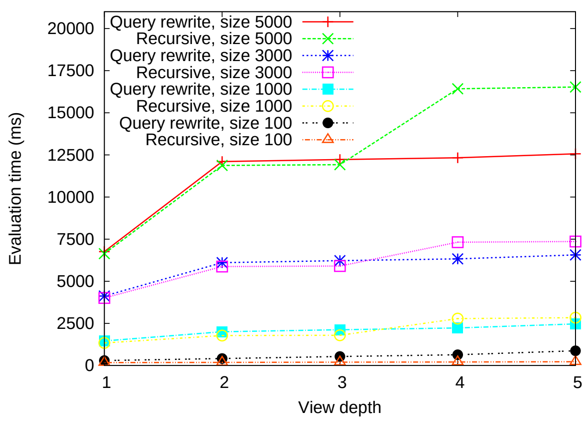

Figure 9: Query rewrite versus recursive query evaluation for deep views distributed over broadband. For small results recursive evaluation has very good performance; for deep views and large results, the query rewrite technique outperforms recursive evaluation.

In contrast, the performance of the query rewrite technique is approximately constant for views deeper than two. Indeed, the bulk of transfers (the results) occur only over one hop (from the base node to the 'root' node). Hence, for queries with large-size results on deep views, rewrite is much more efficient than recursive evaluation. For a 5000-filename query result and a depth of five, the benefit of applying the query rewrite technique is 24%. For small-size results (500 filenames), on the other hand, recursive evaluation is faster than query rewrite even for deep views: results are small and comparable in transmission time to view definitions.

Thus, on slow networks, recursive evaluation works well for small results and small view depths, while query rewrite improves performance for large results and deep views. For the most general case, in which queries are tree-shaped (both deep and wide), query execution time would be dominated by the deepest branch of the tree.

5.3 Scalability

So far, we have shown how our prototype performs when queries come one by one, for different complex query structures. We now show how the prototype scales as the number of incoming queries grows.

When HomeViews performs a distributed query, some nodes execute local queries, while others simply forward requests and results between peers. Because local query execution is much more expensive than forwarding operations (see Section 5.2), we only benchmark the scalability of a HomeViews instance when all incoming query requests involve a local query.

We used a closed-loop benchmark to measure the throughput of the system. After issuing a large number of concurrent requests (100), we generate a new request whenever an old request is completed. Figure 10 shows the throughput for simple queries involving local search, for different result sizes. For small result sets, the system can support up to 30 concurrent requests. For increasing result sizes HomeViews' throughput degrades as fast as Beagle's own throughput (see the Beagle component in Figure 7).

Using more efficient query engines would perhaps improve the system's throughput. Overall, for our targeted medium-

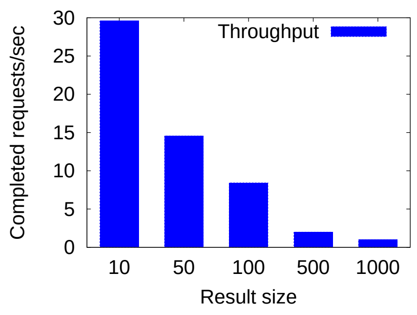

Figure 10: HomeViews throughput for different query result sizes.

scale environment of hundreds of friends who share photos, blogs, videos, or other media over broadband, we do not expect that queries on each node will have high frequency. HomeViews (possibly based on Spotlight) should thus easily support the expected workload, even on popular nodes.

5.4 Discussion

Our microbenchmarks show the parameters that characterize our system's performance and enable us to derive the scalability of the system in real deployments. Our results demonstrate that our prototype is sufficiently fast to be practical in medium-scale environments. For local queries with large-size results, Beagle dominates query execution times. Using a faster local query engine, such as Spotlight, could significantly improve performance and scalability. At the same time, the query engine would still account for the majority of the execution time. For queries executed remotely over slow networks, transmission latency adds significantly to the time. On fast networks, the depth and breadth of views have little influence on recursive query evaluation times. On slow networks, a simple rewrite of views in terms of base views yields good query execution performance even when result sizes are large.

Caching is known to increase a system's performance, availability, and scalability. In HomeViews, applications can cache results from queries according to their own freshness policy to avoid running queries at small time intervals. Also, file contents can be cached; this allows the system to reduce network traffic by transferring only new files or updates to existing files. An in-depth study of the effects of caching and replication on system performance is beyond the scope of our current study.

6. RELATED WORK

In recent years, tools such as WinFS [28], Mac OS X Spotlight [38], and Google Desktop [14] have emerged, enabling users to create database-style views over their data. Personal Information Management systems (*e.g.*, [7, 25]) have begun to explore new techniques for organizing and searching personal information. In particular, the Haystack [25] project enables users to define "view prescriptions" that determine the objects and relationships that an application displays on the screen. Our work builds on the same idea of using views to organize personal data, but our goal is to facilitate the sharing and composition of these views in a P2P environment.

Peer-to-peer systems have become popular for sharing digital information [3, 26]. The main goal of these systems is for *all* participants to share *all* their public data with *all* others. These systems thus focus on powerful and efficient search and retrieval techniques (*e.g.*, [18, 21, 30]). In contrast, HomeViews focuses on *selective* sharing of different data items with different users. HomeViews is also geared toward a medium-scale system rather than the millions of users common in peer-to-peer file-sharing systems.

Operating systems and databases enable access control (and thus selective sharing) by providing mechanisms that associate privileges with users [12, 16, 19, 22, 33].

Significant work focuses on the flexibility, correctness, and efficiency of these mechanisms (*e.g.*, [35, 36]), making them well-suited for many application domains. From the perspective of sharing personal information, however, these techniques suffer from the same administrative burden: someone must create and manage user accounts. HomeViews avoids this overhead by decoupling access rights from user identities. Federated digital identities [8, 20, 31] have been proposed to allow registered users of an administrative domain to access resources from another administrative domain without requiring registration with the later. Federated identities assume a contract or prior coordination between the participating administrative domains. HomeViews has no such requirement.

Another selective sharing technique is to encrypt data with multiple keys and distribute different keys to different users [29]. This approach is suitable only for static data sets that can be encrypted once and published. More dynamic sharing is possible [4] if users run secure operating environments. HomeViews enables dynamic sharing without this restriction.

The capability protection model has been previously applied to operating systems [40, 42], languages [24], and architectures [17, 32]. Our sparse capabilities are related to previous password capability systems [5, 34, 40]. HomeViews integrates concepts and mechanisms from capability systems into database views in a distributed peer-to-peer system.

7. CONCLUSION

This paper described HomeViews, a new peer-to-peer middleware system that simplifies the construction of distributed, personal-information-sharing applications. HomeViews facilitates ad hoc, peer-to-peer sharing of data between unmanaged home computers. Key to HomeViews is the integration of a dynamic view-based query system with capability-based protection in a peer-to-peer environment. With HomeViews, applications can easily create views, compose views, and seamlessly integrate local and remote views. Sharing and protection are accomplished without centralized management, global accounts, user authentication, or coordination of any kind.

We prototyped HomeViews in a Linux environment using the Beagle search engine for keyword queries. Our implementation and design show that capabilities are readily supported by a query language such as SQL, which enables integrated view definition and sharing. We implemented two applications on top of HomeViews, a simple file-sharing application and a port of the Gallery photo-sharing application. Our experience with Gallery in particular shows the ease of supporting protected peer data sharing on top of HomeViews. Finally, our measurements demonstrate the negligible cost of our protection mechanism and the practicality of our approach for medium-scale environments.

8. ACKNOWLEDGMENTS

We would like to thank Phil Bernstein, Tanya Bragin, Alon Halevy, Nodira Khoussainova, Sam Madden, Alexander Moshchuk, Charles Reis, and Dan Suciu for their useful feedback on early versions of this paper.

This work was supported by National Science Foundation Grants NSF-614975, NSF-0627367, NSF-0430477, and NSF-0132817, by an Alfred P. Sloan Foundation Fellowship, by the Wissner-Slivka chair, and by the Torode Family Endowed Career Development Professorship.

9. REFERENCES

[1] M. Anderson, R.D. Pose, and C.S. Wallace. A Password-Capability System. *The Computer Journal*, 29(1):1–8, 1986.

[2] Beagle: Quickly find the stuff you care about. `http://beagle-project.org/Main_Page`, 2006.

[3] BitTorrent. BitTorrent Home Page. `http://bittorrent.com/`, 2006.

[4] L. Bouganim, F. Dang Ngoc, and P. Pucheral. Client-based access control management for XML documents. In *Proc. of the 30th VLDB Conf.*, September 2004.

[5] J.S. Chase, H.M. Levy, M.J. Feeley, and E.D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Trans. on Computer Systems*, 12(4), 1994.

[6] Dc++. `http://dcplusplus.sourceforge.net/`, 2006.

[7] X. Dong and A. Halevy. A platform for personal information management and integration. In *Proc. of the CIDR Conf.*, January 2005.

[8] M. Erdos and S. Cantor. Shibboleth architecture draft v05. `http://shibboleth.internet2.edu/docs/draft-internet2-shibboleth-arch-v0%5.pdf`, 2002.

[9] Flickr. Flickr Home Page. `http://flickr.com/`, 2006.

[10] M. Franklin, A. Halevy, and D. Maier. From databases to dataspaces: a new abstraction for information management. *SIGMOD Record*, 34(4), 2005.

[11] Gallery. Gallery: Your photos on your website. `http://gallery.menalto.com/`, 2002.

[12] H. Garcia-Molina, J.D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2002.

[13] P. Gathani, S. Fashokun, and R. Jean-Baptiste. Microsoft SQL Server version 2000: Full-text search deployment. White Paper. `http://support.microsoft.com/`, May 2002.

[14] Google. Google Desktop: Info when you want it, right on your desktop. `http://desktop.google.com/`, 2006.

[15] A. Halevy, M. Franklin, and D. Maier. Principles of dataspace systems. In *Proc. of the 2006 PODS Conf.*, June 2006.

[16] M.A. Harrison, W.L. Ruzzo, and J.D. Ullman. Protection in operating systems. *Comm. of the ACM*, 19(8), 1976.

[17] M.E. Houdek, F.G. Soltis, and R.L. Hoffman. IBM System/38 support for capability-based addressing. In *Proc. of the 8th Int. Symposium on Computer Architecture*, May 1981.

[18] R. Huebsch, J.M. Hellerstein, N. Lanham, B. Thau Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *Proc. of the 29th VLDB Conf.*, September 2003.

[19] iFolder. Howto: Enabling sharing with Gaim. `http://www.ifolder.com/index.php/HowTo:Enabling_Sharing_with_Gaim`, 2006.

[20] Internet2. Shibboleth. `http://shibboleth.internet2.edu`, 2006.

[21] H.V. Jagadish, B. Chin Ooi, Kian-Lee Tan, Q. Hieu Vu, and R. Zhang. Speeding up search in peer-to-peer networks with a multi-way tree structure. In *Proc. of the 2006 SIGMOD Conf.*, June 2006.

[22] V. Jhaveri. WinFS team blog: Synchronize your WinFS data with Microsoft Rave. `http://blogs.msdn.com/winfs/archive/2005/09/08/462698.aspx`, 2005.

[23] M.B. Jones and R.F. Rashid. Mach and Matchmaker: kernel and language support for object oriented distributed systems. In *Conf. on Object Oriented Prog. Systems, Languages, and Applications*, October 1986.

[24] E. Jul, H.M. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Trans. on Computer Systems*, 6(1), February 1988.

[25] D. Karger, K. Bakshi, D. Huynh, D. Quan, and V. Sinha. Haystack: A customizable general-purpose information management tool for end users of semistructured data. In *Proc. of the CIDR Conf.*, January 2005.

[26] Kazaa. Kazaa Home Page. `http://kazaa.com/`, 2006.

[27] H.M. Levy. *Capability-Based Computer Systems*. Digital Press, 1984.

[28] S. Mehrotra. WinFS team blog: What a week. `http://blogs.msdn.com/winfs/archive/2005/09/01/459421.aspx`, 2001.

[29] G. Miklau and D. Suciu. Controlling access to published data using cryptography. In *Proc. of the 29th VLDB Conf.*, September 2003.

[30] W. Siong Ng, B. Chin Ooi, Kian-Lee Tan, and A. Zhou. PeerDB: A P2P-based system for distributed data sharing. In *Proc. of the 19th ICDE Conf.*, March 2003.

[31] OASIS. Oasis security assertion markup language (SAML). `http://www.oasis-open.org/committees/security`.

[32] E.I. Organick. *A Programmer's View of the Intel 432 System*. McGraw-Hill, 1983.

[33] M. Tamer Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, second edition, 1999.

[34] R. Pose. Password-capabilities: Their evolution from the Password-Capability System into Walnut and beyond. *IEEE Computer Society*, 2001.

[35] S. Rizvi, A. Mendelzon, S. Suharshan, and P. Roy. Extending query rewriting techniques for fine-grained access control. In *Proc. of the 2004 SIGMOD Conf.*, June 2004.

[36] A. Rosenthal and E. Sciore. Administering permissions for distributed data: Factoring and automated inference. In *Proc. of IFIP WG11.3 Conf.*, 2001.

[37] J.S. Shapiro, J.M. Smith, and D.J. Farber. EROS: a fast capability system. In *Symposium on Operating Systems Principles*, pages 170–185, 1999.

[38] Spotlight: Find anything on your Mac instantly. Technology Brief `http://www.apple.com/macosx/features/spotlight/`, 2006.

[39] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. of the ACM SIGCOMM'01 Conference*, August 2001.

[40] A.S. Tanenbaum, S.J. Mullender, and R. van Renesse. Using sparse capabilities in a distributed operating system. In *Proc. of the 6th ICDCS Conf.*, 1986.

[41] USAToday. Usatoday: U.s. asks internet firms to save data. `http://www.usatoday.com/tech/news/internetprivacy/2006-05-31-internet-r%ecords_x.htm`, 2002.

[42] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The kernel of a multiprocessor operating system. *Comm. of the ACM*, 17(6), June 1974.

[43] Yahoo! Yahoo! photos home page. `http://photos.yahoo.com/`, 2006.

[44] YouTube. Youtube: Broadcast yourself. `http://youtube.com/`, 2006.