Capability-Based Access Control for Peer-to-Peer Data Sharing

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 describes SharedViews, a peer-to-peer data management system that simplifies file organization and facilitates file sharing and protection among home users on the Internet. The key innovation of SharedViews is the integration of queries and dynamic views from database systems with a capability-based protection model from the operating systems world. Users organize their files using views and share their views by exchanging capabilities. We show that the resulting system is easy to use and provides clients with flexible protection, but without the account creation and centralized protection management problems inherent in current shared-data systems.

We have prototyped SharedViews on a small network of Linux-based personal computers. We present the architecture of our prototype and use simple measurements to demonstrate the practicality and performance of our approach.

1. Introduction

Today's homes are filled with rich collections of digital data, such as photos, videos, email, Web pages, and other documents. Some key challenges for users are organizing, managing, and sharing this information with family, friends, and others connected through broadband networks. As their personal data repositories have grown in scale and sophistication, users have increasingly adopted two new kinds of tools: peer-to-peer [3, 20] and Web-service oriented [8, 35] file-sharing systems, and desktop search tools, which let users locate files using queries or organize their files using views [31, 11, 22].

This paper describes SharedViews, a system that integrates these two functions and simplifies the organization, management, and protection of distributed data. SharedViews allows users to: (1) create database-style views over their file repositories; (2) selectively grant (and later revoke) access to their views to other remote users; and (3) create and share views that are defined over their local files, local views, and views imported from other users. SharedViews views are *dynamic*: users can share dynamically changing data sets, rather than just static copies of their files. Moreover, they can compose these dynamic data sets easily and independently of the location of the underlying files on the Internet. No existing systems provide the combination of all of these features at the same time in an ad hoc, unmanaged, home environment such as ours.

Sharing systems require protection mechanisms to prevent unauthorized access to private data. Most sharing systems rely on user accounts and access control lists to provide this protection. However, accounts and ACLs are difficult to manage in a distributed or peer environment. In contrast, SharedViews gives users a simple, lightweight, and flexible mechanism for controlling access to their shared views. Our protection mechanism is based on capabilities [6, 21]. Originally developed in the context of object-based operating systems, a capability bundles together a view name with access rights. Users give each other access to their data simply by exchanging capabilities to their views, much like users enable each other to view their private Web pages by exchanging URLs.

We show that a capability-based access control model can be easily integrated into a query language such as SQL, requiring only a small set of changes. The integration enables seamless definitions of new views on top of previously defined local and remote views, and the subsequent sharing of these views without coordinated protection management. Capabilities also enable the rewrite and optimization of the resulting distributed queries, leading to good query execution performance.

Capabilities enable the same access rights as other schemes. Users can grant rights to read, update, insert, or delete data, and even modify view definitions. In this paper, however, we focus on the implications and performance of giving other users read-only access to views. This type of sharing lets us demonstrate the benefits of the SharedViews approach, while leaving the additional consistency issues related to properly handling updates for future work.

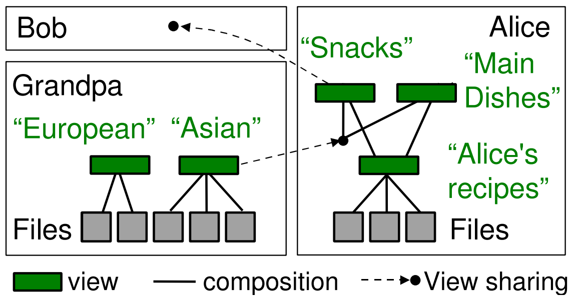

Figure 1. **A simple organizing and sharing scenario.**

1.1. A Motivating Example

Alice's Grandpa, a cooking enthusiast, maintains his own recipe database as a simple set of files. Grandpa likes to organize his recipes by origin: European Food, Asian Food, American Food, etc. To do this, he creates a set of *views*, which are simple lists (directories) of recipe files resulting from keyword queries over the recipe database.

Alice loves Asian food, so Grandpa shares his Asian recipes with her. Alice has her own set of recipes that she would like to merge with Grandpa's Asian recipes, but she prefers to organize them according to dish type: Main Dishes, Snacks, Salads, Desserts, etc. Of course, Alice and Grandpa want the data sharing to be *dynamic*, so that whenever Grandpa adds a new recipe, Alice will see it the next time she looks at her views.

Alice has recently made a new friend, Bob, whom Grandpa does not know. Alice decides to share some of the recipes with Bob. However, she only trusts Bob with her Snacks recipes; the rest are family secrets. With this scenario, Bob has access to all snack recipes, independently of whether Alice or Grandpa created them. Figure 1 illustrates the private and shared views composing this scenario.

1.2. Contributions

Supporting our recipe scenario with current tools is difficult at best. This paper describes how SharedViews easily supports this style of flexible and dynamic data organization, sharing, and protection. Overall, we make the following contributions:

1. We extend the idea of organizing files in a file system using dynamic views with a mechanism for seamlessly sharing and composing these views.
2. We define a lightweight access control and sharing mechanism based on the integration of views with a capability protection model. Our mechanism requires no centralized or coordinated account management.
3. We propose simple extensions to SQL that enable capability-based access control. We show that our extensions facilitate the composition of views defined in different administrative domains, while preserving fine-grained access control in each domain.
4. We present the design and implementation of SharedViews, a peer-to-peer data sharing system based on our lightweight sharing and access control mechanism. SharedViews supports the organization, sharing, and composition of *dynamic* views over the Internet.
5. We demonstrate that SharedViews enables powerful sharing scenarios with good performance during query execution, while being simple to use.

We have implemented SharedViews and evaluate its performance through analysis and experiments. We show that the capability-based protection mechanism imposes a negligible overhead even for distributed queries. The performance of our prototype is good for medium-sized result sets, even over broadband (*e.g.*, the evaluation of a query returning 1000 filenames takes under 2 seconds). For larger-sized queries, optimizations such as query rewrite before execution and streaming of results are necessary to achieve good performance. Overall, however, the results demonstrate that a system such as SharedViews can deliver sufficient performance to be usable in practice.

The rest of the paper is organized as follows. We first describe capability-based access control and present our SQL-based language in Section 2. We discuss the architecture of SharedViews in Section 3, describe a prototype implementation in Section 4, and evaluate its performance in Section 5. Section 6 presents related work and Section 7 concludes our work.

2. A Capability-based Approach

In this section, we provide an overview of capability-based access control and show how it facilitates sharing while simplifying system administration. We then describe SharedViews' query language that integrates capabilities into SQL in a natural way.

2.1. Capabilities and Protection

A capability is a token that designates an object (such as a file or a view) and gives the holder authority to perform actions on that object (*e.g.*, reading a file or querying a view). A capability consists of two parts: the *name* that uniquely identifies the object in the Internet, and the set of *access rights* for that object.

A capability represents a *self-authenticating permission* to access the specified object in the specified ways. The capability is like a ticket: possession of a capability is proof of the holder's rights to access the object. Without a capability for an object, a user cannot "name" it or access it.

To be self-authenticating, capabilities must be *unforgeable*; *i.e.*, it must be impossible for a user to fabricate a capability, or for the receiver of a capability to modify the rights bits or to change the "name" field to gain access to

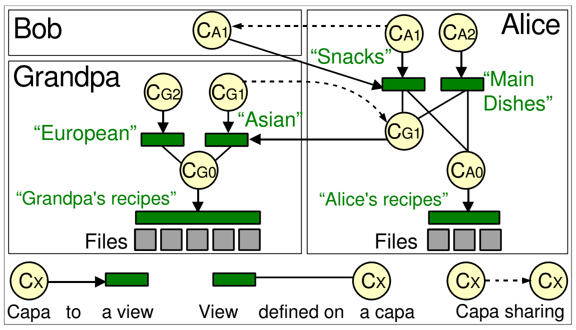

Figure 2. **Solving the scenario with capabilities.**

a different object. Previous systems have guaranteed this property in several ways: *e.g.*, through encryption [33], by storing capabilities in the OS kernel [34], or by using hardware tag bits to identify capabilities in memory [13]. We describe how we accomplish unforgeability in Section 3.2.

Capabilities facilitate sharing because they can be easily passed from user to user as a way to grant access. The sharing requires no user accounts, no user authentication, and no centralized protection structures. Figure 2 illustrates the use of capabilities in our recipe-sharing scenario. When users create views, the get capabilities to these views. They use these capabilities to access their views, create other views, or share views by passing the capabilities to each other.

2.2. Capabilities vs. Access Control Lists

To illustrate the implications of capability protection in our loosely-coupled distributed setting, we contrast capabilities with the traditional alternative, the access control list (ACL). The two schemes are at different ends of a spectrum: capabilities favor ease of sharing and management, while ACLs favor tight access control and access logging.

Let's return to our scenario in which Alice wants to share her Snacks view with Bob. In the most general case, with ACLs, this requires that both Alice and Grandpa (who does not know Bob) first create accounts for Bob on their machines. Once these accounts are created, Alice and Grandpa must grant Bob permission to access their views by adding Bob (now that he has accounts) to the appropriate ACLs. Once all that is done, Alice can pass Bob the name of her view. Finally, Bob can execute his query by authenticating himself directly to Alice's and Grandpa's machines. As an alternative, Alice could serve as mediator for all of Bob's accesses to her Grandpa's recipes, but this is not necessarily easier to setup, and, as shown in Section 5, can degrade query execution performance in many configurations.

In contrast, our capability-based sharing and composition mechanism is extremely lightweight. It requires no account registration or user authentication. Any node can independently generate a capability for an object and pass it to a second party. The second party can use the capability without knowing where it came from, or pass it on to others.

While capabilities greatly simplify data sharing, confinement and tracking are more difficult. For example, in an ACL scheme it is easy to log every access to an object, attributing it to the responsible user. This is difficult with capabilities, because there are no user identities. Therefore, ACLs make sense in commercial environments where access tracking is crucial. However, we believe that capabilities are better suited to our environment of unmanaged home users cooperating in peer-to-peer data sharing.

2.3. Query Language

We adopt a modified version of SQL as the query interface for SharedViews. We show that enabling capability-based access control requires only minor changes to SQL. The resulting language is simple, enables seamless view sharing and composition across different administrative domains, and could be used in other applications.

SharedViews models the file system as a single relation, called `Files`. Each tuple in the relation represents one file. The schema of the relation is the set of all known file attributes (*e.g.*, name, author, date, music genre, picture resolution). File contents are also included in the relation either under the 'text' or 'binary' attributes. The advantage of using a single relation is that files of different types can be returned as part of a single query. If a file does not support an attribute, it has a `NULL` value for that attribute.

On top of this relation, views are defined in terms of predicates on file attributes and contents. Views can also be composed with union, set difference, and intersection operators.

In SharedViews, users must specify a capability for every view that appears in a query. The specified capabilities provide two functions. First, a capability is a handle for a view, *i.e.*, it *names* the view. Second, the capability verifies its holder's authority to access the view in specific ways. Therefore, users do not need to authenticate themselves, since the capabilities are self-authenticating.

Note that multiple capabilities for the same view can exist for several reasons. For example, a user might create multiple capabilities for a view to give different people different access rights. Or, a user could create different capabilities for different recipients so that the capabilities could be selectively revoked at a later time if necessary.

2.3.1. Specifying Capabilities With Queries

Table 1 summarizes the details of our query language. As noted above, SharedViews users present capabilities to execute queries on views, and this is reflected in our query language. Because a capability identifies exactly one view, capabilities can be used to name views in the `FROM` clause. For example, with her capability C_{G1} to Grandpa's Asian recipes, Alice can select only those using ginger and created since June 2006 as follows:

New/modified statement			Return Type	Meaning
SELECT $*$	FROM	*Cap* [WHERE ...]	Set of tuples	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
REVOKE	Cap_1	USING Cap_2	Void	Revoke capability Cap_1

Table 1. SQL modifications, *Cap*, Cap_1, Cap_2 are capabilities, *rights* is a string encoding access rights

```
SELECT * FROM C_G1
WHERE  date > '2006-06-01'
AND    CONTAINS(text,'ginger')
```

where `date` is the file creation date, `text` is the file content, and the query selects all attributes, including the file content. SharedViews supports keyword queries with a simplified form of the `CONTAINS` predicate used by SQL Server [10]. In SharedViews, 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.

With this approach, the semantics of `SELECT` statements remain unaltered. Only the view naming scheme changes.

2.3.2. Creating Views

Users create views with a standard `CREATE VIEW` statement, where capabilities again name any underlying views. More importantly, the `CREATE VIEW` statement *returns* a capability to the newly created view. For example, Alice has a capability C_{A0} to the view containing all her recipes and a capability C_{G1} to Grandpa's Asian recipes view. She can create the Snacks view as follows:

```
CREATE VIEW Snacks AS
SELECT * FROM C_A0 WHERE CONTAINS(text,'snack')
UNION
SELECT * FROM C_G1 WHERE CONTAINS(text,'snack')
=> C_A1
```

where the right arrow denotes the returned capability, C_{A1}, that Alice will use to access her new Snacks view. Similarly, users specify capabilities to views rather than view names when they want to alter or delete a view.

When a new user first accesses SharedViews, she has no capabilities and cannot perform any operations. To bootstrap the system, we add a new `CREATE BASEVIEW` statement that returns a capability to a view providing access to all the files in the file system visible to the user. The underlying file system access control determines this set of files. From this initial capability, which has all rights enabled, the user can execute queries and create additional views. As an example, Grandpa's base view in Figure 2 is labeled "Grandpa's recipes", and his capability to it is C_{G0}.

2.3.3. Capability Restriction and Revocation

To share access to a view, a user can directly pass the capability returned by the `CREATE VIEW` statement. As previously noted, however, users may want to create a more restricted capability to give to their friends. They may also later want to revoke the capabilities they gave out.

To support these operations, we introduce two new statements: `RESTRICT` and `REVOKE`. Given a valid capability `X`, `RESTRICT X RIGHTS rights` creates a new capability that refers to the same view as `X`. The `RIGHTS` clause is an enumeration of all the rights *enabled* on the view; only rights present in `X` can be carried forward to the restricted capability. For example, before Alice shares her Snacks view with Bob, she can produce a new capability, C'_{A1}, from her capability C_{A1} as follows:

```
RESTRICT C_A1 RIGHTS SELECT => C'_A1
```

She will then email C'_{A1} instead of C_{A1} to Bob, giving him only the ability to read recipe files and not, for example, to look up the definition of the view in the catalog. Currently, SharedViews supports the following rights: `SELECT` (read right), `DROP` (right to delete the view), `ALTER` (right to modify the view definition), `REVOKE` (right to revoke capabilities defined for the view), and `CATALOG_LOOKUP` (right to look up the view definition in the catalog). File creation, removal, and updates are currently performed outside of SharedViews, through the file system. Hence, only the owner of a file can modify it[1].

Given two valid capabilities C_{A1} and C'_{A1} to the same underlying view, if C_{A1} has `REVOKE` right enabled, then `REVOKE` C'_{A1} `USING` C_{A1} revokes capability C'_{A1}. Any subsequent use of C'_{A1} will fail. In our scenario, for example, this statement would be used by Alice to revoke Bob's capability to the Snacks view.

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

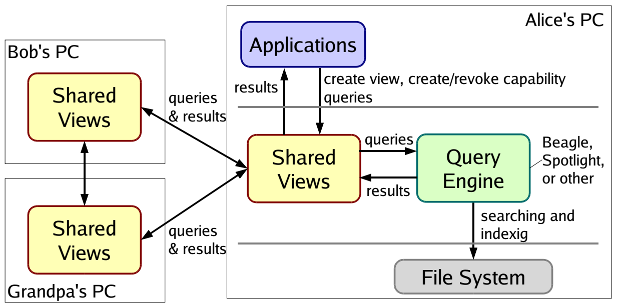

Figure 3. **SharedViews system architecture.**

2.3.4. Catalog Information Lookup

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

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

To simplify catalog lookups, we introduce the shorthand notation `CATALOG OF` to refer to the results of the above query. For example, Alice can look up the definition of Grandpa's Asian recipes view with the statement: `SELECT definition FROM CATALOG OF` C_{G1}.

In summary, the main change we propose to SQL – from which most other changes derive – is to use capabilities to access and name views. Although it was designed with our application in mind, the resulting language is general. No constructs are specific to our environment. The language can also easily be extended: access rights can be broadened to include updates, capabilities could be associated with other types of objects, such as tables, etc. The set of operators could also be extended to include joins.

3. The SharedViews System

We now present the architecture and implementation of SharedViews. After describing its high-level system architecture, we present the detailed implementation of capabilities, the query processing algorithms used by the system, and some details about our prototype sharing system.

3.1. Basic system architecture

Figure 3 shows the SharedViews system architecture. At the lowest level, the database on each node is a traditional file system. SharedViews is a middleware layer that sits between applications and the query engine[2]. Users interact with applications, which present a graphic interface to views and capabilities. In response to user actions, applications issue requests to SharedViews using the query language from the previous section.

[2]We use the Beagle search engine [1] in our implementation, but other engines (*e.g.*, Mac OS Spotlight [31] or WinFS [22]) could be used instead.

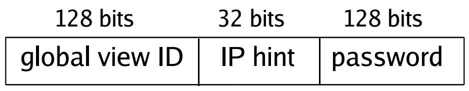

Figure 4. **Capability implementation in SharedViews.**

For example, when the user asks to create a new view, the application sends a `CREATE VIEW` request to SharedViews. SharedViews validates the request, creates a new capability for the view, registers the new view and capability in its internal catalog (described below), and returns the capability to the application.

When the application submits a query on a capability, SharedViews first checks in its catalog whether the corresponding view is local. If the view is local, the capability appears in the catalog, and the request is permitted, SharedViews invokes the query engine to perform the query, and returns the result to the application. If the capability is for a remote view, the local SharedViews instance contacts the SharedViews instance on the remote node and forwards it the query (which includes the capability). The remote SharedViews instance validates the capability, performs the query, and returns the query result to the requesting node. We discuss more sophisticated query execution scenarios in Section 3.3.

3.2. Capability Implementation

Figure 4 shows the structure of our *view* capabilities. Our capabilities are "protected" through the use of sparse random numbers in an astronomically large space. There are three parts to a SharedViews capability, as shown in Figure 4. 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. Second, a 32-bit *IP hint* contains the IP address of a node that, with high likelihood, either contains or can locate the object. Third, a 128-bit random password ensures the authenticity of the capability.

We chose this structure to meet the needs of our environment. Our capabilities are simple to generate and easy to pass from user to user. Managing capabilities requires no special privilege. The protection is probabilistic; however, the probability of guessing a valid capability is vanishingly small. For security, capabilities are best sent over the network using a secure communication protocol (*e.g.*, secure email). Overall, this scheme favors simplicity and ease of use over absolute security, which we believe is the appropriate tradeoff for our intended application domain. In general, we expect that objects will not move in our network and the IP hint will find an object in most cases. If the hint fails, we fall back on a conventional distributed directory [24] or distributed hash scheme [32] for location.

Node-local view table (ViewTable)

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

Node-local capability table (CapTable)

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

Figure 5. Capability and View Catalog Tables.

Figure 5 shows the per-node catalog tables that hold view and capability information. For every 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). Similarly, the node's capability table (*CapTable*) contains one entry for each capability minted to a *locally known view*. The CapTable entry stores the capability's password, the access rights, and the global view ID of the associated view.[3] The combination of global view ID and password forms the primary key of CapTable. Given a capability, SharedViews checks whether an entry for the capability exists in CapTable and whether the access rights match the requested operation. If an entry containing the password cannot be found, then the capability is *invalid*. Revocation of a capability is straightforward and requires removing an entry from the table or zeroing the associated rights field.

The combination of a view capability with the unique identifier of a file (*e.g.*its path) is called a *file* capability and can be thought of as a handle to the file (or tuple, in the more general case). It is used in our particular view-based data organization application to access large attributes such as the content of the files resulting from a view evaluation. More precisely, file capabilities are used to enable queries of the form:

```
SELECT A1[,A2,...] FROM C
WHERE  fileID=X
```

where `A1`,`A2` are attributes of the relation `Files`, `C` is a view capability, and `X` is a file ID.

We introduce the notion of file capability because of our special search-like application. Our application returns query results in the same way that Web/desktop search engines return search results. That is, the result of a view evaluation is a list of capabilities to those files that matched the query. As will be seen in Section 4.1, the contents and other large attributes of files are never selected by a query in our application. Instead, the query selects an attribute in the `Files` relation, `FileCap`, which is filled by the SharedViews query processing engine with the file capability.

[3]The IP hint field is not stored in CapTable, since the IP hint for all capabilities on a node will be that node's IP address.

Upon its invocation, a file capability needs to be checked for validity. First, the comprised view capability is validated, as described above. Next, SharedViews verifies that the file is indeed part of the view. As file capabilities are not a constituent of SharedViews in general, but only offer support for our particular application, they are no further discussed in this Section, but the notion will be referred in Section 4.

3.3. Query Processing

SharedViews can process queries in several ways. The choice depends in part on the `CATALOG_LOOKUP` rights of the capabilities involved in the query. Recall that the `CATALOG_LOOKUP` right gives the holder of a capability the right to look up the definition of the underlying view.

Recursive evaluation: If capabilities do *not* have the `CATALOG_LOOKUP` right, then SharedViews evaluates the query recursively. Recursive evaluation pushes queries from server to server 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 SharedViews nodes do not perform arbitrary computation on behalf of other nodes. SharedViews drops queries from remote nodes if these queries access views that are not locally defined (lines 13-15 in the algorithm).

Query rewrite and optimization: If capabilities include the `CATALOG_LOOKUP` right, SharedViews 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 single query, different capabilities can have different rights. Thus query evaluation typically combines the above 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.

Implications of the `CATALOG_LOOKUP` right: Allowing others to look up view definitions enables query rewrite and optimization, thus potentially improving query execution performance (as shown in Section 5.2.2). There are situations, however, when a user may not want to allow others to look up the definition of a view. In our scenario, if Bob is allowed to look up the definition of the Snacks view in Alice's catalog, he can gain direct access to Alice's recipes and her Grandpa's Asian recipes, because the view definition contains the underlying capabilities to these views. Alice

```
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.    Translate Global View ID field of $C_0$ into server location (e.g., use IP hint)
2.    If $C_0$ can be evaluated locally then
3.        Look up $C_0$ in local catalog and verify validity (see Section 3.2)
4.        If $C_0$ invalid then return ERROR
6.        Look up view definition in local catalog, (use Global ViewID field as key)
7.        If $V$ is a base view then
8.            Forward query to query engine and return results
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.       Perform operators (union/...) on result sets above and return results
13.   Else if the query comes from local user then
14.       Recursively evaluate the query remotely and return results
15.   Else return ERROR
```

Figure 6. **Recursive Query Evaluation Algorithm.**

may thus want to prevent Bob from looking up catalog information to protect her family secrets. Additionally, if Bob gains capabilities to the underlying views, he can recreate the Snacks view locally and Alice loses control over Bob's access to that view.

4. The SharedViews Prototype

To test the practicality and performance of SharedViews, we prototyped the system on Linux using the Beagle [1] desktop search engine. We implemented view creation, view definition lookup, and recursive query evaluation based on capabilities. Managing and sharing of capabilities are done outside of SharedViews, as will be seen from the GUI discussion in the subsequent section.

4.1. The Graphical User Interface

We prototyped our GUI using a Web browser communicating with an underlying Web server. In the prototype, each node runs an Apache Web server that exports a Web interface to SharedViews. From a Web browser, people connect to the local Web server and fill in forms to create or evaluate views. The Web server scripts then connect to the local SharedViews engine and forward SQL statements to it. The results of these statements (if any) are presented to the user in the form of dynamic Web pages.

One can think of many other GUI choices for our system, including a file system interface, one standalone application, etc. Several factors encouraged us to use the above-described approach, among which: the simplicity of creating graphical user interfaces using Web forms, the Web browsers' embedded content display functionality, and today's users' familiarity with Web form interfaces and Web link sharing. Nonetheless, SharedViews is independent of this choice.

The Web forms that users fill in to create/evaluate views imitate a limited subset of our query language (presented in Section 2.3). Namely, we admit one or more single-level `SELECT` statements combined by the operators: `UNION`, `INTERSECT`, `EXCEPT`. We support `SELECT` statements of the form:

```
SELECT Name,FileCap FROM <C>
WHERE  <Q>
```

where `C` is a capability, `Q` is a selection expression, `Name` refers to the file name attribute, and `FileCap` is the file capability fictive attribute. Placed between angled brackets are the text fields that the user needs to fill in.

The selection expression is the SQL variant of the Beagle search syntax [2]. There are only two exceptions from SQL, in which we favored the syntax of Beagle (or of any other search engine) as more appropriate for our application: the possibility to omit the `AND` operator between two terms and to include keywords directly in the SQL statement, avoiding the use of the `CONTAINS` predicate. These defaults were used to simplify the user interface and to make it more uniform with that of Web/desktop search engines, which people are already used to. For example, for Alice to look for all Snacks made from ginger and eggs, her selection expression (`Q`) would be: `ginger egg`, instead of `CONTAINS(text,ginger) AND CONTAINS(text,egg)`.

An important aspect of the `SELECT` statements supported by our GUI is that we only admit the selection of file names and file capabilities. The motivation for this is our particular search-like application, which separates the view evaluation and the retrieval of file content (or other attributes) in two sequential steps that the user needs to take (see Section 3.2).

Our GUI hides capabilities and their structure behind Web links. When a view is created, the dynamic page returned to the user contains the capability to the view in the form of a link that people can bookmark, email to friends, and use to evaluate the view or create new views on top of it. Similarly, when a view is evaluated, the returned dynamic page contains links that hide the file capabilities and allow the user to fetch the content of those files. Each Web link embeds the fields of a capability: the globally unique view ID and password for a view capability, and an additional file ID for a file capability.

Given that these links contain sensitive information, concerns arise regarding their storage and transmission. We use secure connections (HTTPS, SSL, and secure email) as protection from eavesdropping over communication lines and rely on the file system protection to ensure that the storage on the local machine is kept safe and private.

As mentioned above, we rely on existing tools for the management of our capabilities to views: bookmarks are used to save and organize capabilities (or, rather, links) and secure email is used to share them with other people. Such tools have the advantage of being familiar to people, which should render the system easier to use. One possible disadvantage of using bookmarks to store capabilities is that people may forget to bookmark the capabilities they received from the system. We address this issue by automatically saving the capabilities returned by the view creation into special files under the user's home directory. These files are protected by the file system and are themselves indexed by the search engine. Consequently, the user can search for them if she forgets to bookmark her precious link to a view. For example, if Alice cannot find her capability to the Snacks view, she can find the file storing that capability by evaluating the query:

```
SELECT Name FROM C0
WHERE  type=cap snack
```

where C0 is the capability to the base view, type represents the extension of the file, and snack is the keyword[4].

4.2. The SharedViews Engine Implementation

The implementation of the SharedViews engine follows the specifications and algorithms presented throughout Section 3. There are, however, several issues that became visible only at implementation time, and which we discuss in this section.

4.2.1. Recursive Evaluation: Handling Errors

When executing a distributed recursive view evaluation, many things can go wrong: intermediate machines can go down, connections can be interrupted, etc. No matter the situation, SharedViews must keep the following two invariants of view evaluation:

(i) faulty view evaluation must not reveal results that would not be returned under normal (*i.e.*working) conditions, and

(ii) SharedViews must do the best effort to return as many results as possible.

Let us take an example to clarify the above two invariants. Suppose that Alice and Bob shared views A and B respectively with Chuck. Next, Chuck wants to share with Donna all the files from Alice, except the files from Bob. For this purpose, Chuck creates a new view, C, which is the set difference of the two views A and B, and then shares a restricted capability (with CATALOG_LOOKUP right disabled) with Donna:

```
CREATE VIEW C AS
SELECT * FROM C_A
EXCEPT
SELECT * FROM C_B
=> C_C

RESTRICT C_C RIGHTS SELECT
=> C'_C
```

[4]Note the simplified, Beagle-like query syntax.

```
Input: $C$ = $A$ op $B$,
where $A$,$B$,$C$ are three views, and op is an operator
Output: (ResultCode, ResultSet of $C$)
1.   (Code$_A$, R$_A$) = recursive_evaluation ($A$)
2.   (Code$_B$, R$_B$) = recursive_evaluation ($B$)
3.   Switch (op) of
4.       case UNION: return ($Code_A \vee Code_B$, $R_A \cup R_B$)
5.       case INTERSECT:
6.           if ($Code_A \vee Code_B$ is ERROR) then return ($Code_A \vee Code_B$, $\emptyset$)
7.           else return (0, $R_A \cap R_B$)
8.       case SET_DIFFERENCE:
9.           if (Code$_B$ is ERROR) then return ($Code_A \vee Code_B$, $\emptyset$)
10.          else return (Code$_A$, $R_A - R_B$)
```

Figure 7. Algorithm to Treat Errors in Recursive Query Evaluation.

where C_A and C_B are Chuck's capabilities to views A and B respectively, C_C is Chuck's capability to view C, and C'_C is Donna's new capability to view C.

With this example, whenever Donna evaluates C'_C, she must only be allowed to see those files in A that are *not* in B. This invariant (which is an example of (i)) needs to hold even in the case of an error, such as Bob's machine is down. On the other hand, if the operator were a UNION (in short denoted $C = A \cup B$), it would make sense for Donna to see at least the files offered by A, even though B is not available (which is an example when invariant (ii) needs to hold). Still, even in this latter case, suppose that Donna now wants to share with Emma a view E, defined as the set difference of one of Donna's own views, D, and view C (in short, $E = D - C = D - (A \cup B)$). In this case, whenever Bob's machine is down, Emma should not be able to view more files from D than when Bob is up.

To guarantee invariant (i), we propagate errors bottom-up on the recursion path and treat them by returning the empty set whenever a *restrictive* operator (INTERSECT or EXCEPT) occurs. To allow best effort delivery (invariant (ii)), the *non-restrictive* operator UNION is treated differently: we do not treat the error at once, but we still propagate it up on the recursive path. The algorithm in figure 7 shows the details of how error treating is done with recursive evaluation.

4.2.2. The View Materialization Cache

As discussed in Section 3.2, the content of a file is retrieved when the user selects the corresponding Web link from the list of links of a materialized view. Upon selection of one of these links, SharedViews validates the file capability embedded in the link. For this, it needs to validate the view capability and then to check that the file indeed belongs to the result set. The latter check may take a lot of time, because it requires at least a Beagle query, which is very expensive, as will be seen in Section 5.

To speed up file fetching after view evaluation, our prototype caches the results from view evaluations. The re-

Local ViewMatTable

global view ID	file ID	file path
...	...	...

Figure 8. **The View Materialization Table.**

sults are saved in the ViewMatTable, whose format is given in Figure 8. When a file needs to be verified as part of a view, SharedViews checks whether a materialization[5] for that view exists in ViewMatTable, and uses the (global view ID, file ID) pair as index in this table to verify that the file can indeed be accessed via the view capability.

4.2.3. Open Issues

Several issues are still left as future work:

- Caching and replication are known to improve important properties of a distributed system, among which availability and scalability. Except for the local view materialization table, we do not maintain any other caches, either of files or of remote view materialization tables.
- The browser bookmark interface can sustain only a very limited number of views. Beyond that limit, the organization and management of views is likely to become a burden on the user. To this end, mechanisms to create, maintain, and display a hierarchy of views are required.
- A powerful selective capability revocation mechanism is supported by SharedViews. However, to help users find the capability they want to revoke, a mechanism to map between capabilities and the identities of those with whom each capability is shared is needed. We envision such a mechanism as being offered by a separate application, which will keep track of the sharing of capabilities, but the implementation of it remains as future work.

4.3. A Web of Personal Files

The prototype described in this section allows the integration of users' file systems into *a Web of personal files*. Views are created using links to other views and are shared as Web links. Using SharedViews, people can integrate others' dynamic collections of personal files into the local collection, much like people integrate others' Web pages into their own, by adding reference links to them.

To further inspect the power of our system, let us refer to publishing, which has been growing in popularity (YouTube [36] is only one example of service that became famous for this functionality). Our prototype enables such publishing of personal files 'for free'. By posting the Web link to a view onto a public page, a user will have indexing crawlers (*e.g.*from Google) crawl his view and files, making them available to the world. Of course, important concerns regarding the scalability of the system now rise, and our paper does not claim to answer them.

[5]Note that there should exist a notion of freshness of a view materialization, but is not yet defined in our implementation.

5. Evaluation

In this section we analyze the overhead of our capability-based access control scheme and measure the query execution performance of our prototype.

Our experiments were run on a heterogeneous collection of Dell PCs running Fedora Core 5 and Beagle 0.2.6. At the high end 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. 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 selects them.

5.1. Overhead of Capability-Based Access Control

We first analyze the overhead of capability-based protection and compare it to that of ACLs.

Space Overhead: In our system, all protection-related information is stored in the capability table (CapTable), which grows with the number of views created on the node and the number of capabilities created for each view. With ACLs, the overhead depends on the ACL data structures. For a sparse matrix, the overhead grows linearly with the number of views and the number of users with access to each view. Assuming the latter is equal to the number of capabilities per view, the only difference between the two schemes is the extra Passwd field we store with each capability. Even with 5000 views and 10 capabilities per view, the difference translates into only 781 KB, which is small with today's storage systems. With a capability-based scheme, users also need to store their capabilities (either created or received). Since our capabilities are 288 bits, the overhead of even 10,000 capabilities is only a negligible 351 KB. Overall, the space overhead of capabilities is relatively inconsequential in our environment and is comparable to the space required for ACLs.

Time Overhead: At runtime, capability-based protection requires a catalog lookup to verify the capability. Since

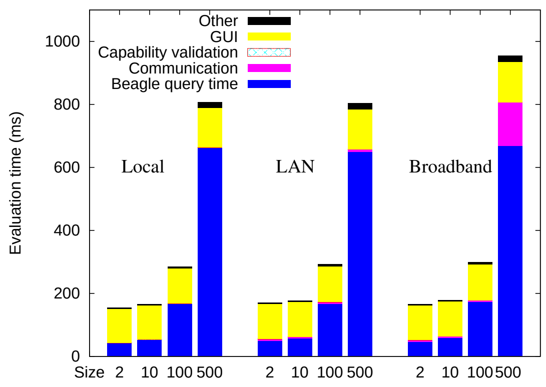

Figure 9. Query execution-time breakdown for simple queries on local and remote views and for different result sizes. The local query processing time (Beagle) forms the bulk of total query execution even for remote views.

the `CapTable` is small, we expect it to remain in memory leading to a negligible overhead even with many capabilities per query. An ACL-based scheme incurs a similar overhead, as it also needs to look up access rights for each object involved in the query. Once again, the overhead of the two techniques is thus comparable.

5.2. Performance of SharedViews

We now evaluate the performance of our prototype's query execution. Our goal is to determine (a) the impact of different system components on overall performance, and (b) whether SharedViews is sufficiently fast to be usable in practice to organize and share data.

Our results examine two kinds of queries that we call *simple* and *complex*. The simple queries are one level only; that is, they involve a single view itself defined directly over a base view. The complex queries involve views whose definitions include multiple other views composed in various ways.

5.2.1. Evaluation of Simple Queries

We first present the performance of evaluating simple queries. We measure the performance of both local and remote evaluations for different query result sizes. The remote access uses 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 in the near future). As previously noted, our queries return file names, *i.e.*, we evaluate queries of the form `SELECT filename FROM` *cap*.

Figure 9 shows the breakdown of query evaluation time into components for small query result sizes. Each value is the average over 50 trials. For the local and LAN configurations, most of the query execution time is due to Beagle and the GUI. The capability validation time and other overhead of SharedViews (view definition lookup and caching of query results) are negligible, although the latter overhead increases slowly with the size of the results. The result transmission time becomes noticeable for slow connections, but for small result sizes it remains fairly short in comparison to the overall query execution time.

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

Table 2. Local and remote evaluation of simple queries with large-size results. Times are averages of 50 trials and exclude the GUI overhead. As the result size increases, the result transmission over broadband becomes the bottleneck.

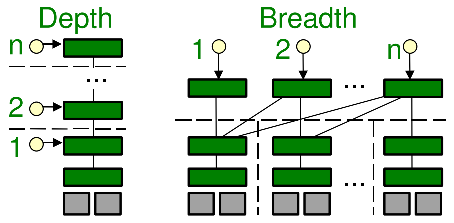

Figure 10. Depth and breadth of views. Dashed lines are machine boundaries; solid lines denote view composition.

Table 2 shows the query execution times for larger-size query results. Query execution is fast for medium-size results, both for local and remote views (under 2 seconds for 1000 filenames). Although transmission delays cause the evaluation times to be high on slow networks when the result size is large, techniques such as result streaming can be employed to reduce the user-perceived latency of the response.

5.2.2. Evaluation of Complex Queries

We now analyze the performance of evaluating complex queries. These are queries on views with more complex definitions. 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 10 depicts deep and broad views, as used in our experimental setup. To create a view of a given depth, a view defined on the base view is initially created on 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 un-

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

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

Table 3. Recursive evaluation of complex queries on a LAN. Reported times are in ms, exclude the GUI, and are averages over 50 trials. View composition affects recursive evaluation little over fast networks.

til 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 3 shows the results of *recursive* query evaluation over deep or broad views on a LAN, where the transmission costs are small and thus the increases in execution time show mainly SharedViews' overhead.

As shown in the table, increasing the depth from 1 node to 5 nodes leads to an increase in query execution time of 28% on average for the recursive evaluation and a 100-file result, while for a 5000-file result the same increase is only 3%. Similarly, a 4-level increase in view breadth results in a 45% increase in evaluation time when each query returns 100 file names. When each query returns 5000 file names (and 20,000 file names are gathered at the root), the penalty of the 4-level increase in breadth is only 17%. Thus, as the query result size increases, the overhead due to the large depth or breadth becomes insignificant in comparison to the total cost (which is dominated by Beagle). The increase for broad views is larger than the increase for deep views, because increasingly many 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 so short.

Hence, SharedViews scales well with the depth and breadth of views distributed across a fast network.

Figure 11 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, due to the large network transfers that occur from hop to hop, back on the recursive path (*e.g.*, when the depth increases from 1 node to 5 nodes, query execution time increases by 80% for a query returning 3000 file names).

In contrast, the performance of the query rewrite technique is approximately constant for views deeper than two. Indeed, the bulk of the transfers (the results) occur only over

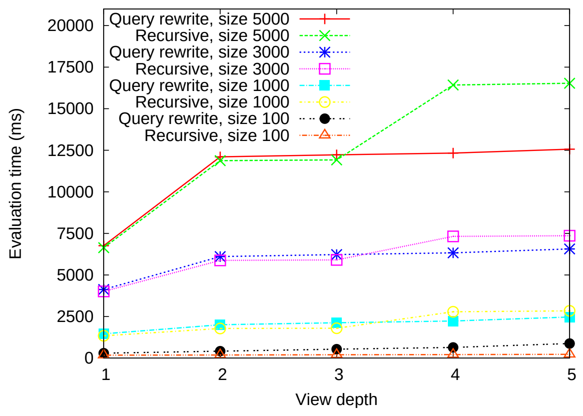

Figure 11. Query rewrite versus recursive query evaluation for deep views distributed over broadband. While for small results recursive evaluation has very good performance, for deep views and large results the query rewrite technique outperforms recursive evaluation.

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 5, 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, as results are small and comparable in transmission time with view definitions.

Hence, on slow networks, recursive evaluation works well for small results and view depths, while query rewrite improves the performance for large results and deep views.

5.3. Summary

Our results show that our prototype is sufficiently fast to be practical in medium-scale environments. The runtime overhead of our capability-based protection scheme is small and comparable to using ACLs. Query execution times are dominated by the query engine, Beagle, and by the network transmission times. 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.

6. Related Work

In recent years, several tools such as WinFS [22], Mac OS X Spotlight [31], and Google Desktop [11] have emerged, enabling users to create database-style views over their data. Personal Information Management systems (*e.g.*, [7, 19]) explore new techniques for organizing and searching personal information. In particular, the

Haystack [19] 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 across different administrative domains.

Peer-to-peer systems have become popular for sharing digital information [3, 20]. 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.*, [14, 16, 25]). In contrast, SharedViews focuses on *selective* sharing of different data items with different users. SharedViews 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 [9, 12, 15, 17, 27]. Significant work focuses on the flexibility, correctness, and efficiency of these mechanisms (*e.g.*, [29, 30]) 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. SharedViews avoids this overhead by decoupling access rights from user identities.

Another selective sharing technique is to encrypt data with multiple keys and distribute different keys to different users [23]. This approach is only suitable for static data sets that can be encrypted once and published. More dynamic sharing is possible [4] if users run secure operating environments. SharedViews enables dynamic sharing without this restriction.

The capability protection model, first introduced by Dennis and Van Horn [6], has been previously applied to operating systems [33, 34], languages [18], and architectures [13, 26]. A survey of such systems can be found in [21]. Our sparse capabilities are related to previous password capability systems [5, 28, 33]. SharedViews integrates the concepts and mechanisms from capability systems into database views in a distributed peer-to-peer system.

7. Conclusion

This paper described SharedViews, a new system that facilitates ad hoc, peer-to-peer sharing of data between unmanaged home computers. SharedViews integrates capability-based protection with a dynamic view-based query system. The result is an Internet data-sharing system that greatly simplifies the organization and protected sharing of personal digital data. With SharedViews, users can easily create views, compose views, and share views using capabilities. They can also mint new capabilities with restricted rights or can revoke previously transferred capabilities. Sharing and protection are accomplished without centralized management, global accounts, user authentication, or coordination of any kind.

We prototyped SharedViews in a Linux environment using the Beagle search engine for keyword queries. Our implementation and design show that capabilities are readily integrated into a query language such as SQL, which enables seamless view definition and sharing. Finally, our measurements demonstrate the negligible cost of our protection mechanism and the practicality of our approach.

References

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

[2] Beagle: Searching data. `http://beagle-project.org/Searching_Data`, 2006.

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

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

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

[6] J. Dennis and E. Van Horn. Programming semantics for multiprogrammed computations. *Comm. of the ACM*, 9(3), March 1966.

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

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

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

[10] 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.

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

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

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

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

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

[16] H. V. Jagadish, B. C. Ooi, K.-L. Tan, Q. H. 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.

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

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

[19] 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.*, Jan. 2005.

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

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

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

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

[24] P. Mockapetris and K. J. Dunlap. Development of the domain name system. In *Proc. of the ACM SIGCOMM'88 Symp.*, 1988.

[25] W. S. Ng, B. C. Ooi, K.-L. Tan, and A. Zhou. PeerDB: A P2P-based system for distributed data sharing. In *Proc. of the 19th ICDE Conf.*, Mar. 2003.

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

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

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

[29] 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.

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

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

[32] 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*, Aug. 2001.

[33] 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.

[34] 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.

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

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