The Organization and Sharing of Web-Service Objects with Menagerie

Roxana Geambasu, Cherie Cheung, Alexander Moshchuk, Steven D. Gribble, and Henry M. Levy
University of Washington, Seattle, WA
{roxana, cherie, anm, gribble, levy}@cs.washington.edu

March 19, 2007

Abstract

The radical shift from the PC desktop to Web-based services is scattering personal data across a myriad of Web sites, such as Google, Flickr, YouTube, MySpace, and Amazon S3. This dispersal poses significant new challenges for users, making it more difficult for them to: (1) organize, search, and archive their data, much of which is now hosted by Web sites; (2) create heterogeneous (multi-Web-service) object collections and share them in a protected way; and (3) manipulate Web objects with standard applications or build new tools or scripts that operate on those objects.

This paper presents Menagerie, a software framework that addresses these challenges. Menagerie creates an integrated file and object system from heterogeneous, personal Web-service objects dispersed across the Internet. Our Menagerie architecture has two key parts. The Menagerie Service Interface (MSI) defines a common Web-service API for object naming, protection, and access. The Menagerie File System (MFS) lets desktop applications and Web services manipulate remote Web objects as if they were local files. Our experience shows that Menagerie greatly simplifies the construction of new applications that support collections of heterogeneous Web objects and fine-grained protected sharing of those objects. We describe the Menagerie architecture and implementation, present several novel applications we developed on Menagerie, and provide measurements that show the practicality of our approach.

1 Introduction

Despite enormous advances in distributed and Web-based computing, many users still rely on their desktop operating systems for data storage and application processing. Files remain the basic paradigm for storing programs and data. Users organize files into folders; they search files, execute them, manipulate them with applications or viewers, compose them in documents, email them to friends, and so on. The desktop OS provides mechanisms for naming, protecting, and sharing files, as well as high-level services such as backup and restore.

This desktop-centric view is rapidly yielding to a new generation of Web services that is profoundly affecting the nature of data and applications. Web-based data storage and sharing services now house terabytes of personal data. For example, users share photos through Flickr [44] or Yahoo! Photos [46]; they publish videos on YouTube [47]; they keep diaries on Blogger [13]; and they store files on services such as Amazon S3 [3]. Similarly, Web-based "software-as-a-service" applications entice users away from the desktop to read email through Gmail or Hotmail; store and share spreadsheets using ThinkFree [34] or Google Docs [14]; and edit videos and photos using Remix [1], Jumpcut [45], and hosted Photoshop. In the near future, the PC desktop will likely be inconsequential for many of our data and application needs.

This brave new world brings many advantages, including simplified software management for the user and the application provider, ease of publishing, and ubiquitous access. On the other hand, it poses significant new challenges. For example:

1. *Data organization and management.* The shift to Web services scatters users' data across the Internet where it is housed by a myriad of Web services. How will the user organize and manage her remotely stored data? Consider Ann, a professor of Biology preparing for her Biology 101 class. Typically, she creates a new directory for each class that contains grade spreadsheets, student photos, lecture slides, class handouts, and student emails. Ann now prefers Flickr for photos, Google Docs for spreadsheets, ThinkFree for handouts, Powerpoint for slides, and Hotmail for class communications. How would Ann create a unified view of these data and applications? Once the class is over, how would she archive all her class data *as a unit* to ensure its safety for future use?

2. *Protected data sharing.* Although publishing is simplified in the Web service environment, protected sharing becomes more difficult. For example, Ann wishes to share her class directory, which includes confidential grade information, with Bob, a professor who will be taking over her class for a week. Does Ann need to share each object explicitly with Bob? Must Bob create accounts on Yahoo, Google, Microsoft, ThinkFree, etc., to access Ann's data? Even in a world of

single-sign-on accounts, how would Ann describe this collection of distributed data to share it with Bob?

3. *Data manipulation and processing.* Web services restrict the operations on their objects; e.g., they export a limited API (if any) and expose only a small set of user commands through the browser. In contrast, the power of a system such as Unix derives, in part, from its simple data-processing commands (`cat`, `grep`, etc.) that can be composed together or extended to manipulate data in new ways. In the Web services world, how does the user create simple programs or scripts to process data housed by the service? Must data be manually downloaded, manipulated, and then uploaded? Or will users cede to the service all data processing, surrendering the ability to easily add new data-processing functions? How do users search for personal data across multiple Web services? Are users locked in by services? For example, suppose Ann wants to switch from Hotmail to Gmail – can she easily retain her contacts, or better yet, all her messages and attachments, or must she abandon her Hotmail data?

We designed and implemented *Menagerie* to address these challenges. Menagerie is a system that integrates heterogeneous Web-service objects through uniform naming, protection and access. Menagerie enables new applications that easily organize, manage, manipulate, protect, and share user personal Web objects. For example, Figure 1 shows how Ann can use a Menagerie-based application on her PC to create a folder that "contains" photos from her Flickr account and Google spreadsheets and documents from her Google account. She can then share this folder and its objects with Bob. Bob uses Menagerie on his PC to access Ann's objects or manipulate them with Unix programs *without* needing to know their locations on Flickr and Google.

The key Menagerie components include: (1) the Menagerie Service Interface (MSI), an API for inter-Web-service communication, and (2) the Menagerie File System (MFS), which uses the MSI to provide a simple file-object view of a user's Web objects. MFS allows users to manipulate Web objects using unmodified existing programs. For example, using Menagerie, a Unix user can archive all of her Flickr photos with a single-line Unix `tar` command. Moreover, she can `untar` that archive into a *different* photo system, such as Picasa, thereby moving seamlessly from one service to another.

We implemented Menagerie on Linux and integrated it with a number of Web services, including Gmail, Google Docs, Flickr, YouTube, and Yahoo!Mail. We then used it to construct a set of new applications, including the Menagerie Desktop Service, which provides a unified Internet desktop to help users organize, manipulate, and share their personal Web objects, and the Menagerie Group Sharing Service, which allows groups of users to share Web objects with each other.

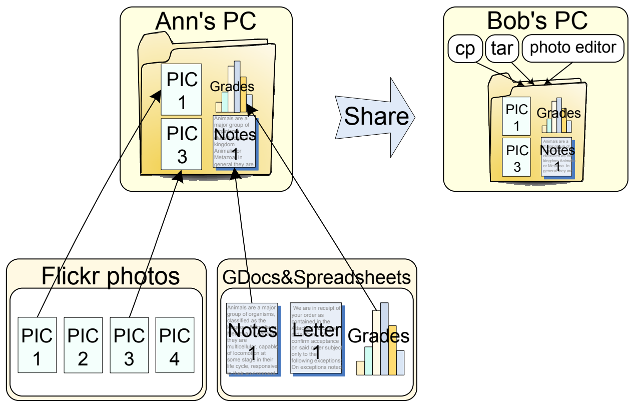


Figure 1: **Example Menagerie Scenario.** Using an Object Organization Web application built on Menagerie, Ann created a new folder that links to some of her objects from Flickr and Google Docs. She can share this folder with Bob, who uses Menagerie to access Ann's folder on his PC and manipulate its objects with Unix programs.

Our experience demonstrates that a set of straightforward protocols and components can help users combine the ease of use, publishing, and ubiquitous access advantages of Web services with the organizational and data processing advantages of the desktop. We further show that the performance of Menagerie is highly competitive with existing Web services and distributed file systems.

In the remainder of this paper, we first present Menagerie's high-level architecture. Section 3 then explains the implementation of Menagerie, focusing on the Menagerie Service Interface and the Menagerie File System. Section 4 describes the applications we built on top of Menagerie. We evaluate the performance of our system in Section 5. Section 6 describes related work, and we summarize and conclude in Section 7.

2 Architecture

Three basic goals drive the Menagerie architecture:

1. *Object location transparency.* Users should be able to create collections of heterogeneous objects and access them without needing to know which service is responsible for each object.

2. *Fine-grained protection and sharing of objects and collections.* Users should be able to share individual objects or collections of objects that are hosted on one or more Web services.

3. *Generic data access and backward compatibility.* Existing programs should be able to manipulate objects hosted by Web services. Similarly, it should be easy to write new programs or scripts that manipulate these objects.

Meeting these goals in the current Web services environment can be daunting. In the current desktop environment,

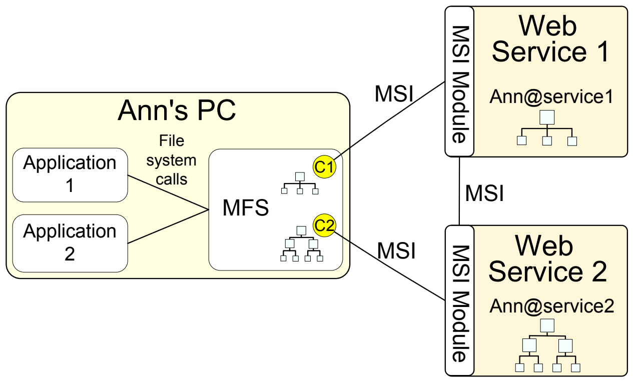


Figure 2: **Menagerie Architecture.** The right side shows two Web services that support the Menagerie Service Interface (MSI). The MSI Module implements MSI operations and exports an object name hierarchy for each user's objects. The left side of the figure shows Ann's PC, which is running the Menagerie File System. Through MSI operations, MFS can mount Ann's object name hierarchies on her PC using capabilities (C1 and C2) for her hierarchies. She can then run applications that access the remote objects using Unix file system calls.

the OS creates a set of simplifying standards for programs and data. In contrast, each service in the Web environment has its own naming, protection, representation, operations interface, and other conventions, which complicates integration and inter-operation. In the long term, our goal is to create a common object architecture on top of existing services. Yet our generic data access goal for programs leads us to an approach that treats objects as files, since files are the common data storage and processing vehicles for existing applications.

Our solution therefore uses both object and file paradigms, which we extend with Web-related features to support new Web-oriented applications. As a result, our architecture bridges the gap between Web services and traditional operating systems. In the short term, Menagerie offers a framework for experimenting with Web service integration. By building new applications on top of Menagerie, we gain insight into the properties needed to better integrate Web services and operating systems in the future.

2.1 Architectural Overview

Figure 2 shows the Menagerie architecture. Each of the two Web services on the right supports the *Menagerie Service Interface* (MSI), a communication protocol that provides object naming, protection, and access operations. Through MSI, the services *export* an object name hierarchy for each of their users. The figure shows Ann's object hierarchies on Web Service 1 (consisting of three objects arranged as a flat list) and Web Service 2 (organized in a two-level structure).

On the left of the figure is Ann's PC, which is running the *Menagerie File System* (MFS). Using capabilities C1 and C2 that Ann supplied for her Web Service name hierarchies, MFS imported and *mounted* those hierarchies on her PC. As a result, Ann's Web objects can be *named* through her local file system name space. This is indicated by the trees in MFS that mirror Ann's Web service trees. Once her Web service name hierarchies are mounted, Ann can run standard applications that operate *transparently* on her remote Web objects.

Before describing MSI and MFS is more detail, we look at how Menagerie handles naming and protection.

2.2 Object Naming

Object naming is key to any distributed architecture, and Menagerie's naming system must support several needs. First, it must provide unique identifiers for objects across heterogeneous Web services. Second, it must provide user-readable names that are meaningful to users and correspond to the way users name objects inside of a Web service.

Each object in Menagerie is identified with a *service-local ObjectID*. The fact that ObjectIDs are unique within a service (but not globally unique) suits the current object addressing model of most Web services, i.e., it gives services the liberty to create and name new objects independently. Menagerie can then create globally unique object names by combining the service-local ObjectIDs with services' DNS names.

Users associate user-readable (text) names with their Web objects, such as Google spreadsheets. As described above, each Menagerie Web service exports an *object name hierarchy* that contains the user-readable names for all objects for each of its users. The hierarchical structure imitates the logical structure that the service exposes to the user. For example, Flick offers users a two-level structure consisting of sets (albums) and photos within each set; therefore, Flickr could choose to export a two-level name hierarchy. Google Docs has only a flat structure and could export a one-level hierarchy that contains all of a user's document and spreadsheet names. Individual objects in a service are named relative to the exported hierarchies, e.g., Ann@Flickr/Disneyland/Mickey-photo or Ann@GDoc/class-grades.

2.3 Protection

As we have seen, Menagerie was designed to support fine-grained sharing of objects and object collections. Such sharing is difficult for Web services because each service enforces its own authentication and control system. While we desire a global, distributed protection system that simplifies object sharing and provides data transparency, we cannot require Web services to totally cede control of the objects they manage.

For this reason, Menagerie supports a *hybrid capability-based protection system.* A Menagerie *capability* is a token that contains the globally unique name for an object and a set of object-specific access rights. Possession of a capability gives the holder the right to access the object in the specified ways. Capabilities support sharing because they are easy to pass from user to user; Menagerie's capabilities are passed in the form of URLs that can be mailed, bookmarked, etc. They support protection because they are difficult or impossible to forge. However, a Menagerie capability is subject to control by the Web services whose object it names.

Each Menagerie service must mint its own capabilities in a manner consistent with the Menagerie architecture and MSI, which supports capability creation and protection operations. A service can divide its object rights into two types: *open-access rights* and *closed-access rights*. An open-access right gives the user of the capability free access to the specified operation; e.g., if the right allows the holder to read the object, then the service will return the object's contents when presented with a capability with that bit set. A closed-access right, however, requires additional authentication; when a capability with a closed-access right is presented, the user must authenticate himself before the service will perform the requested operation. In most cases, this will require an account for the service. However, note that having an account is not sufficient – the user must still possess a capability with an appropriate rights bit that they received from the object's owner.

This hybrid system lets services provide simple controlled access to some (or all) object functions while requiring more restricted access to others if they consider it important to do so. Today's Web services employ a wide range of schemes in the spectrum of choices. For example, Flickr provides a mechanism by which a user can authenticate himself, request a "token" for one of his objects, specify a set of rights enabled by that token, and pass it to an application (or other user) as a way to grant access. This is essentially a capability scheme. Myspace, on the other hand, lets users share their private Web objects only with their "friends," who are registered users of Myspace. Google Calendar offers users "secret URLs" to their calendars that they can give to friends or embed in blogs or Web pages. These URLs are a type of capability that can be used to view, but not modify, the user's calendar. To share a calendar with update rights, the user must provide the receiver's username to the service (effectively adding it to an ACL). Menagerie's protection system easily supports all of these options.

Our protection scheme meets our fine-grained sharing goal. Limited sharing is very simple: the user creates a capability and passes it to someone, who does not need to have an account to perform a limited set of actions on the shared object. To access the object fully, however, the receiver has to decide whether it is worth the effort of creating a new account if he does not already have one. Protected group sharing is much easier as well. If the user wants to share a resource with a group of people, he need not specify all of the usernames in his group to the service. Instead, he simply creates a capability and gives it to his friends (e.g., via email, via a blog, or via a group sharing site where the group already exists).

2.4 The Menagerie File System

The Menagerie File System (MFS) is a Menagerie component that lets users *mount the name hierarchies* exported by Web services onto their machines. Mounting a name hierarchy integrates that object name space into the local file system name space, letting users name their Web objects as if they were files. This gives users a unified view of local objects and Web objects and makes it easier to create new organizations based on those objects.

Once a user mounts a Web service name hierarchy, he can then navigate through his objects on that Web service using existing applications that manipulate files and directories. Users can thus archive their data, copy files to and from a hierarchy, search through all their Internet data, etc. Neither the application nor the user need know where the data is stored or how it is fetched. Thus, MFS makes existing desktop applications useful in the Web object environment. For example, in Figure 2, Ann can easily make a copy of one of her Web objects using the Unix `cp` command without knowing the source of that object.

By exporting a file system interface, MFS also facilitates the building of new applications that perform useful functions on Web service objects. Using MFS as a building block, we constructed a variety of Web and desktop applications for Internet data organization and sharing with minimal effort. We describe these applications in Section 4.

It is worth contrasting our approach with NFS [27]. Like NFS, MFS lets users graft a remote name space onto a local file name space. However, mounting an object name space is not the same as mounting an NFS tree. First, NFS lacks flexibility for protection and sharing; its unit of sharing is essentially the NFS mount, so sharing an object requires that the recipient mount the *entire* NFS tree in which the object resides. Second, the NFS protocol is inflexible and inappropriate for Web services that use URLs, HTML tags, etc. That is, Menagerie objects are not really files, and ensuring the proper behavior of an object in response to some user actions (such as double-clicking on the object) requires additional metadata structures that are not supported by NFS. Finally, supporting the high-level MSI interface is much easier than implementing an NFS server on existing Web services that often maintain data in their own custom storage infrastructures.

2.5 The Menagerie Service Interface

The Menagerie Service Interface (MSI) is an API that Web services must implement in order to support Menagerie. MSI's functions are grouped into the four categories shown in Table 3 and described below: hierarchy traversal functions, file system functions, Web-related functions, and protection-related functions.

Hierarchy traversal functions. As we have seen, every Web service exports a name hierarchy for each of its

Hierarchy traversal functions

getattr(capa, object_ID) returns *FS-like attributes*
list(capa, object_ID) returns *list of object names and IDs*

File system functions

read(capa, object_ID) returns *byte[]*
write(capa, object_ID, name, content)
mkdir(capa, parent_ID, name)
delete(capa, object_ID)
move(capa, object1_ID, object2_ID)

Web-related functions

embed_tag(capa, object_ID) returns *string*
get_URL(capa, object_ID) returns *string*
search(capa, parent_ID, keywords) returns *list names, IDs*

Protection-related functions

create_capa(capa, root_node_ID, rights)
returns *capa, closed_rights*
revoke_capa(object_capa, revoke_capa)

Figure 3: **The MSI interface**. This table shows the parameters and return types of each function. MSI services must support the hierarchy and protection-related functions, and may support the others. Depending on the combination of functions that services implement, some applications may be enabled and others disabled.

users. We consider each node in the hierarchy to be either an intermediate node or a leaf object. *Intermediate nodes* are named sets or directories, while *leaf nodes* are individual objects. For example, in our implementation, Flickr sets (albums) are intermediate nodes and photos are leaf objects; Gmail threads are intermediate nodes, while Gmail messages are leaf objects.

A Menagerie capability can name any subtree of a hierarchy, including the entire hierarchy and leaf nodes. This allows users to share both individual objects and object collections. MSI functions therefore require at least two input parameters to refer to an individual object: (1) a capability that names a hierarchy (tree or subtree), and (2) the objectID of an individual object (set or leaf) within that hierarchy.

Two MSI functions, `list` and `getattr`, support name hierarchy functions. Given the ObjectID of an intermediate node in a hierarchy, the `list` function returns the names of all the objects below that node, as well as their unique IDs. `Getattr` returns the file system-like attributes of an object, given its unique ID. Attributes include the object type (intermediate or leaf object), time of last modification, etc. Applications can therefore traverse an object hierarchy using these MSI functions.

MSI constrains neither the structure of an exported hierarchy nor the granularity of its objects; these are determined solely by the Web service. For example, Gmail could export a four-level hierarchy: folders/labels, threads, messages, and email contents and attachments. Alternatively, Gmail could expose a deeper hierarchy by decomposing message contents into separate objects, one for each of its attributes, such as 'to', 'from', 'subject', etc., resulting in a possibly more flexible, yet more complicated, hierarchy.

File system functions. MSI's file-oriented operations allow new and existing filesystem-based applications (such as tar, file explorers, image editors, etc.) to manipulate hosted objects. Services support these applications by implementing a small set of file methods, such as `read`, `write`, `mkdir`, `delete`, and `move`.

Menagerie does not attempt to implement low-level file operations through this MSI interface. Those operations are left to the services. Instead, we have chosen a very coarse-grained access granularity in the style of early distributed file systems such as CFS [29]. The `read` operation returns the entire contents of a leaf object, i.e., it downloads the object. The `write` creates a *new* object (or replaces an old one) by uploading the entire object, while `mkdir` creates a new intermediate node in the hierarchy. This granularity is high level enough to be convenient both for applications to use and for services to implement.

Web-related functions. These MSI operations exist to preserve the Web-oriented nature of the objects managed by Web services. The `embed_tag` and `get_URL` functions serve two purposes. First, they support the building of new general-purpose Web applications for Internet data management. Like desktop file-organization tools, these applications must be general enough to handle any type of data. Using the `embed_tag` and `get_URL` functions, applications can embed Internet objects into their Web pages in an abstract manner, preserving the "Webby" aspect of each object type. For example, `embed_tag` returns an `<img>` tag for a Flickr photo and an `<object>` tag for a YouTube video. Second, the two functions allow the source Web service to control the appearance of its objects on other Web services and to limit the amount of information provided to unauthenticated users. For example, YouTube can watermark its videos to show the company's logo (as it already does); Gmail could release a short summary of an email for unauthenticated users by returning the HTML code block that displays that summary.

MSI supports an explicit Web-related `search` function. This is both a feature and a performance optimization. Search has become the most common way for users to find objects across large data sets. As a consequence, most services already offer a search function; e.g., users search through their emails on Gmail and through their videos on YouTube. By including the `search` function in MSI, we enable an integrated search through the user's Internet data without requiring it to be downloaded and indexed locally.

Protection-related functions. Since MSI supports a capability protection model, its API provides functions for creating and revoking capabilities to nodes in an exported name hierarchy. The `create_capa` function mints a new capability to an object, given a valid capability for the hierarchy in which the object exists. Revoking a capability requires two capabilities, the `object_capa` to be

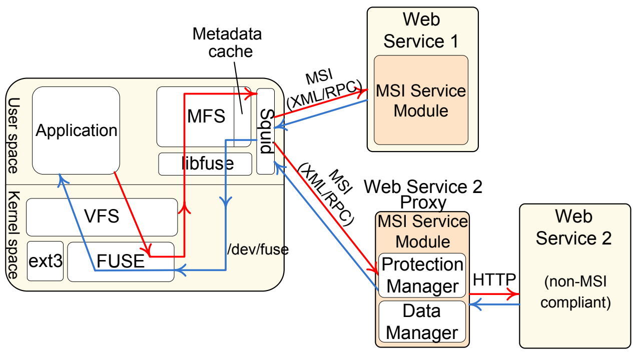


Figure 4: **Menagerie Implementation.** MFS is implemented as a userspace file system. File system calls from applications are redirected through the FUSE kernel module into the MFS user program. Native MSI services implement MSI functions directly through XML-RPC, while proxies provide MSI protection and naming functions on behalf of existing HTTP-based Web services, such as YouTube.

revoked and a `revoke_capa` for the same object with the `REVOCATION` right enabled that permits the revocation. This prevents users from revoking arbitrary capabilities they are given by others.

3 Implementation

We implemented a Menagerie prototype and used it to integrate Web objects stored in five popular Web services: Gmail, Yahoo! Mail, Flickr, YouTube, and Google Spreadsheets. We also implemented two native MSI Web services that we will describe in Section 4.

Figure 4 shows the structure of our Linux-based prototype. On the left side is a node (PC or Web Server) running an application that accesses Web objects through the Menagerie File System. We implemented MFS as a user-space module built on the FUSE (File System in Userspace [32]) infrastructure. FUSE consists of a kernel-level component and a user-level library (libfuse). MFS is linked with the library, which in turn communicates with the kernel-level FUSE component to implement application-generated file operations. To boost performance we use two caches. MFS has an internal metadata cache for rapid retrieval of short-lived file system metadata and uses the Squid [7] cache proxy to store objects returned from MSI `read` calls.

The right side of Figure 4 shows two Web services and their MSI Service Modules. Service 1, a native MSI service, supports MSI functions directly through an internal MSI Service Module that speaks XML-RPC. Since we cannot modify existing HTTP-based Web services (such as Service 2) to add MSI support, we built external *proxies* for each such service. For example, in Menagerie, all MSI operations on YouTube objects are directed to a YouTube MSI proxy in the network. The proxy contains two parts, as shown in Figure 4. The protection manager implements Menagerie's protection model, mapping its capabilities onto the Web service's protection mechanism. The data

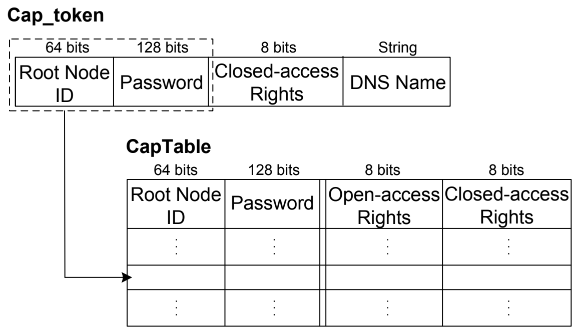


Figure 5: **Protection structures.** CapTable is a hashtable that maintains all the capabilities given out by a service. Our capabilities are protected by a 128-bit random password. The capability's root node ID and password are used to validate that capability in CapTable on each access.

manager exports user hierarchies from the service's object structures and implements all other non-protection-related MSI functions. The proxy communicates with other MSI services using XML-RPC and communicates with its proxied service via HTTP.

We now describe in detail the implementation of the protection mechanism, MSI service modules, and MFS.

3.1 Protection

Menagerie uses *password capabilities* [8, 33] for protection; they protect against forgery by associating with each capability a "password" chosen from an astronomically large number space. A Menagerie capability points to the root-node of a subtree in an exported user's name hierarchy, providing access to all objects in that subtree. A capability contains four parts (Figure 5): the DNS name for the service, a 64-bit root-node ID, a 128-bit random password, and the closed-access rights. Protection is probabilistic; guessing a capability from scratch requires guessing a 192-bit number. The closed-access rights are included so that an application holding a capability can tell which operations require authentication.

Figure 5 also shows the data structure (CapTable) used in our protection managers. The CapTable is a hash table that registers all capabilities ever issued by a specific service. CapTable stores each capability's root node ID, 128-bit password, and open-access and closed-access rights. The rights fields are 8-bit masks, containing one bit for each of the eight possible rights. A rights bit can be set in at most one of the two rights fields and ORing the two rights fields yields the complete set of rights enabled by the capability.

Every MSI method called on an object passes two parameters: a capability token for the subtree in which the object exists and the objectID of the object in that subtree. If the protection manager does not find a corresponding (`root node ID, password`) pair in its CapTable, the capability is invalid. If the operation is permitted and requires no further authentication, the protection manager forwards the request to the data manager for processing. If the operation requires authentication, the protection manager performs the authentication using the username and user password transmitted with the call. To handle existing services that do not support MSI, we configured each protection manager statically with a small user account database to simulate a service-local manager with access to the service's account database.

For the user's convenience, we encode and pass capability tokens as *URLs*. When a user requests a capability from a service, the service returns a URL that embeds the capability. For example, a capability token obtained from Gmail might be encoded as `http://gmail.com/MSI?cap_rootID=...&cap_passwd=...&cap_closed_rght=...`, where the dots represent text-encoded values of the corresponding fields. In this way, capability sharing is similar to URL sharing in the Web. For example, for Ann to share with Bob a folder exported from her desktop service, she would request a capability for that folder from the Web service and copy and paste the URL into an email to Bob. For security, the capability should be sent over secure email.

Revocation in our scheme simply requires zeroing the rights fields or removing the capability entry from the CapTable.[1]

3.2 Data Managers

The data manager for a Web service exports its MSI user hierarchies and implements file system and Web-related MSI functions. We chose XML-RPC as the MSI protocol due to its simplicity and flexibility. This choice also simplifies caching, since the protocol runs over HTTP, which is understood by Squid.

For existing Web services, our proxies export an appropriate object hierarchy. For YouTube and Google Spreadsheets, for example, we provide a one-level hierarchy consisting of all the user's videos or spreadsheets, respectively. For Gmail we export a four-level hierarchy: folders/labels, threads, messages, and email content and attachments. The Yahoo mail hierarchy is similar but excludes the thread level, which Yahoo lacks. For Flickr, the user hierarchy consists of sets and photos. Finally, our Menagerie Desktop Web application, which allows users to create arbitrary structures to organize their Internet objects, exports whatever depth is dictated by those structures (see Section 4).

We use open-source Python Web service libraries [18, 38] to implement data managers wherever possible, extending them to support the complete set of MSI functions. For services that provide developer APIs (Flickr, YouTube, and Google Spreadsheets), the data manager implementation is straightforward; we simply issue the appropriate REST [10] or SOAP [37] calls to fetch the needed data and construct the MSI return value from the XML result. For example, to fetch the list of photos in a Flickr set, we issue a 'flickr.photosets.getPhotos' request to the Flickr REST API service [43].

Some services do not provide developer APIs to operate on their data (namely, Gmail and Yahoo!Mail). Building proxies for these services was much more difficult and required us to emulate browser-service communication protocols. We used several techniques to reverse engineer existing protocols and implement our proxies: for example, we extracted information from Web page source code; we used Ethereal [9] to trace browser-server communication to gather redirection targets; we scraped pages; and we identified situations where required information could be pieced together only from multiple page reads. Overall, our experience suggests that while a system like Menagerie can be built transparently *without* service support, even small amounts of such support greatly simplify the task.

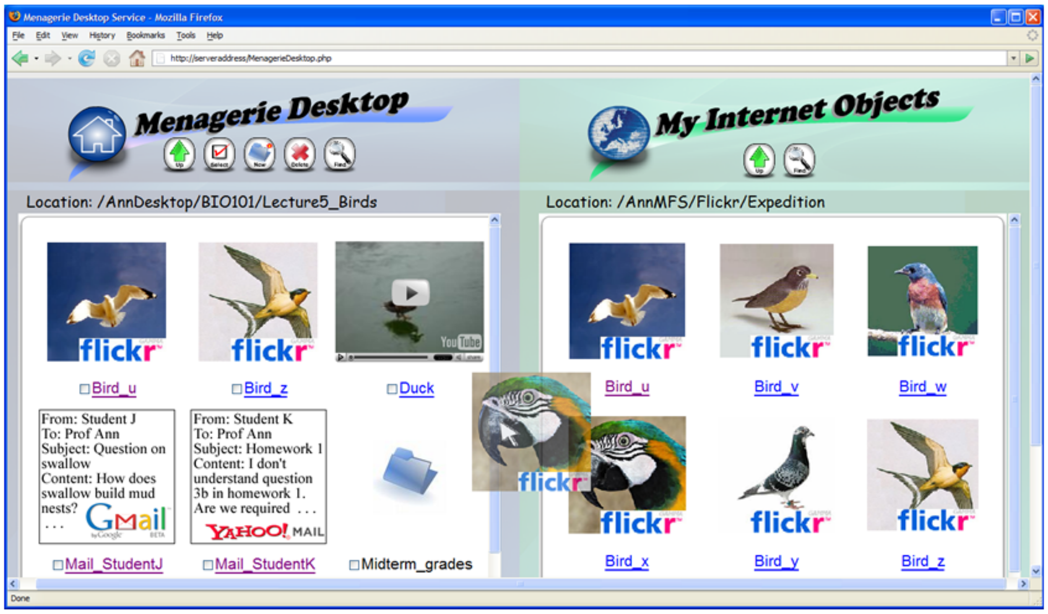
3.3 MFS Implementation

For programming and debugging convenience, we prototyped the Menagerie File System in Python on top of FUSE. Users mount their Web service name hierarchies using MFS, and for each mounted Web service, MFS mirrors the user's hierarchy for that service in a separate directory. For example, if Ann mounts her Flickr and Gmail hierarchies, she will see two directories under her MFS mountpoint, one for each service. Each directory will appear to contain her objects as exported by a service; in reality, however, MFS only maintains capabilities to the mirrored objects. When MFS performs an operation on an object, however, it also caches any object metadata that operation returns.

To mount a service hierarchy, the user or application simply creates a new top-level directory (e.g., with `mkdir`) in the MFS mountpoint (e.g., `/mfs`), specifying the name of the new directory and the capability for the service hierarchy to mount. MFS then creates a directory with the given name to mirror the object tree addressed by the capability. For example, suppose that Bob has a capability $\texttt{Capa}_i$ for one of Ann's photo sets on Flickr. To mount Ann's set on his local machine, Bob issues the command: `mkdir "/mfs/AnnPhotos Capa`$_i$`"`; this causes Ann's Flickr set to be mounted under the MFS directory named AnnPhotos.

All file system operations within the MFS mountpoint are directed by the Linux Virtual File System (VFS) to FUSE, which passes them to MFS. MFS then performs the appropriate Menagerie function on the specified object. Therefore, calls such as `getattr`, `readdir`, `read`, `write`, and `mkdir` are translated by MFS into corresponding MSI filesystem-related and hierarchy traversal calls to the appropriate Web service. For example, to list the contents of his AnnPhotos directory, Bob issues the command `ls /mfs/AnnPhotos`. This causes MFS to send an XML-RPC to the Flickr MSI service proxy, giv-

[1]While the service could maintain the revoked capability's (ID,password) in its CapTable to prevent dangling references, this is not strictly required given the 192 bits that uniquely identify a capability.


Menagerie Desktop Service - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://serveraddress/MenagerieDesktop.php
Menagerie Desktop
Up Select New Delete Find
My Internet Objects
Up Find
Location: /AnnDesktop/BIO101/Lecture5_Birds
Location: /AnnMFS/Flickr/Expedition
flickr
YouTube
Bird_u
Bird_z
Duck
Bird_v
Bird_w
Bird_x
Bird_y
Gmail
YAHOO! MAIL
Mail_StudentJ
Mail_StudentK
Midterm_grades
Done

From: Student J
To: Prof Ann
Subject: Question on swallow
Content: How does swallow build mud nests? . . .

From: Student K
To: Prof Ann
Subject: Homework 1
Content: I don't understand question 3b in homework 1. Are we required . . .

Figure 6: **Screenshot of the Menagerie Desktop Service (MDS).** The screenshot shows how Ann organizes her BIO101 class resources. The right half of the page shows Ann's photos on Flickr that she took during her expeditions. The left half shows Ann's desktop folder for the BIO101 lecture on Birds. Her BIO101/Lecture5_Birds folder contains two Flickr photos, a YouTube video, some emails related to the lecture, and a subfolder. Ann is now dragging a photo of a rare bird onto her MDS desktop to add it to the new directory.

ing it the capability `Capa`$_i$ and the objectID corresponding to Ann's photo set. The call returns the list of Ann's photo names in the set and the list of uniqueIDs of the photos. Other file operations translate similarly to appropriate Web service actions. For example, the command `rm /mfs/Flickr/expedition/owl` deletes the owl photo in Ann's expedition photo set. The command `mkdir /mfs/Gmail/BobMsgs` creates a new BobMsgs label/folder in Ann's Gmail account.

Since most MSI functions require objectIDs, MFS keeps track of the mappings between local inode numbers and object IDs. MFS obtains objectIDs either from `getattr` calls or from `ls` calls and saves them in a hashtable indexed by the local inode number. When an operation to a file or directory occurs, MFS translates the path into an inode number and looks up the objectID.

3.4 Caching

MFS uses two caches. A file system metadata cache saves short-lived inode attributes, parts of which are invalidated whenever an update is performed through MFS. We use the Squid [7] proxy cache to store large, long-lived objects from XML-RPC calls (particularly `read` and `embed_tag`). This cache has two effects: it reduces the download time for Web objects and more important, it reduces the workload on the Web services by reducing the number of object requests that are made to them.

We employ a standard Web-caching policy, i.e., by using the XML-RPC protocol, which is based on HTTP, we can leverage HTTP's cache timeout specification mechanism. We modified the XML-RPC library to supply `Cache-control` headers based on the timeout values included in the service modules' RPC responses. These headers are used by Squid to cache the data transparently to both the service modules and MFS. Thus, the services control the cache timeouts of their data.

4 Menagerie Applications

This section presents examples of the use of Menagerie in Web and desktop environments. We first describe two new native Web services that we built on top of the Menagerie File System. We then show three simple examples of the use of existing desktop commands to perform interesting and useful functions on Web objects. Our presentation is brief, in part because these applications were simple and quick to build, reflecting the power of the Menagerie approach.

4.1 Menagerie Desktop Service

The Menagerie Desktop Service (MDS) is a Web-based application that lets users organize, manage, and share their Web service objects. MDS users can mount hierarchies exported by multiple Web services, create new directories on an MDS "virtual desktop," populate those new directories with objects from multiple Web services, and share the directory hierarchies with others.

MDS has a simple graphic user interface similar to those used by Total Commander [11], or WinSCP [25]. Figure 6 presents a screenshot of the MDS interface, showing how Ann uses MDS to organize her BIO101 class resources. Initially, Ann registered with MDS, logged into the service, and mounted her Web service hierarchies (e.g., Flickr, Gmail, and YouTube) by pasting capabilities for those hierarchies into a Web form. MDS retained the capabilities so it could remount those hierarchies whenever she logs in.

The MDS Web page is divided in half. On the right is a view of Ann's Web service objects. When she logs into MDS, this view contains a folder for each of her mounted Web services. Clicking on a folder opens it to reveal the objects inside. In Figure 6, Ann has previously clicked on her Flickr folder and then opened her set of expedition photos. The right pane thus shows photos from that Flickr set. The left pane of the MDS interface presents a virtual desktop used to construct new object organizations. Users can create new multi-level directories and populate them by simply dragging and dropping objects from the right to the left pane. In the figure, Ann created a directory for her lecture on birds; that directory currently includes several bird photos from Flickr, a YouTube video on ducks, emails from Gmail and Yahoo!mail, and a subdirectory that contains grade spreadsheets. The figure shows that Ann is in the process of dragging a Flickr bird photo onto her desktop.

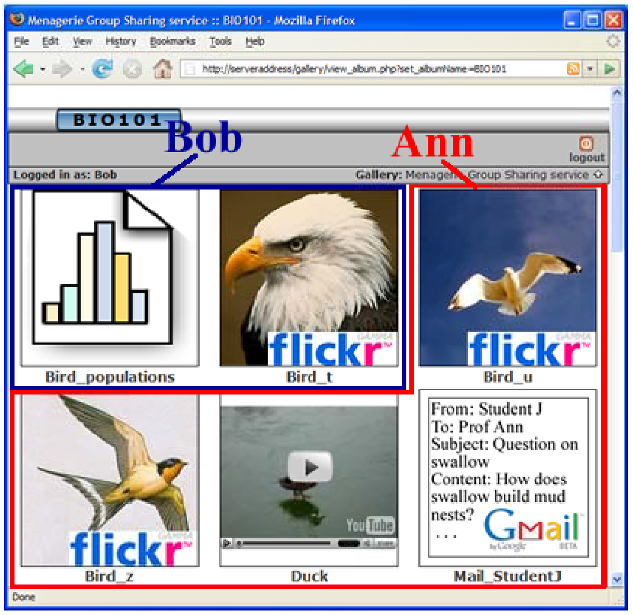
Unlike Total Commander or WinSCP, MDS does not fetch or move the contents of a Web object when a user drags it to the desktop. The MDS desktop is only organizational; objects remain in their respective Web services. Instead, MDS populates the user's organizational directories with symbolic links pointing to the objects in the user's mounted hierarchies. Although the Web objects are not downloaded, MDS can still embed thumbnails of the objects in its desktop page. To retrieve the HTML code that can display the thumbnail for a specific object, MDS reads the object's `EMBED_TAG` extended attribute value supplied by MSI, which causes MSI to issue an `embed_tag` call to the appropriate service. Thus, MDS treats objects generically and can handle objects of any type.

Our MDS implementation includes an MSI service module, which allows users to export new organizational structures. In this way, a user can request a capability for her MDS organization hierarchy and share her organization with others by passing that capability. Because MDS is a native MSI service, it requires no proxy.

MDS provides powerful and useful features but was extremely easy to build on top of our Menagerie prototype. With the Menagerie infrastructure in place, one developer implemented MDS in roughly 3 days. The MDS codebase contains 275 lines of code; 131 lines of Php code implement the application's logic, while the remaining code is related to HTML formatting.

4.2 Menagerie Group Sharing Service

The Menagerie Group Sharing Service (MGS) is a Web application that lets users form groups and share collections of Web objects or trees from object hierarchies on various Web services. Think of MGS as a kind of MySpace for


Menagerie Group Sharing service :: BIO101 - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://serveraddress/gallery/view_album.php?set_albumName=BIO101
BIO101
Bob
Ann
logout
Logged in as: Bob
Gallery: Menagerie Group Sharing service
Bird_populations
flickr
Bird_t
flickr
Bird_u
flickr
Bird_z
YouTube
Duck
GMail
Mail_StudentJ
Done

From: Student J
To: Prof Ann
Subject: Question on swallow
Content: How does swallow build mud nests?
. . .

Figure 7: **Screenshot of the Menagerie Group Sharing Service.** The screenshot shows how Ann and Bob, both users of the MGS service and members of the BIO101 group, share objects with their group. Ann shared some of her Flickr bird photos, a YouTube video, and an email, while Bob added a Flickr photo and a Google spreadsheet of bird populations.

groups rather than individuals.

We implemented MGS by modifying Gallery 1 [20], a popular open-source photo-sharing application. Our version of Gallery runs on MFS, displays any type of resource (not just photos), and supports user groups. Figure 7 presents a screenshot of our Gallery-based service, in which Ann and Bob have created a BIO101 group to share class information with their students. Ann shared several of her photos, a video, and an email on the BIO101 group page; she does not want to share her entire MDS directory with the class because it contains confidential emails and spreadsheets. Bob added one of his Flickr photos and his spreadsheet on bird population. All resources are displayed in the group's Web page on MGS. Adding resources to the page is similar to adding resources in MDS; the user pastes a capability into a form to give MGS access to an object or hierarchy.

Modifying Gallery to build MGS took a single day for one developer. We added only 73 lines of code, modified 3, and removed 91. Of the 73 new lines of code, 32 lines were related to HTML formatting.

4.3 Desktop Applications

Through its filesystem, which enables Web objects to be accessed as abstract files, Menagerie supports new or existing applications that manipulate collections of data scattered across Web services. We now show three simple examples of the use of existing desktop commands to perform important functions on Web objects.

Backup and Restore. Today's users have backup tools for safely archiving their desktop data. However, for user data stored by Web services, users must trust the service to maintain their data, perhaps forever. In addition, while many services offer persistence guarantees, very few offer versioning and the ability to recover from accidental deletion or modification of the user's objects.

Using Menagerie, a backup or restore of Web objects can be accomplished almost trivially with a simple set of existing applications or commands, such as `tar` and `untar` in UNIX. For example, assume that Ann has MFS on her PC and has mounted her Web service hierarchies, including her Flickr objects. To backup her "Expedition" photo set on Flickr, she would simply issue the following commands:

```
cd /mfs/Flickr
tar -cvzf /backup/Expedition.tgz Expedition
```

This creates a `tar` archive file in the /backup folder on Ann's PC of all of the photos in that Flickr set. Similarly, if Ann accidentally deletes that set and later wishes to restore it, she can do that easily as well:

```
cd /mfs/Flickr
tar -xvzf /backup/Expedition.tgz
```

Changing email providers. Many users maintain multiple email services and accounts for different purposes. For example, a user might have one email account for work, a personal account for communicating with friends, and a third for sending messages to "untrusted" third parties (e.g., those who might return junk mail). With the rapidly changing technical, business, and social environment, users may wish to migrate from one Internet or desktop mail system to another, or to consolidate multiple accounts. While some email services support interchange, this is not a general feature. Creating a new email account is much easier than migrating old messages. Menagerie greatly simplifies the task of email migration. For example, migrating from one mail account (e.g., on Yahoo!mail) to another (e.g., on Gmail) requires the following command:

```
cp /mfs/Yahoo/*/*/msg  /mfs/Gmail
```

This command processes all of the folders and message directories in the user's Yahoo!Mail, copying each `msg`, which contains the contents of an individual email, to the Gmail account. The result is to send each message to the user's Gmail account where it will appear in his Inbox folder. This solution is not perfect, though, for two reasons. First, the copy forwards all emails but does not recreate the same folder structure; a simple loop that first creates the folders (labels) easily solves this problem. Second, our implementation places attachments and message content in separate files, which makes copying an email with attachments more difficult; a 10-line script (omitted here) deals with this by combining the attachment and message into a single file before copying it to Gmail.

The key that enables this one-line email exchange is in the MSI service proxies that we built for Web email services (Yahoo and Gmail). Our proxies encode emails for transit in XML that identifies the parts of the message. In other words, the proxies implement a common schema for email messages. Given this schema, it becomes trivial to use MFS to download a message from one service and upload it correctly into another.

Synchronizing email contacts. Although some email services let users import contacts from other services, they do it in an ad-hoc manner in which each Web service knows how to fetch contacts only from the most popular other services.

With Menagerie, multi-email contact synchronization is simple because the distribution is transparent. In particular, the application need only understand contact formats and how to unify them. Since our proxies for Yahoo mail and Gmail export the same contact formats, as noted above, we can leverage existing file synchronization tools, such as Unison [23]. For example, synchronizing the contacts between Gmail and Yahoo email accounts can be done as follows:

```
cp /mfs/Yahoo/contacts/* /tmp/Y
cp /mfs/Gmail/contacts/* /tmp/G
unison /tmp/Y /tmp/G
cp /tmp/Y/* /mfs/Yahoo/contacts
cp /tmp/G/* /mfs/Gmail/contacts
```

In this example, the user copies his contacts into a local temporary file prior to running Unison because Unison creates its own temporary files in the directories it synchronizes. In Menagerie, executing Unison directly on the Web service files would result in the creation and then removal of new contacts on the Web service. To avoid this overhead, we first download the contacts locally, run Unison on them, and then upload the unified contact set. Note that we rely on the user to resolve conflicts, since neither Gmail nor Yahoo mail reports the time of the last contact modification.

4.4 Summary

In this section we described two new Web services built on top of Menagerie and the use of Unix commands to manipulate Web objects through MFS. In the first case, we showed how useful services can be constructed quickly and easily on top of Menagerie. In the second case, we showed the power of exporting Web objects through MFS. Our examples of using Unix commands to manipulate Web objects are not meant to be complete, but instead to stimulate the imagination of what might be possible (or even easy) given a Menagerie-style integration of the Internet services with the desktop.

5 Evaluation

This section evaluates the performance of the Menagerie prototype. We have not optimized or tuned Menagerie's performance to date; rather, our goal was to build a straightforward and extensible framework for experimentation. Nonetheless, our results demonstrate that performance of our current prototype is competitive with other remote access Web technologies and is fast enough to be usable in

Service	Oper.	Menagerie (ms)	Total (ms)	Menagerie percent
Gmail	ls	37	250	14.0%
	read	128	1,549	8.2%
Ymail	ls	35	955	3.6%
	read	121	3,943	3.0%
Flickr	ls	35	364	9.6%
	read	74	1,624	4.5%
GDocs	ls	41	348	11.7%
	read	122	3,194	3.8%

Table 1: **Menagerie latency compared to total latency for directory listing (ls) and remote data read (rd) on several services.** Menagerie is a small fraction of the total latency for existing Web services.

practice. A kernel-level implementation in C could clearly be faster than our user-level Python implementation, but we do not believe that this difference would be noticeable to clients in most cases.

For our measurements we ran MFS, Squid, and the applications on a Dell PC with an Intel P4 3.2 GHz CPU with 2GB of memory. We ran the MSI proxies for existing services on a separate machine with a similar configuration. Both machines ran Fedora Core 5 (kernel version 2.6.18), Squid 2.6, OpenOffice 2.1, and Firefox 1.5. The two machines were connected via a 100Mbps switch, but for some experiments we installed software traffic control on the Ethernet devices to simulate slower (broadband) connections.

5.1 Menagerie Performance Breakdown

We first examine the latency for two simple operations performed on Web services through Menagerie. Table 1 shows the latency for a directory listing and a remote data read through MFS to Gmail, Yahoo Mail, Flickr, and Google Docs. The data is read by a *cat* of a 47MB file. The table shows that Menagerie represents only a small fraction of the total latency (less than 15%) for these operations. Not surprisingly, network latency and service time dominate. For example, the Flickr directory listing takes 364 ms to complete, of which 35 ms (9.6%) are spent in Menagerie components (MFS, MSI, proxy).

To examine Menagerie's performance in more detail, we exclude the network and Web service times and account for the time spent in the various Menagerie components. For this measurement, we logged messages at key places (e.g., just before MFS issues an XML-RPC request to a proxy, when the corresponding RPC function is called in the proxy, etc.) and computed the time spent in different components by subtracting the timestamps of the appropriate messages. To avoid problems due to clock desynchronization between the proxy and MFS machines, we perform centralized network logging onto a third machine. Factored into the Menagerie latency is the time spent in six of its components: (1) the FUSE kernel module (including all kernel-space/user-space switchings), (2) MFS, (3) XML-RPC, (4) Squid, (5) the protection manager (capability validation, credential translation, etc.), and (6) the proxy (which includes parsing and building requests to Web services).

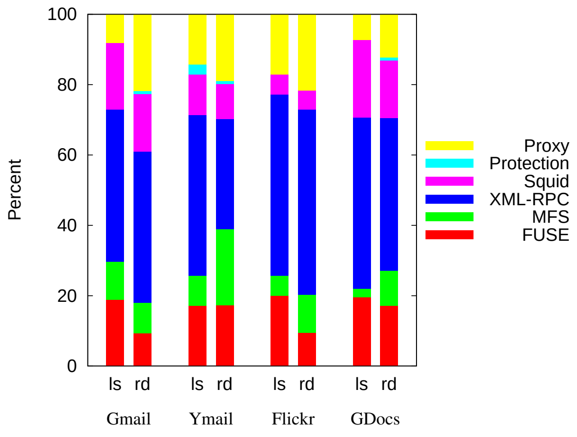


Figure 8: **Breakdown of latency by Menagerie component for directory and read operations on four Web services.** The Python-based XML-RPC library dominates the Menagerie latency, while FUSE and the proxies are the next largest factors.

	Time (ms)		
	Cache hit	Cache miss (LAN)	Cache miss (broadband)
ls	6	20	37
read	1,129	1,472	55,161
write	N/A	2,460	54,032

Table 2: **Performance of list, read, and write file system operations for the MStore simple storage service**. Cached results return rapidly, while network transmission time dominates on cache misses over broadband.

Figure 8 breaks down latency for each of these six components. The dominating latency is attributable to the Python-based XML-RPC, which represents at least about one third of total latency in all cases, and about half on average. The time for FUSE and kernel/user switching represents as much as 20% of the total; this is due to our user-level MFS prototype and would be reduced in a kernel-level implementation of MFS. The use of proxies has a smaller impact on total latency, on average about 15.2%. Finally, execution in the MFS module is not a significant cost on its own. Thus, the greatest potential for improvement lies in the XML-RPC system. However, given the small cost of Menagerie compared to network latency and Web service time, it is not clear that such optimization is warranted.

5.2 Menagerie File System Performance

We now examine the performance of the Menagerie File System by executing list, read, and write operations on a 5MB file. For these measurements we built a simple storage service, called MStore, which exports a flat object hierarchy through MSI and can therefore be accessed using MFS. In this way we can focus exclusively on MFS by eliminating the latency due to service-internal processing and HTTP service communication.

Table 2 shows the execution times for the file system

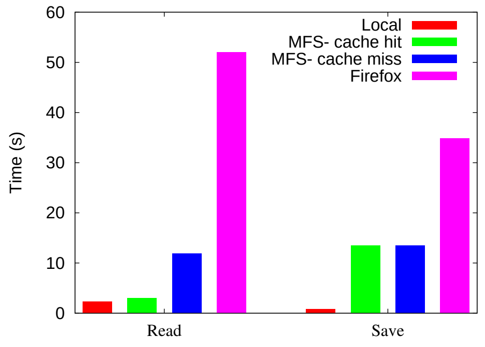


Figure 9: **Performance comparison of four spreadsheet-handling scenarios.** A comparison of opening and saving a spreadsheet in four cases: OpenOffice access to a local spreadsheet; OpenOffice access to a Google spreadsheet (shown for both a cache hit and cache miss); and Firefox browser access to a Google spreadsheet. An MFS cache hit is comparable to the local file case for reads; a cache miss is slower but still significantly faster than Firefox. For writes MFS is slower than the local case but still much faster than the browser.

operations under three conditions: (1) when the requested object is found in the Squid cache, (2) when a cache miss occurs and MStore is connected by a 100Mbps LAN, and (3) when a cache miss occurs when MStore is connected by a (simulated) low-end broadband link (two-way 1 Mbps bandwidth, 20 ms delay). Cached and high-speed LAN operations are relatively fast: 20 ms or less for list and below 2.5 seconds for the 5MB-file read/write. Our download/upload semantics for read/write influence the performance of these operations; we require MFS to store a temporary file for each downloaded/uploaded file to deal with application block-level access. As a possible improvement for reads, since Squid already stores the file as it comes in, we could integrate the local Squid cache with MFS to permit a cached file to be accessed directly by MFS.

When the proxy is accessed over a slow connection, read/write times increase dramatically and are dominated by network transmission time: sending a 5MB file over a 1Mbps link takes at least 41.9 seconds (in the ideal case), and thus Menagerie accounts for at most 24% of the total read time. As shown in the figure, caching improves performance significantly, especially when the proxy and MFS are separated by a slow network, which is usually the case for home users: hitting in the cache results in 98% improvement compared to the broadband time to read a 5MB file.

5.3 Application Performance

We next compare the performance of Menagerie with other Web alternatives for executing a common application function. For example, consider the task of opening, modifying, and saving a spreadsheet. Traditionally, users invoked desktop applications (such as Microsoft Excel [21] or OpenOffice [31]) to perform this task. With the advent of rich Ajax-based interfaces for online document editing (such as Google Spreadsheets), users can now perform the same task via their browsers. Menagerie presents a third alternative: users could open a Google spreadsheet with a local application (through MFS) and save their modifications back to Google.

Our experiment compares these three scenarios using OpenOffice, which we drive through an X-windows virtual keyboard program [28] to eliminate user-introduced latencies. Our experiment emulates the following operations: the user opens a spreadsheet, modifies 2 cells, and saves the file. We measure the time for reading and rendering a spreadsheet and for saving the spreadsheet by monitoring events from the OpenOffice window. In all cases, we handle the same 200KB spreadsheet that contains 139 lines and 100 columns of numbers (no graphs or formulas). We exclude application startup time.

Figure 9 shows the performance of four open/save scenarios: on a local spreadsheet with OpenOffice, on a Google spreadsheet accessed by OpenOffice through MFS (we show both a cache hit and cache miss on the object), and on a Google spreadsheet using the Firefox browser. From the figure, we see that the MFS-based solution achieves significantly better performance than the Web-browser-based one. Opening a spreadsheet with OpenOffice via MFS is at least 4.5 times faster than performing the same task with the browser, even for cache misses. For cache hits, the MFS solution is relatively close to the local solution. Saving the spreadsheet with MFS is approximately 2.3 times faster than saving it through the browser.

The slow speed of the browser solution is due mostly to the Ajax application and its required rendering and server communication in dealing with a 200KB spreadsheet. This may be addressed in the future with better optimized Ajax engines. At the moment, however, specialized applications (such as OpenOffice) achieve much better performance for the same task. Therefore, Menagerie not only supports new functions on Web objects (as witnessed by applications in Section 4), but also enables the use of existing applications to handle Web objects in a performance-competitive way.

6 Related Work

Menagerie builds upon many earlier efforts in distributed file systems and Web technologies. File systems for the World Wide Web – such as WebFS [35], UFO [2], and Alex [6] – provide read/write access to a global namespace comprising the entire Web. These tools can access Web objects that are addressable via URLs but cannot handle user-personal Web objects. That is, many of today's user objects are not directly addressable via URLs, because services produce them dynamically in response to HTTP POST requests. Menagerie provides the user with a global, unified namespace for personal Web objects that exposes internal Web service structures.

WebDav [39] provides a file system abstraction to access remote Web objects. Its main focus is the support of collaborative authoring of objects hosted by a Web service, providing good concurrency control under the context of that service. Menagerie provides a unified object namespace and fine-grained protection *across* services, which lets users access and share collections of objects hosted by multiple services.

Networked and distributed file systems – such as NFS [27], AFS [15], and Coda [19] – provide access to remote files. They do not support the HTTP namespace, however, and their protocols are limited to file-related operations. The Plan 9 [24] file system views all digital resources as files. We build upon this idea to make Web objects accessible through a simple file interface. SemanticFS [12] proposed the idea of using a virtual directory interface for search. We use this idea to support integrated search over a user's Web objects. Unlike all these systems, MFS supports Web-related functions, such as service-local searching and fetching of embed tags, and provides fine-grained sharing via capabilities.

Capability protection has been used in many operating systems and distributed systems [8, 30, 33, 41]. We borrow the password-capability model [4] from previous systems such as Amoeba [33] and Opal [8]. Our hybrid capability mechanism is related to the authorized/unauthorized pointer model first used in the IBM System/38 [5], which merges capabilities with ACL-based authentication. Menagerie capabilities give services the choice of automatically authenticated access via capabilities or controlled access that combines capabilities and user authentication.

Single sign-on systems (such as Microsoft Passport [22]) have been proposed to allow users to login to many services with a single account. While single sign-on simplifies user account management, it does not address several of our goals, such as fine-grained sharing and support for heterogeneous collections of Web-service objects.

Many communication technologies are currently in use. CORBA [36] and DCom [26] provide communication standards but suffer from compatibility and firewall problems. Web communication protocols – such as SOAP [37], REST [10], and XML-RPC [40] – address these problems by using self-describing XML messages and HTTP. We chose XML-RPC as the communication protocol between proxies and MFS because of its simplicity. However, this choice is independent of our architecture, and we could easily use another protocol instead.

Various file system interfaces to Web services let users access their personal Web objects and run desktop applications on them [16, 17]. None of these supports the integration of resources from multiple Web services or the sharing of heterogeneous Web objects.

Tools such as bookmarks and bookmarking Web services [42] can be used to organize and share Web objects that are directly addressable via URLs. However, these techniques fall short for user-personal objects, which are not always addressable via URLs (e.g., Gmail messages).

While Menagerie is closely related to these previous systems, it is unique in its integration of: (1) global naming and fine-grained protection for user-personal Web service objects, (2) transparent access to those objects using standard applications, and (3) extended functions supporting needed Web operations (such as search). Finally, we believe that Menagerie is the first system designed to address new problems created by the rapid expansion of Web storage and software services.

7 Conclusions

The move from PC-centered to Web-based computing and data storage poses new challenges for users and applications. This paper described Menagerie, a software framework that supports uniform naming, protection, and access for *personal objects* stored by Web services. The Menagerie Service Interface lets clients import and manipulate personal object structures that reside in and across Web services. Built on MSI, the Menagerie File System lets existing and new applications access those objects remotely and transparently through a conventional set of file operations. Menagerie speeds and simplifies the creation of new Web services and applications that support novel organizations and sharing of complex multi-Web-service objects.

We designed and implemented a Menagerie prototype and integrated a set of existing Web services: Gmail, Google Docs, Flickr, YouTube and Yahoo! Mail. We also built two interesting Menagerie services: the Menagerie Desktop Service and the Menagerie Group Sharing Service. Our experience with Menagerie underscores the power of this approach and its potential for enabling new Web object organization and sharing tools. Finally, our measurements show that a Menagerie-like system can provide performance commensurate with existing Web-object access techniques.

References

[1] Adobe. Web-based Video Remix. `http://www.adobe.com`, 2007.

[2] A. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman. UFO: A personal global file system based on user-level extensions to the operating system. *ACM Transactions on Computer Systems*, 16(3):207–233, 1998.

[3] Amazon.com, Inc. Amazon S3. `http://aws.amazon.com/s3`, 2007.

[4] M. Anderson, R. Pose, and C. Wallace. A password capability system. *The Computer Journal*, 29(1):1–8, 1986.

[5] V. Berstis. Security and protection in the IBM System/38. In *Proc. of the 7th Symposium on Computer Architecture*, 1980.

[6] V. Cate. Alex – A Global File System. In *Proc. of the USENIX File System Workshop*, 1992.

[7] A. Chadd, R. Collins, H. Nordstrom, G. Serassio, S. Wilton, A. Rousskov, and D. Wessels. Squid

Web Proxy Cache. `http://www.squid-cache.org/`, 2006.

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

[9] Ethereal, Inc. Ethereal: Network Protocol Analyzer. `http://www.ethereal.com/`, 2007.

[10] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[11] C. Ghisler. Total Commander. `http://www.ghisler.com/`, 2007.

[12] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O'Toole. Semantic file systems. In *Proc. of the 13th ACM Symposium on Operating Systems Principles*, 1991.

[13] Google, Inc. Blogger. `http://blogger.com`, 2007.

[14] Google, Inc. Google Docs and Spreadsheets. `http://docs.google.com`, 2007.

[15] J. Howard, M. L. Kazar, S. G. Meneea, D. A. Nichols, M. Satyanarayanan, R. N. Sidebothham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, 1988.

[16] M. R. Jain. FlickrFS. `http://manishrjain.googlepages.com/flickrfs`, 2005.

[17] R. Jones. GmailFS. `http://richard.jones.name/google-hacks/gmail-filesystem/`, 2004.

[18] B. Jorgensen. flickrapi. `http://beej.us/flickr/flickrapi/flickrapi.html`, 2007.

[19] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. In *Proc. of the 13th ACM Symposium on Operating Systems Principles*, 1991.

[20] B. Mediratta. Gallery: Your photos on Your Website. `http://gallery.menalto.com/`, 2007.

[21] Microsoft Corp. Excel home page. `http://office.microsoft.com/en-us/excel/default.aspx`, 2007.

[22] Microsoft Corporation. Microsoft Passport. `http://www.passport.com/`, 2007.

[23] B. C. Pierce. Unison: File Synchronizer. `http://www.cis.upenn.edu/~bcpierce/unison/`, 2001.

[24] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, 1995.

[25] M. Prikryl. WinSCP: Free SFTP and SCP Client for Windows. `http://winscp.net/eng/docs/start`, 2007.

[26] W. Rubin and M. Brain. *Understanding DCOM*. Prentice Hall, 1999.

[27] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun Network Filesystem. In *Proc. of the Summer 1985 USENIX Conf.*, 1985.

[28] T. Sato. Virtual Keyboard for X Window System. `http://homepage3.nifty.com/tsato/xvkbd/`, 2004.

[29] M. D. Schroeder, D. K. Gifford, and R. M. Needham. A caching file system for a programmer's workstation. In *Proc. of the 10th ACM Symposium on Operating Systems Principles*, 1985.

[30] J. Shapiro, J. Smith, and D. Farber. EROS: a fast capability system. In *Proc. of the 17th ACM Symposium on Operating Systems Principles*, 1999.

[31] Sun Microsystems. Openoffice.org. `http://www.openoffice.org/`, 2007.

[32] M. Szeredi. Filesystem in Userspace. `http://fuse.sourceforge.net/`, 2007.

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

[34] ThinkFree, Corp. ThinkFree. `http://www.thinkfree.com/common/main.tfo`, 2006.

[35] A. Vahdat, T. Anderson, M. Dahlin, E. Belani, D. Culler, P. Eastham, and C. Yoshikawa. WebOS: Operating system services for wide area applications. In *Proc. of the 7th Symposium on High Performance Distributed Computing*, 1998.

[36] S. Vinoski. CORBA: integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 14(2), 1997.

[37] W3C. SOAP. `http://www.w3.org/TR/soap/`, 2004.

[38] Waseem. Libgmail. `http://sourceforge.net/projects/libgmail/`, 2004.

[39] E. J. Whitehead, Jr. and Y. Y. Goland. WebDAV: A network protocol for remote collaborative authoring on the web. In *Proc. of the European Conf. on Computer Supported Cooperative Work, Denmark*, 1999.

[40] D. Winer. XML-RPC Specification. `http://www.xmlrpc.com/spec`, 1999.

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

[42] Yahoo! Inc. Del.icio.us Social bookmarking. `http://del.icio.us`, 2007.

[43] Yahoo! Inc. Flickr API. `http://www.flickr.com/services/api/`, 2007.

[44] Yahoo! Inc. Flickr Photo Sharing. `http://flickr.com`, 2007.

[45] Yahoo! Inc. Jumpcut. `http://www.jumpcut.com/`, 2007.

[46] Yahoo! Inc. Yahoo! Photos. `http://photos.`

yahoo.com, 2007.
[47] YouTube, Inc. YouTube: Broadcast Yourself. http://youtube.com, 2007.