Organizing and Sharing Distributed Personal Web-Service Data

Roxana Geambasu, Cherie Cheung, Alexander Moshchuk,
Steven D. Gribble, and Henry M. Levy

Department of Computer Science & Engineering
University of Washington, Seattle, WA, USA 98195
{roxana, cherie, anm, gribble, levy}@cs.washington.edu

ABSTRACT

The migration from desktop applications to Web-based services is scattering personal data across a myriad of Web sites, such as Google, Flickr, YouTube, and Amazon S3. This dispersal poses 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 their data with standard applications or scripts.

In this paper, we show that a Web-service interface supporting standardized naming, protection, and object-access services can solve these problems and can greatly simplify the creation of a new generation of object-management services for the Web. We describe the implementation of Menagerie, a proof-of-concept prototype that provides these services for Web-based applications. At a high level, Menagerie creates an integrated file and object system from heterogeneous, personal Web-service objects dispersed across the Internet. We present several object-management applications we developed on Menagerie to show the practicality and benefits of our approach.

Categories and Subject Descriptors

H.3.4 [**Systems and Software**]: Distributed systems; H.3.5 [**Online Information Systems**]: Web-based services, Data sharing

General Terms

Design, Performance

Keywords

Web services, Menagerie, data sharing, data organization

1. INTRODUCTION

The Web is catalyzing a transition from PC-based software and file systems to Internet-based applications and Web services. In the past, users relied solely on their desktop systems to execute applications and store their personal data. Today, many desktop applications have feature-rich "software-as-a-service" counterparts, including Web-based email systems, media editing tools, and office productivity suites. Similarly, services such as Flickr, YouTube, Blogger, and Amazon's S3 allow users to store, edit, and share their data via the Web.

WWW 2008, April 21–25, 2008, Beijing, China.
ACM 978-1-60558-085-2/08/04.

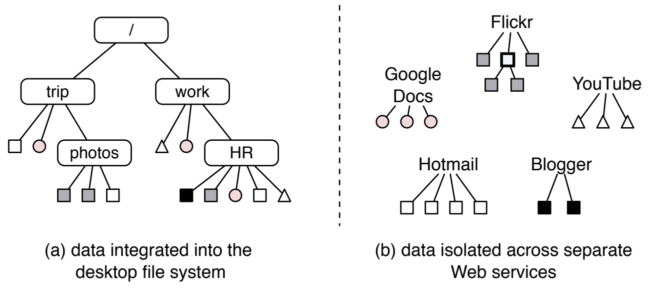

/
trip
work
photos
HR
(a) data integrated into the desktop file system
Flickr
Google Docs
YouTube
Hotmail
Blogger
(b) data isolated across separate Web services

Figure 1: **PCs vs. Web services.** In the desktop-centric world, users can organize and share their application data through the file system. In today's Web, data is increasingly trapped inside the Web service that operates on it.

Web-based services offer compelling advantages over traditional desktop software. Specifically, users can access their Web services and data through multiple devices from anywhere in the world. The Web eliminates administrative tasks such as software installation and update, and it facilitates the network effects that come from having a large community of connected users.

However, PC-based systems have compelling advantages of their own, many of which arise from the functions provided by the desktop operating system and file system. The OS supports a set of common, beneficial services that we take for granted. Users can name, organize, and access all of their files within a single hierarchical namespace, irrespective of which applications natively operate on them (Figure 1a). Similarly, applications written by different software vendors can interact with each other through the protected sharing interfaces exposed by the OS, providing users with new composite functions.

As the transition to the Web continues, we risk losing the advantages enjoyed by desktop systems. Users' personal data and objects are frequently trapped inside the different Web services that they use (Figure 1b). Consequently, users and services face a set of significant new challenges:

Data organization and management. On the desktop, a user can create a folder to hold related files and objects. On the Web, users' data is scattered across the Internet, where it is housed by a myriad of independent Web services. Given this, how can she organize, manage, archive, or search her Web objects and files *as a unit*?

Protected data sharing. While publishing is greatly simplified in the Web service environment, protected sharing, particularly at a fine grain, becomes more difficult. For example, how should one user share a specific subset of her objects with another user? Does the other user need to create accounts on all relevant Web services, and if so, do all of these services support the restricted sharing of only a select object subset?

Data manipulation and processing. Web services restrict the operations that can be performed on their objects: they typically export a limited API 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. How should we balance the need for Web services to retain ownership over the data and functions they provide, with the benefits that would be gained by allowing third parties to extend services?

This paper examines these challenges. First, we discuss the principles and requirements that must underlie any solution. Next, we discuss the design and implementation of Menagerie, a proof-of-concept system that embodies our solution principles. Menagerie consists of two primary components: (1) the Menagerie Service Interface (MSI), an API that facilitates inter-Web-service communication and access control, and (2) the Menagerie File System (MFS), a software layer that allows "composite Web services" to integrate remote Web objects into a local file system namespace, reducing the engineering effort required to access and manipulate remote data.

To demonstrate the value of our approach, we have prototyped several new Web applications on top of Menagerie. Our experience shows that it is possible to combine the ease of use, publishing, and ubiquitous access advantages of Web services with the organizational, protected sharing, and data processing advantages of desktop systems.

2. MOTIVATION

In this section, we use a simple motivating scenario to expose some of the shortcomings of the Web. From this scenario we derive a set of required properties that a solution must have to overcome these limitations.

2.1 A Simple Scenario

Figure 2 illustrates our simple motivating scenario. We consider Ann, a product manager for a small company. Ann has moved wholeheartedly to Web services for both her personal and business data and information processing needs. Specifically, Ann uses Flickr to manage her photo albums, Google Docs for spreadsheets and word processing files, Hotmail to communicate with colleagues and family, and Schwab to view an interactive stock ticker and maintain her personal financial information.

Ann likes to keep her data well organized. In the past, she used her PC's desktop manager to create folders in which her related files were stored or linked. Since many of her documents are now Web-based, she would like to create virtual Web folders that are populated with links to the appropriate Web objects and collections. For example, she would like to collect all of her product marketing resources into a single folder, in spite of the fact that they are spread across many Web services.

Ann also wishes to securely share some of her virtual folders with her colleagues, granting them access to view and edit the folders' contents. However, she does *not* want her colleagues to have access to all of her business files or to her personal files. In addition, not all of her colleagues have accounts on the same Web services as Ann.

Finally, Ann is extremely careful with her valuable data and wants to prevent against accidental deletion or an operational Web service failure. She would therefore like to use a third-party archival service to maintain historical versions of all of her Web objects and virtual folders.

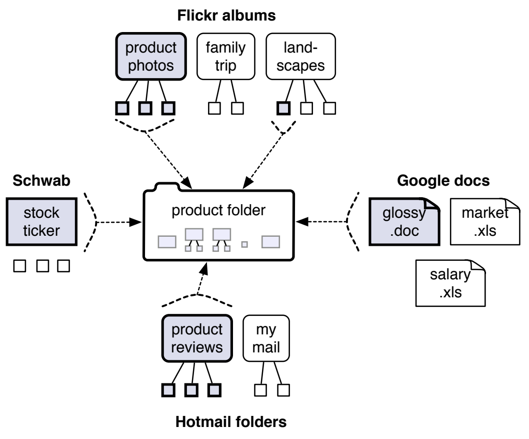


Figure 2: **Motivating Scenario.** Ann would like to create a new folder that links to some of her Flickr, Hotmail, Google Docs, and Schwab objects. As well, she wants to share the folder and its contents with her colleagues, who do not have accounts on all of these services.

2.2 Challenges

Given the limitations of today's Web, it is extremely difficult for Ann to accomplish her goals or for third-party Web services to help her. Ann faces three classes of obstacles:

Naming. The Web services in our example provide users with the abstraction of objects that can be manipulated in various ways. Unfortunately, not all of the services expose objects with a predictable, stable URL; instead, some objects are externally presented by the Web service as a diffuse collection of HTML elements, images, frames, and JavaScript, whose URLs might be dynamically generated. Accordingly, users and third-party services have no easy way to *name* each of the objects that Ann wishes to collect into her virtual folders.

Protection. Ann needs to share some of her objects with her colleagues and with the third-party archival service, but she faces several protection-related impediments. Each Web service has implemented its own particular authentication, authorization, and sharing scheme. Thus, Ann's colleagues may need to create accounts on all services to fully access her shared objects.

Even if single-sign-on accounts existed across the Web, many services fail to offer flexible and fine-grained protection. In some cases, sharing is all-or-nothing. For such services, allowing Ann's colleagues access to her professional objects may also reveal her personal data. Sharing also may be limited in some ways; for example, some Web services do not allow the sharing of different subsets of objects with different subsets of users. Finally, some services provide secure URLs that the user can hand out to grant object access, but many of these services do not support the selective granting of write access or the revocation of rights.

Ann wants to grant her associates access to a single virtual folder, implicitly giving them access to all of the objects within it. Unfortunately, those objects are scattered across many different services, each with its own authorization scheme. Short of Ann giving a third-party aggregation service all of her Web credentials and trusting that service with her objects, such sharing cannot be achieved.

Externalization and embedded rendering. Most Web services do not expose object data directly to users and third party services. Instead, they graphically present objects and interaction controls as embedded elements within Web pages. In contrast, on desktop systems, the filesystem permits many programs, including file managers, file sharing applications, editors, archivers, and security scanners, to process the same data objects.

To realize our scenario, Web services must provide additional functions that most of them lack today. In particular, they must export externalized representations of their objects to allow third-party services, such as archival or indexing services, to operate on that data. For simple third-party services, the structure and semantics of the externalized representation does not matter: the object can be exported as an opaque set of bytes. For richer services, a standardized or well-known representation, such as MIME for email, would be more valuable.

Finally, Ann and her colleagues rely on a third-party service to create and access virtual folders, and to browse the files within them. To support this, origin Web services should provide useful metadata and facilitate composite graphical interfaces that would allow the objects to be rendered and operated on within arbitrary Web pages. Flash movies exported by sites such as YouTube are good examples of this.

2.3 Requirements of a Solution

In the PC-centric world, the operating system provides abstractions, system call interfaces, and utilities to help applications and users overcome the challenges we describe above. In the Web, there is no single trusted layer that users, browsers, and services can rely on. We therefore believe that a new service interface must be defined and adopted to provide the interoperability and integration needed to realize even our simple motivating scenario.

This service interface could be defined via conventions on top of the HTTP protocol (e.g., REST[8]), or new special-purpose protocols could be designed for this purpose. Regardless, the challenges we described motivate three clear requirements that the service interface must support:

1. **Uniform object namespace.** To address the naming challenge described above requires a single global namespace in which all personal data objects are embedded. That is, all of the objects and object collections that users manipulate should have a permanent, globally unique name within this namespace, allowing the Web service, its users, and third-party composite services to discover and depend upon these names.
2. **Fine-grained protection.** To support data sharing and composite services, a Web service must provide fine-grained protection of objects and collections. It should be possible for the user to share only a portion of her objects from a service, while keeping the other objects private. It should also be simple to aggregate and share collections of distributed objects.
3. **Unified minimal object access.** The combination of a global, hierarchical namespace and fine-grained, protected sharing of personal data allows users and services to find and share objects with each other. To be useful, however, the objects must support some standard set of access functions. As we argued above, the minimal set must include the ability for objects to be embedded and rendered within an arbitrary Web page, and for object data to be externalizable.

The next section presents the architecture and implementation of *Menagerie*, a proof-of-concept prototype we have developed to meet the challenges we have described. Menagerie allows us to experiment with new Web applications that support the organization and sharing of collections of heterogeneous Web service objects. We will describe those applications in Section 4.

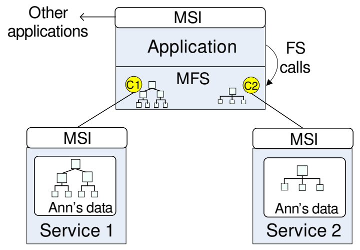


Figure 3: **The Menagerie Prototype.** The figure shows two Web services that export Ann's objects, a composite Web application built using the MFS layer, and the MSI capabilities (c_1 and c_2) that the application uses to access the objects.

3. THE MENAGERIE PROTOTYPE

This section describes the structure and implementation of our Menagerie prototype. Menagerie consists of two principle elements: the Menagerie Service Interface and the Menagerie File System. We briefly introduce these elements here and then describe them in more depth in the remainder of this section.

The Menagerie Service Interface (MSI) is an inter-Web-service communications API that is comprised of object naming, protection, and access operations. MSI defines a uniform, hierarchical name space into which Web services *export* the names of their objects. MSI supports fine-grained sharing of Web objects through the use of *hybrid capabilities*. This protection scheme allows users without service accounts to name and access objects, while also giving services the ability to limit the actions of such users. MSI also specifies a standard set of *object-independent access functions* for Web services. These functions support object reading and writing, rendering, and metadata export. While our goal is to design an interface that Web services can easily adopt, our prototype implementation also shows that Menagerie is deployable even without Web service support.

The Menagerie File System (MFS) simplifies the development of new, composite Web applications. MFS mounts remote MSI object hierarchies into a local file system name space, allowing an application to access remote Web objects through a standard file system interface. Figure 3 depicts a composite Web application that uses MFS to access the Web objects exported by two MSI-capable Web services.

The remainder of this section describes in detail MSI's naming, protection, and content operations. Figure 4 shows the functions we have implemented to date. This small set was sufficient to build our example applications; as we gain more experience, we expect the interface to evolve and grow.

3.1 Object Naming

We designed naming in Menagerie with two goals in mind. First, users must be provided with meaningful object names that correspond to the way users name objects inside of a Web service. Second, composite applications must be provided with global, unique identifiers for the objects they access, even though those objects are scattered across heterogeneous Web services.

In Menagerie, each Web service exports an *object name hierarchy* for each of its users. This hierarchy contains the user-readable names of all objects that each user can access. The structure of this hierarchy and the granularity of each object within it are left entirely up to the service, but it typically imitates the logical structure that the service exposes to its users. For example, Flickr offers its users abstractions associated with sets of objects (photo albums) and objects within each set (photos); therefore,

Namespace functions
list(capa, object_ID) returns *list of object names and IDs*
mkdir(capa, parent_ID, name)
getattr(capa, object_ID) returns *object attributes*

Protection functions
create_capa(capa, object_ID, rights) returns *new capa*
revoke_capa(object_capa, revoke_capa)

Content and Metadata functions
read(capa, object_ID) returns *byte[]*
write(capa, object_ID, name, content)
get_summary(capa, object_ID) returns *string*
get_URL(capa, object_ID) returns *string*

Figure 4: **The MSI interface**. This table shows the parameters and return types of each function. MSI services must support the naming and protection-related functions, and may optionally support the others.

Flickr could choose to export a three-level name hierarchy (e.g., `Ann/Disneyland-album/Mickey-photo`).

Each object in Menagerie is identified using a *service-local ObjectID*, which is unique within the service and independent of the object's location in the hierarchy. Using the service-local ObjectIDs, Menagerie mints globally unique object identifiers by combining the service-local ObjectIDs with services' DNS names. By making ObjectIDs unique on each service (as opposed to globally unique), we give services the liberty to create and name new objects independently. By making an object's ID independent of the object's location within the service's hierarchy, we ensure that caching and other optimization opportunities are preserved even if the object can be reached via multiple paths.

Three functions in MSI support name hierarchy operations: `list`, `getattr`, and `mkdir`. Given the unique ID of a collection node in a hierarchy, `list` returns the names of all the children of that node, as well as their unique IDs. `Getattr` returns the attributes of the object with the given ID, including the type of object, a capability for the object (see Section 3.2), the size of the object in bytes, and various additional metadata. `Mkdir` adds a collection object to the hierarchy. Individual objects are created using the MSI `write` function, as we will see in Section 3.3.

3.2 Protection

While designing Menagerie's protection model, we considered the two fundamental access control mechanisms: capabilities and access control lists (ACLs). These mechanisms generally lie at opposite ends of a spectrum. Capabilities simplify sharing, while ACLs enable tight access control and user access tracking. While our goal is to simplify fine-grained, distributed object sharing, we must also provide services with the ability to control and track access to their data.

Menagerie therefore adopts a *hybrid capability-based protection system*, which combines the benefits of both mechanisms. A Menagerie capability is an unforgeable token that contains the globally unique ID for an object and a set of 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 encoded in URLs that can be emailed or embedded in Web pages.

However, a Menagerie capability is also subject to control by the Web service whose object it names. A service can divide its object rights into two types: *open-access rights* and *closed-access rights*. An open-access right gives the holder of the capability direct access to the specified operation without further authentication; e.g., if the

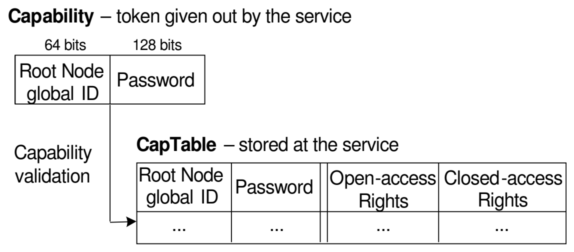


Figure 5: **Hybrid Capability Protection.** A capability provides access to objects within a sub-hierarchy rooted in the object identified by Root Note ID. Open-access rights allow direct object access on the basis of a valid capability, while closed-access operations also require user authentication.

right allows the user to read the object, then the service will return the object's contents when presented with a capability with the read bit set. Since a capability is not associated with any principal, an "open-access" operation cannot be attributed to a particular user.

A closed-access right, however, requires additional authentication. To perform an operation associated with a closed-access right, a capability with that right enabled is necessary but not sufficient: the user must also authenticate himself before the service will perform the operation. In most cases, this will require an account on that service. By "closing access" to an operation, the service can track the user that invokes the operation, or enhance the user's experience with personalized functions.

To implement capabilities, we use the *password-capability* model [4, 24]. The structure of a Menagerie capability is shown in Figure 5. The capability specifies a globally unique ID of a node in a service's hierarchy and it authorizes access to the entire subhierarchy rooted in that node. The capability also contains a long "password" – a random field chosen from an astronomically large number space. The password is generated by the service at capability creation time and ensures that the capability cannot be guessed. A service stores information about all capabilities it creates in a table called *CapTable*, whose structure is also shown in Figure 5. Because the service stores the capability rights, they cannot be forged by users.

As seen in Figure 4, every MSI method call passes at least two parameters: a capability token for an ancestor of the accessed object within the service's hierarchy and the object's ObjectID. Upon an MSI invocation, the service checks that the ancestor relationship holds and that a corresponding (`root node ID, password`) pair can be found in its CapTable. If not, the capability is invalid and the operation fails.

MSI provides functions for creating and revoking capabilities: `create_capa` and `revoke_capa`. When a user requests a capability from a service (using `create_capa`), the service returns a URL that embeds the new capability. In this way, capability sharing is similar to URL sharing in the Web. Revocation of a capability simply zeroes the rights fields in the capability's CapTable entry. To prevent arbitrary users from revoking capabilities, revocation requires a valid capability to the same object with the `REVOCATION` right enabled.

Several current Web services already use slight variations of a hybrid-capability protection model, which confirms the applicability of our approach. As one example, Flickr and other Yahoo! services provide "browser-based authentication [33]," which is essentially a capability-based scheme; it allows users to obtain a "token" for an object, specify a set of rights enabled by that token, and pass the token to an application. As another example, Google Calendar offers users "secret URLs" to their calendars, which they can give to friends. 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 add the sharee to the service's ACL.

Our hybrid-capability protection scheme meets our fine-grained sharing goal: it simplifies limited sharing of objects and collections, while providing services with control over more important operations. Menagerie's protection system is flexible enough to support all of the protection policies we encountered in the Web.

3.3 The Object Content Access Interface

MSI provides composite Web applications with two different ways to access objects. First, for mashup-style applications, Menagerie permits a composite application to embed an object from a remote service within a Web page. The remote service is responsible for the presentation and interaction controls of that embedded object, similar to how YouTube provides embeddable, interactive objects for displaying video.

To support building expressive composite GUIs, MSI defines a set of metadata access functions, including `get_summary` and `get_URL`. The `get_summary` function returns an HTML snippet that describes the object visually. For example, `get_summary` returns an `<img>` tag for a Flickr photo's thumbnail, an `<object>` tag for a YouTube video, and a summary for a Gmail email. The Menagerie Web Object Manager application in Section 4.1 uses this function to present distributed collections in a visual manner.

Similarly, `get_URL` provides the link to the object's URL within the parent service. Just as today's desktop file manager applications use a file-application binding database in systems like Windows to launch the appropriate application when the user double-clicks on a file, our Menagerie Web Object Manager uses URLs to redirect the user back to the parent service when a user clicks on a particular object.

Second, for composite applications that need to directly manipulate object contents, MSI provides a small, standard set of *object-independent* access functions. These functions, which include `read`, `write`, and `delete`, allow an application to download, manipulate, and upload the objects directly from Web services. MSI does not mandate any particular object representation or format: how a service chooses to externalize an object is entirely its own choice. Some composite applications, such as the archival service we described in Section 2.1, do not need to understand an object's format. Others, such as an indexing, image editing, or video distillation service, will need knowledge of the object's format. Over time, we expect services will gravitate towards standard object types and formats.

3.4 Implementation

Figure 6 shows the structure of our prototype Menagerie implementation. We chose to define MSI as an XML-RPC [30] layer on top of HTTP, so that services can make use of standard Web programming toolkits and frameworks to define, access, and export MSI functions. As well, by using XML-RPC, we could take advantage of existing Web caching components (such as Squid) within our composite applications to improve their performance.

To experiment with composite Menagerie applications, we needed to access Web services that support MSI. As an incremental deployment strategy, we have built *MSI proxies* for existing (non-MSI) services. An MSI proxy implements the MSI functions and Menagerie protection model on behalf of a service, making it MSI compliant without needing to modify the service itself. To date, we have implemented proxies for five popular Web services: Gmail, Yahoo! Mail, Flickr, YouTube, and Google spreadsheets.

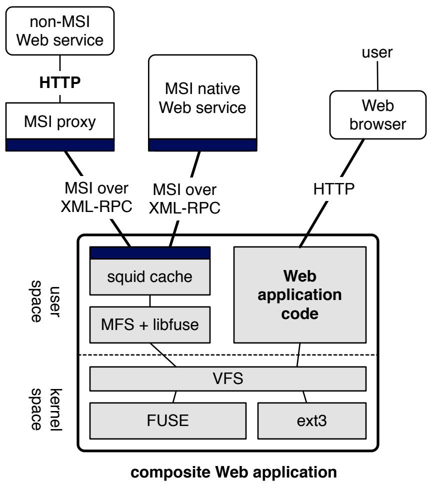


Figure 6: **Prototype implementation.** Our prototype system uses proxies to bridge legacy Web services to MSI. Composite Web applications can make use of MFS, which is implemented using the FUSE user-level file system framework. We have implemented MSI using XML-RPC, which is itself layered on HTTP.

For services that provide developer APIs, we found it easy to implement proxies, as we could simply bridge between the services' REST or SOAP functions and our associated MSI functions. For services that do not provide developer APIs, building proxies was more challenging, as we had to use awkward and unstable Web scraping techniques to access the appropriate service functions and objects. Overall, however, proxies are a more secure and practical incremental deployment path than requiring each composite service to perform Web service scraping in its own way.

3.4.1 The Menagerie File System

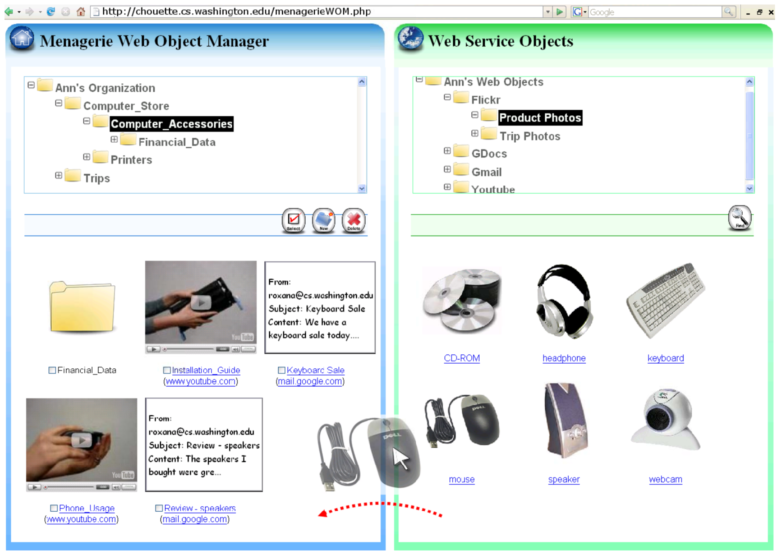
The Menagerie File System (MFS) is a user-level file system based on FUSE [23] that simplifies building composite applications. MFS lets a composite application *mount the MSI name hierarchies* exported by Web services into its local file system. As a result, an application can access remote MSI objects using standard file system operations and user-level programs.

To mount a service hierarchy, the composite application must receive a capability for that hierarchy from the user and then provide the capability to MFS. Once mounted, the service can then use standard file system commands and tools, such as `cp` and `tar`, to operate on the objects. These tools issue system calls such as `getattr`, `readdir`, `read`, and `write`. The calls get passed via VFS [15] to MFS, and then translated into the corresponding MSI calls on the remote Web services. As well, metadata functions in MSI are exposed as extended file system attributes through MFS.

To boost MFS's performance, we provide composite services with two caches. MFS has an internal metadata cache for rapid retrieval of short-lived file system metadata, and it uses the Squid [9] cache to store data returned by MSI `read` and `get_summary` functions.

3.5 Summary

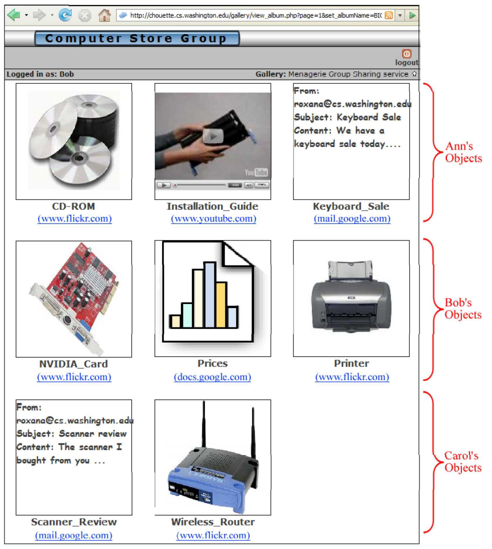
In this section, we described the architecture and implementation of our prototype Menagerie system. Through the use of Web service proxies and the MFS support layer, we made it possible for both existing and new Web services to communicate with each other through our Menagerie service interface. In the next section of the paper, we demonstrate the practicality and usefulness of our



From:
roxana@cs.washington.edu
Subject: Keyboard Sale
Content: We have a
keyboard sale today....

From:
roxana@cs.washington.edu
Subject: Review - speakers
Content: The speakers I
bought were gre...

(a) Menagerie Web Object Manager (WOM)



From:
roxana@cs.washington.edu
Subject: Keyboard Sale
Content: We have a
keyboard sale today....

From:
roxana@cs.washington.edu
Subject: Scanner review
Content: The scanner I
bought from you ...

(b) Menagerie Group Sharing Service (MGS)

Figure 7: **Screenshots of two Menagerie-based Web applications.** (a) This figure shows how Ann organizes her product-related Web objects using WOM. The right half shows the thumbnails of Ann's product photos on Flickr. The left half shows one of Ann's organizational folders, which already contains some objects. Ann is now dragging a product photo onto her new folder. (b) This figure shows how Ann and her colleagues Bob and Carol, all users of the MGS service and members of the `Computer Store` group, share objects with their group. Ann has shared some of a Flickr product photo, a YouTube video, and an email, Bob has added two Flickr photos and a Google spreadsheet of product prices, and Carol has put one Gmail email and one Flickr photo.

approach by building a set of powerful, easy to construct, composite Web applications.

4. MENAGERIE APPLICATIONS

As the trend towards Web-based applications continues, we believe that applications that support organizing, sharing, and manipulating distributed Web service objects will become increasingly important. This section presents several applications that we built using our Menagerie prototype. Our goal is to demonstrate the types of applications that Menagerie enables, and to show how Menagerie simplifies their implementation.

4.1 The Menagerie Web Object Manager

The Menagerie Web Object Manager (WOM) is a composite Web application that lets users organize and share their distributed Web objects. With WOM, users can create new virtual folders, populate those folders with collections of distributed Web objects, and share the folders with other users or services. WOM is a generic desktop, similar to file managers like Nautilus or Windows Explorer, but for Web objects. A WOM user can access and manipulate all of her Web service objects using the WOM Web interface; behind the scenes, WOM mounts and operates on the object hierarchies exported by the user's services.

The screenshot in Figure 7(a) shows how Ann organizes the Web resources for her business. When Ann first created her WOM environment, she mounted her Web service hierarchies (Flickr, Gmail, Google Docs, and YouTube) by pasting capabilities for those hierarchies into a Web form. WOM retains those capabilities and remounts the hierarchies using MFS whenever she logs in.

The WOM Web page is split in half. On the right, the user can navigate through her objects and mounted hierarchies. The expandable tree on top lists Ann's currently mounted hierarchies. In Figure 7(a), Ann has opened her Flickr `Product Photos` album.

The left side presents the user's virtual Web object folders. Users can create directory hierarchies and populate them by simply dragging-and-dropping objects from the right side of the interface to the left. In the figure, Ann has created a `Computer_Store` directory, containing sub-directories for `Computer_Accessories` and `Printers`. Ann is currently populating her `Computer_Accessories` folder; that directory includes two instructional YouTube videos, two customer emails from Gmail, and a folder with financial Google spreadsheets. The figure shows that Ann is in the process of dragging a Flickr product photo onto her `Computer_Accessories` directory.

WOM is only organizational; objects remain stored and managed by their respective Web services. Clicking on a object leads back to its origin Web service. For example, clicking on a Google spreadsheet in Ann's virtual folder causes Google Docs to popup a browser with that spreadsheet opened. Although the Web objects are only linked to the virtual folder, WOM can still render thumbnails of the objects. To retrieve the HTML code that displays the thumbnail for a specific object, WOM reads the object's `SUMMARY` extended attribute from MFS, which causes MFS to issue a `get_summary` call to the appropriate service.

Our WOM implementation exports MSI, which allows users to further export their new organizational structures. For example, a user can request a capability for a WOM virtual folder hierarchy and share that folder with other people and services by passing them that capability. Because WOM is a native MSI service, it requires no proxy.

WOM provides useful organization and sharing features, yet it was easy to build on top of our Menagerie prototype. One developer implemented WOM in roughly 3 days. The WOM codebase contains 275 lines of code: 131 lines of PHP code containing the application logic, and the remainder to perform HTML formatting.

4.2 The 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 from their Web services. MGS is similar to MySpace, but it is targeted at groups rather than individuals. That is, while WOM lets a single user create and share virtual object organizations, MGS lets several users share a single virtual desktop.

We implemented MGS by modifying Gallery 1 [18], a popular Web-based photo sharing application. Hence, MGS borrows its GUI from Gallery. We enhanced Gallery to run on Menagerie, to display any type of resource (not just photos), and to support user groups. Figure 7(b) presents a screenshot of MGS, in which Ann, Bob, and Carol have created a group called `Computer Store Group` to share business information amongst themselves and with other colleagues. Ann has shared a photo, a video, and an email on the group page; she does not want to share her entire WOM `Computer_Store` directory with the colleagues because it contains confidential financial data. Bob has added two Flickr photos and a spreadsheet with product prices, and Carol has added an email and a photo. All resources are displayed in the group's Web page on MGS. Adding resources to the page is similar to adding resources in WOM; 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. The conversion required only 73 new lines of code (32 related to HTML formatting), modification of 3 lines, and removal of 91 lines from Gallery.

4.3 MFS-based Applications

The WOM and MGS examples show how new Web-object management services can leverage the global naming, protection, and unified access functions that Menagerie provides. As well, the Menagerie File System lets any application treat Web objects as abstract files. As a result, services can apply *existing* file-based programs or scripting languages to remote Web objects or to the kinds of Web-object collections that Menagerie enables. Below, we give several examples of the power of this "backwards compatibility" provided by MFS.

Backup and Restore Service. 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, as no generic Web object backup-and-restore application exists.

Using Menagerie, a backup-restore service that operates on distributed Web object collections can be built with a simple set of existing applications or commands, such as `tar` and `untar` in UNIX. For example, suppose that Ann wants to back up her *distributed* WOM `Computer_Store` folder. Ann provides the capability to that folder to the backup-restore service, which uses the capability to mount Ann's object hierarchy. To the service, Ann's distributed Web objects look like a local UNIX file tree. Therefore, the backup-restore service can archive Ann's objects with the following commands:

```
cd /mfs/Ann/WOM
tar -czf /backups/Ann/Computer_Store.tgz \
        Computer_Store
```

This creates a `tar` archive in the /backups folder on one of the backup-and-restore Web service's machines. Provided that all capabilities involved have the `READ` right enabled, the `tar` causes backup-restore's MFS to `read` the contents of each object recursively, first from WOM and then from the appropriate hosting service. The resulting archive will contain the entire `Computer_Accessories` folder hierarchy and the contents of all the distributed objects in it. Similarly, the service can use `untar` to restore those objects at a later time.

Changing email providers. Users may wish to migrate from one Internet mail system to another, or to consolidate multiple accounts. While some email services support interchange, this is not a general feature. Menagerie can simplify the task of email migration. For example, a new third-party Web application for migrating from one mail account (e.g., Yahoo!Mail) to another (e.g., Gmail) could be built on Menagerie through MFS using the following, perhaps surprisingly simple, command:

```
cp /mfs/Ann/Yahoo/*/*/msg  /mfs/Ann/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 command assumes that the new changing-email-providers application has mounted Ann's Gmail and Yahoo!Mail hierarchies. The result is to send each Yahoo message to the user's Gmail account, where it will appear in her Inbox folder.

This example needs further explanation. First, this email exchange is facilitated by the fact that our Menagerie proxies for Gmail and Yahoo!Mail implement a common XML-based schema for emails. If the services did not export the same email format, the new application would need to perform a schema mapping for each email. Second, the copy command does not recreate the same folder structure; a simple loop that first creates the folders (labels) easily solves this problem. Finally, 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.

While the full explanation of this process is more complex than the single-line `cp` command above, the example shows the power of providing UNIX file access to object hierarchies.

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 easier 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. For example, a Web application for synchronizing the contacts between Gmail and Yahoo!Mail accounts can be done as follows:

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

In this example, the new contact synchronizer application copies the user's 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.

The MFS Desktop Bridge. MFS was designed to simplify building composite Web services, but it is also valuable as a desktop operating system component. By running MFS on a desktop, the

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.

user can mount and access her Web objects as files within the file system. As a result, the user can take advantage of desktop applications to operate on Web data: MFS acts as a bridge between the user's desktop and Web environments. For example, using MFS, we have used Adobe Photoshop to edit Flickr photos, Microsoft Excel to operate on a spreadsheet stored within Google Docs, and Nautilus to navigate through Web objects, all without changing the applications themselves.

4.4 Summary

In this section we presented example applications built using Menagerie. We showed how Menagerie lets services access Web objects through existing desktop applications and command languages. Our examples are not meant to be complete, but instead to stimulate the imagination of what is possible given the features that Menagerie provides. Overall, our examples demonstrate two key points. First, a common set of naming, protection, and access operations for Web services greatly simplifies the creation of new organization and sharing services for heterogeneous Web objects. Second, a file-access facility for Web objects provides a powerful path to leverage legacy command languages and applications in the new world of software as a service.

5. EVALUATION

In this section we evaluate Menagerie, focusing on three questions. First, what additional latency does the Menagerie layer add to Web service data accesses, compared to direct access without Menagerie? Second, which internal components of Menagerie are most responsible for overhead? Third, how does the performance of the MFS Desktop Bridge compare to other methods for editing personal data?

We have not spent effort to optimize or tune Menagerie; 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 practice.

For our measurements we created a Menagerie measurement service that ran Menagerie (including MFS, FUSE, and the Squid cache) and the measurement applications on 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, Squid 2.6, and Firefox 1.5. The two machines were connected via a 100Mbps switch.

5.1 Menagerie Overhead

For our first question – the additional cost of Menagerie in accessing Web service data – we measured the latency for two simple operations performed on Web services through Menagerie. Table 1

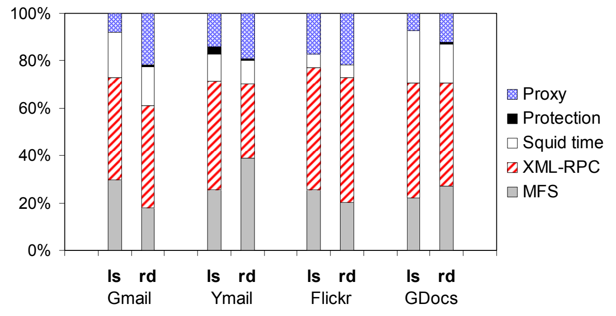


Figure 8: **Breakdown of latency by Menagerie component for directory and read operations on four Web services.** The Python-based XML-RPC library dominates the Menagerie latency; MFS is the next largest factor.

shows the latency for a directory listing and a remote data read invoked through Menagerie/MFS to Gmail, Yahoo Mail, Flickr, and Google Docs. The data is read by an application performing a *cat* of a 4.7MB 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, and the proxy).

To answer our second question – where the time goes inside of Menagerie – 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, such as just before MFS issues an XML-RPC request to a proxy, or when the corresponding RPC function is called in the proxy, and computed the time spent in different components by subtracting the timestamps of the appropriate messages.

Factored into the Menagerie latency is the time spent in five of its components: (1) the Menagerie File System (MFS), which has both user-mode and kernel-mode components, (2) XML-RPC, (3) the Squid cache, (4) the Menagerie protection manager, including capability validation and credential translation, and (5) the Web service MSI proxy, which includes parsing and building requests to the existing Web services.

Figure 8 breaks down the latency for these five components. In most cases, the dominating latency is caused by the Python-based XML-RPC, which represents about half of the total latency on average. The time spent in MFS represents from 20% to 38%, due primarily to our user-level file system code; this could be reduced in an all-kernel-level implementation. The use of Web-service proxies has a smaller impact on total latency, on average about 15.2%. The cost of the protection system is negligible in all cases. Overall, then, the greatest potential for improvement lies in the XML-RPC system. Given the small cost of Menagerie compared to network latency and Web service time, however, it is not clear that such optimization is warranted.

5.2 MFS Desktop Bridge Performance

Consider the task of opening, modifying, and saving a spreadsheet. Traditionally, users invoked desktop applications such as OpenOffice 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. The Menagerie Desktop Bridge presents a third alternative; for example, users can operate on a remote Google spreadsheet using local PC-based spreadsheet applications.

We compare these three scenarios in Figure 9, which shows the

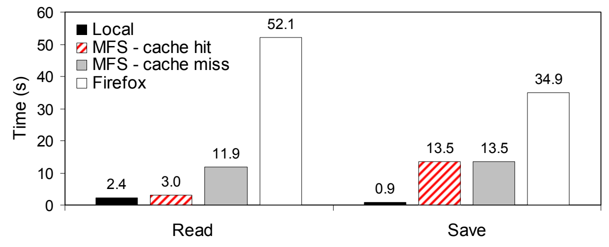


Figure 9: **Performance comparison of four spreadsheet-handling scenarios.** A comparison of opening and saving a spreadsheet in four cases: OpenOffice access to a local spreadsheet; OpenOffice access to a remote Google spreadsheet via the MFS bridge, shown for both a cache hit and cache miss; and Firefox browser access to a remote Google spreadsheet.

performance seen by the user when opening an identical spreadsheet, modifying 2 cells, and saving the file. For the MFS bridge scenario, we report the performance for two cases: OpenOffice hitting in the MFS squid cache, and missing in the cache. We used Firefox to access a remote Google spreadsheet in our third scenario.

For cache hits, the MFS solution performs nearly as well as accessing a local file. Surprisingly, saving the spreadsheet with MFS is 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 our 200KB spreadsheet. This may be addressed in the future with more optimized Ajax engines.

Overall, Menagerie supports new functions on Web objects, as witnessed by applications in Section 4, and it 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 Web technologies, protection system, and extensibility. The Semantic Web [2] effort, languages for describing Web service interfaces [5], and service communication protocols [8, 27, 30] enable applications to find and integrate Web service content. In this work, we identify the key components that any Web service interface must provide in order to enable a particular set of applications: generic applications for organizing, sharing, and processing Web data objects within user Web accounts.

Recently, the problems caused by the dispersal of users' data on the Web have received increasing attention from Web service providers. Web-data aggregation sites (e.g., iGoogle [10], Facebook [7], SecondBrain [21]), and Web-data processing applications [20] allow users to aggregate their Web objects from a set of supported locations, share them, or process them. Each of these applications must face the challenges of Web-account data integration (Section 2.3) on its own: it needs to devise its own naming for Web objects, often request full control from the user on his remote Web accounts, and write code to retrieve data from each service. Solving such challenges for each application is inefficient. Thus, we propose a new common service interface, which, if adopted, would facilitate the building of applications, including some of the ones enumerated above [20, 21].

Many individual Web services expose programmatic interfaces. Recently, some social applications have agreed to support OpenSocial, a common set of JavaScript and Google Data APIs for accessing social information [11]. Menagerie and OpenSocial have very similar goals. However, Menagerie is more general, as it is not restricted to social applications, while OpenSocial's API has the benefit of being tuned towards the needs of social application programmers. This tradeoff between generality and specificity is common to many systems [6].

The need to decouple user-account data from Web services and expose it to third-party applications has been recently formulated in the W5 project [16]. While some of their concepts overlap with Menagerie's, including fine-grained protection and data access, our contribution consists of a concrete instantiation of the required common interface, a working implementation, and experience with building useful applications to validate our approach.

The idea of using operating system concepts and abstractions to address problems on the Web has been used previously. WebOS [25] provides OS abstractions for building large-scale applications over the wide-area, including global naming and authentication. Menagerie provides functions typically fulfilled by the OS on the desktop to Web applications operating on the user's Web data. Similarly, Web file systems [1, 26, 29] enable the integration of Web resources with the local file system. Unlike Menagerie, these systems do not offer any support for sharing heterogeneous collections of objects. Specific Web services provide file system interfaces that let users access their Web objects and run desktop applications on them [13, 14]. None of these supports the integration of resources from multiple Web services or the sharing of heterogeneous Web objects. Yahoo! pipes [32] allows users to integrate RSS feeds and mashup Web site data using a visual, UNIX pipe-like editor. Unlike Menagerie, Yahoo! pipes does not facilitate the fine-grained, protected sharing of personal Web objects.

Capability-based protection [17] has been used in many operating systems and distributed systems [4, 22, 24, 31]. Our hybrid capability mechanism resembles the authorized/unauthorized pointer model first used in the IBM System/38 [3], which merges capabilities with ACL-based authentication. Menagerie capabilities give Web services the choice of automatically authenticated access via capabilities or controlled access that combines capabilities and user authentication.

Single sign-on systems have been proposed to allow users to login to many services with a single account [19]. While single sign-on simplifies user account management, it does not address fine-grained sharing and support for heterogeneous collections of Web-service objects.

Some projects have looked at improving the security of mashups within the current browsers [28, 12]. Most of these provide protection mechanisms for sharing of resources within the browser, while Menagerie's protection mechanism provides controlled sharing of objects within a Web service with a third-party application.

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 embedded rendering.

7. CONCLUSIONS

The move from PC-centric to Web-based computing and data storage poses new challenges for users and applications. This paper described the organizational, sharing, and data-processing problems faced by users and creators of modern Web services. We presented Menagerie, a software framework that supports uniform naming, protection, and access for *personal objects* stored by Web services. We designed and implemented a Menagerie prototype and integrated a set of existing Web services: Gmail, Google Docs, Flickr, YouTube and Yahoo!Mail. Using Menagerie, we built organization and sharing services for personal objects, including the

Menagerie Web Object Manager and the Menagerie Group Sharing Service. Our experience with Menagerie and its applications underscores the power of this approach and its potential for enabling and simplifying the construction of new composite Web services. Our measurements show that a Menagerie-like service interface can provide performance commensurate with existing Web-object access techniques.

8. ACKNOWLEDGMENTS

We thank Charlie Reis and Tanya Bragin for their feedback on this work and paper. This research was supported in part by the National Science Foundation under grants CNS-0132817, CNS-0430477, and CNS-0614975, by the Torode Family Endowed Career Development Professorship, by the Wissna-Slivka Chair, and by gifts from Nortel Networks and Intel Corporation.

9. REFERENCES

[1] 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 TOCS*, 16(3):207–233, 1998.

[2] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. Scientific American, 2001.

[3] V. Berstis. Security and protection in the IBM System/38. In *Proceedings of the 7th ISCA*, 1980.

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

[5] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web service definition language (WSDL). W3C, 2001.

[6] Dawson R. Engler and M. Frans Kaashoek. Exterminate all operating system abstractions. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems (HotOS-V)*, Orcas Island, WA, May 1995.

[7] Facebook. `http://www.facebook.com/`, 2007.

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

[9] National Laboratory for Applied Network Research. The Squid Internet Object Cache. `http://squid.nlanr.net`.

[10] Google, Inc. iGoogle. `http://google.com/ig`, 2005.

[11] Google, Inc. OpenSocial. `http://code.google.com/apis/opensocial/`, 2007.

[12] C. Jackson and H.J. Wang. Subspace: secure cross-domain communication for web mashups. In *WWW '07*, pages 611–620, 2007.

[13] Manish Rai Jain. FlickrFS. `http://manishrjain.googlepages.com/flickrfs`, 2005.

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

[15] S.R. Kleiman. Vnodes: an architecture for multiple file system types in Sun UNIX. In *Summer USENIX Conference Proceedings*, Atlanta, GA, June 1986.

[16] M. Krohn, A. Yip, M. Brodsky, R. Morris, and M. Walfish. A world wide web without walls. In *Proceedings of HotNets-VI*, Atlanta, GA, November 2007.

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

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

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

[20] Picnik, Inc. `http://www.picnik.com/`, 2007.

[21] SecondBrain. SecondBrain: All your Internet Content. `http://www.secondbrain.com/`, 2007.

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

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

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

[25] A. Vahdat, T. Anderson, M. Dahlin, E. Belani, D. Culler, P. Eastham, and C. Yoshikawa. WebOS: Operating system services for wide area applications. In *Proceedings of HPDC '98*, July 1998.

[26] Amin Vahdat, Paul Eastham, and Thomas Anderson. WebFS: a global cache coherent filesystem. Technical report, UC Berkeley, December 1996.

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

[28] H.J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and communication abstractions for web browsers in MashupOS. In *Proceedings of 21st ACM Symposium on Operating Systems Principles*, October 2007.

[29] E. James Whitehead, Jr. and Yaron 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.

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

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

[32] Yahoo!, Inc. pipes. `http://pipes.yahoo.com/pipes/docs`.

[33] Yahoo, Inc. Browser-Based Authentication (BBauth). `http://developer.yahoo.com/auth/`, 2007.