Secure File System Services for Web 2.0 Applications

Francis Hsu Hao Chen

Department of Computer Science
University of California, Davis
Davis, CA, USA 95616
{fhsu,hchen}@cs.ucdavis.edu

ABSTRACT

We present a design for a file system that provides a secure file storage service for Web 2.0 applications. Currently, each Web application stores its own user data. This not only burdens the applications with storing, managing, and securing user data but also deprives users from controlling their own data. With recent proposals of secure client-side cross-domain communication mechanisms, we can provide an independent file system service to Web applications. This service returns the control over user data back to the users, where users can share or restrict access to their files as they wish, and relieves web application servers from the contractual or regulatory obligation of safeguarding user data.

Categories and Subject Descriptors

H.3.5 [**Online Information Services**]: Web-based services, Data sharing; D.4.6 [**Security and Protection**]: Access controls

General Terms

Design, Security

Keywords

web application, mashup, file system

1. INTRODUCTION

A typical host-based computing environment consists of an operating system, a file system, and applications. Even though some of them may overlap (e.g., an operating system may contain a local file system, an application may reside on a file system), they are independent entities and can be provided by different vendors. Particularly, application vendors do not provide file systems and vice versa. The separation of file systems from applications provides several benefits. First, it simplifies application development. Developers can focus on the application logic; they need not manage data from multiple users and the related access control issues. Second, it facilitates sharing data between applications. Since the file system manages all the data, all applications share the same interface for accessing data, under the access control policy of the operating system. Last, it provides better security and privacy protection for users. Since the file system manages user data, vulnerabilities in an application cannot compromise user data when the user is not running the application.

In Web 2.0, web applications are hosted on web servers and run in browsers. Compared to traditional host-based applications, these applications provide several advantages to the users: they need not be installed on the client computer; they can run on any hardware and software platform that has a compatible browser. As a result, we witness that the number and popularity of these applications quickly increase.

However, current web applications have also lost several advantages of host-based applications. One big difference between host-based and web-based applications is that web applications merge the storage of data into the applications themselves. A user must go though the web application to get to his data.

This decision could result from three factors. First, web applications running on a web server gain performance benefits from keeping the file system nearby, since all of the user's requested operations travel over the network back to the web server. Second, web applications also benefit from designing their data storage component to optimally serve the requirements of web application. Third, since the Same Origin Policy on browsers prevents cross-domain communications, it is difficult for client-side code from one web site to access stored user data on another web site.

As a result of hosting the application and user data storage on the same web site, current web applications lose several benefits of host-based applications. First, they obligate their hosting web servers to provide data storage for all users of that application. This increases the cost to the web sites, and requires each user to have a separate account and storage at each web site. Second, it makes sharing data among different web applications difficult. Since each web application provides its own data storage, it likely has a home-grown data access API that is incompatible with other applications. Another issue is access control. Each data store likely has its own access control policy and user accounts. Sharing data among multiple web applications data stores like these can be complicated. Finally, users lose control of their data. User data is scattered across all the web sites that host web applications. The user has to register separately with each of these web sites.[1] The confidentiality and integrity of user data are at the mercy of the security and honesty of the web sites.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CCSW'09, November 13 2009, Chicago, Illinois, USA.
Copyright 2009 ACM 978-1-60558-784-4/09/11 ...$10.00

[1] Single sign-on systems partially alleviate this burden.

We argue that we should separate user data storage from the web application. The three previously mentioned obstacles to this approach are disappearing. First, some web applications are beginning to rely on third-parties for data storage. With cloud computing, those web applications no longer have their data store residing on same web server serving that web application. Instead the web application communicates with another entity providing the services for data storage and retrieval. These web applications lose the flexibility of a customized local storage system and must rely on the API of the storage provider. The web applications however, make this exchange for the scalability benefits from the storage provider. Second, some web applications can run outside of a web server. Thick-client AJAX versions of traditional desktop application like mail clients and word processors can run mainly in the browser on the client machine. Therefore, the advantage of hosting the file system on the web server, such as performance, is diminishing. Third, recently researchers have proposed mechanisms, such as MashupOS [19] and OMash [4], for enabling client-side cross-domain communications. Their motivation is to enable secure Web mashups, where an integrator creates contents from multiple providers. These techniques would also enable data storage services independent of web applications.

We wish to place a storage web application at the same level as the web application providing other services. We propose a secure file system service for web applications. Like existing cloud storage systems, the service provides a file system at the server and an API to web applications at the client. We, however, make the file system directly accessible to the user via his browser. We provide the user access to his data on the storage service, without the filter of other web applications. We can view the file system service and the web application as two providers in a Web mashup in the browser. After the browser loads the mashup integrator, web application and file system service, the integrator introduces and connects the web application to the file system service. Separating the file system service from web applications have many advantages for both web applications and users:

- Benefits to web applications: First, Web applications need not provide the file system service and its related administrative tasks, such as user registration and access control. This reduces the complexity and cost of developing and servicing web applications. Second, since web applications invoke the file system service via an API, it offloads the complexity of implementing the file system to the file system service. Third, since the web application may work with any file system service that implements the expected API, the user may freely choose a file system service that best suits his/her needs. For example, some web applications wish to be able to run in the offline mode. Direct browser support for structured client-side file system or browser plugins, such as Google Gears, allows the application to cache data locally with the browser, but requires the individual web applications to support them. Using our proposed file system service, the user can simply connect the web application to a file system service with local backend file system without requiring any modification to the web application.

- Benefits to users: Users should have the freedom to decide what happens to the data they create with applications. This existed with traditional desktop applications (modulo DRM and proprietary file formats), but was lost in the move to web applications. By default, the data is locked up on the application servers, and the application developers need to provide functionality to free the data to be shared. By separating the application provider from the file system provider, users can gain the flexibility of interchanging file system and application providers for their data. This also improves the user's ability to keep his data private. He can enforce access control at any time, independent of the application. Applications have access to the user's data only when needed. This narrows the window of vulnerability. Application providers that may have weaker security can not accidentally reveal a user's information during a breach.

2. OVERVIEW

2.1 Web Applications

From a user's perspective, a web application is any program run within a web browser. Once we look behind the scenes, we see that many different types of programs make up the category of web applications. The browser displays a presentation layer of the web application that a users sees, but the code used to create that presentation can start from different sources. There is a spectrum of web application based on their design for code execution and data processing. The execution of web application code ranges from being primarily server-based to primarily browser-based.

- *Server-based* web applications are programs that run all of their code on a server communicating with the browser over the network. The interaction with these programs usually consist of a repeated sequence of a synchronous web page retrieval and load followed by user input submission.

- *AJAX* web applications push some code execution into the browser. They ship a Javascript client component to improve the user experience by having the client perform some processing of user input and data locally and communicate with the server process without page reloads.

- *Mashup* web applications process data not only from the user but also from other web-accessible data sources by interacting directly with other web applications that are loaded in the same browser.

- *Offline capable* web applications can run when disconnected from the network for some period of time. They download all the necessary executable code and store it within the browser. When disconnected from the network, all code execution takes place within the browser. An offline application may reconnect to a server later when network access is available to perform additional processing. A completely offline web application would be able to connect to a server once to retrieve the code, and never need reconnect. This would be equivalent to downloading a program executable to run locally on the computer.

For each of these types of web applications that need to store user data, we believe that the web application can be separated from the storage of user data. A web application does not need to store user data at the same location is it executing. We already see this in the case of web applications with a server-based component that can outsource storage to a third party like Amazon S3. In this paper we propose a storage solution for web applications with browser-based components. As the browser becomes a richer platform, providing functionality equivalent to full operating systems, web applications

can take advantage of the platform by running more code on the browser.

2.2 Goals

We propose a mashable file system service for web applications. We have two design goals:

- The file system should be easy to use by application developers.

  At the most basic level, a file system provide a way for a client to store a chunk of data and retrieve that data. The file system provides a handle or name by which the client uses to designate a particular chunk of data. For ease of use, we can layer on top of the basic storage mechanism additional APIs.

  For desktop replacement web applications, we provide an API of the file system designed to mimic that of POSIX. Since host-based applications use POSIX-style API to access the file system, our API facilitates porting host-based applications to a web application architecture. Web applications requiring other data storage semantics like the object store of Amazon S3, can make use a simpler API that is a subset of the functionality provided by the POSIX-style API.

- The file system should let users own their data. A web application should not dictate where users store their data, should not be able to access the data when the user is not running the application, and should not restrict how the user uses the data. By contrast, current web applications decide where to store the user's data, have constant access to the data regardless whether users are running the applications, and restrict the data to be used only by the application running on the same server.

2.3 Web Application Design

From the user's perspective, the move away from desktop applications to web applications has centralized computing in a way similar to the earlier computing environment of mainframe computer and connected terminals. Although the user is now reliant on a bevy of web application services instead of a single system, he still relinquishes some control over his computing resources.

Cloud computing has allowed web applications to untether from a particular server and have their computation, data storage, and network bandwidth requirements serviced by third-parties. We believe that pushing user-data storage into the cloud, also to be serviced by third-parties, would give web applications and users more flexibility in their use of computing resources.

A user-centered storage system for web applications not only returns more control to the user to run the decentralized application that he was accustomed to on the desktop, but also enables new types of applications. With user-centered data storage, a single web application can have a better view of all of the a user's data. An web application like a trusted personalized search engine could crawl the user data storage and answer search queries over all the data. Previously, a user would only have been able to search over publicly published data with a web search engine, or rely on the separate web applications to individually performs the searches.

To illustrate the design of our file system service, we draw an analogy between a web-based computing environment to a host-based environment. The latter has an operating system, a file system, and applications, where the operating system connects the file system with the applications. In a web-based computing environment, we let the mashup integrator to connect the applications with the file system service, assuming the role of an operating system.

2.4 Requirements

Our proposed file system service requires that the browser provide secure cross-domain communication. More specifically, the browser can isolate contents from different domains and yet allow controlled communications between the domains. The former requirement is provided by the same origin policy, which is a standard security policy in modern browsers. The latter requirement is being fulfilled by browser cross-domain communication mechanisms.

2.5 Example

In the example in Figure 1, *editor.com* provides an application for editing photos, and *organizer.com* provides an application for organizing photos. Neither of them stores any user data. Instead, a user chooses to store his photos at *filesystem.com*. *integrator.com* provides an application that connects the two applications to the file system. When the user visits *integrator.com*, *integrator.com* creates three frames and loads the main pages from *editor.com* (containing the editor), *organizer.com* (containing the organizer), and *filesystem.com* (containing the client code for the file system) into the frames. *integrator.com* also sets up communication channels between the frame from *editor.com* and that from *filesystem.com*.

3. DESIGN

3.1 Components

To the web browser, the mashable file system can be treated as another web application. Its API is retrieved via HTTP as a Javascript library. It would be combined by a mashup integrator with other web applications needing a file system for storage. Our mashable file system service runs as a provider, which exposes a file system API to a web application running as another provider. As is typical for mashup providers, the file system service has a server and client component.

3.1.1 Server component

The server component provides persistent data storage and exposes an API via a file system access protocol for the client. Since the client is JavaScript code running in an unmodified browser, this API must be accessible by JavaScript in browsers. Since a file system service deploys both its server and client components, the API between these components need not be standardized, because interoperability between server and client components from different file system services in unnecessary. This gives the developers the freedom to optimize their API.

Since web applications interact only with the client component of the file system service, the location of the file system server is transparent to web applications. The server could run either remotely or locally (on the same machine as the browser). When the server is on the network, the access protocol must be over HTTP since the browser restricts the client component, which is in JavaScript, to use HTTP. When browsers provide data storage capabilities, such as the *localStorage* object of HTML5, they also can function as file system servers. In this case, the client component of our file system service can use the API provided by the browser to access the local data storage. Note that no browser modification is necessary in this case. Since web applications are being deployed with offline capabilities, providing file system services both remotely and

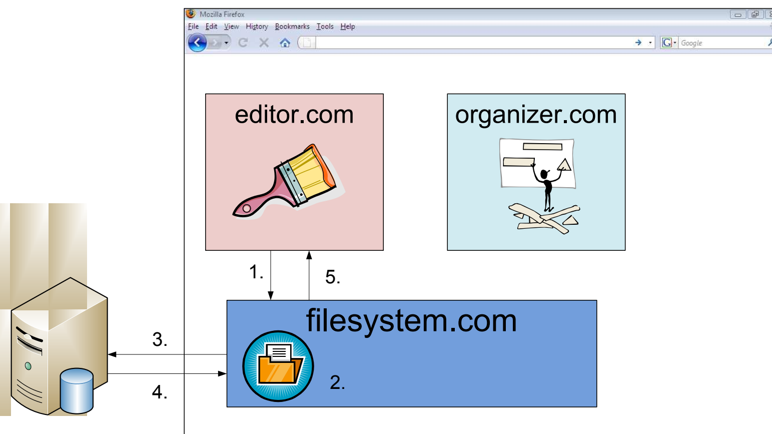

Figure 1: Example of file system service used by two web applications. Step 1: The application from *editor.com* requests to open a file. Step 2: The file system client component displays a file browser where the user chooses an existing or new file name. Step 3: The file system client component requests the server component to open the file. Step 4: The file system server component opens the file, obtains a capability for this file, and returns the capability to the client component. Step 5: The file system client component returns the capability to the application. Thereafter, the application can access the file via the capability.

locally is desirable. Compared to the current approach where web application developers have to handle offline mode explicitly, the file system API can provide offline storage transparently to web applications.

3.1.2 Client component

For desktop-replacement web applications, the client component of our file system service provides similar functions to that of a networked file system. It exposes an API to web applications and communicates with the server component of the file system service. We design the API between the client component and web applications to be similar to the POSIX file API to facilitate porting host-based applications to the web. A basic client can simply translate between the POSIX-like JavaScript API (between the client component and web applications) and a remote file system protocol (between the client and server components). Advanced clients may provide file replication, caching, and online/offline access.

Unlike a traditional client component of a networked file system, our client component provides an interface to allow the user to authorize access to his files. The user interface provides a file browser for the user to delegate access to his files. When a web application requests to create or open a user file (not application configuration files, see Section 3.2), the client component of the file system service opens the file browser, which allows the user to select an existing file or name a new file.

We could place the file browser in the mashup integrator or the web application, but both these options are inferior to our choice of placing the file browser in the file system client component. If we place the file browser in the integrator, each file open request from the web application would have to go to the integrator first before it reaches the client component. This could cause a performance penalty. More importantly, this design would have granted the integrator all the privileges that the authenticated user has on the file system: because the user selects files in the UI provided the integrator, the client component, having no knowledge of the user's action, would have to grant access to all the user's files. By contrast, since we chose to place the file browser in the file system client component, we need to provide no file system privilege to the integrator. Since the user interacts directly with client component, the client component will not authorize any file access without the user's consent. By the same reasoning, our design choice is superior to placing the file browser in the web application.

3.2 Access control

The file system service regulates access by a capability system. After the browser loads the client component of the file system service, the client component asks the user to log in. The integrator connects API calls from the web applications to the API functions in the client component of the file system service. When a web application requests a user file, the client component of the file system service displays a file browser where the user selects a file or creates a new file. Then, the file system creates a capability for this file and sends the capability to the web application. The application can subsequently access the file using this capability.

We choose capabilities over access control lists for access control because capabilities mesh well with the decentralized nature of web applications. We expect the user to find new web applications all the time and to want to share file access with new friends. Maintaining an access control list at the file service would be unwieldy and not scale. With capabilities, we empower the user to delegate access as he sees fit.

The designer needs to decide on the granularity of the capabilities. Does a capability allow access to only one file or a set of files? If a capability refers to a directory, does the capability allow access to the files in all the subdirectories? The file browser displayed by the client component could include check boxes to allow the user to make these choices.

Our access control scheme so far requires the user to approve each file access request (unless the user chooses a directory and allows access to all the files in the directory). This is appropriate for user files. User files are created by applications for users and can be potentially shared with other applications. Examples include documents, images, address books, and bookmarks. Since users

manage and manipulate their files directly (e.g., backing up, deleting, or emailing their documents), it is reasonable to expect users to make intelligent access control decisions on these files. On the other hand, applications also create and access application-specific data, such as configuration and temporary files. Since users do not typically manage or manipulate these files, we cannot expect users to make intelligent access control decisions on these files. Fortunately, since these files rarely need to be shared with other applications, we can simply create a directory for each application, and returns a capability that allows the application full access rights to this directory. To allow the application to access this directory across sessions, we could name the directory by the application's domain name or a hash of its public key and enforce access control based on this.

In this scheme, granting access to other users would be similar to granting access to a web application. When a users wishes to share a file with others, he simply needs to share the capability. While our design does not directly recognize the other users, this task can be delegated to a file sharing web application. The user first grants access to a file he wants to share to the file sharing web application. The file sharing web application would then pass on the capability it was granted to the recipients' web application that eventually accesses the file.

The longevity of these capabilities can also be user controlled. A user may explicitly expire a capability. This would usually occur when the user is done using the web application requesting the file and the web application would then signal the file system service to close the file. The user or file system service may also expire a capability based on time, access count, or other parameters. This form of expiration may be useful for capabilities given to a web application that may have crashed without properly closing opened files, or for capabilities that may need to persist beyond a web application session, such as for the file sharing web application previously mentioned.

3.3 Usage example

Continuing from the example in Section 2.5 where integrator.com loads three frames from two web application (*editor.com* and *organizer.com*) and one file system service (*filesystem.com*). The user creates a new picture in the application from *editor.com*. When the user saves the file, the application calls `open()` in the client component from *filesystem.com* (Step 1 in Figure 1). The client component displays a file browser where the user chooses an existing or new file name (Step 2 in Figure 1). Then, the client component makes a request of the server to open the file, obtains a capability for this file, and returns the capability to the application (Step 3, 4, 5 in Figure 1). The application then uses this capability to write to the file. When the user wishes to add this file to the photo organizer from *organizer.com*, the above process repeats.

4. IMPLEMENTATION

We developed a proof-of-concept file system server and client. The two components work together to provide data storage services to browser-based web applications. The client, written in Javascript, provides the file system API for other Javascript applications. It primarily translates and redirects storage requests from the web application to the file system server. We chose a simple JSON-RPC protocol over the HTTP transport provided by the browser. The server is also a software layer processing the network requests into actual storage of the data.

4.1 Authentication

The user first authenticates to the file system service. The mechanism for authentication is not important to our design. When a web application wishes to use the file system, the integrator must introduce the file system to the client. The integrator provides the filesystem with the identity of the application desiring services and provides the application a handle to directly make file system calls. We trust the integrator to identify the applications correctly; we have no recourse if the integrator is malicious. The file system service needs to know the identity of the application to provide access to application-specific storage.

4.2 Storage API

Once the application has a handle to make calls into the file system client, it can use the full API provided to store data on the file system. Currently, we provide a basic POSIX-like API, with file operations like *open()*, *read()*, *write()*, *truncate()*, etc. The POSIX API provides a capability-like system when issuing file descriptors for file access.

The file system client intervenes on *open()* calls for user data with the file system user interface. The iterface prompts the user to grant the application access to one or more user data files. The user additionally has the option to specify the type of access granted (read-write/read-only). The handles returned by the *open()* calls are opaque identifiers. These opaque identifiers are essentially capabilities for access to the file. Web applications wishing to share files can simple exchange the file handles once they are created. These capabilities grant access as long as no application issues a *close()* for the handle. For other API calls, the file system client serves as a simple relay of these API calls to the file system server.

5. SECURITY ANALYSIS

5.1 Threat model

We assume the trustworthiness of the following entities:

- The mashup integrator, which connects the file system service to the web application. It truthfully conveys the identity of the web application to the user and the file system service, and it does not violate the confidentiality and integrity of the file system.
- The file system. It can authenticate its users securely.
- The browser. The browser isolates contents from different domains but allows secure cross-domain communication.

5.2 Security benefits

Compared to current practice of storing user data on the same server as the web application, our file system service has the following security advantages:

- It reduces the risk of user data compromise in the case of an application server compromise. Using our file system service, a web application has no access to user data when the user is not running the application. Therefore, when an attacker breaks into a web application server, he can compromise only the data of the users who are running the application. As another benefit, it relieves the web application from complying with regulatory mandates for user data security because the application server stores no user data.

- The use of capabilities for file access protects the user authentication credentials at the file system service even when an application or integrator server is compromised, because the credentials never leave the file system service. The capabilities also restrict the web application to only the files that the capabilities allow.

5.3 Potential attacks

5.3.1 Network MITM

We prevent network Man-In-The-Middle attacks with SSL/TLS for connections to integrator and file system. If we do not trust the user not to bypass SSL/TLS by ignoring warnings, the applications themselves can use a Diffie-Hellman key exchange to secure their communication. The user does need a secure connection to the integrator to bootstrap this process.

5.3.2 Malicious web applications

A malicious web application could attack the application specific storage or the user data. With quotas, we can limit resource exhaustion attacks on application specific storage. The application needs to be handed a capability to have access user data. This requires the user to consciously give that access to the application. In order to accomplish this, the malicious application must misrepresent itself to both the integrator and to the user when he is granting the capability. Any damage is restricted only to the file for which the capability is assigned.

6. RELATED WORK

6.1 Networked file systems

Networked file systems allow for distributed storage and access of data. The Sun Network File System (NFS) [15, 16], originally designed for sharing among a small set of mutually-trusting workstations, developed over several versions to now support access over wide-area networks with strong security mechanisms for end to end mutual authentication and integrity. Other network file systems like the Andrew File system [10] and CIFS [3] developed in a similar fashion and also now scale to service large distributed networks of computers.

As the web grew to become one of the most pervasive computing platforms, network file systems were created or adapted to fulfill the needs of this new area. WebFS [18] provides a global file system over HTTP with support for functionality needed by many distributed Internet applications. WebDAV [5] focuses on web authoring specifically and extends HTTP to a read-write platform for web clients. WebDAVA [12] provides for file sharing over HTTP with flexible access controls using user issued access credentials. WebNFS [2] adapts the NFS protocol for the web, by extending the semantics of the NFS protocol to support web browser clients by creating a lightweight binding mechanism. Even though these protocols target web browsers as clients, they are not fully supported by current web browsers. They require additional software to enable the browser to communicate with the file system servers. Web applications running in the browser do not interface with these network file systems in through the browser.

The web browser itself is changing to support the more complex AJAX web applications with new data storage requirements. The HTML5 specification [11] proposes structured client-side storage that is being adopted by some browsers and projects such as Google Gears [8] provide similar mechanisms as a browser add-on. They enable web applications to store data locally as name/value pairs or directly into a database. Keeping data into these local browser containers ties the data to a single instance of the browser. Each web application needs to handle its own data synchronization individually among the different browser instances of a user. Data sharing among web applications is possible, but is still limited by the same origin policy. By contrast, our system provides for arbitrary data file sharing governed by the user.

6.2 Web application storage

With cloud computing, web applications can run without being installed to a particular server, by renting on-demand computing resources from a third-party and run “in the cloud”. Services like Amazon S3 [1], provide web applications with network accessible storage. Such cloud storage systems have rudimentary access control settings, since the cloud storage systems are primarily designed provide the storage service to a single web application. Cloudviews [7] proposes a system to construct views over datasets on a cloud storage service. Web applications define these database-style views of their data and can then selectively share the view with other web applications. Menagerie [6] provides a virtual file system composed of data from heterogeneous web applications by providing an interface for a web application to export their data into a namespace and a file system interface that combines these namespaces. A user can then mount this virtual file system on his local computer and manipulate it with standard file system commands. We take a user-centric approach where data is stored separately from the application and provide web applications with interfaces to the storage. The user then also retains control over the sharing of his data.

6.3 Capability systems

Capability systems are an access control mechanism that associate an unforgable object identifier with set of rights. They permit simple and transparent sharing of those rights. A holders of the capability only needs to transmit the identifier to another grant access. They work well as authorization mechanisms for distributed systems where access control may be decentralized. OAuth [13] and delegation permits [9] provide mechanisms to grant and transfer arbitrary authorizations among web applications. Our file system could be extended to use one of these systems for file access control, however we maintain our own access control system for simplicity. CapaFS [14] encodes capabilities into the names of files on a file system to allow a user to share files with dynamic groups of other users. In our file system, we provide a similar mechanism for granting access to different web applications as well as users. The design of our file management user interface is inspired by CapDesk [17] where the user directly grants capabilities for file access to an application via a file browser.

7. CONCLUSION

We have proposed a design for a web based file system to provide a storage service to web applications. Such a file system would free web applications from developing and administering their own storage mechanisms. The web applications can instead rely on a POSIX-like standard file system interface, much like host-based applications. Our file system provides another component for a web-based computing environment that will simplify application development and provide greater ease of use to end users.

Users of web applications that rely on this file system gain more control over their data since their files are independently managed from the web applications. Our file capability system allows users

to grant new web applications access to existing file and revoke access at any time. By providing the familiar file metaphor for a user's data, we can make it easier for users to understand when data sharing takes place and for users to control the access of their data.

8. REFERENCES

[1] Amazon Simple Storage Service (S3). `https://s3.amazonaws.com/`.

[2] B. Callaghan. WebNFS Client Specification. RFC 2054 (Informational), Oct. 1996.

[3] Common internet file system (cifs) technical reference. `http://www.snia.org/tech_activities/CIFS/CIFS-TR-1p00_FINAL.pdf`.

[4] S. Crites, F. Hsu, and H. Chen. OMash: enabling secure web mashups via object abstractions. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 99–108, New York, NY, USA, 2008. ACM.

[5] L. Dusseault. HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV). RFC 4918 (Proposed Standard), June 2007.

[6] R. Geambasu, C. Cheung, A. Moshchuk, S. D. Gribble, and H. M. Levy. Organizing and sharing distributed personal web-service data. In *Proceeding of the 17th international conference on World Wide Web*, pages 755–764, Beijing, China, 2008. ACM.

[7] R. Geambasu, S. D. Gribble, and H. M. Levy. CloudViews: communal data sharing in public clouds. In *HotCloud '09 Workshop on Hot Topics in Cloud Computing*, 2009.

[8] Google gears. `http://gears.google.com/`.

[9] R. Hasan, M. Winslett, R. Conlan, B. Slesinsky, and N. Ramani. Please permit me: Stateless delegated authorization in mashups. *Computer Security Applications Conference, Annual*, 0:173–182, 2008.

[10] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1):51–81, 1988.

[11] HTML 5 draft recommendation. `http://www.whatwg.org/specs/web-apps/current-work/`.

[12] A. Levine, V. Prevelakis, J. Ioannidis, S. Ioannidis, and A. D. Keromytis. Webdava: An administrator-free approach to web file-sharing. In *WETICE '03: Proceedings of the Twelfth International Workshop on Enabling Technologies*, page 59, Washington, DC, USA, 2003. IEEE Computer Society.

[13] OAuth core 1.0. `http://oauth.net/core/1.0/`.

[14] J. T. Regan and C. D. Jensen. Capability file names: separating authorisation from user management in an internet file system. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, pages 17–17, Berkeley, CA, USA, 2001. USENIX Association.

[15] R. Sandberg, D. Goldberg, S. Kleinman, D. Walsh, and B. Lyon. Design and implementation of the sun network filesystem. In *Proceedings of the Summer 1986 USENIX Conference*. USENIX Association, 1985.

[16] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. Network File System (NFS) version 4 Protocol. RFC 3530 (Proposed Standard), Apr. 2003.

[17] M. Stiegler and M. S. Miller. E and capdesk: Pola for the distributed desktop. `http://www.combex.com/tech/edesk.html`.

[18] A. M. Vahdat, P. C. Eastham, and T. E. Anderson. Webfs: A global cache coherent file system. Technical report, UC Berkeley, 1996.

[19] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and Communication Abstractions for Web Browsers in MashupOS. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP 2007)*, pages 1–16, New York, NY, USA, October 2007. ACM.