Separating Web Applications from User Data Storage with BSTORE

Ramesh Chandra, Priya Gupta, and Nickolai Zeldovich
MIT CSAIL

ABSTRACT

This paper presents BSTORE, a framework that allows developers to separate their web application code from user data storage. With BSTORE, *storage providers* implement a standard file system API, and *applications* access user data through that same API without having to worry about where the data might be stored. A *file system manager* allows the user and applications to combine multiple file systems into a single namespace, and to control what data each application can access. One key idea in BSTORE's design is the use of *tags* on files, which allows applications both to organize data in different ways, and to delegate fine-grained access to other applications. We have implemented a prototype of BSTORE in Javascript that runs in unmodified Firefox and Chrome browsers. We also implemented three file systems and ported three different applications to BSTORE. Our prototype incurs an acceptable performance overhead of less than 5% on a 10Mbps network connection, and porting existing client-side applications to BSTORE required small amounts of source code changes.

1 INTRODUCTION

Today's web applications have *application-centric* data storage: each application stores all of its data on servers provided by that application's developer. For example, Google Spreadsheets [14] stores all of its documents on Google's servers, and Flickr [37] stores all of its photos at `flickr.com`. Coupling applications with their storage provides a number of benefits. First, users need not setup or manage storage for each application, which makes it easy to start using new apps. Second, each application's data is isolated from others, which prevents a malicious application from accessing the data of other applications. Finally, users can access their data from any computer, and collaborate with others on the same document, such as with Google Spreadsheets. Indeed, application-centric data storage is a natural fit for applications that perform server-side data processing, so that both client-side and server-side code can easily access the same data.

Although the application-centric data model works well in many cases, it also has a number of drawbacks. First, users are locked into using a single application for accessing any given piece of data—it is difficult to access the same data from multiple applications, or to migrate data from one application to another. By contrast, in the desktop environment, `vi` and `grep` have no problems accessing the same files. Second, application developers are forced to provide storage or server resources even if they just want to publish code for a new application, and even if users' data is already stored in other applications. By contrast, in the desktop environment, `http://gnu.org/grep` might only distribute the code for `grep`, and would not maintain servers to service users' `grep` requests. The makes it difficult for application developers to build client-side web applications.

To address this problem, we present BSTORE, a system that allows web application developers to decouple data storage from application code. In BSTORE, data can be stored in *file systems*, which provide a common interface to store and retrieve user data. File systems can be implemented by online services like Amazon S3 [4], so that the data can be accessed from any browser, or by local storage in the browser [12, 36], if stronger privacy and performance are desired. Multiple file systems are combined into a single namespace by the *file system manager*, much like the way different file systems in Unix are mounted into a single namespace. Finally, applications access data in BSTORE through the file system manager, without worrying about how or where the data is stored.

One challenge facing BSTORE is in providing security in the face of potentially malicious applications or file systems. While the application-centric model made it impossible for one application to access another application's data by design, BSTORE must control how each application accesses the user's shared data. Moreover, different users may place varying amounts of trust in file systems: while one user may be happy to store all of their data in Amazon S3, another user may want to encrypt any financial data stored with Amazon, and yet another user may want his grade spreadsheets to be stored only on the university's servers.

A second challenge lies in designing a single BSTORE interface that can be used by all applications and file systems. To start with, BSTORE's data storage interface must be flexible enough to support a wide range of application data access patterns. Equally important, however, is that any *management* interfaces provided by BSTORE be accessible to all applications. For example, any application should be able to specify its own access rights delegation or to mount its own file systems. If our design were to allow only the user to specify rights delegation, applications might be tempted to use their own file system manager when they find the need to specify finer-grained

access control policies or mount application-specific file systems. This would fracture the user's file namespace and preclude data sharing between applications.

A final challenge for BSTORE is to support existing web browsers without requiring users to install new plug-ins or browser extensions. While a browser plug-in could provide arbitrary new security models, we want to allow users and application developers to start using BSTORE incrementally, without requiring that all users switch to a new browser or plug-in simultaneously.

BSTORE's design addresses these challenges using three main ideas. First, BSTORE presents a unified file system namespace to applications. Applications can mount a new file system by simply supplying the file system's URL. A file system can either implement its own backend storage server or can use another BSTORE file system for its storage. Second, BSTORE allows applications to associate free-form *tags* with any files, even ones they might not have write access to. Using this single underlying mechanism, BSTORE enables an application to organize files as it chooses to, and to delegate access rights to other applications. Finally, BSTORE uses *URL origins* as principals, which are then used to partition the tag namespace, and specify rights delegation for files with different tags.

To illustrate how BSTORE would be used in practice, we ported a number of Javascript applications to BSTORE, including a photo editor, a `vi` clone, and a spreadsheet application. All of the applications required minimal amount of code changes to store and access data through BSTORE. We also implemented several BSTORE file systems, including an encrypting file system and a checkpointing file system. Our prototype of BSTORE incurs some in-browser processing overheads, but achieves overall performance comparable to using `XMLHttpRequest` directly for a typical home network connection.

The rest of this paper is organized as follows. Section 2 provides motivation and use cases for BSTORE. Section 3 details BSTORE's design, and Section 4 discusses our prototype implementation. Section 5 describes our experience using BSTORE in porting existing Javascript applications and in implementing file systems. Section 6 evaluates BSTORE's performance overheads, and Section 7 discusses some of BSTORE's limitations. Related work is discussed in Section 8 and finally Section 9 concludes.

2 MOTIVATING EXAMPLES

BSTORE can benefit the users and developers of a wide range of web applications, by giving users control over their data, by making it easier for applications to share data, and by removing the need for application developers to provide their own storage servers. An existing web application that has its own storage servers can also use BSTORE to either export its data to other applications, or to access additional data that the user might have stored elsewhere. The rest of this section describes a few use cases of BSTORE in more detail.

User wants control over data storage. In the current web application model, application developers are in full control of how user data is stored. Unfortunately, even well-known companies like Google and Amazon have had outages lasting several days [2, 23, 30], and smaller-scale sites like the Ma.gnolia social bookmarking site have lost user data altogether [21]. To make matters worse, the current model does not allow users to prevent such problems from re-occuring in the future. Some sites provide specialized backup interfaces, such as Google's GData [13], but using them requires a separate backup tool for each site, and even then, the user would still be unable to access their backed-up data through the application while the application's storage servers were down.

With BSTORE, users have a choice of storage providers, so that they can store their data with a reliable provider like Amazon or Google, even if they are using a small-scale application like Ma.gnolia. If the user is concerned that Amazon or Google might be unavailable, they can set up a *mirroring* file system that keeps a copy of their data on the local machine, or on another storage service, so that even if one service goes down, the data would still be available.

Finally, users might be concerned about how a financial application like Mint.com stores their financial data on its servers, and what would happen if those servers were compromised. Using BSTORE, users can ensure that their financial data is encrypted by mounting an encrypting file system, so that all data is encrypted before being stored on a server, be it Mint.com or some other storage provider. In this model the user's data would still be vulnerable if the Mint.com application were itself malicious, but the user would be protected from the storage servers being compromised by an attacker.

User accesses photos from multiple applications. There is a wide range of web applications that provide photo editing [9], manipulation [26], sharing [15], viewing [32], and printing [31]. Unfortunately, because of the application-centric storage model, users must usually maintain separate copies of photos with each of these applications, manually propagate updates from one application to another, and re-upload their entire photo collection when they want to try out a new application. While some cross-application communication support is available today through OAuth [25], it requires both applications to be aware of each other, thus greatly limiting the choice of applications for the user.

With BSTORE, all photo applications can easily access each other's files in the user's file system namespace. The

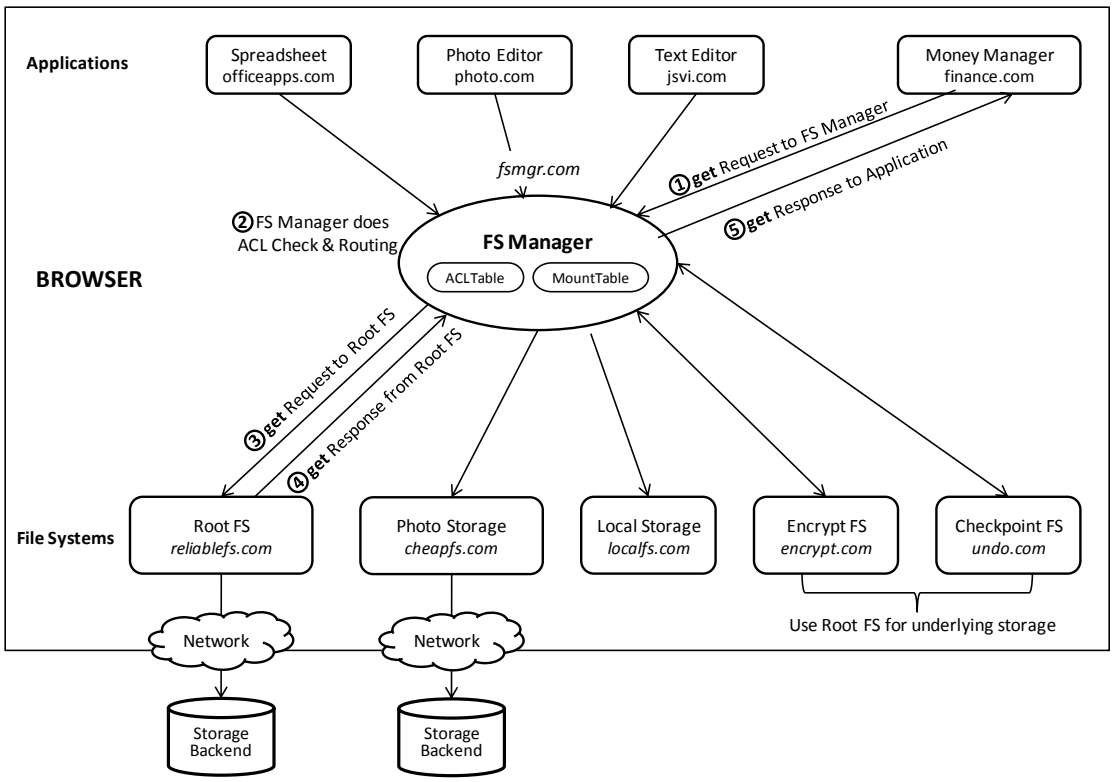

Figure 1: Overview of the BSTORE architecture. Each component in the browser corresponds to a separate window whose web page is running Javascript code. All requests to the BSTORE file system are mediated by the FS manager. Arrows (except in the **get** response flow) indicate possible direction of requests in the system.

user can also specify fine-grained delegation rules, such as granting Picasa access to lower-resolution photos for publishing on the web, without granting access to full-resolution originals. Web sharing applications such as Picasa could still store separate copies of photos on their servers (for example, for sharing on the web); this could be done either through an application-specific interface, as today, or by mounting that user's Picasa file system in BSTORE's file system manager.

Small-scale developer builds a popular application. In addition to common web applications like GMail and Picasa, there are a large number of niche web applications written by developers that might not have all of Google's resources to host every user's data. For example, MIT's Haystack group has written a number of client-side web applications that are popular with specific groups of users. One such application is NB [22], which allows users to write notes about PDF documents they are viewing. Currently, NB must store everyone's annotations on MIT's servers, which is a poor design both from a scalability and security perspective. BSTORE would allow individual users to provide storage for their own annotations. Many other small-scale web application developers are facing similar problems in having to provision significant server-side storage resources for hosting essentially client-side web applications, and BSTORE could help them as well.

3 DESIGN

The BSTORE design is based on the following goals:

Independence between applications and storage providers. In an ideal design, applications should be able to use any storage providers that the user might have access to, and the user should be able to manage their data storage providers independent of the applications. In particular, this would enable a pure Javascript application to store data without requiring that application's developer to provide server-side storage resources.

Egalitarian design. BSTORE should allow any application to make full use of the BSTORE API, and avoid reserving any special functionality for the user. One example is access control mechanisms: we would like to allow applications to be able to subdivide their privileges and delegate rights to other applications. Another example is mounting new file systems into the user's namespace: any application should be able to mount its own server-side resources into BSTORE, or to encrypt its data by mounting an encrypting file system.

No browser modifications. BSTORE should not require browser modifications and should work with existing browser protection mechanisms (same origin policy or SOP).

3.1 Overview

Figure 1 illustrates BSTORE's architecture. A user's BSTORE is managed by a trusted file system manager (FS manager). It mediates all file system requests in the system. Applications (shown at the top of the figure), send their file system requests to the FS manager, which routes them to the appropriate file system (shown at the bottom of the figure). The FS manager also performs access control during this request routing process.

The Javascript code for the FS manager, applications, and file systems run in separate protection domains (browser windows) and are isolated from each other by the browser's same origin policy. All communication happens via `postMessage`, which is the browser's cross-domain communication mechanism.

The FS manager allows users and applications to mount different file systems and stitch them together into a unified namespace. A BSTORE file system exports a flat namespace of files, and does not have directories. Files do not have a single user-defined name; instead each file has a set of tags. An application can categorize a file as it sees fit by setting the appropriate tags on that file. At a later time, it can search the file system and recall files that match a specific tag expression. Using tags in this manner for file system organization allows each application the flexibility to organize the file system as it chooses.

File tags are also the basis for access control in BSTORE. An application uses tags to delegate a subset of its access rights to another application. The FS manager keeps track of these delegations in the ACL table and enforces access control on every file system request.

3.2 BSTORE API

Table 1 shows the API calls exported by the FS manager to each application, and Table 2 shows the API calls exported by each BSTORE file system to the FS manager.

The API calls in Table 1 are divided into two categories, shown separated by a line in the table. The calls in the top category are file system calls that correspond directly to API calls in Table 2. The calls in the lower category are FS manager-only calls related to access rights delegation and creating mountpoints. Among the file system calls, **create** and **search** operate on a file system handle (*fs_h*), and the rest of the calls operate on a file handle (*fh*). The **get** and **set** calls operate on entire objects; there are no partial reads and writes. This design choice is consistent with the behavior of majority of web applications, which read and write entire objects, and with other web storage APIs, such as Amazon S3 and HTML5 local storage.

FS manager API call	Return on success	Rights needed
create(fs_h, init_tags)	fh, ver	write to creator tag
search(fs_h, tag_expr)	fh_list	read matching files
set(fh, data, [match_ver])	ver	write on file
get(fh, [match_ver])	ver, data	read on file
stat(fh)	ver, size	read on file
delete(fh, [match_ver])	—	write on file
settag(fh, tag)	ver	read on file
gettag(fh)	ver, tag_list	read on file
rmtag(fh, tag)	ver	read on file
setacl(target_principal, tags, perms)	—	—
getacl([target_principal])	delegation_list	—
rmacl(target_principal, tags, perms)	—	—
encrypt(plaintext)	ciphertext	—

Table 1: API exported by the FS manager to each BSTORE application. The rights column shows the rights needed by an application to perform each API call.

FS API call	Return on success
create(init_tags, acltagset)	fh, ver
search(tag_expr, acltagset)	fh_list
set(fh, data, acltagset, [match_ver])	ver
get(fh, acltagset, [match_ver])	ver, data
stat(fh, acltagset)	ver, size
delete(fh, acltagset, [match_ver])	—
settag(fh, tag, acltagset)	ver
gettag(fh, acltagset)	ver, tag_list
rmtag(fh, tag, acltagset)	ver

Table 2: API exported by each BSTORE file system to the FS manager. The FS manager fills in the *acltagset* based on the requesting application's access rights.

The rights column in Table 1 shows the access rights needed by an application to perform each API call. When an application makes a file system call, the FS manager uses the access rights delegated to the application to fill in the *acltagset* shown in API calls in Table 2. The file system uses the *acltagset* to perform access control, as will be described in more detail in Section 3.5.

Current browsers use the URL domain of a web application's origin as the principal while applying same origin policy to enforce protection between browser windows. Since one of our goals is to not modify browsers, we also choose URL domains of an application's or file server's origin as principals in BSTORE.

3.3 Tags

A tag is an arbitrary label that a principal can attach to a file (e.g., *low-res* on a photo file). A file can have multiple tags set by different principals. Tagging is the underlying mechanism using which an application (or user) can both organize the BSTORE namespace in ways meaningful to it, as well as to delegate access rights to other applications.

Applications use **settag** to tag a file. An application can tag other applications' files as long as it can read those files. This allows an application to categorize files in BSTORE in a manner that best fits its purpose. For example, a photo editor application can tag a photo file with the date of the photo, location at which it was taken, or the names of people in it. To avoid applications clobbering each others tags, each principal gets its own tag

namespace, and can only set tags within that namespace. So, a *low-res* tag set by Flickr is actually *flickr.com#low-res*, and is different from *low-res* tag set by Google Picasa (which is *picasa.com#low-res*).

In BSTORE, tag-based search is the only mechanism by which applications can lookup files and get handles on them. **search** takes a file system handle and a tag query expression and returns files in that file system whose list of tags satisfy the tag query, and which are readable by the requesting application. The tag query expression supports simple wildcard matches. **settag** and **search** together allow an application to organize the BSTORE namespace, and recall data in ways meaningful to it.

Applications use **gettag** to retrieve a list of all tags for a file, including those set by other applications. **rmtag** allows an application to delete only the tags that were set by the same principal as the application.

3.4 File systems

Every mounted BSTORE file system has a Javascript component running in a browser window. This Javascript component exports the BSTORE file system API to the FS manager, and services requests from it. Some file systems store their data on network storage or on local storage (e.g., *reliablefs.com* and *localfs.com* in Figure 1). Others, called *layered file systems*, store their data on existing BSTORE file systems and layer additional functionality on them (e.g. *encrypt.com* encrypting data and storing it on *reliablefs.com*).

All file systems (except the root file system) are specified in mountpoint files. A mountpoint file for a file system contains information to mount the file system, in a well known format. This information includes the URL for the file system Javascript code, file system configuration parameters, and mountpoint tags. Just as it creates other files, an application uses the **create** call to create a mountpoint file with the right configuration information in the right format. Other applications can use the mountpoint file to access the file system referenced by it, as long as the access control policy allows it. Since mountpoint information can contain sensitive data (such as credentials and mountpoint tags), the application uses the FS manager's **encrypt** API call to encrypt the information and stores the encrypted information in the mountpoint file. The FS manager encrypts data using a key that is initialized when it is first set up, as described in Section 3.8.

To access a file system, an application passes a handle to the file system's mountpoint file to **create** and **search** API calls. If the file system is not already mounted, the FS manager uses this handle to read the mountpoint file, decrypt the mountpoint information, and launch the file system in a separate browser window using this information. Once the file system is initialized, the FS manager adds the file system to its mount table and routes requests to it.

Storing the mountpoint information encrypted allows an application to safely give other applications access to a file system, without giving away credentials to the file system. An application can also safely give other applications a copy of the mountpoint files it created, which can be used for mounting without leaking any sensitive information.

A user or application that created a mountpoint file may want to tag all files in that file system with a particular tag to enforce access control on all files in that file system. It is cumbersome to tag each file individually with that tag at mount time. Instead, BSTORE allows mounting file systems with *mountpoint tags* that are logically applied to every file on that file system.

3.5 Access control

A BSTORE principal obtains all its access rights through delegation from other principals. As the base case, the file system's principal is assumed to have full access to all files on that file system. A principal A can specify a *delegation rule* to delegate a subset of its access rights to another principal B. Delegation is transitive and principal B can in turn delegate a subset of rights it obtained from A to another principal C. Given all the delegation rules in the system, BSTORE computes the access rights that are transitively granted by any principal A to another principal B, by computing the transitive closure of the delegation rules. The access rights for an application with principal A to a file on principal F's file system are the rights that are transitively delegated by F to A.

Delegation rules use file tags to specify the files to which they apply. Since tags are application-defined properties on files, this allows an application (say, Picasa) to easily express intentions such as "Flickr can read all my low resolution photos on any file system," without having to search through the entire file system for low resolution photos and adding Flickr to each file's access control list.

The delegation rules described above are decentralized, and each file system can independently delegate access, maintain delegation rules from other principals, and compute transitive delegations itself. However, in practice, it is convenient for users to specify their access control policy in a single place. To achieve this, all delegation rules are centralized in the FS manager. File systems also follow a convention that, when mounted, they delegate all access to the FS manager. This allows the FS manager to make access control decisions for that file system (by using delegation from the FS manager's principal to specific applications). In case a file system does not want to use the FS manager's centralized ACL manager (e.g. in the case of an application mounting its own

application-specific file system), it can choose not to follow this convention.

The FS manager maintains an ACL table of all delegation rules. A delegation rule is of the form $\langle A, B, T, R \rangle$, and signifies that the rights R (read or read_write) are delegated from principal A to principal B for every file that is tagged with all the tags in tag set T. Going back to the example above, Picasa's intentions translate to a delegation rule $\langle$*picasa.com*, *flickr.com*, *picasa.com#low-res*, *read*$\rangle$. Any principal (application or file system) can invoke **setacl** and **rmacl** API calls on the FS manager to add and remove delegation rules. Both these calls take B, T, and R as arguments. A principal can use the **getacl** call to retrieve its delegations to a particular target principal. It can obtain all its delegations to any principal by omitting the target principal parameter to **getacl**.

The FS manager computes and maintains the transitive closure of the delegation rules added by all the principals. On a request from application A to file system F, the FS manager looks up the transitive closure and includes all the rights transitively delegated by F to A in the request before sending it to F (these rights are indicated by argument *acltagset* in Table 2). The FS manager does not fetch tags from F and do the rights check itself to avoid an extra round trip. The file system F checks the rights before performing the operation.

BSTORE's access control rules allow one principal to delegate access based on tags from another principal. In our example, if the user's low-resolution photos are already tagged *kodak.com#low-res*, Picasa can delegate access to *flickr.com* based on that tag, instead of tagging files on its own. However, this opens up the possibility of *kodak.com* colluding with flickr: if kodak tags all files with this tag, flickr will be able to access any file accessible to Picasa. The FS manager makes this trust relation explicit by verifying, on each ACL check, that the principals in a delegation rule's tags are also allowed to perform the operation in question. Thus, in our example, Picasa's delegation rule would not apply to files tagged *kodak.com#low-res* that *kodak.com* was not able to access itself.

3.6 File creation

An application (say *app.com*) creates a file using the **create** call, by specifying a handle to the target file system (say on *fs.com*). The application also specifies a list of initial tags to set on the new file. In addition, the FS manager sets a special creator tag (*fsmgr.com#app.com_creator*) on the file.

For create to succeed, *app.com* should have write access to the tag set consisting of the mountpoint tags of the file system *fs.com* and the creator tag. The creator tag is also used to delegate all access rights on that file to the creator application. This is done by the FS manager adding a delegation rule $\langle$*fsmgr.com, app.com, fsmgr.com#app.com_creator, read_write*$\rangle$.

3.7 File versions

Since BSTORE is shared storage, multiple applications could be concurrently accessing a file. To help applications detect concurrent modifications, the BSTORE API provides compare-and-swap like functionality. All files are versioned. Mutating file system calls take an optional *match_version* argument and fail with a `EMODIFIED` error if the current file version does not equal *match_version*. A file's version number is automatically incremented on every **set** operation, and most API calls also return the current file version. Versioning proves useful in other scenarios as well: it can be used to keep track of change history, as in the case of the checkpointing file system described in Section 5.2.2, and to support transparent local caching of network storage.

3.8 Bootstrapping

We now give a brief overview of how a user interacts with BSTORE, including initial setup of the different components and subsequent use.

FS manager. We imagine that users will be able to choose from multiple implementations of FS manager from different vendors, depending on whom they trust and whose UI they prefer. Once a user chooses a FS manager, she sets up her browser by browsing to the FS manager and configuring it with the information to mount the root file system. The FS manager stores this mount information in the browser's local storage and uses it for subsequent sessions. It also mounts the root file system, and sets up BSTORE by initializing its encryption key and delegation rules table. The user needs to repeat the browser setup step for every browser she uses. The BSTORE setup only happens once. During normal use, the FS manager also provides the user with a UI to create and manage delegation rules.

Applications. For each application, the user adds delegation rules to the FS manager to grant the application access to specific files in BSTORE. The user initializes the application with the FS manager URL, which it uses to launch the FS manager if it is not already launched. The application also stores the FS manager URL in its network storage or local storage for subsequent sessions.

During normal use, when the user needs to choose a file, applications present a file chooser interface to the user. The file chooser allows her to search specific file systems for files with specific tags, and to choose a file from the results.

File systems. The user obtains an account for file systems that store data on the network. File systems provide a friendly user interface using which the user can create a mountpoint file containing the parameters for that particular file system.

For example, when setting up the *Photo Storage* file system from *cheapfs.com* (shown in Figure 1), the file system UI prompts the user for her FS manager URL, file system on which to create the mountpoint file, the initial tag for the mountpoint file and mountpoint tags, and creates an encrypted mountpoint file that includes the user's credentials to access the file system.

When the file system is launched during normal use, it does not need any input from the user—the FS manager sends it the mountpoint information stored in the mountpoint file as part of the file system initialization, which includes the credentials required to access the file system.

3.9 Putting it all together

We now describe an example that illustrates how the different BSTORE components work together.

Say a user uses the *Money Manager* application shown in Figure 1 to compute her taxes. Assume that the tax files are tagged with *finance.com#tax*, where the principal of *Money Manager* is *finance.com*, and that the tax files are stored on *reliablefs.com*.

When the user launches *Money Manager*, it launches the user's FS manager in a separate browser window and initializes communication with it. It then sends the FS manager a **search** request to search for files with tag *finance.com#tax* on the *reliablefs.com* file system. The FS manager mounts the file system (if necessary) by launching it in a separate browser window. Using its delegation rules table, the FS manager computes the *acltagset* for *finance.com* on *reliablefs.com* and forwards the request to *reliablefs.com* file system with the *acltagset*. The file system retrieves all files that are tagged with *finance.com#tax* and returns only the handles of those files which are readable by *finance.com*, as specified by the *acltagset*. This result is routed back to *Money Manager* by the FS manager.

Money Manager uses these file handles to fetch the file data by issuing **get** requests; these requests follow a similar path as the **search** requests. Figure 1 illustrates the request flow for a **get** request from *Money Manager* to the *reliablefs.com* file system.

4 IMPLEMENTATION

We built a prototype of BSTORE, including the FS manager and a storage file system, which together comprise 2199 lines of Javascript code, 712 lines of PHP, and 136 lines of HTML.

All BSTORE client-side components are written in Javascript and work with the Firefox and Google Chrome browsers. Each component runs in a separate browser window, and is loaded from a separate domain. Inter-domain communication within the browser is via `postMessage`. The origin domain of the caller is included by the browser platform as part of the `postMessage` call, and is used as the caller principal.

4.1 Storage file system

The Javascript component of the prototype storage file system implements the BSTORE storage API. The network storage component is written in PHP, and currently runs on a Linux/Apache stack. Communication between the file system Javascript and the PHP components is via `XMLHttpRequest` POST. All information in the request and response, except for file data, is JSON encoded. File data in a **set** request is sent as a binary byte stream. File data in the response of a **get** request is Base64 encoded, since `XMLHttpRequest` does not support byte streams in the response. The response is also gzip encoded to offset the increase in size due to Base64 encoding.

User authentication to the file system is via username and password. The network storage uses a ext3 file system for storage and BSTORE files are stored as files in the file system. A file's version number is stored as an extended attribute of the ext3 file and file tags are stored in a database. The tags database is indexed for efficient search.

4.2 FS manager

To keep track of delegation rules, BSTORE represents each principal with a file in the root file system, containing the delegation rules from that principal to other principals. To avoid file system access on every access check, the FS manager caches these rules in memory. A cached entry for a principal is invalidated if the delegation rules were changed for that principal in that FS manager. To account for other FS manager instances, the cache entries also have an expiry time (currently 5 minutes) and are refreshed in the background. A better approach would be to have the FS manager be able to register for notifications from the file system when the files storing the delegation rules are modified. Due to this and other similar examples, we are considering adding version change notifications to BSTORE.

5 APPLICATIONS AND FILE SYSTEMS

To illustrate how BSTORE would be used in practice, we implemented one Javascript application that uses BSTORE, and ported three existing Javascript web applications to use BSTORE. We also implemented encryption and checkpointing file systems, to demonstrate the use of layered file systems. Table 3 summarizes the amount of code involved in these applications and file systems, and the rest of this section describes them in more detail.

5.1 Applications

The following applications work with our current BSTORE prototype:

Application/FS	Original LOC	Modified LOC
Shell	337	NA
Pixastic	4,245	81
jsvi	3,471	74
TrimSpreadsheet	1,293	66
EncryptFS	239	NA
CheckpointFS	595	NA

Table 3: Lines of Javascript code for BSTORE applications and file systems.

Shell provides basic commands that the user can use to interact with the storage system, including search, set, cat, stat, and unlink. We implemented this application from scratch.

Pixastic [28] is an online Javascript photo editor that performs basic transformations, such as rotate, crop, and sharpen, on a user's images. The version on the author's website allows a user to upload an image from their computer, perform transformations on it, and download it back to their computer. We modified Pixastic to work with BSTORE, so that a user can load, manipulate, and save images to BSTORE.

jsvi [18] is a vi-clone written purely in Javascript. The original version supports most vi commands and is a fully functional editor, but does not allow a user to load or save files (even to her local desktop). It only temporarily saves the file contents to a HTML text area, but they are lost once the web page is refreshed. Our modified jsvi works with BSTORE, and loads and saves files like a typical vi editor.

TrimSpreadsheet [34] is an open-source Javascript spreadsheet engine, and lets a user work with spreadsheet data in the browser. In the original application, the spreadsheet data was stored as HTML markup, which meant spreadsheets could be edited in a browser but the changes did not persist across browser sessions. We modified TrimSpreadsheet so that it can save and load spreadsheet data from BSTORE.

Modifying Pixastic, jsvi, and TrimSpreadsheet to work with BSTORE was straightforward and involved less than a day's worth of work to understand each application and add code to interact with BSTORE. Table 3 gives a tally of the lines of code for each application; for the applications we modified, the table also gives the number of lines changed to port the application to BSTORE. As can be seen from the table, porting applications required relatively few modifications to the application code.

5.2 Layered file systems

This section describes the motivation, design details, and usage of the encryption and checkpointing file systems. Each file system is a few hundred lines of code and involved a few days of effort to implement (as opposed to a couple of months of work to implement the base system). Given this experience, we feel that layering additional functionality on an existing BSTORE system is relatively easy.

5.2.1 Encrypting file system

Consider the scenario described in Section 2, where a user wants to encrypt her financial data before storing it. The user does not trust the underlying storage provider enough to store sensitive data in the clear, but trusts encryption functionality provided by, say, `encrypt.com`. In BSTORE, she can configure her applications to talk to the underlying storage via `encrypt.com`'s encrypting file system. We built a sample encrypting file system, *EncryptFS*, for this scenario, which works as described below.

EncryptFS provides a setup UI that the user can use to create a mountpoint. The setup process is similar to that of the Photo Storage file system, described in Section 3.8. The resulting mountpoint file stores the encryption password and the backing file system that EncryptFS uses to store its data. Once the mountpoint is created, the user adds a delegation rule to the FS manager allowing EncryptFS access to its backing file system. The user then proceeds to configure her financial application to use EncryptFS as discussed in Section 3.8.

We use the jsCrypto library from Stanford to perform cryptographic operations in Javascript [33]. We use the OCB mode of encryption with block size, key, IV, and MAC being 16 bytes. On a **set** request, the object contents are encrypted using the encryption key. The MAC and IV are attached to the beginning of the encrypted content, and the request is then written to the storage. On a **get** request, the encrypted content is fetched from storage, MAC and IV are extracted, the rest of the content is decrypted, the MAC on relevant tags is verified and the decrypted content returned.

Tags are critical in making access control decisions and so an untrusted FS cannot be trusted to return the correct tags. To get around this problem, EncryptFS MACs tags during **settag** and on a **gettag** will only return tags that have a valid tag. EncryptFS is relatively simple and does not prevent replay or rollback attacks. We can use well known techniques, such as those in the SUNDR file system [19], to get around such attacks.

5.2.2 Checkpointing file system

Imagine a user who wants to try a new application, say a photo retouching software that runs through her photo collection and enhances images that are below a certain quality. She does not trust the reliability of the software and does not know whether it would leave her photo collection in a damaged state if she lets it run on it. The simple checkpointing file system, *CheckpointFS*, that we describe here, helps with this situation by keeping an undo log of all the changes that the application made to

the storage system, and at the end of the application run, giving the user an option either to commit the changes or to revert them.

The set up for CheckpointFS is similar to that of EncryptFS, except that the mountpoint configuration in this case only consists of the backing file system where CheckpointFS stores the data and its undo log.

CheckpointFS records undo information in its log for every mutating operation. The undo information consists of the operation performed, timestamp, and version information for the file on which the operation was performed. In addition, for **settag** and **set**, a copy of the file tags and file contents respectively, is stored in the undo log. CheckpointFS stores these undo log records in memory and dumps them every minute to log files (numbered sequentially). These records could also be stored in browser local storage instead of memory if crash safety is an issue.

CheckpointFS keeps logging requests until the user indicates in its UI that she is done with her application session. At this point, all in memory logs are dumped, and CheckpointFS temporarily stops accepting further requests. The user is then given the choice to either to rollback to previous checkpoint or to commit the changes thereby wiping out the old checkpoint and creating a new one. If the user chooses to rollback, all the logs are read and the version information is checked to make sure that no other application performed an intervening mutating operation on the backing file system that will be clobbered by the rollback.

Though the current UI is simple and provides only one checkpoint, the information CheckpointFS logs could be used to provide more finer grained rollback capabilities. For example, CheckpointFS could store multiple checkpoints and allow the user to rollback to any previous checkpoints, it could automatically take a checkpoint at regular intervals, or it could provide a finer-grained undo of a subset of files.

6 BSTORE PERFORMANCE

For BSTORE to be practical, it should have acceptable overhead and its performance should be competitive with alternate data access mechanisms for web applications. Web applications today typically use `XMLHttpRequest` (XHR) to asynchronously GET data from, and POST data to web servers. We ran a set of experiments to measure the performance of BSTORE under different configurations, and compared it with XHR GET and XHR POST. We also measured the overhead of BSTORE's layering file systems, and compared BSTORE's performance on two different browsers.

For all experiments, the BSTORE file system server was an Intel Xeon with 2GB of RAM running Debian Squeeze, with Apache 2.2 and PHP 5.3.2. The client machine was an Intel Core i7 950 with 12GB of RAM running Ubuntu 9.10. The web browsers we used were Firefox 3.5.9 and Google Chrome beta for Linux. The local network between the client and the server is 100Mbps ethernet.

Size	BSTORE-Get	XHR-Get	BSTORE-Set	XHR-Post
1 KB	17.6 ms	5.0 ms	18.9 ms	5.3 ms
5 KB	18.6 ms	6.0 ms	19.0 ms	5.9 ms
10 KB	19.7 ms	6.6 ms	19.4 ms	6.5 ms
100 KB	40.2 ms	18.8 ms	34.0 ms	15.7 ms
500 KB	117.5 ms	66.6 ms	102.3 ms	59.4 ms
1 MB	225.9 ms	141.2 ms	174.8 ms	116.8 ms

Table 4: Comparison of get and set operation performance on a BSTORE file system to XHR-get and XHR-post.

6.1 BSTORE file system performance

In our first experiment, we compare the performance of BSTORE **get** and **set** with XHR GET and XHR POST on a local network. The experiment consists of fetching and writing image files of various sizes ranging from 1 KB to 1 MB. The server side for XHR GET and XHR POST is a simple PHP script that sends back or accepts the required binary data. The BSTORE **get** and **set** requests are to the root file system, and mirror the request flow illustrated in Figure 1, with delegation rules set to allow read and write access to the required files for the application running the experiment. Since this experiment is on a local network, it highlights the overhead of BSTORE mechanisms, as opposed to the network transfer cost. The web browser used in this experiment is Firefox.

The results of the experiment are shown in Table 4. To remove variability, we ran 24 runs of the experiment and removed the runs with the two highest and two lowest timings. The numbers shown in the table are the average times of the remaining 20 runs. From the table, we see that XHR operations are about three times faster than BSTORE operations for small files. For a large file of 1 MB size, BSTORE-Get is about 60% slower than XHR-Get and BSTORE-Set is about 50% slower than XHR-Post. Most of this overhead is due to processing within the browser. For example, for BSTORE-Set on a 1 MB file, 19.9% of the time is spent in encoding/decoding data in the browser, 5.7% in communication within the browser using `postMessage`, and 73.6% in communication between the storage file system Javascript and backend server using XHR POST.

BSTORE-Get is slower than BSTORE-Set due to gzip overhead. Recall from Section 4 that the requests from BSTORE storage file system Javascript to the backend are in binary; the responses, however, are Base64 encoded and gzipped, as XHR does not support byte streams in the response. This means that BSTORE-Get involves compression of data on the backend and decompression in the browser, which is not present in BSTORE-Set. The overhead for these operations dominates the total time as the local network is fast.

Size	BSTORE-Get	XHR-Get	BSTORE-Set	XHR-Post
1 KB	117.7 ms	105.0 ms	117.6 ms	105.5 ms
5 KB	217.5 ms	205.0 ms	268.4 ms	255.3 ms
10 KB	218.9 ms	205.5 ms	268.9 ms	255.3 ms
100 KB	894.0 ms	927.7 ms	981.0 ms	1059.7 ms
500 KB	4223.2 ms	4172.9 ms	4451.5 ms	4409.4 ms
1 MB	8622.3 ms	8500.9 ms	8978.7 ms	8916.6 ms

Table 5: BSTORE FS performance on a 1Mbps, 100ms latency network.

Size	BSTORE-Get	XHR-Get	BSTORE-Set	XHR-Post
1 KB	27.5 ms	15.2 ms	29.1 ms	15.6 ms
5 KB	37.4 ms	16.6 ms	48.7 ms	36.1 ms
10 KB	38.5 ms	20.6 ms	49.5 ms	36.9 ms
100 KB	116.2 ms	94.8 ms	138.3 ms	119.8 ms
500 KB	451.4 ms	423.9 ms	498.9 ms	472.9 ms
1 MB	901.1 ms	863.2 ms	982.1 ms	935.1 ms

Table 6: BSTORE FS performance on a 10Mbps, 10ms latency network.

Though overhead of BSTORE requests seem high, they represent performance on an unrealistically fast network. On a more realistic network with lower bandwidth, the network cost dwarfs BSTORE overhead as the next experiment illustrates. Furthermore, these overheads are primarily due to lack of support for binary data in XHR responses and in `postMessage`, and can be significantly reduced by adding this support. The `responseBody` attribute of XHR, being considered by W3C for a future version of the XHR specification, supports binary byte streams, and is a step in this direction.

6.2 Wide-area network performance

In order to evaluate BSTORE overhead in real-world networks, we ran the same experiment as above on simulated networks with realistic bandwidths and latencies. We chose two network configurations: a 1Mbps network with 100ms round-trip latency, and a 10Mbps network with 10ms round-trip latency. The slow networks are emulated using the Linux traffic control tool (tc).

The results are shown in Tables 5 and 6. We see from the tables that the BSTORE overhead, as compared to plain XHR, drops considerably. Overhead of BSTORE-Get over XHR-Get for a 1 MB file drops from 60% in local network to 4% on 10Mbps network, and 1.4% on 1Mbps network. Similarly, overhead of BSTORE-Set over XHR-Post for a 1 MB file drops from 50% in local network to 5% on 10Mbps network, and 0.7% on 1Mbps network. It is clear from these results that the browser overheads in BSTORE become insignificant compared to network cost for realistic networks.

Another point illustrated in these tables is the effect of gzip in slower networks. For a 1 MB file on slower networks, BSTORE-Get is faster than BSTORE-Set, whereas the opposite held true on the local network. This is because gzip reduces the number of bytes transferred over the network; on slow networks the resulting time saved more than offsets the time taken to compress and decompress the data.

6.3 Performance of layered file systems

The previous experiments focused on the scenario of an application accessing data on the BSTORE root file system. BSTORE also supports layered file systems. In this experiment, we measure the overhead of layered file systems on the same workload as the previous experiments. We use three layered file systems: a null layered file system which passes data back and forth without any modification, and *EncryptFS* and *CheckpointFS* described in Section 5.2. The measurements are performed on the local network.

Table 7 shows the results. From the table we see that the overhead of a null layered file system is small—about 6% for BSTORE-Get and 7% for BSTORE-Set, for the 1 MB file. This overhead is due to the extra `postMessage` calls and encoding/decoding as requests pass through the layered file system and FS manager. We believe this overhead is reasonable; also, on a slower network, this overhead becomes a smaller fraction of the overall time, further reducing its impact.

For *EncryptFS*, the bulk of the time is spent in cryptographic operations. For a 1 MB file, decryption takes 3095ms (85.1% of Get time) and encryption takes 2534ms (90.3% of Set time). We also observed that a `postMessage` that follows a crypto operation takes more than an order of magnitude longer than other `postMessage` calls. We believe that this variability is a characteristic of the Firefox Javascript engine. We confirmed our suspicion by testing `postMessage` times after a CPU intensive tight loop, and observing that it does indeed take an order of magnitude longer than normal.

For *CheckpointFS*, Get performance is close to that of null layered file system, as it does not do anything on a Get. The overhead in its Set operation is due to logging—this involves an extra RPC to fetch the old file contents and store them in the log. For this experiment, the old file was always 1 KB in size.

6.4 Browser compatibility

Today people use many different browsers, and an important consideration for a good web application framework is cross-browser support. We primarily implemented BSTORE for Firefox, but were able to run it on Google Chrome with a small modification. XHR POST on Chrome does not support sending binary data, and so we had to change our implementation to send data in Base64 encoded form. We predict that porting BSTORE to other browsers such as IE and Safari will require some changes, mainly because of the difference in ways these browsers handle events, `postMessage`, and some other differences in Javascript engine.

To see how BSTORE performs on Chrome, we ran the first experiment using Chrome. Table 8 shows the results as compared to performance on Firefox. Chrome is slower

Size	No Layering		Null Layered FS		EncryptFS		CheckpointFS	
	Get	Set	Get	Set	Get	Set	Get	Set
1 KB	17.6 ms	18.9 ms	19.2 ms	19.9 ms	24.9 ms	28.2 ms	19.2 ms	37.8 ms
5 KB	18.6 ms	19.0 ms	20.3 ms	20.4 ms	45.0 ms	47.2 ms	20.4 ms	37.3 ms
10 KB	19.7 ms	19.4 ms	21.4 ms	21.1 ms	63.4 ms	65.2 ms	21.8 ms	38.8 ms
100 KB	40.2 ms	34.0 ms	43.0 ms	36.8 ms	300.5 ms	292.3 ms	45.0 ms	53.7 ms
500 KB	117.5 ms	102.3 ms	132.9 ms	110.0 ms	1494.5 ms	1359.8 ms	136.0 ms	123.2 ms
1 MB	225.9 ms	174.8 ms	238.5 ms	187.8 ms	3636.0 ms	2806.8 ms	247.4 ms	208.6 ms

Table 7: Performance of various layered file systems under BSTORE performing get and set operations.

Size	Firefox		Chrome	
	Get	Set	Get	Set
1 KB	17.6 ms	18.9 ms	14.8 ms	15.6 ms
5 KB	18.6 ms	19.0 ms	15.8 ms	16.1 ms
10 KB	19.7 ms	19.4 ms	17.4 ms	18.6 ms
100 KB	40.2 ms	34.0 ms	47.6 ms	44.6 ms
500 KB	117.5 ms	102.3 ms	141.8 ms	143.4 ms
1 MB	225.9 ms	174.8 ms	258.4 ms	256.4 ms

Table 8: Performance of BSTORE on Firefox and Chrome.

than Firefox on larger files, and the difference is primarily due to slower `postMessage` and slower XHR POST. For example, for a 1 MB file get, `postMessage` was 30.2ms in Chrome as compared to 7.2ms in Firefox, and XHR POST was 226.1ms in Chrome as compared to 202.9ms in Firefox. From these results, overall we can conclude that BSTORE works reasonably well in Chrome.

7 DISCUSSION

In its current form, BSTORE does not support all applications. Collaborative web applications (such as email and chat) need a server for reasons other than storage, and BSTORE does not eliminate the need for such servers. Social applications, such as Facebook, require sharing data between users. BSTORE currently does not support cross-user sharing; for one, the principals do not include a notion of a user. We plan to explore extending BSTORE to support cross-user sharing, perhaps by building on top of OpenID. In the meanwhile, social applications can still use BSTORE to store individual users' files, and implement cross-user sharing themselves.

The principal of an application in BSTORE is the URL origin from where the application is loaded. This makes it difficult to support serverless applications, where the Javascript code for the application can be hosted anywhere, or even passed around by email. BSTORE could be extended to add support for principals that are a hash or a public key corresponding to application code.

We have chosen to support an object get and set API in BSTORE, which works well for many applications, including the ones we used in our evaluation. Likewise, our tagging model fits well with the data model of existing applications like GMail and Google Docs [14], and can be also used to express traditional file-system-like hierarchies. However, some applications may require a richer interface for querying their data, such as SQL, and tags cannot express the full range of such queries. Storing an entire SQL table as a file in BSTORE may be acceptable for small data sets, but accessing large data sets efficiently would require adding a database-like interface along the lines of Sync Kit [5].

With BSTORE, users potentially have to worry about providing storage, whereas in the current model all storage is managed by the application provider. The simplicity of today's model could also be supported with BSTORE, where each application mounts its own storage in the user's file system manager, with the added benefit that applications can now easily share data with each other. At the same time, the user could have the option of procuring separate storage from a well-known online storage provider, such as Amazon S3 or Google, which would then store data for other applications the user runs, and backup data from existing application stores.

Due to the level of indirection between tags and the associated access rights, a user may inadvertently leak rights by tagging a file incorrectly without realizing it. To avoid this risk, the file tagging UI in applications and FS manager can resolve and display the new principals being granted access due to the addition of the tag.

Currently, all BSTORE components run in separate browser windows; this can present the user with too many windows. This can be mitigated by running the FS manager in a window and all the file systems as iframes within the FS manager window. If extending the browser is feasible, a browser extension to support "background windows" would provide a better user experience.

8 RELATED WORK

The idea of a unified file system namespace has been explored in earlier systems like Unix and Plan 9 [27]. Moreover, various distributed file systems [1, 6, 16, 29] provide a uniform storage platform, along with the security mechanisms needed for authentication and access control. BSTORE addresses new challenges posed by web applications, including the need for different storage models (using tags and tag search), the need for applications to mount their own file systems, and the need for flexible delegation of access control, without requiring any changes to client-side browsers.

Similar to tags in BSTORE, semantic file systems [11] and the Presto document system [8] provide alternate file

system organization using file attributes, in addition to the hierarchical file system namespace.

SUNDR [19] provides a network file system designed to store data securely on untrusted servers, and ideas from it would be directly applicable to designing a better encrypting file system for BSTORE.

Google gears [12], HTML5 [36] local storage, and Sync Kit [5] aim to improve web application performance and enable offline use with client-side persistent storage. However, these mechanisms still provide isolated storage for each application, and do not address sharing of data between applications.

There are also a number of browser plug-ins that provide alternative environments for deploying client-side applications [3, 7, 20, 38]. While some of them provide machine-local storage, none of them provide *user*-local storage that is available from any machine that the user might access. Accessing data stored in BSTORE from one of these environments currently requires going through Javascript; in the future, BSTORE could support native bindings for other execution environments.

Some websites provide mechanisms, such as REST APIs or OAuth, for users to access their data from external services. OAuth is an open protocol that allows a user to share her web application data with another web application from a different origin. However, unlike BSTORE, both applications should know of each other before hand, limiting the number of applications that can use this. Also, OAuth requires involvement of the web application servers and cannot support Javascript-only applications with no backend. Google provides external access to user data through APIs utilizing the Google Data Protocol [13] in a similar manner. BSTORE simplifies data sharing by avoiding the need for all applications to know about each other ahead of time, and does not require server involvement for data sharing.

Menagerie [10] allows sharing of a user's personal data between multiple web applications and provides standardized hierarchical naming and capability-based protection. However, like OAuth, it requires backend servers to communicate with each other. Also, unlike BSTORE's tag-based mechanisms, Menagerie's file systems don't support per-application file system organization and delegation of access rights based on file properties.

Cloud computing and storage services such as Amazon S3 [4], and Nirvanix [24] provide web application developers with the option of renting storage and on-demand computing. However, developers still need to bear the costs of renting the server capacity, and make data management decisions on behalf of users. BSTORE allows users to control their own data, such as by encrypting, mirroring, or backing it up.

Cross-origin resource sharing [35] provides a mechanism for client-side cross-origin requests. This allows for pure client-side apps to access data from other websites which will in turn implement cross domain authentication and access control. However, using this mechanism alone does not provide a single namespace for all user data, and does not provide an access control and delegation mechanism such as that provided by BSTORE.

Hsu and Chen [17] describe the design of a secure file storage service for Web 2.0 applications. While the motivation for their work is similar to ours, there are a number of limitations of their work that BSTORE's design addresses. First, their file system doesn't support a unified namespace and there are no mountpoints. It cannot support delegation, encryption, or checkpointing, and doesn't support versioning, which means that applications sharing data can run into problems. Finally, BSTORE's tags allow applications to annotate each others' files, and to delegate specific access, without requiring write privileges, something that is not possible in Hsu and Chen's system.

9 CONCLUSION

This paper presented BSTORE, a framework for separating application code from data storage in client-side web applications. BSTORE's architecture consists of three components: *file systems*, which export a storage API for accessing user data, a *file system manager*, which implements the user's namespace from a collection of file systems, and enforces access control, and *applications*, which access user data through the file system manager. A key idea in BSTORE is the use of tags on files. Tags allow applications to organize their data in different ways. An application also uses tags to designate the precise files it wants to delegate rights for to other applications, even if it cannot write or otherwise modify those files itself. The BSTORE file system manager interface is egalitarian, allowing any application to specify delegation rules or mount new file systems, in hopes of avoiding the need for applications to supply their own file system manager.

A prototype of BSTORE is implemented in pure Javascript, which runs on both the Firefox and Chrome browsers. We ported three existing applications to run on BSTORE, which required minimal source code modifications, and wrote one new application. We also implemented three file systems, including ones that transparently provide encryption or checkpointing capability using an existing file system for storage. Finally, our prototype achieves reasonable performance when accessing data over a typical home network connection.

ACKNOWLEDGMENTS

We thank our shepherd, John Ousterhout, as well as Jon Howell, Adam Marcus, Neha Narula, and the anonymous reviewers for providing feedback that helped improve this paper. This work was supported by Quanta Computer and by Google.

REFERENCES

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

[2] S. Aaronson. Off the grid. http://scottaaronson.com/blog/?p=428.

[3] Adobe. Adobe Flash. http://www.adobe.com/flashplatform.

[4] Amazon. Amazon simple storage service. http://aws.amazon.com/s3/.

[5] E. Benson, A. Marcus, D. Karger, and S. Madden. Sync Kit: A persistent client-side database caching toolkit for data intensive websites. In *Proceedings of the World Wide Web Conference*, 2010.

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

[7] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation*, San Diego, CA, December 2008.

[8] P. Dourish, W. K. Edwards, A. LaMarca, and M. Salisbury. Using properties for uniform interaction in the presto document system. In *Proceedings of the ACM Symposium on User Interface Software and Technology (USIT)*. ACM, 1999.

[9] FotoFlexer. Fotoflexer: The world's most advanced online image editor. http://www.fotoflexer.com.

[10] R. Geambasu, C. Cheung, A. Moshchuk, S. D. Gribble, and H. M. Levy. The organization and sharing of web-service objects with menagerie. In *Proceedings of the World Wide Web Conference (WWW)*, 2008.

[11] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. toole. Semantic file systems. In *13thACM Symposium on Operating Systems Principles*, pages 16–25. ACM, 1991.

[12] Google. Gears: Improving your browser. http://gears.google.com/.

[13] Google data protocol. http://code.google.com/apis/gdata/.

[14] Google docs. http://docs.google.com/.

[15] Google. Picasa web albums. http://picasaweb.google.com.

[16] J. H. Howar, 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 Transactions on Computer Systems*, 6(1):51–81, 1988.

[17] F. Hsu and H. Chen. Secure file system services for web 2.0 applications. In *Proceedings of the ACM Cloud Computing Security Workshop*, Chicago, IL, November 2009.

[18] jsvi – javascript vi. http://gpl.internetconnection.net/vi/.

[19] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 91–106, San Francisco, CA, December 2004.

[20] Microsoft. Silverlight. http://silverlight.net/.

[21] R. Miller. Ma.gnolia data is gone for good. http://www.datacenterknowledge.com/archives/2009/02/19/magnolia-data-is-gone-for-good/.

[22] MIT Haystack Group. NB 2.0. http://nb.csail.mit.edu/.

[23] A. Modine. Web startups crumble under amazon s3 outage. http://www.theregister.co.uk/2008/02/15/amazon_s3_outage_feb_2008/.

[24] Nirvanix. http://www.nirvanix.com/.

[25] Oauth. http://oauth.net.

[26] Photofunia. http://www.photofunia.com.

[27] R. Pike, D. Presotto, K. Thompson, H. Trickey, and P. Winterbottom. The use of name spaces in plan 9. *ACM SIGOPS Operating System Review*, 27(2):72–76, 1993.

[28] Pixastic – online javascript photo editor. http://www.pixastic.com.

[29] 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*, 1985.

[30] S. Shankland and T. Krazit. Widespread google outages rattle users. http://news.cnet.com/widespread-google-outages-rattle-users/.

[31] Shutterfly. http://www.shutterfly.com.

[32] Slideroll. Slideroll online slideshows. http://www.slideroll.com.

[33] E. Stark, M. Hamburg, and D. Boneh. Symmetric cryptography in javascript. In *Proceedings of the Annual Computer Security Applications Conference*, 2009.

[34] TrimSpreadsheet. http://code.google.com/p/trimpath/wiki/TrimSpreadsheet.

[35] W3C. Cross-origin resource sharing: Editor draft. http://dev.w3.org/2006/waf/access-control/, December 2009.

[36] W3C. HTML 5 editor's draft. http://dev.w3.org/html5/spec/, January 2010.

[37] Yahoo. flickr. http://flickr.com.

[38] B. Yee, D. Sehr, G. Dardyk, B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2009.