Tolerating Business Failures in Hosted Applications

Jean-Sebastien Legare, Dutch T. Meyer, Mark Spear, Alexandru Totolici,
Sara Bainbridge, Kalan MacRow, Robert Sumi, Quinlan Jung, Dennis Tjandra,
David Williams-King, William Aiello, and Andrew Warfield
University of British Columbia, Vancouver, BC, Canada

Abstract

Users of hosted web-based applications implicitly trust that those applications, and the data that is within them, will remain active and available indefinitely into the future. When a service is terminated, for reasons such as the insolvency of the business that is providing it, users risk the immediate loss of software functionality and may face the permanent loss of their own data. This paper presents Micasa, a runtime for hosted applications that allows a significant subset of application logic and user data to remain available even in the event of the failure of a provider's business. By allowing users to audit application dependence on hosted components, and maintain externalized and private copies of their own data and the logic that allows access to it, we believe that Micasa is a first step in the direction of a more balanced degree of trust and investment between application providers and their users.

1 Introduction

"`On July 3, 2012, picplz will shut down permanently and all photos and data will be deleted. [...] Thank you for your support of picplz and we apologize for any inconvenience this may cause you.`"

—Message received by the users of picplz, an (insufficiently) well-financed photo sharing app, on June 1, 2012 [16].

SoCC'13, 1–3 Oct. 2013, Santa Clara, California, USA.

ACM 978-1-4503-2428-1.
http://dx.doi.org/10.1145/2523616.2523618

Will the cloud-based applications that you use today still exist in ten years? What would you lose if they were to discontinue service tomorrow?

As a growing amount of the software that we use—both as individuals and as organizations—is offered in the form of hosted services, questions like these demand careful consideration. Application hosting is a competitive and operationally expensive market, and provider business models do not always prove to be sustainable. As has already been the case with a number of real systems, the abrupt application end-of-life (EOL) that follows the decision to discontinue a given service risks the loss of both software functionality and user data [5, 9, 13, 10, 16].

Interestingly, this exposure to risk is not a necessary property of hosted applications: the consolidation of application logic and the storage of user data within an application provider's servers is simply the way that systems have been built in the past, and is a model that is supported by most popular development frameworks. Moreover, building a large-scale hosted application is a challenging problem unto itself, and providers have understandably chosen to invest efforts in developing and scaling their own applications rather than providing features that anticipate their own demise.

We believe that the risk presented by application EOL is significant. As application markets evolve over the next decade, it seems very likely that additional applications will cease operations, resulting in inconvenience and potentially even considerable expense for users. As a result, users may hesitate to invest time in new applications, and organizational software procurement processes may place priority on established and incumbent applications. The perceived risk of using a new service will further challenge the ability of new entrants to innovate and succeed in the application marketplace.

In this paper, we describe *Micasa*, a web-based application runtime that treats the sudden and permanent unavailability of an application provider as a recoverable failure mode. Micasa makes the trust that users are placing in an application service provider explicit, by allowing large portions of application data and function-

ality to operate *independently* of the provider's hosting environment. Our system aims to find a balance that preserves the benefits of today's hosted applications—including the ease of adoption, maintenance, and software upgrading—while allowing providers to clearly demonstrate to users that their data and a relevant subset of application functionality will remain available in perpetuity.

Micasa applications are partitioned by developers into server- and client-side components. Client-side application logic, written in JavaScript and HTML5, is stored alongside user data in a third-party storage service (such as Amazon's S3), chosen by the user. Under normal operation, the provider is responsible for maintaining central, private data and computationally demanding functionality. However, in the event that the provider is no longer available, the application is capable of continuing to offer a subset of functionality, even "social" features requiring interaction with data owned by other users, using only the client-side code and associated storage services. To ensure that the exposure to lost functionality remains explicit, Micasa includes a browser-side monitor that audits RPC interactions with hosted components of the application, and also allows users to "unplug" applications, simulating provider failure.

While some hosted applications have provided interfaces for users to "take out" [8] their data, the result is generally a large volume of JSON- or XML-encoded data, leaving no mechanism for users to usefully interact with the contents. Further, as data representations and schemas may change over time, writing third-party tools to interact with these backups has proven to be a challenging task [21]. By packaging application logic for data access and presentation alongside user data, Micasa ensures that user data is preserved in a manner that is more likely to be usable immediately upon EOL and that can be preserved, in an archival sense, for long periods into the future.

Micasa takes advantage of rich, browser-side execution environments and user-facing storage services in order to achieve a clearer degree of trust between users and application providers. While it does not protect the entirety of application functionality in the event of EOL, we believe it is a useful first step. In particular, the risk mitigation enabled by Micasa allows upstart providers to make clear claims to potential customers about service longevity even in the face of end-of-life (EOL), which provides a competitive advantage over services that cannot (or choose not to) make similar claims.

1.1 Challenges

New applications written with Micasa can provide users with a clear guarantee of both features of an application and the set of their own data that will remain available even after EOL. Our system seeks to preserve the scalability, availability, and performance goals of today's centralized application models, without entrusting a single fallible entity with the hosting of data and application logic. Our approach is to move user data out to external cloud storage services and create an access path to this data for the application. Many characteristics of hosted applications make this decentralization difficult:

Single point of authority and control. Centralized control services, accessible with a well-known identifier (DNS name or URL) act as a rendezvous for client browsers that are unable to communicate directly with one another.[1] This control service updates clients on every visit, enforces authentication, authorization, input validation, and serialization of requests as per the desired application policies.

Proprietary information hiding. Centrally hosted applications provide a convenient location to store data invisible to clients, such as the exhaustive list of registered users, algorithms, and keys.

Scalability. Centrally hosted providers benefit from elastic scalability within a single operational environment. A provider can use cloud computing to grow, shrink, and relocate their compute power to adapt to changing user demand and maintain suitable performance levels.

Global view. Centrally hosted providers benefit from a global view on all data in the system. This is useful for building fast search engines, spam detectors, and enforcing constraints across all data (e.g., uniqueness of user email addresses). Finally, storing all data centrally and controlling access to it allows application developers to decide on storage formats and infrastructure, and evolve them over time.

Micasa applications are distributed by a central hosting provider. When the provider is available, the application benefits from all the advantages listed above. Unlike traditional applications however, Micasa applications can preserve core functionality in the event that the service is discontinued. When this occurs we say that the application has become "unplugged".

Micasa eliminates the need for the application provider to mediate access to user data and protect data integrity. However, unplugged applications are not exact analogs of today's centralized applications—we do not attempt to distribute proprietary information, nor preserve a global view on all data.

Micasa will support certain classes of applications better than others when unplugged. Applications which are heavily based on individual user-data-driven views

[1] There are upcoming browser peer-to-peer technologies, but they require addressable proxies to establish connections.

such as blogs or photo galleries are the easiest to support. With Micasa's data interface, applications can share data objects between users, and support user comments and ratings (TwoCans, in Section 2, is an example of this).

On the other hand, Micasa is less suitable for applications that rely heavily on proprietary or global data, e.g., a web search engine, or a matchmaking dating site. There is still value in using Micasa for these applications however, because Micasa allows archiving, indexing, and searching both the content in a user's personal store, as well as content shared by the user's "friends". For instance, a hypothetical Netflix-Micasa application might not offer recommendations when unplugged (because computing those might require ratings of all users), but still allow an individual user to look at (or search through) the list of all the previously viewed items and ratings in their social graph (HotCRP-P, in Section 5.2, covers such archiving and search).

Our implementation focuses on web applications, and therefore we limit unplugged operation to computations supported by modern web browsers. Also, we do not offload any of the application logic to storage-side services, aside from access control checks. This implies, for instance, that a webmail service built on Micasa could support, once unplugged, access to inbox contents and sharing of messages via our sharing API, but not reception of email via SMTP from other mail servers.

Applications that serve cached pages for high performance or that offer notifications to their users, such as Twitter or Flickr, can still do so while the provider is present. After EOL, Micasa can offer viable fallback modes of operation. For instance, notifications can be replaced with polling, and caches can be replaced with direct access to user data stores (Section 5.3 covers the caching example). Other examples of unplugged functionality are summarized in Table 1.

2 Architecture

We will explain Micasa's architecture through a motivating application example. Figure 1 is a high-level architecture diagram of *TwoCans*, a shared chat and messaging system similar to services such as Google Chat, Google+ Hangouts, or Facebook Chat. Its design is representative of a typical Micasa application. TwoCans is normally available as a browser-based application from a provider at a well-known URL. However, if the application is ever discontinued, TwoCans still has access to chat histories, as well as the ability to interact with existing known contacts. A key requirement to enable this post-EOL functionality is that user content not be stored by the central server, or else it could disappear with it.

The software provider distributes the TwoCans source

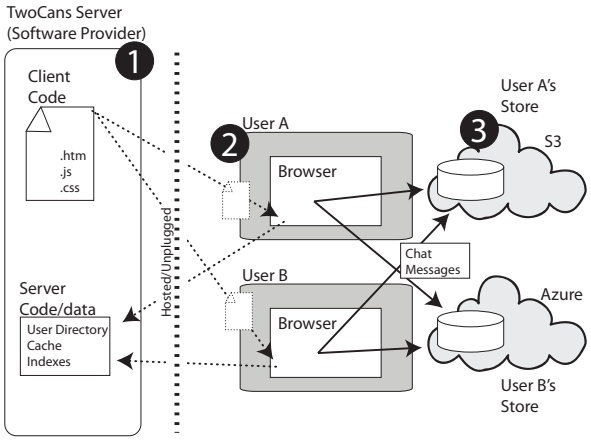

Figure 1: TwoCans, a typical Micasa application.

code from their servers, labelled as (1) in Figure 1. We discuss how this Micasa application differs from a traditional web application in Section 2.1. Label (2) shows this same code running inside the user's browser, where it links with the Micasa library, which we discuss in Section 2.2. This library includes a secure monitor, which ensures that the source code abides by the rules for unplugged applications. A code cache is also available locally at the client, for both the application code and the library itself.

In Micasa applications, users provide the application with the routable name of a Micasa-compatible data store of their choice (e.g., during registration), denoted by Label (3) in Figure 1 for TwoCans. Users store personal content on their chosen data store and retrieve the content of other users from their respective data stores. That is, users do not interact directly as in peer-to-peer systems nor do they interact using a service provider as a relay. Rather, they interact using the personal stores as intermediaries in what we call *peer-to-store* communication. The programming interface of the personal data store is discussed in Section 2.3.

Users could in principle run their own storage server on a home network. However, we assume that most users will use a commercial storage provider for their Micasa applications to take advantage of the durability, availability, and reachability of a commercial provider. A user may, but need not, use the same storage provider for every Micasa application.

By using third-party storage we are not simply exchanging dependence on one service for another. Internet-scale storage is now mature, highly reliable, and revenue-generating. The cloud storage business model depends on protecting the integrity of the data stored, and providing users with the ability to retrieve it. While dominant cloud storage providers have proven thus far to be stable and lasting service offerings, Micasa still al-

		Converted feature	
Feature Name	Classification (§1.1)	Unpluggable?	Solution Summary
ACL / Confidentiality	single-point	yes	Client-side crypto. Users store encrypted blobs and metadata. Group keys can be shared with closed caps.
User Registration	prop. information hiding	lost	No new users can register. Known registered users can be remembered however.
Content Discovery	global view	lost	Requires access to global data. Limited form of discovery possible out-of-band (e.g., URLs in emails).
Notifications	scalability	degraded	Polling for object modification time changes or pub-sub mechanisms implemented with append operations.

Table 1: Common application features, their categorization, and likely replacements in an unplugged application.

lows application code and data to be migrated from one storage provider to another. We assume that if a large storage provider were to go out of business, it would provide its customers with sufficient time to perform migration. We discuss data store migration in Section 2.3.4.

2.1 Micasa Applications

A chief challenge in Micasa is to survive the failures associated with provider end-of-life without sacrificing the many benefits conferred by centralized, cloud-based application architectures. In particular we wish to preserve performance and availability at scale. Rather than attempt to replace existing application models outright with a peer-to-peer architecture [23], our philosophy is to embrace the same core approach that centrally hosted applications use today, but endeavour to remove the availability of application providers themselves as a central point of failure.

The client-side code of Micasa applications makes heavy use of dynamic HTML changes and modern browser features from HTML5, such as CORS [18], to fetch resources from multiple third-party services. Like traditional web applications, the application provider serves the client-side code, and maintains global application data—data that is not owned by any particular user or group of users. Providers may also cache some user data, to accelerate certain operations.

Our model introduces an additional role for a storage service provider, which is to guard access to a user's authoritative copy of their data. A disconnection from the application service provider does not affect this role.

The monitor shipped with the Micasa library allows users to check whether application functionality remains available in the absence of the service provider (discussed in Section 2.2.2). We expect users will find this even more useful than a simple data check-out feature, and not take this as a sign that the service provider expects business failure.

2.2 Clients

Clients have access to a local code cache modelled after the HTML5 offline cache, that can be updated by simply visiting the application provider's website. This cache is periodically synchronized to the user's per-application personal data store so that it persists on durable storage.

Micasa applications require our client-side library, libeol, to be installed in the browser. libeol provides a JavaScript API, called Capability Storage Interface (CAPSI), for interacting with user data stores from client-side code. Our user data store API is described in Section 2.3. libeol also monitors network access to both the application provider, `tc.example.org` in the example, and storage providers. The installation of libeol simply consists of registering a new browser extension—only a few clicks are required.

Except for the presence of libeol, TwoCans possesses all of the regular functionality and appearance of a normal web 2.0 application: user actions in the view can issue RPCs to the provider to retrieve additional dynamic content (e.g., search and forms).

This form of deployment follows typical web navigation paradigms and makes trying out new applications very easy. We believe that Micasa applications could also be packaged in forms compatible with current browser "hosted application or local app" concepts such as Chrome Apps or Mozilla Apps [7, 15].

2.2.1 Connecting to Data Stores

Micasa's client library exposes CAPSI, which is used to request any data to and from Micasa data stores. Table 2 lists its methods, grouped by category. CAPSI allows users to create isolated stores for each application, called namespaces. A user can gain write access to one of the namespaces he or she owns by logging into the data provider from the application associated with that namespace. A session is only valid within a single namespace, for that client, on that application. Once

Category	Methods
Writing	`putlist`, `putblob`, `append`
Sharing	`mkget`, `mkappend`
Reading	`getlist`, `getblob`, `getnsroot`
Deleting	`delete`, `revoke`
Accounting	`login`, `logout`, `mkns`

Table 2: Methods in CAPSI.

an authenticated session is established with its storage provider, a client can create, manipulate, and share objects from the namespace with other users. Clients can also read or append objects from other storage providers, *unauthenticated*, provided that a valid reference to that object is presented. This way, no extra login procedures are required to access other user stores.

2.2.2 Library Installation and Audit

The client library places a monitor around the application code, which intercepts, classifies, logs, and possibly blocks all external requests issued. To audit an application's dependence on its provider, users can use the monitor to launch Micasa-enabled applications in "unplugged" mode. When launched this way, the monitor simulates the absence of the provider by artificially breaking all provider-bound connections passing through the application interface, verifying any of the provider's claims about robustness to EOL.

In order for the library to properly monitor the application, its code is loaded in the user's browser first, before the application starts loading. The early injection allows the monitor to gain access to the unmodified JavaScript environment. The library injects JavaScript code in the top of the page to enable CAPSI functionality, as is done in other JavaScript instrumentation scenarios (e.g., MugShot [14]).

When a network request is intercepted, the monitor consults a file supplied by the application developer. The Interface Definition Language (IDL) file simply contains a set of URL regular expressions that classify outgoing requests (more details on the format is available in Section 4.1.1). The monitor finds a matching entry in the file based on the request environment, and a policy decision is applied based on the entry's type and plugged/unplugged status. The monitor records the network interaction (and decision) into an audit log, and can optionally report the contents of the log back to a web server external to the browser.

This mechanism allows motivated parties with sufficient expertise to validate any warranty claims the provider could make about the application, and thus helps prove the provider's good intentions. For instance, if an application offered a private or local search into a user's own data (e.g., Section 5.2), an investigative user could trigger search actions in the UI while unplugged and verify in the logs that the application does not contact a central application server to gather search results.

2.2.3 Personal Search

A central provider is very useful for indexing and searching across all of the data associated with an application. However, if all of the indexing and search functionality is located at the provider and the provider fails, the clients are left without an ability to search even their own data. Micasa makes it feasible for clients to keep a rich index. Clients can maintain an index over their own content with references to their own store, as well as an index over all of the data to which they have been given access. An index mapping keys to capabilities can be built incrementally during the process of retrieving content from other users' stores. In addition, from the list of capabilities in the index, a client can periodically recursively crawl the content to which it has access to update the index for mutable objects that may have changed. We demonstrate the personal search capability of Micasa in our conference management system, called HotCRP-P, which is detailed in Section 5.2.

2.3 Data Stores

In Micasa, browser applications interact with user data stores. Currently, we require that these data stores export a capability interface, CAPSI. For a given application, different users are free to choose different data stores, and a given user is free to choose different data stores for different applications. While users may manage their own CAPSI-store, we envision a Micasa ecosystem that includes a number of commercial cloud storage providers that export an interface with the following:

Provider compatibility Users can choose their storage service, and can access stores of other users regardless of their chosen provider. The data should be migratable, so that users can change providers.

Sharing and access control Users can create data objects and share them with other users. References to objects can be used to control access to the information, and encryption can be used to protect the confidentiality of the data.

Revocation Users can revoke access to a previously shared object. This is the opposite of sharing.

Write Access Control To protect the storage footprint of user data stores, we disallow arbitrary writes to data

Field	Description
Object Reference	Type and content address of object
Server Reference	Used to resolve server hostname
Issuer	Userid of object owner
Audience	Userid of users who is allowed access
Root Reference	Cap. from which this cap. is derived
Timestamps	Creation, and (optional) expiry times
HMAC	Signature of the cap. with server's key

Table 3: Anatomy of a capability.

stores of other users. We do, however, provide a permissions mechanism which emulates append-style semantics for applications. Given the appropriate permissions, one user may append a reference to an object they own into a list of references in another user's data store.

Fine- to Coarse-Grained Sharing Users can share an individual object, or group multiple objects into collections and then share them all at once.

Unfortunately, the permissions interfaces exposed by existing cloud storage services such as Dropbox, Azure, and S3, are a poor fit for these requirements: they force data to either be open to the public, or to be shared with a named user on the same storage service. Similarly, their access control primitives make it impossible to safely support append-only semantics. Instead we developed and implemented the CAPSI storage interface. This interface is immediately deployable, for example, as an EC2-based service with an S3-based back-end. As storage services have an incentive to attract more customer data, we assume that if Micasa-style applications became popular, commercial storage providers would add CAPSI as a native interface to their service (or alternatively provide access control primitives that allow a compatible client interface to exist).

2.3.1 Capability Servers

With data stores distributed across multiple independent storage providers, as in our model, using authentication and access control lists to implement the sharing of objects between users would require either numerous user registrations (e.g., a user would need an authentication mechanism for each of the storage providers used by his/her friends) or a trusted third party identity provider. Instead, we have chosen capabilities as the basis of our sharing model. This way, access to objects can be communicated over email, or be embedded inside pages, etc.

2.3.2 Structure

In this section we briefly describe the structure of our capabilities, displayed in Table 3.

We define two types of objects: *blobs* (files) and *lists* (folders). Lists allow capability grouping and nesting. Both types can be shared with other users. Sharing a capability to an object will share both the object and all of its descendants. For applications that require different types of navigation, such as searching by tags, lists store application metadata for each entry that can be used to build search indexes.

The objects themselves are either mutable or immutable. Immutable objects are referenced by their content address (content hash). Mutable objects have a unique name on the underlying data store.

Capabilities describe three types of rights over an object: *OWNER*, which is all-rights, *GET*, which is read-only, and *APPEND*, which allows a controlled form of write sharing (only for lists). We found these to be sufficient to build rich applications.

Other fields are used to reduce the scope of capabilities. Limited periods of validity can be set using the expiry time and creation time fields. Also, capabilities contain reserved fields for issuer and audience to determine the owner of an object, and limit access to some users in cases where the capability server can authenticate the requester (see Section 3.1).

A server reference field can be resolved to locate the capability server that manages the object. This indirection allows data migration (Section 2.3.4), and transformation of capabilities into URLs. Lastly, every capability generated by the server includes an HMAC over the properties in the capability tuple, computed with its own secret key, to avoid spoofing.

2.3.3 Data Access

Data in Micasa is represented by capability URLs. These define both the Internet location and access permissions of the data, and have a well-defined format that is understood by all storage providers that support CAPSI.

Figure 2 gives a common example usage of our library, in which Alice is sharing some data with Bob. For illustration, we describe this process in the context of a private chat session between Alice and Bob in TwoCans.

We assume Alice gains access to Bob's chat room page by either navigating tc.example.org, or receiving a URL out-of-band. This page contains a capability to the list of messages in the chat as well as an *APPEND* capability to append to the list. To add a message to this chat, Alice needs to first upload a file containing the message to her storage provider. To do so, she types in her message and submits it, which causes the page to upload

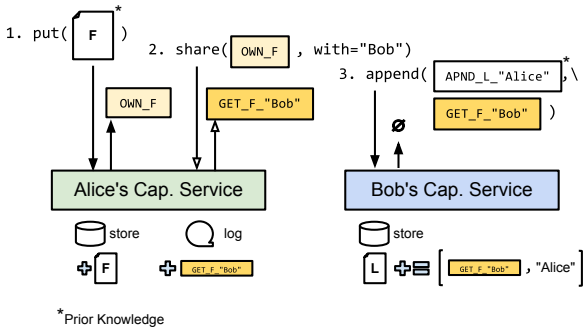

Figure 2: Alice shares data with Bob. In this example, Alice's application is interacting with both Alice's and Bob's data stores. Prior to Alice performing a put for `F`, Bob has a list `L` of capabilities in his store and Alice has a capability `APND_L_"Alice"` to append to that list.

the file (step 1). After the upload, the response from Alice's storage provider is a capability that indicates that she owns the file. An *OWNER* capability is a proof of ownership of that file and gives Alice read-write-delete permissions over it. The application then converts that *OWNER* capability into a form suitable for sharing (with Bob), a *GET* capability (step 2). The *GET* capability created is logged by her CapServ, so that Alice may revoke it or migrate it in the future (explained in Section 2.3.4).

Alice transfers this *GET* capability to Bob, by invoking her *APPEND* capability on Bob's server (step 3). A tag parameter (`''Alice''` in Figure 2), taken from the append capability, is copied to the new entry in Bob's list, so that he may know which target was used to append. The next time Bob reads the chatroom list, he will detect the new message entry, and can retrieve Alice's file directly from her data store using the capability for that file now sitting in the list.

2.3.4 Data Migration

In Micasa, in order to mitigate service provider lock-in, users store their data with an independent storage provider. We emphasize that by doing so, we are *not* trading one form of lock-in for another. Current storage services have a good track record, and we believe they are less likely to go out of business than many other cloud services. Moreover, the services are effectively built to allow users to simply and easily access and retrieve their data.

In cases where data store migration is desired, Micasa is able to do so while preserving access control and availability. To achieve this, capabilities previously constructed by one server must continue working after a data store has migrated. Because the keys used to sign and verify capabilities are private to each CAPSI storage provider, capabilities issued for objects at the original server cannot be verified at the new server. To migrate to a new server, we first copy all user objects, followed by the capability generation log (one entry for each call to `mkget` or `mkappend` where the capability has not yet expired) to the new server. When the new server receives a request with an unverifiable signature, it can check the log for the existence of a matching legacy capability record, and allow access accordingly.

The server reference field in the capability must always be resolved to the active server that stores the object. Once the data and capability records have been copied, the final step in data migration consists of updating the resolved value of the server reference to point to the new capability service. Our prototype server and client use DNS as a resolution mechanism, but we are investigating other approaches based on email addresses and web discovery protocols.

3 Maintaining Service Integrity

For traditional hosted applications, the central servers play a crucial role in maintaining the integrity of the service. For example, they ensure data authenticity, data integrity, and data consistency through validation and sanitizing. Servers also attempt to minimize excessive use of the service (e.g., by bots) or other abusive behaviour. Micasa-style applications do not have the luxury of a central point of enforcement to implement these measures. It is therefore important that the application developer compensate for the absence of a central server by correctly using the tools provided by the library. Below we discuss how this can be achieved.

3.1 Data Authenticity and Integrity

Certain situations require proving that objects are authentic to particular users or that access requests have been issued by specific users.

Micasa capabilities are *open* by default, meaning that simply bearing the capability is sufficient to invoke the rights it describes. An optional tag attribute allows differentiating capabilities to the same object, but cannot be relied upon to tell apart two users bearing the same capability.

While authentication could be provided in Micasa through *closed* capabilities, which require the bearer to *prove* his or her identity before a request is executed (e.g., with passwords or email addresses), it is also possible to authenticate requests and data objects via digital signatures, using cryptographic information stored elsewhere in the application. This does require additional key-management complexity, but is less demanding on

the capability system, and thus the TwoCans application described in Section 5.1 is implemented in this manner.

CAPSI capabilities on their own offer basic support for verifying content integrity. As shown in Section 2.3.2, immutable object capabilities expose the content hash of objects, which can be verified on the client with JavaScript routines or native plugins.[2] Storage providers falsely reporting content can be added to a blacklist and pruned out.

Verifying the integrity of mutable objects is also feasible, but requires information external to the capability. In this case, the library provides the tools to digest content and verify signatures, but the application is responsible for providing the expected values.

3.2 Data Consistency

To ease application development, CAPSI forces all writes to be isolated and serialized per-object. In our implementation, if the underlying data store can only offer eventual consistency on updates, then updates to an object are logged by the capability service until propagation completes.[3] Updates could also use a version parameter to provide a "conditional put" mechanism.

It is possible to use this consistency model to perform more complex transactional operations, as long as all participating processes cooperate. Unfortunately, this is impossible to enforce if users are not trusted. Our framework is therefore limited to unplugged features that can be supported with single-object atomic operations.

3.3 Validation and Sanitization

Applications impose many restrictions on the actions users can take and the data they submit. Coding practices recommend that users' submitted data be validated and sanitized *before* it is persisted. This guarantees that all content on the server satisfies an accepted format.

In unplugged applications however, validating only before submission is insufficient. Because users have full control over their own stores, the data of other users must be verified before it is used. To this end, libeol provides basic common content validators, such as enforcing length bounds on responses and HTML escaping.

The amount of validation needed will depend on the type of application. We have found in practice that non-global uniqueness checks (e.g., a user can only post once on a photo) and chronological checks are simple to implement. Immutability checks (e.g., forbid edits) require chaining digests and are more complicated. Fortunately, expensive validation operations can be short-circuited by memoizing content hashes of objects previously visited.

3.4 User Store Abuse

Users must pay for bandwidth and space on their storage provider, which represents a new system element vulnerable to abuse. While there are no foolproof solutions, capability servers can mitigate certain forms of abuse.

Rate of access can be controlled by the capability service, through rate-limiting or the insertion of CAPTCHAs. In terms of controlling space usage, users need only worry about other users appending capabilities to their lists; capability strings are relatively small (less than 1KiB), and if this became an issue a maximum length on list objects could be imposed.

It is generally the responsibility of the application provider to rid the application of spammy content and fake accounts. To prevent that sort of abuse, unplugged applications will need to rely on client-side databases and spam engines, or third party spam services. Moderators can still flag inappropriate content, but it is the client-side code that would need to filter it out from view.

3.5 Missing User Data

In a centralized hosted service, any revocation or deletion of data by users is mediated by the application. Thus, there will not be any data missing unexpectedly at page build time. In Micasa, however, a user can access and manage data in an application's namespace in his own store via CAPSI out-of-band of the application, such as via a namespace file explorer app. Owners can also revoke access to data that they have previously shared. Capability servers ensure that invoking revoked rights will fail. For the application, this possibility translates into "holes" within pages.

Developers should provide fail-safes for missing content. They should account for the eventuality that content has been revoked, at least at the granularity of the "un-share" operations defined in the application. For instance, if the application allows changing privacy settings on pictures, then access to any picture could be revoked along with all associated information. In this case, the error condition is detected (e.g., 404 Not Found), and the application can replace the image with a placeholder or another visual indicator.

4 Implementation

We have implemented a CAPSI-compliant capability service called CapServ, a client-side library for building Micasa applications called libeol, as well as several applications which will be described later in Section 5.

[2] libeol checksums unencoded network payloads (binary) using JavaScript and XHR Level 2 features.

[3] Propagating updates takes only a few seconds in practice.

The capability service is composed of less than 3K lines of Python. It is run as a python-WSGI application. It supports three storage backends for data objects: POSIX local file system, Amazon S3, and Microsoft Azure. However, in an ideal deployment scenario, the capability service would be implemented directly by the cloud storage provider.

Our client library, libeol, runs in unmodified browsers. There are two subsystems in the library. First, there is the CAPSI subsystem, which is invoked by the application to access capabilities. This alone is written in approximately 3K lines of Java Google Web Toolkit (GWT) code that compile down to around 120KiB of uncompressed obfuscated JavaScript (33KiB compressed). This subsystem also has bindings for web applications written only in JavaScript (i.e., without GWT). The second subsystem is the monitor, which is divided in two parts. The first part, the in-page monitor, runs in the page's JavaScript environment. The second, the external monitor, runs as a Chrome browser extension.

The in-page monitor is loaded at the very start of each page load. Its responsibility is to bootstrap a communication channel between the page and the external monitor, and provide some hooks that the application can use.

As an extension, the external monitor has the privileges necessary to interpose on and audit all network connections. It captures network events that would be otherwise impractical or expensive to capture from the JavaScript environment, namely network requests triggered by embedding objects in the DOM (e.g., image tags). It also presents a GUI to unplug the application (Section 4.1.2). Cross-browser compatibility is future work, and may benefit from a JavaScript sandbox such as TreeHouse [11].

4.1 Before and After Unplugging

In Micasa we leverage HTML5 application cache manifests [19] to identify client-side resources that should be preserved in the absence of the service provider. These application cache manifests can be defined for developers to allow "offline" mode functionality, and can be used as a mechanism to speed up application load times on subsequent visits.

We assume that clients are running our extension when they visit a Micasa-enabled website. Micasa applications are installed to the code cache with the initial visit to a page featuring a special eol marker:

```
<html manifest="man.appcache" eol="true">
```

If the manifest referenced is new to the code cache storage or has changed, then the manifest itself, and all contained resource references are stored. Chrome extensions cannot, as of this writing, programmatically access the browser's application cache contents directly. Our implementation thus introspects the DOM to find the location of the cache manifest and parse its entries. For each entry, a new URL to the corresponding entry in the Micasa cache is created, and a page mapping is updated from original resource URL to cached resource URL. The resources are cached externally to the browser so that they have a longer lifetime than the browser cache.

After unplugging, the local application cache may need to be repopulated from the external code cache. Our implementation redirects web request URLs according to the previously constructed mapping. This is currently achieved with Chrome extensions' webRequest module[4]. Programmatic access to the application cache would also be preferable for this task, because redirects performed with this module are unfortunately not origin-preserving for top-level documents.

4.1.1 Manifest Specification

The HTML5 cache-manifests are defined per-page, not per-domain. However, we assume in our implementation that there will be a single Micasa application per domain name, and that the application will be a "single-page" application. This simplification allows the extension to associate one-to-one applications and domain names, and allows users to unplug applications on demand more easily. We reuse the HTML5 cache manifest syntax and semantics for Micasa applications, except that in order for CAPSI requests to succeed, a wildcard entry must be added to the NETWORK section of the manifest. This entry informs the browser that requests outside the static set of cached resources should be allowed if there is network connectivity.

In addition to the set of static UI resources, the manifest also lists a server IDL file, in JSON format. The purpose of this file is to categorize the requests seen by the application monitor. Entries in this file declare a method name (key), a human-readable description of the operation (an intention), a type label (provider, third party, or CAPSI), and a list of expressions used to match the URLs belonging to the entry. The file is evaluated at load time by the in-page monitor, and its information is communicated to the external monitor.

The external monitor matches outgoing network requests to entries in the IDL file, and will block or allow the connection, depending on the type of the entry and plugged/unplugged status. The default policy when unplugged is to block those requests with type "provider", and those that match no entries (fall through).

The provider may attempt to make bogus claims that certain features are unpluggable, and make them appear to be so via mislabelling. However the monitor provides

[4]Proxies are an alternate solution, but complicate deployment.

an audit trail by logging all outgoing requests. A deception exposed in an audit by any customer risks alienating all customers.

4.1.2 The Unplugged Event

UI controls in our monitor extension toolbar allow the user to disconnect from a service provider on demand. This unplugs the application, switches filtering rules in the monitor, and notifies the running application of the state change. Users could use this control to simulate a disruption to provider services, and test that the application features continue to work as advertised.

After failing a server request due to an unexpected network or server error, applications must determine whether the error is transient, or permanent (unplugged). The application code can trigger the application to unplug on its own, for instance if repeated attempts to reach a server all fail. However, to help the application decide faster, Micasa defines new runtime page events, `unplugged` and `plugged`, that fire according to the current state of the monitor.

Applications can listen for these events to dynamically change their behaviour. For instance, applications could determine if certain features should be presented to the user, or to pick alternate implementations of a particular feature.

HTML5 offline mode [19] defines similar events for cases where the browser is experiencing a network outage. Our unplugged scenario is similar in that the application service appears unavailable, but different because unplugged clients can still rely on the network to perform CAPSI calls, or access third-party services.

5 Evaluation

We wish to create applications that can tolerate permanent disconnection from their provider, but we also want these applications to have good performance, be functional, feature-rich, and practical to build. Those are difficult criteria to meet when considering that the convenience of a central server can be lost. We evaluated the practicality of our prototype by building many different applications, listed in Table 4 with their size in lines of client-side code. Of the list, two will be explained in further detail in the following sections. The last section of the evaluation benchmarks Micasa-model applications on a Flickr-based data set.

App. Name	SLOC	Description
TwoCans	1500	IM System (§5.1).
HotCRP-P	10K	Permanent HotCRP (§5.2).
Lenscapes	2200	Photo album sharing.
Data Viewer	650	Namespace file explorer.

Table 4: Micasa Web Applications and their size.

5.1 TwoCans: Messaging System

Our first application showcases the core functionality provided by our prototype library. We use libeol to implement a multi-reader multi-writer system, a pattern common to multiple online social web applications. This pattern often takes the form of comment lists, page votes, a "friend wall", etc. Multiple instances of this pattern can be duplicated and composed to create pages of arbitrary complexity (e.g., a comment list on a photo, itself inside a shared photo album).

We place this pattern at the core of TwoCans, a multi-party Internet messaging system, written from scratch. It is a conceptually simple application that allows multiple users to exchange text messages in chat rooms. It differs from typical chat programs in that writers retain ownership of the messages they send out. The messages are exchanged between peers using the stores as intermediaries in a peer-to-store fashion. The implementation consists of about 1500 lines of client-side code (excluding libeol), and a small central server implementation of under 300 lines of Python.

To initiate a conversation, users create a list to host the messages. To add a message in the conversation, users first upload message text to their store, then append the resulting capability to the conversation list. The owner of the conversation list can act as a moderator, and revoke access to one of the users. By design, if the owner of a conversation deletes the conversation, the conversation is lost (unless a copy is taken). Also, users can revoke access to their own messages. The TwoCans central server provides discovery of other users and public chat rooms via search. When unplugged, these features go away, but users retain the ability to communicate in chat rooms in which they are already members. Furthermore, inviting users in a chat room can always be done by sharing an invite URL out-of-band.

TwoCans protects against message spoofing via message RSA signatures, generated with utilities in libeol. An author obtains a signature with a libeol function call over the message plaintext, the current timestamp and the unique chat room ID. Signatures are appended along with message plaintext. Other chat members verify them using the author's public key, which is retrieved from the central application server (but could be retrieved from an external service), and cached at the client.

Overall, the TwoCans service consists in a directory of registered users and their public keys, as well as a search index on the subjects of active conversations. Scalabil-

ity of the service relies mostly on the cloud storage providers of the users.

5.2 HotCRP-P: Permanent HotCRP

In this section, we demonstrate that Micasa can support the needs of modern applications, with reasonable developer efforts, by refitting the conference management software HotCRP to work in a Micasa environment. HotCRP-P, our modified application, includes all of the original application functionality, plus the *permanent* ability to search through all reviews and papers the user has ever had access to, regardless of the conference server's availability. The difficulty in this case lies in teaching HotCRP that it can be unplugged, change certain UI flows accordingly, and ensure users can search at all times.

HotCRP consists of about 20K lines of PHP code that mixes data and logic in the page that it renders (like many PHP applications). To change HotCRP into a cacheable web application, we went through the tedious process of extracting UI logic from the server-side HTML generation templates for most of the conference core functionality. The resulting UI resources comprise around 10K lines of JavaScript, 700 lines of HTML, and 200 lines of CSS. Porting an application that from the start embraces Web2.0 further (e.g., no top level page changes, AJAX-heavy) would be easier.

We then modified the login flow in the client to allow connecting the user to a Micasa store, before logging in to the HotCRP server. We modified the paper submission, abstract modification, and review submission flows on the client to work with capabilities (i.e., upload object to store, obtain capability, and submit capability to server). We made corresponding changes on the server so that it stored both the capabilities and a cached version of the full object. Loading these pages as a viewer works in the reverse way: capabilities are fetched from the server via a REST-ful interface (using key parameters such as the paper's id) and objects are retrieved from Micasa stores. If the papers are unavailable, the server's cached copy can be used instead.

We implement the permanent search feature by building a client-side index of page contents, keyed with the paper id of the page being viewed. The index itself is periodically saved to the user's store. Rather than writing our own indexer, we modified Apache Lucene (version 3.3.0) to work in a Java applet. The patch for this is around 1400 lines of Java code. The client stores the capabilities, so that page content can be reconstructed.

The permanent-access feature is only enabled when the application is unplugged. In that mode, authentication is no longer possible. HotCRP-P thus replaces the login flow with a search query flow. After a successful search, corresponding pages can be displayed, and cached capabilities retrieved.

5.3 Client Performance Overheads

We measure the performance impact of migrating from a centralized service to one running in a Micasa environment, in a benchmark modelled after Flickr picture pages. The benchmark consists in rendering picture pages, along with respective user comments, and images embedded in these comments.

We take measurements on multiple picture pages of varying complexity. Pages are representative of a random sample of Flickr picture pages, and were sampled from the "fresh" (recently-uploaded) and "7-day popular" Flickr feeds. The number of comments, the size of the comment text, the images inlined with a comment, the image data, and number of comment authors match the online version of the page in the feeds, and vary across all of the sampled pages. We maintain Flickr's limit of 20 comments shown per page. However, it is not uncommon for comments to embed a variable number of additional images. We run the benchmark on three different versions of the pages, and perform point to point performance comparisons.

The first version is our baseline, a static version of the page with comment text and image tags inlined. That is, all the URLs to images and the comment data are known once the index file is loaded. It simulates an application server that can generate page content instantly (i.e., no page-generation overhead). The server also serves data for all images, scripts, and CSS referenced in the page.

The second implementation uses libeol. In this version, comment and image data are retrieved across CAPSI stores. The base content of the page (HTML, JavaScript, CSS, icons, logos etc.) is devoid of user data. Client-side logic fetches and displays all of the dynamic content in the page from a single top-level capability list for that page. We assign each user involved in the page, i.e., the picture owner and all comment authors, to one store sampled from a population of 10 distinct CapServ stores, according to a Zipf popularity ranking of $1/r$. Users are also assigned a key pair for signing comments. The public key is stored in a key repository on the application server. Individual comments and embedded images are stored as immutable blobs. References to the comments and images are stored inside a list for each comment, and references to these comments are appended to a single comment list owned by the owner of the page. Comments are digitally signed with RSA by including content hashes for comment text, embedded image references, and other fields from the capabilities.

The reference to the appendable comment list, as well as a reference to the main picture is stored inside the top

level list. Overall, the client recurses 3 lists of capabilities to obtain references to all objects to be displayed in the page. To verify signatures on comments written by authors unknown to the client, missing public keys are fetched from the key repository, and cached to avoid future lookups.

The third implementation is similar to the second, but benefits from application server caching. Page loading is sped up by retrieving a cache file from a central application service. This simulates an application that caches capabilities added to the page by users (i.e., whenever a new comment is appended). The cache file contains a "flattened" view of the capability structure of the page. It contains capabilities to the user data (comment blobs and embedded image blobs), but not the data itself. In this version, signature verifications on author comments are skipped because the cache file is assumed to be authoritative. However, object digests must still be computed to match expected values in the cache file.

We run all of the CapServ hosts in 32bit Ubuntu 10.04 micro-instance VMs on Amazon EC2 (west coast), and configure them to store objects on their local block storage. Our client runs Google Chrome Stable 25, on a Ubuntu Desktop 10.04 64bit, with a Core i5 750 CPU (4 cores, no hyper-threading), connected to the university's public network.

5.3.1 Results

The bandwidth overhead for any page and version depend upon the number of comments and data objects. The amount of application data loaded is the same across the three versions (images and comments are the same size), so any additional data transferred comes from capability lists, capability strings referring to data objects, and user keys. The two versions using Micasa have the advantage that client-side code is cached at the client, and need not be downloaded. The bandwidth consumption overhead of the version with caching over the static one is minimal, around 6%. In the worst case, when server caching is unavailable, the extra work involved in retrieving keys and reconstructing the capability structure places this overhead at 23% over static[5].

In contrast with bandwidth requirements, the impact of using Micasa on page load times is more complex to characterize. We compared, pairwise, the total time needed to load all the comments and images on the page. The cumulative distribution of overheads in page load times over the static version is shown in Figure 3. For example, around 80% of all pages without caching have

[5]The median values are within 3% of the means, and we measured bandwidth consumption using Charles 3.6.5 HTTP proxy, with caching disabled in both the browser and proxy.

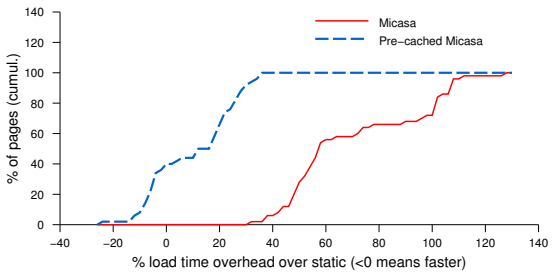

Figure 3: CDF of the percentage load time overhead versus the static baseline, compared on a page-by-page basis.

an overhead of 100% or less over static[6], and similarly, all pages with caching have no more than 40% overhead over static. Results shown are for 50 Flickr pages. For each page and version, we compare median load times over 21 repeated page views. Our sample set's static pages have 72.8 objects to retrieve on average (69.5 median), and have average load times of 1026.8 milliseconds (1006.5 median).

The overheads with respect to the static case are due to a number of factors. First and foremost, they are due to data dependencies that increase the overall time needed to construct pages. In the Micasa version without caching, four levels of capabilities need to be traversed before images and comments can be inserted in the DOM. Three of these levels are already unrolled with caching. Similarly, the caching version does not need to fetch keys and can skip signature verifications.

Generally speaking, capabilities on a same given level can be retrieved in parallel, but not before their parent list has been retrieved and parsed. In practice, this parallelism is subject to per-host-port connection limits, and a global limit set in the browser (16 and 35, respectively, in our experiment).

Second, in isolation, fetching blobs from our prototype CapServ incurs penalties over static file fetches from Apache 2.2.14. The overheads on the server consist of a base cost for capability verification (SHA1-HMAC), the Python language runtime, and a WSGI connector that passes data between the server application and Apache. This is added to the client overheads caused by the libeol invocation, and the client-checksum routine performed over response bodies. Figure 4 shows median overheads of performing `getblob` on various sizes over 200 trials, on a local network. The slowdown is approximately a multiplicative factor of 3 between static and Micasa, and 3 from Micasa to Micasa with client-side JavaScript SHA1 [17].

Third, as mentioned earlier, both Micasa versions benefit from local caching of client-side code.

[6]$(t_{caching}(page) - t_{static}(page))/t_{static}(page)$

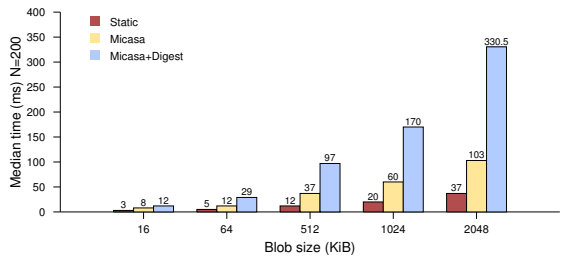

Figure 4: `getblob` overhead.

Fourth, in our tested scenario, some Micasa pages gain an advantage over the static case due to increased data parallelism. In the Micasa cases, the browser can fetch data from up to ten stores based on the number of personal stores involved. Recall that in our static case, a single server delivers content. While we did this to model a startup service, more mature providers deploy multiple servers to improve page load times.

Overall, we find that with all factors combined, our Micasa prototype offers promising performance. Many further optimizations could be applied to improve its performance. For instance, our server could benefit from request batching, or recursive list fetches, and the client digest computation could be moved off the main thread into a worker.

6 Related Work

The idea of separating web application code from data is not new. It is a core concept in Web 2.0, and has been applied for many different purposes. Some use it to solve a form of data lock-in [2, 25], by empowering users to place data in personal stores in the cloud. Unfortunately, their storage models are mostly designed for single-user applications. In the case of unhosted, receiving updates from other users is only possible by granting coarse OAuth write access to a mediating service. They therefore do not address the challenges inherent to preserving social features without a central server, nor do they address building applications that are compatible both with and without the server.

W5 [12] also proposes an architecture allowing users to retain control of their data by separating it from application logic. W5, however, is designed to restrict data flow to providers. This falls outside the scope of Micasa; we focus on EOL functionality.

Menagerie [6] presents a system that allows applications to aggregate data across cloud services, by encapsulating access to objects inside capabilities. In both Menagerie and Micasa, capabilities provide uniform access mechanisms to objects stored on heterogeneous services, and are at the basis of sharing. However, their goal is to aggregate data dispersed on the Internet. It does not allow service continuity without the provider. Our capability-based API alone provides features similar to other existing capability systems [22, 20]. Our system was developed independently, but it appears possible to create a portability layer to support our needed semantics on these other systems.

SPORC [4] and Frientegrity [3] address the problem of having groups of users collaborate on untrusted centralized servers. Whereas our goal is to test that application providers can provide continued application functionality, their main goal is to protect the user's data integrity by detecting misbehaving servers. They rely on developers using operational transformation for merging event conflicts, whereas we allow more familiar ways of programming. Our system's storage organization also resembles Persona [1]'s decentralized storage. They solve the particular problem of protecting data privacy in a distributed private social network context using attribute-based encryption, whereas we build a framework for letting applications disconnect from providers.

Many choose to do away with a centralized service and provide essential functionality inside smaller federated instances of the same application. Two famous examples are Diaspora [23] which replaces a centralized social network with federated "pods" users can join, and OpenPhoto [24], a photo-management and sharing application which allows users to run their own OpenPhoto servers. Users gain some control over their data privacy because they can choose the servers that will host it. However, their APIs are specific to a single application. Running multiple different applications with this design would require trusting and maintaining a different server each time. Our APIs on the other hand can generalize to multiple applications, and can benefit from the convenience of a central server.

7 Future Work and Discussion

As the Micasa framework adds complexity to an application, it is natural to ask what incentives developers have to use it. We believe the answer starts with users. If users are particularly interested in certain functionality, developers have an incentive to provide it. Micasa is motivated by the belief that many users would find features that mitigate against lock-in and EOL very appealing, and conversely, that users find the lack of such mitigation a deterrent to investing their time and data.

In a Micasa application, the capital and operational costs of providing the service are effectively distributed across the service provider and selected storage providers (i.e., those hosting content for users registered

for that service). Many new providers, and even some who run at scale already find it economical to leverage online storage in this fashion, for example, Netflix stores its entire movie library on S3. Furthermore, Micasa enables a new range of monetization strategies. For example, a service provider may have partnership agreements with select storage providers and flow user fees and ad revenue through these partnerships. After a user's storage provider is specified, an application could ensure that it be given a frame for serving an advertisement.

In our current prototype we require storage providers to export the CAPSI API. In our prototype, we demonstrate how to fulfill this requirement using an EC2 instance provisioned with suitable storage. Moreover, given that storage providers have an incentive to drive new business, if Micasa-style applications became popular, there would be an incentive to adopt the CAPSI API. Nonetheless, there is an obvious bootstrapping problem.

As part of our current efforts, we are working in two directions. First, we are exploring protocols for emulating the CAPSI API using the existing bucket and access control mechanisms on S3. We intend for the additional complexity on the client side to be encapsulated as part of the Micasa library. More generally, we assume that some diversity among distinct storage providers is to be expected and we are currently exploring a more flexible model. In this model, the library and monitor would support a number of storage APIs. Applications would interact with the "application-side" of the monitor using a generic application API. The "storage-side" of the monitor would be capable of interacting with any of a number of supported storage APIs. The monitor would translate between the application API and a particular storage API based on the domain of the client's storage provider. Even within this model, certain minimal requirements, beyond those currently fulfilled by current storage providers, may need to be fulfilled by any Micasa-compatible storage API in order to achieve both abuse prevention and proper access control.

Although we have not emphasized the following up until now, Micasa provides users with more control of their own content than in current centralized applications. In current apps, when a user deletes some content, she cannot be sure that well-crafted `get` commands by other parties will not retrieve the content. In contrast, in Micasa a user can revoke any or all of the capabilities for an object or delete the object on his store by virtue of his owner permissions. With respect to this added control we make three points. First, such added control may be appealing to users and used as a marketing tool by Micasa applications. Second, we emphasize that Micasa does not preclude an application provider from caching or mining user data in any form they like. In this sense, we do not believe Micasa deprives application providers from existing monetizing avenues, such as advertisement, or performance measures.

Finally, and conversely, the Micasa architecture may in fact provide the means for protecting the privacy of user data from the service provider. We are currently building cryptographic and key management APIs into the Micasa library to allow for confidential sharing among user defined groups. As future work we will explore practical techniques for the Micasa monitor to police against exfiltration of content and keys to the provider. Such privacy-preserving apps could still allow for provider-side caching of encrypted objects. Services could still be supported with untargeted ads. Users may be willing to directly pay the incremental costs between targeted and less targeted ads for the added privacy. Whether they are willing is a market question, one which our architecture allows developers to explore.

8 Conclusion

In this paper we introduce the Micasa architecture for writing applications which treat service provider EOL as a recoverable failure mode. The model assumes an ecosystem of storage providers that export a common API and a client-library for writing applications that adhere to this API. Micasa users cache client-side application code and upload their content to personal stores rather than to the service provider directly. While the service provider is in business, Micasa applications enjoy the benefits of existing centralized services, but if the provider discontinues service for any reason, Micasa clients can continue to use the application in an unplugged mode. When unplugged, Micasa clients retain the functionality and data access promised by the developer in perpetuity. Micasa also enables users to audit the developers adherence to the expected unplugged behavior at any time.

We demonstrated the feasibility of the architecture by building two applications, each with different data sharing characteristics. In addition, we emulated the load times that Flickr pages would have if delivered as a Micasa application, and found the overhead of the numerous `HTTP` fetches acceptable due to the concurrency afforded by modern web browsers and the possibility of future speed improvements for client-side computations. Hence, a provider building applications with the Micasa platform can offer minimal EOL risk to new users while still being able to deliver features and performance comparable with existing centralized services.

Acknowledgements We thank Brendan Cully and Jake Wires for their many contributions, our shepherd Roxana Geambasu and reviewers for their feedback, and NSERC ISSNet for their support of this work.

References

[1] R. Baden, A. Bender, N. Spring, B. Bhattacharjee, and D. Starin. Persona: an online social network with user-defined privacy. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, SIGCOMM '09, pages 135–146, New York, NY, USA, 2009. ACM.

[2] R. Chandra, P. Gupta, and N. Zeldovich. Separating web applications from user data storage with BSTORE. In *Proceedings of the 2010 USENIX conference on Web application development*, WebApps'10, pages 1–1, Berkeley, CA, USA, 2010. USENIX Association.

[3] A. J. Feldman, A. Blankstein, M. J. Freedman, and E. W. Felten. Social networking with frientegrity: privacy and integrity with an untrusted provider. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 31–31, Berkeley, CA, USA, 2012. USENIX Association.

[4] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. Sporc: group collaboration using untrusted cloud resources. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–, Berkeley, CA, USA, 2010. USENIX Association.

[5] G. Fox. Yahoo Sets the Date of GeoCities' Death. `http://www.pcmag.com/article2/0,2817,2350024,00.asp`.

[6] R. Geambasu, C. Cheung, A. Moshchuk, S. D. Gribble, and H. M. Levy. Organizing and sharing distributed personal web-service data. In *Proceedings of the 17th international conference on World Wide Web*, WWW '08, pages 755–764, New York, NY, USA, 2008. ACM.

[7] Google. Chrome Apps. `https://chrome.google.com/webstore`.

[8] Google. Google Takeout. `http://www.google.com/goodtoknow/manage-data/takeout/`.

[9] Google. Official Blog: A fall sweep. `http://googleblog.blogspot.ca/2011/10/fall-sweep.html`.

[10] Google. Official Blog: A second spring of cleaning. `http://googleblog.blogspot.ca/2013/03/a-second-spring-of-cleaning.html`.

[11] L. Ingram and M. Walfish. TreeHouse: JavaScript sandboxes to helpWeb developers help themselves. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 13–13, Berkeley, CA, USA, 2012. USENIX Association.

[12] M. Krohn, A. Yip, M. Brodsky, R. Morris, M. Walfish, et al. A world wide web without walls. *Proceedings of HotNets-VI, Atlanta, GA*, 2007.

[13] S. Lohr. Google to End Health Records Service After It Fails to Attract Users. `http://www.nytimes.com/2011/06/25/technology/25health.html`.

[14] J. Mickens, J. Elson, and J. Howell. Mugshot: deterministic capture and replay for javascript applications. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[15] Mozilla. Web Application Manifest Format and Management APIs (W3C Editor's Draft). `http://mozilla.github.com/webapps-spec/`.

[16] S. Musil. Instagram competitor PicPlz to shut down in July. `http://news.cnet.com/8301-1023_3-57446282-93/instagram-competitor-picplz-to-shut-down-in-july/`.

[17] J.-C. Sirot. jcsirot/digest.js. `https://github.com/jcsirot/digest.js`.

[18] W3C. Cross-Origin Resource Sharing. `http://www.w3.org/TR/cors/`.

[19] W3C. HTML5 Offline Web Applications. `http://www.w3.org/TR/2011/WD-html5-20110525/offline.html`.

[20] Z. Wilcox-O'Hearn and B. Warner. Tahoe: the least-authority filesystem. In *Proceedings of the 4th ACM international workshop on Storage security and survivability*, StorageSS '08, pages 21–26, New York, NY, USA, 2008. ACM.

[21] Why Can't I Restore My Twitter and Facebook Data? `http://blog.backupify.com/2011/01/12/why-cant-i-restore-my-twitter-and-facebook-data/`.

[22] Camlistore Project Homepage. `http://camlistore.org/`.

[23] Diaspora Project Homepage. http://diasporaproject.org/.

[24] OpenPhoto Project Homepage. http://theopenphotoproject.org/.

[25] Unhosted: Freedom from web 2.0's monopoly platforms. http://unhosted.org.