freedom.js: an Architecture for Serverless Web Applications

William Scott, Raymond Cheng, Arvind Krishnamurthy, Thomas Anderson
University of Washington

ABSTRACT

Delivering web software as a service has grown into a powerful paradigm for deploying a wide range of Internet-scale applications. However for end-users, accessing software as a service is fundamentally at odds with free software, because of the associated cost of maintaining server infrastructure. Users end up paying for the service in one way or another, often indirectly through ads or the sale of their private data.

In this paper, we aim to enable a new generation of portable and free web apps by proposing an alternative model to the existing client-server web architecture. freedom.js is a platform for developing and deploying rich multi-user web apps, where application logic is pushed out from the cloud and run entirely on client-side browsers. By shifting the responsibility of where code runs, we can explore a novel incentive structure where users power applications with their own resources, gain the ability to control application behavior and manage privacy of data. For developers, we lower the barrier of writing popular web apps by removing much of the deployment cost and making applications simpler to write. We provide a set of novel abstractions that allow developers to automatically scale their application with low complexity and overhead. freedom.js apps are inherently sandboxed, multi-threaded, and composed of reusable modules.

We demonstrate the flexibility of freedom.js through a number of applications that we have built on top of the platform, including a messaging application, a social file synchronization tool, and a peer-to-peer (P2P) content delivery network (CDN). Our experience shows that we can implement a P2P-CDN with 50% fewer lines of application-specific code in the freedom.js framework when compared to a standalone version. In turn, we incur an additional startup latency of 50-60ms (about 6% of the page load time) with the freedom.js version, without any noticeable impact on system throughput.

1. INTRODUCTION

In the last decade, we have seen an incredible evolution of web technologies and the way software is created, distributed, and consumed. Software as a service (SaaS) allows software developers to provide complete fully-featured applications over the Internet, accessed by users using a thin client and often just a web browser. Client-side code in this model is generally only responsible for presenting responsive user interfaces, basic offloading, and funneling events between the server and user. This is in stark contrast to native software which is sold, licensed, or given away for free to run entirely on an end user's machine. The portability of web apps[1] has turned the Web into one of the most successful write-once-run-(almost)-everywhere platforms for software.

However, treating software as a service has profound implications on the nature of software applications. The majority of application logic is run on servers controlled by the software provider, which means that it is impossible for users to change or even to determine what the software does. Service providers have a unique ability to modify their software at any time and leverage their users' data at will. Users pay for this server infrastructure through their private data, advertisements, or purchases, many times unknowingly.

In this paper, we introduce an alternative model to the datacenter-centric cloud model now prevalent in industry. In freedom.js, web apps are written to run entirely in the browser and in many cases do not require a server presence at all. Instead, back end application logic is performed on the client where it can directly interact with the user interface (UI) in a manner that is consistent with existing UI frameworks. In order to match the portability of existing web apps, our framework is built in the language of the Web, JavaScript, such that freedom.js apps can run in unmodified browsers without installing any native software. Ultimately, freedom.js apps look no different to the end-user than existing web apps, but we believe that the benefits of writing applications in the freedom.js framework (privacy, low barrier to entry, resilient connectivity, and composable services) are desirable properties to both developers and users.

Building serverless web apps is a challenging premise. Modern web apps depend on servers for a variety of purposes, such as highly available access to persistent data, centralized configuration, and security. In order to redesign web apps without dependence on servers, this functionality would need to be rewritten to function entirely on the client. However, client-side JavaScript lacks the centralized world view and many of the fundamental primitives needed to provide useful services. Much like operating systems, browsers provide local abstractions like

[1]Following modern convention that 'application' and 'app' are interchangeable, we use 'app' throughout for consistency.

persistent storage to a hard drive, but lack higher-level abstractions that facilitate multi-user multi-device interactions needed by modern web apps. Because browser development has been driven by client-server applications, these are abstractions that browsers are not motivated to provide by themselves. Further, developers often organize client-side code by simply including existing scripts, a model that is reminiscent of statically linking libraries into a global address space with a single thread of execution. Studies have shown that there are limitations to code complexity [32] and security [24] when web apps are written in this fashion.

By tackling these challenges, we hope to once again enable free software in the age of the Web, where users can freely download, modify, and execute the code for a multitude of interactive web apps. The current model makes the developer responsible for scaling their service to handle customer traffic, an expensive proposition. Developers are often pressured to convert to a commercial model earlier in the development process than they might otherwise, with obvious consequences to end-user privacy and encouraging walled gardens. As one of the few exceptions, Wikipedia spent over $1.8 million in 2011 for server operating costs, all supported by donations [7]. In the freedom.js model, developers can write and distribute free software without worrying about how to maintain and scale application infrastructure. freedom.js applications are designed to naturally self-scale with the number of users accessing the application. By doing so, we hope to lower the barrier to entry and foster a healthy community of new user-supported free web software, in the spirit of the Linux community.

In order to achieve this goal, freedom.js exposes new abstractions for client-to-client resource usage. Browsers already provide the underlying facilities needed to build such an alternative architecture. In particular, new HTML5 standards provide mechanisms to store large quantities of data, and WebRTC provides a mechanism for direct network communication between two instances of a web app. Using these primitives, freedom.js provides a number of fundamental building blocks for free apps, including replicated storage, user rendezvous, and network transport. In order to promote secure and reusable code, our platform (a) decomposes complex web applications into separated modules of functionality, (b) provides a secure runtime for executing modules concurrently in isolated sandboxes, (c) dynamically links shared modules, and (d) offers a number a vital primitives to support the breadth of applications we are used to today. Fundamentally, freedom.js represents a new way to think about how web applications are structured and executed, treating the browser as a modern networked operating system.

We have built a number of applications on top of freedom.js, including a messaging application, a social file synchronization tool, a P2P content delivery network (CDN), and an anti-censorship overlay network. Through these applications, we demonstrate that freedom.js apps are responsive and achieve acceptable performance. We also compare two implementations of our P2P-CDN, one using freedom.js and one without, and show that our platform incurs a small overhead in startup latency in return for drastic reductions in code complexity. We were able to implement our P2P-CDN on freedom.js with just 311 lines of code, 50% smaller than our stand-alone implementation. In practice on a standard laptop, we experienced just 50-60ms of additional startup latency with the freedom.js version (about 6% of the page load time) without any noticeable impact on system throughput.

In the rest of this paper, we illustrate the following contributions:

- We designed the freedom.js platform to make it easy for developers to write rich multi-user web apps that run entirely in the browser.
- We demonstrate that it is possible and practical to achieve client-only solutions for a number of web apps. We implemented a number of applications on top of freedom.js and discuss how existing applications can be built in this new programming model today.

We next explain the goals of freedom.js (Section 2) and its design and implementation (Section 3). We then describe our representative applications and evaluate their performance (Section 4). We finish with a discussion of future work (Section 5), and how freedom.js compares to existing systems (Section 6).

2. BACKGROUND

The freedom.js approach to building web applications has the potential for high scalability and low cost in comparison to conventional web services. By revisiting the underlying architecture of web app design, we can offer new answers for reliability, extensibility, and transparency. We designed freedom.js to meet the following goals:

Simple and Extensible: A primary goal of freedom.js is to make applications simple to write and deploy. By adopting the web browser as the common medium, freedom.js apps are immediately portable across a wide range of operating systems and devices. Web developers can use their existing knowledge of JavaScript to develop applications, leveraging the Web's built-in security model, rich markup language, and direct compatibility with existing web services. In the spirit of JavaScript, our system is as extensible as the applications it enables, allowing developers to extend freedom.js to suit their needs. The abstractions provided by our system encourage the

use of or standardization on a set of low-level interoperation standards, ultimately leading to simpler and cleaner application code. freedom.js also fits into the model-view-controller model of building applications, allowing many existing web applications to maintain their current client-side code when migrating to freedom.js. By removing the requirement that web developers host their own web apps, we further lower the barrier to entry.

Composable applications: The current web architecture does not lend itself to ubiquitous web app composability. Existing web stacks are large vertical silos, where the presentation and data are closely intertwined in server-side application logic. Browsers enforce the "same origin policy" of isolation, which has pushed web APIs up to the application level such that each service controls a custom Javascript API. System-level abstractions such as pipes are fundamentally at odds with the web's current architecture. freedom.js imposes a new model of building applications that facilitates code sharing and reuse. Developers are encouraged to write applications as a collection of small modules that each encompass a piece of functionality. Applications are then assembled with the support of many modules, mirroring the use of third party libraries in server applications. For example, our anti-censorship tool and file-sharing applications use the same network transport implementations. If a file-sharing application develops a new network transport with improved performance, the anti-censorship tool immediately sees the improvement as well.

Make scaling automatic: In existing web services, web developers use expensive and complex datacenter configurations to provide scalability and reliability. They are responsible for provisioning resources to meet demand and respond to service disruptions. freedom.js promotes a different model where instead of writing a central application with access to global state, applications must be decomposed into per-user behavior that runs locally on the client's machine. Thus, applications naturally scale as each participant contributes sufficient resources to handle their usage costs.

Give users more control: Existing web apps are an all-or-nothing proposition. If a user wants to use the application, they inherently agree to let the service do what it wants, including use an arbitrary number of third-party delegates. Because the application logic of freedom.js apps runs entirely on the client's machine, our model gives clients the ability to define fine-grained security policies for data management, hardware access, and system services to constrain misbehavior. We also give DIY users the ability to replace or extend arbitrary modules in freedom.js apps. For example, an advanced user could modify how and where their data is stored. While this could potentially break expected application behavior, we believe it is important to give users the ability to control the applications they are powering.

While freedom.js enables a multitude of new applications on the Web, it is important to note that it is not a complete replacement for datacenter services and not all web apps can be written in our platform. We focus on multi-user applications that do not need to face the physical world and do not require computation on data across multiple users. For example, shopping websites (with physical fulfillment centers) and VoIP to PSTN gateways still require web servers for those interfaces. Similarly search, multi-user analytics, and collaborative filtering are hard problems to solve in a distributed setting. While we do not claim that freedom.js provides new insights to these issues, we make it easy to hook into existing web services. We believe that freedom.js encompasses a large, high impact space, and has the ability to replicate functionality from websites like Facebook, Twitter, Wikipedia, YouTube, Blogger, and Craigslist, as well as to enable previously unseen functionality.

3. DESIGN AND IMPLEMENTATION

freedom.js was designed for easy deployment and to fit with users' existing perceptions of how web apps operate. As such, users access freedom.js compatible applications identically to any other web service. freedom.js apps can be written as websites on a web server[2] or as packaged apps distributed on the Chrome WebStore or Firefox Marketplace. freedom.js apps are particularly well suited for 'offline' usage. Moreover, the look and feel of freedom.js applications is no different from traditional web apps, and developers are encouraged to use their favorite visual toolkits to create user interfaces. freedom.js modules are designed to fit easily into most existing JavaScript user interface frameworks.

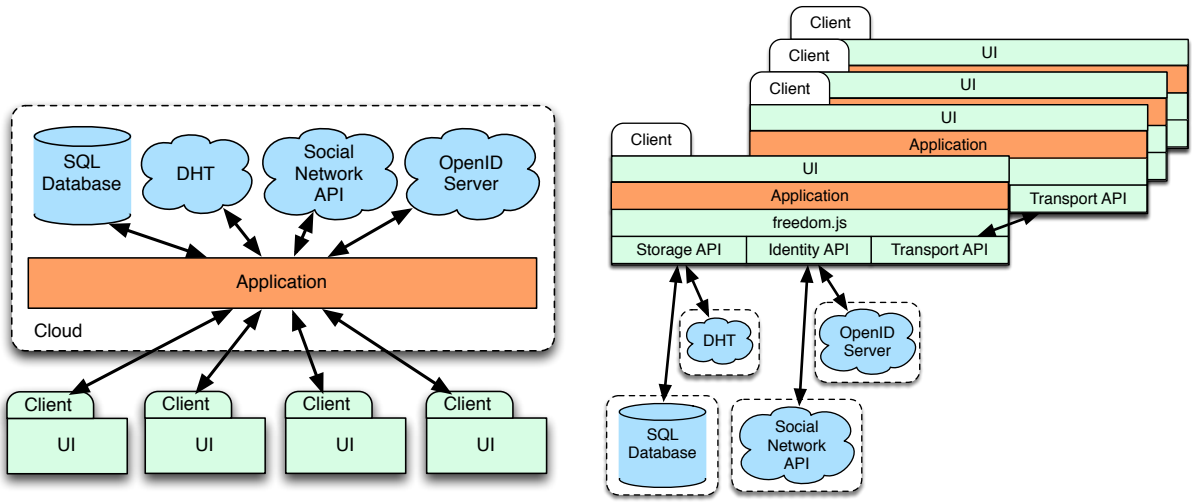
While no limits are imposed on the user interface, the global structure of freedom.js applications will look fundamentally different from existing cloud-based apps. In the next few sections, we describe how an application is structured and how development occurs in freedom.js.

3.1 System Model

3.1.1 Overview

In existing cloud-based frameworks, developers have an immense amount of flexibility in how they write their applications. As long as the server speaks HTTP to the client and delivers a valid HTML document for the client to render, the developer can run any server software stack

[2]The HTML5 Application Cache allows hosted applications to take advantage of freedom.js's benefits even in the face of an overloaded server.



(a) Cloud-based applications currently place most application logic on web servers with access to global state which directly interact with other services. Client browsers usually only deal with rendering the user interface.

(b) freedom.js applications are browser-centric. Each application contains the logic for a single user and accesses low-level services, such as storage, through the freedom.js API. Each browser communicates with each provider individually.

Figure 1: A comparison of the global architecture of cloud-based apps and freedom.js apps

they want. Figure 1a shows the layout of a typical cloud-based web application. The majority of application logic resides on cloud servers, which act as a single centralized instance with a global view of system state. The client is responsible only for visual presentation. The servers can run on any software stack that it chooses (e.g. Linux/Apache/PHP) and the servers communicate with other servers to provide necessary components. For example, the application may communicate with a MySQL server or a Cassandra cluster for access to persistent storage. The application may communicate with OpenID servers or other social networks to identify users. Each of these interfaces is defined by the web service, which makes it difficult to switch between services, such as in the case of migrating data between different databases.

In contrast, freedom.js applications are written in the language of the web, JavaScript, and are confined to the APIs provided by freedom.js and the browser. Each application runs locally on the client machine and interacts with the freedom.js platform using a defined set of APIs. Applications are restricted from accessing other services outside of these APIs. The platform provides APIs for manipulating freedom.js modules, persistent storage, network transport, social identity, visual output, and task management. All of these APIs can be implemented by a variety of *providers*. We define a provider as a freedom.js module that declares that it implements a particular freedom.js API. For example, the transport API could be backed by a module that used the WebRTC standard or through privileged network sockets. The freedom.js platform includes a set of built-in providers, and developers can bundle their own providers for use with their application. Figure 1b shows the layout of a typical freedom.js application. Note that while the application logic resides entirely on the client, providers may include a server component. For example, one built-in identity provider communicates with XMPP services[3] to retrieve a user's list of friends. In many cases, freedom.js applications may not require any server components at all.

3.1.2 System Components

A freedom.js app is composed of a number of isolated freedom.js modules. Modules are instantiated at load time, and are not dynamically created or destroyed. Each module is defined by a manifest, as shown in Figure 2b. The manifest file contains the module's name, description, version, script location, required permissions, and a list of dependencies. Thus, running modules form a tree of dependencies, with a communication channel between each parent-child pair. Our manifest convention is common for JavaScript applications, and compatible manifests are used to describe Node.js libraries, CommonJS packages, and Chrome applications.

When a freedom.js app is loaded, each node in the module dependency tree is instantiated and executed in a sandboxed web worker thread. Web workers are a browser feature allowing for multiple concurrent threads of execution to execute simultaneously in isolation. In particular, web workers do not provide access to the DOM, and can be restricted to prevent other external communications. In other words, there is no shared state and each

[3]XMPP is the standard instant messaging protocol.

```
<html>
...
  <script type="text/javascript"
      src="freedom.js"
      data-manifest="manifest.json">
  </script>
  <script type="text/javascript">
    var sendButton = document.getElementById('send');
    sendButton.addEventListener('click', function() {
      window.freedom.emit('send');
    }, true);
    window.freedom.on('recv', function(msg) {
      var messageLabel = document.getElementById('
          message');
      messageLabel.innerHTML = msg;
    });
  </script>
...
</html>
```

index.html

(a) Initializing a freedom.js module involves specifying an entry point manifest file. The outer script can then communicate with the module using message passing on freedom.emit(...) and freedom.on(...).

```
{
  "name": "FreeChat",
  "description": "A Secure, WebRTC based chat client",
  "version": 0.2.1,
  "main": "chat.js",
  "dependencies": {
    "identity": "bundledIdentity/manifest.json"
  },
  "permissions": [
    "identity",
    "transport"
  ]
}
```

manifest.json

(b) Each module must have a manifest file, which describes script location, module dependencies, and needed permissions.

Figure 2: Integration of freedom.js code into a webpage

module runs concurrently in a separate isolated JavaScript runtime. If a module requests a permission, such as "transport", it is given a transport object, which it can use for API calls. Modules may also claim to implement an API by specifying a "provides" manifest key.

Development begins by adding the *freedom.js* library to an applications source code directory, or linking to a known freedom.js implementation—the same process as any other JavaScript library. As shown in Figure 2a, the HTML page must include the freedom.js library script and point towards the root module manifest file. There can only be one root module manifest file, which serves as the root module in the dependency tree. The page can communicate with the root module by calling *window.freedom.emit(...)* and *window.freedom.on(...)*. The *emit/on* event syntax is a common JavaScript convention found in Dojo, Node, and YUI. In the chat example shown here, the page listens for "recv" events and displays incoming messages in a message label. When the

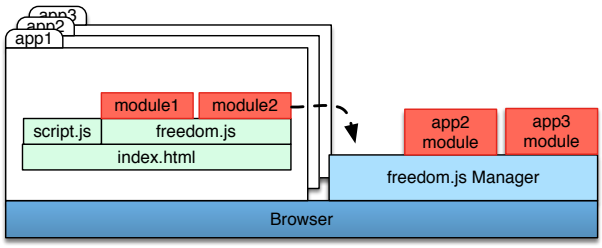


Figure 3: Modules run in sandboxed web workers, which can interact with the outer HTML page using message passing. The freedom.js Manager contains an identical execution environment for freedom.js modules, augmented with additional privileged providers. The arrow indicates the installation of an unprivileged module into the freedom.js Manager.

"send" button is clicked, the current message is emitted to the root freedom.js module.

Figure 3 shows a diagram of how freedom.js code lives and executes in a web app. By default, each API is backed by a provider that can run entirely in the context of an unprivileged webpage. For example, the default storage provider leverages LocalStorage, which allows websites to store up to 5MB of persistent data. The default transport provider uses the WebRTC API to provide direct peer-to-peer channels. These are capabilities that any standard website can utilize. However, an application may desire access to privileged APIs such as a storage provider with larger storage quotas. To address this scenario, we introduce the freedom.js Manager. freedom.js Manager is a browser extension that does not contain any native code and runs within the browser's security model. When a freedom.js application needs access to higher privileged providers, it checks to see if the freedom.js Manager has been installed. If not, freedom.js displays a UI to ask users to install the extension, which only needs to be installed once per client machine, regardless of the number of freedom.js applications. When the freedom.js Manager is installed, trusted freedom.js modules are executed in the extension's runtime environment, and the *freedom.js* client library becomes a proxy for message passing to the now migrated module. Each module is versioned and updated upon access by an updated application.

Once installed, freedom.js Manager gives application modules access to a larger set of providers. These providers generally rely on access to privileged extension API's such as larger storage primitives, web requests to arbitrary hosts, and raw network sockets. Some of the privileged providers that we have built include an XMPP client for identity, TCP sockets for transport, and a non-space-constrained key-value store for storage. Develop-

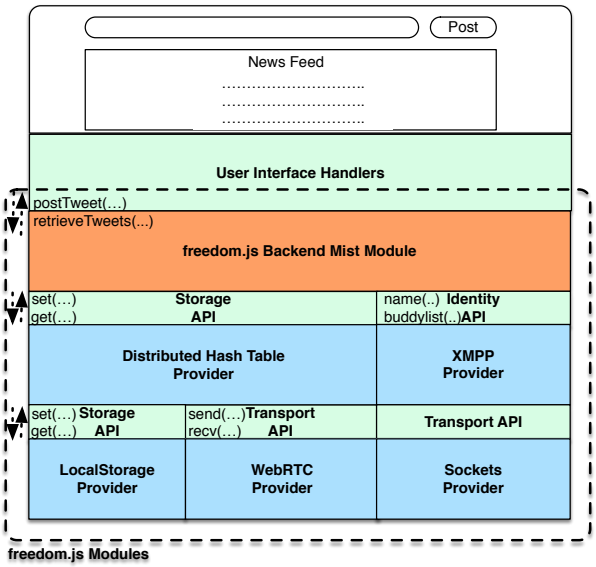


Figure 4: The layout of a microblogging service on freedom.js. Modules compose to create higher-level services. We use a local storage provider and peer-to-peer transport to support a DHT module. The microblog back end then uses the DHT in conjunction with an XMPP client to power the user interface.

ers must request the proper permissions from the user for access to privileged providers and users are given the ability to deny the request for any particular API or provider. In this case, freedom.js automatically degrades service to an unprivileged provider. Additionally, modules installed to freedom.js Manager have the ability to run even when the user closes the application using them. This is highly valuable for providers that rely on peers for availability and notifications.

This division of labor incentivizes participation from both developers and users. Developers are incentivized to use freedom.js in order to lower their operating costs and gain access to peer resources. freedom.js also gives developers access to APIs to build novel applications not otherwise possible on the web, such as privacy-preserving services and anonymizing overlay networks. On the other hand by installing freedom.js Manager, users are given direct control over how their resources are used with a permissions model defined in Section 3.4 and access to a novel set of services.

3.1.3 A Microblog Example

Consider a microblogging web app, such as Identi.ca or Twitter. Figure 4 describes the layout and module interfaces for such an app. A developer would program the front end HTML/CSS/JS in the same manner as they would now using a model-view-controller framework. However, instead of syncing the data model to the cloud using HTTP requests, we communicate with the root freedom.js module that runs the microblog back end, which exports a similar interface as a web server would (*postTweet(...), retrieveTweets(...), etc.*).

The microblog back end module interacts with an identity provider that stores the user's identity and a list of identities to follow. The microblog also interacts with a highly available distributed hash table (DHT) storage provider to store tweets publicly and retrieve other users' tweets. The DHT provider in turn further uses the LocalStorage storage provider and WebRTC transport provider to implement a DHT such as Kademlia or Chord.

In the case where freedom.js Manager is not installed, the LocalStorage provider would be limited to just 5MB of storage per user. Thus, the total capacity of the DHT is limited. Since the uptime of DHT peers is limited to the amount of time that a user has the tab open, additional replication is required. When freedom.js Manager is installed on a client, the LocalStorage provider gains access to a larger amount of storage and the DHT module can run even when the application is inactive, improving capacity and robustness of the application. The developer can incentivize use of the freedom.js Manager by offering discriminating features based on the resource contributions of the user.

3.2 Module Management

In order to support freedom.js as a broadly applicable framework, freedom.js must offer isolation between components such that applications behave correctly, even when they are accessed in a web browser with no trust of either the application or framework. We opt to work with the existing web model, which allows freedom.js to take full advantage of the constrained design space, and use HTML5 message passing, web workers, and sandboxing to allow multiple modules to coexist peacefully. To achieve this, we implement solutions for launching modules, event handling, installation of applications, and a design for long running tasks.

3.2.1 Launching Modules

Modules are designed to be inherently isolated with only message passing mechanisms to communicate between modules and the outer page. This design allows for both reuse and run-time swapping of freedom.js providers like storage and rendezvous. In our framework, each module is globally identified by the canonical URL of the respective manifest. In the manifest, a freedom.js module can describe any other components it depends on. These dependencies may be described in one of two ways: as a 'static' dependency, or as a 'dynamic' permission. Static dependencies describe a specific other manifest, which will be executed in a separate sandbox,

but will be accessible through a generic message passing channel. The two code bases are assumed to cooperate, and will be treated as having the same level of permissions - freedom.js requires an application request all permissions used by its dependencies. Dynamic permissions, in contrast, interact not through an arbitrary message passing interface, but through a freedom.js defined API. This API describes the semantics of methods, events, and properties of an object exposed to the caller, which must be implemented by a provider. Permissions for a provider accessed through a defined API may be higher than those of a consuming application, and will be presented to the user separately. Some examples of freedom.js defined APIs are storage and P2P transport. For both static and dynamic dependencies, all processes required for a freedom.js application to execute are defined at the start of execution by the manifest, and each code module is loaded into a separate, sandboxed, web worker.

3.2.2 Events

In order to allow freedom.js modules to communicate with each other, a `freedom` object is provided to each module. This object exposes channels to dependent modules. Each channel provides a bidirectional JSON transport between modules, and upon this abstraction we offer an event metaphor for code convenience. For dependencies on freedom.js API providers, a predefined interface is provided, and the freedom.js platform filters channel messages to conform to that interface. This does not prevent misuse of the interface, but serves as an encouragement to use the defined API, in order to promote composability.

3.2.3 Installation

The freedom.js library alone does not provide any additional access to hardware to an untrusted web application. However, designing applications in this way provides a natural mechanism for providing a fine-grain trust model for applications with freedom.js Manager. Browsers have previously provided only a coarse-grained trust model for web content: either navigate to a page and assume no trust of the content, or install a browser extension through a high-friction process. In contrast, freedom.js aims to allow for smaller, commonly desired, granules of trust to be allocated through a light-weight mechanism. As such, users can delegate specific freedom.js modules access to additional permissions, such as allowing access to additional storage to a storage provider, by installing it to the freedom.js Manager.

The freedom.js Manager polices this privilege escalation. This extension exposes its presence to freedom.js applications, and can spawn web worker tasks for privileged providers. Using existing browser security mechanisms, and in particular the use of sandboxed frames to drop permissions within the extension, we can ensure that untrusted code code run within the extension is not granted permissions besides those assigned by the user. Trusted APIs granted to a freedom.js module are exposed via a channel on the `freedom` object identical to those of other predefined APIs. This design allows the Manager to only be installed once, and specific freedom.js applications and providers can then be granted additional fine grain permissions through UI reflective of the danger associated with those permissions.

3.2.4 Module Lifetime

While an application is open, all modules are guaranteed to be instantiated and available for communication. Module lifecycle outside of active use is designed in the spirit of mobile applications and existing browser extensions. If an application has been installed to the freedom.js Manager, but is not accessed by any active application, it may be shutdown to free resources. The developer may listen for *load* and *unload* events to handle state and maintain consistency. An installed module that has been shut down can be awoken by one of three triggers

- The user opens an application using the module.
- An alarm scheduled by the module fires.
- The module receives a message from a dependent module, such as the notification to the DHT module when a peer tries to retrieve an element.

Thus, the developer does not need to reason about the memory consumption of modules. The freedom.js Manager automatically allocates memory and instantiates modules as necessary for the liveness of any freedom.js application.

3.3 System Calls

Each freedom.js module can request the permission to use a particular API in its manifest. The system calls accessible to modules are divided into three APIs: identity, storage and transport. Figure 5 enumerates the currently supported system calls. In Section 5 we discuss future APIs that will be added to the freedom.js platform.

3.3.1 Identity Management

The identity provider serves two purposes. It provides mechanisms for a user to manage its own identity and social network. It also provides a low-bandwidth unreliable message passing mechanism. For example, an XMPP service fits well into this criteria. Users have a global identifier and a list of friends. Users can also pass messages between friends in real-time, which makes the system useful as a rendezvous point. An application may choose to aggregate multiple identity providers to help

Social			Storage			Transport		
Name	Type	Description	Name	Type	Description	Name	Type	Description
login	method	Login to network	clear	method	Clear store	open	method	Open connection
getProfile	method	Get user profile	get	method	Get key	send	method	Send data
sendMessage	method	Send a message	remove	method	Remove key	close	method	Close connection
logout	method	Logout	set	method	Set key/value	onMessage	event	Incoming data
onChange	event	Roster change	onChange	event	Value change	onClose	event	Connection closed
onMessage	event	Incoming message						
onStatus	event	Network status						

Figure 5: freedom.js APIs for interacting with social, storage, and transport providers. We support method calls, as well as asynchronous event messages from the provider.

users coalesce otherwise disjoint social networks.

While, it is natural to create identity providers that tie a user's social network into freedom.js, the identity API can also be used to provide generic rendezvous. For example in one of our applications, a custom identity provider assigns each peer a pseudonym and manipulates what gets returned as the user's friends in order to properly match content producers and consumers.

3.3.2 Self-Scaling Storage

The goal of the storage API is to provide a simple key-value interface to a multitude of persistent back ends. To facilitate desired properties by the developer, storage providers are annotated with properties. These properties describe whether the storage is a temporary cache, persistent on the local machine, or distributed and highly available. The properties also describe the scope of visibility. A storage provider can operate in a single global namespace across all applications and users, a global namespace only amongst users of the application, a global namespace among different applications of the same user, or a local namespace specific to the instance. Lastly, the properties define the lifetime of data and the storage quota.

As such, different providers can provide various subsets of these properties. A freedom.js app can also reference multiple storage providers by requesting storage with different properties. For example, it may use a provider backed by local memory as a temporary cache, a provider backed by LocalStorage for device settings, and a provider backed by a reliable DHT for publishing to other users.

3.3.3 Transport

The transport API abstracts a variety of network layer mechanisms for establishing direct peer-to-peer connections between browsers. The reference implementation includes providers backed by WebRTC, native sockets, and Flash RTFMP. In typical operation, a client first calls *create(. . .)*, which returns an *offer* object containing a session description and the client's location. The client then sends the *offer* to the peer it wants to connect to using the identity API as a rendezvous service. The peer accepts the *offer* and generates a *response*, which is sent back to and accepted by the originating client.

Within this abstraction, the transport provider may be implemented to use ICE/STUN/TURN to traverse NATs. Transport could be relayed on an multi-hop overlay network for anonymization, or masked using an obfuscation protocol. Similarly to the storage API, providers can declare specific properties on the type of transport. A developer can request a transport that is reliable, provides in-order delivery, and/or encrypted.

3.4 Permissions Management

A developer always has the ability to use any API, backed by a provider that works entirely in an unprivileged web context. For example, transport is always available through an unprivileged WebRTC provider. In the manifest of a module, developers can request access to privileged freedom.js APIs, such as native sockets or larger storage quota. The developer also has the ability to insert hints for preferred providers or providers that hold certain properties, like reliable transport. When a freedom.js module is installed into the freedom.js Manager, the user is presented with a dialog to override individual permissions. For example, a user can enable the privileged storage providers, but turn off the access to privileged transport providers for an application. In the case where a user denies a permission, the module will see the default unprivileged provider. Users can subsequently modify permissions at any time in the freedom.js Manager. This same interface allows users to swap out one provider for another, giving them ultimate control over application functionality. While this opens the possibility of breaking application functionality, correct distributed services must already handle individual node failures, limiting impact to the individual user's instance.

4. APPLICATIONS

In order to evaluate the performance and simplicity of freedom.js, we implemented a number of applications on top of our platform. In this section, we divide the applications into three broad classes: multi-user sharing, content delivery and overlay networks. These applications

were built on a shared set of freedom.js APIs, allowing for a high degree of code reuse, such that in all cases the majority of application-specific code was user interface logic. Through these applications, we also demonstrate the use of freedom.js Manager for access to trusted APIs.

4.1 Multi-User Sharing

freedom.js naturally enables a broad space of multi-user interactions. The identity API provides built-in access to the notion of a user's identity and social network. The storage API can be used as a reliable mailbox abstraction between users and high-bandwidth interactions can occur directly between peers using transport providers. All of the following applications were written such that they are possible to run without servers. An application may contact a third-party server to satisfy a particular identity provider, such as to connect to an existing Facebook profile. This allows freedom.js to handle existing network effects, where one group of users may be connected through Facebook, while another, overlapping, set of users communicates primarily through AIM or Windows messenger.

4.1.1 Chat

Using the freedom.js APIs, it was trivial to build a chat application that automatically hooked into a variety of existing chat networks. We ported the existing node-xmpp library, written for the Node.js server-side JavaScript runtime, to work in Chrome as an identity provider. Users can link an arbitrary number of XMPP accounts across various servers, such as ones from Google, Facebook and Apple. In our implementation, we used the identity provider merely as a signalling channel for establishing direct peer-to-peer connections using the WebRTC transport provider. Messages were then sent directly between peers. This minimal chat application consisted of 143 lines of code, all used to describe the visual layout and respond to incoming and outgoing message events.

4.1.2 File Synchronization

Agora is a social file synchronization tool that offers similar file-sharing capabilities to Dropbox. This application makes use of all three of the freedom.js APIs described above: Identity is used to find friends and to create shared folders. Transport is used to synchronize files between participants, so that the different users see a consistent view of shared files. Storage is used to keep a local copy of shared data. These three freedom.js APIs made it simple to implement this system, with module boundaries naturally lining up with class interfaces.

Agora consists of 3 major components, a front end user interface (UI), the root *Agora* back end module, and an aggregate identity provider. The front end implementation was constructed using the Backbone.js, a popular framework for designing interactive UIs. We modified the *sync* primitive, originally used to synchronize the local data model with a remote server, to instead communicate with our back end module. Thus, the front end was written essentially identical to existing web apps. We made use of drag-and-drop APIs to allow files to be dragged between the user's file system and the Agora application.

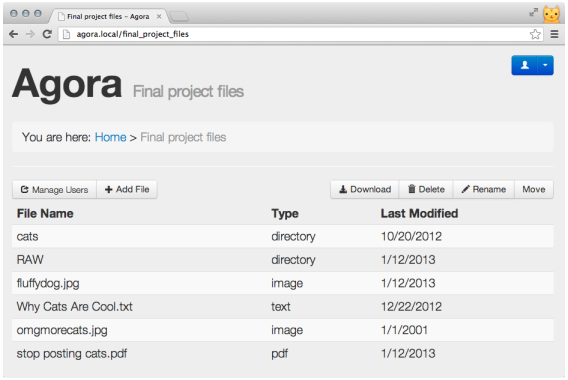


Figure 6: Screenshot of our social file synchronization tool. Users access and interact with the application in the same way as traditional web app.

The *Agora* back end exposes a similar interface to what a RESTful web service would provide. This interface includes calls such as retrieveFiles, putFile, getFile, createSharedFolder, and modifySharedFolder. In order to store more than 5MB, the limit for unprivileged providers, the storage provider requests access to the freedom.js Manager and freedom.js automatically handles the interaction between front and back end. The back end also manages automatic file synchronizations of shared folders when a user is online. We use established vector clock techniques [10] to synchronize files between users. Note that we can reuse the same identity and transport providers as our chat program to find friends and communicate between peers.

4.1.3 Photo and Video Sharing

Using our file synchronization infrastructure, it was easy to create an application for sharing rich media like photos and videos. We designed a new front end interface to present images and videos in a tiling layout, allowing users to zoom images and play videos in place. We reused the Agora freedom.js modules to handle storage and synchronization between users. We imagine this same back end can serve as a powerful common service for a variety of sharing applications, such as microblogs, blogs, and music. Users can also easily design and replace one front end with another, a capability difficult to achieve in existing websites.

4.2 Content Delivery

The ability to communicate between peers makes the browser a powerful platform for content delivery. Proposals have been introduced for many years to leverage users as a content delivery network (CDN) [1, 8, 39, 42]. freedom.js makes it easy to research, experiment, and design new forms of content delivery tailored towards bulk data, streaming media, privacy, or security. Existing server-based websites generally use CDNs to deliver static content like videos. We implemented a simple P2P content delivery network called *InstaCDN* on top of freedom.js. For comparison, we reimplemented the same design as a standalone JavaScript library without freedom.js.

With a number of InstaCDN microbenchmarks, we aim to show that designing applications in freedom.js produces code that is clean, simple, and reasonably performant. Our implementation with freedom.js was less than half the size of the stand-alone version. Our experiments were conducted using Chrome 27 on a 1.8 GHz Apple laptop. We found that loading the freedom.js library added an additional 50-60ms of startup latency and the two versions had no considerable difference in throughput.

4.2.1 InstaCDN Design

In InstaCDN, a central server keeps track of recent clients that download each image. Thus, the server acts as a global tracker of image resources. A developer that wants to use InstaCDN loads a special library into their source code and annotates images by specifying a 'data-src' attribute. Annotated images indicate to the library that these resources should be fetched from peers and should only be retrieved from the server if no clients have the resource. In the freedom.js version of this application, an outer script searches the DOM for annotated images and requests these resources from a back end module. The back end module implements a simple fetch method, where images are fetched based on a canonical URL. We wrote a custom identity provider to interface with the central tracker. Each client is identified by a globally unique pseudonym and reports their currently cached resources to the tracker. In this case when a client requests a resource, the 'roster' returned by the server is the set of clients that have recently cached desired resources. This server then acts as a signaling channel for establishing direct peer-to-peer connections using the transport provider. In the InstaCDN version without freedom.js, we moved all of the back end application logic into the outer page context.

4.2.2 Code Complexity

Figure 7 shows the lines of code for our two implementations of InstaCDN. The freedom.js platform causes application logic to be divided into discrete modules. This design pattern allowed us to easily decompose our application into individual services, and improved testability. Even with the additional manifest files, our freedom.js implementation was less than half the length of the standalone version. With freedom.js, we were able to take advantage of built-in transport providers and message passing primitives for connecting components. In the implementation without freedom.js, we had to re-implement much of the boiler plate code involved with establishing WebRTC connections. Without isolation primitives, a bug can bring down the entire page, as opposed to just a single web worker.

	File	LOC
With freedom.js	outerpage.js	52
	instacdn-manifest.json	14
	instacdn.js	185
	identity-manifest.json	10
	identity.js	50
	Total	311
Without freedom.js	instacdn.js	718
	Total	718

Figure 7: Our freedom.js implementation of InstaCDN is decomposed into relatively small modules of functionality. In total, our freedom.js version is less than half the size of our stand-alone version without freedom.js.

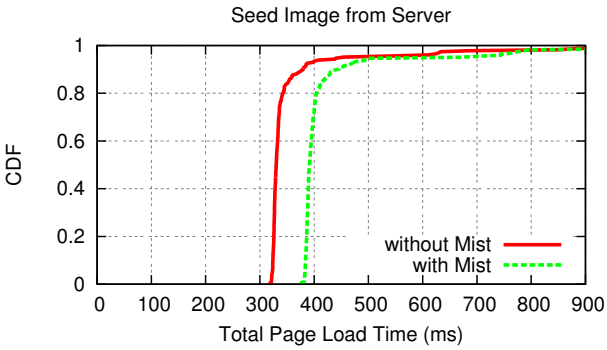


Figure 8: When fetching images from servers using the InstaCDN library, our webpage loaded 60ms slower with the freedom.js implementation when compared to an implementation implemented on raw JavaScript APIs.

4.2.3 Startup Latency

In this experiment, we wanted to quantify the additional startup latency for loading a freedom.js app. The application was served from a web server running on the local machine to approximate a locally stored application. We ran our central tracker in a separate cloud-based service with a 75ms round-trip time from the evaluation client. We also stored the seed image in a cloud-based service, where an HTTP GET for the image took ap-

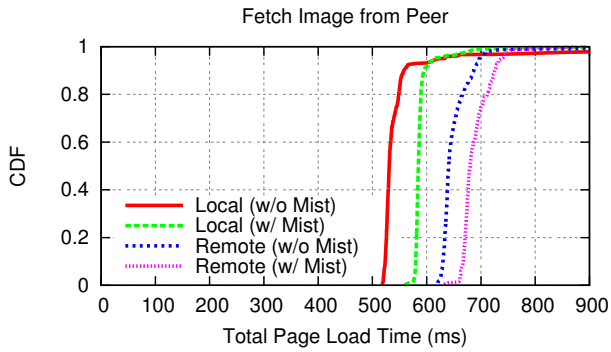


Figure 9: When fetching images from peers using the InstaCDN library, the additional latency overhead incurred by freedom.js was 6% higher than a comparable implementation on native JavaScript APIs.

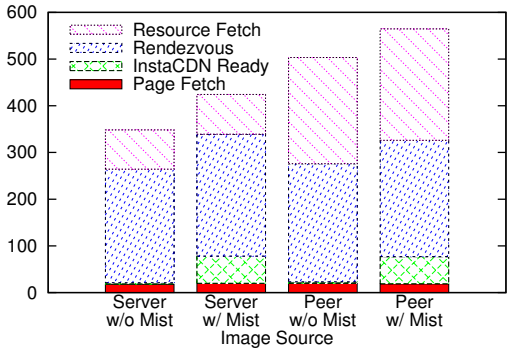


Figure 10: Breakdown of components of page load time. The freedom.js library takes on average 57ms to load before modules can begin to run. Fetching from peers is over twice as expensive as fetching from servers due to the signaling needed to establish a session.

proximately 80ms. Figure 8 and 9 show the end-to-end latency of loading an InstaCDN enabled page with a single image, measured as the time from page request to complete rendering on screen, which includes the image fetch. As shown in Figure 8 when the image is fetched from the server, the median page load time is 330ms. The freedom.js page loads in 391ms. The extra 61ms is consistent with the overhead of starting a separate JavaScript VM in an isolated web worker when loading the freedom.js library.

Figure 9 compares the cost of serving content between peers on a local LAN and with that of different residences in the same city. When InstaCDN fetches from peers, we find that the end-to-end load time increases to 530ms without freedom.js and 584ms with freedom.js. In other words, page load times suffer 193ms of overhead in end-to-end latency. We found this to be caused by the multiple round trips currently used to initiate a WebRTC connection between peers, which is not necessary when fetching directly from a server. WebRTC data channels on Chrome are being actively developed and we anticipate this performance to improve steadily as the technology matures.

When we fetch data from a remote peer on a cable connection we found additional page load times proportional to the network latency between peers as expected. These numbers were in line with the latencies found by [42]. While transport setup is only one component of page load time, it does impose a significant overhead. An important benefit of the freedom.js Manager is to allow peer connections to remain established across page loads, so that subsequent pages on a domain can skip the peer-establishment process. Note that the additional overhead of starting the freedom.js platform represents only 6% of total latency for fetching remote images with InstaCDN.

Figure 10 shows the latency breakdown of a page load. The solid bar represents the time to retrieve the page and begin rendering. For the freedom.js version, loading the freedom.js library takes an additional 57ms before modules can begin running, represented by the second checkered bar. Each version then communicates with the tracker to determine if any peers have the resource cached, represented by the dotted bar. Finally, the image is fetched from either the server or peer, represented by the striped bar. Note that peer-to-peer image fetches take considerably longer than fetches from the server. We attribute this latency to the session setup. WebRTC incurs multiple round-trips of communication to establish an agreement and potentially traverse NATs. Loading freedom.js incurs a small overhead when compared to the full page load time. It is also important to note that due to the realtime nature of modern rendering engines, webpages will not block on image fetches. The webpage will become responsive as soon as the document is loaded and the appropriate JavaScript handlers are loaded.

4.2.4 Performance Benchmarks

Figure 11 shows the throughput in terms of images per second that a peer using InstaCDN can serve to other peers. For implementations with and without freedom.js, the throughput peaks at around 6 concurrent clients. We attribute this to the relatively expensive process of session establishment between peers, which occurs before each image fetch. Due to this artifact of the WebRTC implementation, performance will be drastically improved if freedom.js Manager kept connections open once established. Note that the relative overhead of freedom.js module isolation is negligible in comparison to the other costs of establishing peer-to-peer communication.

4.3 Overlay Networks

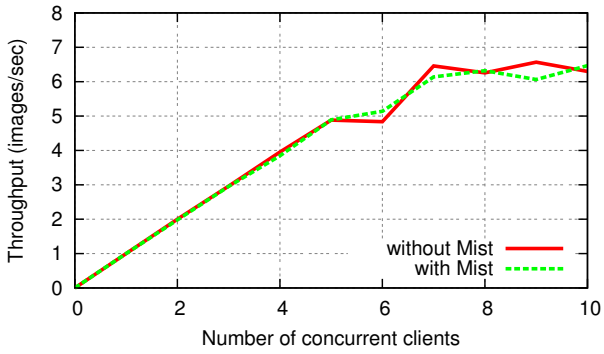


Figure 11: Throughput of a InstaCDN peer serving images to other peers. Our current implementation can only support on the order of 6 concurrent clients fetching images. Note that the relative overhead of freedom.js modules is negligible compared to the overhead in establishing WebRTC communications.

A final class of applications we built on the freedom.js platform are based on overlay communication between users. Communication between freedom.js users using the transport API serves as a foundation for more advanced overlay structures, such as anonymization networks. Users can tunnel traffic through their friends or through users found through other identity providers.

4.3.1 Social Proxy

uProxy is a social web proxy for Chrome built entirely in JavaScript. Users are presented with a list of friends and can ask friends to act as private web proxies. We make use of the same identity providers from our social messaging application. To support uProxy, we implemented a SOCKS5 proxy entirely in JavaScript as a freedom.js module and use the transport API to communicate between users. uProxy can be used as an anti-censorship tool for friends to help each other bypass censors. This tool can also be used to route around Internet failures [9].

4.3.2 Multiple Vantage Points

Research has shown that network and service providers have been found to modify web content for a variety of reasons. Reports have shown entities involved in price discrimination [30], censorship, and filtering on mobile networks [40]. Using the uProxy network, we can route traffic through multiple vantage points on the Internet and detect inconsistencies. Inconsistencies are then reported back to the user, who may choose to access particular domains through different routes.

4.4 Developer Experience

We gathered experience surveys from five developers that used freedom.js to build the applications above. From their feedback, we found points that validated elements of our design and new ways to improve the freedom.js development experience. For developers with experience with JavaScript packages, such as the ones used by node.js and browser extensions, defining manifests and modules in freedom.js was an easy transition.

The main difficulty developers had was in adapting to freedom.js's mechanism for inter-module communications. In our implementation, we only allowed messages to be passed between modules. In the future, we may allow a developer to define a RPC-like interface to a module, facilitating easier ways to reason about application behavior. Developers found the complexity of handling all possible message reorderings a complex task. The synchronization extension for Backbone.js made it drastically easier to reason about behavior without a custom messaging protocol. We plan to create similar compatibility layers with other popular JavaScript frameworks.

The debugging process for freedom.js apps can also be improved. Browser tooling and debugging support has made tremendous advances in recent years, but debugging of web workers is less refined. freedom.js replicates the browser console object to facilitate logging, but we found that it was easy for debugging messages to overwhelm the developer. In the future, we plan to improve console debugging, and output stack traces of errors.

5. DISCUSSION

5.1 New APIs

In Section 3.3 we lay out a core subset of APIs in use by freedom.js providers. Due to space constraints, we do not describe in detail the API for rendering views on a screen (i.e. for authentication with an identity provider). In the future, we can imagine a number of new APIs that can be added to freedom.js to promote additional interfaces needed in distributed systems. These include an interface for maintaining consensus between nodes and one to facilitate resource trading. One could imagine sharing computational, storage, or network resources between users in order to achieve certain properties like high availability on Agora. We could also explore the use of virtual currencies or a reputation system.

5.2 Multi-device

As internet usage increasingly shifts to mobile platforms, managing multiple devices appropriately becomes critical to the freedom.js story. Mobile browsers have committed to developing feature-parity with their desktop cousins, with some technical features (like WebRTC) already available on mobile devices, and others (like browser extensions) expected to arrive soon. As such, we expect mobile devices to act as first-class freedom.js devices, but with less availability and resources to con-

tribute. We expect per-user storage providers to emerge, and for users who rely primarily on mobile devices for freedom.js usage to be able to acquire resources from an always on desktop or cloud providers in place of doing all work locally on the mobile device. This could either be fully abstracted through a resource sharing provider implementation, or simply by renting a VM in which a headless browser acts as a persistent agent for the user.

6. RELATED WORK

Providing serverless interactive applications has been a longstanding goal of the research community, and freedom.js draws on a large corpus of previous work:

Scalable storage: Reliable scalable storage is a core component to any distributed application. Distributed hash tables (DHTs) have been developed as a scalable mechanism to store data across peers in a network [29, 37]. Various DHT designs have been proposed to improve robustness of the storage system by tackling problems such as Sybils [25], consistency [22], churn [26], and locality [20]. freedom.js takes advantage of this work by allowing any DHT to be written as a module and subsequently provide storage for other applications. DHTs have been used as a core building block for a variety of serverless P2P applications such as microblogging [36, 41], social networks [11,17], and content publishing [15], and we aim to enable similar applications to be easily written on top of freedom.js.

Peer-to-peer platforms Platforms for writing peer-to-peer (P2P) applications have existed in a number of domain-specific implementations. BitTorrent [1] has gained significant popularity as a file-sharing protocol. A number of BitTorrent clients support writing applications or plugins to augment the file-sharing experience [6]. Tor [19] is an anonymizing overlay that exposes arbitrary traditional web services as anonymous hidden services within the network. Other related work describes similarly hiding existing web services within a blocking-resistant anti-censorship tool [35]. freedom.js is unique in that it aims to provide a general platform for writing web applications entirely in the browser, without native software. Previous proposals have called for using P2P network stacks to help open source web applications scale for free [14]. freedom.js expands on this vision by specifying an API and an execution environment that expose useful primitives and sandbox isolation.

Alternative web architectures: freedom.js takes inspiration from a number of recent advances in web-related research. Systems have been proposed to use P2P file-transfer as a replacement for traditional content delivery networks in the browser [1, 8, 39, 42]. We show in Section 4 how these services could be written in freedom.js with fewer lines of code. Tent [4] and Diaspora [2] are two efforts to create protocols for federated social network servers, much like the XMPP chat protocol. Unhosted [5] and BStore [12] provide a number of tools to detach web applications from where the data is stored. Most of these tools, such as RemoteStorage [3], can be used as freedom.js modules.

We also take inspiration from a large background of work in security and isolation in the web. Treehouse [24] explores the use of WebWorkers to provide isolation between scripts within a web application, a technique similarly used in our system. Tacoma [16] and Embassies [23] are a two proposals to use client-side virtual machines to achieve stronger isolation between web application containers, but with a higher deployment cost. Js.js [38] sandboxes third-party scripts in a Javascript interpreter written in Javascript. Other proposals have been incorporated to provide stronger isolation between web containers in the browser [13, 33].

MapJAX [31] exposes new abstractions for interacting with server-side data using Javascript objects, a technique that we take inspiration from for our templated provider interfaces. Hails [21] and DBTaint [18] investigates the use of information flow control (IFC) to protect user data from being overshared. Gibraltar [27] and Maverick [34] are systems that further push the boundaries of client-side application logic by exposing hardware devices to web applications. Liberated [28] is a development environment where the server and client both run in the browser to facilitate application debugging. We are excited about the possibility of incorporating these concepts, such as IFC, hardware access, and debugging environments, in future work with freedom.js.

7. CONCLUSION

While the Web has served as a powerful medium for deploying software at scale, its current architecture is directly at odds with enabling rich multi-user applications that are free, easily customizable, and composable. By taking advantage of modern advancements in browser APIs, freedom.js provides an alternative model for building such web apps by removing servers from the equation. freedom.js apps are simple to write, easily composable, automatically scale, and gives users far more control over application behavior than existing web apps.

To demonstrate that building free serverless web apps are not only possible, but also practical, we implemented the freedom.js platform in JavaScript. We built a multitude of applications on top of freedom.js, including a messaging application, a social file synchronization tool, and a peer-to-peer (P2P) content delivery network (CDN). Through microbenchmarks we show that freedom.js apps incur minimal performance overhead, but enable drastic reductions in code complexity. We hope that with freedom.js, we can support a vibrant community of free software on the Web.

8. REFERENCES

[1] BitTorrent. http://www.bittorrent.com

[2] Diaspora. https://github.com/diaspora/diaspora

[3] RemoteStorage. http://remotestorage.io/

[4] Tent. https://tent.io

[5] Unhosted. https://unhosted.org/

[6] Vuze Plugins. https://www.vuze.com/plugins/

[7] Wikimedia Foundation Annual Report. http://upload.wikimedia.org/wikipedia/commons/4/48/WMF_AR11_SHIP_spreads_15dec11_72dpi.pdf 2011.

[8] P. Aditya, M. Zhao, Y. Lin, A. Haeberlen, P. Druschel, B. Maggs, and B. Wishon. Reliable client accounting for p2p-infrastructure hybrids. In *Proceedings of USENIX NSDI*, 2012.

[9] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient overlay networks. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 131–145, New York, NY, USA, 2001. ACM.

[10] S. Balasubramaniam and B. C. Pierce. What is a file synchronizer? In *Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom '98)*, Oct. 1998.

[11] S. Buchegger, D. Schiöberg, L.-H. Vu, and A. Datta. Peerson: P2p social networking: early experiences and insights. In *Proceedings of the Second ACM EuroSys Workshop on Social Network Systems*, pages 46–52. ACM, 2009.

[12] R. Chandra, P. Gupta, and N. Zeldovich. Separating web applications from user data storage with bstore. 2010.

[13] E. Y. Chen, J. Bau, C. Reis, A. Barth, and C. Jackson. App isolation: get the security of multiple browsers with just one. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pages 227–238. ACM, 2011.

[14] R. Cheng, W. Scott, A. Krishnamurthy, and T. Anderson. FreeDOM: a new baseline for the web. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 121–126. ACM, 2012.

[15] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Designing Privacy Enhancing Technologies*, pages 46–66. Springer, 2001.

[16] R. S. Cox, J. G. Hansen, S. D. Gribble, and H. M. Levy. A safety-oriented platform for web applications. In *Security and Privacy, 2006 IEEE Symposium on*, pages 15–pp. IEEE, 2006.

[17] L. A. Cutillo, R. Molva, and T. Strufe. Safebook: A privacy-preserving online social network leveraging on real-life trust. *Communications Magazine, IEEE*, 47(12):94–101, 2009.

[18] B. Davis and H. Chen. Dbtaint: cross-application information flow tracking via databases. In *2010 USENIX Conference on Web Application Development*, 2010.

[19] R. Dingledine, N. Mathewson, and P. Syverson. Tor: the second-generation onion router. In *USENIX Sec.*, 2004.

[20] M. J. Freedman, E. Freudenthal, and D. Mazieres. Democratizing content publication with coral. NSDI, 2004.

[21] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. C. Mitchell, and A. Russo. Hails: protecting data privacy in untrusted web applications. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, pages 47–60, 2012.

[22] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Scalable consistency in scatter. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 15–28. ACM, 2011.

[23] J. Howell, B. Parno, and J. Douceur. Embassies: Radically refactoring the web. *NSDI*, 2013.

[24] L. Ingram and M. Walfish. Tingram2012treehousoreehouse: Javascript sandboxes to help web developers help themselves. In *Proceedings of the USENIX Annual Technical Conference*, 2012.

[25] C. Lesniewski-Lass and M. F. Kaashoek. Whanau: A sybil-proof distributed hash table. In *7th USENIX Symposium on Network Design and Implementation*, pages 3–17, 2010.

[26] J. Li, J. Stribling, T. Gil, R. Morris, and M. Kaashoek. Comparing the performance of distributed hash tables under churn. *Peer-to-Peer Systems III*, pages 87–99, 2005.

[27] K. Lin, D. C. J. Mickens, L. Z. F. Zhao, and J. Qiu. Gibraltar: exposing hardware devices to web pages using ajax. In *Proceedings of the 3rd USENIX Conference on Web Application Development*, pages 7–7. USENIX Association, 2012.

[28] D. Lipman. *LIBERATED: a fully in-browser client and server web application debug and test environment*. PhD thesis, University of Massachusetts, 2011.

[29] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. *Peer-to-Peer Systems*, pages 53–65, 2002.

[30] J. Mikians, L. Gyarmati, V. Erramilli, and N. Laoutaris. Detecting price and search discrimination on the internet. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 79–84. ACM, 2012.

[31] D. Myers, J. Carlisle, J. Cowling, and B. Liskov. Mapjax: Data structure abstractions for asynchronous web applications. In *Proceedings of the 2007 USENIX Annual Technical Conference*, 2007.

[32] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. You are what you include: Large-scale evaluation of remote JavaScript inclusions. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2012.

[33] C. Reis and S. D. Gribble. Isolating web programs in modern browser architectures. In *Proceedings of the 4th ACM European Conference on Computer Systems*, pages 219–232. ACM, 2009.

[34] D. W. Richardson and S. D. Gribble. Maverick: Providing web applications with safe and flexible access to local devices. In *Proceedings of the 2011 USENIX Conference on Web Application Development (June 2011), WebApps*, volume 11, 2011.

[35] W. Scott, R. Cheng, J. Li, A. Krishnamurthy, and T. Anderson. Blocking-resistant network services using Unblock. Technical report, University of Washington Computer Science and Engineering, 2014.

[36] P. St Juste, D. Wolinsky, P. O. Boykin, and R. J. Figueiredo. Litter: A lightweight peer-to-peer microblogging service. In *Privacy, Security, Risk and Trust (PASSAT), IEEE Third International Conference on Social Computing (SocialCom)*, pages 900–903. IEEE, 2011.

[37] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[38] J. Terrace, S. Beard, and N. P. K. Katta. JavaScript in JavaScript (js.js): Sandboxing third-party scripts. 2012.

[39] J. Terrace, H. Laidlaw, H. E. Liu, S. Stern, and M. J. Freedman. Bringing p2p to the web: Security and privacy in the firecoral network. In *International Workshop on Peer-to-Peer Systems IPTPS 2009*, 2009.

[40] Z. Wang, Z. Qian, Q. Xu, Z. Mao, and M. Zhang. An untold story of middleboxes in cellular networks. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 374–385. ACM, 2011.

[41] T. Xu, Y. Chen, J. Zhao, and X. Fu. Cuckoo: towards decentralized, socio-aware online microblogging services and data measurements. In *Proceedings of the 2nd ACM International Workshop on Hot Topics in Planet-scale Measurement*, page 4. ACM, 2010.

[42] L. Zhang, F. Zhou, and R. S. Alan Mislove. Maygh: Building a CDN from client web browsers. In *In Proceedings of the 8th European Conference on Computer Systems EuroSys*, 2013.