FreeDOM: a New Baseline for the Web

Raymond Cheng, Will Scott, Arvind Krishnamurthy, Thomas Anderson
University of Washington
{ryscheng,wrs,arvind,tom}@cs.washington.edu

ABSTRACT

Free web services often face growing pains. In the current client-server access model, the cost of providing a service increases with its popularity. This leads organizations that want to provide services free-of-charge to rely to donations, advertisements, or mergers with larger companies to cope with operational costs.

This paper proposes an alternative architecture for deploying services that allows more web services to be offered for free. We leverage recent developments in web technologies to combine the portability of the existing web with the user-powered scalability of distributed P2P solutions. We show how this solution addresses issues of user security, data sharing, and application distribution. By employing an easily composable communication interface and rich storage permissions, the FreeDOM architecture encourages flexible interactions between applications while enforcing privacy controls. We demonstrate the applicability of this architecture by presenting a SQL database and a community-supported Wiki as case studies.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed applications

General Terms

Design, Reliability

1. INTRODUCTION

Free web services, such as Wikipedia and OpenStreetMap, are some of the most influential sources of information in the world. By offering open distribution and a rich collaborative environment, these services have provided value to countless individuals and organizations, and they have in turn become platforms for further innovation. For example, rather than investing in the capital-intensive effort of mapping the world, companies like Apple and Foursquare can leverage OpenStreetMap maps, allowing them to focus on their core business. As open source software enables new systems to build on existing libraries, free web services can allow new services to leverage community contributions.

However, unlike open source software, free web services aren't really free. Wikipedia must raise large amounts of money to support its server operations and bandwidth costs. Some organizations, like Wordpress, freely release the technology for their service, but leave it up to users to handle hosting costs. Many developers of free apps are faced with a dilemma: write a popular app and lose a little money, write a viral app and lose a lot of money. As the success of Wikipedia has shown, there can be enormous value to society and the rest of the web from services that do not leverage their data for financial gain. However, the number of free web services will remain very small if they all must fund their own operations and growth.

Peer-to-peer (P2P) technologies provide a way to distribute data in a cheap and scalable manner, where resources scale with the user community. However, most P2P systems lack the portability, accessibility, and ease of use of the web's client-server model. Heterogeneity in devices, operating systems, and network configurations, makes the P2P model prohibitive for most developers.

While it is not our goal to remove all of the costs of developing a free web service, we do want to make it possible to *scale* web applications at very low cost. We leverage a number of technology trends in support of our vision. The explosion of new APIs and browser capabilities have transformed the Web from a collection of static pages to a set of rich web applications, written within a write-once-run-anywhere framework. Particularly interesting is the introduction of the ability for client browsers to directly communicate with other client browsers, using technologies such as WebRTC [6], Socket APIs, and RTMP [4].

In this paper, we propose an architecture for browser-based services that exposes flexibility beyond a traditional client-server model. This architecture is structured as a set of common services and a support library for applications. We ex-

plain the primary benefits gained by this model: low cost scaling, user privacy, application resiliency, and ease of development, and why we believe it is important to create these services and popularize this broader form of web communication in the near future.

This architecture evolved from the development of a distributed database application, presented in Section 4. Our original goal was to offload computation from the server by sharing client caches. We realized that much of the work could be abstracted to provide a service layer for a broader class of applications. The FreeDOM model in this paper is in early development, and the APIs and services we present are not finalized implementations.

In Sections 2 and 3 we motivate and explain our design. We then present our distributed SQL database that runs across browsers, and a model for a distributed wiki application as case studies of the FreeDOM model in Section 4. Section 6 discusses context and related work, and Section 7 concludes.

2. MOTIVATION

FreeDOM leverages evolving web standards in order to empower developers to develop low cost self-scaling web services. Wikipedia spent $1.8 million just on Internet hosting from July 2010 to June 2011, relying entirely on financial donations to support the foundation [1]. As existing P2P software has shown, there exist ample excess resources on end-hosts that can be leveraged to provide high-value services.

We hope that FreeDOM can lower the barrier of entry to community-supported applications. These applications should operate seamlessly with the existing web experience, and require no extra effort from the user's point of view. For example, there is no need to install a plugin or third-party software, as all of the necessary runtime elements now exist in the browser. Instead, users participate by accessing a FreeDOM-enabled web application as they would any other cloud-backed web application. While there are a few widely used open source web applications today, we believe the web can easily support the same thriving environment that surrounds Linux, where developers contribute and build upon existing services.

Perhaps just as exciting is the possibility of a new class of applications that are enabled by the evolving capabilities of the browser. By detaching applications from the strict client-server model where users and their data are inherently tied to the server's infrastructure, we envision more flexible and interconnected applications. The FreeDOM architecture offers a number of additional benefits:

- **User-controlled data**

  Modern web services collect a growing amount of personal data. In part this is encouraged by the economics of supporting the service. Users tend to be at the whim of the web application, which can decide how much control a user ultimately has over their own data. This fact can lead to both lock in and violations of privacy [18]. The FreeDOM data model keeps users in control of their data. We remove the motivation for applications to exploit users for revenue. Furthermore, we can integrate privacy-preserving mechanisms like Tor [9].

- **Self-scaling infrastructure**

  There exists a wide body of research into scalable storage systems, such as distributed hash tables [17, 16] and social backup services [15]. By integrating with the web, these applications can leverage the storage, network, and computational resources of other users to create interesting hybrid architectures. Self-scaling storage systems can be used to mitigate flash crowds, employ smarter caching techniques, improve locality of data, and of course lower operating costs.

- **Resilient connectivity**

  Research has shown that the Internet suffers from frequent partial outages [14]. When a user accesses a traditional web app, it is up to the application developer to employ smart failover strategies. These solutions can entail significant engineering costs and code complexity for each additional "9" of reliability. By leveraging the inherent distribution of users across the globe, FreeDOM applications can directly employ intelligent routing schemes that react to partial failures [5]. By removing dependencies on central points of failure, distributed web apps may also offer better DoS resistance.

- **Composable services**

  Modern web applications are large vertical silos, containing both the application itself and all user data. Some services may expose parts of their functionality through custom outward-facing API's. This practice places the burden on third-party developers to support each custom API with which they want to interact. The FreeDOM model encourages a common data sharing mechanism so that new services can be composed of common foundational elements, like a reliable storage system or a social graph. We see this model as a successor to Unix pipes, where individual services can be chained together to produce higher level functionality with minimal work.

2.1 Why Now?

Web technologies have reached the level where they can provide a rich environment for applications and services. The browser as a platform is an enticing environment because the assumption of untrusted code allows applications to be run without worrying about risks to the user. Now that this technology exists, our work aims to broaden the class of applications supported by the web.

To motivate the need for FreeDOM, we must ask what the browser won't provide by itself. The broad answer is that browsers are not going to solve issues of client-to-client resource usage, since they are still driven by a client-server
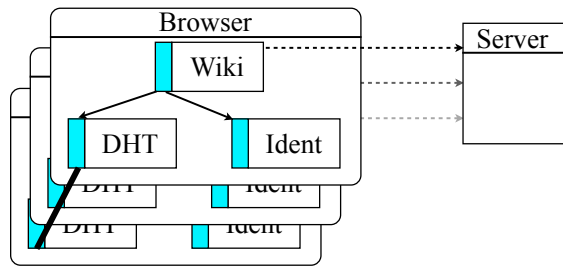
Figure 1: Example FreeDOM service composition. A wiki application interacts not only with a central server, it also relies on other services for identity and storage. The shaded portion of applications represents the common FreeDOM library.

model. Likewise, browsers do not provide support for arbitrary distribution of data, or a way to communicate with other services in a generic way.

However, browsers do provide the technology needed to solve these problems. In particular, WebRTC provides a mechanism for direct communication between untrusted web applications. NaCl, a Chrome technology, allows legacy libraries written in almost any language to be run safely in a web context. Proposals like web components [8] and a standard method for message passing between web sites are moving towards composability and interaction between services. FreeDOM aims to solve the remaining problems needed to support free, community driven services.

FreeDOM represents an alternative to the datacenter-centric cloud model now prevalent in industry. We believe that all of the benefits of FreeDOM - privacy, low barrier of entry development, resilient connectivity, and composable services - are desirable to both developers and users, and that our model can provide a viable alternative for services currently forced to run in the cloud.

3.   DESIGN OVERVIEW

The FreeDOM model can be instantiated as a common library that resides within applications. This library exposes a FreeDOM API above those normally found in the browser, providing inter-app and inter-user communication. The FreeDOM library needs no additional privileges beyond those of installed apps themselves.

A FreeDOM application consists of a set of files that can be rendered in the context of a web browser. Typically, it will consist of HTML files, Javascript, images, and style information. The power of the FreeDOM model comes from the ability of different applications to easily interact, and in a powerful set of core services offering reliable storage, cross-browser communication, access policies, and identity management.

3.1   System Components

3.1.1   Permissions and Storage Policy

There exist a number of privileged browser API's, such as geolocation, persistent storage, and networking that are currently protected from arbitrary access. Applications must request permission from the user to access these privileged API's. Furthermore, individual permissions may be revoked at any time by the user. FreeDOM exposes a number of additional services such as distributed storage, that follow the existing permissions model. Regardless of the user's choice, FreeDOM exposes a single API to the developer that will always work. For example, a user may use Google, Facebook, or a PKI as their identity provider. In all cases, the decision is masked by the FreeDOM API for seemless operation.

Additionally, we believe that in a community-supported service, users must be able to have control over the behavior of an application. They must be able to protect themselves from participating in illegal activities, such as inadvertently hosting illicit content. In general, it should be up to the application developer to ensure that mechanisms are put in place to remove content. However, users must always have the option to block specific content, communication channels, or to uninstall and purge an entire application. Ultimately, users and developers are still liable for their own actions using community resources.

3.1.2   Application Distribution

Applications are distributed as signed archives, as they are in the current model for browser applications and extensions. We plan to create an app management service to help police applications and mitigate spam and malware. Such a service is able to use social channels to suggest and retrieve applications, and blacklists to protect against malware.

With FreeDOM's authentication mechanism, developers can opt to distribute updates through any means. Regardless of the source of the application, the browser can easily verify the signature of update packages, ensuring that the package was signed by the desired developer. FreeDOM applications can thus be hosted from curated app stores, through social channels, or directly from other peers.

3.1.3   Cross-app data sharing

Much like Unix pipes, FreeDOM provides a standard mechanism for inter-app communication. FreeDOM applications send and receive messages from a designated location in the application. This mechanism, along with a simplified discovery process, allows applications to easily invoke other services. FreeDOM provides intermediation of this communication, allowing for dynamic binding of services not based on the type of interface. Like stdin/stdout, the receiver is responsible for interpreting transmitted data. Applications may also share data through the storage service, much like a traditional operating system.

Several technologies help with this design. Data can be

stored with the browser's filesystem API and referenced via a URL. The URL serves as a capability to access data from other applications. Similarly, the web's `postMessage` primitive allows for cross application communication, but additional work is required in FreeDOM to provide fine grain access controls beyond those enforced by the same origin policy.

3.1.4 Reliable storage

Reliable storage is an important building block for many community driven applications, since the ability to retrieve resources from other users rather than a central server is critical in minimizing service cost. Storage in the FreeDOM model is a service exposed to other applications; requests to the service are communicated as with other services.

Browsers use several mechanisms for storage today. Local storage provides a key-value store accessible to both trusted and untrusted applications. Additional APIs exist in the form of WebSQL, IndexDB, and a Filesystem API. The last, while not standardized, provides a sandboxed file abstraction and allows stored objects to be later accessed as web resources. These mechanisms are similar to the FreeDOM service, but the FreeDOM service also manages access between applications and controls when applications can interact with each others data. The ability to share stored data follows from our view that free services must be composable.

3.1.5 Messaging

Communication between clients is an important enabler for a large class of applications and is needed for our vision of community supported services. The pragmatic approach to messaging is to provide it as a service, and allow messages to be transported by whichever methods are available.

Technically, message passing requires both a format and a transport mechanism. The format of choice for web applications has standardized on JSON, and there is little reason to deviate from that standard. Browsers provide multiple built-in transport mechanisms which can be (or will shortly be able to be) used for communication with peers. The W3C standardized protocol is WebRTC, which can set up client-client channels. Many browsers also include mechanisms for extensions to make use of native Socket APIs. Additionally, we foresee services which offer transport over social networks, for example using Google chat, Facebook messages, or Twitter direct messages.

The messaging API enforces finer grained policies that dictate how applications can communicate. For example, the transport layer should never allow applications to setup arbitrary network sockets to Internet hosts. Instead, the security policy dictates which users an application can communicate with. A default policy for untrusted applications would allow them to interact with remote instances of themselves.

3.1.6 API

The goal of the FreeDOM API is to make it easy for ser-

vices to find each other, interact, and manage permissions. This goal results in several design choices which we believe together construct a powerful API. First, service interfaces need to allow extensibility of both producers and consumers. The communication API should both allow an application to easily initiate communication with instances of itself running in other browsers, and allow a new service to offer transport. Second, APIs should be implemented so that applications do not have to worry about permissions. If an application attempts to write to storage, some storage provider should always be available. The application must be able to know which service it is talking to, as in the case of a back up service wanting to ensure data is indeed saved reliably. Third, the API should be simple to encourage adoption. An identity API should not expose a complex OAuth API, but instead should directly provide an identity key or provide attestation.

4. CASE STUDIES

4.1 Collaborative WebDB

In order to explore the challenges and requirements of the FreeDOM architecture, we prototyped and implemented a distributed static SQL database management system (DBMS) called Collaborative WebDB. It runs as an untrusted web application in the browser and provides a read-only SQL interface to other applications. The authoritative copy of the database is hosted on a remote server, but instances of Collaborative WebDB can cache table pages during the course of operation. Clients independently generate query plans that fetch data on-demand from local and peer caches as well as the server database. In the development of this service we were able to visit many of the different architectural challenges present in the FreeDOM model: How should instances of the application communicate? How do clients know what data other clients have? What trust model and permissions does such an application require?

Traditional databases are demanding and highly performant pieces of software, and Collaborative WebDB demonstrates the ability of modern web browsers to support complex applications. Unique to the FreeDOM model, Collaborative WebDB takes advantage of the other peers in the network, offloading query execution from the server and running the entire SELECT pipeline in the browser. Many data collections follow a zipf access pattern, which means that "hot" data (such as the items appearing on the home page of a web store) will have high probability of being in client caches. This makes a distributed database an appealing candidate for offloading server load.

4.1.1 Implementation

Collaborative WebDB was built by writing a custom data provider to SQLite. The entire package was then compiled into Javascript using the Emscripten compiler[1]. The result-

---

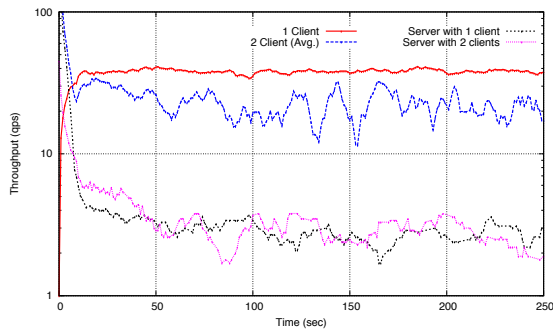[1] `http://github.com/kripken/emscripten/wiki`

Figure 2: Queries per second satisfied by the server and clients over time in Collaborative WebDB. As pages are cached on clients' local machines, server load decreases over time. Even with two clients, server workload remains the same, while total system throughput increases.

ing Javascript file is loaded into an HTML file and run in an untrusted context. While our experiments test the performance in Javascript, it is also possible to compile the same source into a NaCl module, requiring only minimal changes in the communication interface between the compiled module and client code. On top of the database we added a simple table viewer, which executed and displayed SQL queries by passing messages to the database service. We separated the viewer and DBMS in separate applications, and used HTML5 message passing to communicate between the two layers. FreeDOM's message passing, supported by upcoming browser support for web components, allows the interfaces of these two services to be de-coupled, so that the database can accept requests from clients other than our viewer.

We also wrote a Chrome extension providing a WebRTC-like message passing abstraction between active clients on different computers. We used a small web-socket server to coordinate topology and to tell clients which other clients were using the service. With this knowledge, clients can send messages to other peers directly.

4.1.2 Evaluation

In order to evaluate our implementation, we accessed the Collaborative WebDB web application from multiple clients. A server hosted a 1GB database from the TPC-H benchmark [2], and each client ran a synthetic workload of SQL queries selected as a random projection of columns from a random table, at an offset. We chose offsets using a Pareto distribution parameterized by a skew coefficient of 0.5, resulting in 82% of queries referencing the first two pages of table data. This form of distribution models many popularity rankings in the real world. This model was chosen to emulate the usage pattern created from browsing through a product catalog in an online store.

Figure 2 shows that when the experiments were started, clients began with empty caches and the server was heavily loaded. Over time, the server's workload dropped, and a growing percentage of queries could be satisfied from either a local cache, or a peer cache. With two clients, the server handled requests at the same rate, but the query throughput in the system increased.

4.2 Distripedia: A User powered wiki

We have also done a paper design for a distributed wiki in the FreeDOM model. The wiki example tackles many of the fundamental challenges in FreeDOM. How do user accounts and moderation work? How are updates published? What is the developer responsible for? We found that by building on a set of simple primitives, we could easily answer all of these questions.

The key decision in Distripedia is splitting application logic from data. Data, in the form of actual wiki content and historical revisions, are stored with the reliable storage service, a single DHT across all users. This service contains keys for each page in the wiki, with a pointer to accepted revisions, and signed by the application provider. Edits are made by committing a new revision to the DHT, and then submitting the new revision key to the central service. In this way, the application developer is able to maintain control over moderation and access control by choosing how to update the signed pages in the DHT - but the service can continue to function even when the central server unavailable.

5. DISCUSSION

While we were able to build some interesting applications in the FreeDOM model, there still are many challenges to solve in order to make FreeDOM applications robust, secure, and practical.

- **Portability.** Portability of applications, especially ones that access rich API's, has long been a difficult problem. Building web applications provides the ability to run one application across desktops, mobile devices, and gaming consoles alike. However as an evolving standard, most browsers do not have identically consistent behavior, and many implement additional browser-specific API's. For example, not all browsers support or plan to support NaCl, and extension models for browsers are notoriously vendor specific.

- **Low latency services.** Many distributed P2P applications have notoriously bad latency. Hence, user-facing applications that depend on remote data must take into consideration the latency of retrieving necessary data. Research has shown that applications with poor latency drastically impact usability [11] but techniques exist to minimize latency in presence of unreliable participants [19].

- **Flexibility.** One major design goal for FreeDOM is to be able to adapt to new use cases. In desktop systems, the concept that all data is a file has proved very successful, allowing disparate applications to work together. This document was prepared through cooperation of many separate applications which collectively compose LATEX. Web

applications have not standardized on common communication channels in the same way. Partially this is because presentation and data are intermixed, forcing desired data to be extracted from HTTP responses. Another factor is that authorization and data are often entangled in web APIs, further siloing applications into API specific communication.

- **Privacy.** In the web, users have very primitive control over their data. In a typical web application, the user has no insight into how their data is stored or distributed. While a developer may expose some subset of controls to the user, the developer generally has free reign, only limited by local laws and terms of service. We hope that by allowing more services to be operated for free, there will be less incentive or need for services to violate user privacy.

6. RELATED WORK

Google Chrome has made great progress creating a portable application layer on top of the web with Native Client (NaCl) [20]. Chrome is quickly providing full native-yet-portable APIs to applications, which along with NaCl will allow for native apps with a common DOM-based UI to run across platforms. Many of the developments to date have focused on improving the web experience for isolated web applications. Our argument is the need for inter-application communication, stronger user controls, and reliable storage.

Content distribution networks like Akamai [3] and Coral-CDN [12] have explored the value of user resources. In the case of Akamai, P2P users augment the serving capacity of their existing CDN infrastructure. Without the financial implications of charging for access to peer resources, the Free-DOM model aligns with user incentives to support a free service.

Several systems have developed novel models for distributing web content, ranging from fully content-centric models like Freenet [7], to anonymous P2P publishing like Tor hidden services [10]. Other efforts such as hoodwink.d[2] and the Nethernet [13] have experimented with building applications on top of existing websites. FreeDOM looks to leverage the many lessons learned from these and other systems to build a new robust framework that is tightly integrated with the fabric of the web.

7. CONCLUSION

Community powered services have had an incredible impact on the web and society. Unfortunately, even cloud hosting can be an expensive proposition for these services at scale, which shifts developer incentives from free distribution to monetization. We have introduced an alternative model supporting a more flexible communication pattern, which allows users to directly support these services with their own resources. Our vision takes lessons from the success of open

source software on the personal computer, which used simple API primitives and enabled easy application composition. This paper lays out the challenges faced by our approach, and then demonstrates our design with case studies of a P2P database and a community powered wikipedia. By tapping into powerful new web standards, we hope to empower an entirely new class of free, community-supported services.

8. REFERENCES
[1] Wikimedia Foundation Annual Report. wikimedia.org, 2011.
[2] TPC Benchmark H Standard Specification, 2012.
[3] P. Aditya, M. Zhao, Y. Lin, A. Haeberlen, P. Druschel, B. Maggs, and B. Wishon. Reliable Client Accounting for P2P-Infrastructure Hybrids. In *NSDI*, 2012.
[4] Adobe. Real-Time Messaging Protocol specification. Working draft, adobe.com, 2009.
[5] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *SOSP*, 2001.
[6] A. Bergkvist, D. C. Burnett, C. Jennings, and A. Narayanan. WebRTC 1.0: Real-time communication between browsers. Working draft, W3C, February 2012.
[7] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: a distributed anonymous information storage and retrieval system. In *PETS*, 2001.
[8] D. Cooney and D. Glazkov. Introduction to Web Components. Working draft, W3C, May 2012.
[9] R. Dingledine and N. Mathewson. Design of a blocking-resistant anonymity system. torproject.org.
[10] R. Dingledine, N. Mathewson, and P. Syverson. Tor: the second-generation onion router. In *USENIX Sec.*, 2004.
[11] J. B. Eric Schurman. The User and Business Impact of Server Delays, Additional Bytes, and HTTP Chunking in Web Search. In *Velocity*. O'Reilly, 2009.
[12] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing Content Publication with Coral. In *NSDI*, 2004.
[13] J. Hall. Passively Multiplayer Online Games. Master's thesis, USC, 2007.
[14] E. Katz-Bassett, C. Scott, D. R. Choffnes, I. Cunha, V. Valancius, N. Feamster, H. V. Madhyastha, T. Anderson, and A. Krishnamurthy. LIFEGUARD: Practical Repair of Persistent Route Failures. In *SIGCOMM*, 2012.
[15] J. Quintard. *Towards a worldwide storage infrastructure*. PhD thesis, University of Cambridge, September 2010.
[16] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Spring LNCS. Vol. 2218*, 2001.
[17] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *SIGCOMM*, pages 149–160, 2001.
[18] J. Turow, J. King, C. J. Hoofnagle, A. Bleakley, and M. Hennessy. Americans Reject Tailored Advertising and Three Activities That Enable It. *SSRN eLibrary*, Sept. 2009.
[19] L. Wang, V. Pai, and L. Peterson. The Effectiveness of Request Redirection on CDN Robustness. In *OSDI*, 2002.
[20] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *IEEE Symposium on Security and Privacy*, 2009.

---

[2] `http://github.com/whymirror/hoodwinkd`