Exploiting HTML5 Technologies for Distributed Parasitic Web Storge[⋆]

Martin Kruliš, Zbyněk Falt, Filip Zavoral

Parallel Architectures/Applications/Algorithms Research Group
Faculty of Mathematics and Physics, Charles University in Prague
Malostranské nám. 25, Prague, Czech Republic
{krulis,falt,zavoral}@ksi.mff.cuni.cz

Abstract. Current web technologies have been leaping forward, especially since the introduction of HTML5. The web browsers of the day implement various APIs for the client-side scripts, such as elaborate data storage or advanced network connectivity. We propose, how to combine these technologies to create distributed data storage using the web environment. We have implemented a prototype framework as a proof of concept and explored the most problematic issues which require to be researched further. Systems that would use this storage may benefit from the fact that the users do not need to install or configure any type of client application, since the only thing required is the web browser. Web-based distributed data storage can be used as an extension of server storage, for data caching, and in various other applications.

Key words: html5,web storage,distributed,data,parasitic

1 Introduction

World wide web played a role of open interactive platform for information exchange, business, or entertainment for over twenty years. It begun as a simple technology for presenting text documents, but it has evolved significantly in the past decades. In the last few years, its technologies leaped forward as HTML5 [1] emerged, which is not only a new version of the HTML language, but it also integrates specifications for client side scripting and APIs for advanced browser features. It shifted the web browsers from simple web page presenters into an application platform, which even become basis for lightweight operating systems [2].

The ability of executing scripts in the browser raises many security issues, since the user have virtually no way how to verify that these scripts do not perform malicious routines and will not pose a threat. The client scripting environments in the browsers must find a very fine balance between security and provided functionality, which is required by complex client-server applications. Furthermore, the browsers must implement these specifications correctly to ensure at least some level of protection.

⋆ This paper was supported by Czech Science Foundation (GAČR) projects P103/14/14292P and P103/13/08195 and by specific research SVV-2014-260100.

The HTML5 technologies are designed to support more complex client-side scripts for more elaborated client-server web applications. Particularly, HTML5 advanced these browsers capabilities:

- GUI design, graphics, and multimedia support,
- communication capabilities (Server-sent events, WebSockets, WebRTC),
- client-side data and file management (File API, Web Storage, Indexed DB),
- client-side computations (Web Workers, WebCL).

The line between legitimate resource utilization and their exploitation is quite thin and not formally defined. Therefore, it is reasonable to assume that these technologies will remain in the browsers for the perceivable future time, despite the possibility they might be misused by parasitic applications.

1.1 Contributions and Outline

Our particular interest turns towards data storage capabilities and communication capabilities of the browsers. Many current users of the web have a lot of free disk space in their computers and sufficient connectivity. This space may be utilized to store or cache data of the web applications they use. It may also be exploited by a web application, or its embedded component such as ad banner, to secretly infest the users hard drive with their own data.

This paper explores the limitations of current HTML5 technologies and propose a way how these technologies may be used or misused to create a distributed data storage. We have implemented a prototype framework that tests the feasibility of this idea. The prototype let us identify the most problematic areas of this domain and outline possible solutions for them. We have also proposed possible applications which may use these technologies for legitimate reasons and in a parasitic way – i.e., stealing the disk space from the users.

The paper is organized as follows. Section 2 revises related work. The HTML5 technologies relevant to our work are presented in Section 3. Section 4 proposes a design of distributed data storage and outlines possible problems that will require further research. Possible applications are presented in Section 5 and Section 6 concludes the paper.

2 Related Work

Utilizing unused resources from desktop computers is not a new idea. Many systems, such as Entropia [3] or SETI@home [4], have successfully used idle CPU in the past. With new HTML5 technologies, this task become even easier. For instance a WeevilScout framework [5] was built to utilize computational power of the web browsers [6] for bioinformatic tasks.

In this work, we focus on utilizing the spare storage capacities of ordinary computers to form a distributed data storage. Distributed storage systems of the day have evolved from providing basic means of remote data, to offering services like high availability, anonymity, redundancy, and archival [7, 8].

Several projects already looked into scavenging unused storage on desktop computers. Farsite [9] is a replication-based serverless distributed file system built on Windows desktops that uses the Byzatine-fault protocol to manage file and directory metadata. Freeloader [10] framework aggregates unused desktop storage space into a shared cache space for hosting large scientific datasets. It utilizes file stripping across a set of machines in order to provide high data throughput. Storage@home [11] is a distributed storage infrastructure developed to solve the problem of backing up and sharing large data using a distributed model of volunteer managed hosts.

As a peer-to-peer storage system, CFS [12] stores data blocks reliably using distributed hash tables. It arranges blocks over a number of nodes to provide fault tolerance, high scalability, and load balancing. Ivy [13] is a file system using logs to saving both data and meta-data. The logs are saved in the distributed hash table across the unreliable peers. The Freenet [14] file sharing network stores anonymized documents and allows them to be retrieved later by an associated key. Files on Freenet are split into multiple small blocks, with additional blocks added to provide redundancy. Also peer-to-peer file sharing networks such as BitTorrent [15] or DC++ [16] can be considered as representants of such a class of distributed storage systems. Using decentralized network, the users share their files with other members.

Although all these storage systems proved their usability and applicability, they suffer from one substantial drawback from a user's point of view: they require to install a client software. To our best knowledge, no distributed storage system operates without the need of such additional client component.

3 HTML5 and New Web Technologies

In this section, we will briefly introduce the HTML5 technologies, which have emerged in the past few years and which may be used to create distributed data storage using web browsers of common users. These technologies are implemented in the leading desktop browsers, thus available for most web users.

3.1 Persistent Data Storage

The HTTP was designed as strictly stateless protocol, with no user session support. This flaw was partially remedied by introducing *cookies* – a simple mechanism, that allows server to register session data at client side, which are resent automatically with each request. Cookies do not satisfy complex needs of modern web applications, thus more elaborate solutions were devised in HTML5.

Web Storage. The web storage [17] was originally designed with the same motivation as cookies – to store structured data at client side. Unlike cookies, the web storage is completely maintained by JavaScript, so the programmer may use it in different ways. There are two types of web storage interfaces for two different scenarios – the session storage and the local storage.

The *session storage* is designed for scenarios when the application needs to store session data, but the user may run multiple sessions in different windows. Therefore, the storage is tied to a specific site and specific opened window/tab.

The *local storage* is designed for scenarios when the application needs to store data at the client side, but these data span throughout the opened windows. It is also particularly useful when the data needs to overlive the current user session. The storage is tied to a specific site and the data are persistent.

Both storages use simple API that allows saving key-value pairs, where both the key and the value are simple strings. The keys are treated as unique identifiers and the API allows the application to iterate over all stored keys. The specification also includes some security restriction. The most important restriction in our case is the disk quotas for the stored data, i.e., the browser does not permit the script to store more data then specified by the quota.

Indexed Database. More complex data management problems require more elaborate solutions. For this purpose, the W3 consortium presented the Index Database API [18], which provide basic database functionality for the client scripts. The IndexDB API replaces Web SQL Database API, which was deprecated by W3C, yet some implementations exist.

The indexed database API allows the JavaScript at client side to create indexed data stores and operate them in a transaction-safe way. The data are usually stored in efficient persistent data structures (such as B-trees), which support efficient insertion, deletion, key-based searching, and deterministic (ordered) enumeration.

3.2 Communication Capabilities

The first direct communication capabilities were introduced into client-side web scripting with the asynchronous HTTP API (`XMLHttpRequest`). This API allowed the client code to perform its own requests on the background, thus synchronize client and server application status without explicit user actions. However, the communication must still be initiated from the client side, which complicates situations when the server needs to notify clients.

One of the first techniques designed for server-initiated notifications (also denoted AJAX-push or reverse AJAX) was Comet [19]. This technique uses long-held HTTP requests that allow the server to initiate the transfer over a connection created by the client. The client sends an asynchronous HTTP request to the server demanding a status update. If no updates are available, the server postpones the response whilst the connection remains open. When the status changes, the server notifies all clients by sending the responses to all pending requests. Unfortunately, the long-held HTTP requests are not managed very well by current HTTP servers (e.g., Apache or Microsoft IIS).

An extension of the Comet approach was presented by the Server-sent events API. This API uses also long-held HTTP connections, but it allows multiple events to be sent over an opened HTTP request and standardizes both the message format and the client-side API that process these messages.

WebSocket. The WebSocket protocol is a HTTP-compatible protocol, which allows upgrading regular HTTP connections into a WebSocket connections. The WebSocket connection reuses underlying TCP (or SSL/TLS) channel of the original connection to transfer data frames in either direction. Both sides (former HTTP client and server) become equal communication partners, i.e., they can send a message over to their peer at any time.

The WebSocket client API [20] is quite simple. It handles only the creation and the destruction of the communication channel and sending/receiving messages. The protocol supports both textual and binary data, thus the JavaScript may send or receive strings, `Blob` objects, and `ArrayBuffer` objects.

WebRTC. The HTTP protocol and the WebSockets are designed for client-server communication. In some cases, a direct communication between clients may be required. WebRTC [21] is a technology, which was designed for real-time peer-to-peer communication between web browsers. The technology was originally intended for multimedial transfers (e.g., video phone calls), but it handles data transfers as well.

The browsers first create a `RTCPeerConnection`, which is a logical representation of the connection. This connection is usually routed using ICE technologies for NAT traversal, since most browsers are connected to the internet from a private network. The RTC connection allows the transfer of one or multiple streams. A stream may be either media stream, which can be directly captured from a web camera and directly displayed in an appropriate HTML5 element, or a data stream, which has the same behaviour and API as a WebSocket connection.

4 Web Browser as Distributed Storage Platform

A distributed data storage system (as almost any database system) usually employs layered architecture. Lower layers are responsible for raw data operations while upper layers add user-friendly functionality, such as transparent format transcoding, management of logical data entities, integrity constraints, transaction support, or security. These high-level features are quite well understood and beyond the scope of this work. Hence, we focus on the lowest level – the utilization of HTML5 technologies for raw data storage and transfer.

4.1 Infrastructure

The data storage infrastructure can be divided into three logical components:

- the master (running mostly on a server or on multiple servers),
- the clients (the connected web browsers),
- and the communication protocol they use.

The master controls the entire storage. It integrates all data distribution logic and initiates all data transactions. It can be implemented as one centralized server, as a small cluster of servers located in one server room, or even as a large distributed system. Theoretically, some parts of the master functionality can be performed in the browsers; however, these details are not important from the data storage point of view. Hence, we will present the master as a single server.

The most important role of *the clients* is to provide their storage space for the system. Some of these clients may also use the system functions (i.e., retrieve/store user data); however, we primarily focus on the data management. The clients require to execute only a minimalist script which connects to the master and waits for commands. All operations are issued from the master and the client simply process them. There are three basic kinds of operations:

- metadata operations (client identification, capabilities assessment),
- data retrieval (list, read),
- and data modifications (insert, update, delete).

The protocol is designed to work as a simple RPC for the client operations. The data transfers are bidirectional and server-initiated, thus we selected the WebSockets as the underlying protocol. Each operation has its own request message and response message. The client process the messages sequentially and for each incoming request generates one response. Figure 1 illustrates this model.

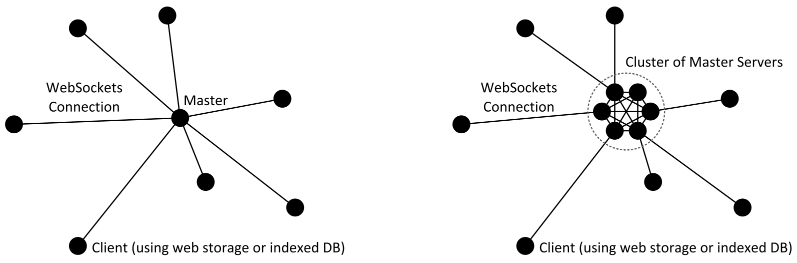
WebSockets Connection
Master
Client (using web storage or indexed DB)
Cluster of Master Servers
WebSockets Connection
Client (using web storage or indexed DB)

Fig. 1. Data storage with centralized data transfers

Optimizing Data Transfers. Even though the basic concept may suggest that the major data flow will be between master and clients, there are many situations, when the data needs to be transferred between clients directly. For instance, when a data block needs to be replicated from one nodes to another or when a client requests data, which are stored on adjacent nodes.

The WebRTC peer-to-peer connections may be used to optimize data transfers between the clients (browsers). The WebRTC creates data streams, which are operated by the same API as the WebSockets. Hence, it will be easy to adapt the implementation to accomodate this feature. Even though the WebRTC may reduce the master-client communication, the uplink to the master is still required, since the master controls the performed operations.

Figure 2 depicts the augmentation of centralized data storage where peer-to-peer data channels are used to optimize the data traffic between clients. The peer-to-peer channels are created on demand only between those clients that need to communicate and they may be closed when no longer needed.

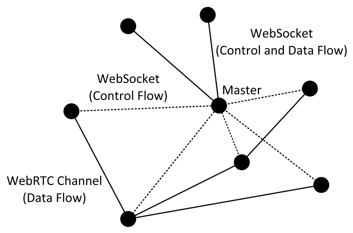

Fig. 2. Distributed data storage with peer-to-peer channels

4.2 Practical Observations

We have implemented the prototype of the data storage layer. The client was implemented as a simple web page which connects to the master using WebSocket [20] channel and handle commands sent by the server. Both web storage [17] and Indexed DB API [18] were tested for the client-side data management. The master was implemented as Node.js application that distributed data to the connected clients and verified the accessibility of the data.

Addressing all technical details is way beyond the scope of this work, but we have identified the most problematic issues and proposed solutions for them.

Resource Limitations. One of the key issues is the the limitation of user resources, especially the disk space. Current browser use the default quota of 5 MB per site for local storage [17]. This limit is to low to create large distributed data storage even if millions of users are participating. The quota can be configured in all major browsers, however, changing this setting is not very convenient. The IndexedDB API [18] is designed for significantly larger data, but it still apply some form of quotas. Fortunately, the IndexedDB apply only soft quotas and the browser interactively prompts the user when the limit is to be exceeded. Furthermore, a Quota API [22] is currently being specified by the W3 consortium, which should provide even more convenient way to manage the quotas.

Another resource that might limit the system is the network connectivity. The desktop computers are usually connected by some form of wired connection, such as xDSL (phone lines) or FTTx (dedicated metallic or optical lines). According to various resources, the average connection speeds are oscillating between 1 and 100 MBps depending on the location. Furthermore, we have to consider that many technologies (like the ADSL) use asymmetric connections, where the upload is several times slower than the download.

The storage and the data transfer issues are even more serious if we consider portable devices such as tablets or smartphones. Regular user hardly notices if we allocate one gigabyte on a desktop computer, but the same amount of data cannot be easily taken from a smartphone. Furthermore, mobile devices are usually connected via cellular networks which are much slower and often limit the data transfers by fair user policies.

Volatility of the Clients. Most of the distributed systems expect that the participating nodes are devoting their efforts to fulfill the objectives of the system. Although there are methods of dealing with many forms of failures, the system usually expects that the nodes are available most of the time. The web users, on the other hand, connect to the system at their own discretion without any regards to the system requirements.

The system must employ specific measures to ensure data availability. Prehaps the most straightforward solution is to select sufficient data replication level. More elaborate approach would be to study individual behaviour of the users and data requests. These strategies will require intensive future research.

Security. The system must provide some guarantees regarding the stored data, such as persistence, integrity, or privacy. Our distributed data store does not fulfill any of these requirements, since the clients do not provide any guarantees. The data may be easily corrupted, erased, or accessed by unauthorized users. Some of the security requirements may be fulfilled by employing check sums, security hashes, cryptography, and data replication. However, this topic is too broad and beyond the scope of this paper.

5 Web Storage Applications

We have projected initial estimates concerning the web storage capacity based on existing web application and outlined a few possible use cases to illustrate the applicability of the storage.

5.1 Capacity Analysis

We have based our analysis on the statistical data of Facebook [23], which is probably the largest web application as it has the most active users. At the beginning of year 2014, the Facebook had 1.31 billion active users, from which 51.9% used mobile or handheld devices to access the application. The number of users is expected to grow in the future as it has in the past (e.g., the annual growth between 2012-2013 was 22%). If we utilize 1 GB on average of each active desktop user, we could get approximately 600 PB of raw capacity.

Utilizing this raw capacity effectively will be quite a challenging task. Even the users of such popular application as the Facebook exhibit very irregular behaviour in visiting the web page. The active users spend about 640 million minutes each month, which is only one minute per user in average. On the other hand, 48% of all users connect to the web page on a daily basis.

These statistics suggest that the matter of data replication cannot be solved by a simple random approach, but careful study of user behaviour has to be conducted. Furthermore, the application may employ additional methods like benefits to encourage the users connect and stay on the web page.

5.2 Legitimate Usage

The data storage and sharing systems usually require some form of a client application. If such systems are built on the top of web technologies, they may provide clients which can be directly used in the web browser without explicit installation or configuration.

Existing applications, in which the users share some content, can use the distributed storage as a cache to reduce the data transfers of their internal servers. For instance, Facebook can use this cache for photographs, Google Doc for documents, etc. When the data are requested by a user, they may be downloaded from a peer who is online instead of downloading them from a server.

A different approach can be used by applications which have many users, but do not require storing or caching large amounts of data. These applications may offer the capacity of the distributed storage to third parties. The concept is similar to web advertisement where a web page shows ad banners, however in this case, the users support the application by donating their own disk space.

5.3 Parasitic Usage

The data storage can be also misused by malicious applications to steal the disk space from the users. For instance, the idea of acquiring the data storage at clients by the means of ad banners can be used both legitimately and parasitically, depending on whether the user is informed about the intent or not. However, when used parasitically, the application needs to use quite elaborate way of user targeting in order to create at least somewhat persistent data storage.

To acquire even more clients, the parasitic storage malware may resort to exploit vulnerabilities of other web applications. When a victim application is not correctly protected against script injection attacks [24], the parasitic application may inject the client code. The users of the victim application then become also clients of the parasitic distributed storage without realizing it.

6 Conclusions

We have proposed a novel idea how existing HTML5 technologies may be used or even misused for a distributed data storage. The storage clients require only modern web browser, which is a piece of software that is present on virtually every desktop computer. We have implemented prototype framework as a proof of concept and outlined additional problems that require our attention.

In the future work, we would like to address the remaining problem, especially analyze the user behaviour. If we are able to predict client up times based on behavioural models, we will be able to distribute data among the client nodes in more efficient way. Furthermore, we would like to explore possibilities of creating distributed database system, that will also distribute the query evaluation plans and execute them partially on the client nodes.

References

1. Hickson, I., Hyatt, D.: Html5. W3C Working Draft, May (2011)
2. Pichai, S., Upson, L.: Introducing the Google Chrome OS. The Official Google Blog (2009)
3. Chien, A., Calder, B., Elbert, S., Bhatia, K.: Entropia: architecture and performance of an enterprise desktop grid system. Journal of Parallel Distributed Computing **65** (2003) 597–610
4. Univ. of Berkeley: SETI@Home. http://setiathome.ssl.berkeley.edu/ (2006)
5. Reginald, C., Putra, G., Belloum, A., Koulouzis, S., Bubak, M., de Laat, C.: Distributed Computing on an Ensemble of Browsers. (2013)
6. W3C: Web Workers. http://www.w3.org/TR/workers/
7. Thanh, T., Mohan, S., Choi, E., Kim, P.: A taxonomy and survey on distributed file systems. NCM 08, Fourth International Conference on Networked Computing and Advanced Information Management (2008)
8. Hasan, R., Anwar, Z., Yurcik, W., Brumbaugh, L., Campbell, R.: A survey of peer-to-peer storage techniques for distributed file systems. International Conference on Information Technology: Coding and Computing, ITCC 2005 **2** (2005)
9. Adya, A., Bolosky, W., Castro, M., Cermak, G., Chaiken, R., Douceur, J., Howell, J., Lorch, J., Theimer, M., Wattenhofer, R.: Farsite: Federated, available, and reliable storage for an incompletely trusted environment. Proceedings of the 5th Symposium on Operating Systems Design and Implementation (2002)
10. Ma, X., Vazhkudai, S.S., Zhang, Z.: Improving data availability for better access performance: A study on caching scientific data on distributed desktop workstations. Journal of Grid Computing (2009)
11. Beberg, A.L., Pande, V.S.: Storage@home: Petascale Distributed Storage. Parallel and Distributed Processing Symposium (2007)
12. Dabek, F., Kaashoek, M., Karger, D., Morris, R., Stoica, I.: Wide-area cooperative storage with cfs. Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP 01) (2001)
13. Muthitacharoen, A., Morris, R., Gil, T., Chen, B.: Ivy: A read/write peer-to-peer file system. Proceedings of 5th Symposium on Operating Systems Design and Implementation (2002)
14. Clarke, I., Sandberg, O., Wiley, B., Hong, T.W.: Freenet: A Distributed Anonymous Information Storage and Retrieval System. Designing Privacy Enhancing Technologies. Lecture Notes in Computer Science (2001)
15. Pouwelse, J.e.a.: The bittorrent p2p file-sharing system: Measurements and analysis. Peer-to-Peer Systems (2008) 205–216
16. : DC++. http://dcplusplus.sourceforge.net/ (2013)
17. W3C: Web Storage. http://www.w3.org/TR/webstorage/
18. W3C: Indexed Database API. http://www.w3.org/TR/IndexedDB/
19. McCarthy, P., Crane, D.: Comet and Reverse Ajax: The Next-Generation Ajax 2.0. Apress (2008)
20. W3C: The WebSocket API. http://dev.w3.org/html5/websockets/
21. W3C: WebRTC 1.0: Real-time Communication Between Browsers. http://dev.w3.org/2011/webrtc/editor/webrtc.html
22. W3C: Quota Management API. https://dvcs.w3.org/hg/quota/raw-file/tip/Overview.html
23. Statistic Brain Research Institute: Facebook Statistics from 1.1.2014. http://www.statisticbrain.com/facebook-statistics/
24. Spett, K.: Cross-site scripting. SPI Labs (2005)