Return the Data to the Owner: A Browser-Based Peer-to-Peer Network

Dennis Boldt and Stefan Fischer
Institute of Telematics
University of Lübeck
Lübeck, Germany
Email: {boldt,fischer}@itm.uni-luebeck.de

***Abstract*—The paper covers the concept of a browser-based peer-to-peer network, which supports a decentralized, redundant and encrypted data storage. The core is a JavaScript-based socket API, which facilitates creating and accepting arbitrary TCP/IP connections from within a browser. This API builds upon a WebSocket SOCKS5 Proxy. This is essential, because the sandbox of a browser does not allow plain socket connections. We used this Socket API to implement, to the best of our knowledge, the first Browser-based peer-to-peer network based on the Chord protocol. Additionally, we implemented the first JavaScript-based forward error correction based on Reed-Solomon coding to handle the recovery of lost data. Our network circumvents user-generated content stored on powerful central servers operated by huge companies which allows the creation of user profiles, the placement of customized advertisements and a possible interface for intelligence agencies to access the central stored data. Our results show, that our approach works with reasonable performance for files up to 100 KB.**

Keywords-Browser-based peer-to-peer network; Berkeley Sockets API; SOCKS5; Chord; Reed-Solomon.

I. INTRODUCTION

For quite a while now, user-generated content web pages such as Wikipedia, Youtube, Twitter or Flickr are ubiquitous. Moreover, plenty of well-known desktop applications such as several widely-used office suites have been made usable through the browser interface. This works in a way that the owner of the web applications provide the platforms and the users are generating the corresponding content by accessing the applications through their web browsers (or specialized apps) and store it on the servers of the web applications' owners. Many users see this as the core problem: the server providers control the data in such a way, that they can create user profiles and provide customized advertisements. Moreover, they provide embeddable code snippets for like-buttons (e.g., Facebook and Google Plus), videos (e.g., Youtube) or messages (e.g., Twitter) which are embedded in millions of independent web pages [1]. Based on this, it is even more easy to generate perfect customized user profiles, especially if a single company provides dozens of heterogeneous applications which appear independent.

In our approach for a solution, we focus on the browser as an independent platform, because it is one of the most often used applications spread over all operating systems in the world. Furthermore, it is not per se necessary to install an additional runtime environment (e.g., Java or .NET Framework), because browsers support JavaScript out of the box. Finally the browsers are getting faster from release to release and new technologies such as HTML5 are better supported.

This paper provides a way to get rid of the dependency on web applications provided by huge companies. The user-generated data will be stored in an encrypted format, decentralized and redundant in the browsers of the users. The architecture we developed is a browser-based peer-to-peer (P2P) network based on well-known technologies and protocols such as Berkeley Sockets, SOCKS5, Chord, Reed-Solomon coding and HTML5 features like WebSockets.

The rest of this paper is organized as follows. Related work is introduced in Section II, the core architecture of the browser-based P2P network is proposed in Section III. The experimental results are provided in Section IV. Section V gives an outlook on future work. Section VI concludes the paper.

II. RELATED WORK

Classical applications in computer networks are based on client-server architectures, where a central server provides services which are used by many clients. In contrast there are peer-to-peer (P2P) networks, in which every participant is both client and server (peer) at the same time. In the last 15 years, plenty of P2P systems were developed. Well known examples are the first generation P2P networks Napster (1999) with a centralized lookup, or Gnutella (2000) with a decentralized flooding-based lookup.

Both approaches do not scale well which led to the development of a second generation of P2P networks like Tapestry [2] or Chord [3]. These approaches are more structured and, as a result, much more scalable. Also, P2P-based distributed storage systems were invented, where systems like OceanStore [4], Cooperative File System (CFS) or Wuala [5] were created. The latter is based on Network Coding [6].

A. Chord

A common structured P2P network protocol is Chord [3]. $P = \{p_1, \ldots, p_N\}$ is the set of N peers within the network. A hashing function h assigns to every peer $p \in P$ an unique ID $h(p) \in \{0 \ldots 2^m - 1\}$, where m should be sufficiently

 ISBN: 978-1-61208-361-2

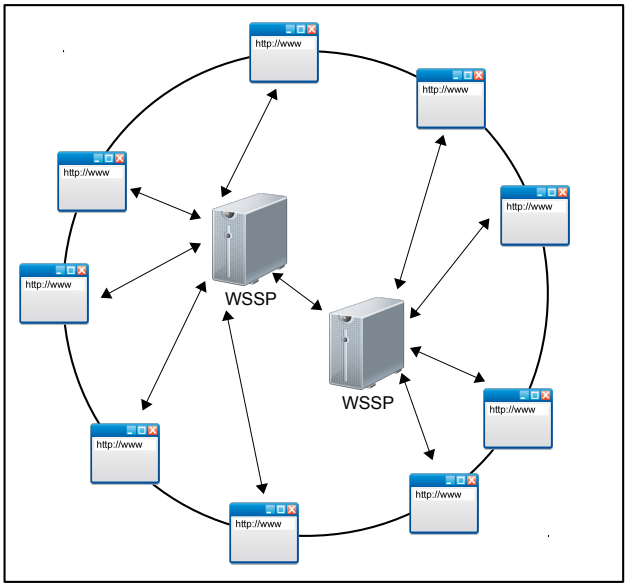

Figure 1: High level architecture: the browser-based peer-to-peer network based on a Chord-ring

huge. The ID for a peer is calculated by hashing the peer's IP address and port number, for example with SHA-1, where m is 160 bit. Based on these IDs, all peers are placed on a ring with a corresponding key space $\{0 \dots 2^m - 1\}$, the so-called Chord-ring. Based on this ring topology, peer p is in charge for the key space between the predecessor's ID and the own ID: $(h(pred(p)), h(p)]$.

The core operation of a P2P network is to find the node n which is in charge for a given m-bit key key; in Chord this can be expressed as $n = succ(key)$. This operation can easily be implemented by walking along the ring of peers in $O(N)$ steps. Chord improves this inefficient lookup to $log(N)$ steps with an additional routing table, the so-called *finger table*. Each finger table has m entries (fingers), where each $finger(i)$ bridges a distance of 2^{i-1} on the ring. Thus, the first finger is the successor and the last finger points to a peer which is at least half the ring away. The Chord protocol also handles joining and leaving of peers. To join to the network, a peer must know an existing peer from the network (bootstrapping). When a peer joins or leaves, the corresponding key spaces are changing.

All applications must be implemented on top of this routing protocol. Chord's core application is a *Distributed Hash Table* (DHT) for storing application data in the network. A DHT supports two operations, where *key* is a SHA-1 hash and *value* is an array of binary data:

1) *put(key, value)*
2) *value = get(key)*

Because the DHT and Chord are using the SHA-1 hashing function, both are using the same key space. Therefore, putting and getting data yields to the function *succ(key)* provided by Chord.

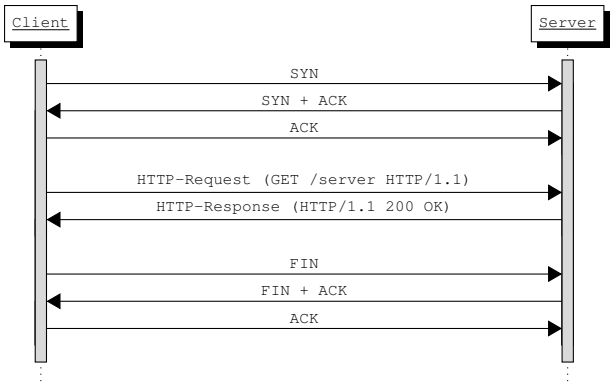

Figure 2: Schematic sequence diagram showing an XMLHttpRequest exchange.

B. WebRTC

WebRTC is a JavaScript API which enables web browsers to have real-time communications (RTC) [7]. WebRTC supports P2P audio and video communications. It also supports P2P data channels to send binary data between two peers. To create a P2P connection, WebRTC does a connection handshake between two peers over a signaling channel based on XMLHttpRequests or WebSockets using the Session Description Protocol (SDP) defined in RFC 4566 [8]. The WebRTC specification itself is still work in progress. To enable P2P connections between peers, some simple WebRTC-based P2P APIs like PeerJS [9] were developed. Current research topics on WebRTC are media streaming [10][11][12] and browser-based Content Delivery Networks (CDN) like Maygh [13], Tailgate [14] or PeerCDN [15].

III. System Architecture and Implementation

The architecture of our browser-based P2P network is shown in Figure 1. Because we implemented the Chord protocol, all browser-peers are placed on a ring. To set up the network, it has to be possible to create a socket connection to another browser, and to bind a socket to listen for incoming connections. This leads to the core problem: every browser-based application runs in a sandbox which has no capability to handle raw TCP/IP sockets. There are three ways for a browser to communicate to the world: *WebRTC*, *XMLHttpRequests* and *WebSockets*. The first was mentioned before, therefore we are focusing on the last two.

XMLHttpRequest Level 2 [16] (XHR) allows a web application to send HTTP requests to a server asynchronously. It is located on top of HTTP in the TCP/IP protocol stack. A connection starts with a kind of a HTTP-based handshake. Because HTTP is on top of TCP, this leads to a TCP handshake followed by an HTTP request. The HTTP response (typically containing XML, JSON, HTML, or binary data) from a server can be embedded in the application without reloading the web page. Because HTTP is a request-response protocol, a web application needs to request a resource continuously to get real time updates (polling). The message sequence for an XHR exchange is shown in Figure 2.

The WebSocket Protocol is defined in RFC 6455 [17] and allows real two-way communications. A connection starts

 ISBN: 978-1-61208-361-2

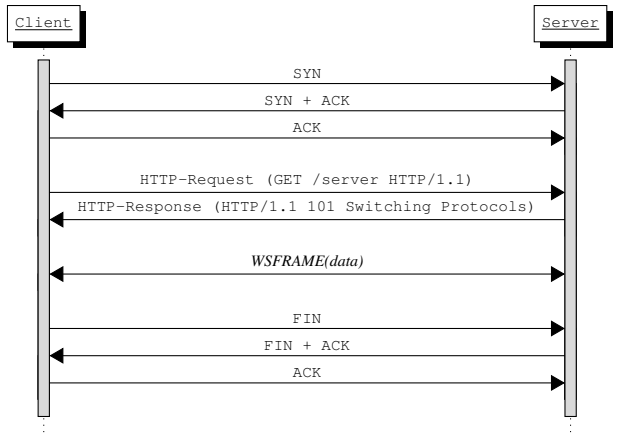

Figure 3: Schematic sequence diagram showing a connection with WebSockets

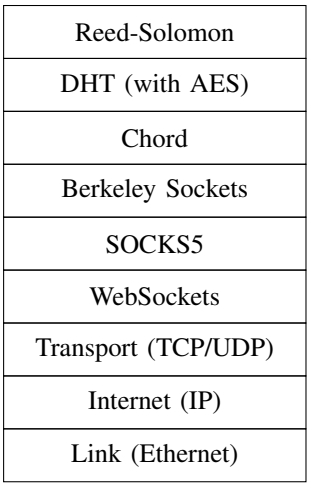

Figure 4: Low level architecture: Technology stack of the browser-based peer-to-peer network.

similar to a XHR connection. The difference is, that the HTTP response from the server signals that client and server are switching the protocol to WebSocket Frames (`HTTP/1.1 101 Switching Protocols`). These frames are similar to TCP segments. As soon the connection is initiated, client and server can send binary data, i.e., HTML5 Typed Arrays [18], to each other immediately. A typical WebSocket connection is shown in Figure 3.

The *SOCKS Protocol Version 5* (SOCKS5) is a protocol defined in RFC 1928 [19] to realize a proxy server. A proxy forwards connections of clients to servers and vise versa. SOCKS5 supports TCP/UDP, IPv4/IPv6 and authentication. It starts with an handshake to exchange the authentication methods with the *identifier/method selection message*. Afterwards a *socks request*, e.g., CONNECT/BIND request, with a given IP address and port number is send to the proxy which creates a connection or binds a socket. In the TCP/IP stack, SOCKS5 operates between the transport layer and the application layer.

A. The WebSocket SOCKS5 Proxy (WSSP)

The WSSP is the core component in our architecture which is implemented in Java and uses the Netty framework [20]. Netty supports zero-copy and uses New I/O (NIO), which is a non-blocking I/O based on Buffers and Channels. Netty perfectly fits in or architecture, because it already provides handlers for WebSockets. On top of the WebSockets, we implemented the SOCKS5 protocol to allow arbitrary TCP/IP connections.

Connect to a socket: To connect to a socket, a WebSocket connection to the WSSP must be established by a client. Afterwards, the WebSocket connection is used to send the method selection message followed by the CONNECT request. The WSSP connects to the given IP/port and sends a SOCKS5 reply over the WebSocket connection to the client. Finally, the connection is established.

Bind to a socket: To bind to a socket, a WebSocket connection to the WSSP must be established by a server. Afterwards, the WebSocket connection is used to send the method selection message followed by the BIND request. The WSSP binds to the given IP/port and sends a SOCKS5 reply over the existing WebSocket connection to the server and the binding is established. The Chord-ID is calculated by hashing this IP and port with SHA-1. On an incoming connection, the WSSP sends a new SOCKS5 reply over the existing WebSocket connection to the server. The server creates a new WebSocket connection to WSSP. The WSSP binds this new WebSocket connection to the incoming connection and a separate channel for each connection is established.

B. JavaScript APIs

In the following, we are going through the layers of our technology stack shown in Figure 4. We assume the link, internet and transport layer to be well-known and we focus on the application layers on top of the transport layer.

1) The WebSocket API: The WebSocket API [21] implements the WebSocket Protocol, which is located on top of HTTP; after switching the protocol on top of TCP/IP. The WebSocket API is supported by all common browsers.

2) JavaScript SOCKS5 API: Together with the WSSP, we implemented an appropriate JavaScript SOCKS5 API, which builds upon WebSocket API. With this API it is possible to create socket connections and to listen for incoming connections within sandboxed web applications.

3) JavaScript Berkeley Sockets API: The *Berkeley Sockets API* (BSD API) [22] provides well-known functions like `socket()`, `connect()`, `bind()` or `close()` to enable inter-process communication (IPC) for any application via TCP/IP. Figure 5 shows how this functions can be used by a server to bind to a socket and for a client to connect to a socket. The original BSD API is written in C and is part of every UNIX system. The BSD API is situated in the TCP/IP stack between the transport layer and the application layer as well. We adopted the BSD API on top of the JavaScript SOCKS5 API. With our JavaScript Berkeley Sockets API it is easy to create IPC within the browser.

 ISBN: 978-1-61208-361-2

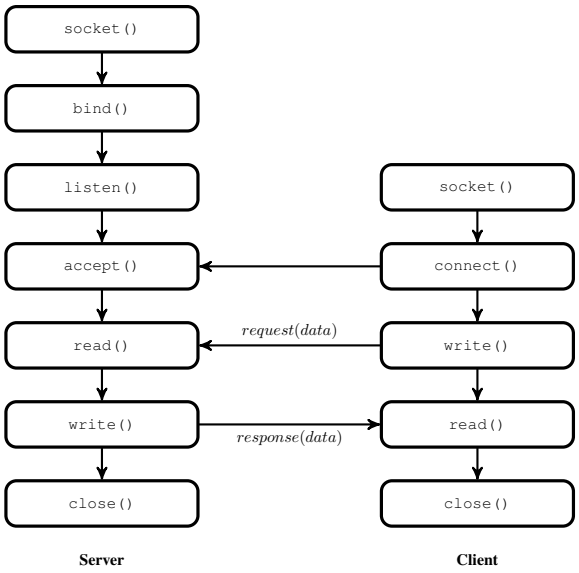

Figure 5: Client and Server with the Berkeley Socket API (based on [23])

4) JavaScript Chord API: With the *JavaScript Chord API* we implemented, to the best of our knowledge, the world-wide first JavaScript-based P2P network on top of the JavaScript Berkeley Sockets API. To join the network, we implemented a simple lookup server where all nodes which have joint are registered.

5) JavaScript Distributed Hash Table API: Our JavaScript Distributed Hash Table API uses the JavaScript Chord API to provide the two DHT operation *put* and *get*. It supports symmetric encryption and decryption of data using AES with a given *key_symm*, e.g., a password. This symmetric key can differ from data to data. The encryption and decryption are calculated on the client side. Therefore, the password never leaves the computer of the client. The key, which represents the storage position of the data in the Chord-ring, can be determined by calculating the SHA-1 hash of the data. The packet format can be seen in Listing 1. For transferring, this packet is converted into binary format using the library binarize.js [24]. The data are stored in-memory by using a JavaScript object.

```
{
  key : sha1(data),
  value : aes_enc(data, key_symm)
}
```

Listing 1: The DHT packet in JSON format. It contains the key and the encrypted data.

6) JavaScript Reed-Solomon API: We showed, how we store the application data in a decentralized and encrypted way in the users' browsers. In huge networks, nodes typically join and leave rather frequently. It is also possible that a node fails. In this case, all data stored at this node is lost. To ensure the availability of all data stored in the P2P network, identical copies located on different nodes, e.g., replicates located at different hashes, can be used. It is quite unrealistic to upload the same file multiple times to the network. Even the probability of losing all replicates is quite high. Therefore, our JavaScript Reed-Solomon API uses the Reed-Solomon Coding defined as polynoms on finite fields [25][26][27]. It is a so-called erasure code, which does forward error corrections on binary data. A file is split in n data parts. Based on these n data parts, m linear independent recovery parts are calculated. The coding uses Galois Fields – $GF(2^8)$ – for binary data, on which the mathematical operations $+$, $-$, $*$, $/$ are defined. To recover the whole file, just a subset of n arbitrary parts from all data and recovery parts is needed. This coding was first used for CDs.

IV. Experimental Results

Imagine the following use case: an user or a web application wants to store/load a file, e.g., a picture, within/from the DHT (see Figure 6). A password can be selected.

Putting data into the DHT: First of all, the application calculates a corresponding SHA-1 hash. Afterwards the Reed-Solomon encoding is called, which splits the file in n data parts and calculates m recovery parts. If a password is selected, these parts are encrypted using AES. Otherwise this step is skipped. Finally all (encrypted) parts are uploaded to the P2P network using the DHT.

Getting data from the DHT: Obtaining a file works the opposite way: an application downloads n arbitrary parts from the P2P network using the DHT. If a password is given, the AES decryption is called, which returns the decrypted parts. Then the whole file is decoded using Reed-Solomon. Finally the SHA-1 hashsum can be calculate to check the integrity.

We ran our experiments in-memory with the browsers Firefox 27, Chrome 33 and the server-side environment Node.js 0.10.21. Firefox is based on the JavaScript engine *SpiderMonkey* [28], while the last two are based on the *V8 JavaScript Engine* [29]. Node.js allows to run JavaScript as a standalone application without the need of a browser. Because of this, it also provides raw socket access to the application.

For our experiments we used a Dell Latitude E6420 with an Intel Core i5-2520M CPU with 2.50 GHz and 8 GB RAM. The file size in the experiments varies from 1 Byte up to 1 GB.

A. Network Independent Evaluation

The calculation of SHA-1 hashing values, AES encryptions or Reed-Solomon encodings for big files is quite CPU-intensive. This leads to the problem that the browsers are freezing while an API is working. To remedy this issue, the *Web Worker API* [30] was introduced. This API provides independent threads to JavaScript applications for long-running calculations. We use Web Worker within the browsers for the aforementioned APIs, which are computing intensive algorithms. We also use already existing implementations for SHA-1 [31] and AES [32], the latter is implemented in counter mode. We implemented the Reed-Solomon coding.

From Figures 7 to 9 we can see, that the duration for almost all network independent experiments is slowest in Firefox. Node.js is the fastest for small files, because it does not use Web Worker. When the file size increases, Chrome becomes

 ISBN: 978-1-61208-361-2

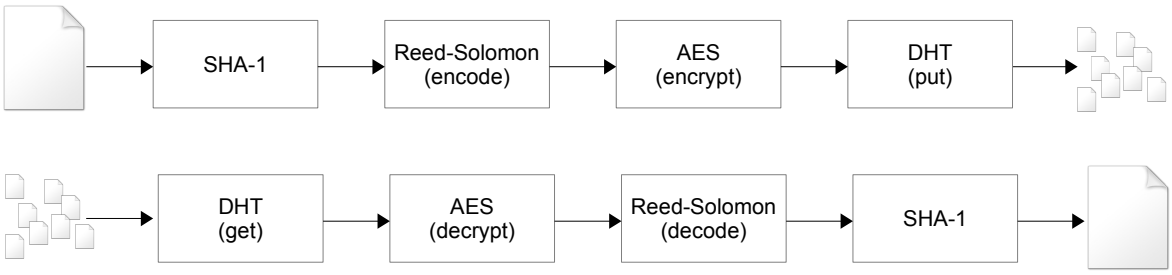

Figure 6: Workflow to put/get a file into/from the DHT.

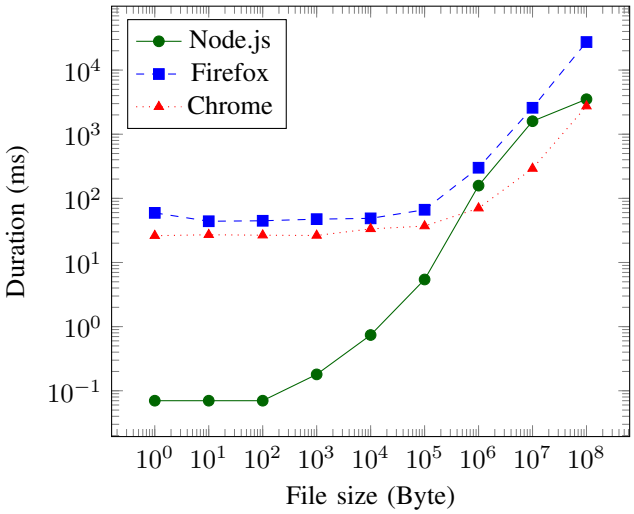

Figure 7: SHA-1 hashsums calculation for 100 files.

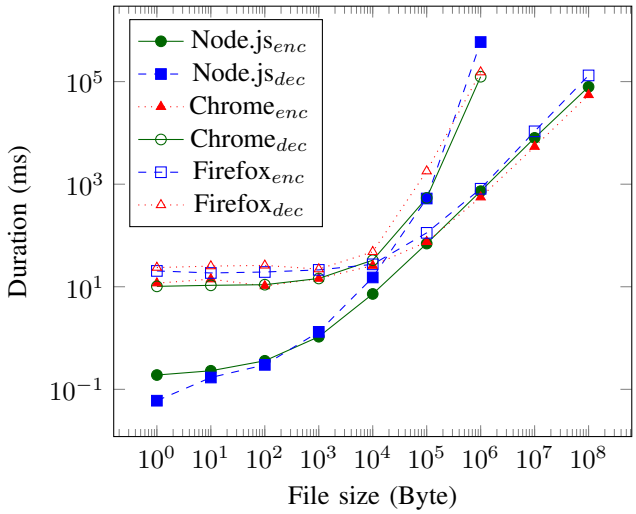

Figure 8: Encrypt/decrypt AES for 100 files with a key size of 256 bits.

faster than Node.js, even though both are using the same JavaScript engine. Figure 7 shows that Chrome is faster than Node.js for SHA-1 hashsum calculations for files bigger than 10 MB. The result for AES with key size of 256 bits can be seen in Figure 8. Up to a file size of 100 KB, encryption and decryption take almost the same time on all platforms (up to 1,7 seconds). For a file of size 1 MB the decryption varies from 2 to 6 minutes, which is not acceptable. Therefore, it is much faster to split a file in smaller parts of maximum 100 KB before the encryption step, than encoding the whole file. This fits in our architecture, because the splitting is done by the Reed-Solomon coding already. Figure 9 shows the result of the Reed-Solomon coding. With file sizes smaller than 1 MB, the coding does not need more than one second.

B. Network Evaluation

First of all we measured the round-trip time (RTT) of a packet with ICMP and afterwards we created a WebSocket connection to the WSSP. We observed that the duration of a WebSocket connection just depends on the RTT of the network. For example: the duration for a WebSocket connection is 80ms. Then the RTT is 40ms, because the WebSocket protocol needs two RTTs to set up a connection: one for the TCP handshake and one for the HTTP handshake (see Figure 3). We ran the following experiments just in Node.js, because the duration does not differ between Firefox, Chrome and Node.js.

We evaluated our WSSP-based approach, which is used in usual browsers, against a version with direct socket access provided by Node js. For this, we implemented a simple ECHO-server, which copies and returns the received message. We created a Chord-ring with two nodes, connected to the same WSSP with a RTT of 40ms.

WSSP-approach: The binding of a socket by a server takes 180ms. This results in four RTTs (160ms): two for the WebSocket connection to the WSSP and two for the SOCKS5 binding. The remaining 20ms are processing time by Node.js and the WSSP. A connection by the client to the bound socket takes 360ms. This yields to 8 RTTs (320ms): two to create the WebSocket connection and two for the SOCKS5 connect request by the client; two for the new WebSocket connection and two for the SOCKS5 request by the server to bind the incoming connection (see Section III-A). The remaining 40ms are processing time by Node.js and the WSSP. The ECHO of the data takes 85ms, which corresponds to 2 RTTs: one between the client and the WSSP and one between the WSSP and the server.

Raw sockets: With raw sockets we get rid of the WebSocket connections and the SOCKS5 messages. The binding happens immediately and the connection needs just one RTT (40ms).

The putting/getting of data with DHT just depends on the

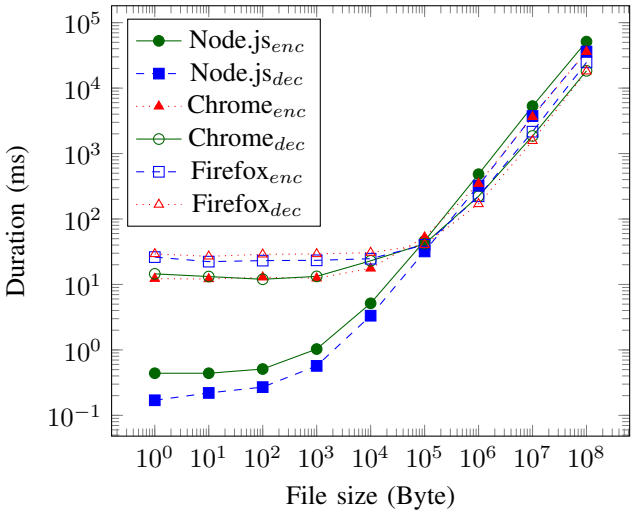

Figure 9: Encode/decode Reed-Solomon for 100 files with $n = 10$ and $m = 10$.

connection duration from our previous experiment and the network bandwidth. Therefore, we did not run a DHT-based experiment, because it will just show the maximum network bandwidth.

V. Future Work

Future work aims to improve the performance, the bootstrapping, the maintenance and the security.

Performance: To increase the performance, we are working on a WebRTC-based P2P network, even if it does not allow arbitrary TCP/IP connections like our approach. Therefore, we expect better experimental results, e.g., less RTTs, in the future.

Bootstrapping of the network: One big challenge is the bootstrapping of the network, because in the Chord protocol every node needs an entry node to join the network. The current implementation uses a simple lookup server where all active nodes are registered. This obviously does not scale in a network with millions of nodes.

Security: Currently we are using symmetric encryption of the data with AES. Thus, sender and receiver must know the *key_symm*, e.g., the password. An improvement will be the usage of asymmetric keys, e.g., RSA [33]. With this, we can update our packet format, where *key_async* can be the own public key (storage of data for the own purpose) or the private key (storage of data for public purpose). Like this, the *key_symm* is stored in the DHT packet itself (see Listing 2). This allow using random symmetric keys for every data part.

Maintenance of the data: In the current implementation, data stored at a node (DHT) is moved, if a node joins or leaves the network. This is a core feature of Chord. This does not scale, if a user must wait until all data, e.g., some hundred MB, are moved. Usually this leads to the failure of the node and the loss of the in-memory stored data (this also include a change of the IP address). Therefore, a challenge is the maintenance of the data stored in the P2P network. The network needs

```
{
  key : sha1(data),
  value : [
      aes_enc(data, key_symm),
      rsa_enc(key_symm, key_async)
    ]
  }
}
```

Listing 2: The improved DHT packet in JSON format. It contains the key, the encrypted data and the encrypted symmetric key.

to make sure that all data are always available, especially in the future, after millions of node joins, leaves and fails. To handle this, we already use the Reed-Solomon coding. Also the Reed-Solomon parts are lost over the time. If less than n parts are available, a Reed-Solomon encoded file cannot be decoded. Therefore, we need to maintain the Reed-Solomon parts, if the availability of a file is vulnerable, e.g., recovering the parts.

VI. Conclusion

In this paper, we introduced the approach of a browser-based P2P network, which is a possible platform to return the data to the owner. The experiments showed that our approach works with reasonable performance for files up to 100 KB, which fits the usual web traffic. Bigger files are split in smaller parts to keep the performance. The performance of the JavaScript engines of all browsers is going to be improved, while the limitation of the computing intensive algorithms SHA-1, AES and Reed-Solomon is mainly the CPU.

Acknowledgements

We would like to thank Philipp Abraham, Tobias Braun, Florian Burmann, Marvin Frick, Bennet Gerlach, Syavoosh Khabbazzadeh, Florian Lau and Dennis Pfisterer for helpful comments, debugging and testing our implementation.

References

[1] J. Schmidt, “Das Like-Problem (The like problem),” Heise Security, 04 2011. [Online]. Available: http://heise.de/-1230906 [Retrieved: May, 2014]

[2] B. Zhao, J. Kubiatowicz, and A. D. Joseph, “Tapestry: An infrastructure for fault-tolerant wide-area location and routing,” University of California Berkeley, Computer Science Department, Tech. Rep. UCB Technical Report UCB/CSD-01-1141, 04 2001.

[3] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, “Chord: A scalable peer-to-peer lookup service for internet applications,” in ACM SIGCOMM Computer Communication Review, vol. 31, no. 4. ACM, 2001, pp. 149–160.

[4] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton et al., “Oceanstore: An architecture for global-scale persistent storage,” in Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems, ser. ASPLOS IX. New York, NY, USA: ACM, 2000, pp. 190–201.

[5] “Wuala - Secure Cloud Storage.” [Online]. Available: http://www.wuala.com [Retrieved: May, 2014]

[6] M. Martalo, M. Picone, R. Bussandri, and M. Amoretti, “A practical network coding approach for peer-to-peer distributed storage,” in IEEE International Symposium on Network Coding (NetCod). IEEE, 2010, pp. 1–6.

 ISBN: 978-1-61208-361-2

[7] A. Bergkvist, D. C. Burnett, C. Jennings, and A. Narayanan, "WebRTC 1.0: Real-time communication between browsers," W3C Working Draft, WD-webrtc-20130910, Sep. 2013. [Online]. Available: http://www.w3.org/TR/webrtc/ [Retrieved: May, 2014]

[8] M. Handley, V. Jacobson, and C. Perkins, "SDP: Session Description Protocol," RFC 4566 (Proposed Standard), Internet Engineering Task Force, Jul. 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4566.txt [Retrieved: May, 2014]

[9] M. Bu and E. Zhang, "PeerJS - Simple peer-to-peer with WebRTC." [Online]. Available: http://peerjs.com [Retrieved: May, 2014]

[10] F. Rhinow, P. P. Veloso, C. Puyelo, S. Barrett, and E. O. Nuallain, "P2p live video streaming in webrtc."

[11] J. K. Nurminen, A. J. Meyn, E. Jalonen, Y. Raivio, and R. G. Marrero, "P2P media streaming with HTML5 and WebRTC," in IEEE International Conference on Computer Communications. IEEE, 2013.

[12] A. J. Meyn, "Browser to Browser Media Streaming with HTML5," Master's thesis, Technical University of Denmark, Lyngby, Denmark, 2012.

[13] L. Zhang, F. Zhou, A. Mislove, and R. Sundaram, "Maygh: Building a cdn from client web browsers," in Proceedings of the 8th ACM European Conference on Computer Systems. ACM, 2013, pp. 281–294.

[14] S. Traverso, K. Huguenin, I. Trestian, V. Erramilli, N. Laoutaris et al., "Tailgate: handling long-tail content with a little help from friends," in Proceedings of the 21st international conference on World Wide Web. ACM, 2012, pp. 151–160.

[15] J. Wu, Z. Lu, B. Liu, and S. Zhang, "PeerCDN: A novel p2p network assisted streaming content delivery network scheme," in Computer and Information Technology, 2008. CIT 2008. 8th IEEE International Conference on. IEEE, 2008, pp. 601–606.

[16] A. van Kesteren, "XMLHttpRequest Level 2," W3C Working Draft, WD-XMLHttpRequest-20120117, Mar. 2012, retrieved: May, 2014. [Online]. Available: http://www.w3.org/TR/XMLHttpRequest2/

[17] I. Fette and A. Melnikov, "The WebSocket Protocol," RFC 6455 (Proposed Standard), Internet Engineering Task Force, Dec. 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6455.txt [Retrieved: May, 2014]

[18] D. Herman and K. Russell, "Typed array specification," Khronos.org, 07 2011. [Online]. Available: https://www.khronos.org/registry/typedarray/specs/latest/ [Retrieved: May, 2014]

[19] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones, "SOCKS Protocol Version 5," RFC 1928 (Proposed Standard), Internet Engineering Task Force, Mar. 1996. [Online]. Available: http://www.ietf.org/rfc/rfc1928.txt [Retrieved: May, 2014]

[20] "Netty project." [Online]. Available: http://netty.io/ [Retrieved: May, 2014]

[21] I. Hickson, "The Websocket API," W3C Candidate Recommendation, CR-websockets-20120920, Sep. 2012. [Online]. Available: http://www.w3.org/TR/websockets/ [Retrieved: May, 2014]

[22] W. R. Stevens, UNIX network programming. Addison-Wesley Professional, 2004, vol. 1.

[23] S. Markey, "Manage mobile cloud socket connections," 01 2013. [Online]. Available: http://www.ibm.com/developerworks/cloud/library/cl-mobilesockconnect [Retrieved: May, 2014]

[24] E. Kitamura, "binarize.js – binarize arbitrary js object into ArrayBuffer." [Online]. Available: https://github.com/agektmr/binarize.js [Retrieved: May, 2014]

[25] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," Journal of the Society for Industrial & Applied Mathematics, vol. 8, no. 2, 1960, pp. 300–304.

[26] J. S. Plank, "A tutorial on reed-solomon coding for fault-tolerance in raid-like systems," Softw., Pract. Exper., vol. 27, no. 9, 1997, pp. 995–1012.

[27] J. S. Plank and Y. Ding, "Note: Correction to the 1997 tutorial on reed–solomon coding," Software: Practice and Experience, vol. 35, no. 2, 2005, pp. 189–194.

[28] "SpiderMonkey." [Online]. Available: https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey [Retrieved: May, 2014]

[29] "V8 JavaScript Engine." [Online]. Available: http://code.google.com/p/v8/ [Retrieved: May, 2014]

[30] I. Hickson, "Web Workers," W3C Candidate Recommendation, CR-workers-20120501, May 2012. [Online]. Available: http://www.w3.org/TR/workers/ [Retrieved: May, 2014]

[31] "JavaScript sha1 function." [Online]. Available: http://phpjs.org/functions/sha1/ [Retrieved: May, 2014]

[32] C. Veness, "JavaScript Implementation of AES Advanced Encryption Standard in Counter Mode." [Online]. Available: http://www.movable-type.co.uk/scripts/aes.html [Retrieved: May, 2014]

[33] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," Communications of the ACM, vol. 21, no. 2, 1978, pp. 120–126.

 ISBN: 978-1-61208-361-2