Practical Web-based Delta Synchronization for Cloud Storage Services

He Xiao, Tsinghua University
Zhenhua Li, Tsinghua University
Ennan Zhai, Yale University
Tianyin Xu, UCSD

Abstract

Delta synchronization (sync) is known to be crucial for network-level efficiency of cloud storage services (*e.g.,* Dropbox). *Practical* delta sync techniques are, however, only available for PC clients and mobile apps, but not web browsers—the most pervasive and OS-independent access method. To understand obstacles of web-based delta sync, we implemented a traditional delta sync solution (named WebRsync) for web browsers using JavaScript, and find that WebRsync severely suffers from the inefficiency of JavaScript execution inside web browsers, thus leading to frequent stagnation and even crashing. Given that the computation burden on the web browser mainly stems from data chunk search and comparison, we reverse the traditional delta sync approach by lifting all chunk search and comparison operations from the client side into the server side. Inevitably, this brings enormous computation overhead to the servers. Hence, we further leverage locality matching and a more efficient checksum to reduce the overhead. The resulting solution (called WebR2sync+) outpaces WebRsync by an order of magnitude, and it is able to simultaneously support ~7300 web clients' delta sync using an ordinary VM server based on a Dropbox-like system architecture.

1 Introduction

Recent years have witnessed enormous popularity of cloud storage services, such as Dropbox, Google Drive, iCloud Drive, and Microsoft OneDrive. They have not only provided a convenient and pervasive data store for billions of Internet users [5], but also become a critical component of numerous online applications (*e.g.,* Dropbox's support for DocuSign, Google Drive's support for Gmail, and OneDrive's support for Office 365).

The popularity of cloud storage services inevitably brings tremendous network traffic overhead to both the client and cloud sides [15]. Therefore, a lot of efforts have been made to improve the network-level efficiency of cloud storage services, including batched synchronization (sync), deferred sync, delta sync, compression and deduplication [12, 14, 17, 18]. Among these efforts, delta sync is known to be of particular importance for its fine granularity (*i.e.,* the client only sends the changed content of a file to the cloud), thus achieving significant traffic savings in the presence of users' file edits [19].

Unfortunately, delta sync is currently only practical for PC clients and mobile apps, but not web browsers—the most pervasive and OS-independent access method [17]. For example, after a file f is edited into a new version f' by the user, Dropbox's PC client or mobile app only uploads the altered bits to the cloud; in contrast, the web browser has to upload the whole content of f' to the cloud. This gap severely affects web-based user experiences in terms of both sync performance and traffic cost.

To understand the potential obstacles of web-based delta sync, we implement a delta sync solution (referred to as WebRsync) for web browsers using JavaScript based on `rsync` [7], the de facto delta sync protocol for PC clients. Also, we develop an automated tool (called StagMeter) to accurately measure the stagnation of web browsers. Our experimental results show that WebRsync severely suffers from the inefficiency of JavaScript running inside web browsers. Under typical file editing workloads, WebRsync is slower than PC client-based delta sync by up to 25 times, thus causing web browsers to frequently freeze and even crash.

Specifically, when a user edits a file from f to f', WebRsync first requests the server side to execute (data) chunk *segmentation* and *fingerprinting* operations on f, and then requests the client side to perform chunk *search* and *comparison* operations on f'. During the process, the computation overhead on the client side is larger than that on the server side by around 7 times. More in detail, the client-side computation burden mainly stems from chunk search (~65%) and comparison (~22%).

Motivated by the above observations, our first effort is to "reverse" the WebRsync process by handing all chunk search and comparison operations over to the server side. Meanwhile, chunk segmentation and fingerprinting operations are shifted to the client side. The resulting solution is referred to as WebR2sync, denoting web-based reverse `rsync` (more details are described in §3.1 and Figure 4).

Although WebR2sync significantly cuts the computation burden on the web client (and thus effectively avoids stagnation/crashing), it brings enormous computation overhead to the server side. To this end, we make two-fold additional efforts to optimize the server-side computation overhead. First, we exploit the locality of

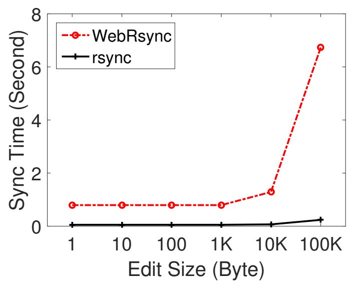

Figure 1: Average sync time *vs.* edit size to a typical text file.

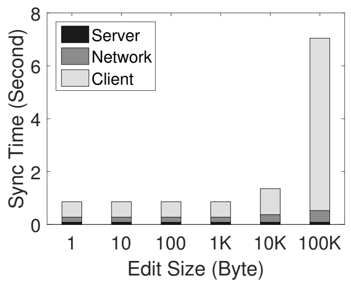

Figure 2: Sync time decomposition for WebRsync.

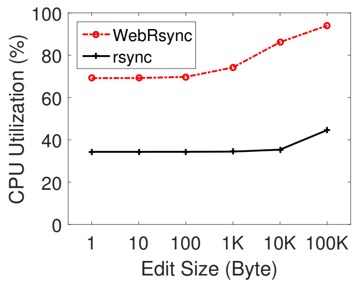

Figure 3: Average CPU utilization *vs.* edit size to a typical text file.

users' file edits which can help bypass most (∼90%) chunk search operations in real usage scenarios. This can reduce nearly a half of server-side computation overhead. Second, by replacing the relatively secure, expensive MD5 algorithm with the more efficient, pseudorandom SipHash algorithm, we can reduce the complexity of chunk comparison by around 6 times.

Through the above efforts, our final solution (named WebR2sync+) outpaces WebRsync by around an order of magnitude, approaching the performance of PC client-based `rsync`. Also, it is able to simultaneously support ∼7300 web clients' delta sync using an ordinary VM server based on a Dropbox-like system architecture. This throughput (∼7300) is as 4 times as that of WebR2sync and as 9 times as that of NoWebRsync [1]. All source codes of WebRsync (including StagMeter), WebR2sync and WebR2sync+ are publicly available at `https://WebDeltaSync.github.io`.

2 Motivating Study

To quantitatively understand the reason why web-based delta sync is not supported by today's cloud storage services, we implement the WebRsync solution and measure its performance using the StagMeter tool as follows.

WebRsync. We implement WebRsync by adapting the working procedure of `rsync` (demonstrated in Figure 4(a)) to the web browser scenario. Moreover, the architecture of WebRsync follows Dropbox's system architecture. Specifically, we implement the client side based on the HTML5 File APIs [6] and the WebSocket protocol, using 1500 lines of JavaScript code. The server side is developed based on the node.js framework, using 500 lines of node.js code and 600 lines of C++ code. Similar to Dropbox (on the server side), the web service runs on a VM server hosted on Aliyun ECS [2], and the file content is stored in object storage hosted on Aliyun OSS [3]. More details on the server, client and network configurations are described in §4.1 and Figure 7.

To compare the performance of WebRsync and `rsync`, we do a random edit (an insertion or a deletion) with different sizes on a *typical* text file every 10 seconds. Here *typical* means that we use a real-world cloud storage dataset released in [17], where the average file size is nearly 1 MB and the median file size is 7.5 KB. As shown in Figure 1, the sync time of WebRsync is significantly longer than that of `rsync` by 14–25 times. In other words, WebRsync is much slower than `rsync` on handling the same file edit. Further, we break down the sync time of WebRsync into three stages as depicted in Figure 2. Clearly, the vast majority of sync time is spent at the client side, indicating that the slowness of WebRsync is owing to the inefficiency of the browser. Additionally, Figure 3 shows that the CPU utilization of WebRsync almost doubles that of `rsync`, because JavaScript programs consume enormous CPU resources.

In summary, WebRsync costs not only more sync time but also more computation resources on the client side, thus causing the web browser to frequently stagnate and even crash. Here "stagnate" means that the browser does not react to user actions (*e.g.,* mouse clicks) in time, and "crash" means that the browser never reacts to user actions. We further develop the StagMeter tool as follows to measure the stagnation of the browser.

StagMeter. Although stagnation of the browser can be perceived by users, they can hardly be quantified. Thus, we automate the measurement of stagnation time by integrating a snippet of JavaScript code (referred to as StagMeter) into the browser. The snippet periodically[2] prints the current timestamp on the concerned web page (*e.g.,* the web page that executes delta sync). If the current timestamp (say *t*) is successfully printed at the moment,

[1]NoWebRsync refers to the approach that uploads the entire file for synchronization without delta sync. Note that NoWebRsync is the common approach for web-based access adopted by current cloud storage services such as Dropbox, SugarSync and iCloud Drive.

[2]By default we set the period as 100 ms, so as to simulate the minimum intervals of common web users' operations.

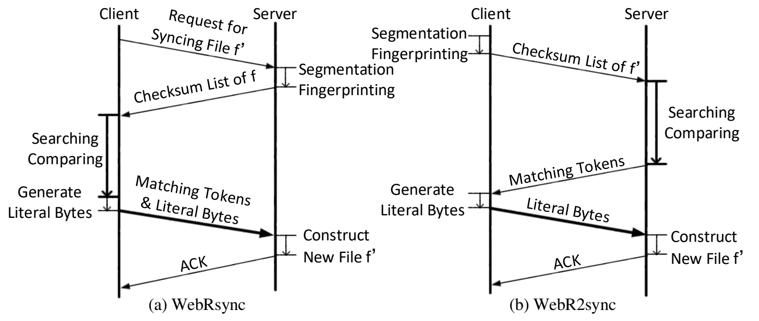

Figure 4: Flow charts of WebRsync and WebR2sync.

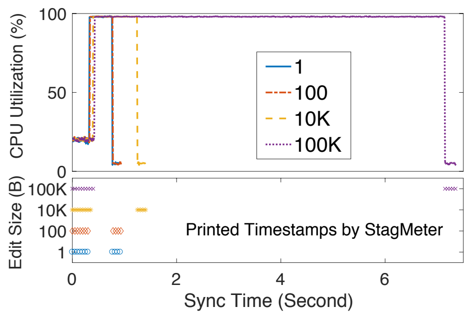

Figure 5: Printed timestamps for the first second and the associated CPU utilization.

there is no stagnation; otherwise, there is a stagnation and then the printing of the current timestamp will be postponed to $t' > t$. Thereby, the corresponding stagnation time is calculated as $t' - t$.

Using StagMeter, we measure and visualize the stagnation of WebRsync (on handling various file-edit workloads) in Figure 5. Here StagMeter only attempts to print 10 timestamps for the first second. Therefore, spaces between consecutive timestamps represent stagnation, and larger spaces imply more severe stagnation. Meanwhile, as indicated in Figure 5, stagnation cases are directly associated with high CPU utilization.

3 Design and Implementation

In this section, we first present the WebR2sync solution which implements the reverse process of WebRsync, and then describe the server-side optimizations for mitigating the computation overhead on servers.

3.1 WebR2sync

Before we describe the design of WebR2sync, we first review the work flow of WebRsync as a comparison. As demonstrated in Figure 4(a), in WebRsync when a user edits a file from f to f', the client instantly sends a request to the server for the file synchronization. On receiving the request, the server first executes (data) chunk *segmentation* and *fingerprinting* operations on f (which is available on the cloud side), and then returns a *checksum list* of f to the client. Except for the last chunk, each data chunk is 8 KB in size. Thus when f is 1 MB in size, its checksum list contains 128 weak rolling 32-bit checksums as well as 128 strong 128-bit MD5 checksums [7]. After that, based on the checksum list of f, the client first performs chunk *search* and *comparison* operations on f', and then generates both the *matching tokens* and *literal bytes*. The matching tokens indicate the overlap between f and f', while the literal bytes represent the novel parts in f' relative to f. Both of them are sent to the server for constructing f'. Finally, the server returns an acknowledgment to the client to conclude the process.

As depicted in Figure 4(b), WebR2sync implements the reverse process of WebRsync by handing computation intensive search and comparison operations over at the server side, and meanwhile shifting the lightweight segmentation and fingerprinting operations to the client side. Accordingly, the checksum list of f' is generated by the client and the matching tokens are generated by the server, while the literal bytes are still generated by the client. Note that on the server side, the search and comparison operations are implemented in C/C++ rather than in JavaScript. Therefore, WebR2sync can not only avoid stagnation/crashing for the web client, but also effectively shorten the duration of the whole sync process.

3.2 Server-side Optimization

Although WebR2sync significantly reduce the computation burden on the web client, it brings enormous computation overhead to the server side. We make two-fold efforts to optimize the server-side computation overhead.

Exploiting locality of file edits in chunk search. When the server receives a checksum list from the client, WebR2sync uses a 3-level chunk searching scheme to figure out matched chunks between f and f', as demonstrated in Figure 6 (following the 3-level chunk searching scheme of `rsync` [7]). Specifically, in the checksum list of f' there is a 32-bit weak rolling checksum (calculated by the Adler32 algorithm [13]) as well as a 128-bit strong MD5 checksum for each data chunk in f'. When this checksum list is sent to the server, the server leverages an additional *(rolling checksum) hash table* where every entry is a 16-bit hash code of the 32-bit rolling checksum [7]. The checksum list is then sorted according to the 16-bit hash code of the 32-bit rolling checksums. Note that a 16-bit hash code can point to multiple rolling/MD5 checksums. To find each matched chunk

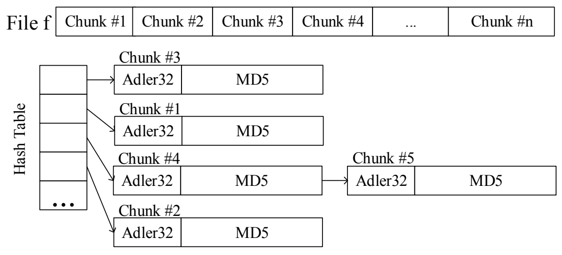

Figure 6: The 3-level chunk searching scheme used by `rsync` and WebR2sync.

between f and f', the 3-level chunk searching scheme always goes from the 16-bit hash code to the 32-bit rolling checksum and further to the 128-bit MD5 checksum.

The 3-level chunk searching scheme can effectively minimize the computation overhead for general file-edit patterns, particularly random edits to a file. However, it has been observed that real-world file edits typically follow a local pattern rather a general/random pattern [20, 24–26]. This offers an opportunity to bypass a considerable portion of (unnecessary) chunk search operations. In essence, give that edits to a file are typically local, when we find that the i-th chunk of f' matches the j-th chunk of f, the $(i+1)$-th chunk of f' is highly likely to match the $(j+1)$-th chunk of f. Therefore, we "simplify" the 3-level chunk searching scheme by directly comparing the MD5 checksums of the $(i+1)$-th chunk of f' and the $(j+1)$-th chunk of f. If the two chunks are identical, we move forward to the next chunk; otherwise, we return to the regular 3-level chunk searching scheme.

Replacing MD5 with SipHash in chunk comparison. Besides exploiting locality, we notice that the majority of server-side computation overhead is attributed to the calculations of MD5 checksums. Thus, we wonder whether the usage of MD5 is necessary in chunk comparison.

MD5 was initially designed as a cryptographic hash function for generating secure and low-collision hash code [23], which makes it quite computation intensive. In our scenario, however, it is not necessary to use such a computationally expensive hash function, because our purpose is just to obtain a low collision probability. In fact, we can employ the HTTPS protocol for data exchange between the web client and server to ensure the security. Driven by this insight, we decide to replace MD5 with a lightweight pseudorandom hash function [11] in order to reduce the computational overhead.

Quite a few pseudorandom hash functions can satisfy our goal, such as Spooky [16], FNV [21], CityHash [22], SipHash [10], and Murmur3 [9]. Among them, some are very lightweight but have high collision probabilities. For example, the computation overhead of MD5 is around 5 to 6 cycles per byte [4] while the computation

Table 1: A comparison of pseudorandom hash functions

Hash Function	Collision Probability	Cycles per Byte
MD5	Low	5.58
Murmur3	High	0.33
CityHash	High	0.23
FVN	High	1.75
Spooky	High	0.14
SipHash	Low	1.13

overhead of CityHash is merely 0.23 cycle per byte [8], but the collision probabilities of CityHash is quite high. On the other hand, some pseudorandom hash functions have extremely low collision probabilities but are slow. As listed in Table 1, SipHash is a sweet spot—its computation overhead is about 1.13 cycles per byte and its collision probability is acceptable (keep safe under flood attack). By replacing MD5 with SipHash in our web-based delta sync solution, we manage to reduce the computation complexity of chunk comparison by nearly 6 times.

3.3 Implementation

The client side of WebR2sync+ is implemented based on the HTML5 File APIs, the WebSocket protocol, and a Javascript implementation of SipHash-2-4 [1], with 1700 lines of JavaScript code in total. The server side of WebR2sync+ is developed based on the node.js framework and C++ processing module. The former (500 lines of node.js code) handles the user requests, and the latter (1000 lines of C++ code) embodies the reverse delta sync process together with the two-fold optimizations.

4 Evaluation

In this section, we show the performance and overhead of WebR2sync+ in comparison to WebRsync, WebR2sync and PC client-based `rsync` under typical workloads.

4.1 Experiment Setup

Like WebRsync, the server side of WebR2sync+ adopts a Dropbox-like system architecture by running the web service on a VM (with a quad-core Intel Xeon CPU @2.5GHz and 16-GB memory) hosted on Aliyun ECS, and all file content is stored in object storage hosted on Aliyun OSS. The ECS VM and OSS storage locate at the same data center so there is no bottleneck between them. Also, the client side of WebR2sync+ is integrated into the Google Chrome browser (Windows version 56.0) running on a laptop with a quad-core Intel Core-i5 CPU @2.8GHz, 16-GB memory, and an SSD disk.

More in detail, the server side and client side lie in different cities (*i.e.,* Shanghai and Beijing) and different ISPs (*i.e.,* China Unicom and CERNET), as depicted in

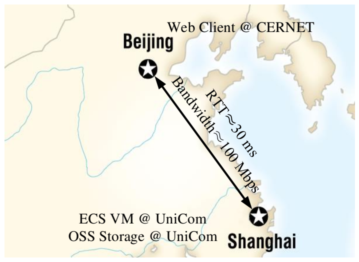

Figure 7: Basic experiment setup visualized in a map of China.

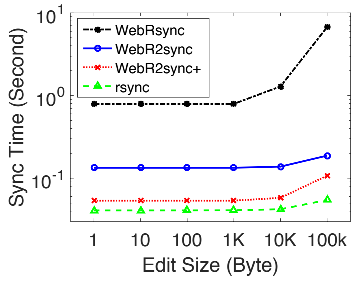

Figure 8: Average sync time *vs.* edit size to a typical text file.

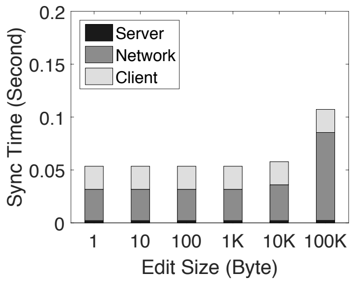

Figure 9: Sync time decomposition for WebR2sync+.

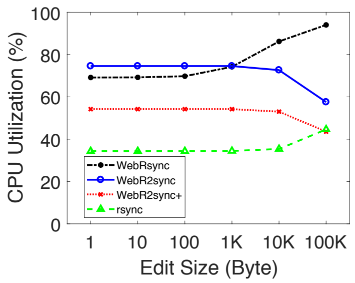

Figure 10: Client-side average CPU utilization *vs.* edit size to a typical text file.

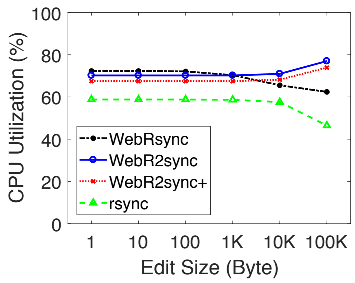

Figure 11: Server-side average CPU utilization *vs.* edit size to a typical text file.

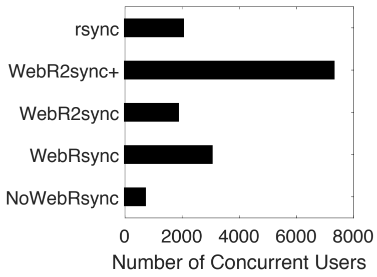

Figure 12: Number of concurrent users supported by a single VM server under typical workloads.

Figure 7. Their network RTT is ∼30 ms and their network bandwidth is ∼100 Mbps. To evaluate the real-world performance of WebR2sync+ (mentioned in §2), we do a random edit with different sizes to a typical text file every 10 seconds, thus emulating various workloads of practical users' file edits [17].

4.2 Results

Synchronization time. We measure the sync time of WebR2sync+. As demonstrated in Figure 8, the sync time of WebR2sync+ is substantially shorter than that of WebR2sync (by 2 to 3 times) and WebRsync (by 15 to 20 times). In other words, WebR2sync+ outpaces WebRsync by an order of magnitude; it has comparable performance as PC client-based `rsync`.

Further, we break down the sync time of WebR2sync+ into three stages, as shown in Figure 9. Comparing Figures 9 and 2 leads to two major observations. First, the majority of sync time is spent at the client side for WebRsync, while it is spent for network latency for WebR2sync+. Therefore, the web browser does not stagnate or crash. Second, the server-side sync time is still much shorter than the client-side sync time, which confirms the efficacy of our server-side optimizations.

CPU utilization. We show the client-side and server-side CPU utilizations in Figure 10 and Figure 11 respectively. On the client side, WebR2sync+ consumes less CPU resources than WebRsync and WebR2sync, while PC client-based `rsync` consumes the least CPU resources. On the server side, the CPU utilizations of all solutions are similar.

Throughput. WebR2sync+ can simultaneously support ∼7300 web clients' delta sync using an ordinary VM server under typical workloads (refer to Figure 12). This throughput is as 4 times as that of WebR2sync/`rsync` and as 9 times as that of NoWebRsync.

5 Future Work

In the near future, we plan to evaluate the performance of WebR2sync+ in more aspects and details, such as the traffic overhead, deduplication and delta-encoding rates, and the efficiency of the three optimizations (*i.e.,* reverse `rsync`, locality matching, and a weaker checksum).

6 Acknowledgments

This work is supported by the High-Tech Research and Development Program of China ("863–China Cloud" Major Program) under grant 2015AA01A201, the National Natural Science Foundation of China (NSFC) under grants 61471217, 61432002 and 61632020, and the CCF-Tencent Open Fund under grant RAGR20160105.

References

[1] A Javascript Implementation of SipHash-2-4. https://github.com/jedisct1/siphash-js.

[2] Aliyun ECS (Elastic Compute Service). https://www.aliyun.com/product/ECS.

[3] Aliyun OSS (Object Storage Service). https://www.aliyun.com/product/OSS.

[4] eBACS: ECRYPT Benchmarking of Cryptographic Systems. https://bench.cr.yp.to/results-hash.html.

[5] Internet users in the world. http://www.internetlivestats.com/internet-users/.

[6] Reading Files in JavaScript Using the File APIs. https://www.html5rocks.com/en/tutorials/file/dndfiles/.

[7] rsync Web Site. http://www.samba.org/rsync.

[8] J. Alakuijala, B. Cox, and J. Wassenberg. Fast Keyed Hash/Pseudo-random Function Using SIMD Multiply and Permute. *arXiv preprint arXiv:1612.06257*, 2016.

[9] A. Appleby. Murmur3 hash function. https://github.com/aappleby/smhasher.

[10] J.-P. Aumasson and D. J. Bernstein. SipHash: a Fast Short-input PRF. In *International Conference on Cryptology in India*, pages 489–508. Springer, 2012.

[11] M. Bellare, R. Canetti, and H. Krawczyk. Keying Hash Functions for Message Authentication. In *Annual International Cryptology Conference*, pages 1–15. Springer, 1996.

[12] Y. Cui, Z. Lai, X. Wang, N. Dai, and C. Miao. QuickSync: Improving Synchronization Efficiency for Mobile Cloud Storage Services. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking (MobiCom)*, pages 592–603. ACM, 2015.

[13] P. Deutsch and J.-L. Gailly. Zlib Compressed Data Format Specification Version 3.3. Technical report, RFC Network Working Group, 1996.

[14] I. Drago, E. Bocchi, M. Mellia, H. Slatman, and A. Pras. Benchmarking Personal Cloud Storage. In *Proceedings of the 13th ACM Internet Measurement Conference (IMC)*, pages 205–212. ACM, 2013.

[15] I. Drago, M. Mellia, M. Munafò, A. Sperotto, R. Sadre, and A. Pras. Inside Dropbox: Understanding Personal Cloud Storage Services. In *Proceedings of the 12th ACM Internet Measurement Conference (IMC)*, pages 481–494. ACM, 2012.

[16] B. Jenkins. Spookyhash: A 128-Bit Noncryptographic Hash, 2012. http://burtleburtle.net/bob/hash/spooky.html.

[17] Z. Li, C. Jin, T. Xu, C. Wilson, Y. Liu, L. Cheng, Y. Liu, Y. Dai, and Z.-L. Zhang. Towards Network-level Efficiency for Cloud Storage Services. In *Proceedings of the 14th ACM Internet Measurement Conference (IMC)*, pages 115–128. ACM, 2014.

[18] Z. Li, X. Wang, N. Huang, M. A. Kaafar, Z. Li, J. Zhou, G. Xie, and P. Steenkiste. An Empirical Analysis of a Large-scale Mobile Cloud Storage Service. In *Proceedings of the 16th ACM Internet Measurement Conference (IMC)*, pages 287–301. ACM, 2016.

[19] Z. Li, C. Wilson, Z. Jiang, Y. Liu, B. Zhao, C. Jin, Z.-L. Zhang, and Y. Dai. Efficient Batched Synchronization in Dropbox-like Cloud Storage Services. In *Proceedings of the 14th ACM/IFIP/USENIX International Middleware Conference (Middleware)*, pages 307–327. Springer, 2013.

[20] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezis, and P. Camble. Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, pages 111–123, 2009.

[21] L. C. Noll. FNV Hash. http://www.isthe.com/chongo/tech/comp/fnv/.

[22] G. Pike and J. Alakuijala. Introducing Cityhash. http://google-opensource.blogspot.com/2011/04/introducingcityhash.html.

[23] R. L. Rivest et al. RFC 1321: The MD5 Message-digest Algorithm. *Internet activities board*, 143, 1992.

[24] W. Xia, H. Jiang, D. Feng, and Y. Hua. SiLo: A Similarity-Locality based Near-Exact Deduplication Scheme with Low RAM Overhead and High Throughput. In *Proceedings of the 2011 USENIX*

Annual Technical Conference (ATC), pages 26–28, 2011.

[25] W. Xia, H. Jiang, D. Feng, and Y. Hua. Similarity and Locality Based Indexing for High Performance Data Deduplication. *IEEE Transactions on Computers*, 64(4):1162–1176, 2015.

[26] W. Xia, Y. Zhou, H. Jiang, D. Feng, Y. Hua, Y. Hu, Q. Liu, and Y. Zhang. FastCDC: a Fast and Efficient Content-defined Chunking Approach for Data Deduplication. In *Proceedings of the 2016 USENIX Annual Technical Conference (ATC)*, pages 101–114, 2016.