Towards Web-based Delta Synchronization for Cloud Storage Services

He Xiao
Tsinghua University

Zhenhua Li *
Tsinghua University

Ennan Zhai
Yale University

Tianyin Xu
UIUC

Yang Li
Tsinghua University

Yunhao Liu
Tsinghua University

Quanlu Zhang
Microsoft Research

Yao Liu
SUNY Binghamton

Abstract

Delta synchronization (sync) is crucial for network-level efficiency of cloud storage services. 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 the obstacles of web-based delta sync, we implement a delta sync solution, WebRsync, using state-of-the-art web techniques based on `rsync`, the de facto delta sync protocol for PC clients. Our measurements show that WebRsync severely suffers from the inefficiency of JavaScript execution inside web browsers, thus leading to frequent stagnation and even hanging. 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 to the server side. Inevitably, this brings considerable computation overhead to the servers. Hence, we further leverage locality-aware chunk matching and lightweight checksum algorithms to reduce the overhead. The resulting solution, WebR2sync+, outpaces WebRsync by an order of magnitude, and is able to simultaneously support 6800–8500 web clients' delta sync using a standard VM server instance based on a Dropbox-like system architecture.

1 Introduction

Recent years have witnessed considerable popularity of cloud storage services, such as Dropbox, SugarSync, Google Drive, iCloud Drive, and Microsoft OneDrive. They have not only provided a convenient and pervasive data store for billions of Internet users, but also become a critical component of other online applications. Their popularity brings a large volume of network traffic overhead to both the client and cloud sides [28, 37]. Thus, a lot of efforts have been made to improve their network-level efficiency, such as batched sync, deferred sync, delta sync, compression and deduplication [24, 25, 27, 37, 38, 46]. Among these efforts, delta sync is of particular importance for its fine granularity (*i.e.,* the client only sends the changed content of a file to the cloud, instead of the entire file), thus achieving significant traffic savings in the presence of users' file edits [29, 39, 40].

Unfortunately, today delta sync is only available for PC clients and mobile apps, but not for the web—the most pervasive and OS-independent access method [37]. After a file f is edited into a new version f' by users, Dropbox's PC client will apply delta sync to automatically upload only the altered bits to the cloud; in contrast, Dropbox's web interface requires users to manually upload the *entire content* of f' to the cloud.[1] This gap significantly affects web-based user experiences in terms of both sync speed and traffic cost.

Web is a fairly popular access method for cloud storage services: all the major cloud storage services support web-based access, while only providing PC clients and mobile apps for a limited set of OS distributions and devices. One reason is that many users do not want to install PC clients or mobile apps on their devices to avoid the extra storage and CPU/memory overhead; in comparison, almost every device has web browsers. Specially, for the emerging cloud-oriented systems and devices (*e.g.,* Chrome OS and Chromebook) web browsers are perhaps the only option to access cloud storage.

To understand the fundamental obstacles of web-based delta sync, we implement a delta sync solution, WebRsync, using state-of-the-art web techniques including JavaScript, WebSocket, and HTML5 File APIs [14, 18]. WebRsync implements the algorithm of `rsync` [15], the de facto delta sync protocol for PC clients, and works with all modern web browsers that support HTML5. To optimize the execution of JavaScript, we use asm.js [4] to first implement the client side of WebRsync in efficient C code and then compile it to JavaScript. To unravel the performance of WebRsync from the users' perspective, we further develop StagMeter, an automated tool for accurately quantifying the *stagnation* of web browsers, *i.e.,* the browser's not responding to user actions (*e.g.,* mouse clicks) in time, when applying WebRsync.

Our experiments show that WebRsync is severely af-

*Corresponding author. Email: lizhenhua1983@gmail.com

[1] In this paper, we focus on *pervasive* file editing made by any applications that synchronize files to the cloud storage through web browsers, rather than *specific* web-based file editors such as Google Docs, Microsoft Word Online, Overleaf, and GitHub online editor. Technically, our measurements show that the latter usually leverages specific data structures (rather than delta sync) to avoid full-content transfer and save the network traffic incurred by file editing.

fected by the low execution efficiency of JavaScript inside web browsers. Even under simple (or says one-shot) file editing workloads, WebRsync is slower than PC client-based delta sync by 16–35 times, and most time is spent at the client side for performing computation-intensive chunk *search* and *comparison* operations.[2] This causes web browsers to frequently stagnate and even *hang* (*i.e.,* the browser never reacts to user actions). Also, we find that the drawback of WebRsync cannot be fundamentally addressed through native extension, parallelism, or client-side optimization (§4).

Driven by above observations, our first effort towards practical web-based delta sync is to "reverse" the WebRsync process by handing all chunk search and comparison operations over to the server side. This effort also enables us to re-implement these computation-intensive operations in efficient C code. The resulting solution is named WebR2sync (Web-based Reverse `rsync`). It significantly cuts the computation burden on the web client, but brings considerable 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 users' file edits which can help bypass most (up to ∼90%) chunk search operations in real usage scenarios. Second, by leveraging lightweight checksum algorithms, SipHash [20] and Spooky [17] instead of MD5, we can reduce the complexity of chunk comparison by ∼5 times. The final solution is referred to as WebR2sync+, and we make the source code of all our developed solutions publicly available at `https://WebDeltaSync.github.io`.

We evaluate the performance of WebR2sync+ using a deployed benchmark system based on a Dropbox-like system architecture. We show that WebR2sync+ outpaces WebRsync by an order of magnitude, approaching the performance of PC client-based `rsync`. Moreover, WebR2sync+ is able to simultaneously support 6800–8500 web clients' delta sync using a standard VM server instance under regular workloads [3]. Even under intensive workloads, a standard VM instance with WebR2sync+ deployed can simultaneously support 740 web clients.

2 Delta Sync Support in State-of-the-Art Cloud Storage Services

In this section, we present our qualitative study of delta sync support in state-of-the-art cloud storage services. The target services are selected for either their popularity (Dropbox, Google Drive, Microsoft OneDrive, iCloud Drive, and Box.com), or representativeness in terms of techniques used (SugarSync, Seafile, QuickSync, and DeltaCFS). For each service, we examined its delta sync support with different access methods, using its latest-version (as of April 2017) Windows PC client, Android app, and Chrome web browser. The only exception occurred to iCloud Drive for which we used its latest-version MacOS client, iOS app, and Safari web browser.

Service	PC Client	Mobile App	Web Browser
Dropbox	Yes	No	No
Google Drive	No	No	No
OneDrive	No	No	No
iCloud Drive	Yes	No	No
Box.com	No	No	No
SugarSync	Yes	No	No
Seafile [16]	Yes	No	No
QuickSync [25]	Yes	Yes	No
DeltaCFS [51]	Yes	Yes	No

Table 1: Delta sync support in 9 cloud storage services.

To examine a specific service with a specific access method, we first uploaded a 1-MB [4] highly-compressed new file (f) to the cloud (so the resulting network traffic would be slightly larger than 1 MB). Next, on the user side, we appended a single byte to f to generate an updated file f'. Afterwards, we synchronized f' from the user to the cloud with the specific access method, and meanwhile recorded the network traffic consumption. In this way, we can reveal if delta sync is applied by measuring the traffic consumption—if the traffic consumption was larger than 1 MB, the service did not adopt delta sync; otherwise (*i.e.,* the traffic consumption was just tens of KBs), the service had implemented delta sync.

Based on the examination results listed in Table 1, we have the following observations. First, delta sync has been widely adopted in the majority of PC clients of cloud storage services. On the other hand, it has never been used by the mobile apps of any popular cloud storage services, though two academic services [25,51] have implemented delta sync in their mobile apps and proved the efficacy. In fact, as the battery capacity and energy efficiency of mobile apps grow constantly, we expect delta sync to be widely adopted by mobile apps in the near future [36]. Finally, none of the studied cloud storage services supports *web-based* delta sync, despite web browsers constituting the most pervasive and OS-independent method for accessing Internet services.

3 WebRsync: The First Endeavor

WebRsync is the first workable implementation of web-based delta sync for cloud storage services. It is implemented in JavaScript based on HTML5 File APIs [18] and WebSocket. It follows the algorithm of `rsync` and thus keeps the same behavior as PC client-based ap-

[2] In contrast, when a user downloads a file from the cloud with a web browser, the client-side computation burden of delta sync is fairly low and thus would not cause the web browser to stagnate or hang.

[3] Detailed description of simple, regular, and intensive workloads we use in this work is presented in §6.2.

[4] We also experiment with files much larger than 1 MB in size, *i.e.,* 10 MB and 100 MB, and got the same results.

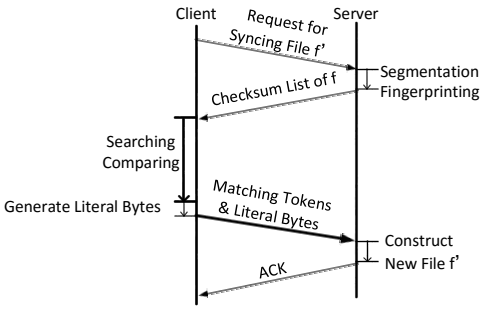


Figure 1: Design flow chart of WebRsync.

proaches. Although it is not a practically acceptable solution, it points out the challenges and opportunities of supporting delta sync under current web frameworks.

3.1 Design and Implementation

We design WebRsync by adapting the working procedure of `rsync` to the web browser scenario. As demonstrated in Figure 1, 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 fixed-size 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 typically 8 KB in size. Thus when f is 1 MB in size, its checksum list contains 128 weak 32-bit checksums as well as 128 strong 128-bit MD5 checksums [15]. 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*. Note that search and comparison operations are both conducted in a byte-by-byte manner on *rolling* checksums; in comparison, segmentation and fingerprinting operations are both conducted in a chunk-by-chunk manner so they incur much lower computation overhead. 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.

We implement the client side of WebRsync based on the HTML5 File APIs [18] and the WebSocket protocol, using 1500 lines of JavaScript code. Following the common practice to optimize the performance of JavaScript execution, we adopt the asm.js language [4] to first write the client side of WebRsync in C code and then compile it to JavaScript. The server side of WebRsync is developed based on the node.js framework, with 500 lines of node.js code and 600 lines of C code; its architecture follows the server architecture of Dropbox (as an example of the state-of-the-art industrial cloud storage services). Similar to Dropbox, the web service of WebRsync runs on a VM server rent from Aliyun ECS [2], and the file content is hosted on object storage rent from Aliyun OSS [3]. More details on the server, client and network configurations are described in §6.1 and Figure 14.

3.2 Performance Benchmarking

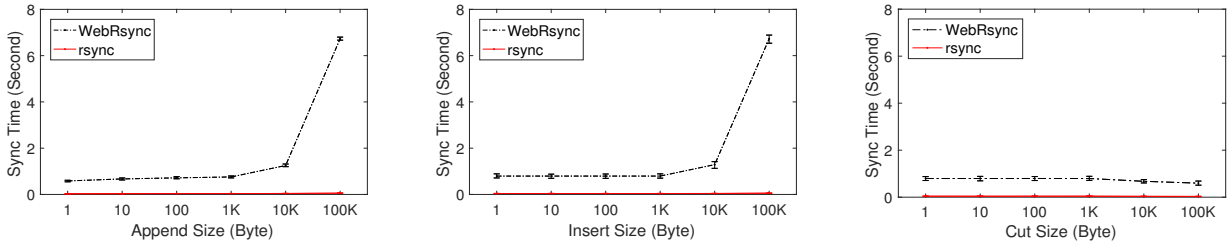
We first compare the performance of WebRsync and `rsync`. We perform random append, insert, and cut [5] operations of different edit sizes (ranging from 1 B, 10 B, 100 B, 1 KB, 10 KB, to 100 KB) upon real-world files collected from real-world cloud storage services. The dataset is collected in our previous work and is publicly released [37], where the average file size is nearly 1 MB. One file is edited for only once, and it is then synchronized from the client side to the server side. For an insert or cut operation, when its edit size reaches or exceeds 1 KB, it is first dispersed into a certain number of (typically 1–20) *continuous sub-edits* [6] to simulate the practical situation of a user edit, and then synchronized to the server. For each of the three different types of edit operations, we first measure its average sync time corresponding to each edit size, and then decompose the average sync time into three stages: server, network, and client. Moreover, we measure its average CPU utilization on the client side corresponding to each edit size.

As shown in Figure 2, for each type of file edit operations the sync time of WebRsync is significantly longer than that of `rsync` (by 16–35 times). In other words, WebRsync is much slower than `rsync` on handling the same file edit. Among the three types of file edits, we notice that syncing a cut operation with WebRsync is always faster than syncing an append/insert operation (for the same edit size), especially when the edit size is relatively large (10 KB or 100 KB). This is because a cut operation reduces the length of a file while an append/insert operation increases the length of a file.

Furthermore, we decompose the sync time of `rsync` and WebRsync into three stages: at the client side, across the network, and at the server side, as depicted in Figures 3a and 3b. For each type of file edits, around 40% of `rsync`'s sync time is spent at the client side and around 35% is spent at the network side; in comparison, the vast majority (60%–92%) of WebRsync's sync time is spent at the client side, while less than 5% is spent at the network side. This indicates that the sync bottleneck of WebRsync is due to the inefficiency of the web browser's executing JavaScript code. Additionally, Figure 3c illustrates that the CPU utilization of each type of file edits in WebRsync is as nearly twice as that of `rsync`, because JavaScript programs consume more CPU resources.

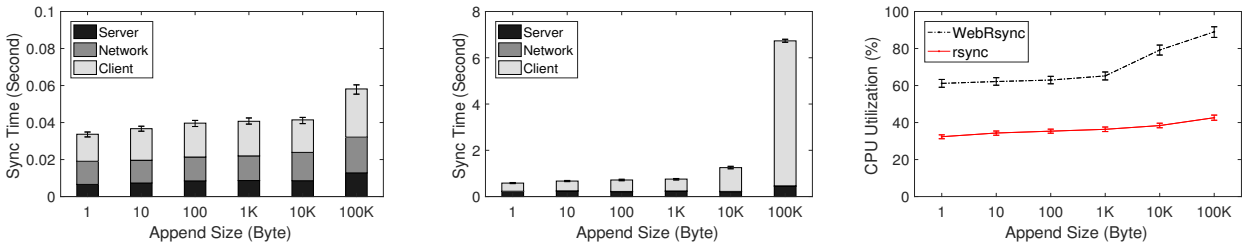
[5] Here "cut" means to remove some bytes from a file.

[6] A *continuous sub-edit* means that the sub-edit operation happens to continuous bytes in the file. More details are explained in § 5.2, especially in Figure 12 and Figure 13.



(a) Append. (b) Insert. (c) Cut.

Figure 2: Average sync time using WebRsync for various sizes of file edits (including append, insert, and cut) under a simple workload. The error bars show the minimum and maximum values at each point.



(a) Breakdown of the sync time of `rsync`. (b) Breakdown of the sync time of WebRsync. (c) Average client-side CPU utilization.

Figure 3: Breakdown of the sync time of (a) `rsync` and (b) WebRsync for append operations, as well as the corresponding average client-side CPU utilizations. The situations for insert and cut operations are similar.

3.3 Measuring Stagnation with StagMeter

As discussed in §3.2, WebRsync not only leads to more sync time, but also costs more computation resources at the client side. The heavy CPU consumption causes web browsers to frequently stagnate and even hang. To quantitatively understand the stagnation of web browser perceived by users, we develop the StagMeter tool to measure the stagnation time by automatically integrating a piece of JavaScript code into the web browser [7]. StagMeter periodically [8] 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, there is no stagnation; otherwise, there is a stagnation and then the printing of the current timestamp will be postponed to $t' > t$. Therefore, the corresponding stagnation time is calculated as $t' - t$.

Using StagMeter, we measure and visualize the stagnations of WebRsync (on handling the three types of file edits) in Figure 4. Note that StagMeter only attempts to print 10 timestamps for the first second. Therefore, spaces between consecutive timestamps represent stagnation, and larger spaces imply longer stagnations. As indicated in all the three subfigures, stagnations are directly associated with high CPU utilizations.

[7] We can also directly use the native profiling tool of the Chrome browser to visualize the stagnation, whose results we found more complicated to interpret than those of StagMeter.

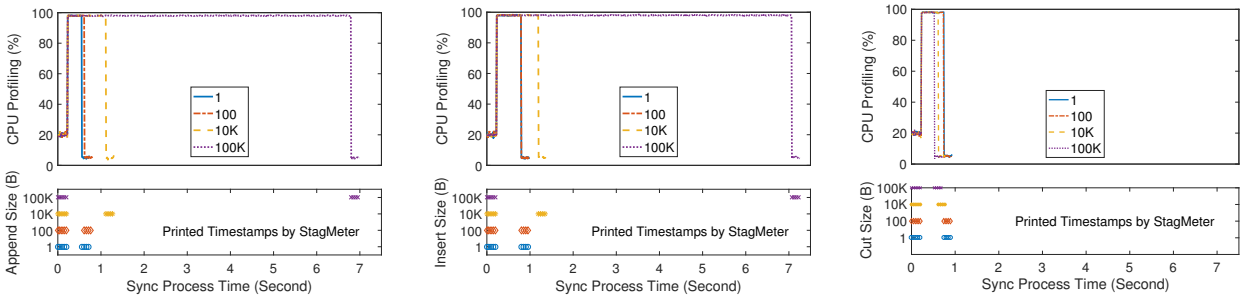
[8] By default we set the period as 100 ms, so as to simulate the minimum intervals of common web users' operations.

4 Native Extension, Parallelism, and Client-side Optimization of WebRsync

This section investigates three approaches to partially addressing the drawback of WebRsync. For each approach, we first describe its working principle, and then evaluate its performance using different types of file edits.

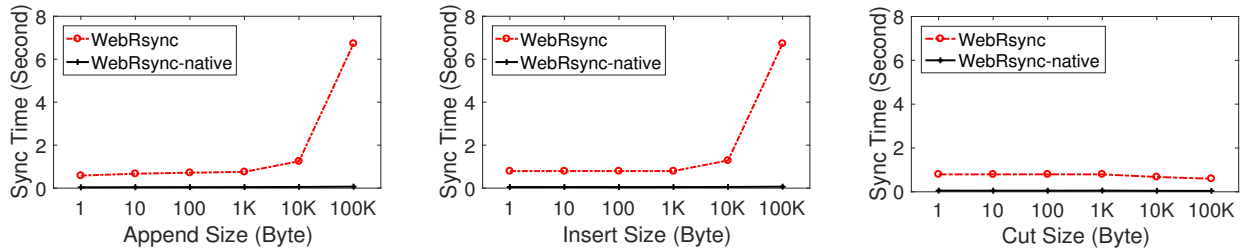
WebRsync-native. Given that the sync speed of WebRsync is much lower than that of the PC client-based delta sync solution (`rsync`), our first approach to optimizing WebRsync is to leverage the *native client* [13] for web browsers. Native client is a sandbox for efficiently and securely executing compiled C/C++ code in a web browser, and has been supported by all mainstream web browsers. In our implementation, we use the Chrome native client to accelerate the execution of WebRsync on the Chrome browser. We first use HTML5 and JavaScript to compose the webpage interface, through which a user can select a local file to synchronize (to the cloud). Then, the path of the selected local file is sent to our developed native client (written in C++). Afterwards, the native client reads the file content and synchronizes it to the cloud in a similar way as `rsync`. When the sync process finishes, the native client returns an acknowledgement message to the webpage interface, which then shows the user the success of the delta sync operation.

Figure 5 depicts the performance of WebRsync-native, in comparison to the performance of original WebRsync. Obviously, WebRsync-native significantly reduces the



(a) Append. (b) Insert. (c) Cut.

Figure 4: Stagnation captured by StagMeter for different edit operations and the associated CPU utilizations. The stagnation time is illustrated by the discontinuation of the timestamp on the sync process time.



(a) Append. (b) Insert. (c) Cut.

Figure 5: Average sync time using WebRsync-native for various sizes of file edits under a simple workload.

sync time of WebRsync, in fact close to the sync time of `rsync`. Accordingly, the CPU utilization is decreased and the stagnation of the Chrome browser is fully avoided. Nevertheless, using native client requires the user to download and install extra plug-in components for the web browser, which essentially impairs the usability and pervasiveness of WebRsync-native.

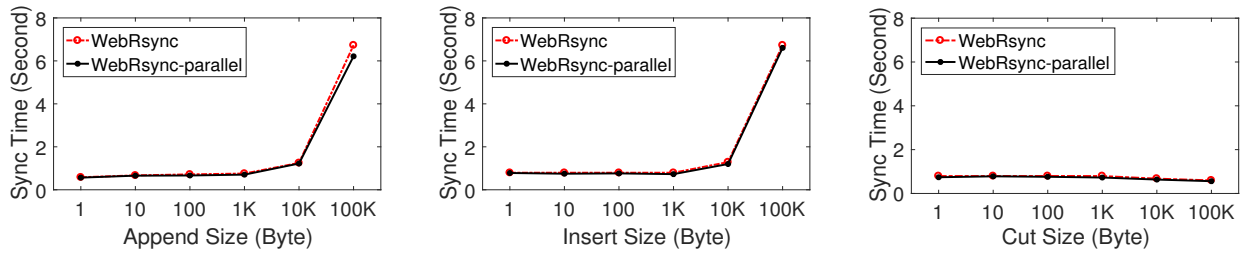
WebRsync-parallel. Our second approach is to use HTML5 *web workers* [10] for parallelism or threading. Generally speaking, when executing JavaScript code in a webpage, the webpage becomes unresponsive until the execution is finished—this is why WebRsync would lead to frequent stagnation and even hanging of the web browser. To address this problem, a web worker is a JavaScript program that runs in the background, independently of other JavaScript programs in the same webpage. When we apply it to WebRsync, the original single JavaScript program is divided to multiple JavaScript programs that work in parallel. Although this approach can hardly reduce the total sync time (as indicated in Figure 6) or the CPU utilizations (as shown in Figure 7, the upper part), it can fully avoid stagnation for the Chrome browser (as shown in Figure 7, the lower part).

WebRsync+. Later in §5.2 we describe in detail how we exploit users' file-edit locality and lightweight hash algorithms to reduce server-side computation overhead. As a matter of fact, the two-fold optimizations can also be applied to the client side. Thereby, we implement the two optimization mechanisms at the client side of WebRsync by translating them from C++ to JavaScript, and the resulting solution is referred to as WebRsync+. As illustrated in Figure 8, WebRsync+ stays between WebRsync and WebR2sync+ in terms of sync time, which is basically within our expectation. Further, we decompose the sync time of WebRsync+ into three stages: at the client side, across the network, and at the server side, as depicted in Figure 9. Comparing Figure 9 with Figure 3b (breakdown of the sync time of WebRsync into three stages), we find that the client-side time cost of WebRsync+ is remarkably reduced thanks to the two optimization mechanisms. However, WebRsync+ cannot fully avoid stagnation for web browsers; instead, it can only alleviate the stagnation compared to WebRsync.

Summary. With the above three-fold efforts, we conclude that the drawback of WebRsync cannot be fundamentally addressed via solely client-side optimizations. That is to say, we need more comprehensive solutions where the server side is also involved.

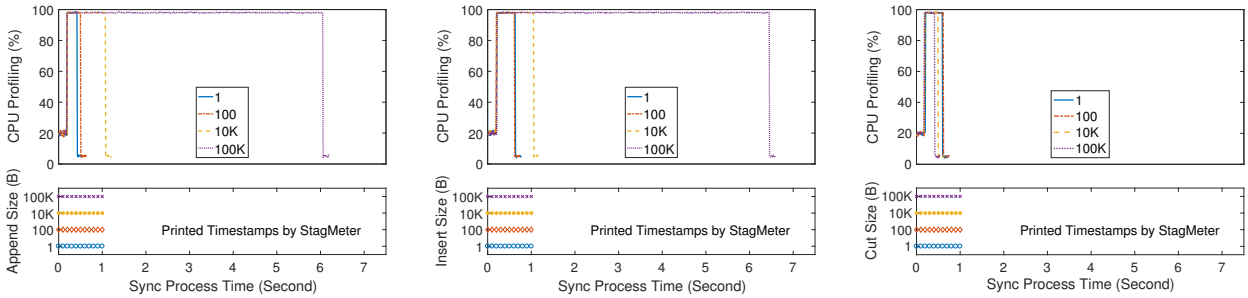
5 WebR2sync+: Web-based Delta Sync Made Practical

This section presents WebR2sync+, the practical solution for web-based delta sync. The practicality of WebR2sync+ is attributed to multi-fold endeavors at both



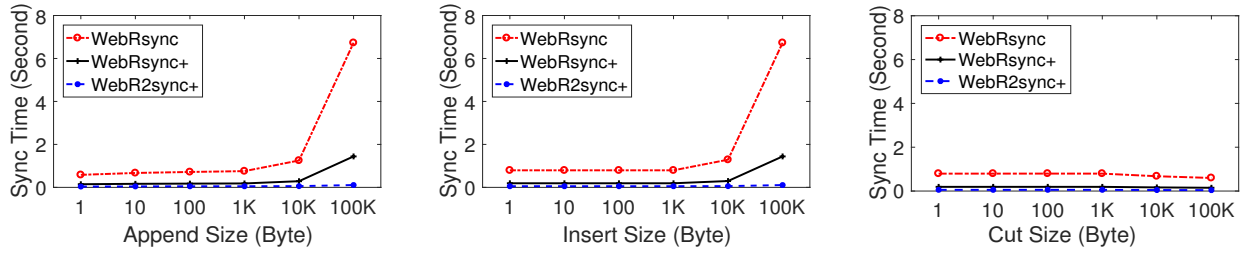
(a) Append. (b) Insert. (c) Cut.

Figure 6: Average sync time using WebRsync-parallel for various sizes of file edits under a simple workload.



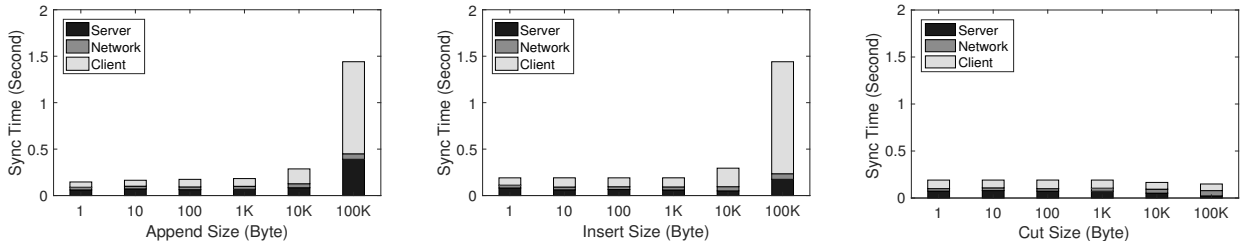
(a) Append. (b) Insert. (c) Cut.

Figure 7: Although WebRsync-parallel is unable to reduce the CPU utilizations (relative to WebRsync), it can fully avoid stagnation for the Chrome web browser by utilizing HTML5 web workers.



(a) Append. (b) Insert. (c) Cut.

Figure 8: Average sync time using WebRsync+ for various sizes of file edits under a simple workload.



(a) Append. (b) Insert. (c) Cut.

Figure 9: Breakdown of the sync time of WebRsync+ (shown in Figure 8) for different types of edit operations.

client and server sides. We first present the basic solution, WebR2sync, which improves WebRsync (§5.1), and then describe the server-side optimizations for mitigating the computation overhead (§5.2). The final solution that combines both WebR2sync with the server-side optimizations is referred to as WebR2sync+ in §5.3.

5.1 WebR2sync

As depicted in Figure 10, to address the overload issue, WebR2sync reverses the process of WebRsync (c.f., Figure 1) by moving the computation intensive search and comparison operations to the server side; meanwhile, it shifts the lightweight segmentation and finger-

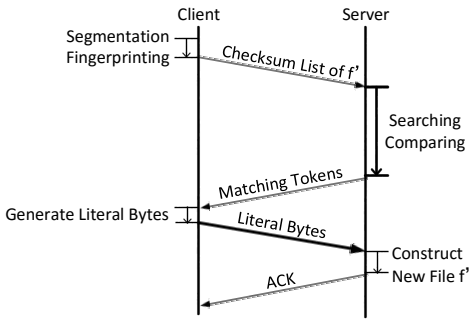


Figure 10: Design flow chart of WebR2sync.

printing operations to the client side. Compared with the workflow of conventional web-based delta sync, in WebRsync, 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 this allows us to implement the search and comparison operations in C rather than in JavaScript at the server side. Therefore, WebR2sync can not only avoid stagnation for the web client, but also effectively shorten the duration of the whole delta sync process.

5.2 Server-side Optimizations

While WebR2sync significantly cuts the computation burden on the web client, it brings considerable computation overhead to the server side. To this end, we make two-fold additional efforts to optimize the server-side computation overhead.

Exploiting the 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 shown in Figure 11 (which follows the 3-level chunk searching scheme of `rsync` [15]). Specifically, in the checksum list of f' there is a 32-bit weak rolling checksum (calculated by the Adler32 algorithm [26]) and 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* whose every entry is a 16-bit hash code of the 32-bit rolling checksum [15]. 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 and MD5 checksums. Thereby, to find each matched chunk 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

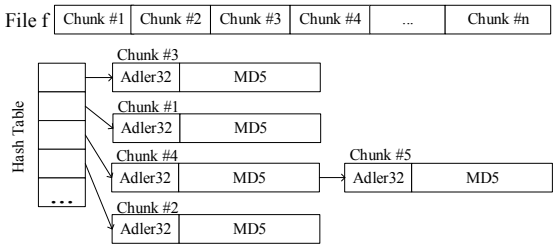
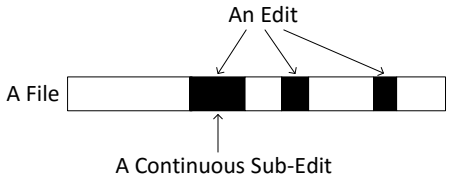


Figure 11: The three-level chunk searching scheme used by `rsync` and WebR2sync.



(a) An edit consists of several continuous sub-edits.

(b) The worst case of a file edit in terms of locality.

Figure 12: An example of continuous sub-edits due to the locality of file edits: (a) the relationship between a file edit and its constituent continuous sub-edits; (b) the worst-case scenario in terms of locality.

follow a local pattern rather than a general (random) pattern, which has been exploited to accelerate file compression and deduplication [41,47–49]. To exemplify this observation in a quantitative manner, we analyze two real-world fine-grained file editing traces with respect to Microsoft Word and Tencent WeChat collected by Zhang *et al.* [51]. The traces are fine-grained since they leveraged a loopback user-space file system (Dokan [7] for Windows) to record not only the detailed information (*e.g.,* edit type, edit offset, and edit length) of users' file operations but also the content of the updated data. In each trace, a user made several *continuous sub-edits* to a file and then did a save operation, and this behavior repeated for many times. Here a continuous sub-edit means that the sub-edit operation happens to continuous bytes in the file, as demonstrated in Figure 12. Our analysis results, in Figure 13, show that in nearly a half (46%) of cases a user saved 1–5 continuous sub-edits, thus indicating fine locality. Besides, in over one third (35%) of cases a user saved 6–10 continuous sub-edits, which still implies sound locality. On the other hand, in only a minority (5%) of cases a user saved more than 16 continuous sub-edits, which means undesirable locality.

The locality of real-world file edits offers us an opportunity to bypass a considerable portion of (unnecessary) chunk search operations. In essence, given that edits to a file are typically local, when we find that the i-th chunk

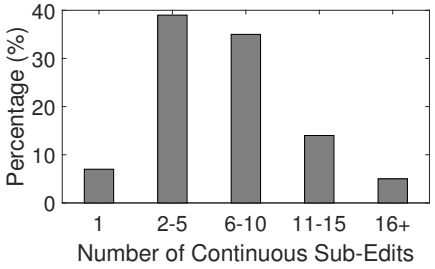


Figure 13: A real-world example of file-edit locality. The number of continuous sub-edits is highly clustered.

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 simply move forward to the next chunk; otherwise, we return to the regular 3-level chunk searching scheme.

Replacing MD5 with SipHash in chunk comparison. By exploiting the locality of users' file edits as above, we manage to bypass most chunk search operations. After that, 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 codes [43], which makes it computationally expensive. In our scenario, it is not necessary to use such an 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 [22] in order to reduce the computational overhead.

Quite a few pseudorandom hash functions can satisfy our goal, such as Spooky [17], FNV [9], CityHash [5], SipHash [20], and Murmur3 [12]. Among them, some are very lightweight but vulnerable to collisions. For example, the computation overhead of MD5 is around 5 to 6 cycles per byte [8] while the computation overhead of CityHash is merely 0.23 cycle per byte [19], but the collision probability of CityHash is quite high. On the other hand, some pseudorandom hash functions have extremely low collision probability but are a bit slow. As listed in Table 2, SipHash seems to be a sweet spot — its computation overhead is about 1.13 cycles per byte and its collision probability is acceptably low. By replacing MD5 with SipHash in our web-based delta sync solution, we manage to reduce the computation complexity of chunk comparison by nearly 5 times.

Hash Function	Collision Probability	Cycles Per Byte
MD5	Low ($< 10^{-6}$)	5.58
Murmur3	High ($\approx 1.05 \times 10^{-4}$)	0.33
CityHash	High ($\approx 1.03 \times 10^{-4}$)	0.23
FNV	High ($\approx 1.09 \times 10^{-4}$)	1.75
Spooky	High ($\approx 9.92 \times 10^{-5}$)	0.14
SipHash	Low ($< 10^{-6}$)	1.13

Table 2: A comparison of candidate pseudorandom hash functions in terms of collision probability (on 64-bit hash values) and computation overhead (cycles per byte).

Although the collision probability of SipHash is acceptably low, it is slightly higher than that of MD5. Thus, as a fail-safe mechanism, we make a lightweight full-content hash checking (using the Spooky algorithm) in the end of a file synchronization, so as to deal with possible collisions in SipHash chunk fingerprinting. We select the Spooky algorithm because it works the fastest among all the candidate pseudorandom hash algorithms (as listed in Table 2). If the full-content hash checking fails for the synchronization of a file (with an extremely low probability), we will roll back and re-sync the file with the original MD5 chunk fingerprinting.

5.3 WebR2sync+: The Final Product

The integration of WebR2sync and the server-side optimization produces WebR2sync+. The client side of WebR2sync+ is implemented based on the HTML5 File APIs, the WebSocket protocol, an open-source implementation of SipHash-2-4 [1], and an open-source implementation of SpookyHash [11]. In total, it is written in 1700 lines of JavaScript code. The server side of WebR2sync+ is developed based on the node.js framework and a series of C processing modules. The former (written in 500 lines of node.js code) handles the user requests, and the latter (written in 1000 lines of C code) embodies the reverse delta sync process together with the server-side optimizations.

6 Evaluation

This section evaluates the performance of WebR2sync+, in comparison to WebRsync, WebR2sync and (PC client-based) `rsync` under a variety of workloads.

6.1 Experiment Setup

To evaluate different sync approaches, we set up a Dropbox-like system architecture by running the web service on a standard VM server instance (with a quad-core Intel Xeon CPU @2.5GHz and 16-GB memory) rent from Aliyun ECS, and all file content is hosted on object storage rent from Aliyun OSS. The ECS VM server and OSS storage are located at the same data center so there is no bottleneck between them. The client side of WebR2sync+ was executed in the Google Chrome

Figure 14: Experiment setup in China.

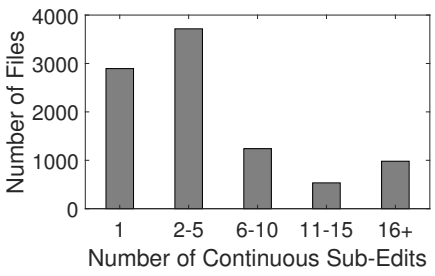


Figure 15: File-edit locality in the source files of two successive Linux kernel releases (versions 4.5 and 4.6).

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. 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 14. The network RTT is ∼30 ms and the network bandwidth is ∼100 Mbps. Therefore, the network bottleneck is kept minimal in our experiments so that the major system bottleneck lies at the server and/or client sides. If the network condition becomes much worse, the major system bottleneck might shift to the network connection.

6.2 Workloads

To evaluate the performance of WebR2sync+ under various practical usage scenarios, as compared to WebRsync, WebR2sync, and `rsync`, we generate *simple* (*i.e.,* one-shot), *regular* (*i.e.,* periodical), and *intensive* workloads. To generate simple workloads, we make random append, insert, and cut operations of different edit sizes against real-world files collected from real-world cloud storage services. The collected dataset is described in §3.2. One file is edited for only once (the so-called "one-shot"), and it is then synchronized from the client side to the server side. For an insert or cut operation, when its edit size $\geq$ 1 KB, it is first dispersed into 1–20 continuous sub-edits and then synchronized to the server.

Regular and intensive workloads are mainly employed to evaluate the service throughput of each solution. To generate regular workloads, we still make a certain type of edit to a typical file but the edit operation is executed every 10 seconds. To generate a practical intensive workload, we use a benchmark of over 8755 pairs of source files taken from two successive releases (versions 4.5 and 4.6) of the Linux kernel source trees. The average size of the source files is 23 KB and the file-edit locality is generally stronger than that in Figure 13 (as shown in Figure 15). Specifically, we first upload all the files of the old version to the server side in an FTP-like manner. Then, we synchronize all the files of the new version one by one to the server side using the target approaches (including `rsync`, WebRsync, WebR2sync, WebR2sync with SipHash, and WebR2sync+). There is no time interval between two sequential file synchronizations.

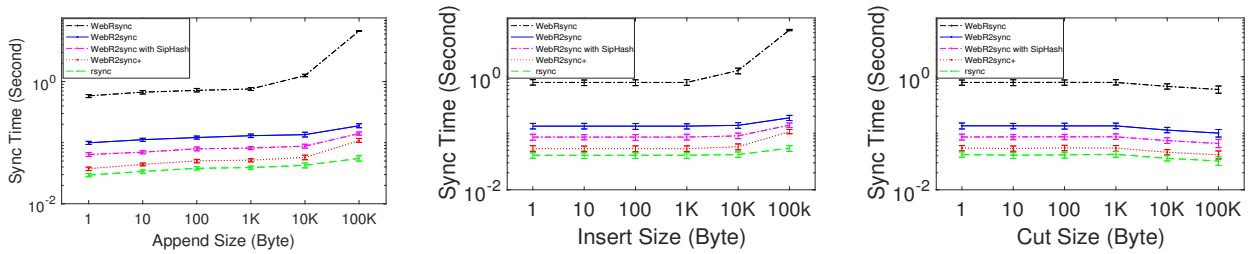
6.3 Results

This part presents our experiment results in four aspects: 1) *sync efficiency* which measures how quick a file operation is synchronized to the cloud; 2) *computation overhead* which explains the difference in sync efficiency of the studied solutions; 3) *sync traffic* which quantifies how much network traffic is saved by each solution; and 4) *service throughput* which shows the scalability of each solution using standard VM server instances.

Sync efficiency. We measure the efficiency of WebR2sync+ in terms of the time for completing the sync. Figure 16 shows the time for syncing against different types of file operations. We can see that the sync time of WebR2sync+ is substantially shorter than that of WebR2sync (by 2 to 3 times) and WebRsync (by 15 to 20 times) for every different type of operations. Note that Figure 16 is plotted with a log scale. In other words, WebR2sync+ outpaces WebRsync by around an order of magnitude, approaching the speed of PC client-based `rsync`. Furthermore, we observe that the sync time of WebR2sync with SipHash always lies between those of WebR2sync and WebR2sync+. This confirms that neither of our server-side optimizations (SipHash and locality exploiting, refer to §5.2) is indispensable.

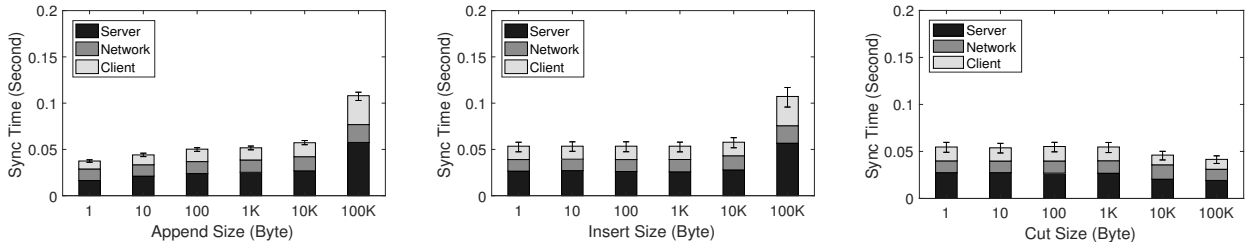
Similar as Figure 3b, we further break down the sync time of WebR2sync+ into three stages as shown in Figure 17. Comparing Figure 17 and Figure 3b, we notice that the majority of sync time is attributed to the client side for WebRsync, while it is attributed to the server side for WebR2sync+. This indicates that the computation overhead of the web browsers in WebRsync is substantially reduced in WebR2sync+, which also saves web browsers from stagnation and hanging.

Computation overhead. Moreover, we record the client-side and server-side CPU utilizations in Figure 18 and Figure 19, respectively. On the client side, WebRsync consumes the most CPU resources while



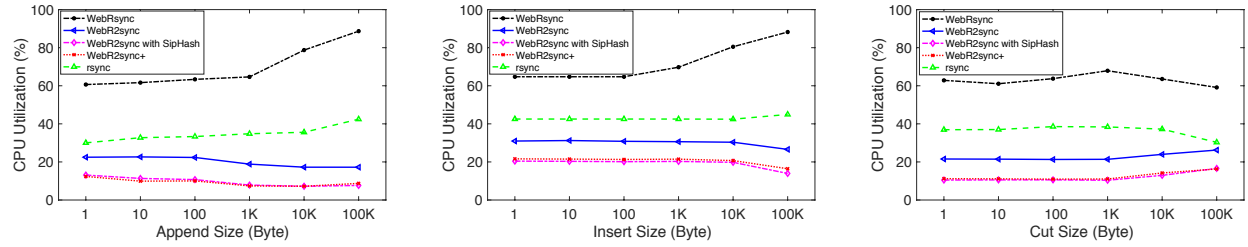
(a) Append. (b) Insert. (c) Cut.

Figure 16: Average sync time of different delta sync approaches for various sizes of file edits under a simple workload.



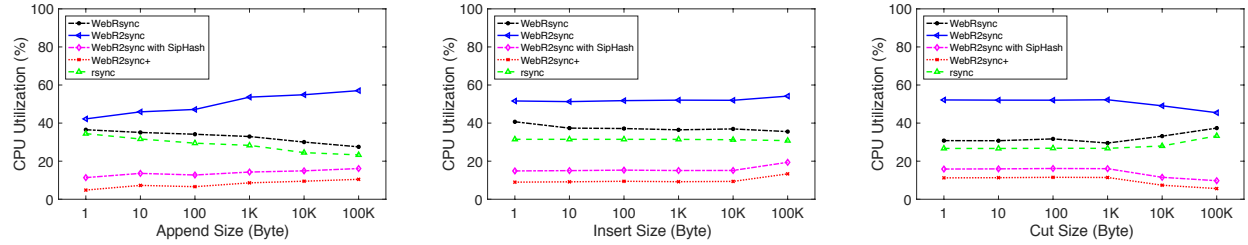
(a) Append. (b) Insert. (c) Cut.

Figure 17: Breakdown of the sync time of WebR2sync+ (shown in Figure 16) for different types of edit operations.



(a) Append. (b) Insert. (c) Cut.

Figure 18: Average **client**-side CPU utilization of different delta sync approaches under a simple workload.



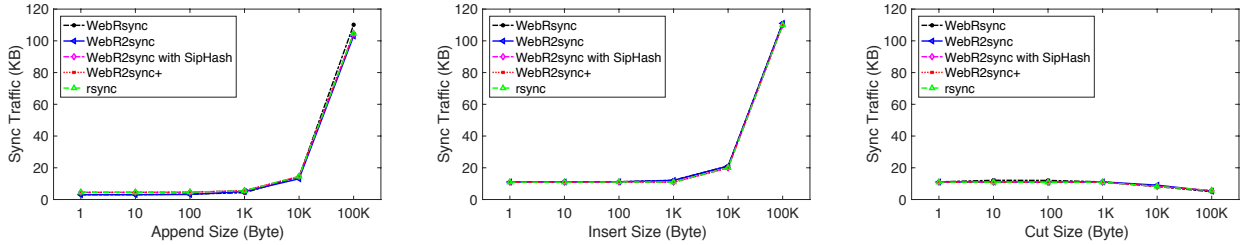
(a) Append. (b) Insert. (c) Cut.

Figure 19: Average **server**-side CPU utilization of different delta sync approaches under a simple workload.

WebR2sync+ consumes the least. PC client-based `rsync` consumes nearly a half CPU resources as compared to WebRsync, and the CPU utilization of WebR2sync lies between `rsync` and WebR2sync+. Owing to the moderate ($< 30\%$) CPU utilizations, both the clients of WebR2sync and WebR2sync+ do not exhibit stagnation.

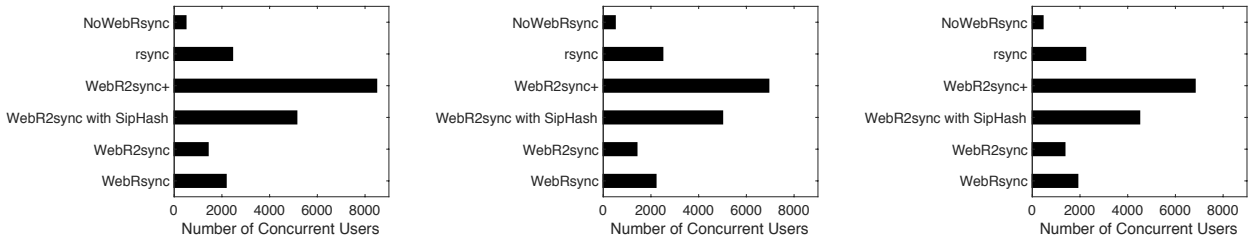
On the server side, WebR2sync consumes the most CPU resources because the most computation-intensive chunk search and comparison operations are shifted from the client to the server. On the contrary, WebR2sync+ consumes the least CPU resources, which validates the efficacy of our two-fold server-side optimizations.

Sync traffic. Figure 20 illustrates the sync traffic consumed by the different approaches. We can see that for any type of edits, the sync traffic (between 1 KB and 120 KB) is significantly less than the average file size ($\sim$1 MB), confirming the power of delta sync in improving network-level efficiency of cloud storage services.



(a) Append. (b) Insert. (c) Cut.

Figure 20: Sync traffic of different sync approaches for various sizes of file edits under a simple workload.



(a) Append. (b) Insert. (c) Cut.

Figure 21: Number of concurrent clients supported by a single VM server instance (as a measure of service throughput) under regular workloads (periodically syncing various sizes of file edits).

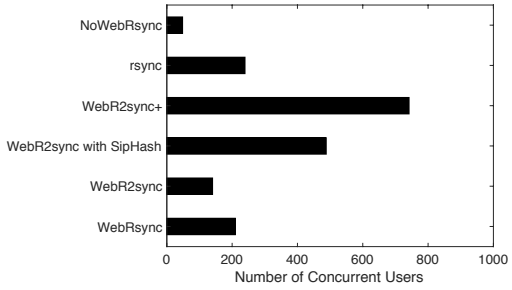


Figure 22: Number of concurrent users supported by a single VM server instance under intensive workloads (syncing two versions of Linux source trees).

For the same edit size the sync traffic of an append operation is usually less than that of an insert operation, because the former would bring more matching tokens while fewer literal bytes (refer to Figure 1). Besides, when the edit size is relatively large (10 KB or 100 KB), a cut operation consumes much less sync traffic than an append/insert operation, because a cut operation brings only matching tokens but not literal bytes.

Service throughput. Finally, we measure the service throughput of WebR2sync+ in terms of the number of concurrent clients it can support. In general, as the number of concurrent clients increases, the main burden imposed on the server comes from the high CPU utilizations in all cores. When the CPU utilizations on all cores approach 100%, we record the number of concurrent clients at that time as the service throughput. As shown in Figure 21, WebR2sync+ can simultaneously support 6800–8500 web clients' delta sync using a standard VM server instance under regular workloads. This throughput is as 3–4 times as that of WebR2sync/`rsync` and as ∼15 times as that of NoWebRsync. NoWebRsync means that no web-based delta sync is used for synchronizing file edits, *i.e.,* directly uploading the entire content of the edited file to the cloud. Also, we measure the service throughput of each solution under intensive workloads (which are mixed by the three types of edits, refer to §6.2). The results in Figure 22 indicate that even under the intensive workloads, WebR2sync+ can simultaneously support 740 web clients' delta sync using a single VM server instance.

7 Related Work

Delta sync, also known as delta encoding or delta compression, is a way of storing or transmitting data in the form of differences (deltas) between different versions of a file, rather than the complete content of the file [6]. It is particularly useful for network applications where file modifications or incremental data updates frequently happen, *e.g.,* storing multiple versions of a file, distributing consecutive user edits to a file, and transmitting video sequences [33]. In the past 4 decades, a variety of delta sync algorithms or solutions have been put forward, such as UNIX `diff` [32], `Vcdiff` [34], WebExpress [31], Optimistic Deltas [21], `rsync` [15], and content defined chunking (CDC) [35].

Due to its efficiency and flexibility, `rsync` has become the de facto delta sync protocol widely used in practice. It was originally proposed by Tridgell and Mackerras in 1996, as an algorithm for efficient remote update of data over a high-latency, low-bandwidth network link [45]. Then in 1999, Tridgell thoroughly discussed its design, implementation, and performance in [44]. Being a standard Linux utility included in all popular Linux distributions, `rsync` has also been ported to Windows, FreeBSD, NetBSD, OpenBSD, and MacOS [15].

According to a real-world usage dataset [37], the majority (84%) of files are modified by the users for at least once, thus confirming the importance of delta sync on network-level efficiency of cloud storage services. Among all mainstream cloud storage services, Dropbox was the first to adopt delta sync (more specifically, `rsync`) in around 2009 in its PC client-based file sync process [39]. Then, SugarSync, iCloud Drive, and Seafile followed the design choice of Dropbox by utilizing delta sync (`rsync` or CDC) to reduce their PC clients' and cloud servers' sync traffic. After that, two academic cloud storage systems, namely QuickSync [25] and DeltaCFS [51], further implemented delta sync (`rsync` and CDC, respectively) for mobile apps.

Drago *et al.* studied the system architecture of Dropbox and conducted large-scale measurements based on ISP-level traces of Dropbox network traffic [28]. They observed that the Dropbox traffic was as much as one third of the YouTube traffic, which strengthens the necessity of Dropbox's adopting delta sync. Li *et al.* investigated in detail the delta sync process of Dropbox through various types of controlled benchmark experiments, and found it suffers from both traffic and computation overuse problems in the presence of frequent, short data updates [39]. To this end, they designed an efficient batched synchronization algorithm called UDS (update-batched delayed sync) to reduce the traffic usage, and further extended UDS with a backwards compatible Linux kernel modification to reduce the CPU usage (recall that delta sync is computation intensive).

Despite the wide adoption of delta sync (particularly `rsync`) in cloud storage services, practical delta sync techniques are currently only available for PC clients and mobile apps rather than web browsers. To this end, we introduced the general idea of web-based delta sync with basic motivation, preliminary design, and early-stage performance evaluation using limited workloads and metrics [50]. In this paper, our work is conducted based on [50] while goes beyond it in terms of techniques, evaluations, and presentations.

8 Conclusion and Future Work

This paper presents a series of efforts towards a practical solution of web-based delta sync for cloud storage services. We first leverage the state-of-the-art techniques (including `rsync`, JavaScript, HTML5 File APIs, and WebSocket) to develop an intuitive web-based delta sync solution named WebRsync. Despite not being practically acceptable in terms of performance, WebRsync effectively helps us understand the obstacles to support web-based delta sync. Particularly, we observe that the inefficiency of JavaScript execution significantly stagnates the sync process of WebRsync. Thereby, we propose and implement WebR2sync+, a practical web-based delta sync solution by moving expensive chunk search and comparison operations from the client side to the server side. It combines with optimizations at the server side that exploit the locality of users' file edits and uses lightweight pseudorandom hash functions to replace the traditional expensive cryptographic hash function. WebR2sync+ outpaces WebRsync by an order of magnitude, and is able to simultaneously support around 6800–8500 web clients' delta sync using a standard VM server instance under a Dropbox-like system architecture.

We are investigating the following aspects as the future work. First, we are looking for a seamless way to integrate the server-side design of WebR2sync+ with the back-end of commercial cloud storage vendors (like Dropbox and iCloud Drive). Specifically, WebR2sync+ needs to cooperate with data deduplication, compression, bundling, *etc.* [23, 27]. Moreover, we would like to explore the benefits of using more fine-grained and complex delta sync protocols, such as CDC and its variants [30, 42, 49]. In addition, we envision to expand the usage of WebR2sync+ for a broader range of web service scenarios, not limited to web browsers and cloud storage services. For example, when a user wants to use a web-based app to upload a file f' to a common web server (such as Apache, Nginx, or IIS) which has already stored an old version of the file (f), web-based delta sync has the great potential to reduce network traffic and operation time. In this case, the major challenge lies in the requirement of modifying the web server implementation; minimizing the modification efforts is under investigation.

Acknowledgments

We thank the anonymous reviewers for their positive and constructive comments. Besides, we appreciate the valuable guidance and detailed suggestions from our shepherd, Vasily Tarasov, during the revision of the paper. In addition, we thank Yonghe Wang for helping with some measurements during the preparation of the paper. This work is supported by the High-Tech R&D Program of China ("863–China Cloud" Major Program) under grant 2015AA01A201, the NSFC under grants 61471217, 61432002, 61632020 and 61472337. Ennan Zhai is partly supported by the NSF under grants CCF-1302327 and CCF-1715387.

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] asm.js, a strict subset of JavaScript that can be used as a low-level, efficient target language for compilers. http://asmjs.org.

[5] CityHash. https://opensource.googleblog.com/2011/04/introducing-cityhash.html.

[6] Delta encoding, the Wikipedia page. https://en.wikipedia.org/wiki/Delta_encoding.

[7] Dokan: An user mode file system for Windows. https://dokan-dev.github.io.

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

[9] FNV Hash. http://www.isthe.com/chongo/tech/comp/fnv/.

[10] HTML5 Web Workers. https://www.w3schools.com/html/html5_webworkers.asp.

[11] Javascript version of SpookyHash. https://github.com/jamesruan/spookyhash-js.

[12] Murmur3 Hash Function. https://github.com/aappleby/smhasher.

[13] Native Client for Google Chrome. https://developer.chrome.com/native-client.

[14] Reading Files in JavaScript using the HTML5 File APIs. https://www.html5rocks.com/en/tutorials/file/dndfiles/.

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

[16] Seafile: Enterprise file sync and share platform with high reliability and performance. https://www.seafile.com/en/home.

[17] Spookyhash: A 128-Bit Noncryptographic Hash. http://burtleburtle.net/bob/hash/spooky.html.

[18] Using files from web applications. https://developer.mozilla.org/en-US/docs/Using_files_from_web_applications.

[19] ALAKUIJALA, J., COX, B., AND WASSENBERG, J. Fast Keyed Hash/Pseudo-random Function Using SIMD Multiply and Permute. *arXiv preprint arXiv:1612.06257* (2016).

[20] AUMASSON, J.-P., AND BERNSTEIN, D. SipHash: a Fast Short-input PRF. In *Proc. of the International Conference on Cryptology in India* (2012), Springer, pp. 489–508.

[21] BANGA, G., DOUGLIS, F., RABINOVICH, M., ET AL. Optimistic Deltas for WWW Latency Reduction. In *Proc. of ATC* (1997), USENIX, pp. 289–303.

[22] BELLARE, M., CANETTI, R., AND KRAWCZYK, H. Keying Hash Functions for Message Authentication. In *Proc. of Crypto* (1996), Springer, pp. 1–15.

[23] BOCCHI, E., DRAGO, I., AND MELLIA, M. Personal Cloud Storage Benchmarks and Comparison. *IEEE Transactions on Cloud Computing (TCC) 5*, 4 (2015), 751–764.

[24] BOCCHI, E., DRAGO, I., AND MELLIA, M. Personal Cloud Storage: Usage, Performance and Impact of Terminals. In *Proc. of CloudNet* (2015), IEEE, pp. 106–111.

[25] CUI, Y., LAI, Z., WANG, X., DAI, N., AND MIAO, C. QuickSync: Improving Synchronization Efficiency for Mobile Cloud Storage Services. In *Proc. of MobiCom* (2015), ACM, pp. 592–603.

[26] DEUTSCH, P., AND GAILLY, J.-L. Zlib Compressed Data Format Specification Version 3.3. Tech. rep., RFC Network Working Group, 1996.

[27] DRAGO, I., BOCCHI, E., MELLIA, M., SLATMAN, H., AND PRAS, A. Benchmarking Personal Cloud Storage. In *Proc. of IMC* (2013), ACM, pp. 205–212.

[28] DRAGO, I., MELLIA, M., MUNAFÒ, M., SPEROTTO, A., SADRE, R., AND PRAS, A. Inside Dropbox: Understanding Personal Cloud Storage Services. In *Proc. of IMC* (2012), ACM, pp. 481–494.

[29] E, J., CUI, Y., WANG, P., LI, Z., AND ZHANG, C. CoCloud: Enabling Efficient Cross-Cloud File Collaboration based on Inefficient Web APIs. *IEEE Transactions on Parallel and Distributed Systems (TPDS) 29*, 1 (2018), 56–69.

[30] EL-SHIMI, A., KALACH, R., KUMAR, A., OTTEAN, A., LI, J., AND SENGUPTA, S. Primary Data Deduplication – Large Scale Study and System Design. In *Proc. of ATC* (2012), USENIX, pp. 285–296.

[31] HOUSEL, B., AND LINDQUIST, D. WebExpress: A System for Optimizing Web Browsing in a Wireless Environment. In *Proc. of MobiCom* (1996), ACM, pp. 108–116.

[32] HUNT, J., AND MACILROY, M. *An Algorithm for Differential File Comparison*. Bell Laboratories New Jersey, 1976.

[33] HUNT, J., VO, K., AND TICHY, W. An Empirical Study of Delta Algorithms. *Software Configuration Management* (1996), 49–66.

[34] KORN, D., AND VO, K.-P. Engineering a Differencing and Compression Data Format. In *Proc. of ATC* (2002), USENIX, pp. 219–228.

[35] KRUUS, E., UNGUREANU, C., AND DUBNICKI, C. Bimodal Content Defined Chunking for Backup Streams. In *Proc. of FAST* (2010), USENIX, pp. 239–252.

[36] LI, Z., DAI, Y., CHEN, G., AND LIU, Y. *Content Distribution for Mobile Internet: A Cloud-based Approach*. Springer, 2016.

[37] LI, Z., JIN, C., XU, T., WILSON, C., LIU, Y., CHENG, L., LIU, Y., DAI, Y., AND ZHANG, Z.-L. Towards Network-level Efficiency for Cloud Storage Services. In *Proc. of IMC* (2014), ACM, pp. 115–128.

[38] LI, Z., WANG, X., HUANG, N., KAAFAR, M., LI, Z., ZHOU, J., XIE, G., AND STEENKISTE, P. An Empirical Analysis of a Large-scale Mobile Cloud Storage Service. In *Proc. of IMC* (2016), ACM, pp. 287–301.

[39] LI, Z., WILSON, C., JIANG, Z., LIU, Y., ZHAO, B., JIN, C., ZHANG, Z.-L., AND DAI, Y. Efficient Batched Synchronization in Dropbox-like Cloud Storage Services. In *Proc. of ACM/IFIP/USENIX Middleware* (2013), Springer, pp. 307–327.

[40] LI, Z., ZHANG, Z.-L., AND DAI, Y. Coarse-grained Cloud Synchronization Mechanism Design May Lead to Severe Traffic Overuse. *Tsinghua Science and Technology 18*, 3 (2013), 286–297.

[41] LILLIBRIDGE, M., ESHGHI, K., BHAGWAT, D., DEOLALIKAR, V., TREZIS, G., AND CAMBLE, P. Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality. In *Proc. of FAST* (2009), USENIX, pp. 111–123.

[42] MUTHITACHAROEN, A., CHEN, B., AND MAZIERES, D. A Low-bandwidth Network File System. *ACM SIGOPS Operating Systems Review 35* (2001), 174–187.

[43] RIVEST, R., ET AL. RFC 1321: The MD5 Message-digest Algorithm. *Internet activities board 143* (1992).

[44] TRIDGELL, A. *Efficient Algorithms for Sorting and Synchronization*. 1999.

[45] TRIDGELL, A., MACKERRAS, P., ET AL. The `rsync` Algorithm. *The Australian National University* (1996).

[46] XIA, W., JIANG, H., FENG, D., DOUGLIS, F., SHILANE, P., HUA, Y., FU, M., ZHANG, Y., AND ZHOU, Y. A Comprehensive Study of the Past, Present, and Future of Data Deduplication. *Proceedings of the IEEE 104*, 9 (2016), 1681–1710.

[47] XIA, W., JIANG, H., FENG, D., AND HUA, Y. SiLo: A Similarity-Locality based Near-Exact Deduplication Scheme with Low RAM Overhead and High Throughput. In *Proc. of ATC* (2011), USENIX, pp. 26–30.

[48] XIA, W., JIANG, H., FENG, D., AND HUA, Y. Similarity and Locality Based Indexing for High Performance Data Deduplication. *IEEE Transactions on Computers (TC) 64*, 4 (2015), 1162–1176.

[49] XIA, W., ZHOU, Y., JIANG, H., FENG, D., HUA, Y., HU, Y., LIU, Q., AND ZHANG, Y. FastCDC: a Fast and Efficient Content-defined Chunking Approach for Data Deduplication. In *Proc. of ATC* (2016), USENIX, pp. 101–114.

[50] XIAO, H., LI, Z., ZHAI, E., AND XU, T. Practical Web-based Delta Synchronization for Cloud Storage Services. In *Proc. of HotStorage* (2017), USENIX.

[51] ZHANG, Q., LI, Z., YANG, Z., LI, S., GUO, Y., AND DAI, Y. DeltaCFS: Boosting Delta Sync for Cloud Storage Services by Learning from NFS. In *Proc. of ICDCS* (2017), IEEE, pp. 264–275.