Dsync: a Lightweight Delta Synchronization Approach for Cloud Storage Services

Yuan He[1], Lingfeng Xiang[2], Wen Xia[1,4,5], Hong Jiang[2], Zhenhua Li[3], Xuan Wang[1], and Xiangyu Zou[1]

[1] School of Computer Science and Technology, Harbin Institute of Technology, Shenzhen, China
[2] Department of Computer Science & Engineering, University of Texas at Arlington, Arlington, USA
[3] School of Software, Tsinghua University, Beijing, China
[4] Cyberspace Security Research Center, Peng Cheng Laboratory, Shenzhen, China
[5] Wuhan National Laboratory for Optoelectronics, Wuhan, China

Abstract*—Delta synchronization (sync) is a key bandwidth-saving technique for cloud storage services. The representative delta sync utility, `rsync`, matches data chunks by sliding a search window byte by byte, to maximize the redundancy detection for bandwidth efficiency. This process, however, is difficult to cater to the demand of forthcoming high-bandwidth cloud storage services, which require lightweight delta sync that can well support large files. Inspired by the Content-Defined Chunking (CDC) technique used in data deduplication, we propose Dsync, a CDC-based lightweight delta sync approach that has essentially less computation and protocol (metadata) overheads than the state-of-the-art delta sync approaches. The key idea of Dsync is to simplify the process of chunk matching by (1) proposing a novel and fast weak hash called FastFp that is piggybacked on the rolling hashes from CDC; and (2) redesigning the delta sync protocol by exploiting deduplication locality and weak/strong hash properties. Our evaluation results driven by both benchmark and real-world datasets suggest Dsync performs 2×-8× faster and supports 30%-50% more clients than the state-of-the-art `rsync`-based WebR2sync+ and deduplication-based approach.

***Index Terms*—rsync, content-defined chunking, cloud storage**

I. Introduction

The promises of being able to access data anywhere and anytime have made personal cloud storage services with application APIs such as Dropbox [1], GoogleDrive [3], and iCloud [4] increasingly popular. The burgeoning cloud storage services impose challenges to both the clients and servers with increasing network transmission and compute overheads, which motivates service providers to propose solutions. DropBox, for example, has adopted delta synchronization (sync for short) to reduce the amount of data traffic between the client and server [12], [19]. In spite of this, the sync traffic volume of DropBox still takes up to 4% of the total traffic due to the frequent interactions between clients and servers for calculating the delta (modified) data, according to a recent study [13] on several campus border routers. Therefore, reducing the traffic and compute overhead in synchronization remains an important challenge to be addressed.

As a classic model of synchronizing files between two hosts, `rsync` is capable of effectively reducing the traffic volume. Generally speaking, the `rsync` process synchronizes the client/server files f' (a modified version of f) and f in three phases as follows: ① The client sends a sync request to the server, and the server splits the server file f into fix-sized chunks, which is followed by the server calculating the fingerprints of the chunks to be sent to the client as the Checksum List. ② The client uses a sliding window moving on the client file f' byte-by-byte, to match possible duplicate chunks with the Checksum List from the server. ③ After finishing the byte-by-byte matching, the client obtains the mismatched chunks, referred to as Delta Bytes, and send them to the server, after which the server reconstructs the client file f' according to the Delta Bytes and the server file f. Several recent studies, including DeltaCFS [34], PandaSync [28], and WebR2sync+ [32], have improved over and on the basis of `rsync` in various aspects. However, the chunk-matching process using a byte-by-byte sliding window remains largely unchanged in these `rsync`-based approaches, which can be very time-consuming especially when the files to be synchronized are increasingly large in high-bandwidth cloud storage systems [22]. For example, according to our study and observation, WebR2sync+ [32] spends $\sim 10\times$ longer time on the byte-by-byte chunk-matching process than on the data transferring process over the Gigabit network when synchronizing a large file of 1GB.

The current delta sync is difficult to cater to the demand of cloud storage, given the inevitable challenges of inherently higher end-user network bandwidth and larger cloud-hosted file size. We notice that residential connectivity to the Internet has already reached 1 Gbps in access bandwidth, and the newly emerging 5G connectivity can even exceed 1 Gbps in access bandwidth. Accordingly, the cloud-hosted files have been growing in both quantify and (single-file) size. As a consequence, we have to rethink and innovate the current design of delta sync in order to catch up with this trend as well as to satisfy the user experience.

Recently, data deduplication, a chunk-level data reduction approach, has been attracting increasing attention in the design of storage systems [29]. Rather than using a byte-wise sliding window in the traditional compression approaches, data deduplication splits the files into independent chunks, usually using a technique called Content-Defined Chunking (CDC), and

* L. Xiang and Y. He equally contributed to this work.

then detects duplicates chunk-by-chunk according to chunks' fingerprints [23], [31]. Specifically, the CDC technique uses a small sliding window (e.g., size of 48 bytes) on the file contents for finding chunk boundaries and declares a chunk boundary found if the hash value of the sliding window (i.e., the file contents) satisfies a pre-defined condition. As a result, the chunk boundaries are declared according to the file contents instead of the file locations, which adequately addresses the 'boundary-shift' problem due to file modifications and thus detects more duplicates for data reduction. Therefore, we believe that the CDC technique can be effectively used in the sync protocol to eliminate the need for the time-consuming byte-wise comparison.

Although CDC has the potential to simplify the chunk-matching process for redundancy detection, incorporating CDC into the `rsync` protocol introduces new challenges, including extra compute overhead due to the rolling-hash based chunking and low redundancy detection ratio due to the coarse-grained chunk-matching after CDC. To this end, we propose Dsync, a lightweight CDC-based delta synchronization approach for cloud storage services, with less compute and protocol (metadata) overheads. To fully utilize the CDC approach in delta synchronization and address the challenges as mentioned above, we make the following four critical contributions in this paper:

- We use FastCDC to split the client and server files into independent chunks, which helps simplify the chunk-matching process in the `rsync` protocol.
- Proposing a novel weak hash, called FastFp, to replace Adler32 in `rsync` by effectively piggybacking on the Gear hashes generated from FastCDC. The new FastFp (CDC + weak hash) is much faster than Adler32 used in `rsync` while achieving nearly a comparable hash collision ratio.
- Redesigning the client/server communication protocol to reduce both compute overhead and network traffic by: ⓐ first checking weak hash and then computing & matching the strong hash of the weak-hash-matched chunks, to reduce most of the unnecessary compute of strong hash on the mismatched chunks, and ⓑ merging the consecutive weak-hash-matched chunks into a single large chunk to reduce the size of Match Token (metadata overhead) for network interactions in Dsync.
- Comprehensive evaluation driven by both real-world and benchmark datasets illustrates that Dsync performs 2×-4× faster and supports 30%-50% more clients than the state-of-the-art `rsync`-based WebR2sync+ [32] and the traditional deduplication-based solution [23].

The rest of this paper is organized as follows. Section II introduces the background and related work. In Section III, we discuss the deficiency of the state-of-the-art WebR2sync+ approach and the potential of the CDC-based approach. Section IV describes the design and implementation details of Dsync. Section V presents the evaluation results of Dsync, including comparisons with the latest `rsync`-based WebR2sync+ and deduplication-based solutions. Finally, Section VI concludes this paper and outlines future work.

II. Background and Related Work

Generally speaking, there are two approaches to synchronize a native file from the client to the server in the cloud storage service, full sync and delta sync. The former, which simply transfers the whole file to the server, is suitable for small files [28], while the latter, which only transfers the modified data of a file (i.e., the delta) to the server to minimize the network traffic, is suitable for large files.

Delta sync shows the most significant advantage when the files are frequently modified, e.g., files with multiple versions or consecutive edits. A recent study [13] on several campus border routers indicates that the traffic volume of DropBox, which uses delta synchronization, accounts for 4% of the total traffic due to the frequent interactions between clients and servers for calculating the delta (modified) data, highlighting the significance of reducing network traffic volume with delta synchronization. Actually, delta sync approaches have been widely studied by many works of literature including Unix `diff` [16], `Vcdiff` [18], WebExpress [15], `rsync` [26], Content-Defined Chunking (CDC) [9], [23] and delta encoding algorithms [21], [25], [30]. Representative sync techniques supported by the state-of-the-art cloud storage services are summarized in Table I and discussed next.

a) Sync tools supported by industry.: Commercial cloud storage services include Dropbox, GoogleDrive, OneDrive, Seafile and et cetera. [8]. The PC clients of both Dropbox and Seafile support delta sync, where DropBox's sync is based on Fix-Sized Chunking (FSC) and Seafile's sync employs Content-Defined Chunking (CDC). To alleviate the compute overhead, the Android client of Seafile uses full sync to avoid energy consumed by the delta calculation in sync. Besides, other cloud storage services such as GoogleDrive and OneDrive also choose to support full sync for simplicity.

b) Sync tools proposed by academia.: Most of the research proposals for sync tools support delta sync for better performance (to save network bandwidth and accelerate network transmission). `rsync` is a known synchronization protocol first proposed by Tridgell to effectively synchronize a file between two hosts over a network connection with limited bandwidth [26], [27]. This approach is later adopted as the standard synchronization protocol in GNU /Linux [5].

Recent studies, such as DeltaCFS [34], PandaSync [28], and WebSync [32], make innovative improvements on top of `rsync`. Specifically, DeltaCFS directly records the minor modifications to reduce compute and network overheads due to frequent synchronization. PandaSync strikes a tradeoff between full sync and delta sync since full sync can effectively reduce the round trip time (rtt) for small files between the client and server. WebR2sync+ [32] makes the first attempt to implement delta sync over Web browsers, which is a dominant form of Internet access. To avoid the influence of the poor performance of browsers when executing compute-intensive byte-wise comparison of hash values, WebR2sync+ shifts the

TABLE I
DATA SYNCHRONIZATION TECHNIQUES.

Source	Full Sync[1]	*Delta Sync*	
		Local Buffer	*Chunking[2] Methods*
DropBox (W/A) [8][3]	×	√	FSC
Seafile (W) [6]	×	√	CDC
Seafile (A) [6]	√	×	×
GoogleDrive (W/A) [8]	√	×	×
OneDrive (W/A) [8]	√	×	×
`rsync` [5]	×	×	FSC
DeltaCFS [34]	×	×	FSC
PandaSync [28]	√*	×	FSC
WebSync [32]	×	×	FSC
QuickSync [8]	×	√	CDC
LBFS [23]	×	×	CDC
UDS [20]	×	×	FSC
Dsync	×	×	CDC

1 √: full sync, ×: delta sync, and √*: selective full sync.
2 FSC: Fix-Sized Chunking, CDC: Content-Defined Chunking.
3 W: windows client, A: android client.

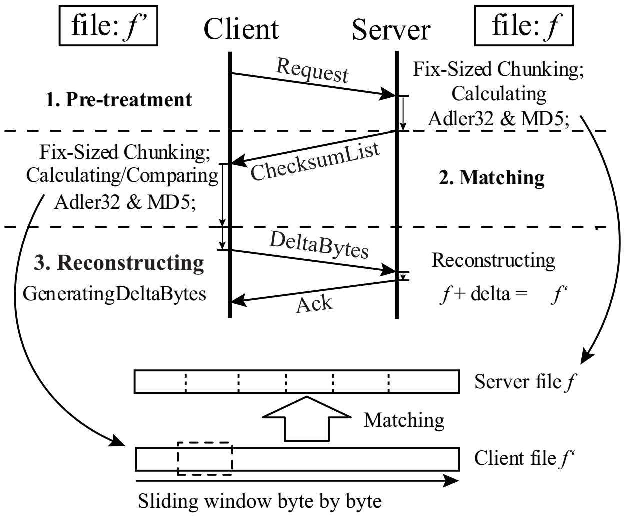


Fig. 1. Workflow chart of `rsync`.

comparison to the server and replaces the cumbersome MD5 hash function with a lightweight hash function called SipHash. Due to the boundary shifting problem, QuickSync [8] employs the CDC approach rather than `rsync`, where a dynamical chunking strategy is used to adapt to bandwidth changes and remove more redundancy of the local file. As one of the pioneers in data deduplication, LBFS [23] splits the file into chunks using the CDC technique, calculates and then compares their SHA1 fingerprints to detects duplicate chunks. It finally transmits the non-duplicate chunks from the client to the server. UDS (Update-batch delayed sync) [20] uses a batched sync strategy to avoid bandwidth overuse due to frequent modifications on the basis of Dropbox.

In summary, as shown in Table I, delta sync with local buffer requires the client to have extra storage and compute overheads while full sync is simple but not bandwidth-efficient. Despite the advantages of `rsync`-based synchronization approaches, the exceptionally high compute overhead of the `rsync` protocol due to byte-wise comparison, and hash calculation severely limits its applicability to resource-constrained client systems, especially for synchronizing large files. Thus, in this paper, we focus on providing a lightweight, portable delta sync approach on resource-constrained client systems, via Web browsers on Mobile phones, IoT devices and et cetera. Thus, it is quite inconvenient for such a resource-constrained Web browser to maintain the metadata buffer of the client files for delta sync. On the other hand, deduplication-based sync [23] offers a possibility to avoid byte-wise chunk-matching in `rsync`. Note that our work is substantially different from all the previous works (shown in Table I) in that it is the first attempt to combine the CDC approach with the traditional `rsync` model to effectively synchronize data when the client is resource-constrained, without local buffer and sufficient compute capacity for executing `rsync`-like delta sync protocols.

III. CHALLENGES AND MOTIVATION

Problems of rsync. To better understand the strengths and weaknesses of `rsync` [27], we first illustrate how it works with the help of Figure 1 that shows a three-phase workflow.

- In Phase 1, when a client needs to synchronize a file, it first sends a request (including the name of the local file f') to the server. Upon receiving the client request, the server starts to chunk the server file f by Fix-Sized Chunking and calculate weak but fast-rolling hashes (i.e., Adler32) as well as a strong but slow hash (i.e., MD5) of the chunks as their weak and strong fingerprints. Both weak and strong fingerprints are included in the Checksum List and sent to the client.
- In Phase 2, the client slides a fixed-size window on the file f', to generate chunks and their weak hash Adler32 to match with Adler32 fingerprints of file f in the Checksum List. Note that: ① if the current chunk under the sliding window does not match any Adler32 fingerprints in the Checksum List, the window will slide further byte by byte until a matched chunk is found. ② If the weak-hash-matched chunk is found, its strong hash MD5 will be then calculated and checked in the Checksum List to avoid the case of weak hash collision, and the sliding window will slide by the size of the window after confirming the matched chunk. This phase takes a rather long time due to the byte-by-byte sliding window-based chunk matching.
- In Phase 3, the client generates Delta Bytes, including the mismatched chunks and their metadata, and sends them to the server where the file is synchronized/reconstructed with f' = f + Delta Bytes.

WebR2sync+ is implemented based on `rsync`, assuming that the client is a Web browser. Considering the low compute capacity on a Web browser, WebR2sync+ shifts the chunk-matching process from the client to the server. Its workflow is also three-phased, as shown in Figure 2.

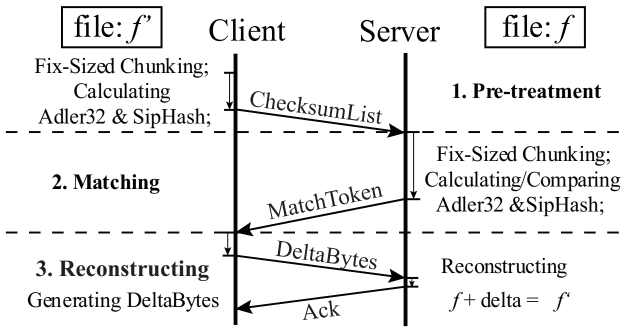


Fig. 2. Workflow chart of WebR2sync+.

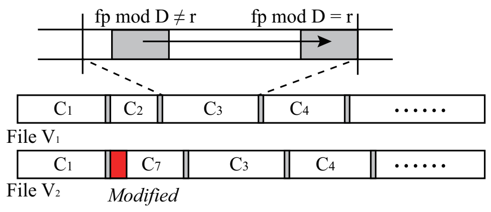
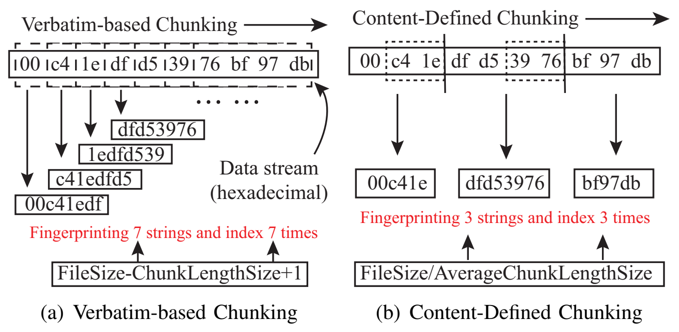


Fig. 3. The CDC technique for the chunk-level data deduplication. A chunk cut-point is declared if the hash value "fp" of the sliding window satisfies a pre-defined condition.

- In Phase 1, in contrast to `rsync`, the pre-treatment (i.e. the chunking and hashing of the file) is moved from the client to the server, and the strong hash is replaced by SipHash, a faster strong hash function.
- In Phase 2, the chunk-matching process is moved from the client to the server, and the strong hash calculation is also changed to SipHash. The operation in this phase is almost the same as `rsync`. After that, the Match Token that indicates which chunks match will be sent to the server.
- In Phase 3, the client generates the Delta Bytes to be sent to the server and file *f'* is reconstructed according to file *f* and Delta Bytes, which is the same as `rsync`.

Although WebR2sync+ improves on `rsync`, it does not address the challenges facing the latter fundamentally. More specifically, in Phase 2, the time-consuming byte-wise chunk-matching remains unchanged in WebR2sync+, which is especially problematic for large files or files with significant modifications. In other words, the matching window needs to slide from the beginning to the end of a file byte by byte, comparing every byte in the worst case [14], [21], as shown in Figure 4(a). By contrast, the Content-Defined Chunking technique used in data deduplication provides an opportunity to avoid the cumbersome calculation of overlapping chunks, as shown in Figure 4(b).

Advantages of Content-Defined Chunking (CDC). CDC is proposed to solve the "boundary-shift" problem. As shown in Figure 3, CDC uses a sliding-window technique on the content of files and computes a hash value (e.g., Rabin [17], Gear [30]) of the window. A chunk cut-point is declared if the hash value satisfies some pre-defined condition. Therefore, as shown in Figure 3, by using the CDC technique, chunks C_3 and C_4 of



Verbatim-based Chunking
00 c4 1e df d5 39 76 bf 97 db
dfd53976
1edfd539
c41edfd5
00c41edf
Data stream (hexadecimal)
Fingerprinting 7 strings and index 7 times
FileSize-ChunkLengthSize+1
(a) Verbatim-based Chunking
Content-Defined Chunking
00 c4 1e df d5 39 76 bf 97 db
00c41e
dfd53976
bf97db
Fingerprinting 3 strings and index 3 times
FileSize/AverageChunkLengthSize
(b) Content-Defined Chunking

Fig. 4. Comparison of Verbatim-based Chunking and Content-Defined Chunking used for string matching. CDC-based approach executes much fewer fingerprinting and searching operations.

file V_2 will still be identified for data deduplication (with file V_1) although their boundaries have been shifted or changed due to file modification.

In contrast, as mentioned earlier, `rsync`-based approaches employ a verbatim-based chunking technique, which is time-consuming. We believe that this CDC technique can be utilized for chunking in `rsync` to significantly reduce the compute cost for hash calculation and chunk indexing. To better demonstrate the advantages of CDC, Figure 4 contrasts the process for chunk boundary identification in `rsync`-based approaches, which we refer to as verbatim-based chunking, and that in CDC-based approaches. CDC-based approaches generate much fewer chunks for fingerprinting and indexing than the traditional `rsync`-based approaches, especially for the file of considerable size.

Disadvantages of CDC-based approach. Obviously, CDC introduces additional compute overhead for delta synchronization, i.e., computing rolling hashes for chunking. Further, CDC leads to slightly more network traffic than `rsync`-based approaches since it may fail to eliminate redundancy among similar but non-duplicate chunks (i.e., the very similar chunks C_2 and C_7 in Figure 3).

Nevertheless, the CDC technique remains attractive because it greatly simplifies the chunk fingerprinting and searching process. To fully leverage the advantages while avoiding disadvantages of the CDC technique, we aim to propose a new `rsync`-based approach using the CDC technique in this paper to simplify the delta sync process while overcoming the aforementioned problems of CDC, i.e., lower compression ratio problem and additional compute overhead.

IV. Design and Implementation

A. Architecture and Algorithm Overview

Architecture overview. To address the problems in `rsync`, Dsync introduces Content-Defined Chunking and makes optimizations above that. As depicted in Figure 5, the Dsync architecture for delta synchronization consists of four key functional components, namely, Content-Defined Chunking, weak/strong hashing, communication protocol, and hash matching.

- Content-Defined Chunking. In Dsync, the client file *f'* (to be synced) and server file *f* will be first divided

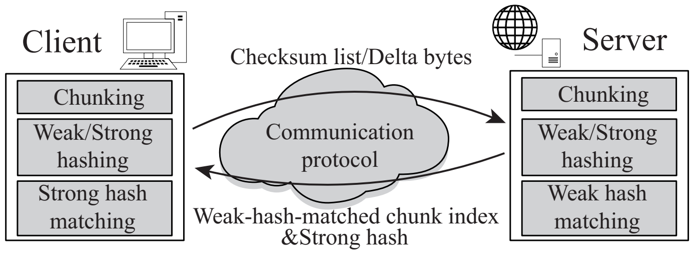


Fig. 5. Dsync Architecture overview.

into several chunks by Content-Defined Chunking (i.e., FastCDC) for future detection of duplicate chunks.

- Weak/strong hashing. In Dsync, two-level hash values (i.e., weak/strong hash) are calculated for checking duplicate chunks. The weak hash, which is fast with low compute overhead, is employed to first quickly check the potential duplicate chunks that, once identified, are further confirmed by the strong hash, which is cryptographically secure, to avoid a hash collision. Otherwise, if the weak hash mismatches, the chunk is marked as unique and strong hash (SHA1) calculation will be skipped.
- Hash matching. In this module, the weak hash values of chunks are compared first to find the potentially-matched chunks on the server and their strong hash values will be further confirmed on clients. Only the chunks whose strong hash values match will be regarded as duplicate chunks. Otherwise, they will be regarded as new chunks (i.e., delta data) and sent to the server.
- Communication protocol. It is responsible for interactions between client and server in Dsync, to check the above mentioned weak and strong hash values of files to obtain the delta data for synchronization ultimately.

Algorithm overview. Introducing FastCDC in `rsync` is presented in Section IV-B, but we find the following two problems:

- CDC algorithm brings extra chunking overhead. We find it feasible to utilize the chunking algorithm to generate a weak hash to replace Adler32 in `rsync`. In this way, the extra chunking overhead counterweights the CPU overhead for generating weak hash fingerprints. It will be discussed in section IV-C.
- The original communication protocol for `rsync`-based synchronization is not efficient for the CDC-based approach. Specifically, we find that the strong hash computation and comparison used in `rsync` would also be unnecessary in Dsync if the chunks' weak hashes mismatch, which motivates us to redesign the communication protocol for Dsync. It will be discussed in section IV-D.

With coordinated operations of the above four components in Dsync, mismatched chunks will be sent to the server as Delta Bytes. A detailed workflow of Dsync with these four key components is presented next.

B. Baseline:Dsync using FastCDC

Among existing proposals aimed to accelerate the chunking speed, FastCDC [31] appears to be a right candidate for `rsync` with low compute overhead for chunking. In our current implementation of Dsync, we directly transplant FastCDC into Dsync for chunking based on WebR2sync+ [32] (one of the latest improved version of `rsync`). Further, we replace SipHash with SHA1 in WebR2sync+, since it is a widely acknowledged cryptographically secure hash algorithm for data deduplication [29], [35]. As shown in Figure 6, the current implementation of Dsync consists of the following three phases:

- *Pre-treatment Phase.* In this phase, the client splits the file into chunks via FastCDC [31] and calculates their weak and strong hash values (i.e., Adler32 and SHA1) as `rsync` does. After that, the fingerprints are sent to the server. Compared with the model of the `rsync` protocol, where files are divided into fix-sized chunks, the CDC approach incurs extra compute overhead. Therefore, this phase is theoretically slower than that of `rsync` (see Figure 2).
- *Matching Phase.* Upon receiving the Checksum List (i.e., Adler32 and SHA1 values of the client file *f'*), the server starts to chunk the server file *f* and calculate hash values as the client does. Then the server searches the Checksum List (of the client file *f'*) for the duplicate chunks in file *f*, and the metadata of matched chunks will be sent back to the client. Our model compares the data chunk by chunk while the `rsync` model compares the data byte by byte if the chunk fingerprints of the server file *f* do not match any hash values of the client file *f'*. Theoretically, our approach is much more efficient than `rsync` in this phase.
- *Reconstructing Phase.* Upon receiving the Match Tokens, the client will send the literal bytes of the chunks that do not match, also known as the Delta Bytes. Based on the Delta Bytes from the client and the server file *f*, the server reconstructs the client file *f'* = *f*+delta. In this phase, Dsync is almost the same as `rsync`.

In essence, this current version of Dsync simplifies the comparison for the duplicate chunks for delta synchronization by employing CDC. But it has two weaknesses as discussed in Section III: ① additional compute overhead from CDC, ② low redundancy detection ratio due to the coarse-grained chunk-matching after CDC. For the first weakness, we will utilize the hash values generated by Content-Defined Chunking in FastCDC, as the weak hash (for duplicate pre-matching), to compensate for additional compute overhead for chunking, as detailed in Section IV-C. For the second weakness, we will redesign the communication protocol to minimize the size of metadata for the Match Token in the matching phase and also minimize the hash calculation in the whole Dsync workflow, as detailed in Section IV-D.

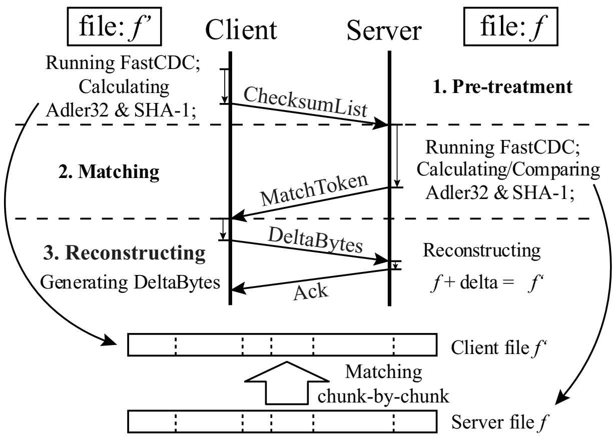


Fig. 6. Preliminary FastCDC-based Dsync prototype.

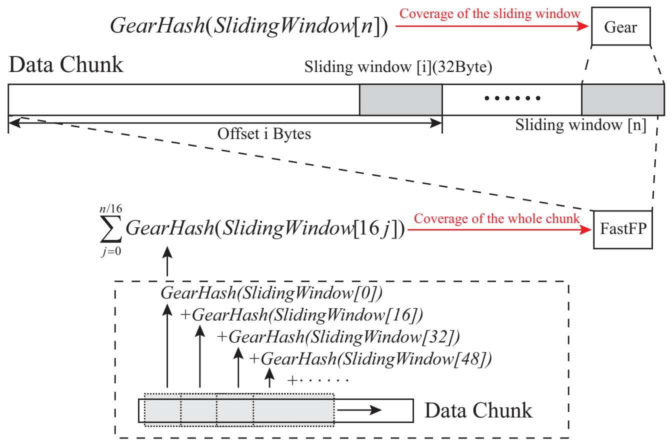

GearHash(SlidingWindow[n])
Coverage of the sliding window
Gear
Data Chunk
Sliding window [i](32Byte)
Offset i Bytes
Sliding window [n]
$\sum_{j=0}^{n/16} GearHash(SlidingWindow[16j])$
Coverage of the whole chunk
FastFP
GearHash(SlidingWindow[0])
+GearHash(SlidingWindow[16])
+GearHash(SlidingWindow[32])
+GearHash(SlidingWindow[48])
+......
Data Chunk

Fig. 7. A schematic diagram of FastFp using Gear hash.

C. FastFp Implementation

As discussed in the last subsection, in the pre-treatment phase in Dsync, the CDC technique incurs extra calculation even though FastCDC is very fast. Since CDC is sliding on the data by using the rolling hash algorithm, it is feasible to quickly obtain the weak hash for Dsync according to the rolling hashes during CDC. This method will replace the additional calculation for weak hash (i.e., adler32, also a rolling hash) in the traditional `rsync` approaches.

$$fp = (fp << 1) + G(b) \tag{1}$$

As shown in Equation 1, FastCDC uses fewer operations to generate rolling hashes through a small random integer table $G[]$ to map the values of the byte contents b, called Gear-based hashing, to achieve higher chunking throughput than other rolling hash algorithms [31]. The Gear hash is also weak, but compared with Adler32 it only represents the hash of several bytes because of "rolling", and thus has a high collision ratio if we directly use the Gear hash as the weak hash for an 8KB chunks. Fortunately, according to our observation on FastCDC-based data deduplication, Gear hash in FastCDC can be improved to achieve similar hashing efficiency as the classic adler32 used in `rsync`.

Therefore, to reduce Gear hash collisions, we develop FastFp, a novel and fast weak hash, as shown in Figure 7. The length of the sliding window is set to 32 Byte. According to the Equation 1, the original Gear hash is only relevant to the content that is 32-Byte away from the cut point due to the shift operation for "rolling". To make the Gear hash representative of the global content, we can combine the Gear hash of different windows through a particular operation, say "+" as an example (operation "$\oplus$" will be compared and evaluated later as well in Section V-B), to generate a new hash called "FastFp". Specifically, the fingerprint of the sliding window will be added every time the window slides over a certain distance, say 16Bye as an instance, and different distances will also be evaluated and study later in Section V-B.

The most significant difference between FastFp and Gear hash is that Gear is only related to a very limited range of the chunk (i.e., the size of the sliding window) while FastFP is related to all the contents of the chunks by using an extra operation '+' to combine many Gear hashes into one hash value (as illustrated in Figure 7). These addition operations are so lightweight as not to influence the overall performance of FastCDC. In this way, FastFp is relevant to the data of the whole chunk.

Therefore, based on the original FastCDC, FastFp is very fast since it only adds one addition ("+") operation after each time the sliding window moves 16 bytes.

D. Client/Server Communication Protocol

In this subsection, we propose a novel communication protocol to reduce the compute overhead while simultaneously minimizing the network traffic.

Reducing strong hash compute overhead. According to our observation of FastCDC-based Dsync prototype, as shown in Figure 6, Dsync can reduce the compute of strong hash on the data chunks whose weak hash values mismatch. Therefore, the strong hash values computed on the clients are essentially not needed in the pre-treatment phase. Based on this observation, we establish a new delta sync protocol that minimizes the compute overhead for strong hash values. The new protocol is shown in Figure 8 with several improvements on the three phases of Dsync as follows:

- In Phase 1, Dsync splits the client file f' into several chunks via FastCDC with their weak hash (FastFp) values generated. Then FastFp hash values and the lengths of chunks are packed into the Checksum List (no longer strong hash SHA1 now) and sent to the server.
- In Phase 2, upon receiving the Checksum List, the server splits the file f into chunks by FastCDC and also obtains these chunks' FastFp. After that, the server searches the Checksum List for the weak hash values of chunks in f. If the weak hash value of a chunk is matched, the server computes the SHA1 hash value of the matched chunk in file f for further confirmation by sending Match Tokens

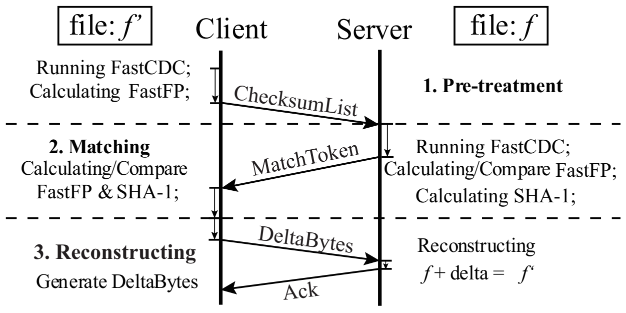


Fig. 8. The redesigned protocol of Dsync to minimize the strong hash calculation.

(including all the matched chunks indices of file *f'* and the SHA1 values of the matched chunks in file *f*) to the client. Then the client checks the SHA1 values of the weak-hash-matched chunks of file *f'* according to the Match Tokens.

- In Phase 3, the mismatched chunks, together with their indices, form Delta Bytes, are then sent to the server. Finally, the server reconstructs the client file *f'* according to the server file *f* and Delta Bytes from the client.

Compared with the preliminary FastCDC-based Dsync protocol in Figure 6, the client does not calculate the SHA1 value until it receives the Match Tokens from the server. In consequence, the task of searching SHA1 values is shifted to the client. Moreover, only the weak-hash-matched chunks will be calculated SHA1 for confirming duplicates; thus, the SHA1 calculation is minimized in Dsync, as shown in Figure 8.

Reducing network traffic. Inspired by the widespread existence of redundancy locality observed by many deduplication studies [29], [35], which states the observation that the duplicate chunks say, A, B, and C, in a file appear in approximately the same order throughout multiple full backups or similar files with a very high probability, we believe that this redundancy locality can also be exploited in Dsync for network traffic reduction. Precisely, the consecutive chunks (potentially duplicate) matched by their weak hashes can be collectively regarded as a much larger chunk, which can significantly reduce the metadata traffic (i.e., the number of Match Tokens) in Dsync.

Figure 9 shows how we merge several consecutive weak-hash-matched chunks into a larger one in Dsync. Upon receiving the Checksum List of the client file, the server also chunks and fingerprints the server file *f* with weak hash as the client does. After that, the server compares the weak hash of its file with that contained in the Checksum List. The consecutive weak-hash-matched chunks will be treated as a single larger chunk to compute strong hash, which will be sent back to the client later and compared. If the strong hash fingerprint of merged single one chunk does not match that of the client, we will transfer the strong hash fingerprints of the constituent chunks when the merged chunk is beyond a predefined threshold (we set it up as 800KB). Otherwise, the mismatched chunk will be directly transferred to the server.

Collision probability. The implementation of merging chunks is based on the assumption that the chunks with the same weak hash value are likely to have the same strong hash value with a very high probability. To make this clear, as we did earlier, we suppose that the probability of weak hash collision is p_1. It can be argued that the main reason for the hash collision of a merged chunk is that the weak hashes of modified chunks among those constituting the merged chunk are the same as their corresponding weak hashes before their modifications. That is to say, hash collision happens on all modified chunks. If the merged chunk contains n chunks of which m are modified, the conditional probability of hash collision of the merged chunk with modifications to exactly m constituent chunks is thus $(p_1)^m$. The final hash collision probability of the merged chunk is the weighted mean of the hash collision probabilities under all modification conditions, and we assume that the probability of any m chunks being modified is c_m. Since $1 \le m \le n$, the probability of a merged chunk with n constituent chunks to be falsely considered matched (false positive) can be computed as

$$P_{Collision} = \sum_{m=1}^{n} c_m (p_1)^m, \tag{2}$$

$$where \qquad \sum_{m=1}^{n} c_m = 1, \quad 0 \le c_m \le 1. \tag{3}$$

According to Equation 2, we can find that the probability of a false positive for detecting a matched merged chunk is the linear combination of the collision probabilities under different modification conditions. Here c_m represents the probability of the various number of colliding chunks occurrence. Whatever c_m is, the collision probability always satisfies:

$$(p_1)^n \le P_{Collision} \le p_1. \tag{4}$$

In other words, the probability of the hashes of the merged chunks colliding is lower than or equal to the hash collision probability of a single chunk, which is acceptably low for our design.

Benefit of merging chunks. Based on the locality of duplicate chunks (redundancy locality), we can merge the weak-hash-matched chunks together to reduce the amount of metadata transmitted. In this scheme, there will be only one strong hash to send for consecutively weak-hash-matched chunks. Specifically, if the file is as large as 10MB and only modified on one data chunk (with avg. chunk size of 8KB for CDC in Dsync), the merged chunk, however, only sends about 60B SHA1 (three chunks, two duplicates, one unique) because there will be only three chunks after merging; while the unmerged chunks will send about 25KB SHA1 (about 427× reduction by Dsync).

Solution to tackle chunk collision. If the strong hash of the merged chunk mismatches, there are two options to handle this problem. One is to compare the strong hashes of the individual chunks of the merged chunk and then transmit the strong-hash-unmatch chunks, and the other is to transmit all the constituent chunks of a merged chunk directly. Our scheme chooses the

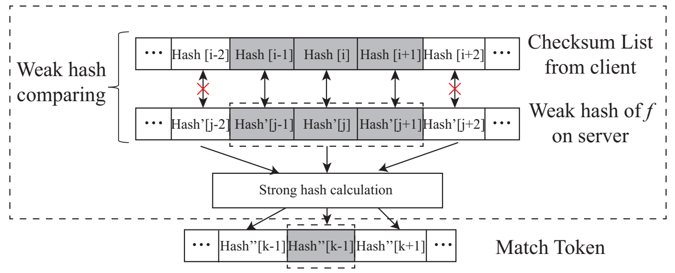


Fig. 9. Merging several consecutive weak-hash-matched chunks into one chunk to reduce metadata overhead.

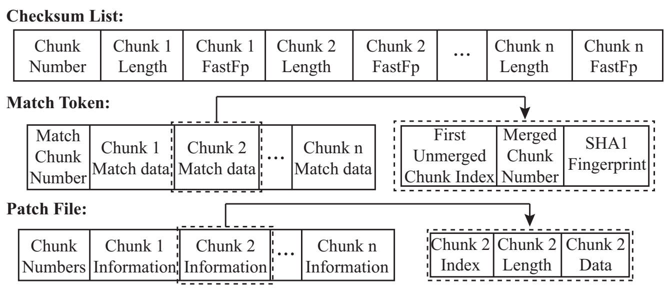


Fig. 10. The detailed format of data transferred in the three network interactions in Dsync.

latter since the hash collision probability of a merged chunk is very low, as discussed above.

Finally, the communication between the client and server is based on the WebSocket protocol. The protocol regulates a per-message-deflate compression extension [33], and most of the mainstream Web browsers support it and the data transmitted is usually compressed by default.

E. Other Implementation Issues

In this subsection, we will discuss other implementation issues in Dsync for better understanding of its design.

Cross-platform based implementation. Dsync is designed as a lightweight Web-based synchronization service accessible on most devices without requiring client buffer. The Web browser is one of the most prevailing accesses to the Internet and a Web-based service has the advantage to be cross-platform, while the PC or portable device clients have to be compiled and maintained for different operating systems. The application scenario of Dsync includes but is not limited to devices with low compute capacity (i.e., many resource-constrained systems, such as Mobile phone, IoT devices and et cetera). Note that data compression is a common practice to reduce the network traffic, and both Dsync and WebR2Sync+ enable the default compression of sync message supported by the Web browser. In general, the data compression is an attempt to trade-off between network traffic size and CPU overhead. Compression slows down the overall synchronization under sufficiently high bandwidth but makes it faster when network is insufficient. It is observed that default compression makes Dsync faster only if the network is below 100 Mbps.

Data structures for interactions. Figure 10 shows how data transferred in the corresponding three key interactions is formatted in Dsync (see Figure 8). In Phase 1, the Checksum List starts with the total number of chunks, followed by the weak hash and length of chunks one by one. In this phase, the data index is not attached, but the chunk sequence is stored on the server. In Phase 2, the Match Token indicates the matched chunks and starts with the total number of chunks (note that here each chunk is merged from the consecutive matched chunks), followed by the metadata of each chunk. The metadata of each chunk is composed of three parts, the first unmerged chunk index, the number of the constituent chunks of a merged chunk and strong-hash fingerprint. In Phase 3, the data also starts with the number of unique chunks , and it is followed by the detailed information of these unique chunks. The chunk information is composed of its index, length, and content. Note that the index mentioned above refers to the sequence number of the chunk in Checksum List.

V. Evaluation

In this section, we first introduce the experimental setup for Dsync. And then we conduct a sensitivity study of Dsync. Besides, an overall comparison between Dsync and the state-of-art `rsync`-based WebR2sync+ [32] and deduplication-based solution [23], [31] is evaluated and discussed.

A. Experimental Setup

Experimental Platform. We run the experiments on a quad-core Intel i7-7700 CPU, 16GB memory PC with Windows 10 operating system and a 6GB RAM, 64GB ROM mobile phone with Huawei honor V10. The PC client runs on Chrome v76.0 (windows) while the mobile phone client is on Chrome v74.0 (Android). Besides, the server runs on node v12.8 with a quad-core virtual machine @3.2GHz (installed Ubuntu Server 16.04 with 16GB memory and 128GB disk). To simulate actual network status, we tune the bandwidth to be 100Mbps and rtt to be 30ms.

Performance Metrics. We evaluate delta sync approaches in terms of two main metrics: *sync time* and *sync traffic*. The *sync time* metric refers to the time spent on the whole sync process. The *sync traffic* metric measures the total amount of data transmitted, including the Checksum List, Match Token, and Delta Bytes, as discussed in Section IV (see Figure 8). Note that, although the transmitted data is compressed by default by the Web browsers, the *sync traffic* measured and evaluated in this paper is the volume of data traffic before compression. For each data point reported we run the experiment five times to obtain a statistically meaningful average measure for the delta sync performance.

Delta Sync Configurations. We build our Dsync on top of the open-sourced WebR2sync+ [32]. Dsync is written in a total of ∼2000 lines of JavaScript and ∼200 lines of C code. The deduplication-based delta sync solution (Dedup for short) is based on LBFS [23] except that we use the latest FastCDC [31] to replace the Rabin-based CDC for higher chunking speed. Therefore, we use open-sourced

WebR2sync+ [32] and FastCDC-based Dedup as baselines for performance comparison with Dsync. To be fair, all the experiments enable the default compression since the original WebR2Sync+ also employs that [32]. In the evaluation, we use the average chunk size of 8KB, which is also used in WebR2sync+ [32] and LBFS [23].

Benchmark Datasets. Silesia [10] is a widely acknowledged dataset for data compression [11] covering typical data types that are commonly used, including text, executables, pictures, htmls and et cetera. According to several published studies on real-world and benchmark datasets [24], [30], the file modifications are made at the beginning, middle, and end of a file with a distribution of 70%, 10%, and 20% respectively [24]. Similar to the operations in QuickSync [8] and WebR2sync+ [32], modifications in the forms of ‘cut’, ‘insert’, and ‘inverse’ are also made on the original data, where ‘inverse’ represents flipping the binary data (e.g., inversing 10111001 to 01000110). To generate the benchmark datasets, we cut 10MB out of Silesia corpus and make modifications to the file with the modification size of 32B, 256B, 2KB, 16KB, 128KB and 1MB on it respectively. Three types of file modifications are made following the pattern described above, and each fraction for the file modifications takes 256B at most. In our benchmark datasets, under 10% modification with each modification of 256B, over 90% of the data chunks will be unique if we use the average chunk size of 8KB for delta synchronization.

Real-World Datasets. In addition to the synthesized datasets above, we use the following four real-world datasets to evaluate the performance of the delta sync approaches:

- *PPT.* We collect 48 versions of a PowerPoint document from personal uses in the cloud storage (totalling 467MB).
- *GLib.* We collect the GLib source code from versions 2.4.0 through 2.9.5 sequentially. The codes are tarred, and each version is of almost 20MB, totalling 860 MB.
- *Pictures.* We use a public picture manipulation dataset [7], which contains 48 pictures in the ‘PNG’ format (totalling 280 MB) with no lossy compression and the manipulation pattern is to paste a certain area of a photo in another place.
- *Mails.* We collect some mails from a public mail dataset [2] and each mail contains the past replies. The mails are tarred with two versions by time, and the total size is about 1839 MB.

B. *Performance of the Weak Hash FastFp*

Hashing speed and hash collision ratio are the two most important metrics for evaluating the effectiveness of weak hash FastFp. In this subsection, to evaluate the hash collision, we generate five files consisting of random numbers for the given size of 1GB, 10GB, and 100GB (totalling 15 files). Since the sync service is implemented in JavaScript (our evaluation environment), we run FastFp, Gear, and Adler32 via node.js on the Chrome browser of the PC with i7-7700 CPU. As has been stated in Section IV-C, the operation includes “XOR” and “PLUS” while the sliding window is applied with the operation every time it slides over 8B, 16B and 32B, totalling 6 different configurations. The Gear hash here refers to the hash value of the sliding window using FastCDC when the boundary of a chunk is found during chunking.

TABLE II
THROUGHPUT AND COLLISION RATIO COMPARISON. HERE FASTFP IS EVALUATED WITH DIFFERENT CONFIGURATIONS: “$\oplus$” IS FOR “XOR”, “+” IS FOR “PLUS”, WINDOW SLIDING DISTANCE IS OF 8B, 16B, 32B.

Algorithms	Thpt (MB/s)	Hash collision ratio under different random file size		
		100GB	10GB	1GB
Adler32	295.6	2.3×10^{-10}	2.3×10^{-10}	2.4×10^{-10}
Gear	527.8	6.8×10^{-7}	6.8×10^{-7}	6.8×10^{-7}
FastFp($\oplus$ 8B)	460.0	2.2×10^{-10}	2.2×10^{-10}	3.1×10^{-10}
FastFp($\oplus$16B)	459.2	2.3×10^{-10}	2.3×10^{-10}	1.5×10^{-10}
FastFp($\oplus$32B)	460.4	2.2×10^{-10}	2.4×10^{-10}	3.1×10^{-10}
FastFp(+ 8B)	396.9	2.3×10^{-10}	2.4×10^{-10}	2.8×10^{-10}
FastFp(+16B)	397.5	2.3×10^{-10}	2.2×10^{-10}	1.8×10^{-10}
FastFp(+32B)	409.6	2.3×10^{-10}	2.3×10^{-10}	4.0×10^{-10}

Table II shows the averaged hashing speeds and collision ratios of Adler32, Gear, and FastFp with different configurations. The evaluation is to calculate the hashes of chunks with an average chunk size of 8KB on the random number workloads, which is often used for evaluating hash efficiency. The metric collision ratio listed in Table II is calculated based on Equation 5 as defined below, which is the total combination of two unique chunks with identical weak hash fingerprint divided by the combination of picking any two chunks from all the chunks.

$$p_{collision} = \frac{\sum_{i=2}^{n} n_i C_i^2}{C_n^2} \tag{5}$$

Here i represents the number of collision occurrences for a given weak hash while n_i corresponds to the number of such i-times colliding weak hash fingerprints (e.g., if ten different chunks are sharing the same weak hash and there are seven such weak hash fingerprints, then i equals to 10 and n_i equals to 7). The results exactly prove the uniformity of our FastFp. Essentially speaking, picking two colliding chunks is equivalent to finding a new chunk that has the same weak hash fingerprint as the given one. Therefore, the probability is supposed to be $1/2^{32}$ if the output of the hash function is uniformly distributed, which is consistent with our experiments. For a small file, say 100MB (12800 chunks in total and according to table II, colliding probability is 2.3×10^{-10} at maximum), the maximum colliding-chunk number is calculated as only 0.04 (according to Equation 5, we can get the biggest colliding-chunk number when $n_i = 0$ if $i \neq 2$) when chunked with FastFp at an average size of 8KB, indicating a quite low possibility for collision occurrence.

Table II suggests that under the different file sizes, the hash speeds of FastFp reaches almost 400-460MB/s, about 50% higher than Adler32. Besides, FastFp using “$\oplus$” is faster than using “+” while the sliding distance nearly does not influence the hashing throughput. The fast speed of FastFp is due to that FastFp (running on the top of Gear) consumes fewer

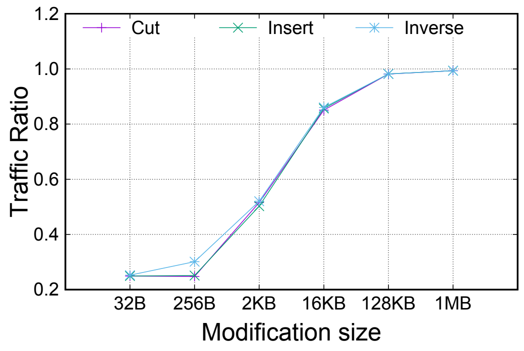


Fig. 11. Total traffic ratio as a function of the modification size, after/before applying the Dsync protocol that merges consecutive weak-hash-matched chunks.

instructions for the rolling hash comparing with Adler32 [31]. For the collision ratio issue, it can also be seen that the bigger the dataset is, the higher the probability of hash collision will be (i.e., finding the two different chunks have the same weak hash). Here we can observe that FastFp and Adler32 have almost identical hash collision ratios that are about three orders of magnitude lower than the original Gear, while the operation type and sliding distance has almost no influence on the collision ratio. The high collision ratio of Gear stems from the fact that it only involves the sliding window of contents in the chunk boundary. It is worth noting that the hash collision ratio is acceptably low since our synced files are usually smaller than 1GB.

In summary, FastFp (CDC + weak hash), a new hash function utilizing the byproduct of (hash values generated by) FastCDC, is 1.5X faster but no weaker than the widely acknowledged Adler32 weak hash used in `rsync`. To achieve the best performance, we choose FastFp with "XOR" operated every time the window slides over 16B as default for other experiments.

C. Performance of the Redesigned Protocol

Our redesigned sync protocol in Dsync aims to compute as little strong hash (i.e., SHA1) as possible while minimizing the amount of metadata transmitted over the network.

Figure 11 shows the ratio of total traffic over the network after/before applying our Dsync protocol that merges consecutive weak-hash-matched chunks. When there are fewer modifications (e.g., only inserting 32B), the total traffic is reduced by 70% $\sim$ 80% in Dsync. As the size of modifications grows, this ratio approaches 100%, and the trends for the three types of modifications are almost identical. This is because, as more modifications are made to the file, the traffic is gradually dominated by the Delta Bytes since fewer, if any, weak-hash-matched chunks will be found. Specifically, for a 10MB file using the average chunk size of 8KB, the transmission of almost 1280 chunks' SHA1 values, i.e., totalling 25KB of metadata, will be eliminated. However, the reduced metadata will be far overshadowed by the much larger amount of Delta Bytes transmitted from the client to the server if too many modifications are made to the file.

Figure 12 shows the sync time spent on the client as a function of the modification size in WebR2sync+ and Dsync (with the improved sync protocol) respectively. The bulk of the client time is spent on generating Checksum List in the pre-treatment phase and Delta Bytes in the reconstructing phase. Thus, the sync time for both delta sync approaches is almost the same because both of them must calculate the strong hashes of all chunks. As the size of modifications grows, the client time of our improved protocol decreases significantly starting at 16KB. This is because when more modifications are made to the client file, Dsync detects fewer weak-hash-matched chunks, leading proportionally fewer SHA1 circulations of unmatched chunks. Note that, unlike the `rsync` protocol where the matching process is based on the technique of the aforementioned Verbatim-based Chunking, the matching process in Dsync has been greatly simplified by using the FastCDC technique. That is, the strong hash calculations of mismatched chunks on the client are eliminated owing to the advantages of the CDC approach used in our Dsync design.

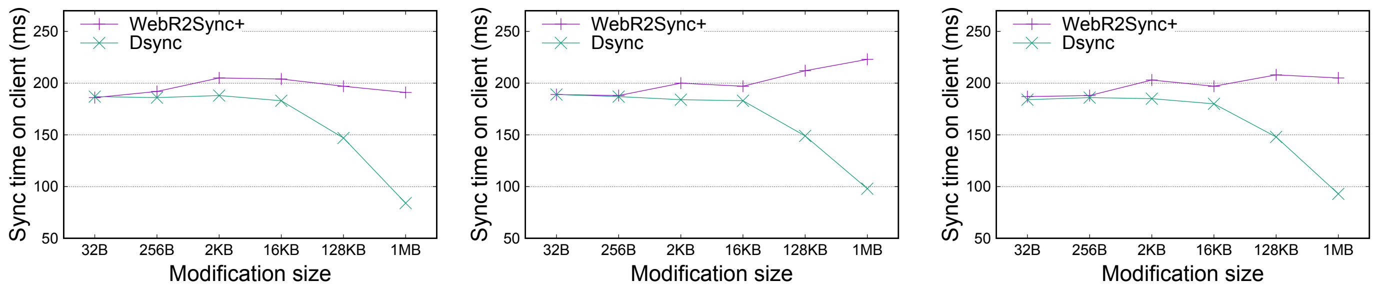
In summary, the redesigned protocol in Dsync can not only effectively reduce the metadata transmitted over the network but also reduce the compute overhead on the client-side.

D. Putting It All Together

In this subsection, we evaluate the overall performance of Dsync. The evaluation includes the sync time breakdown of Dsync, the impact of file size on Dsync performance, and the capacity of supporting multiple clients and et cetera.

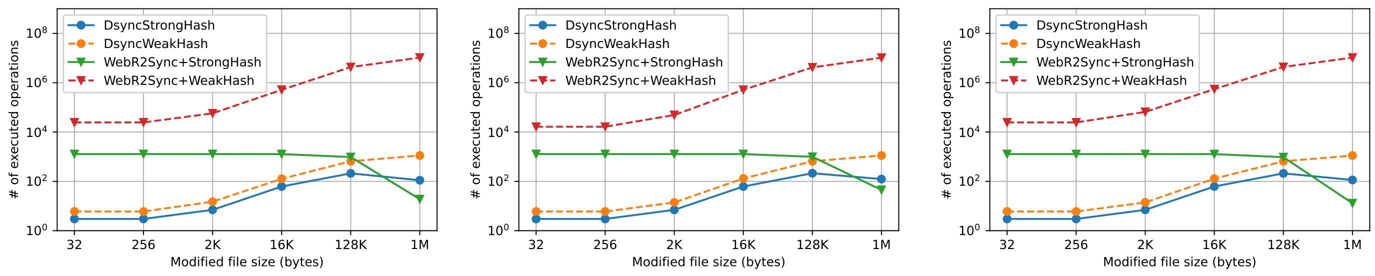
In essence, introducing a CDC technique in an `rsync`-like synchronization service can reduce the CPU overhead for weak hash lookups. Figure 13 shows Dsync's improvement in searching and comparing hash values over WebR2Sync+. When comparing weak hashes, the server constructs and searches a hash table according to the Checksum List. In the next stage, the client only compares the strong hashes of the weak-hash-matched chunks. The curves show a consistent trend under three file modification patterns. The dash lines show the numbers of the weak hash lookups, which suggests that Dsync is almost four orders of magnitude lower than WebR2Sync+, indicating that the CDC approach can greatly reduce the weak hash lookups.

The solid lines in Figure 13 show the results of strong-hash comparisons. The number of strong hash comparison operations represents the number of matched chunks. The more matched chunks we find, the more strong-hash comparisons have to be executed. When file modifications are minor, Dsync compares fewer strong hash fingerprints because of our merging strategy and WebR2Sync+'s ability to find more chunks than Dsync. However, when modification size is 1MB, it's hard to find unmodified chunks for WebR2Sync+, since the distance between two modifications is almost 8KB while Dsync may find the matched chunks smaller than 8KB (the chunk sizes are variable by CDC), as has been stated in Section V-A. Therefore, Dsync has more matched chunks at 1MB modification, which thus needs to compare more strong hashes. Even if WebR2Sync+ may compare fewer strong hash,



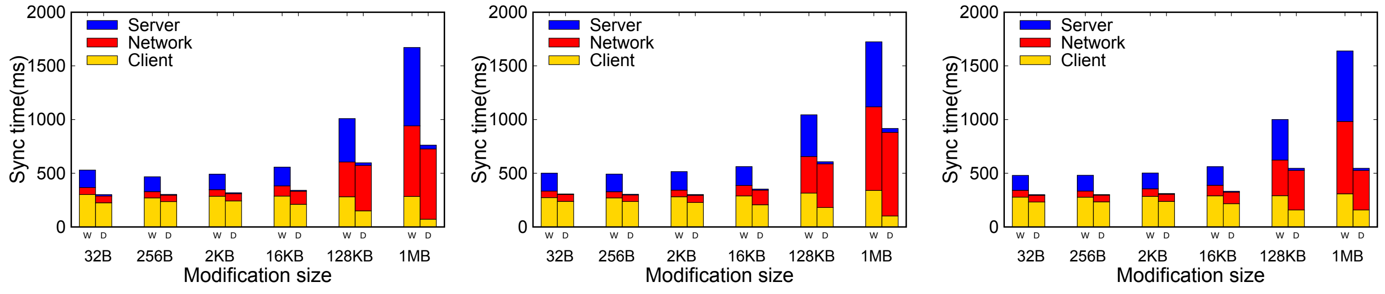
(a) Cut. (b) Insert. (c) Inverse.

Fig. 12. The sync time spent on client as a function of modification size in WebR2sync+ and Dsync.



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

Fig. 13. Numbers of hash-comparing operations of Dsync and WebR2Sync, including both strong and weak hashes.



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

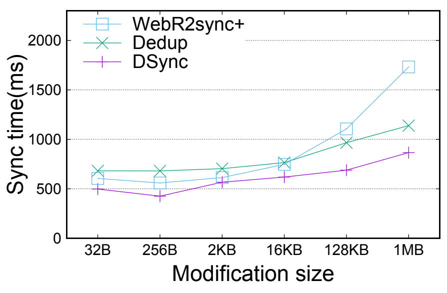
Fig. 14. Sync time breakdown of WebR2sync+ and Dsync as a function of modification sizes.

Dsync calculates fewer hashes and has lower CPU overhead because the strong hash calculation for some chunks is not necessary for the CDC approach (see Section V-C).

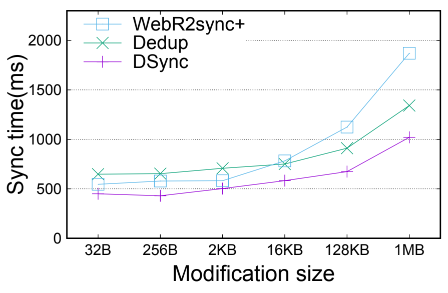
Then we evaluate the breakdown of Dsync's sync time comparing with WebR2sync+, as shown in Figure 14. Compared with WebR2sync+, the time consumed by both the server and the client has been greatly reduced and the time spent on the client has been reduced as well, especially when the size of modifications is larger than 128KB. Note that the necessary time for Delta Bytes transmission (i.e., network time) cannot be reduced by both Dsync and WebR2sync+, we target at providing a lightweight delta sync solution with low compute overhead in this paper.

Figure 15 shows the sync time as a function of the modification size of the three sync approaches. Overall, while Dsync consistently outperforms the other two sync approaches, all three approaches exhibit identical performance trends as the modification size increases. When the modification is light (e.g., less than 2KB), the advantage of Dsync is obvious. As the modification size increases, Dsync's advantage becomes pronounced, outperforming WebR2Sync+ by a factor of two. The dedup-based approach is slower than WebR2Sync+ when modification size is small. Nevertheless, if more modifications are made, Dedup outperforms WebR2Sync+ since it can more effectively handle low-matching scenarios via CDC technique (see Figure 4(b)).

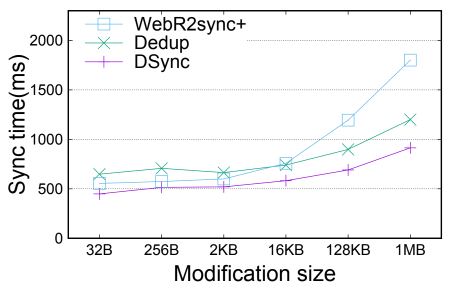
Next, we evaluate the scalability of these approaches in terms of the ability to support multiple clients simultaneously by the measure of CPU utility, with results shown in Figure 16. To support multiple clients synchronizing files simultaneously, we run the client code on nodejs on PC instead of web browsers. In this evaluation, the file modifications are executed



(a) Cut.

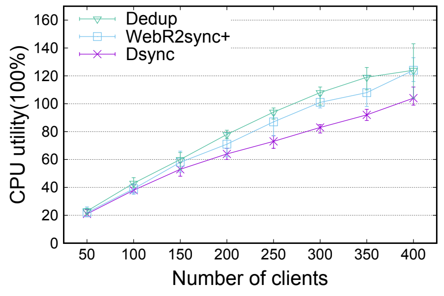


(b) Insert.

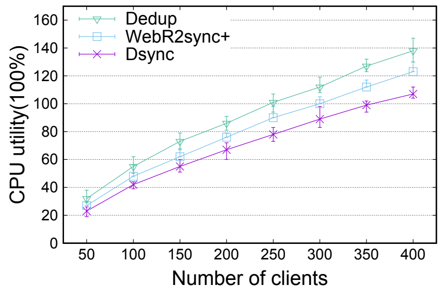


(c) Inverse.

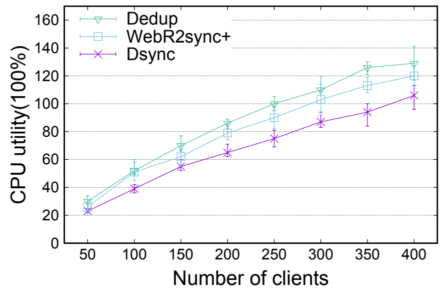
Fig. 15. Sync time of the three delta sync approaches as a function of the modification size.



(a) Cut.



(b) Insert.



(c) Inverse.

Fig. 16. Multiple clients supported by the three delta sync approaches on a single VM server instance.

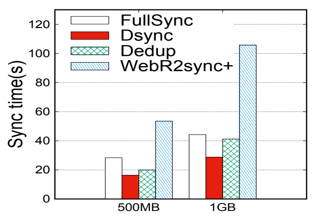
TABLE III
SYNC PERFORMANCE OF THE THREE APPROACHES ON BOTH WINDOWS/ANDROID CLIENT ON FOUR REAL-WORLD DATASETS. NOTE THAT WINDOWS/ANDROID PLATFORMS HAVE THE SAME SYNC TRAFFIC IN THE LAST THREE COLUMNS SINCE THEY ONLY DIFFER IN THE COMPUTE CAPACITY.

	Sync Time(Seconds) (Windows/Android)			*Sync Traffic(MB) (Windows/Android)*		
Dataset	*Dsync*	*WebR2sync+*	*Dedup*	*Dsync*	*WebR2sync+*	*Dedup*
Pictures	17.49/ 53.45	20.27/ 83.90	20.61/ 59.74	94.3	105.3	104.4
PPT	11.14/ 44.69	20.81/ 84.22	15.22/ 56.39	162.2	162.4	164.0
Mail	27.19/117.40	90.46/271.74	48.67/160.06	635.4	638.0	637.6
GLib	41.98/157.46	75.05/299.74	52.03/186.88	497.8	455.4	532.7

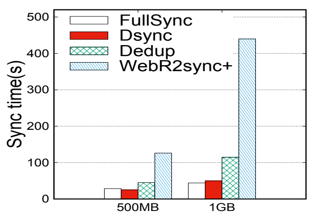
every 10 seconds on each client, and then the clients send the delta sync request. To get the statistical error bounds, we run the experiment 5 times for each point on the graph. Note that in this evaluation, the lower the CPU utility an approach has, the more scalable the approach is. The results indicate that Dsync has the lowest CPU utility consistently among the three sync approaches, making it the most scalable approach. In other words, while Dsync is shown to support delta sync of about 370 clients, WebR2Sync+ and Dedup can only support about 250-300 clients concurrently.

Finally, we run our tests on four real-world datasets. In this evaluation, in addition to the PC client (Windows), we also test the three delta sync approaches on the Web browser of the Mobile phone client (Android). Table III shows the results of the Dsync, WebR2sync+, and Dedup schemes. First of all, the sync services on the mobile phone client are running much more slowly than on the PC client (3×-4× slower), which is because a PC that is installed with Windows has a much more powerful CPU than an Android mobile phone device. Second, the sync time comparison among the three approaches shows the consistent advantages of Dsync over WebR2sync+ and Dedup on both PC and mobile phone clients (about 1.5-3.3× faster). Third, the sync traffic volumes of the three approaches are almost the same. Note that the sync traffic for the Mail dataset is almost the same as that of the original file because the size of a single mail and the modifications are too small and the fully dispersed modifications lead to a very low probability of chunk matching. Therefore, the CDC-based approach is indeed more effective than `rsync`-based approach and Dsync consistently and notably outperforms Dedup and WebR2sync+.

In summary, Dsync runs 1.5×-3.3× faster than the state-of-the-art `rsync` based WebR2Sync+ and Dedue-based approach. The processing time of Dsync on both the client and server sides has been greatly reduced compared with the latest `rsync` based WebR2Sync+.



(a) Modification ratio of 1%.



(b) Modification ratio of 30%.

Fig. 17. Sync time of the four sync approaches for the large files in the high-bandwidth environment.

E. High Bandwidth and Large Files

In this subsection, we evaluate Dsync performance on the large files under the Gigabit network environment, as shown in Figure 17. Here we add the FullSync approach (without calculating the delta data) for better evaluating the other three delta sync approaches. As mentioned earlier, the chunk-matching process is very time-consuming in the `rsync` based approaches, the results shown in Figure 17 suggest that `rsync`-based WebR2sync+ spends 5×-8× more sync time comparing with Dsync, and even much slower than the FullSync approach. This is because WebR2sync+ spends too much time on the process of byte-by-byte chunk-matching on large files while the time spent on the network has been reduced by using a high-bandwidth environment. Meanwhile, Dsync runs much faster than FullSync since it not only reduces the transmission of redundant data but also simplifies the chunk-matching process, which well caters for the delta sync demand of the future cloud storage services with high bandwidth and large files. Note that our client is on the Web browser and server is on a VM server. Thus only about 30% of the total Gigabit bandwidth is consumed in this evaluation.

VI. Conclusion

The traditional `rsync`-based approaches will introduce the heavy computation overhead in the chunk matching process for high-bandwidth cloud storage services. At the same time, CDC-based deduplication simplifies the chunks matching process but brings new challenges of the additional computation overhead and low redundancy detection ratio. In this paper, we propose Dsync, a deduplication-inspired lightweight delta synchronization approach for cloud storage services, to address the above challenges facing combining CDC technique with the traditional `rsync`-based approaches. The critical contributions of Dsync are to develop a fast, weak hash called FastFp by piggybacking on hashes generated from the chunking process of FastCDC and redesigning the delta sync protocol by exploiting deduplication locality and weak/strong hash properties, which makes CDC simplify the delta sync process for cloud storage services, especially for the large file synchronization and high-bandwidth environment. Evaluation results, driven by both benchmark and real-world datasets, demonstrate that our solution Dsync performs 2×-8× faster and scales to 30%-50% more concurrent clients than the state-of-the-art `rsync`-based WebR2sync+ and the latest deduplication-based approach.

VII. Acknowledgments

We are grateful to our shepherd Philip Shilane and the anonymous reviewers for their insightful comments on this work. This research was partly supported by NSFC No. 61972441, No. 61822205, No. 61902093, the Open Project Program of Wuhan National Laboratory for Optoelectronics No. 2018WNLOKF008, the Shenzhen Science and Technology Program under Grant No. JCYJ20190806143405318, Key Technology Program of Shenzhen, China (No. JSGG20170823152809704), Key R&D Program for Guangdong Province under Grant No.2019B010136001, Shandong Provincial Natural Science Foundation (No. ZR2019QF014), and the US National Science Foundation under Grant CCF-1704504 and CCF-1629625. Wen Xia is the Corresponding author (xiawen@hit.edu.cn).

References

[1] Dropbox. https://www.dropbox.com/.
[2] Enron mail dataset. https://bit.ly/2XSUbu2/.
[3] GoogleDrive. https://www.google.com/drive/.
[4] iCloud. https://www.icloud.com/.
[5] rsync. https://rsync.samba.org/.
[6] Seafile: Enterprise file sync and share platform with high reliability and performance. https://www.seafile.com/en/home.
[7] V. Christlein, C. Riess, J. Jordan, C. Riess, and E. Angelopoulou. An evaluation of popular copy-move forgery detection approaches. *IEEE Transactions on Information Forensics and Security*, 7(6):1841–1854, 2012.
[8] Yong Cui, Zeqi Lai, Xin Wang, and Ningwei Dai. Quicksync: Improving synchronization efficiency for mobile cloud storage services. *IEEE Transactions on Mobile Computing*, 16(12):3513–3526, 2017.
[9] Teodosiu Dan, Nikolaj Bjorner, Joe Porkka, Mark Manasse, and Y. Gurevich. Optimizing file replication over limited-bandwidth networks using remote differential compression. *Microsoft Research*, 2006.
[10] Sebastian Deorowicz. Silesia Corpus. https://bit.ly/2xN3hgZ.
[11] Sebastian Deorowicz. *Universal lossless data compression algorithms*. PhD thesis, Silesian University of Technology, 2003.
[12] Idilio Drago, Enrico Bocchi, Marco Mellia, Herman Slatman, and Aiko Pras. Benchmarking personal cloud storage. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 205–212, Barcelona, Spain, October 2013. ACM.
[13] Idilio Drago, Marco Mellia, Maurizio M Munafo, Anna Sperotto, Ramin Sadre, and Aiko Pras. Inside dropbox: understanding personal cloud storage services. In *Proceedings of the 2012 Internet Measurement Conference*, pages 481–494, Boston, MA, USA, November 2012. ACM.
[14] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C Snoeren, George Varghese, Geoffrey M Voelker, and Amin Vahdat. Difference engine: Harnessing memory redundancy in virtual machines. *Communications of the ACM*, 53(10):85–93, 2010.
[15] Barron C Housel and David B Lindquist. Webexpress: A system for optimizing web browsing in a wireless environment. In *Proceedings of the 2nd annual international conference on Mobile computing and networking*, pages 108–116, Rye, New York, USA, November 1996. ACM.
[16] James Wayne Hunt and M Douglas MacIlroy. *An algorithm for differential file comparison*. Bell Laboratories Murray Hill, 1976.
[17] Richard M Karp and Michael O Rabin. Efficient randomized pattern-matching algorithms. *Ibm Journal of Research and Development*, 31(2):249–260, 1987.
[18] David G Korn and Kiem-Phong Vo. Engineering a differencing and compression data format. In *Proceedings of the 2002 USENIX Annual Technical Conference*, pages 219–228, Monterey, California, USA, June 2002. USENIX.

[19] Zhenhua Li, Cheng Jin, Tianyin Xu, Christo Wilson, Yao Liu, Linsong Cheng, Yunhao Liu, Yafei Dai, and Zhi-Li Zhang. Towards network-level efficiency for cloud storage services. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 115–128, Vancouver, BC, Canada, November 2014. ACM.

[20] Zhenhua Li, Christo Wilson, Zhefu Jiang, Liu Yao, Ben Y. Zhao, Jin Cheng, Zhi Li Zhang, and Yafei Dai. Efficient batched synchronization in dropbox-like cloud storage services. *Lecture Notes in Computer Science*, 8275:307–327, 2013.

[21] Josh MacDonald. File system support for delta compression. Master's thesis, Department of Electrical Engineering and Computer Science, University of California at Berkeley, Berkeley, CA, USA, 2000.

[22] Irfan Mohiuddin, Ahmad Almogren, Mohammed Al Qurishi, Mohammad Mehedi Hassan, Iehab Al Rassan, and Giancarlo Fortino. Secure distributed adaptive bin packing algorithm for cloud storage. *Future Generation Computer Systems*, 90:307–316, 2019.

[23] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, pages 174–187, Banff, Alberta, Canada, October 2001. ACM.

[24] Vasily Tarasov, Amar Mudrankit, Will Buik, Philip Shilane, Geoff Kuenning, and Erez Zadok. Generating realistic datasets for deduplication analysis. In *Proceedings of the 2012 USENIX Annual Technical Conference*, pages 261–272, Boston, MA, USA, June 2012. USENIX.

[25] Dimitre Trendafilov, Nasir Memon, and Torsten Suel. zdelta: An efficient delta compression tool. http://cis.poly.edu/tr/tr-cis-2002-02.pdf, 2002.

[26] Andrew Tridgell and Paul Mackerras. The rsync algorithm. https://rsync.samba.org/tech_report/tech_report.html, 1996.

[27] Andrew Tridgell, Paul Mackerras Richard Brent, and Brendan McKay. *Efficient Algorithms for Sorting and Synchronization*. Phd thesis, The Australian National University, 1999.

[28] Suzhen Wu, Longquan Liu, Hong Jiang, Hao Che, and Bo Mao. PandaSync : Network and Workload aware Hybrid Cloud Sync Optimization. In *Proceedings of the 39th International Conference on Distributed Computing Systems*, pages 1–11, Dallas, Texas, USA, July 2019. IEEE.

[29] Wen Xia, Hong Jiang, Dan Feng, Fred Douglis, Philip Shilane, Yu Hua, Min Fu, Yucheng Zhang, and Yukun Zhou. A comprehensive study of the past, present, and future of data deduplication. *Proceedings of the IEEE*, 104(9):1681–1710, Sep. 2016.

[30] Wen Xia, Hong Jiang, Dan Feng, Lei Tian, Min Fu, and Yukun Zhou. Ddelta: A deduplication-inspired fast delta compression approach. *Performance Evaluation*, 79:258–272, 2014.

[31] Wen Xia, Yukun Zhou, Hong Jiang, Dan Feng, Yu Hua, Yuchong Hu, Qing Liu, and Yucheng Zhang. Fastcdc: A fast and efficient content-defined chunking approach for data deduplication. In *2016 USENIX Annual Technical Conference*, pages 101–114, Denver, CO, June 2016. USENIX Association.

[32] He Xiao, Zhenhua Li, Ennan Zhai, Tianyin Xu, Yang Li, Yunhao Liu, Quanlu Zhang, and Yao Liu. Towards web-based delta synchronization for cloud storage services. In *Proceedings of 16th USENIX Conference on File and Storage Technologies*, pages 155–168, Oakland, CA, USA, February 2018. USENIX Association.

[33] Takeshi Yoshino. Compression Extensions for WebSocket. RFC 7692, December 2015.

[34] Quanlu Zhang, Zhenhua Li, Zhi Yang, Shenglong Li, Shouyang Li, Yangze Guo, and Yafei Dai. Deltacfs: Boosting delta sync for cloud storage services by learning from NFS. In *Proceedings of 37th IEEE International Conference on Distributed Computing Systems*, pages 264–275, Atlanta, GA, USA, June 2017. IEEE.

[35] Benjamin Zhu, Kai Li, and R Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of USENIX Conference on File and Storage Technologies*, pages 269–282, San Jose, California, USA, February 2008. USENIX Association.