NetSync: A Network Adaptive and Deduplication-Inspired Delta Synchronization Approach for Cloud Storage Services

Wen Xia, *Member, IEEE*, Can Wei, Zhenhua Li, *Member, IEEE*, Xuan Wang, and Xiangyu Zou

Abstract—Delta sync (synchronization) 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. However, it is difficult for this process to cater to the forthcoming high-bandwidth cloud storage services which require lightweight delta sync that can well support large files. Moreover, `rsync` employs invariant chunking and compression methods during the sync process, making it unable to cater to services from various network environments which require the sync approach to perform well under different network conditions. Inspired by the Content-Defined Chunking (CDC) technique used in data deduplication, we propose NetSync, a network adaptive and CDC-based lightweight delta sync approach with less computing and protocol (metadata) overheads than the state-of-the-art delta sync approaches. Besides, NetSync can choose appropriate compressing and chunking strategies for different network conditions. The key idea of NetSync is (1) to simplify the process of chunk matching by proposing a fast weak hash called FastFP that is piggybacked on the rolling hashes from CDC, and redesigning the delta sync protocol by exploiting deduplication locality and weak/strong hash properties; (2) to minimize the sync time by adaptively choosing chunking parameters and compression methods according to the current network conditions. Our evaluation results driven by both benchmark and real-world datasets suggest NetSync performs $2\times$–$10\times$ faster and supports 30%–80% more clients than the state-of-the-art `rsync`-based WebR2sync+ and deduplication-based approach.

Index Terms—`rsync`, content-defined chunking, network adaptive, cloud storage

1 Introduction

The promises of making data accessible anywhere and anytime have made personal cloud storage services increasingly popular, including Dropbox [1], GoogleDrive [2], and iCloud [3]. The thriving cloud storage services impose challenges on both clients and servers with increasing network and computing overheads, encouraging service providers to propose more effective solutions. DropBox, for example, has adopted delta sync to reduce the data traffic between client and server [4], [5]. In spite of this, the sync traffic volume of DropBox still takes up to 4% of the total internet traffic due to the frequent interactions between clients and servers for calculating the delta (modified) data, according to a study [6] on several campus border routers. Therefore, reducing the traffic and computing overheads, and fully leveraging the current network bandwidth in synchronization remain important challenges to be addressed.

As a classic model to reduce the traffic of synchronizing files between two hosts, `rsync` can effectively reduce the traffic volume. Generally speaking, `rsync` synchronizes file `f'` (a modified version of file `f`) from client to server in three phases as follows: ① Client sends a sync request to server, and server splits the server file `f` into fix-sized chunks, which is followed by server calculating and sending fingerprints (called Checksum List) of those chunks. ② Client uses a window sliding on client file `f'` byte-by-byte, to match possible duplicate chunks within the Checksum List from server. ③ After finishing the byte-by-byte matching, client obtains the mismatched chunks, named Delta Bytes, and sends them to server, after which server reconstructs client file `f'` according to Delta Bytes and server file `f`.

Several recent studies, including DeltaCFS [7], PandaSync [8], and WebR2sync+ [9], have improved `rsync` in various aspects. However, the chunk-matching process using a byte-by-byte sliding window which can be very time-consuming especially when the files are increasingly large in high-bandwidth cloud storage systems [10], remains unchanged in these `rsync`-based approaches. For example, according to our study and observation, WebR2sync+ [9] spends about $10\times$ longer time on the byte-by-byte chunk-

- *Wen Xia is with the Harbin Institute of Technology, Shenzhen, Guangdong 518055, China, and also with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 518109, China. E-mail: xiawen@hit.edu.cn.*
- *Can Wei, Xuan Wang, and Xiangyu Zou are with the Department of Computer Science and Technology, Harbin Institute of Technology, Shenzhen, Guangdong 518055, China. E-mail: weican16hit@gmail.com, wangxuan@hit.edu.cn, xiangyu.zou@hotmail.com.*
- *Zhenhua Li is with Tsinghua University, Beijing 100084, China. E-mail: lizhenhua1983@gmail.com.*

Manuscript received 17 Aug. 2021; revised 26 Nov. 2021; accepted 9 Jan. 2022. Date of publication 25 Jan. 2022; date of current version 4 Apr. 2022.
This work was supported in part by NSFC under Grant 61972441, in part by Shenzhen Science and Technology Program under Grants JCYJ2019080 6143405318, JCYJ20200109113427092, and GXWD20201230155427003-20200821172511002, in part by Guangdong Basic and Applied Basic Research Foundation under Grant 2021A1515012634, and in part by the State Key Laboratory of Computer Architecture (ICT, CAS) under Grant CARCHA202006. (Corresponding author: Wen Xia.)
Recommended for acceptance by J. Wang.
Digital Object Identifier no. 10.1109/TPDS.2022.3145025

matching process than on the data transferring process over the Gigabit network when synchronizing a large 1GB file.

Meanwhile, synchronization should not only focus on redundancy elimination but also need to fully utilize the network bandwidth. Moreover, the imbalance of redundancy elimination and network utilization is significant as the network bandwidth varies. Commonly, the sync services choose finer chunking granularity and may use compressors to eliminate more redundancy under limited networks; however, this strategy may be sub-optimal when the bandwidth is sufficiently high. According to our observation, Dsync[11], a recently proposed sync service, could be slowed down up to 5 times if the best configuration under the 10 Mbps network is applied in a 100 Mbps network.

The advance in network infrastructure makes it harder for the current delta sync to cater to the demand of cloud storage, given the inevitable challenges of an inherently wider range of 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, while some devices still work under low-network bandwidth due to unstable connection or limited power consumption. Accordingly, the cloud-hosted files have been growing in both quantity and (single-file) size. Consequently, we have to rethink and innovate the current delta sync design to catch up with this trend and satisfy the user experience.

This paper mainly focuses on the two most important metrics for sync approaches: sync performance [8] and flexibility [9]. (1) Sync performance (i.e., sync latency or speed), which directly affects the system consistency and throughput, is a metric concerned by most users [8]. On the one hand, higher performance could provide a better service experience for users and avoid inconsistencies among different sync devices [12], [13]. On the other hand, it can also help reduce resources cost (e.g., computation, network, etc.) for service providers. (2) Sync flexibility, i.e., the ability to provide optimal sync performance for users from different resources, requires flexible designs to handle these different situations. To improve these two metrics, in what follows, we will address the above two challenges (i.e., the time-consuming byte-by-byte chunk matching and the poor network utilization), respectively.

For the first problem, recently, data deduplication, a chunk-level data reduction approach, has been increasingly attractive in the design of storage systems [14]. Instead of using a byte-wise sliding window, data deduplication splits files into independent chunks, using a technique named Content-Defined Chunking (CDC), and then detects duplicate chunks according to their fingerprints [15], [16]. FastCDC [16] is a fast and efficient CDC technique widely used in data deduplication (a detailed description of FastCDC is in Section 3, Fig. 3). Specifically, in CDC process, a small window (e.g., size of 64 bytes) slides on the file contents, and a chunk boundary is declared once the hash value of the window (i.e., the file contents) meets a predefined condition. Therefore, the chunk boundaries are determined by the file contents instead of locations in the file, which adequately addresses the "boundary-shift" problem (detailed in Section 3) and thus eliminates more redundancy. Consequently, we believe that the traditional time-consuming byte-wise comparison in `rsync` could be discarded by introducing the CDC technique.

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 computing overhead due to the rolling-hash-based chunking and low redundancy detection ratio due to the coarse-grained chunk matching after CDC.

For the second problem, to achieve better flexibility and maximize network utilization, it is reasonable to change the sync configurations accordingly, rather than finding constant configurations for any network conditions. Concretely, it imposes two challenges: ① monitoring the network condition without affecting the synchronization, ② choosing the best chunking granularity and compression method to minimize the overall sync time. Besides, our implementation is based on Web browser (the most pervasive and OS-independent access) for better platform adaptability.

To this end, we propose NetSync, a lightweight, portable, network adaptive, and CDC-based delta sync approach for the current cloud storage services, with less computing and protocol (metadata) overheads. Generally, to address the challenges mentioned above, we make the following four critical contributions in this paper:

- Introducing Content-Defined Chunking technique (i.e., FastCDC) to `rsync` for matching more identical content and reducing network traffic. Meanwhile, we exploit the chunking process to generate weak hash values for chunks, called FastFP, and replace Adler32 used in `rsync`, which offsets additional computation overhead introduced by Content-Defined Chunking. Evaluation results suggest that FastFP is much faster than Adler32 and achieves a comparable low hash collision ratio.
- Redesigning the communication protocol to reduce both computing overhead and network traffic: ① first, check the weak hash, then compute and match the the weak-hash-matched chunks' strong hash, to reduce most of the unnecessary computing of strong hash on mismatched chunks; ② merge the consecutive weak-hash-matched chunks into a single large chunk to reduce the size of Match Token (metadata overhead) for network interactions in NetSync.
- Minimizing the sync time by the network adaptive design: ① network adaptive compression: choosing the appropriate compressor for transmission according to the current network condition obtained by the net-aware module; ② network adaptive chunking: changing the chunk granularity of the CDC process according to the current network condition to fully utilize the potential of CDC and current network resources, thus minimizing the sync time.
- Comprehensive evaluations driven by real-world and benchmark datasets illustrate that NetSync performs $2\times$–$10\times$ faster and supports 30%–80% more clients than the state-of-the-art `rsync`-based WebR2sync+ [9].

The rest of this paper is organized as follows. Section 2 introduces the background and related work. Section 3 discusses the deficiency of the state-of-the-art WebR2sync+

TABLE 1
Data Sync Techniques

Source	Full Sync[1]	Delta Sync	
		Local Buffer	*Chunking Methods*[2]
DropBox (W/A) [25][3]	×	✓	FSC
Seafile (W) [26]	×	✓	CDC
Seafile (A) [26]	✓	×	×
GoogleDrive (W/A) [25]	✓	×	×
OneDrive (W/A) [25]	✓	×	×
rsync [27]	×	×	FSC
DeltaCFS [7]	×	×	FSC
PandaSync [8]	$\surd^*$	×	FSC
WebSync [9]	×	×	FSC
QuickSync [25]	×	✓	CDC
LBFS [15]	×	×	CDC
UDS [28]	×	×	FSC
NetSync	×	×	CDC

[1]. *✓: full sync, ×: delta sync, and $\surd^*$: selective full sync.*
[2]. *FSC: Fix-Sized Chunking, CDC: Content-Defined Chunking.*
[3]. *W: windows client, A: android client.*

approach, the potential of the CDC-based approach, and the mismatch between the sync policy and the network condition. Section 4 describes the design and implementation details of NetSync. Section 5 presents the evaluation results of NetSync, including comparisons with the latest rsync-based WebR2sync+, deduplication-based solution, and Dsync [11] (a preliminary version of NetSync in which the network adaptive modules are not included). Finally, Section 6 concludes this paper.

2 RELATED WORK

Generally, there are two approaches to sync a modified file from client to server in cloud storage: full sync and delta sync, respectively. The former, which transfers the whole file to server, is suitable for synchronizing small files [8]. In contrast, the latter, which only transfers modified data of a file (i.e., the delta) to server, can minimize the network traffic and is suitable for large file synchronization.

Delta sync has a great advantage when the files are frequently modified, e.g., files with multiple versions or consecutive edits. A recent study [6] on several campus border routers indicates that the traffic volume of DropBox, which uses delta sync, accounts for 4% of the total internet 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 sync. Actually, delta sync approaches have been widely studied by many publications including Unix diff [17], Vcdiff [18], WebExpress [19], rsync [20], Content-Defined Chunking (CDC) [21], [15], and delta encoding algorithms [22], [23], [24]. Representative sync techniques supported by the state-of-the-art cloud storage services are summarized in Table 1 and discussed below.

a) *Sync Tools Supported by Industry*. Commercial cloud storage services include Dropbox, GoogleDrive, OneDrive, Seafile, etc. [29]. The PC clients of both Dropbox and Seafile support delta sync, where DropBox's sync uses 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 research proposes their sync tools to support delta sync for better performance (to save network bandwidth and accelerate network transmission). First proposed by Tridgell, rsync is a well-known sync protocol to effectively synchronize a file between two hosts over a network connection with limited bandwidth [20], [30]. This approach is adopted as the standard sync protocol in GNU/Linux [27].

Recent studies, such as DeltaCFS [7], PandaSync [8], and WebSync [9], make innovative improvements on top of rsync. Specifically, DeltaCFS directly records the minor modifications to reduce computing and network overheads due to frequent synchronization. PandaSync strikes a trade-off 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, while delta sync can effectively reduce network traffic and sync time. WebR2sync+ [9] makes the first attempt to implement delta sync over Web browsers, a dominant form of Internet access. To avoid the influence of the poor performance of browsers when executing computing-intensive byte-wise comparison of hash values, WebR2sync+ shifts the comparison to server and replaces the cumbersome MD5 hash function with a lightweight hash function called SipHash. Due to the boundary-shift problem, QuickSync [29] employs the CDC technique rather than rsync. 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 [15] splits the file into chunks using the CDC technique, calculates and compares their SHA-1 fingerprints to detect duplicate chunks. It finally transmits the non-duplicate chunks from client to server. UDS (Update-batch delayed sync) [28] uses a batched sync strategy to avoid bandwidth overuse due to frequent modifications based on Dropbox.

The notion is that, among the services mentioned above, only Pandasync[8] and QuickSync[29] support network adaptive optimization. Pandasync[8] focuses on choosing different sync strategies for different network conditions. Specifically, it chooses to use full sync or delta sync according to the current network bandwidth to get a better sync performance. QuickSync estimates bandwidth via network congestion window size and chooses a proper chunk granularity to improve sync efficiency.

In summary, as shown in Table 1, delta sync with local buffer requires client to have extra storage and undertake computing overheads while full sync is simple and computationally friendly but not bandwidth-efficient. Despite the advantages of rsync-based approaches, the exceptionally high computing overhead of rsync protocol due to byte-wise comparison and hash calculation severely limits its applicability in resource-constrained client systems, especially for synchronizing large files. Thus, this paper focuses on providing a lightweight, portable delta sync approach for resource-constrained client systems via Web browsers on Mobile phones, IoT devices, etc. Thus, it is quite inconvenient for such a resource-constrained Web browser to maintain the metadata buffer of client files for delta sync.

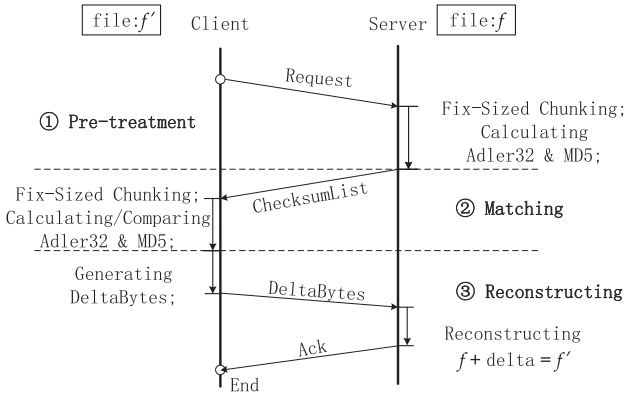


Fig. 1. Workflow chart of rsync.

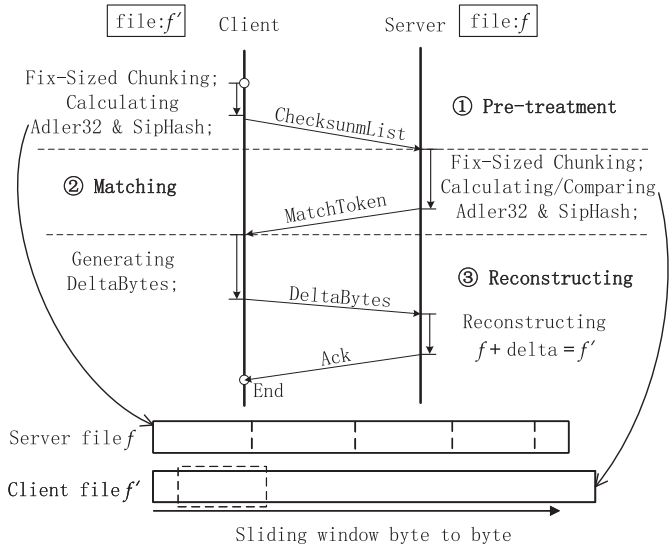


Fig. 2. Workflow chart of WebR2sync+.

Furthermore, inspired by previous efforts on network adaptive design, our delta sync service can dynamically adapt to both limited and sufficiently high bandwidth conditions. On the other hand, deduplication-based sync [15] can avoid byte-wise chunk matching in rsync. Note that our work is substantially different from all the previous work (see Table 1) in that it is the first attempt to combine the CDC technique with the traditional rsync model to effectively synchronize data when client is resource-constrained, i.e., there is no local buffer and sufficient computing capacity for executing rsync-like delta sync protocols on client.

3 CHALLENGES AND MOTIVATIONS

To better understand the strengths and weaknesses of rsync [30], we first illustrate how it works with the help of Fig. 1. It has a three-stage workflow:

- In stage 1, when a client needs to synchronize a file f', it first sends a request with the name of file f' to server. After receiving the client request, server starts to split the server file f, which has the same file name with client file f′, into Fix-Sized chunks (i.e., using Fixed-Sized Chunking). Meanwhile, it calculates weak but fast rolling hash values (Adler32) and strong but slow hash values (MD5) of the chunks as weak and strong fingerprints. Both weak and strong fingerprints are included in Checksum List and sent to client.
- In stage 2, upon receiving Checksum List from server, client slides a fix-sized window on file f'as server does, to generate chunks and their weak hash Adler32 to match with the weak fingerprints (of chunks in file f) in Checksum List. Note that: ① if the current chunk under the sliding window does not match any Adler32 hashes in Checksum List, the window will slide further byte by byte until a match is found. ② If a weak-hash-matched chunk is found, its strong hash MD5 will be calculated and compared with the corresponding chunk's MD5 in Checksum List to confirm if it is a duplicate chunk. Once the MD5 hash values of these two chunks are matched (i.e., the two chunks are duplicates), the sliding window will slide with the size of the fixed window. If the MD5 is mismatched, it will continue the byte-to-byte sliding. This stage takes a rather long time due to the byte-by-byte sliding window-based chunk matching.
- In stage 3, client generates Delta Bytes according to stage 2, which includes the mismatched chunks and their metadata, and sends them to server where file f' will be reconstructed by file f + Delta Bytes.

WebR2sync+ [14] is designed based on rsync and implemented on Web browser for better platform adaptability. WebR2sync+ shifts the chunk-matching process from client to server to adapt to the low computing capacity of Web client. Its workflow is also three-stage, as shown in Fig. 2.

- In Stage 1, in contrast to rsync, the pre-treatment (i.e., the chunking and weak hash computing process) is moved from server to client, and the strong hash MD5 is replaced by a faster hash SipHash.
- In Stage 2, the chunk-matching process is moved from client to server, and the strong hash MD5 is changed into SipHash. The operation in this stage is almost the same as rsync. After that, Match Token that indicates which chunks are matched will be sent to client.
- In Stage 3, client generates Delta Bytes and sends them to server, and file f' is reconstructed according to file f and Delta Bytes, which is the same as rsync.

The Weak Point of Fix-Sized Chunking (FSC for Short). Although WebR2sync+ is improved over rsync, it does not fundamentally solve the challenges faced by rsync. More specifically, WebR2sync+ adopts FSC as its chunking method as the rsync does. However, FSC can not handle the boundary-shift problem: When the file is modified by inserting or deleting, the segmentation points may be shifted backward or forward, resulting in the mismatch of the following chunks. Both WebR2sync+ and rsync adopt the verbatim comparing algorithm to solve this problem: slide a matching window byte-by-byte until finding a matched chunk. However, this algorithm may lead to high annoying computing overhead, and in the worst case, the matching window needs to slide from the beginning to the end of the file byte-by-byte [22]. By contrast, Content-Defined Chunking (CDC) technique used in data deduplication provides an

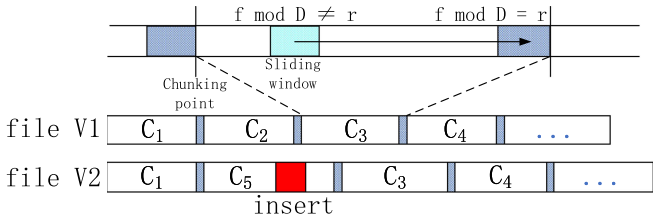


Fig. 3. The CDC technique for chunk-level data deduplication. It applies a sliding window on the content of files and computes a hash value of the window. A chunk cut-point (chunking point) is declared if the hash value "fp" of the sliding window satisfies a predefined condition.

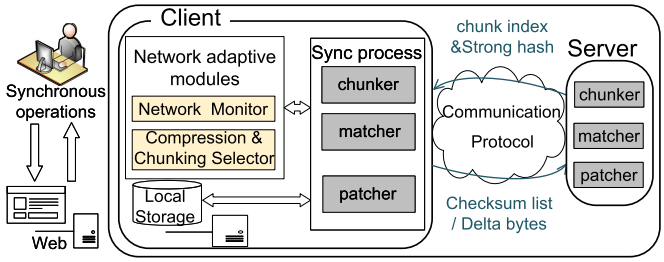


Fig. 4. NetSync system overview.

opportunity to avoid this heavy calculation of overlapping chunks.

Introducing Content-Defined Chunking (CDC) to the `rsync`-Based Process. CDC is proposed to solve the boundary-shift problem. As shown in Fig. 3, CDC applies a sliding window on the content of files and computes a hash value (e.g., Rabin [31], Gear [24]) of the window. A chunk cut-point is declared if the hash value satisfies some predefined condition. That is to say, the chunks are divided according to file contents rather than the boundary index. This enables CDC to solve the boundary-shift problem. As shown in Fig. 3, although the boundary indexes of chunks C3 and C4 in file V2 have been shifted due to file modification, C3 and C4 still can be identified for data deduplication (with file V1) by the CDC technique.

However, the CDC process introduces additional computing overhead for delta sync, i.e., computing rolling hashes for chunking. Further, CDC may lead to more network traffic than `rsync`-based approaches since it may fail to eliminate redundancy among similar chunks (i.e., the very similar chunks C2 and C5 in Fig. 3).

Nevertheless, the CDC technique remains attractive because it greatly simplifies the chunk fingerprinting and indexing process and generates much fewer chunks for fingerprinting and indexing than the traditional `rsync`-based approaches, especially for large files. Therefore, to fully capitalize on the strengths while avoiding weaknesses of the CDC technique, we try to propose a novel and efficient CDC-based sync approach to significantly simplify the delta sync process, as detailed later in Section 4.

The Mismatch Between the Sync Policy and Network Condition. Delta sync eliminates duplicate data chunks and compresses unique chunks to reduce the data transmission time, but it may be fruitless when the network bandwidth is sufficiently high. Commonly, more compacted data transmission incurs higher computing overhead, and inappropriate chunking or compressing could significantly bottleneck the overall sync performance [32]. More specifically, finer chunking and compressor with a higher compression ratio should be applied under a limited network. A coarser chunking granularity and lightweight compressor should be adopted when the network bandwidth is sufficiently high. According to our observation, compared with the optimal configuration, the chunking/compressing module performance that follows the configuration in Dsync is 3.6×/5× slower, and a carefully chosen invariant chunking/compression policy is still 2.4×/2.8× slower in certain bandwidths.

To strike a perfect balance between the computing and transmission overhead for varying network conditions, we develop a network prober that accurately probes the current network bandwidth and a chunking/compression policy selector that empirically finds the optimal configuration and minimizes the overall sync latency.

4 Design and Implementation

4.1 Overview

Architecture Overview. As shown in Fig. 4, NetSync comprises a series of modules: the chunker (i.e., chunking module), chunk matcher, file patcher, network monitor, and network adaptive parameter selector.

- `Chunker`. In NetSync, the client and the server files will be first divided into several chunks by Content-Defined Chunking (i.e., FastCDC) for future duplicate detection.
- `Weak/Strong hashing`. In NetSync, this module is embedded in chunker and matcher. In this module, two-level hash values (i.e., weak/strong hash) are calculated for checking duplicate chunks. The weak hash, which is fast with low computing overhead, is used to 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 weak hash mismatches, the chunk is marked as unique, and thus strong hash calculation will be skipped. Note that the original strong hash SipHash used in WebR2sync+ [9] is substituted with SHA-1 out of cryptography concern [14] in NetSync.
- `Matcher` (Hash matching). In NetSync, the weak hash values of chunks (i.e., FastFP), will be compared to find the potentially-matched chunks on server, and their strong hash values will be further confirmed on clients. Only the chunks whose strong hash values (i.e., SHA-1) are matched will be regarded as duplicates. Otherwise, they will be marked as new chunks (i.e., delta data) and sent to server.
- `Communication protocol`. It is responsible for interactions between client and server in NetSync. It checks the above-mentioned weak and strong hash values of files and ultimately transfers the delta data for synchronization.
- `Network adaptive selector`. This module is responsible for choosing the best configuration in NetSync (i.e., the suitable compressor and chunking granularity). It periodically probes the network condition (`Network Monitor`) and sets up the service

to optimize the sync performance with the best effort from `Compression Selector` and `Chunking Selector`.

General Workflow Overview. Here we briefly illustrate how NetSync works along with the components in Fig. 4. ① `Network Monitor` probes the network condition and sets up the `Sync process` for optimal performance. ② `Sync process` starts to work, client-side `chunker` splits the client file into chunks via FastCDC and sends the metadata to server. ③ The server-side `chunker` splits the server file in the same way. ④ The server-side `matcher` lists weak-hash-matched chunks (i.e., the possible duplicate chunks) between client and server files and sends their ID and strong hash values to client. ⑤ The client-side `matcher` checks strong hashes of these chunks, and `patcher` sends non-duplicate chunks to server. ⑥ The server-side `patcher` integrates the information of duplicate and non-duplicate chunks to restore the client file on server-side, and then the synchronization ends. Note that this is a brief description, and we have hidden many technical details here; a detailed workflow of NetSync with these six key components will be presented in Subsection 4.3 and 4.4.

Key Challenges, Motivations, and Corresponding Designs Overview. As discussed in Section 3, introducing CDC into `rsync`-based approaches simplifies the byte-wise searching. However, we find it has three defects: ① CDC incurs extra computing overhead. ② The original communication protocol for `rsync`-based synchronization is not efficient for the CDC-based approach, specially, the strong hash computation and some network traffic for mismatched chunks are unnecessary in NetSync. ③ The configuration for the intuitive design may be sub-optimal to general cases. In the following subsections, we will analyze and solve these three problems: For defect ①, we utilize the hash value generated by FastCDC and further develop a branch-miss-free extension, named FastFP, to replace Adler32 (in Subsection 4.2). For defect ②, we redesign the communication protocol to eliminate the strong hash computation and metadata transmission for mismatched chunks (in Subsection 4.3). For defect ③, we develop a network-aware module to adaptively find the optimal configuration regarding different scenarios (in Subsection 4.4).

4.2 FastFP: An Efficient Weak Hash by-Product of CDC

As an excellent candidate to substitute FSC for its low computing overhead among the CDC techniques, FastCDC[16] is adopted in NetSync. In this subsection, we elaborate on a novel fast and homogeneous weak hash algorithm (called FastFP) to make full use of the CDC computation results and offset the extra computing overhead introduced by FastCDC.

Intuition. The original chunking process (FSC) has almost no computing overhead under ideal conditions in the `rsync`-based approaches (leave out the boundary-shift problem discussed in Section 1), while a weak fingerprint (Adler32) needs to be calculated after chunking inevitably. As introduced earlier, the CDC algorithm judges the chunk boundary if the hash value of the sliding window satisfies a predefined value. Our intuition is to replace Adler32 with the "CDC by-product" (the hash value generated from the sliding window), i.e., to transform the two processes of chunking (FSC) and weak hashing (Adler32) in `rsync`, into a more efficient process CDC (along with FastFP) in our design. Once the newly constructed weak hash FastFP outperforms Adler32, namely, the sync approach using "CDC + FastFP" outperforms the original `rsync`-based approach with "FSC + Adler32", the extra computing overhead of introducing CDC is offset. Based on this intuition, the newly constructed weak hash should meet the following two key features: uniformity (low hash collision rate) and high speed (faster than Adler32 to help offset CDC overhead).

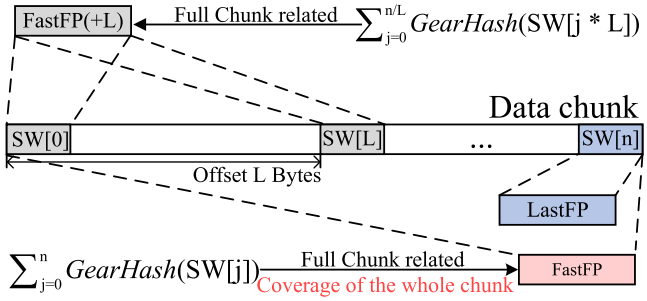


Fig. 5. The calculation and working process of FastFP(+L) and FastFP using FastCDC, note that here "L" is an integer no less than one, which represents the jumping bits of the FastFP(+L), and "SW" represents "Sliding Window".

Uniformity. As shown in Equation (1), FastCDC is a hashing process that generates rolling hashes of a chunk by iteratively shifting the mapped value of each byte content b and summing them up, where the mapping is predefined in the Gear array $G[\,]$ (a random integer array) [16]. During the CDC process, a chunk cut-point is declared if the hash value satisfies a predefined condition. The generated 32-bits "fp", which satisfies the predefined chunk cut-point condition, is named LastFP, as shown in Fig. 5). As a weak hash, it has a much higher collision ratio than Adler32 because it is only relevant to the contents within 32 Bytes from the cut point, while Adler32 covers the whole chunk.

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

To cover all contents of a chunk, we propose FastFP(+L) and FastFP, as shown in Fig. 5. FastFP(+L) adds the hash value of the sliding window every time it strides over a distance of L and can be tuned by changing the operation (ADD or XOR) and the stride (L) to concatenate the fingerprints of the window. Combinations of different stride distance (8/16/32B) and concatenating operations (ADD/XOR) has been evaluated in our previous work [11]. To avoid inconsistencies, unless specifically declared, we set $L = 16$ in our later evaluation. Further, to achieve higher speed, we propose FastFP, which tunes the stride to be 1 byte and it has equivalent uniformity as Adler32, which will be evaluated and discussed in Subsection 5.2.

High Hashing Speed. To speed up the hashing process, we try to make FastFP branchless. As mentioned above, to fully utilize the sliding calculation process of FastCDC, it tries to reduce the calculation by summing up the window's hash values in a predefined stride. However, our latest observation has overthrown the design of stridden summing up. Modern processors work well ahead of the currently executing instructions, and the pipelining works best when the

instructions follow a serial order. However, when a branch is encountered, even with the branch prediction, the CPU still has to discard the speculatively executed results and restart the instruction fetch process [33] if the prediction is incorrect.

The calculation process of the current FastFP(+L) can be summarized in Algorithm 1. If the prediction is correct and the branch is not taken, CPU has to spare one or two more instructions to judge the condition, but it has to spare more CPU cycles to pause, reload instructions and sum up the fingerprint of the sliding window when the branch should be taken. It means summing up the fingerprints with a stride does not save much time but incurs a significant misprediction penalty.

Algorithm 1. The Calculation of FastFP(+L) During Content-Defined Chunking (CDC)

Input: data stream src, length n, position pos, random integer array $G[\,]$[1], jumping bits L
Output: weak hash $fastFP$, cut point i
$fastFP \leftarrow 0$, $fastFP \leftarrow 0$, $Mask \leftarrow 0x0000d9030353$;
for $i \leftarrow pos$; $i < n$; i++; **do**
  $fp = fp << 1 + G[src[\,i\,]]$[2]
  if !$(i \,\%\, L)$ **then**
    $fastFP += fp$; // Update the weak hash;
  end
  if !$(fp \& Mask)$ **then**
    return $(fastFP, i)$; //CDC is finished;
  end
end
return $(fastFP, i)$; //CDC is finished;

1. Length of array $G[\,]$ is 256 (because the byte content range is 0–255).
2. $G[src[\,i\,]]$ means mapping the byte content $src[\,i\,]$ into random bits.

Therefore, we propose FastFP, which is almost the same as FastFP(+L), and the only difference is that FastFP does not jump extra bits. As shown in Fig. 5, "L" is set to 1 in FastFP, and FastFP is FastFP(+1).

Based on the LastFP and strode FastFP(+L), the branchless FastFP is a fast and well-distributed weak hash algorithm, even compared with the widely used weak hash algorithm Adler32 (will be evaluated in Subsection 5.2). Significantly, FastFP has a better hashing efficiency than Adler32, which means replacing Adler32 with FastFP can help offset the additional computing overhead brought by CDC. In other words, the CDC process not only divides chunks but also produces weak hashes in NetSync, and it is more efficient than FSC + Adler32 process in `rsync`.

4.3 Communication Protocol

This subsection proposes a novel communication protocol to eliminate strong hash computation and metadata transmission for mismatched chunks.

Reducing Strong Hash Computing Overhead. According to our observation of the FastCDC-based NetSync prototype, once the weak hash is mismatched, the computation of the strong hash will be wasted (i.e., the strong hash computation is redundant in this case). Therefore, the computation of the strong hash on client in the pre-treatment phase can be reduced. Based on this observation, we construct a new

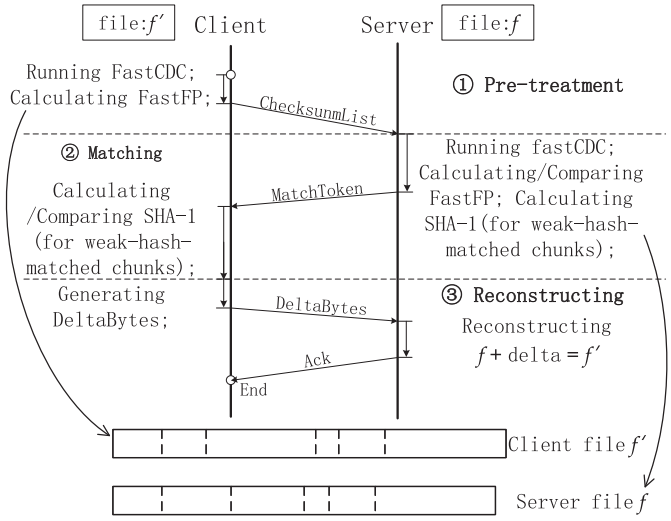


Fig. 6. The redesigned protocol of NetSync to minimize the strong hash calculation.

sync protocol to make the strong hash computation no longer redundant. As shown in Fig. 6, the new protocol has several improvements on the three stages of NetSync:

- In Pre-treatment stage, NetSync splits the client file f' into several chunks via FastCDC and calculates their fingerprints (i.e., FastFP) (note that strong hash values are no longer calculated in this stage). After that, the indexes and fingerprints of the chunks are packed into Checksum List and sent to server.
- In Matching stage, after receiving Checksum List, server splits server file f into chunks by FastCDC and calculates the weak hashes as client does. Then server searches Checksum List to find the weak-hash-matched (i.e., potentially duplicate) chunks. For the weak-hash-matched chunks in file f, server will compute their SHA-1 values for further duplicate confirmation on client. Then server sends Match Tokens, which includes all weak-hash-matched chunks' indices of file f' and SHA-1 values of matched chunks in file f to client. After receiving Match Tokens from server, client checks SHA-1 values of weak-hash-matched chunks of file f' and records the duplicate chunks' IDs.
- In Patching stage, the mismatched chunks, together with their indices, form Delta Bytes and are packed into Patch Tokens and sent to server. Eventually, server reconstructs file f into client file version f' according to Patch Tokens.

Compared with the preliminary FastCDC-based NetSync protocol, client does not calculate SHA-1 values of all chunks. Instead, it only calculates SHA-1 values of weak-hash-matched chunks of file f' according to Match Tokens from server. Thus SHA-1 calculation is minimized, as shown in Fig. 6. In consequence, the task of searching SHA-1 values is shifted from server to client.

Note that some details are omitted in Fig. 6, e.g., there is a query before each synchronization. During the query process, the file's name, along with its directory path and modification time, is used to confirm whether this file is first uploaded, whether it has been modified, etc. The query is

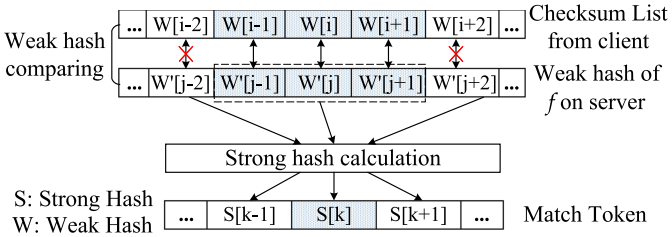


Fig. 7. Merging several consecutive weak-hash-matched chunks into one chunk to reduce metadata overhead.

concise and efficient but not very relevant to the sync process, so `rsync` [20] omits this part in its sync protocol framework, and we present our protocol in the same way.

Reducing Network Traffic. According to several deduplication-related studies [14], [34], duplicate chunks in a file appear in approximately the same order throughout multiple full backups or similar files with a very high probability (called the redundancy locality). The locality can also be exploited in NetSync for network traffic reduction. Specifically, we can collectively merge the consecutive weak-hash-matched chunks (i.e., potential duplicates) to reduce the metadata traffic in NetSync.

Fig. 7 shows how we merge several consecutive weak-hash-matched chunks in NetSync. After receiving Checksum List from client, server splits the server file f into chunks and calculates their weak hash values as client does. Then server compares the weak hash values of its file f with the weak hashes in Checksum List. The consecutive weak-hash-matched chunks will be merged. Subsequently, only one strong hash will be computed and sent back to client for later comparison, in other words, the merge helps reduce the metadata traffic. Suppose the strong hash fingerprint of a merged chunk does not match the client's (denoted as "Merge Collision"). In this case, the mismatched merged chunk will be directly transferred to server if the size of this merged chunk is below a predefined threshold (we set it as 800KB in our design); otherwise, we will require server to re-transfer the strong hash fingerprints of the constituent chunks to client (to further check duplicate chunks in this merged chunk).

The Merge Return. As discussed above, the merge can help reduce the metadata traffic. However, the merge will be penalized when "Merge Collision" happens. After adopting the merge strategy, a weak hash collision may not only affect the collided chunk itself but also affect all the chunks merged with it. As shown in Fig. 7, $chunk[i-1]$, $chunk[i]$, and $chunk[i+1]$ are consecutive weak-hash-matched chunks. Assume that only $chunk[i-1]$ and $chunk[i+1]$ are identical, and $chunk[i]$ is modified (i.e., the weak hash value of $chunk[i]$ collides with corresponding unmodified chunk). If we do not merge, we can eliminate the redundant $chunk[i-1]$ and $chunk[i+1]$. While after merging, we need to transfer all three chunks through the network. To make this clear, we suppose that the probability of weak hash collision between the modified chunk and the original chunk is p. In our case, a merged chunk's hash collision happens only when all the modified constituting chunks' collisions happen simultaneously. Assuming that we are merging n chunks, and they may include several modified chunks. We denote the probability of m chunks being modified as c_m (where $0 \leq c_m \leq 1$, $\sum_{m=0}^{n} c_m = 1$, $0 \leq m \leq n$), and in this case, the probability that the hashes of all modified chunks collide is $P_m = p^m$, where $0 < m \leq n$. Therefore, the probability of "Merge Collision" can be calculated as:

$$P_{MergeCollision} = \sum_{m=1}^{n} c_m p^m \tag{2}$$

Considering $\sum_{m=0}^{n} c_m = 1$, we can conduct that:

$$(1 - c_0)p^n \leq P_{MergeCollision} \leq p \tag{3}$$

Suppose we employ the merge. When Merge Collision happens, we will get a negative return of the chunk size (e.g., $8KB$) per chunk; when there is no merge collision, we will get a positive return of the metadata size (strong hash value ($20B$) + index ($4B$) = $24B$) per chunk. The expected return R (per chunk) will be $\mathrm{R} = (1 - P_{MergeCollision}) * 24B + P_{MergeCollision} * (-8KB)$, so we can get that when $P_{MergeCollision}$ is smaller than 0.003, R will be positive. While according to Equation (3), $P_{MergeCollision}$ will never be larger than the weak hash's collision rate (much smaller than 0.003), which is acceptably low in our design. In summary, the expected merge return is positive. In other words, it is practical to conduct the merge in NetSync.

In addition, the merge also helps reduce the computational overhead for strong hashing (i.e., SHA-1). Although the data volume to be strong-hash fingerprinted remains unchanged after the merge, the number of strong-hash calculating operations will be greatly reduced. More specifically, the SHA-1 calculation is composed of three sub-processes, namely `SHA1_Initial` (initializes a SHA1 structure), `SHA1_Update` (hashes the chunk contents), and `SHA1_Final` (places the results). By using the merge in NetSync, the frequent `SHA1_Initial` and `SHA1_Final` operations will be greatly reduced.

4.4 Network Adaptive Parameter Selector

As discussed in Section 3, the mismatch between bandwidth and sync parameters degrades the overall performance. NetSync wedges two network adaptive modules to overcome this shortcoming, choosing the appropriate chunking and compressing configurations accordingly. As shown in Fig. 9, the optimal workflow of the NetSync protocol differs from Dsync in three aspects: ① The net-aware module may be executed at the beginning of synchronization. In most cases, a previous network condition is preserved and applied for estimation in the subsequent chunking and compression modules; otherwise, it is executed to probe the network condition. ② NetSync introduces a network adaptive compression module (`Compression Selector`) to strike a balance between network and calculation resources. ③ A network adaptive chunking module (`Chunking Selector`) is introduced to reduce the transmitted data properly. As will be evaluated later, the optimized protocol is apt to fully utilize the network, and it reduces up to $2.79\times/1.57\times$ sync time compared with WebR2sync+/Dsync, respectively.

Net-Aware Implementation. Among the open-sourced speed test programs, we use LibreSpeed, a lightweight, high-performance tool supported by most modern browsers [35]. The speed test tool enables the module to run in the

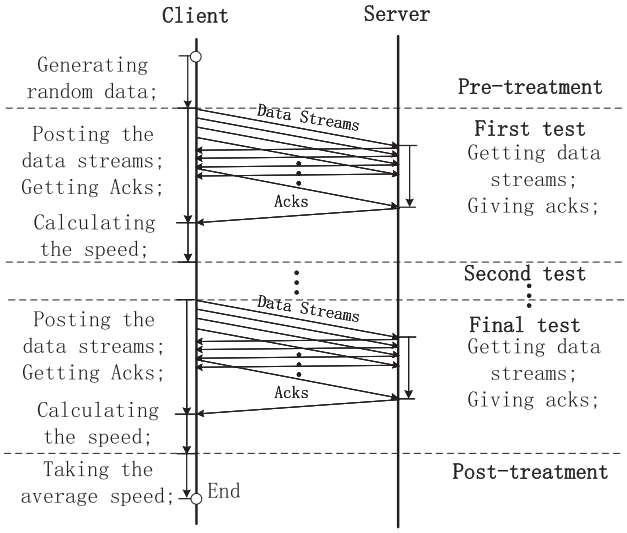


Fig. 8. The workflow chart of the net-aware module.

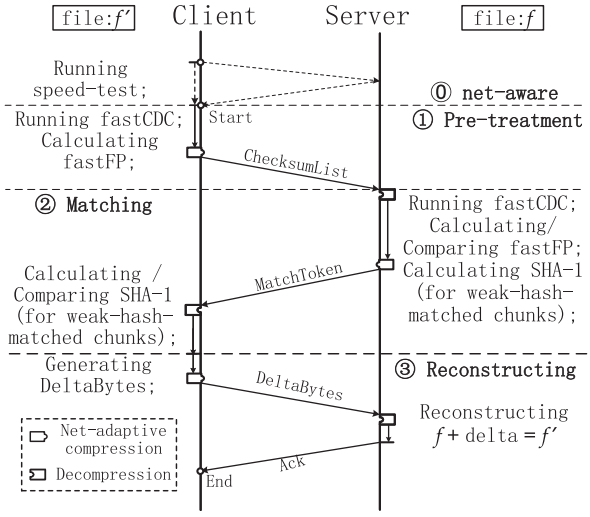


Fig. 9. The complete NetSync protocol.

background [36], but we block the requests to synchronize data during bandwidth estimation out of accuracy concerns. Specifically, as shown in Fig. 8, the net-aware module (i.e., `Network Monitor`) is comprised of three stages:

- Before starting the measurement, the net-aware module first generates some arrays filled with random numbers, each of which takes 1MB or 256KB.
- Then it starts a series of network speed tests, each test posts random arrays in the form of data streams to server, and shortly afterward, it gets Acks. The interval between each test is about 200 milliseconds. Streams are slightly delayed so that they will not end simultaneously, and no compression is introduced to this test. At the end of each test, network bandwidth will be calculated according to the transmitted data volume and the corresponding transmission time and saved into the `speed tests queue`.
- After a series of speed tests, the net-aware module is terminated. We save the average results in `speed tests queue`, which will be used by `Compression Selector` and `Chunking Selector`for the next period.

Network Adaptive Compression. The network transmission accounts for a large proportion of sync latency in NetSync, especially under limited network conditions (e.g., up to 90% in the worst case, as indicated in Section 5). The network time is contributed by the data transmission and the delay to set up the connection, and it can be calculated as shown in Equation (4).

$$t_d = \frac{D}{N} + t_p \tag{4}$$

where D is the data size, N is the network bandwidth, and t_p is the network delay in one direction (i.e., half of round-trip time). To decrease the transmission time, we could either reduce the size of transferred data or use a high-bandwidth network (higher N and lower t_p). Given the network condition, the common practice is to apply data compression. However, the default per-message `deflate` compression extension supported in WebSocket protocol [37] may be sub-optimal regarding network conditions.

Essentially, data compression during synchronization is a trade-off between CPU time and network time. With the compression introduced, the compensated transferring time (compressing + transferring) can be calculated below.

$$t_x(D) = \frac{D * R_x}{N} + t_{cx}(D) + t_{dx}(D) + t_p \tag{5}$$

In Equation (5), apart from the parameters mentioned above, R_x is the compression ratio of compressor x, $t_{cx}(D)$ is the time spent on compressing the data of size D, and $t_{dx}(D)$ is the corresponding decompressing time. In common sense, a compressor with a higher compression ratio takes more CPU time. Under limited network conditions, the sync may not benefit much from fast but insufficient compression and spend much time transferring the data. On the contrary, if the bandwidth is sufficiently high and the data is most compressed, the CPU may be bottle-necked by the compressor. In order to maximize the sync performance in NetSync, we need to strike a balance between compression overhead and network transmission time.

The coefficients R_x, t_{cx}, and t_{dx} cannot be directly inferred in terms of specific compressors. Instead, we could run different compressors and estimate their coefficients. After collecting the features of different compressors, the `Compression Selector` chooses a compressor that minimizes the compensated transferring time in Equation (5).

Network Adaptive Chunking. Apart from the compression module, CDC parameters, especially the chunking size, also affect the computation and communication overheads in the sync process, thus affecting the sync latency. The `rsync`-based approaches [9], [38], and our former work [11] choose 8KB as the expected chunk size. As we choose a smaller chunk size, a series of reactions happen:

- More chunks will be generated, which results in more chunking and hashing overheads.
- More chunks' metadata will be generated, which leads to more overhead for the network transmission.

- A finer chunking granularity also means more data reduction and high network transmission efficiency. Moreover, this saves the computing overhead of the compression during sync.

When we choose a coarser chunking granularity, the series of reactions will behave otherwise accordingly.

Considering all the reactions of changing the chunking granularity described above, we can get a general choice for `Chunking Selector`. When the network bandwidth is low, we can set a smaller chunk size for `chunker`, which helps minimize the network transmission time while increasing computational overheads; otherwise, we set a larger chunk size. Specifically, the initial chunk size is set to 1KB, and with better/worse network conditions, `Chunking Selector` chooses a larger/smaller chunk size (use the chunk sizes of $512B$, $1KB$, and $2KB$ for network bandwidths $(0, 10)$, [10,40), and $[40, +\infty)$ (Mbps) respectively) according to our experimental studies (see Fig. 16 in Subsection 5.4). Moreover, our later evaluation results in Subsection 5.4 indicate that the chunking sizes selected by `Chunking Selector` perform well in NetSync.

5 Evaluation

In this section, we first introduce the experimental setups for NetSync. Then we conduct a sensitivity study of NetSync and give some evaluations. Besides, an overall comparison between WebR2sync+ [9], deduplication-based solutions, Dsync, and NetSync solutions are evaluated and discussed.

5.1 Experimental Setups

Experimental Platform. We conduct our client experiments on PC with Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz 3.19 GHz, 16GB memory and Windows 10 operating system and iPhone 11, a mobile phone with 4GB RAM, 128GB ROM, and IOS14.5. In addition, the server runs on node v14.15.5 with a quad-core virtual machine @3.2GHz (installed Ubuntu Server 18.04 with 16GB memory and 128GB disk). To simulate actual network status, we tune the representative bandwidth to be 10Mbps, 50Mbps,100Mbps, 1Gbps, 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 sync process. The sync traffic metric measures the total amount of data transmitted, including Checksum List, Match Token, and Delta Bytes, as discussed in Section 4 (shown in Figs. 6 and 9). For each data point reported in this section, unless specially explained, we run the experiment five times to obtain a statistically meaningful average measurement for the sync performance.

Delta Sync Configurations. We build our NetSync on top of the state-of-the-art `rsync`-based WebR2sync+ [9]. In Dsync (an intermediate version of NetSync in which the network adaptive modules are not included), we enable the `deflate` compressor and use 8KB as the default avg. chunk size, as recommended in WebR2sync+ [9] and LBFS [15]. In NetSync, unless specially explained, the `Chunking Selector` sets a proper avg. chunk size according to the current network bandwidth (as discussed in Subsection 4.4, $512B$, $1KB$, and $2KB$ for network bandwidths $(0, 10)$, [10,40), and $[40, +\infty)$ (Mbps) respectively). The default jumping bits L is set to 16 in FastFP(+L). In the evaluation, all experimental conditions are the same unless otherwise explained.

Benchmark Dataset. Silesia [39] is a widely acknowledged dataset for data compression [40] covering typical data types that are commonly used, including text, executables, pictures, HTML, etc. According to several published studies on real-world and benchmark datasets [41], [24], the file modifications are made at the beginning, middle, and end of a file with a distribution of 70%, 10%, and 20%, respectively [41]. Similar to the operations in WebR2sync+ [9] and QuickSync [29], 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 the Silesia corpus and modified the file with the modification size of 32B, 256B, 2KB, 16KB, 128KB, and 1MB respectively. Three types of file modifications are made following the pattern described above, and each fraction for the file modifications takes 256B at most.

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

- PPT. We collect 48 versions of a PowerPoint document from personal uses in the cloud storage (totaling 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 about 20MB, totaling 860 MB.
- Pictures. We use a public picture manipulation dataset [42], which contains 48 pictures in "PNG" format (totaling 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 [43], and each mail contains past replies. The mails are tarred with two versions by time, and the total size is about 1839 MB.

The Baseline Sync Performance. As discussed in Section 1, sync performance (sync latency) is considered the most critical factor for cloud storage services. `Rsync` is a classic delta sync approach. However, as shown in Fig. 10, it has been far exceeded by the state-of-the-art Web-based delta sync WebR2sync+ [9], which is an improved delta sync approach based on WebRsync (i.e., one kind of `rsync` [27] approach implemented on Web browser for better platform adaptability). To make our evaluation more straightforward, we use WebR2sync+ rather than WebRsync (Web-based `rsync`) for sync performance comparison in the following experiments.

5.2 Performance of the Weak Hash Implementation

In this subsection, we evaluate the performance of all our weak hash derivatives. Hashing efficiency and uniformity of hash value distribution are the two most important metrics that are the main concern. To evaluate the hash collision rate, we first generate a dataset with 30 files consisting of random numbers for the given size of 1GB, 10GB, and 100GB (10 files each). To make the evaluations rigorous, we execute each experiment 10 times and average the results.

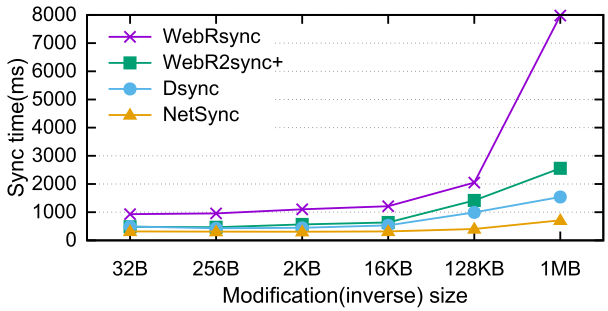


Fig. 10. A glance at sync performance on one of our benchmark dataset (inverse modification). Note that the network bandwidth is 100Mbps; WebRsync (Web-based `rsync`) is the `rsync` approach implemented on the Web browser.

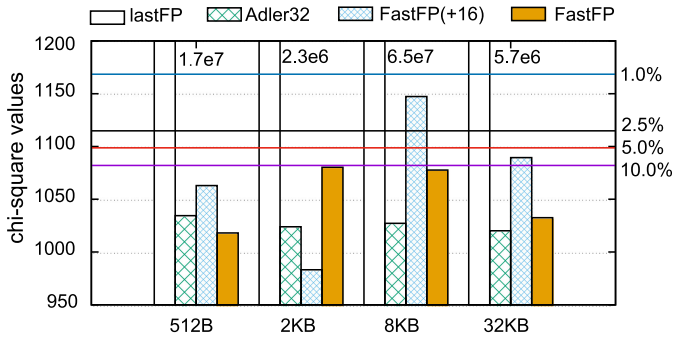


Fig. 11. The Chi-Square uniformity test. Note that the horizontal axis represents different chunk sizes. Here 1%, 2.5%, 5%, and 10% stand for different confidence levels of the Chi-Square-goodness-fit test.

Hashing Throughput. The averaged throughputs of the hash functions with different configurations are shown in Table 2. LastFP has the highest throughput because it is generated directly from the CDC process with no extra overhead introduced. However, our later evaluation (Fig. 11) shows that it is unqualified to be a weak fingerprint for its extremely high collision rate. As shown in Table 2, FastFP(+16) has a much lower throughput than LastFP because it introduces extra computing overhead to obtain a lower collision ratio. More importantly, FastFP has a better hashing efficiency than Adler32, i.e., replacing Adler32 with FastFP can help offset the additional computing overhead brought by CDC. In general, FastFP works well in terms of the hashing efficiency, and it will show a greater advantage in later evaluation (uniformity of hash value distribution).

Uniformity of Hash Value Distribution. Fig. 11 shows the uniformity of all hash function derivatives. Generally speaking, to avoid hash collisions, a hash function should map the data to any probable hash value with an equivalent chance. Therefore, the uniformity of hash function is evaluated via the chi-square-goodness-fit test [44], a widely used method to justify whether the hash value distribution is significantly different from a uniform distribution. The main idea of this test is to hypothesize that the hash value distribution is uniform and calculate a test statistic χ^2 to indicate how much it has diverged from the expected distribution. Equation (6) indicates that the probability of a χ^2 larger than $\chi^2_\alpha(n)$ is α (knows as the confidence level, normally as low as 1% to 5%). Moreover, if we get a large χ^2, the hypothesis is overthrown. Note that a higher confidence level means stricter constraints over the hypothesis, e.g., 10% confidence level is more convincing than 5%.

$$\mathrm{P}\{\chi^2 > \chi^2_\alpha(n)\} = \alpha \tag{6}$$

TABLE 2
The Throughput of Four Hash Algorithms

chunk size	Hash throughput(MBps)			
	LastFP	Adler32	FastFP(+16)	FastFP
512B	471	301	266	334
2KB	559	313	253	343
8KB	623	314	313	368
32KB	664	304	295	355

In our case, we first divide the weak hash value range R into b bins, each bin containing R/b values. Then we traverse weak hash values of all chunks and count the numbers of values that lie in the intervals of bins respectively ($[0, R/b-1]$, $[R/b, 2R/b-1]$, $[2R/b, 3R/b-1]$, ...). After that, we calculate the statistic chi-square value of our sample group, shown in Equation (7), and look up the chi-square distribution table to determine the confidence level. For example, if we divide 8-bits hash values into 4 bins and the numbers of chunks (100 chunks in total) that fall into the bins ($[0x00, 0x3F]$, $[0x40, 0x7F]$, $[0x80, 0xBF]$, $[0xC0, 0xFF]$) are $20, 25, 29, 26$, then b is equal to 4 and n_1 through n_4 are $20, 25, 29, 26$. Furthermore, the expected number of chunks $\bar{n}$ in the perfectly uniform hash value bins should be $100/4 = 25$. As we get a statistic χ^2 value of 1.68, we could refer to the chi-square distribution table and get the critical value for the 10% confidence level is 6.25 and that for the 90% is 0.58. It means our hypothesis is accepted with the 10% confidence level but rejected with a 90% confidence level. If another hash function, $g(x)$, is accepted with a 90% confidence level, $g(x)$ is more uniform.

$$\chi^2 = \sum_{\mathrm{i}=1}^{b} \frac{(\mathrm{n}_i - \overline{n})^2}{\overline{n}} \tag{7}$$

To be more specific, our 32-bit hash function (from 0 to $2^{32}-1$) range is divided into 1024 bins, and we compare SHA-1 hash values of chunks to check if two chunks are identical. The hypothesis test results shown in Fig. 11 indicate that LastFP is always rejected; FastFP(+16) can be accepted with the 1% confidence level; in contrast, both Adler32 and FastFP can be accepted with the 10% confidence level. As concluded in Subsection 4.2, LastFP is only relevant to the contents within 32 Bytes from the cut point. Because of its limited coverage, LastFP is unacceptable, and inversely with more data included in the hash function, FastFP achieves better uniformity than FastFP(+16).

In summary, evaluation results suggest that our proposed FastFP is much faster but no weaker than the widely acknowledged Adler32 (used in `rsync`).

5.3 Performance of the Redesigned Protocol

In this subsection, we evaluate the performance of our redesigned communication protocol which aims to minimize the time-consuming strong hash computation and reduce the amount of metadata transmitted over the network.

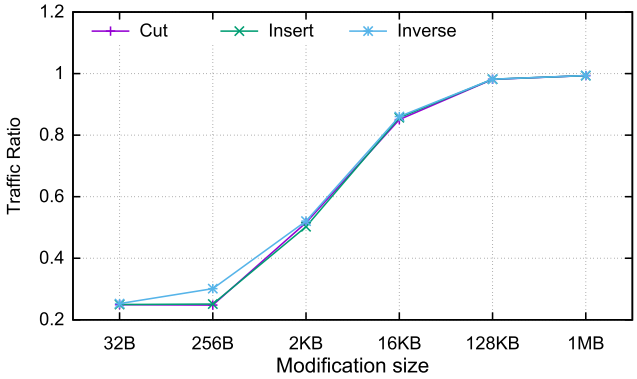


Fig. 12. Total traffic ratio as a function of modification size, after/before applying the NetSync protocol that merges consecutive weak-hash-matched chunks.

Fig. 12 shows the ratio of total traffic (without compression) over the network after/before applying our NetSync protocol that merges consecutive weak-hash-matched chunks. When the modification is light (e.g., inserting 32 bytes or inversing 256 bytes, etc.), the sync traffic will be reduced by 70%–80% in NetSync. As modification granularity grows, the ratio approaches 100%. As modification size increases, the traffic is gradually dominated by Delta Bytes rather than metadata since few, if any, weak-hash-matched chunks will be found. Specifically, when synchronizing a 10MB file with the average chunk size set to 8KB, the transmission of almost 1280 chunks' SHA-1 values, i.e., totaling 25KB of metadata, will be eliminated by the redesigned protocol. However, the reduced metadata will be far overshadowed by the much larger amount of Delta Bytes transmitted from client to server if too many modifications are made to the file.

Fig. 13 depicts the numbers of hash comparing operations in NetSync and WebR2sync+, including both strong and weak hash. We can conclude that owing to the strategy of merging consecutive weak-hash-matched chunks into one chunk, NetSync has much fewer operations for comparing weak and strong hashes than WebR2sync+. To a certain extent, the occurrence of strong hash comparison operations for each synchronization represents the number of matched chunks. The more strong-hash comparisons are executed, the more matched chunks we find (compared within the same sync process). Especially, when the modification size is 1MB, NetSync adopts the merge strategy, but it still has a higher frequency of strong hash comparison than WebR2sync+. That is because it is hard for WebR2sync+ to find unmodified chunks under a coarser modification granularity since the distance between two modifications is almost 8KB. In contrast, NetSync can find matched chunks smaller than 8KB (the chunk sizes are variable by CDC), as stated in Section 3. Therefore, we can also conclude that NetSync gets the lead in finding more duplicate chunks by introducing FastCDC, especially under a coarser modification granularity.

Then we evaluate the sync time of WebR2sync+, Dsync, and NetSync as a function of modification granularity on client under three different network bandwidths, as shown in Fig. 14. The sync time on client, as depicted in Fig. 9, which is mainly composed of generating Checksum List, finding delta bytes, and compressing + decompressing time, can represent the computing overhead and resource occupation of client during the sync. Significantly, the patchdoc compression time occupies a large proportion of client time when the modification granularity is coarser in Dsync and WebR2sync+, where the default compressor is set to `deflate`. When the modification is light (e.g., less than 2KB), NetSync performs better than WebR2sync+ because it employs the FastFP, which helps compensate for the chunking overhead, and the improved protocol, which reduces the strong hash computation of client. When granularity is

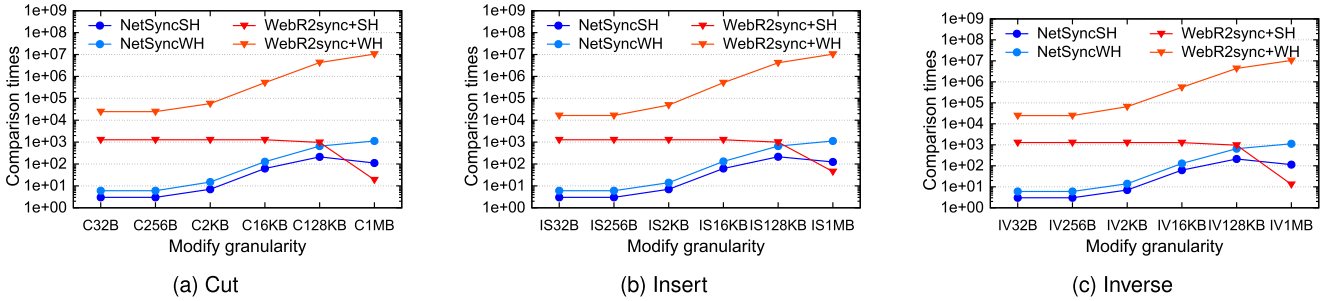


Fig. 13. Numbers of hash-comparing operations of NetSync and WebR2Sync+, including both strong and weak hashes. Note: the "SH" (solid lines) and "WH" (dotted lines) stand for the strong and weak hash comparing operation frequencies, respectively; the network adaptive modules are not included in NetSync here.

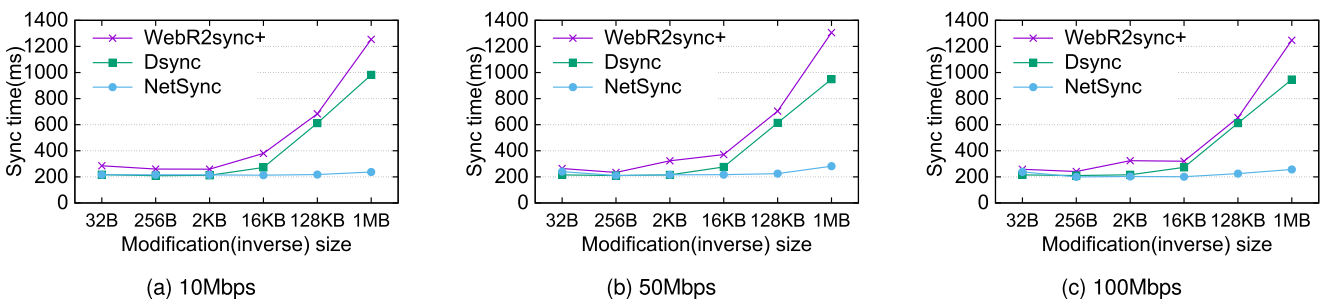


Fig. 14. Client time comparison of NetSync, Dsync, and WebR2sync+ under different network bandwidths.

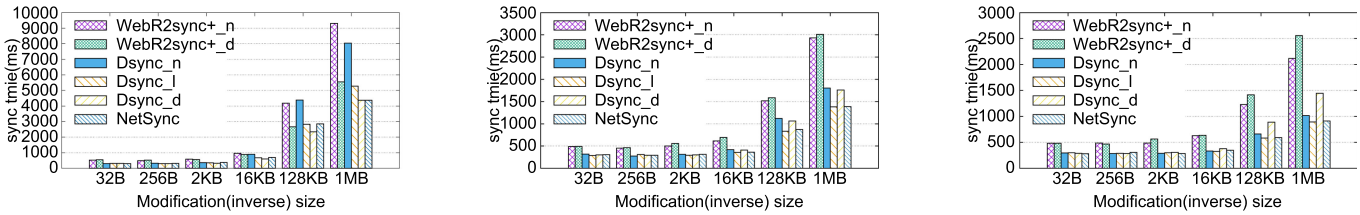


Fig. 15. The sync time of NetSync compared with WebR2sync+ using `none` and `deflate`, and Dsync using `none`, `lz4`, and `deflate` under three different network bandwidths. Note: WebR2sync+_n and WebR2sync+_d represent WebR2sync+ using `none` and `deflate` compression respectively; Dsync_n, Dsync_l, Dsync_d represent Dsync using `none`, `lz4` and `deflate`, respectively; and `Chunking Selector` is not included in NetSync here.

coarser (e.g., 1MB for 10MB file), the ability for FastCDC to discover more duplication reduces the traffic volume and thus reduces the compression time, which finally helps reduce client time. When the network condition is limited, NetSync employs the same compressor as Dsync and thus has the same performance. However, when the network bandwidth is sufficiently high, NetSync employs `none` or `lz4` instead, which leads to better performance. Therefore, with its improvements, NetSync outperforms WebR2sync+ on client under various network bandwidths and different modification granularity.

5.4 Performance of Network Adaptive Modules

In this subsection, we briefly evaluate the performance of the network adaptive compression and network adaptive chunking in NetSync.

5.4.1 Performance of Network Adaptive Compression

To make a good comparison, we test WebR2sync+ with the compression method of `none` and `deflate`, and Dsync with the compression of `none`, `lz4`, and `deflate`, respectively. As shown in Fig. 15, we take three sync groups with different network bandwidths under the inverse-modification benchmark datasets.

As illustrated in Fig. 15, the network conditions can greatly affect the transmission time, and the optimal compressor differs under different network conditions (e.g., `deflate` for 10Mbps while `lz4` for 50Mbps, etc.), and NetSync with the `Compression Selector` shows a great advantage among the three sync methods. It performs 1–2.19× faster than WebR2sync+ and 1–1.57× faster than Dsync. When the network bandwidth is limited, NetSync performs the same as Dsync and sometimes even worse. When the network conditions are better, NetSync completely outpaces its rivals, and the better the network condition is, the better NetSync performs. Modification granularity also has a crucial impact on sync performance. With a coarser granularity, the number of delta Bytes transmitted in the network will be larger, so NetSync performs better.

NetSync cannot always choose the best compressor (e.g., in Fig. 15a, when the inverse size is 128KB, NetSync chooses a sub-optimal compressor) for there may be a gap between the real characteristics and the estimators. However, just as Fig. 15 illustrates, generally, the gap has little impact, so even if the optimal one is not selected, it is likely to choose the sub-optimal solution that is also appropriate for the current synchronization.

5.4.2 Performance of Network Adaptive Chunking

The network adaptive chunking method aims at finding appropriate chunk sizes for different network conditions to minimize the sync time. We evaluate the network adaptive chunking performance by comparing sync efficacy with the non-adaptive edition, Dsync, in terms of the total sync time, because the chunking granularity is closely coupled with every stage and cannot be stripped off for sole evaluation. We keep the compressor invariant (`deflate`) to eliminate the influence of using different compressors.

Fig. 16 shows the normalized sync performance corresponding to different chunk sizes under different network bandwidths. Note that we set the best sync time to zero and the worst to one (normalization), in other words, the lowest point of each polyline (the point with zero ordinate) represents the best chunk size for the current network. As shown in Fig. 16, different network bandwidths correspond to different optimal chunk sizes. Generally, the higher the network bandwidth is, the greater the optimal chunk size will be. More precisely, we can get a general rule for `Chunking Selector` according to Fig. 16: use the optimal chunk sizes of $512B$, $1KB$, and $2KB$ for network bandwidths $(0, 10)$, [10,40), and $[40, +\infty)$ (Mbps) respectively (note that we have tested large quantities of network bandwidths; only the results near the critical optimal chunk sizes are shown in Fig. 16). Especially, the default chunk size (8KB) recommended by LBFS [15] and some other `rsync`-based approaches [9], [38] is overthrown by Fig. 16. Besides, the gap between the optimal and sub-optimal chunk sizes is minor. That is to say, if `Chunking Selector` cannot precisely set the optimal chunk size, it is likely to

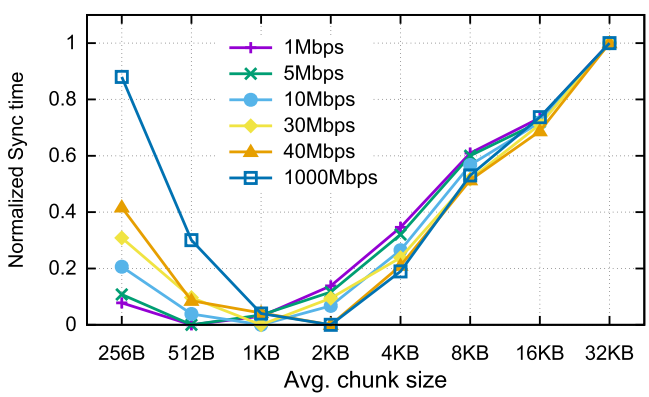


Fig. 16. Normalized sync performance corresponding to different chunk sizes under different network bandwidths. Note that we set the best sync time to zero and the worst to one (normalization), in other words, the lowest point of each polyline (the point with zero ordinate) represents the best chunk size for the current network.

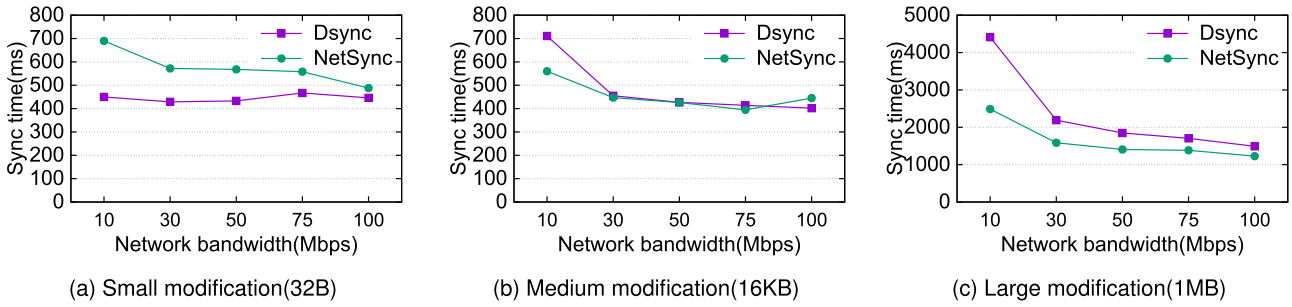


Fig. 17. The overall sync time of NetSync with `Chunking Selector` compared with Dsync under three different modification granularities. Note: the network adaptive `Compression Selector` is not included in NetSync here, and the y-axis scales of the three subfigures are different.

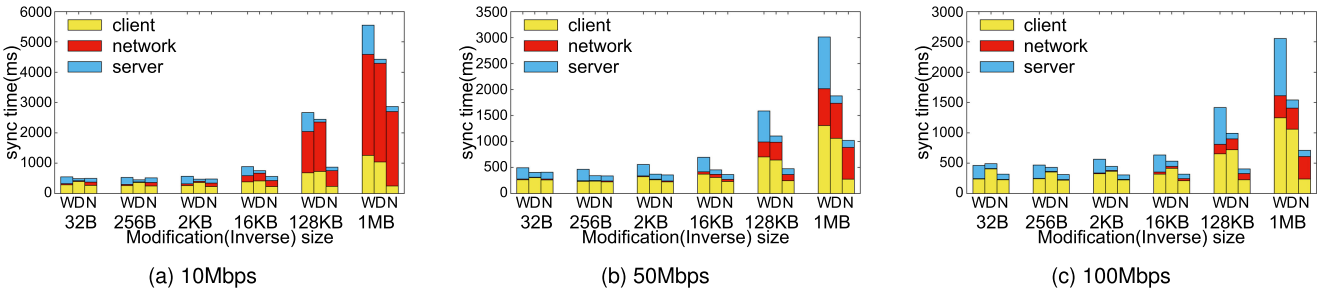


Fig. 18. Sync time breakdown comparison of NetSync, Dsync, and WebR2sync+ as a function of modification sized under different network bandwidths. Note: "N" represents NetSync, "D" represents Dsync, and "W" represents WebR2sync+, respectively.

set a sub-optimal chunk size which is also appropriate. Note that Fig. 16 is conducted on one of our benchmark datasets (inverse), and it has similar results on other (both benchmark and real-world) datasets.

As shown in Fig. 17, the performance of network adaptive chunking indicates two points. First, the network adaptive chunking performs better when modification granularity is coarse, while it is nearly the same when modification is insignificant. It is because the default chunking size of Dsync is already fine enough, and finer granularity is not much of an improvement, not to mention the extra metadata and computing overhead. If the modification is significant, finer granularity can help find more redundancy, thus saving the computing and network overhead. Second, the improvement is more significant when the network bandwidth is higher. Even though chunking size influences the performance of every stage, it is proposed mainly to reduce network traffic. As the bandwidth improves, the proportion of network transmission is decreased, and the significance of transmission reduction declines accordingly.

5.5 Put It All Together

In this subsection, we evaluate the overall performance of NetSync, including the sync time breakdown, sync time on client, and the capacity of supporting multiple clients, etc.

We first evaluate the sync time breakdown of NetSync compared with WebR2sync+ and Dsync. Results are shown in Fig. 18. Compared with WebR2sync+, NetSync and Dsync tend to have a lower client and server time which makes Dsync and NetSync consistently outperform WebR2sync+. NetSync tends to have almost the same sync time as Dsync under limited network conditions (i.e., 10Mbps). When the network bandwidth is sufficiently high (i.e., 50Mbps or 100Mbps), NetSync shows better performance by selecting `lz4` or `none` as its compressor to reduce the compressing and decompressing time. At this time, the network transmission time slightly increases, while client time is greatly reduced, resulting in the reduction of the overall sync time. Modification size also plays a key role in the sync performance. When it is light (e.g., less than 2KB), chunking and hashing time account for most sync time. In this case, NetSync/Dsync performs better than WebR2sync+ for using FastFP. As the modification size increases, the positive effect of the network adaptive chunking begins to show, and NetSync's advantage over WebR2sync+ and Dsync becomes pronounced, especially when the network bandwidth is sufficiently high.

We also evaluate the scalability of WebR2sync+, Dsync, and NetSync in terms of the ability to support multiple clients simultaneously. The main indicator of our concern is the CPU utility which can reflect the calculating capacity of clients during the synchronization to a certain extent, as shown in Fig. 19. In this evaluation, we run the experiment 5 times and get the average values. We also list the statistical error bounds in all subfigures of Fig. 19. Be aware that the lower the CPU utility, the more scalable and capable for sync approaches to support multi-clients in this evaluation. Fig. 19 indicates that NetSync has the lowest overall CPU utility among all tree sync approaches. That is to say, NetSync is the most scalable sync approach. In other words, when a PC can only support 220 clients of WebR2sync+ at most, it can support 300 Dsync clients or nearly 400 NetSync clients.

Finally, we run our tests on four real-world datasets. In this evaluation, we conduct our tests of WebR2sync+, Dsync, and NetSync on PC (Windows) and mobile phone (IOS), as shown in Table 3. On Windows, NetSync is 1.78×–5.26× faster than WebR2sync+ (Dsync is 1.5×–3.3× faster than WebR2sync+). On IOS, the trend is similar to that of

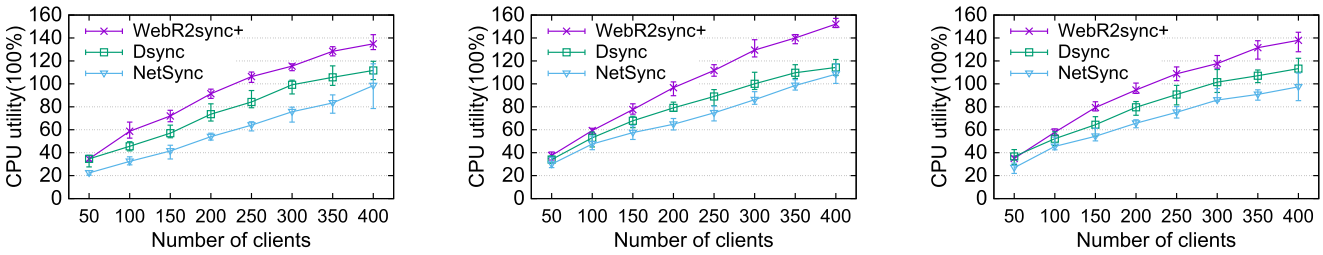


Fig. 19. Multi-Clients comparison of NetSync, Dsync, and WebR2sync+ on a single VM server instance.

TABLE 3
Sync Performance of the Three Approaches on Both Windows / IOS Clients on Four Real-World Datasets

	100Mbps Sync time(s) (Windows/IOS)			50Mbps Sync time(s) (Windows/IOS)			10Mbps Sync time(s) (Windows/IOS)		
Datasets	NetSync	Dsync	WebR2sync+	NetSync	Dsync	WebR2sync+	NetSync	Dsync	WebR2sync+
Pictures	14.2/18.0	19.6/38.1	25.0/41.3	16.6/24.2	28.5/34.5	37.1/53.4	87.6/108.5	94.0/107.8	102.3/112.6
PPT	7.2/9.4	8.7/13.7	12.2/16.9	9.8/13.6	11.9/18.8	15.0/22.6	30.3/33.4	30.5/36.5	40.7/48.3
Mail	29.9/34.9	37.3/54.6	98.2/125.3	41.9/44.3	46.4/63.4	81.5/84.9	83.5/101.0	97.2/104.9	411.5/440.2
GLib	33.9/37.8	49.3/63.2	119.2/149.4	42.8/46.8	57.4/73.6	164.2/193.4	104.7/121.6	109.8/135.2	482.3/518.1

Note that Windows / IOS platforms have the same sync traffic under the same network bandwidths, they only differ in the computing capacity.

Windows. Moreover, the results indicate that as lightweight delta sync, NetSync performs well on the mobile platform, with less storage and computational resources.

5.6 High Bandwidth and Large Files

This subsection evaluate NetSync performance on large files under the Gigabit network environment, as shown in Fig. 20. As mentioned earlier, the chunk-matching process is very time-consuming in the `rsync`-based approaches. The results shown in Fig. 20 suggest that `rsync`-based WebR2sync+ spends $2\times$–$10\times$ more sync time compared with NetSync. 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. Due to the design of network adaptive modules, NetSync has reduced the sync time by $1.25\times$–$1.4\times$ when comparing with Dsync and $3\times$–$10\times$ compared with WebR2sync+.

6 Conclusion

The traditional `rsync`-based approaches introduce heavy computing overhead in the chunk-matching process for cloud storage services. CDC-based deduplication simplifies the chunk-matching process, while it brings new challenges of additional chunking overhead and low redundancy detection ratio. At the same time, current sync approaches are not aware of the challenges brought by different network conditions. This paper proposes NetSync, a network adaptive deduplication-inspired lightweight delta sync approach for cloud storage services, to address the above challenges facing combining the CDC technique with the traditional `rsync`-based approaches. The critical contributions of NetSync are: (1) Developing a fast, weak hash called FastFP by piggybacking on hashes generated from the chunking process of FastCDC; (2) 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 large file synchronization and high-bandwidth environment; (3) Proposing a network adaptive design to automatically choose appropriate compressors and CDC parameters to minimize the sync time. Evaluation results, driven by benchmark and real-world datasets, demonstrate that our solution NetSync performs $2\times$–$10\times$ faster and scales to 30%–50% more concurrent clients than the state-of-the-art approaches.

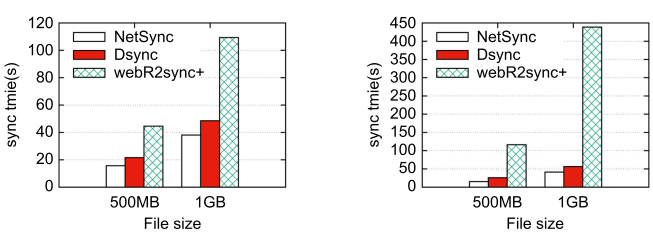


Fig. 20. Sync time of the three sync approaches for the large files (1GB) under the high-bandwidth environment (1Gbps).

Acknowledgments

We are grateful to the anonymous reviewers for their insightful comments and feedback on this work. We also appreciate Lingfeng Xiang, Yuan He, and Junwei Xu for their discussion and technical support. The preliminary manuscript appeared in the proceedings of IEEE MSST 2020. In this journal version, we introduced more techniques to further improve delta sync performance and included additional measurement results from our analysis and testbed experiments.

REFERENCES

[1] Dropbox, INC., "Dropbox," 2014. Accessed: Apr. 04, 2019. [Online]. Available: https://www.dropbox.com/

[2] Google, INC., "Google drive," 2019. Accessed: Apr. 04, 2019. [Online]. Available: https://www.google.com/drive/

[3] Apple, INC., "iCloud," 2019. Accessed: Apr. 04, 2019. [Online]. Available: https://www.icloud.com/

[4] I. Drago, E. Bocchi, M. Mellia, H. Slatman, and A. Pras, "Benchmarking personal cloud storage," in *Proc. Conf. Internet Meas. Conf.*, 2013, pp. 205–212.

[5] Z. Li *et al.*, "Towards network-level efficiency for cloud storage services," in *Proc. Conf. Internet Meas. Conf.*, 2014, pp. 115–128.

[6] I. Drago, M. Mellia, M. M. Munafo, A. Sperotto, R. Sadre, and A. Pras, "Inside Dropbox: Understanding personal cloud storage services," in *Proc. Conf. Internet meas. Conf.*, 2012, pp. 481–494.

[7] Q. Zhang *et al.*, "DeltaCFS: Boosting delta sync for cloud storage services by learning from NFS," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst.*, 2017, pp. 264–275.

[8] S. Wu, L. Liu, H. Jiang, H. Che, and B. Mao, "PandaSync: Network and workload aware hybrid cloud sync optimization," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst.*, 2019, pp. 282–292.

[9] H. Xiao, Z. Li, E. Zhai *et al.*, "Towards web-based delta synchronization for cloud storage services," in *Proc. 16th USENIX Conf. File Storage Technol.*, 2018, pp. 155–168.

[10] I. Mohiuddin *et al.*, "Secure distributed adaptive bin packing algorithm for cloud storage," *Future Gener. Comput. Syst.*, vol. 90, pp. 307–316, 2019.

[11] Y. He *et al.*, "Dsync: A lightweight delta synchronization approach for cloud storage services," in *Proc. 35th Symp. Mass Storage Syst. Technol.*, 2020, pp. 1–14.

[12] M. Abebe, K. Daudjee, B. Glasbergen, and Y. Tian, "EC-store: Bridging the gap between storage and latency in distributed erasure coded systems," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst.*, 2018, pp. 255–266.

[13] A. K. Singh, X. Cui, B. Cassell, B. Wong and K. Daudjee, "MicroFuge: A middleware approach to providing performance isolation in cloud storage systems," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst.*, 2014, pp. 503–513.

[14] W. Xia *et al.*, "A comprehensive study of the past, present, and future of data deduplication," *Proc. IEEE*, vol. 104, no. 9, pp. 1681–1710, Sep. 2016.

[15] A. Muthitacharoen, B. Chen, and D. Mazieres, "A low-bandwidth network file system," in *Proc. 18th ACM Symp. Oper. Syst. Princ.*, 2001, pp. 174–187.

[16] W. Xia *et al.*, "FastCDC: A fast and efficient content-defined chunking approach for data deduplication," in *Proc. USENIX Conf. USENIX Annu. Tech. Conf.*, 2016, pp. 101–114.

[17] J. W. Hunt and M. D. MacIlroy, *An Algorithm for Differential File Comparison*. New York, NY, USA: Murray Hill, 1976.

[18] D. G. Korn and K.-P. Vo, "Engineering a differencing and compression data format," in *Proc. Gen. Track Annu. Conf. USENIX Annu. Tech. Conf.*, 2002, pp. 219–228.

[19] B. C. Housel and D. B. Lindquist, "WebExpress: A system for optimizing web browsing in a wireless environment," in *Proc. 2nd Annu. Int. Conf. Mobile Comput. Netw.*, 1996, pp. 108–116.

[20] A. Tridgell *et al.*, "The rsync algorithm," 1996, [Online]. Available: https://rsync.samba.org/tech report/tech report.htm

[21] D. Teodosiu, N. Bjørner, Y. Gurevich, M. Manasse, and J. Porkka, "Optimizing file replication over limited-bandwidth networks using remote differential compression," *Microsoft Res.*, 2006.

[22] J. MacDonald, "File system support for delta compression," Ph.D. dissertation, Citeseer, Princeton, NJ, USA, 2000.

[23] D. Trendafilov, N. Memon, and T. Suel, "zdelta: An efficient delta compression tool," Citeseer, pp. 1–16, 2002.

[24] W. Xia, H. Jiang, D. Feng, L. Tian, M. Fu, and Y. Zhou, "Ddelta: A deduplication-inspired fast delta compression approach," *Perform. Eval.*, vol. 79, pp. 258–272, 2014.

[25] Y. Cui, Z. Lai, X. Wang, and N. Dai, "QuickSync: Improving synchronization efficiency for mobile cloud storage services," *IEEE Trans. Mobile Comput.*, vol. 16, no. 12, pp. 3513–3526, Dec. 2017.

[26] Seafile, INC., "Seafile: Enterprise file sync and share platform with high reliability and performance." Accessed: May 18, 2019. [Online]. Available: https://www.seafile.com/en/home

[27] T. Andrew *et al.*, "rsync." Accessed: Aug. 01, 2019. [Online]. Available: https://rsync.samba.org/

[28] Z. Li *et al.*, "Efficient batched synchronization in dropbox-like cloud storage services," in *Proc. ACM/IFIP/USENIX Int. Conf. Distrib. Syst. Platforms Open Distrib. Process. Middleware*, 2013, pp. 307–327.

[29] Y. Cui, Z. Lai, X. Wang, and N. Dai, "QuickSync: Improving synchronization efficiency for mobile cloud storage services," *IEEE Trans. Mobile Comput.*, vol. 16, no. 12, pp. 3513–3526, Dec. 2017.

[30] A. Tridgell, "Efficient algorithms for sorting and synchronization," Australian Nat. Univ., pp. 1–115, 1999.

[31] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM J. Res. Develop.*, vol. 31, no. 2, pp. 249–260, 1987.

[32] F. Zhang, J. Zhai, X. Shen, O. Mutlu and X. Du, "POCLib: A high-performance framework for enabling near orthogonal processing on compression," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 2, pp. 459–475, Feb. 2022.

[33] R. E. Bryant, O. David Richard, and O. David Richard, *Computer Systems: A Programmer's Perspective*. Englewood Cliffs, NJ, USA: Prentice Hall, 2003.

[34] B. Zhu, K. Li, and R. H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," in *Proc. 6th USENIX Conf. File Storage Technol.*, 2008, pp. 1–14.

[35] "Librespeed-speedtest," 2019. [Online]. Available: https://librespeed.org/

[36] D. Woolston, "The xmlhttprequest object," *Pro Ajax and the. NET 2.0 Platform*, pp. 41–64, 2006.

[37] T. Yoshino, "Compression extensions for websocket," *Internet Eng. Task Force Request Comments*, vol. 7692, pp. 2070–1721, 2015.

[38] H. Xiao, Z. Li, E. Zhai *et al.*, "Practical web-based delta synchronization for cloud storage services," in *HotStorage 17*, 2017.

[39] S. Deorowicz, "Silesia corpus," [Online]. Available: https://bit.ly/2xN3hgZ/

[40] S. Deorowicz, "Universal lossless data compression algorithms," *Philosophy Diss. Thesis Gliwice*, 2003, Art. no. 38.

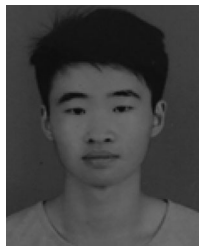
[41] V. Tarasov, A. Mudrankit, W. Buik, P. Shilane, G. Kuenning, and E. Zadok, "Generating realistic datasets for deduplication analysis," in *Proc. USENIX Conf. Annu. Tech. Conf.*, 2012, Art. no. 24.

[42] V. Christlein, C. Riess, J. Jordan, C. Riess and E. Angelopoulou, "An evaluation of popular copy-move forgery detection approaches," *IEEE Trans. Inf. Forensics Secur.*, vol. 7, no. 6, pp. 1841–1854, Dec. 2012.

[43] Enron mail dataset. [Online]. Available: https://bitly.com/2XSUbu2/

[44] C. Henke, C. Schmoll, and T. Zseby, "Empirical evaluation of hash functions for packetid generation in sampled multipoint measurements," in *Proc. Int. Conf. Passive Act. Netw. Meas.*, 2009, pp. 197–206.

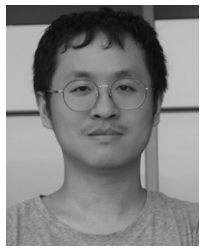
Wen Xia (Member, IEEE) received the PhD degree in computer science from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2014. He is currently an associate professor with the School of Computer Science and Technology, Harbin Institute of Technology, Shenzhen, China. His research interests include data reduction, storage systems, and cloud storage. He has published more than 60 papers in major journals and conferences, including *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Computers*, *Proceedings of the IEEE*, USENIX ATC, FAST, Hot Storage, MSST, DCC, IPDPS, INFOCOM, ICDCS, etc.



Can Wei received BS degree in computer science from the Harbin Institute of Technology, Weihai, China, in 2020. He is currently working toward the MS degree with the Department of Computer Science and Technology, Harbin Institute of Technology, Shenzhen, China. His research interests include data sync, cloud storage and cloud computing.

Zhenhua Li (Member, IEEE) received the BS and MS degrees in computer science and technology, from Nanjing University, Nanjing, China, in 2005 and 2008, respectively, and the PhD degree from Peking University, Beijing, China, in 2013. He is currently an associate professor with the School of Software and BNRist, Tsinghua University. His research interests include cloud computing/storage/ download, big data analysis, content distribution, and mobile internet.

Xuan Wang received the PhD degree in computer sciences from the Harbin Institute of Technology, Harbin, China, in 1997. He is currently a professor and dean with the School of Computer Science and Technology, The Harbin Institute of Technology, Shenzhen, China. His research interests include artificial intelligence, computer network security, computational linguistics, and computer vision. He has published more than 120 academic papers in major journals and conferences.



Xiangyu Zou is currently working toward the PhD degree majoring in computer science with the Harbin Institute of Technology, Shenzhen, China. His research interests include data deduplication, lossy compression and storage systems. He has published several papers in major journals and conferences including FAST, ICDE, *IEEE Transactions on Parallel and Distributed Systems*, ICPP, Cluster, MSST, HotEdge, etc.