DeltaCFS: Boosting Delta Sync for Cloud Storage Services by Learning from NFS

Quanlu Zhang*, Zhenhua Li†, Zhi Yang*, Shenglong Li*, Shouyang Li*, Yangze Guo*, and Yafei Dai*‡
*Peking University †Tsinghua University
‡Institute of Big Data Technologies Shenzhen Key Lab for Cloud Computing Technology & Applications
{zql, yangzhi, lishenglong, lishouyang, guoyangze, dyf}@net.pku.edu.cn, lizhenhua1983@tsinghua.edu.cn

***Abstract*—Cloud storage services, such as Dropbox, iCloud Drive, Google Drive, and Microsoft OneDrive, have greatly facilitated users' synchronizing files across heterogeneous devices. Among them, Dropbox-like services are particularly beneficial owing to the *delta sync* functionality that strives towards greater network-level efficiency. However, when delta sync trades computation overhead for network-traffic saving, the tradeoff could be highly unfavorable under some typical workloads. We refer to this problem as the *abuse of delta sync*.**

To address this problem, we propose DeltaCFS, a novel file sync framework for cloud storage services by learning from the design of conventional NFS (Network File System). Specifically, we combine delta sync with NFS-like file RPC in an adaptive manner, thus significantly cutting computation overhead on both the client and server sides while preserving the network-level efficiency. DeltaCFS also enables a neat design for guaranteeing causal consistency and fine-grained version control of files. In our FUSE-based prototype system (which is open-source), DeltaCFS outperforms Dropbox by generating up to 11× less data transfer and up to 100× less computation overhead under concerned workloads.

I. Introduction

Cloud storage services, such as Dropbox [1], iCloud Drive, Google Drive, and Microsoft OneDrive, have greatly facilitated users' synchronizing files across heterogeneous devices. Lying at the heart of these services is the data synchronization (sync) operation which automatically maps the updates to users' local file systems onto the back-end cloud servers in a timely manner [2].

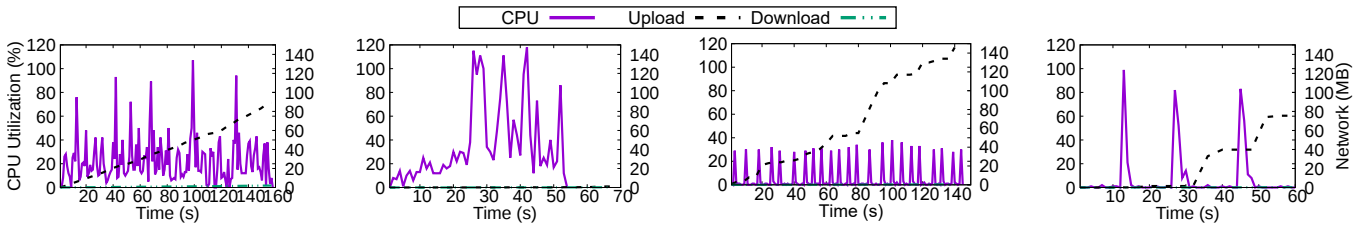
Among today's cloud storage services, Dropbox-like services (*e.g.,* iCloud Drive, SugarSync, and Seafile [3]) are particularly favorable owing to their *delta sync* functionality that strives towards greater network-level efficiency [4], [5]. Specifically, for synchronizing a file's new version on the client, only the incremental parts relative to the base version on the server are extracted and uploaded. This can reduce considerable network data transfer compared with simply uploading the full content of the new-version file (as used by Google Drive, OneDrive, *etc.*). Thus, it is especially useful and economical for operations through wide area networks [2].

However, with increasingly prevalence of cloud storage services, the upper-layer applications are getting more complex and comprehensive, where delta sync can become not only ineffective but also cumbersome. Such applications include SoundHound [6], 1Password [7], continuity [8], [9], and so forth [10], [11]. By carefully examining these applications, we find a common root cause that undermines the effect of delta sync: these applications heavily rely on *structured data* which are stored in tabular files (*e.g.,* SQLite [12] files), rather than simple textual files (*e.g.,* .txt, .tex, and .log files). For example, today's smartphones (like iPhone) regularly synchronize local data to the cloud storage (like iCloud [13]), where a considerable part of the synchronized is tabular data [14].

Confronted with the abovementioned situation, delta sync exhibits poor performance, because it often generates intolerably high computation overhead when dealing with structured data. In detail, executing delta sync on a file requires scanning, chunking, and fingerprinting on the file, thus consuming substantial CPU resources [4], [15]. As a result, blindly applying delta sync to all types of file updates suffers from the one-size-fit-all trap, and this problem is referred to as the *abuse of delta sync* in cloud storage services.

To address the problem, in this paper we design DeltaCFS, a novel file sync framework for modern cloud storage services by learning from the seemingly antique while still widely-used NFS (*i.e.,* Network File System). DeltaCFS stems from a practical insight that for certain types of file updates (*e.g.,* appending bytes to a file, flipping bytes in a file, and removing bytes at the end of a file), delta sync is largely unnecessary. In these cases, directly intercepting file operations (which we refer to as NFS-like file RPC) turns out to be the most efficient way to precisely obtain the modified parts. Hence, by leveraging the hints of typical file update patterns, we adaptively apply two distinct incremental file sync approaches, *i.e.,* delta sync and NFS-like file RPC. The key novelty lies in an elegant combination of the two approaches in a runtime manner, using a customized relation table that extracts the *invariants* of file update patterns. For example, in transactional update (see § II-B) both old-version and new-version files are always kept until the update is done. Our efforts can significantly cut the computation overhead on both client and server sides; meanwhile, the network-level (traffic usage) efficiency is either preserved or optimized.

DeltaCFS also enables a neat design for guaranteeing causal consistency and fine-grained version control of files. File sync should offer basic guarantees of data integrity and consistency, where version control is a plus appealing to users. However, complicated data sync operations among the cloud and clients can make data integrity and consistency fairly vulnerable, since corrupted and inconsistent data on any client is likely



(a) Dropbox (b) Dropbox (c) Seafile (d) Seafile

Fig. 1. **Client resource consumption.** (a)(c) *Synchronizing a Microsoft Word file (12MB), this file is saved 23 times.* (b)(d) *Synchronizing a SQLite file (130MB) which stores chat history, the file is from a popular chat application. During the test, this file is modified 4 times (composed of 85 write operations) with 688KB changed in total.*

to be quickly propagated to all other relevant clients [16]. Based on DeltaCFS, we provide causal consistency for data uploading and guarantee data integrity and consistency with lightweight mechanisms.

We have built a prototype system of DeltaCFS [1] based on FUSE and comprehensively evaluated its performance on both PCs and mobile phones. Compared to Dropbox, up to 11× less data are transmitted by DeltaCFS in the presence of representative workloads, and the savings of computation resources on the client side range from 91% to 99%. On the server side, DeltaCFS also incurs low overhead, 4× to 30× lower than Seafile, an open-source state-of-the-art cloud storage service. [2] Finally, it is worth noting that although DeltaCFS is currently implemented in the user space, it could also be implemented in the kernel, *e.g.,* as an enhanced mid-layer working on top of virtual file system (VFS). Then, the computation overhead of DeltaCFS can be further reduced.

Our contribution can be summarized as follows:

- We pinpoint the "abuse of delta sync" problem in Dropbox-like cloud storage services and dig out its root cause (§ II).
- We design a novel file sync framework which efficiently synchronizes various types of file updates by combining delta sync and NFS-like file RPC (§ III). In particular, we leverage a customized relation table to identify different file update patterns in order to conduct data sync in an adaptive and run-time manner. Also, we utilize lightweight mechanisms to guarantee data integrity/consistency and facilitate version control.
- We build a prototype system of DeltaCFS and compare its performance with representative systems including Dropbox, Seafile, and NFS. Comprehensive evaluation results illustrate the efficacy and efficiency of DeltaCFS (§ IV).

II. Background and Motivation

A. Delta Encoding

Delta encoding was researched a lot for efficient data synchronization on wide area network [17]–[20]. There are two representative algorithms: rsync [4] and CDC [5]. Rsync divides files into fixed-size blocks, and computes rolling/strong checksums to identify the same blocks. Notice that file updates could lead to the shift of data in each fixed-size chunk, so rsync has to redo the checksum computation each time a file is modified, consuming considerable CPU resource. CDC uses **c**ontent **d**efined **c**hunking to detect the boundary of blocks, thus only needs to compute the checksums of changed blocks. But its network efficiency is not as good as rsync due to large chunk size for low overhead of maintaining chuck checksums.

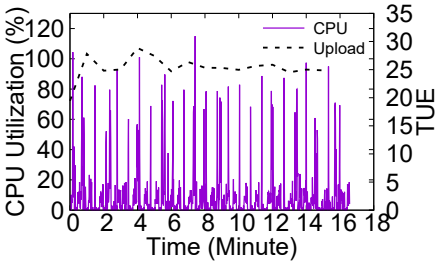


Fig. 2. **Synchronizing WeChat's data through Dropsync on Samsung Galaxy Note3.** *WeChat [23] is a popular chat application. Dropsync [24] is an autosync tool of Dropbox on mobiles. We set WeChat's data folder as Dropsync's sync folder. TUE (**T**raffic **U**sage **E**fficiency) is total data sync traffic divided by data update size [2].*

We examine the efficiency of the above methods in cloud-based data sync systems by investigating two popular instances, *i.e.,* Dropbox and Seafile, which employ rsync and CDC, respectively [21], [22]. As shown in Figure 1(c)(d), Seafile shows moderate CPU consumption, but it uploads a large amount of data given the large chunk size (default value is 1MB). In contrast, Dropbox in Figure 1(a)(b) has much better network efficiency than Seafile, but incurs a high cost of CPU usage. Thus, Dropbox cannot apply rsync to save network traffic for mobile clients. As shown in Figure 2, the traffic utilization is low when we use Dropbox's mobile client to timely synchronize the data of WeChat to the cloud. Also, we see that, even without rsync, the average CPU usage still reaches 9%, and the frequent spikes in CPU usage keep the device staying in high power-consumption mode.

IO consumption is another intrinsic flaw of delta encoding algorithms. In order to obtain the modified content, the whole file has to be scanned. Though it is not shown in Figure 1(b), Dropbox issues over 700MB data read in that test. Excessive data read could be a more serious performance killer than CPU

[1]Both the source code and evaluation traces are available at https://github.com/QuanluZhang/DeltaCFS.

[2]With respect to the server-side overhead, we do not compare the performance of DeltaCFS with that of Dropbox since the server side of Dropbox is generally opaque to us.

consumption, as it occupies disk bandwidth and consumes memory for disk cache. On mobile devices, more storage IO also means more energy waste [25], [26].

The quick takeaway is that CDC is not applicable due to its poor network performance, while rsync puts too much burden on host devices, which is unbearable for wimpy mobile devices. Motivated by these observations, our goal is to design an incremental data sync scheme with both high network efficiency and low CPU consumption, which is also lightweight enough for mobile situation.

B. Design Opportunities

WeChat: 1-2 create-write *f_journal*, 3 write *f*, 4 truncate *f_journal 0*
Word: 1 rename *f t0*, 2-3 create-write *t1*, 4 rename *t1 f*, 5 delete *t0*
gedit: 1-2 create-write *tmp*, 3 link *f f*∼, 4 rename *tmp f*

Fig. 3. **Typical operation sequences for updating files.** *They are from three popular applications on both desktop and mobiles. The numbers mean the sequence of the operations.*

To avoid the *abuse of delta sync*, we leverage the interception of file operations to cut down the CPU usage of delta encoding without sacrificing network efficiency. This design exploits the update patterns of files to do incremental sync. In particular, there are two typical update patterns: *in-place update* and *transactional update* [14].

For in-place update, write operation only affects a small part of the file with the remaining part untouched, such as SQLite file or log file. Figure 3 shows the typical operation sequence of SQLite in WeChat application, a journal file is first created and written, then the third operation (*i.e., 3 write f*) directly writes the changed data into the file *f*, which is exactly the incremental data for the synchronization. For this pattern, our basic idea is to intercept the write operation to directly obtain incremental data, which avoids the checksum computation of delta encoding and also has higher network efficiency than block level delta encoding[3]. We call it NFS-like file RPC.

The other update pattern is transactional update, which is widely used in various applications to avoid file corruption due to system crash. In transactional update, the new version of a file is first written to a temporary file rather than directly overwriting its previous version, then it uses an atomic operation such as *rename*, *link* to replace the previous version. Figure 3 illustrates the operations of this pattern used in Microsoft Word and gedit (a text editor in linux). For transactional update, intercepting written data does not work because any file modification leads to a rewrite of the whole file. For this pattern, our idea is to introduce a very lightweight version of rsync to get the modified content. In particular, the interception of file operations allows us to identify the relationship between the previous version and the new version of a file, which enables a direct comparison between them and thus prunes the high cost of computing strong checksum in rsync.

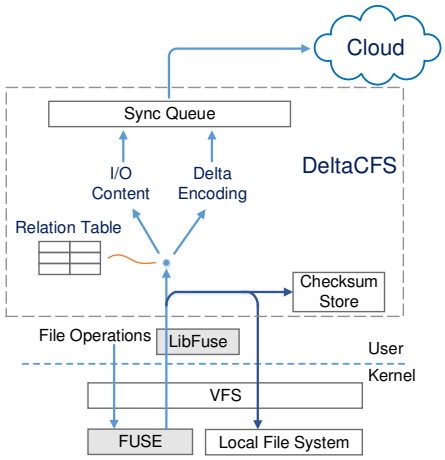


Fig. 4. **DeltaCFS architecture.** *The grey rectangles are modules of FUSE file system.*

III. DeltaCFS Design

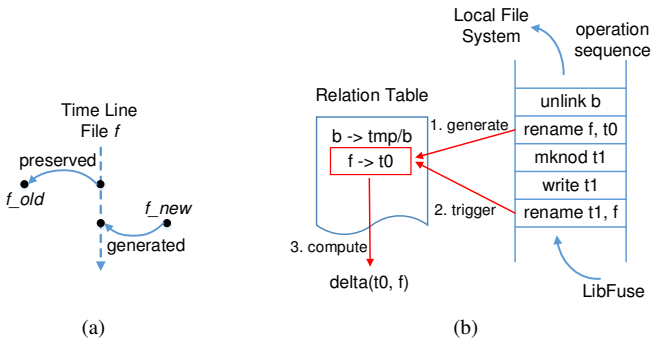
It is nontrivial to combine delta encoding and NFS-like file RPC into one framework, because delta encoding deals with files while NFS-like RPC deals with file operations, they reside in different layers. In order to intercept file operations, we base DeltaCFS on FUSE [27], a popular user space file system. The high level architecture is shown in Figure 4. DeltaCFS retrieves file operations from LibFuse. Before that, file operations are originally issued to FUSE kernel module through VFS and redirected back to LibFuse in user space. This is how FUSE works. All the operations at last are issued back to local file system.

In order to adaptively apply the two sync approaches, we maintain a relation table which triggers delta encoding when necessary. After this process, we get incremental data through either delta encoding or NFS-like file RPC, and enqueue these data to Sync Queue. Then, data in Sync Queue will be uploaded onto cloud. Here, the cloud side should be modified a little to support applying the incremental data generated by clients to the corresponding files on the cloud. Sync Queue is a key component in DeltaCFS, it facilitates the implementation of incremental data sync, making it possible to combine delta encoding and NFS-like file RPC seamlessly. We design a new fine-grained version control mechanism also based on Sync Queue. Moreover, in our design Sync Queue has another functionality which is guaranteeing causal consistency, while data integrity and crash consistency are guaranteed by Checksum Store in user space.

A. Relation Table

We need to identify files' update patterns correctly, in order to apply the most efficient sync approach for each file. Here, we apply NFS-like file RPC by default, and replace it with delta encoding when transactional update is identified.

[3]The delta is at least one data block (*e.g.,* 4KB in rsync) even though only 1 byte is modified



(a) (b)

Fig. 5. **Transactional update.** (a) *Invariant of transactional update.* (b) *The procedure of executing delta encoding based on relation table.*

TABLE I
OPERATING RELATION TABLE AND TRIGGERING DELTA ENCODING.

Create relation entry	1. *rename* operation 2. *unlink* operation
Remove relation entry	1. triggered delta encoding 2. time out (*e.g.,* 2 sec)
Trigger delta encoding	1. file's name equals src 2. file's name already exists

In order to identify transactional update, we extract invariants from this update pattern as shown in Figure 5(a). Since transactional update is mainly for tolerating file corruption caused by system/application crash, one rule it has to follow is that the file's old version cannot be deleted before its new version is safely written. More specifically, the file's old version is first preserved and very shortly the file is created again with its new version atomically (*e.g., rename*). Therefore, by tracking relations between files, *i.e.,* maintaining a relation table, we can identify transactional update.

The relation table, in essence, tracks the transformation of files' names. Each entry in the table is a tuple of two file names (*i.e.,* $src \rightarrow dst$), subject to two rules: 1) src and dst was/is the same file's name; 2) dst exists while src does not. For instance, *rename a, b* generates a relation entry $a \rightarrow b$. This relation table is used to trigger delta encoding. When a file is created, if its name is the same as src of an entry in the relation table, delta encoding is triggered and executed between this file and that entry's dst. Figure 5(b) is an example using the operation sequence of Microsoft Word in Figure 3. The first *rename* generates a relation entry which indicates f's old version is now preserved using file name *t0*. When f is created again by the second *rename*, that relation entry triggers delta encoding between f and *t0*.

Table I shows how to operate relation table and the conditions of triggering delta encoding. Besides *rename*, *unlink* can also generate an relation entry. Specifically, if a file is removed (*i.e., unlink*), instead of immediately deleting this file, we move it into a dedicated folder (*e.g.,* tmp/) temporarily and create a relation for it. This is useful for certain cases. For example, transactional update could be accomplished with a combination of *link* and *unlink* (*e.g., link f f*$\sim$*, unlink f*). Moreover, a bad file update could be first deleting the file then writing its new version. However, if temporarily preserving the file would result in ENOSPC (*i.e.,* no space left on the device) or the deleted one is a directory, the deleted files will not be preserved and thus no relation entry is created. This has little impact on data sync efficiency because very large files or files in a deleted directory are unlikely to trigger delta encoding.

As demonstrated above relation entries can trigger delta encoding. Here delta encoding could also be triggered if the to-be-created file's name has already existed (*e.g., f* in the operation sequence of gedit in Figure 3), this is why there is no need to create relation for *link*. At last, relation entries will be removed from the relation table. If a relation entry triggers delta encoding, it will be removed, while if it does not, it will also be removed after a short period. Since a file update by operating system usually can be done within 1 second, the period can be empirically set in a range of 1 to 3 seconds.

In this new data sync framework, we can further reduce rsync's computational overhead. Note that when executing delta encoding we have both the file's old version and new version locally. Most delta encoding algorithms, such as rsync, are designed assuming the two files are on different machines, so they transmit and compare strong checksums to check whether two data blocks are the same rather than transmitting real data blocks. In our design, we use bitwise comparison to replace strong checksum. It can reduce a lot of computational cost of rsync, as its checksums should be recalculated every time a file is modified.

In-place update usually issues small writes, so NFS-like file RPC is the most efficient sync approach. Here, we further extend our design to deal with the case that in-place update changes a large portion of a file (*e.g.,* more than 50%) and delta encoding could further compress the changes. Since delta encoding is executed locally in DeltaCFS, file's old version is required. For in-place update we use a variant of physical undo logging [28] to preserve file's old version. Specifically, if a write operation is going to overwrite old data, we will copy the old data out before issuing the write operation. Thus, delta encoding can still be executed locally by recovering the file's old version. This has little impact on performance, because the data to be copied out are usually already cached in memory, no disk IO is required.

B. Sync Queue

NFS-like file RPC and delta encoding generate incremental data in different phases. The former generates incremental data during a file is being updated, while the latter generates incremental data after the file update is done. In DeltaCFS, NFS-like file RPC is applied on every file by default, which means written data are intercepted and obtained instantly when file operations are issued. If we upload the written data immediately when they are obtained, it would be useless to execute delta encoding again. So the corresponding written data should be replaced by the incremental data generated by delta encoding before uploading.

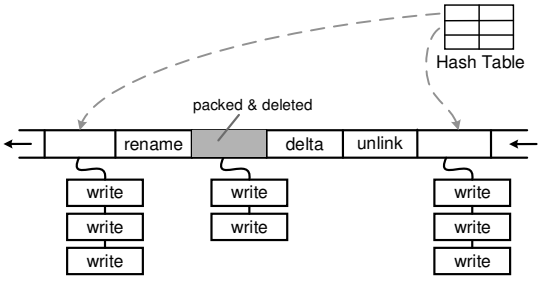


Fig. 6. **Sync Queue.** *The nodes in Sync Queue are uploaded with a delay (e.g., 3 sec).*

Sync Queue is designed to facilitate this operation as shown in Figure 6. Intercepted write operations are enqueued waiting for uploading. If delta encoding is triggered, the corresponding write will be removed from Sync Queue and the generated delta is enqueued instead. Write operations to the same file are linked to one node (called *write node*) in Sync Queue for easy deletion, batching and compression. These write nodes are indexed by a hash table. An incoming write finds its corresponding write node through the hash table. If the node does not exist, it creates a new one and appends it to Sync Queue.

Write node should be packed to be an immutable node if its corresponding file's state is changed, such as closed, renamed, deleted. Imaging if a write node is not packed and its file is renamed away, then a new file with the same name is created and written, the write operations would still be attached to that node, leading to corrupted file content. The nodes between rename and delta in Figure 6 reflect the operation sequence of Microsoft Word in Figure 3. The *write*s to *t1* are attached to the write node which is packed after *t1* is closed. Then this write node is deleted because of triggered delta encoding.

C. Version Control

Since DeltaCFS is an incremental data sync system, it relies on version control to apply incremental data on correct files, especially when multiple clients modify the same file. In our new data sync framework, version control should be carefully designed, to provide proper version granularity, and to minimize server's involvement for less network message passing.

Existing version control mechanisms are not applicable in DeltaCFS. Open-to-close versioning [28], which is commonly used in versioning file systems, does not support timely data sync well as a file can be opened for a long time. Versioning on every write is, however, too aggressive. In fact our Sync Queue design naturally supports versioning on each node in Sync Queue, which is a neat tradeoff between the above two versioning granularities.

The nodes in Sync Queue are usually incremental data which will generate a new version of a file. The problem is who assigns the version number to the nodes in Sync Queue. In existing Dropbox-like systems, version numbers are usually assigned by the server side, because the version is increased only when the new version of a file is uploaded to the cloud. There is almost no cost for the server side to response with a version number. However, in DeltaCFS the nodes in Sync Queue should be assigned version numbers when they are enqueued. If version numbers are retrieved from server each time a node is enqueued, DeltaCFS's performance would be degraded a lot due to high network latency.

Thus, rather than relying on servers, we outsource version assignment to clients. Each client has a monotonically increasing number of its own to version files. Different clients do not have to sync this number with each other even though they share files, since partial order is enough in cloud sync scenario. In order to make version numbers distinct among different clients, we add client ID to version numbers, *i.e.,* $<CliID, VerCnt>$. When uploading a node in Sync Queue, the base version and new version numbers are attached to it. If the base version is the same as the current version on the cloud, the node can be applied normally and the version on the cloud is updated to the new version.

In cloud sync scenario, users can always modify a local file without checking whether it is the latest version. If two clients share a file and edit this file simultaneously, a conflict occurs. This conflict can be easily detected through our version control mechanism, *i.e.,* the incremental data's base version does not equal to the latest version on the cloud. The server side employs the "first write wins" semantic to reconcile the conflict, which is often used in popular cloud sync services. It labels the first received one as the latest version and labels the second one as a conflict version. In DeltaCFS a file becoming a conflict version does not mean we have to drop the incremental data and transmit this file again. Since servers keep recent versions of files, the incremental data can still be applied to the proper file to generate the conflict version. We do not employ automatic merging such as three-way merge, because there are various types of files in desktop environment, automatic merging is only suited to plain text files.

D. Multi-client Data Sync

Multi-client sync can be efficiently supported by DeltaCFS. In DeltaCFS the client uploads the incremental data to the cloud, and the cloud applies those data to the corresponding files. Here, if this client *A* also shares these files with another client *B* (could be a user or a device), then client *B* is virtually equivalent to the cloud, which means the same incremental data can be directly sent to client *B* without additional computation. So in DeltaCFS when the cloud receives data from a client, besides storing the data it also forwards the data to other shared clients.

Conflicts could also occur on those shared clients. For example, a modification to a file by client *A* is forwarded to client *B* by the cloud, at the same time client *B* is modifying this file but has not uploaded the modification. The conflict reconciliation in this case is very similar to that on the cloud, so we do not repeatedly elaborate it.

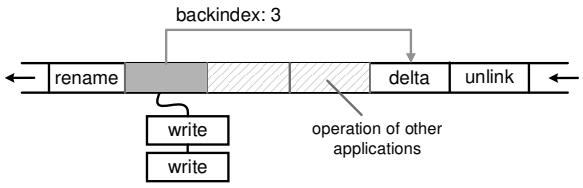


Fig. 7. **Backindex.** *The write node is labeled as deleted when delta is appended.*

E. Integrity and Consistency

Recent trend shows that data integrity and consistency are playing an increasingly important role in data sync scenario, mainly because of the extremely deleterious effect of unintentionally propagating corrupted/inconsistent data among all the data copies [10], [16]. Below we illustrate the great opportunities and convenience brought by our data sync framework to guarantee data integrity and consistency.

Data integrity and crash consistency. Several types of file systems have the ability of detecting corrupted data. However, relying on a specific file system [29], [30] makes the data sync system not portable. Unfortunately, it is also not possible to detect corrupted data if residing above file system such as those Dropbox-like systems, because they cannot tell whether a file modification is from users or from corruption. In our data sync framework, we can maintain checksums of data blocks to detect data corruption, because all file operations go through our system. The checksum in DeltaCFS is independent of file system's internal data structure and data layout.

We partition each file into fixed-length blocks (*e.g.,* 4KB), and generate a checksum for each block. The checksums are stored in a key-value store, *e.g.,* LevelDB [31] in DeltaCFS. When a file is modified, DeltaCFS will update checksums of the involved data blocks accordingly. When a file is read, the data blocks will be verified using the checksums. If corrupted data are detected, we use the correct data on the cloud to recover. Moreover, since rsync also uses the same way to split a file, we can reuse the rolling checksum in rsync as the block checksum, which further reduces the computational cost.

The checksum in DeltaCFS is also used to detect crash inconsistency. In the file system literature, crash inconsistency means inconsistency between file system's metadata and data blocks due to system crash [16], [32], [33]. Since our checksum design still resides above file system, it cannot detect all crash inconsistency within file system. Instead, it is a best-effort mechanism to find out the files in incorrect intermediate state rendered by system crash, which provides enough guarantee for users with little overhead. Normally, after system crash we check every recently modified files by comparing their data blocks with their checksums. If inconsistency is detected, the correct version will be pulled from the cloud, and we let users to decide which version to keep.

Causal consistency. Eventual consistency of data sync renders a lot of anomalies. For example, a photo is created before its thumbnail, if this order is not obeyed when uploading files, it is possible that on a linked device the thumbnail is the latest while the photo is not [16], [34]. Furthermore, object data are created before they are indexed in tabular files, eventual consistency can induce the situation that an object indexed in tabular files does not exist [10]. Unfortunately, these anomalies could lead to data loss or unpredictable behaviors of applications. Thus, causal consistency is more suitable for cloud data sync scenario.

In DeltaCFS causal consistency can be guaranteed in an efficient manner by leveraging Sync Queue. Obviously, if Sync Queue strictly follows the FIFO rule, it could natively guarantee causality. However, the optimizations on Sync Queue such as write node, delta encoding, violate FIFO. For example, there is an operation sequence: create *a*, create *b*, create *c*, delete *a*. If *a* is deleted from Sync Queue before it is uploaded, it is possible for the cloud to only have *b* without *a* and *c*, which is impossible for a strict FIFO queue. To solve this problem, we could take snapshot on Sync Queue periodically like ViewBox [16], and the cloud applies these snapshots in a transactional manner. However, it has two problems, first, when the snapshot is taken, no more changes are allowed on it even though some nodes can be deleted; second, it is not easy to set the snapshot interval, too short degrades performance while too long may induce the loss of latest update.

So we design *backindex* to preserve causality in Sync Queue. For Sync Queue the violation of causality is induced by operating non-tail nodes (*e.g.,* delete, batching *write*s), we add a backindex on the node which is operated again after it has been enqueued. The backindex points to the position where that operation should be appended if Sync Queue follows strict FIFO rule, *i.e.,* the tail of Sync Queue when that operation occurs. For example in Figure 7, when the write node is deleted due to delta encoding, the node has a backindex pointing to that delta node. All the operations covered by the backindex should be applied transactionally on the cloud. If there is interleaving between two backindexes, we merge them to make sure that the consecutive nodes covered by those backindexes are applied in a transactional manner.

Note that in DeltaCFS several files' update might be encapsulated by one backindex, we view them as an atomic operation. Specifically, if one file in this atomic operation has conflict, we label all the files in this operation as conflict, and let users to resolve conflicts manually, for example picking the version they want or merging different versions.

IV. Evaluation

In this section, we study the performance of DeltaCFS under different types of workloads on both PCs and mobiles, and compare it with alternative solutions and popular commercial products to answer several questions: What is DeltaCFS's CPU consumption on both client and server sides? How much network bandwidth is saved by DeltaCFS compared to other solutions? Does DeltaCFS impact local file read/write performance? What is DeltaCFS's ability of guaranteeing data integrity and consistency?

We implemented DeltaCFS with 2858 lines C code for the client side and 435 lines C code for the server side. The librsync library is modified to replace strong checksum (*i.e.,* MD5) with bitwise comparison. We apply lock-free queue technique [35] to implement Sync Queue. The server side only has basic functionality for data sync. Functionalities such as authentication, load balancing are out of the scope of this paper. We use OpenSSL to encrypt all the transmitted messages between the client and server sides.

A. Evaluation Methodology

For the evaluation, unless specified otherwise, we run the experiments on two instances of Amazon EC2 m4.xlarge which is equipped with 4 vCPU (Intel Xeon E5-2676 v3), 16GB memory, and running Ubuntu 14.04 (kernel 3.13). One acts as client and the other is server. We call experiments in this setting "experiments on PC". In order to evaluate the performance of DeltaCFS on mobiles, we use Samsung Galaxy Note3 sn9009 which runs android 4.3 (kernel 3.4) as a client to run experiments. This setting is called "experiments on mobile".

We use two artificial traces and two collected real-world traces, which are typical and diverse enough to cover most usage scenarios. The two artificial traces are append write (40 append operations, each append is around 800KB, the final size of the file is 32MB) and random write (40 write operations to a 20MB file, each write is 1010 bytes) respectively, the interval of the writes are 15 sec. We collected real-world traces from Microsoft Word and Tencent WeChat respectively, using a similar method to [14]. Specifically, we use a loopback user-space file system [4] to collect file operations including the content of the written data. The Word trace is collected when we edit and save a Word document 61 times with its size changing from 12.1MB to 16.7MB. In the WeChat trace, the SQLite file which stores chat history is modified 373 times, and its size changes from 131MB to 137MB. In all the experiments below, we perform 5 runs and report the average numbers.

We compare DeltaCFS with three different sync solutions: Dropbox, Seafile and NFSv4 [37]. For Dropbox, we use its linux version 3.12.5. The client version of Seafile we use is 4.3.2 and the server version is 5.0.2. For experiments on mobiles, instead of comparing with Dropbox Android app which only supports syncing data manually, we compare DeltaCFS with Autosync Dropbox (*i.e.,* Dropsync [24], version 2.7.12) whose backend storage is still Dropbox, but it supports automatic data sync like Dropbox desktop client. Seafile is not compared on mobiles because it does not support automatic data sync and there is no third-party application to do it.

B. CPU Consumption

1) Experiments on PC: The CPU overhead of the four solutions are shown in Table II. On the client side, Dropbox consumes considerable CPU resource compared to other solutions. This is mainly induced by its delta encoding mechanism. Rsync is CPU intensive, and it is very likely that Dropbox offloads checksum recalculation to the client side [38] which means the client rather than the server recalculates checksums of the updated files. Though this reduces the burden of the server, it further overwhelms the client. Other mechanisms in Dropbox also generate some CPU overhead. For example, Dropbox applies deduplication in 4MB granularity [2], which imposes certain computation overhead. Though the deduplication perfectly works for simple data upload, it does not perform well in file editing scenario, in which file content usually shifts for a certain offset [38]. Dropbox also employs network data compression, which consumes some CPU resource. As will be shown in § IV-C, though DeltaCFS does not apply data compression, it shows high network efficiency, thus, the CPU resource used by data compression can be saved. Seafile's low CPU overhead benefits from the delta encoding it uses (*i.e.,* CDC) and its relatively large chunk size *i.e.,* 1MB.

DeltaCFS, on the other hand, presents the lowest CPU usage. For append write and random write, delta encoding is not necessary, DeltaCFS directly retrieves and uploads the written data, while Dropbox and Seafile spare no effort to scan files and calculate delta. So DeltaCFS shows an order of magnitude less CPU overhead even compared to Seafile. For Word trace, though DeltaCFS also applies rsync, our optimization of the rsync algorithm and triggering rsync execution at the right time reduce a lot of CPU overhead. Dropbox performs not as good as we thought, mainly because its delta encoding is triggered by file modification events (*i.e., inotify* [39]) which occurs much more frequently than our relation triggered delta encoding. For the WeChat trace, the advantage of DeltaCFS is very high. Since files in the WeChat trace is larger than that in the other three traces while every update is relatively small, both Dropbox and Seafile consume higher CPU resource to generate delta. On the contrary, DeltaCFS show the opposite trend and its CPU overhead is extremely low, as delta encoding is not applied.

On the server side, DeltaCFS shows very low CPU consumption, the reason is that we have minimized the overhead on the DeltaCFS server, it only needs to apply incremental data generated by clients. Though a full blown server may consume more CPU resource, we believe it cannot be high, because functions such as authentication, high concurrency support, only induce limited CPU overhead. Seafile also imposes relatively low CPU overhead on the server as the server also does not have to calculate chunk checksums. When executing CDC, the checksums for the new chunks will be calculated on the client anyway and can be directly sent to the server. For Word trace, the CPU overhead of NFS is around twice as large as Seafile, because though NFS server does not compute any checksum, it has to send and receive large amount of data through network, as will be illustrated in § IV-C, which also consumes a lot of CPU resource. For WeChat trace, NFS consumes a few CPU cycles due to much less data transmission.

[4]We use Dokan [36] for Windows.

TABLE II

CPU USAGE OF DIFFERENT SYNC SOLUTIONS. *The first four rows are the experiments on PC (i.e., EC2 instance), while the last two rows are the experiments on mobile (i.e., Note3). We are unable to measure Dropbox server's CPU usage. NFS client's CPU usage is also skipped as they are callback functions in kernel. The unit is CPU tick.*

Solutions	Append write		Random write		Word trace		WeChat trace	
	Client	Server	Client	Server	Client	Server	Client	Server
Dropbox	2726	–	3403	–	9787	–	13463	–
Seafile	476	60	540	61	1060	437	3127	247
NFSv4	–	6	–	4	–	884	–	25
DeltaCFS	56	5	42	3	858	89	83	8
Dropsync	28880	–	28048	–	21178	–	25356	–
DeltaCFS	817	8	466	4	7995	176	1141	11

2) Experiments on mobile: The numbers are shown in the last two rows in Table II. Since a CPU tick in different types of CPUs has different computational power, those numbers are not comparable with the numbers in the first four rows in Table II. For append write and random write, Dropsync consumes around 34-59X higher CPU resource than DeltaCFS, because though it does not employ rsync, it has to load the file from disk and transmit the whole file through network evey time the file is modified. Since the bandwidth of wide area network is very low, Dropsync keeps transmitting data during the whole experiment. Word trace and WeChat trace take several minutes to finish, during this period files are uploaded several times, so the CPU consumption is also very high. DeltaCFS shows much lower CPU consumption for Word trace though we apply rsync.

C. Network Transmission

During measuring CPU consumption of different solutions using various traces, we also measured their data transmission.

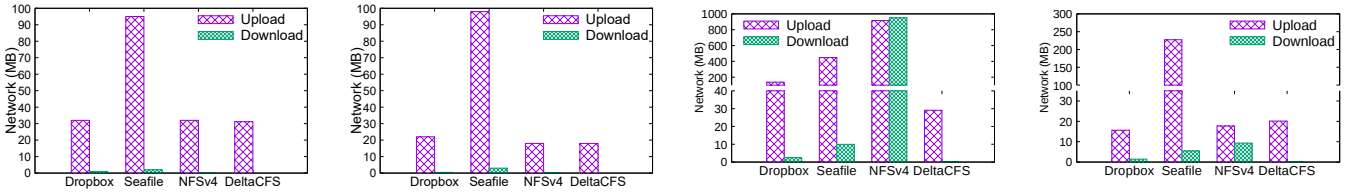
1) Experiments on PC: The results on PCs are shown in Figure 8. For append write, Dropbox, NFSv4 and DeltaCFS present similar upload traffic. It is obvious that NFSv4 and DeltaCFS have similar performance. Dropbox shows good performance due to its small chunk size, while Seafile's chunk size is too large, so it has poor network performance. Random write in Figure 8(b) shows similar numbers to append write. Here Dropbox uploads 4MB more data than NFSv4 and DeltaCFS, because every random write is 1010 bytes while Dropbox's chunk size is 4KB.

Figure 8(c) shows the result of Word trace. We tuned the replay of Word trace for Dropbox by increasing the latency between consecutive updates in order to get its best performance, otherwise Dropbox would directly uploads files without using rsync, which transmits 5 times larger than the number shown in Figure 8(c). However, the best performance of Dropbox still shows nearly 200MB data transmission. The reason is that it employs 4MB deduplication and rsync is only applied within the 4MB block, impacting the effect of delta encoding a lot [38]. Though Seafile shows relatively low CPU overhead, it does not perform well on the network usage, about 470MB data are transmitted from client to server due to large chunk size. NFS uploads extremely large amount of data, since it sends all the write operations. It is surprising that the NFS server also sends similar amount of data back to the client though there is no *read* operation in the trace. This is because Word first writes the file's new version in a temporary file *tmp* and then renames *tmp* to its original name *f*. Though *tmp*'s content is cached on the client, *f*'s content becomes stale, so its new content will be retrieved from the server again [40]. DeltaCFS uploads much less data than other solutions as it employs rsync on the whole file. There is almost no data transmitted from server to client, since the generation of incremental data does not require the involvement of servers. Dropbox also performs well on the download direction due to the offloading of checksum calculation from server to client as mentioned above, which also avoids downloading checksums.

Figure 8(d) shows the result of WeChat trace. Seafile still keeps its large data transmission, again due to large chunk size. Dropbox, however, uploads much less data, because there is no data shift in SQLite files, 4MB deduplication works well in this case. And as we expected, NFS performs well. It still downloads some data because of those non-aligned data *write*s, in which case the data block is first retrieved from the server, *i.e.,* the *fetch-before-write* problem [41]. DeltaCFS presents similar amount of uploaded data to NFS, and it is a little higher because DeltaCFS has to send some control information such as files' versions.

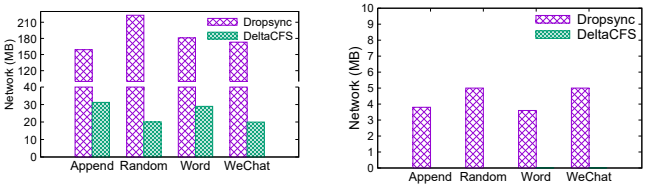
To further understand the performance numbers of WeChat trace, we also use rsync algorithm to figure out how large the incremental data would be for this trace, the result is around 30MB which is larger than Dropbox and DeltaCFS. Dropbox shows much lower amount of data than this number, we suspect it applies data compression (*e.g.,* Snappy [42]). The reason why DeltaCFS transfers less data than this number is that the delta encoding granularity of rsync is 4KB while the file modifications in the WeChat trace are usually smaller than 4KB, in which case directly sending IO operations is more efficient.

2) Experiments on mobile: Figure 9(a) shows the upload traffic on the mobile phone. For append write and random write, Dropsync uploads more than 150MB data, this is consistent with its CPU consumption. The size of uploaded data is smaller in append write than in random write because the file in append write is increased from 0 byte, smaller data



(a) Append Write (b) Random Write (c) Word Trace (d) WeChat Trace

Fig. 8. **Network traffic of experiments on PC.**



(a) Upload Traffic (b) Download Traffic

Fig. 9. **Network traffic of experiments on mobile.**

TABLE III
COMPARISON OF PERFORMANCE ON MICROBENCHMARKS. *DeltaCFSc means DeltaCFS with checksum enabled. The unit is MB/s.*

Workload	Native	FUSE	DeltaCFS	DeltaCFSc
Fileserver	116.0	114.7	78.3	66.9
Varmail	5.5	6.5	4.6	4.6
Webserver	18.8	19.6	19.6	19.5

are uploaded at the beginning of the trace replay. For Word trace, Dropsync's upload traffic is smaller than Dropbox in Figure 8(c), because the mobile phone is less powerful, it only completed limited numbers of sync actions, which has the effect of batching file updates. WeChat trace on mobile has similar upload traffic to Word trace also because of the wimpy CPU and the unintentionally batch manner. DeltaCFS shows similar numbers on mobile to that on PC, because we have fully minimized its CPU overhead, which makes it perfectly feasible to run on wimpy mobiles. Download traffic is small as shown in Figure 9(b), DeltaCFS has almost no download traffic, Dropsync downloads 3MB to 5MB data.

D. Local Read&Write Performance

We run the following experiments on a physical machine which is equipped with 2 Quad-Core AMD Opteron(tm) Processor 8380, 64GB memory, and runs Ubuntu 12.04 (kernel 3.2.0), to test the file IO performance of DeltaCFS client. In this test, we drop the data dequeued from Sync Queue rather than sending them to the server, in order to eliminate the impact of limited network bandwidth. We run microbenchmarks from filebench on different settings: native ext4, loopback FUSE, and DeltaCFS. The results are shown in Table III. For Fileserver, Native and FUSE show similar performance, but actually the response latency of FUSE is nearly twice as high as Native. This latency is covered by multi-thread IO operations. For Varmail and Webserver, FUSE's performance is a little better due to FUSE kernel module's file cache and prefetch mechanism [43]. DeltaCFS's performance is lower than FUSE for Fileserver and Varmail, because Sync Queue becomes full very quickly. While this does not happen in Webserver, that is why FUSE and DeltaCFS show the same number. For Fileserver, DeltaCFS with checksum has less throughput than that without checksum due to the additional latency induced by checksum computation. While this latency is not a problem for Varmail and Webserver, since it is very small compared to disk seek latency. We believe that the performance provided by DeltaCFS is enough for desktop and mobile environment.

TABLE IV
RESULTS OF RELIABILITY TESTS. *"Inconsistent" means the inconsistent data due to system crash. "omit" means the file is ignored, neither uploaded nor downloaded.*

Services	Data		Upload
	Corrupted	Inconsistent	Causal
Dropbox	upload	upload/omit	N
Seafile	upload	upload/omit	N
DeltaCFS	detect	detect	Y

E. Data Consistency

We test the efficacy of the mechanisms in DeltaCFS for data integrity and consistency, and compare it with Dropbox and Seafile. The testbed file system is ext4 (configured as ordered journaling), we inject corrupted data and crash inconsistent data in it [5]. For the data corruption experiment, we inject corrupted data by flipping a bit in a file. After sync clients are restarted, we write 1 byte to that file to see whether the corrupted data is also uploaded. For the crash inconsistency experiment, we cut off the power of the machine during a file in the sync folder is being written. After the machine is powered on, we first inject inconsistent data to simulate crash inconsistency by writing data to the file bypassing the file system [6]. The results are shown in Table IV, as expected, Dropbox and Seafile always upload the corrupted data, whereas DeltaCFS finds out the corrupted data block. For crash inconsistency, there is a high possibility that Dropbox and Seafile will upload the inconsistent file, but not always. This depends on whether they notice that this file is changed (*i.e.,* local version does not equal to the version on the cloud).

[5]We use *debugfs* tool to find out a file's physical location, then directly write the *dev* disk file to inject corrupted/inconsistent data.

[6]This simulates the common inconsistency of ordered journaling, where data are changed while metadata are not.

DeltaCFS can find out the inconsistency in this file and prevent it from being uploaded. At last, we tested the uploading order by generating files of different sizes. DeltaCFS always follows the update order when uploading those files, while Dropbox and Seafile do not guarantee this order, small files are often uploaded first.

V. Related Work

Network file services. NFSv3/4 [37] is a popular network file system, which is perfectly fit for LAN and also has been widely deployed in data centers. Though some optimizations have been proposed on NFS, such as client cache [44], file lease, it still shows poor performance on WAN. This problem is targeted by LBFS [5], which is a low-bandwidth network file system. It designs content defined chunking (CDC) to deduplicate chunks and only transfer modified chunks in an effective manner. This mechanism is also employed in Seafile [3]. Cumulus [45] backups filesystem to the cloud which only provides very simple storage interface (*e.g.,* Amazon S3). It mainly focuses on implementing incremental snapshot on the cloud using the limited interface, and the targeted scenario is relatively simple, no multi-client file sync and sharing. Pangaea [46] is a wide-area file system which supports data sharing among widely distributed users. It is built on decentralized commodity computers, focusing on data availability and replica consistency. Ori [47] is a file system replicated and synced among multiple devices, a fully peer-to-peer solution, supporting data sharing and file system snapshot. It also uses CDC with the average chunk size of 4KB to implement incremental snapshot and efficient data sync. Unlike Ori, DeltaCFS uses cloud-based data sync, with further optimized incremental data sync mechanism.

Cloud synchronization. The growth of cloud service promotes the development of consumer oriented cloud storage service (*e.g.,* Dropbox, OneDrive). A number of research efforts have been made, including performance measurement [2], [15], [48]–[50], performance optimization [15], [38], [51], data integrity/consistency [10], [16], data security [52], [53], and new data sync framework [10], [54]. Among these works, several popular personal cloud storage services are measured in [48], [49], revealing some design choices employed in those systems. ViewBox [16] measures data integrity and consistency levels supported by those services, and designs mechanisms to prevent the propagation of corrupted/inconsistent data. DeltaCFS also guarantees data consistency but through a different and lightweight mechanism. UDS [15] batches frequent, small file updates in order to reduce traffic overuse existed in cloud storage services, which is a simple but efficient way to reduce client's overhead. However, these systems still apply delta encoding for all types of files. In order to guarantee the consistency between tabular and object data, Simba [10] unifies the management of those data types and provides high-level interfaces for mobile apps, but in order to use their system existing applications should be modified a lot, which needs considerable engineering efforts. On the contrary, DeltaCFS does not require any modifications to applications. Bluesky [54] is a network file system backed by cloud storage, using cache mechanism to offer low latency data access. It is still a full-blown NFS, while DeltaCFS targets on cloud data sync scenario. It is possible to design a very high performance network file system for wide area network based on DeltaCFS as it has perfect incremental data sync performance.

Delta encoding and deduplication. Data compression [55] and delta encoding [17]–[20] are commonly used to optimize storage and network usage. TAPER [56] designs four redundancy elimination phases, from coarse-grained deduplication to fine-grained deduplication, finding tradeoff between computation overhead and network efficiency. Similarly, REBL [17] combines compression, deduplication of content-defined chunks, and delta-compression of similar chunks, to achieve more effective data reduction. sDedup [57] focuses on document-oriented database management systems, executes delta encoding against similar documents to reduce data transfer for replicated document DBMSs. DERD [18] dynamically selects the base file which shows a sufficient resemblance from a large collection of files. Since they belong to delta encoding, some types of workload cannot be handled efficiently.

Deduplication techniques have been researched a lot [58]. These techniques are both designed for primary storage [59], [60] and secondary or backup storage [61], [62]. They usually leverage spatial locality and temporal locality of real-word workloads to achieve good data access performance and deduplication rate [59], [62], [63]. The granularity of deduplication can be file-level [56], [64] or block-level [5], usually, block-level presents better deduplication rate.

VI. Conclusion and Future Work

Cloud sync services are experiencing an upsurge in popularity, which makes efficient service provisioning on both client and server sides more and more crucial. In this paper, we rethink the design of the cloud sync service, illustrate new design opportunities, and propose a new cloud-based data sync framework. In DeltaCFS, all types of files are synchronized incrementally with minimized CPU consumption and network traffic. In this design, the load of the server side is minimized as well, servers simply apply incremental data on files. So it becomes possible to use wimpy servers (*e.g.,* Intel Atom Processor) attached with large numbers of disks to provide cloud data sync services. We leave system design on the server side in future work.

Acknowledgment

We would like to thank our shepherd Michael A. Kozuch for his guidance, and ICDCS reviewers for their valuable feedback. This work is supported by State Key Program of National Natural Science Foundation of China under Grant No. 61232004, the High-Tech Research and Development Program of China ("863–China Cloud" Major Program) under grant 2015AA01A201, NSFC under Grant No. 61472009, and Shenzhen Key Fundamental Research Projects under Grant No. JCYJ20151014093505032.

REFERENCES

[1] "Dropbox," https://www.dropbox.com/.

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

[3] "Seafile," https://www.seafile.com/en/home/.

[4] A. Tridgell, P. Mackerras *et al.*, "The rsync Algorithm," 1996.

[5] A. Muthitacharoen, B. Chen, and D. Mazieres, "A Low-Bandwidth Network File System," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, 2001, pp. 174–187.

[6] "SoundHound Inc." http://soundhound.com/.

[7] "1Password Collect all your passwords in one safe." https://1password.com/.

[8] "Use Continuity to connect your Mac, iPhone, iPad, iPod touch, and Apple Watch," https://support.apple.com/en-us/HT204681.

[9] "Collaboration Platform Performance & Continuity," http://www.metalogix.com/solution/collaboration-platform-performance-continuity.

[10] Y. Go, N. Agrawal, A. Aranya, and C. Ungureanu, "Reliable, Consistent, and Efficient Data Sync for Mobile Apps," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[11] "Dropbox adds new tools to make syncing smarter," http://www.pcworld.com/article/2043980/dropbox-adds-new-tools-to-make-syncing-smarter.html.

[12] "SQLite," https://sqlite.org.

[13] "How to backup up your iPhone, iPad, and iPod touch." https://support.apple.com/en-us/HT203977.

[14] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "A file is not a file: Understanding the i/o behavior of apple desktop applications," in *Proceedings of 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

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

[16] Y. Zhang, C. Dragga, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "ViewBox: Integrating Local File Systems with Cloud Storage Services," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST)*, 2014, pp. 119–132.

[17] P. Kulkarni, F. Douglis, J. D. LaVoie, and J. M. Tracey, "Redundancy Elimination Within Large Collections of Files," in *Proceedings of USENIX Annual Technical Conference (USENIX ATC)*, 2004, pp. 59–72.

[18] F. Douglis and A. Iyengar, "Application-specific Delta-encoding via Resemblance Detection," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2003, pp. 113–126.

[19] J. MacDonald, "File System Support for Delta Compression," Ph.D. dissertation, Masters thesis. Department of Electrical Engineering and Computer Science, University of California at Berkeley, 2000.

[20] T. Suel and N. Memon, "Algorithms for Delta Compression and Remote File Synchronization," 2002.

[21] "librsync in dropbox," https://github.com/dropbox/librsync.

[22] "Seafile Data Model," https://manual.seafile.com/develop/data_model.html.

[23] "WeChat: Connecting 800 million people with chat, calls, and more." https://www.wechat.com/en/.

[24] "Autosync Dropbox - Dropsync," https://play.google.com/store/apps/details?id=com.ttxapps.dropsync\&hl=en.

[25] H. Kim, N. Agrawal, and C. Ungureanu, "Revisiting Storage for Smartphones," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[26] J. Li, A. Badam, R. Chandra, S. Swanson, B. L. Worthington, and Q. Zhang, "On the Energy Overhead of Mobile Storage Systems," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST)*, 2014, pp. 105–118.

[27] "FUSE: Filesystem in Userspace," http://fuse.sourceforge.net/.

[28] B. Cornell, P. A. Dinda, and F. E. Bustamante, "Wayback: A User-level Versioning File System for Linux," in *Proceedings of Usenix Annual Technical Conference (USENIX ATC)*, 2004, pp. 19–28.

[29] "What Is ZFS?" http://docs.oracle.com/cd/E23823_01/html/819-5461/zfsover-2.html#gayou.

[30] "Btrfs," https://btrfs.wiki.kernel.org/index.php/Main_Page.

[31] "LevelDB: a light-weight, single-purpose library for persistence with bindings to many platforms." https://leveldb.org.

[32] S. C. Tweedie, "Journaling the Linux ext2fs filesystem," in *Proceedings of the 4th Annual Linux Expo*, 1998.

[33] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis and Evolution of Journaling File Systems," in *Proceedings of USENIX Annual Technical Conference (USENIX ATC)*, 2005, pp. 105–120.

[34] R. Alagappan, V. Chidambaram, T. S. Pillai, A. Albarghouthi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Beyond Storage APIs: Provable Semantics for Storage Stacks," in *Proceedings of 15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[35] J. D. Valois, "Implementing Lock-Free Queues," in *Proceedings of the 7th international conference on Parallel and Distributed Computing Systems*, 1994, pp. 64–69.

[36] "Dokan: an user mode file system for Windows," https://dokan-dev.github.io/.

[37] "Network File System (NFS) version 4 Protocol," https://www.ietf.org/rfc/rfc3530.txt.

[38] S. Li, Q. Zhang, Z. Yang, and Y. Dai, "Understanding and Surpassing Dropbox: Efficient Incremental Synchronization in Cloud Storage Services," in *Proceedings of IEEE Globecom*, 2015.

[39] "inotify - monitoring filesystem events." http://man7.org/linux/man-pages/man7/inotify.7.html.

[40] "Network File System (NFS) version 4 Protocol. 4.2.3. Volatile Filehandle. 9.3.4. Data Caching and File Identity," https://www.ietf.org/rfc/rfc3530.txt.

[41] D. Campello, H. Lopez, L. Useche, R. Koller, and R. Rangaswami, "Non-blocking Writes to Files," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015, pp. 151–165.

[42] "Snappy," http://google.github.io/snappy/.

[43] A. Rajgarhia and A. Gehani, "Performance and Extension of User Space File Systems," in *Proceedings of the 25th ACM Symposium on Applied Computing (SAC)*, 2010, pp. 206–213.

[44] J. J. Kistler and M. Satyanarayanan, "Disconnected operation in the coda file system," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 1, pp. 3–25, 1992.

[45] M. Vrable, S. Savage, and G. M. Voelker, "Cumulus: Filesystem Backup to the Cloud," *ACM Transactions on Storage (TOS)*, vol. 5, no. 4, p. 14, 2009.

[46] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam, "Taming Aggressive Replication in the Pangaea Wide-Area File System," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 15–30, 2002.

[47] A. J. Mashtizadeh, A. Bittau, Y. F. Huang, and D. Mazieres, "Replication, History, and Grafting in the Ori File System," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013, pp. 151–166.

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

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

[50] A. Li, X. Yang, S. Kandula, and M. Zhang, "CloudCmp: Comparing Public Cloud Providers," in *Proceedings of the 10th ACM Internet Measurement Conference (IMC)*, 2010, pp. 1–14.

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

[52] M. Li, C. Qin, and P. P. Lee, "CDStore: Toward Reliable, Secure, and Cost-Efficient Cloud Storage via Convergent Dispersal," in *Proceedings of USENIX Annual Technical Conference (USENIX ATC)*, 2015.

[53] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish, "Depot: Cloud storage with minimal trust," *ACM Transactions on Computer Systems (TOCS)*, vol. 29, no. 4, p. 12, 2011.

[54] M. Vrable, S. Savage, and G. M. Voelker, "Bluesky: A Cloud-Backed File System for the Enterprise," in *Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST)*, 2012, pp. 19–19.

[55] D. A. Lelewer and D. S. Hirschberg, "Data compression," *ACM Computing Surveys (CSUR)*, vol. 19, no. 3, pp. 261–296, 1987.

[56] N. Jain, M. Dahlin, and R. Tewari, "Taper: Tiered Approach for Eliminating Redundancy in Replica Synchronization," in *Proceedings of*

the 4th USENIX Conference on File and Storage Technologies (FAST), 2005, pp. 21–21.

[57] L. Xu, A. Pavlo, S. Sengupta, J. Li, and G. R. Ganger, "Reducing Replication Bandwidth for Distributed Document Databases," in *Proceedings of ACM Symposium on Cloud Computing (SoCC)*, 2015.

[58] D. T. Meyer and W. J. Bolosky, "A Study of Practical Deduplication," *ACM Transactions on Storage (TOS)*, vol. 7, no. 4, p. 14, 2012.

[59] K. Srinivasan, T. Bisson, G. R. Goodson, and K. Voruganti, "iDedup: Latency-aware, inline data deduplication for primary storage," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, 2012, pp. 1–14.

[60] R. Koller and R. Rangaswami, "I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance," *ACM Transactions on Storage (TOS)*, vol. 6, no. 3, p. 13, 2010.

[61] D. Bhagwat, K. Eshghi, D. D. Long, and M. Lillibridge, "Extreme Binning: Scalable, Parallel Deduplication for Chunk-based File Backup," in *Proceedings of IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2009, pp. 1–9.

[62] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezis, and P. Camble, "Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality," in *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST)*, 2009, pp. 111–123.

[63] B. Zhu, K. Li, and R. H. Patterson, "Avoiding the Disk Bottleneck in the Data Domain Deduplication File System," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, 2008, pp. 1–14.

[64] L. L. You, K. T. Pollack, and D. D. Long, "Deep Store: An Archival Storage System Architecture," in *Proceedings of the 21st International Conference on Data Engineering (ICDE)*, 2005, pp. 804–815.