QuickSync: Improving Synchronization Efficiency for Mobile Cloud Storage Services

Yong Cui, Zeqi Lai, Xin Wang, and Ningwei Dai

Abstract—Mobile cloud storage services have gained phenomenal success in recent few years. In this paper, we identify, analyze, and address the synchronization (*sync*) inefficiency problem of modern mobile cloud storage services. Our measurement results demonstrate that existing commercial sync services fail to make full use of available bandwidth, and generate a large amount of unnecessary sync traffic in certain circumstances even though the incremental sync is implemented. For example, a minor document editing process in Dropbox may result in sync traffic 10 times that of the modification. These issues are caused by the inherent limitations of the sync protocol and the distributed architecture. Based on our findings, we propose QuickSync, a system with three novel techniques to improve the sync efficiency for mobile cloud storage services, and build the system on two commercial sync services. Our experimental results using representative workloads show that QuickSync is able to reduce up to 73.1 percent sync time in our experiment settings.

Index Terms—Mobile cloud storage, mobile networks, measurement, synchronization efficiency

1 Introduction

Personal cloud storage services are gaining tremendous popularity in recent years by enabling users to conveniently synchronize files across multiple devices and back up data. Services like Dropbox, Box, Seafile have proliferated and become increasingly popular, attracting many big companies such as Google, Microsoft or Apple to enter this market and offer their own cloud storage services. As a primary function of cloud storage services, data synchronization (sync) enables the client to automatically update local file changes to the remote cloud through network communications. *Synchronization efficiency* is determined by the speed of updating the change of client files to the cloud, and considered as one of the most important performance metrics for cloud storage services. Changes on local devices are expected to be quickly synchronized to the cloud and then to other devices with low traffic overhead.

More recently, the quick increase of mobile devices poses the new demand of ubiquitous storage to synchronize users' personal data from anywhere at anytime and with any connectivity. Some cloud storage providers have extended and deployed their services in mobile environments to support Mobile Cloud Storage Services, with functions such as chunking and deduplication optionally implemented to improve the transmission performance.

Despite the efforts, the sync efficiency of popular mobile cloud storage services is still far from being satisfactory, and under certain circumstances, the sync time is much longer than expected. The challenges of improving the sync efficiency in mobile/wireless environment are threefold. First, as commercial storage services are mostly closed source with data encrypted, their designs and operational processes remain unclear to the public. It is hard to directly study the sync protocol and identify the root cause of sync difficulty. Second, although some existing services try to improve the sync performance by incorporating several capabilities, it is still unknown whether these capabilities are useful or enough for good storage performance in mobile/wireless environments. Finally, as a mobile cloud storage system involves techniques from both storage and network fields, it requires the storage techniques to be adaptive and work efficiently in the mobile environment where the mobility and varying channel conditions make the communications subject to high delay or interruption.

To address above challenges, we identify, analyze and propose a set of techniques to increase the sync efficiency in modern mobile cloud storage systems. Our work consists of three major components: 1) identifying the performance bottlenecks based on the measurement of the sync operations of popular commercial cloud storage services in the mobile/wireless environment, 2) analyzing in details the problems identified, and 3) proposing a new mobile cloud storage system which integrates a few techniques to enable efficient sync operations in mobile cloud storage services.

We first measure the sync performance of the most popular commercial cloud storage services in mobile/wireless networks (Section 2). Our measurement results show that the sync protocol used by these services is indeed inefficient. Specifically, the sync protocol can not fully utilize the available bandwidth in high RTT environment or when

- *Y. Cui, Z. Lai, and N. Dai are with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, P.R. China. E-mail: cuiyong@tsinghua.edu.cn, {laizq13, dnw15}@mails.tsinghua.edu.cn.*
- *X. Wang is with the Department of Electrical and Computer Engineering, Stony Brook University, Stony Brook, New York, NY 11794. E-mail: x.wang@stonybrook.edu.*

Manuscript received 25 Jan. 2016; revised 3 Feb. 2017; accepted 28 Mar. 2017. Date of publication 0 . 0000; date of current version 0 . 0000.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TMC.2017.2693370

TABLE 1
Capability Implementation of Four Popular Cloud Storage Services

Capabilities	Windows				Android			
	Dropbox	Google Drive	OneDrive	Seafile	Dropbox	Google Drive	OneDrive	Seafile
Chunking	4 MB	8 MB	var.	var.	4 MB	260 KB	1 MB	×
Bundling	√	×	×	×	×	×	×	×
Deduplication	√	×	×	√	√	×	×	×
Delta encoding	√	×	×	√	×	×	×	×
Data compression	√	√	×	×	×	×	×	×

The var. refers to variable chunk size.

synchronizing multiple small files. Furthermore, although some services, e.g., Dropbox, have implemented the incremental sync to reduce the traffic size, this technique is not valid in all scenarios. We observe that a document editing process may result in sync traffic 10 times that of the modification.

We further conduct in-depth analysis of the trace data and also apply decryption to identify the root cause of the inefficiency in the sync protocol (Section 3). Based on our studies, the two major factors that contribute to the inefficiency are the inherent limitations of the sync protocol and the distributed storage architecture. Specifically, the deduplication to reduce redundant data transmissions does not always contribute to the sync efficiency. The distributed nature of storage services poses a challenge to the practical implementation of the delta encoding algorithm, and the failure in the incremental sync may lead to a large traffic overhead. The iterative sync scheme suffers from low throughput when there is a need to synchronize a set of files through a slow network.

Based on our observation and analysis, we propose QuickSync, a system with three novel techniques to improve the sync efficiency for mobile cloud storage services (Section 4). To reduce the the sync time, we introduce *Network-aware Chunker* to adaptively select the proper chunking strategy based on real-time network conditions. To reduce the sync traffic overhead, we propose *Redundancy Eliminator* to correctly perform delta encoding between two similar chunks located in the original and modified files at any time during the sync process. We also design *Batched Syncer* to improve the network utilization of sync protocol and reduce the overhead when resuming the sync from an interruption.

We build our QuickSync system on Dropbox, currently the most popular cloud storage services, and Seafile, an popular open source personal cloud storage system (Section 5). Collectively, these techniques achieve significant improvement in the sync latency for cloud storage services. Evaluation results (Section 6) show that the QuickSync system is able to significantly improve the sync efficiency, reducing up to 73.1 percent sync time in representative sync scenarios with our experiment settings. To the best of our knowledge, we are the first to study the sync efficiency problem for mobile cloud storage services.

2 Synchronization Inefficiency

Sync efficiency indicates how fast a client can update changes to the cloud. In this section, we conduct a series of experiments to investigate the sync inefficiency issues existing in four most popular commercial cloud storage service systems in wireless/mobile environments. We will further analyze our observed problems and explain their root causes in Section 3.

2.1 Architecture and Capabilities

The key operation of the cloud storage services is *data sync*, which automatically maps the changes in local file systems to the cloud via a series of network communications. Before presenting the sync inefficiency issues, we first give a brief overview of the typical architecture of cloud storage services and the key capabilities that are often implemented for speeding up data transmissions.

Architecture. A typical architecture of cloud storage services includes three major components [1]: the *client*, the *control server* and the *data storage server*. Typically, a user has a designated local folder (called sync folder) where every file operation is informed and synchronized to the cloud by the client. The client splits file contents into chunks and indexes them to generate the metadata (including the hashes, modified time etc.). The file system on the server side has an abstraction different from that of the client. Metadata and contents of user files are separated and stored in the control and data storage servers respectively. During the sync process, metadata are exchanged with the control server through the *metadata information flow*, while the contents are transferred via the *data storage flow*. In a practical implementation, the control server and the data storage server may be deployed in different locations. For example, Dropbox builds its data storage server on Amazon EC2 and S3, while keeping its own control server. Another important flow, namely *notification flow*, pushes notifications to the client once changes from other devices are updated to the cloud.

Key Capabilities. Cloud storage services can be equipped with several capabilities to optimize the storage usage and speed up data transmissions: *1) chunking* (i.e., splitting a content into a certain size data unit), *2) bundling* (i.e., the transmission of multiple small chunks as a single chunk), *3) deduplication* (i.e., avoiding the retransmission of content already available in the cloud), *4) delta-encoding* (i.e., only transmitting the modified portion of a file) and *5) compression*. The work in [2] shows how the capabilities are implemented on the desktop clients. We further follow the methods in [2] to analyze the capabilities already implemented on the mobile clients. Table 1 summarizes the capabilities of each service on multiple platforms, with the test client being the newest released version before March 1, 2015. In following sections, we will show that these capabilities also make a strong side impact on the sync efficiency.

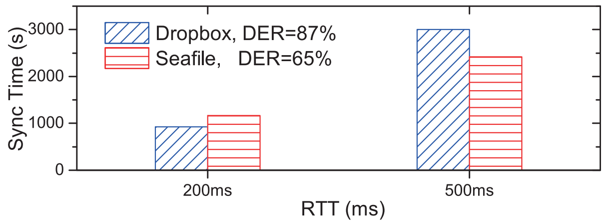


Fig. 1. Lower DER does not always make efficient.

TABLE 2
Three Types of Modification Operations

Operation	Description (assuming the file size is S bytes)
Flip	flip w bytes data at the head, end or random position of the test file.
Insert	insert w random data at the head, end or random position of the test file.
Delete	delete w random data at the head, end or random position of the test file.

2.2 Low DER Not Equal to Efficiency

To evaluate the effectiveness of deduplication in reducing the original transmission data size, the metric *Deduplication Efficiency Ratio* (DER) is defined as the ratio of the *deduplicated file size* to the *original file size*. Intuitively, lower DER means more redundancy can be removed and the total sync time can be reduced. However, our experiment indicates that lower DER *may not* alway make sync efficient.

As only Dropbox and Seafile incorporate the deduplication function, to study the relationship between the sync time and DER, we use `Wireshark` to measure the packet level trace of the two services in a controlled WiFi environment. We use `tc` to tune the RTT for each service according to the typical RTT values in mobile/wireless networks [3]. We only perform measurements on the Windows platform because most services did not implement the deduplication on the Android platform. We collect about 500 MB user data from a Dropbox user and upload these fresh data via the tested services. From the trace captured we can get the sync time and calculate the DER as a ratio of the transmission traffic size and the original traffic size.

Fig. 1 shows that the DER for Dropbox and Seafile are 87 and 65 percent respectively under each RTT setting. Intuitively, a higher DER value would take more time to complete a sync operation. However, we find that in a better network condition (e.g., when the RTT is 200 ms), it costs more time for Seafile to complete the sync.

2.3 Failure of Incremental Sync

To reduce the network traffic for synchronizing changes, some services such as Dropbox leverage the delta encoding algorithm (e.g., rsync [4]) to achieve *incremental sync* instead of *full-file sync*. However, as we will show next, the incremental sync *is not* always available and the client software may synchronize much more data than expected. To evaluate how much additional traffic is incurred, we define a metric *Traffic Utilization Overhead (TUO)* as the ratio of the *generated traffic size* to the *expected traffic size*. When the value of TUO is larger than 1, it indicates additional data are transferred. A large TUO value indicates that more extra data are transmitted to the storage server during a sync process. We conduct two sets of experiments to find out when the claimed incremental sync mechanism fails.

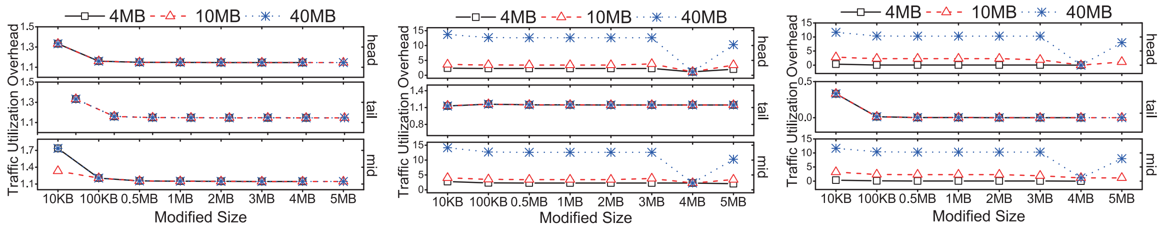
In the first experiment, all operations are performed on synchronized files with both the metadata and contents completely updated to the cloud. We perform three types of basic operation in typical real-world usage patterns: *flip bits*, *insert* and *delete* several continuous bytes at the head, end or random position of the test file, and see how much sync traffic will be generated when the given operation is performed. Table 2 provides the details of these three basic operations. Since 10 KB is the recommended default window size in the original delta encoding algorithm [4], we vary w from 10 KB to 5 MB to ensure that the modification size is larger than the minimum delta that can be detected. To avoid the possible interaction between two consecutive operations, the next operation is performed after the previous one is completed. An operation in each case is performed 10 times to get the average result. Because GoogleDrive and OneDrive have not implemented the incremental sync, they upload the whole file upon the modification, and are expected to have a large amount of traffic even for a slight modification. Thus in this section our studies focus on Dropbox and Seafile.[1]

In Figs. 2 a, 2 b, and 2 c, for Dropbox, interestingly the three types of operation result in totally different traffic sizes. For the flip operation, in most cases the TUO is close to 1. Even when the modification window is 10 KB, the TUO is less than 1.75, indicating that incremental sync works well for flip operations performed at any position. The sync traffic of insert operation is closely related to the position of the modification. The TUO is close to 1 when an insertion is performed at the end of the file, but the generated traffic is much higher than expected when an insertion is made at the head or a random position. Specifically, inserting 3 MB data at the head or random position of a 40 MB file results in nearly 40 MB sync traffic, which is close to the full file sync mechanism. The TUO results for the delete operation are similar to the insert operation. Differently, deleting at the end of the file generates small sync traffic (TUO is close to zero). However deleting at the head or random position leads to larger sync traffic, especially for a large file, e.g., 40 MB (TUO is larger than 10). Another interesting finding is that for both insert and delete operations in Dropbox, the TUO drops to a very low value when the modification window w is 4 MB, where the TUO is close to 1 for the insert operation and close to 0 for the delete operation.

In Figs. 2 d, 2 e, and 2 f, the TUO results of different operations for Seafile are similar. Although the TUO results are close to 1 for large modifications (e.g., modified size ≥ 1 MB), the TUO results are larger than 10 for all modifications smaller than 100 KB. This shows that the incremental sync in Seafile fails to reduce the sync traffic for small modifications, no matter where the changes are made in a file.

In the second experiment, we investigate the sync traffic of performing the modification on the files while the sync data are in the middle of transmissions to the cloud. We first create a 4 MB fresh file in the sync folder, and perform the

1. The latest version of Seafile adds the incremental sync. Therefore, based on our prior conference version we add the measurement for Seafile.



(a) Dropbox: flip (b) Dropbox: insert (c) Dropbox: delete

(d) Seafile: flip (e) Seafile: insert (f) Seafile: delete

Fig. 2. Traffic utilization overhead of Dropbox and Seafile generated by a set of modifications. In this experiment, we perform flip, insert, and delete operation over continuous bytes at the head, end or random position of the test file.

same flip operation as that in the first experiment at a random position with the modification window $w = 512$ KB in every 20 s. Note that the TUO of such an operation is close to 1 in the first experiment, and in the second experiment, the flip operation is performed *immediately* after the file is created while the sync process has not completed. Such a behavior is common for an application such as MS-word or VMware which creates fresh temp files and periodically modifies them at runtime. We vary the number of modifications to measure the traffic size. We also use `tc` to involve additional RTT to see the traffic under different network conditions.

Fig. 3 shows the sync traffic of Dropbox for periodic flip on a 4 MB file with various RTT. Interestingly, for all cases the TUO is larger than 2, indicating that at least 8 MB data are synchronized. Moreover, we observe that the TUO is affected by the RTT. When the RTT is 600 ms, surprisingly the TUO rises with the increase of the modification times. The sync traffic reaches up to 28 MB, 448 percent of the new content size(including both the fresh file and immediate modifications) when the modifications are performed five times. The result of Seafile is similar to that of Dropbox and omitted due to the page limit.

Collectively, our measurement results show that the incremental sync does not work well in all cases. Specifically, for insert and delete operations at certain positions, the generated traffic size is much larger than the expected size. Moreover, the incremental sync mechanism may fail when attempting to synchronize with the files in the middle of the sync process which results in undesirable traffic overhead.

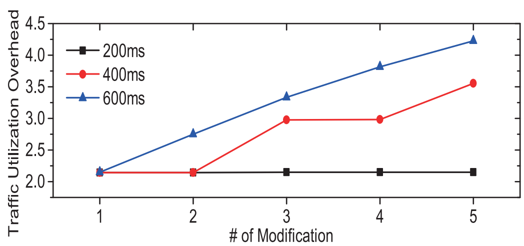


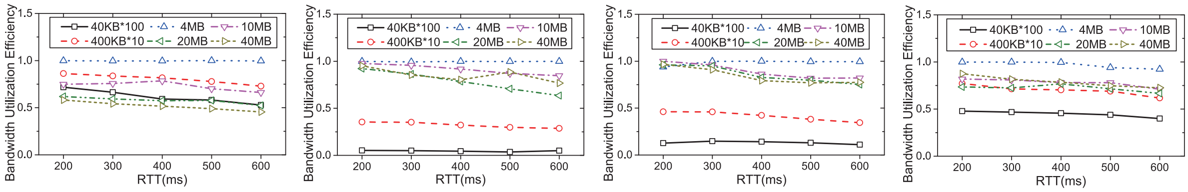
Fig. 3. TUO of synchronizing modification in the middle of sync process.

2.4 Bandwidth Inefficiency

Sync throughput is another critical metric that reflects the sync efficiency. The sync protocol relies on TCP and its performance is affected by network factors such as RTT or packet loss. Because of different system implementations, it is unreasonable to evaluate how the underlying bandwidth of a storage service is used by directly measuring the throughput or latency [2]. To characterize the network usage of sync protocol, we introduce a novel metric, *Bandwidth Utilization Efficiency (BUE)*, which is defined as the ratio of the *practical measured throughput* to the *theoretical TCP bandwidth*. The latter indicates the available bandwidth in steady state and can be estimated by $\frac{Segment\_size*cwnd}{RTT}$, where *cwnd* is the observed average congestion window size during the transmission. The BUE metric is a value between 0 and 1. Rather than measuring the end-to-end throughput, we apply BUE to evaluate how well the cloud storage service can utilize the available network bandwidth to reduce the sync time.

To investigate how the sync protocol utilizes the underlying network bandwidth, we have the Windows and Android clients of Dropbox, GoogleDrive, OneDrive and Seafile run in Wi-Fi and cellular networks (UMTS) respectively. We create a set of highly compressed files (to exclude the effect of compression) with various total sizes in the sync folder and measure the packet-level trace using `Wireshark` and `tcpdump`. We compute the *theoretical TCP bandwidth* based on real-time observed RTT and *cwnd* to calculate BUE. In Wi-Fi networks, we use `tc` to tune the RTT, simulating various network conditions. In cellular networks we change the position to tune the RTT. Each test is performed 10 times to calculate the average result.

The BUE results of all services in WiFi networks with different RTT are shown in Fig. 4. For each service, the BUE of synchronizing 4 MB file is close to 1, reflecting that all services are able to fully utilize the available bandwidth. The traffic size of synchronizing 40 KB*100 files is close to that of 4 MB file, but we observe that the BUE slumps significantly when synchronizing multiple files. This degradation is more serious for GoogleDrive and OneDrive, with their



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

Fig. 4. Bandwidth utilization efficiency of four cloud storage services in various network conditions.

BUE dropping under 20 percent when syncing 40 KB*100 files. For all services, BUE decreases for large files such as 20 or 40 MB and when RTT increases. The degradation of BUE indicates that the sync protocol cannot efficiently utilize the underlying available bandwidth. The decrease of BUE for large RTT indicates that the sync protocol can not well adapt to a slow network. Results in cellular networks are similar and omitted due to the page limit.

3 Root Cause of Sync Inefficiency

Our observations have demonstrated that mobile cloud storage services suffer from sync inefficiency problems. In this section, we analyze the sync protocol and explain the root causes for the inefficiency.

3.1 Pinning Down the Sync Protocol

It is difficult to directly analyze the sync protocol of commercial services such as Dropbox, as they are closed source and most of the network traffic is encrypted. To understand the sync protocol, we exploit both measurement and decryption. Specifically, we first analyze the network traces of all services studied in Section 2 to show the general sync process, and then we hijack the encrypted traffic of Dropbox to understand the protocol details.

Commonality Analysis. Although it is difficult to obtain the protocol details from the encrypted sync, we still can get some high-level knowledge of the protocol by analyzing the packet-level network traces, and our analyses indicate that the sync processes of all services in various platforms commonly have three key stages: *1) sync preparation stage*, the client first exchanges data with the control server; *2) data sync stage*, the client sends data to, or retrieves data from the data storage server. In case that the chunking scheme is implemented, data chunks are sequentially stored or retrieved with a "pause" in between, and the next chunk will not be transferred until the previous one is acknowledged by the receiver; *3) sync finish stage*, the client communicates with the control server again to conclude its sync process.

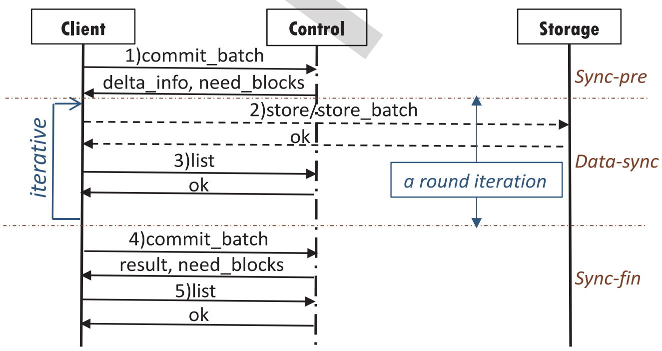


Fig. 5. A typical sync process of Dropbox.

In-Depth Analysis. The Dropbox client is written in Python. To decrypt the traffic and obtain the details of the sync protocol, we leverage the approach in [5] to hijack the SSL socket by DynamoRIO [6]. Although we only decrypt the Dropbox protocol, combining the commonality analysis we think the other three services may follow a sync protocol similar to that of Dropbox.

Fig. 5 shows a typical Dropbox sync workflow when uploading a new file. In the *sync preparation stage*, the file is first split and indexed locally, and the *block list* which includes all identifiers of chunks is sent to the control server in the *commit_batch*. Chunks existing in the cloud can be identified through hash-based checking and only new chunks will be uploaded. Next in the *data-synchronization stage*, the client communicates with the storage server directly. The client synchronizes data iteratively, and in each round of iteration several chunks will be sent. At the end of one round of iteration, the client updates the metadata through the *list* message to inform the server that a batch of chunks have been successfully synchronized, and the server sends an *OK* message in response. Finally in the *sync-finish stage*, the client communicates with the control server again to ensure that all chunks are updated by the *commit_batch*, and updates the metadata.

3.2 Why Less Data Cost More Time

Generally, to identify the redundancy in the sync process, the client splits data into chunks and calculates their hashes to find the redundancy. However, chunking with a large number of hashing operations is computationally expensive, and the time cost and the effectiveness of deduplication are strongly impacted by the chunking method. For instance, fixed-size chunking used by Dropbox is simple and fast, but is less effective in deduplication. Content defined chunking (CDC) [7] used by Seafile is more complex and computation extensive, but can identify a larger amount of redundancy.

In our experiment in Section 2.2, when RTT is 200 ms, Seafile uses the content defined chunking to achieve 65 percent DER. Although the available bandwidth is sufficient, the complex chunking method takes too much time hence its total sync time is larger than Dropbox. However, when the RTT is 500 ms and the bandwidth is limited, lower DER leads to lower sync time by significantly reducing the transmission time. The key insight from this observation is that it is helpful to dynamically select the appropriate chunking method according to the channel condition.

3.3 Why the Traffic Overhead Increases

Although delta encoding is a mature and effective method, it is not implemented in all cloud storage services. One

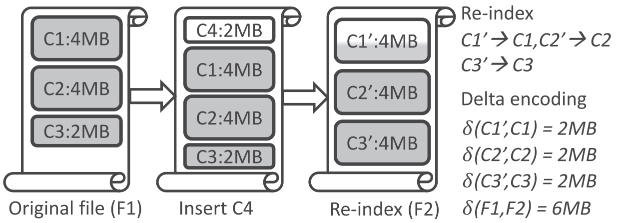


Fig. 6. An example to explain why the incremental sync fails in Dropbox. After inserting 2 MB data (C4) at the beginning of a 10 MB file, Dropbox re-indexes chunks and calculates the delta content.

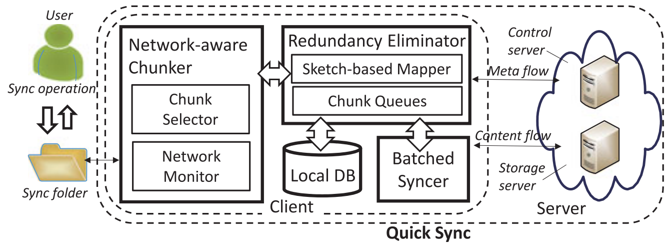


Fig. 7. QuickSync system overview.

possible reason is that most delta encoding algorithms work at the granularity of file, while to save the storage space for lower cost, files are often split into chunks to manage for cloud storage services. Naively piecing together all chunks to reconstruct the whole file to achieve incremental sync would waste massive intra-cluster bandwidth.

Among popular storage clouds, Dropbox implements delta encoding at the chunk granularity. From the decrypted traffic, we find that each chunk has a "parent" attribute to map it to another similar chunk, and the delta encoding is adopted between the two chunks. Fig. 6 shows how Dropbox performs delta encoding at the granularity of chunk when inserting 2 MB data at the head of a 10 MB file. When the file is modified, the client follows the fixed-size chunking method to split and re-index the file. The chunks without hash change are not processed further, so the TUO results of 4 MB window size in Fig. 2 are all close to 1. Otherwise, a map is built based on the relative locations of the original and modified versions, and the delta encoding is executed between mapped chunks. Thus the delta of C1′ and C1 is 2 MB and the total delta is 6 MB, 3 times the insertion size. In Fig. 2, inserting 3 MB data at the head of 40 MB file causes nearly 40 MB the total sync traffic, because all chunks are mapped to different parents after the re-indexing. In this case, the incremental sync fails to only update the changed content. Different from Dropbox, the source codes of Seafile indicate that the minimal modification it can detect is 1 MB, which makes its delta-encoding algorithm very inefficient. Seafile generates much higher unexpected sync traffic for small file modifications.

As discussed in Section 3.1, the metadata is updated after contents are successfully uploaded. Therefore, for a chunk in the middle of sync, if it is modified before sync finishes, the chunk can not be used for delta encoding. In the second experiment in Section 2.3, when the modification happens at the beginning time of the sync process, the client has to upload both the original and modified versions and thus the TUO is at least 2. Moreover, in the case that RTT=600 ms, every modification is performed during the uploading process, and each modified version has to be uploaded entirely.

3.4 Why the Bandwidth Utilization Decreases

Iteration is a key characteristic of the data sync, but may significantly reduce the bandwidth utilization. There are several reasons. First, when synchronizing a lot of chunks smaller than the maximum chunk size, the client has to wait for an acknowledgement from the server before transferring the next chunk. Thus the sequential acknowledgement limits the bandwidth usage, especially when sending a number of small files and RTT is high.

Second, although Dropbox incorporates bundling to bundle small chunks into a bigger one (up to 4 MB) to mitigate the problem, we can still see the throughput slumps between two iterations when synchronizing large files (e.g., 40 MB). Different from other storage services, when transferring multiple big chunks at 4 MB, Dropbox opens up to four concurrent TCP connections during the sync process. At the beginning of a new iteration, the client assigns new chunks for different connections. If one connection has transferred the assigned chunk and received the acknowledgement, it will not immediately start to send the next chunk. Only after the other three connections have finished transmissions, the new chunks are assigned. During the iterations, because of the idle waiting of several connections, the throughput reduces significantly.

Moreover, for GoogleDrive, it opens several new TCP connections, each taking one iteration to transfer one chunk. For instance, it totally creates 100 storage flows in 100 iterations to synchronize 100 small files. Such a mechanism would incur additional overhead for opening a new SSL connection and extend the slow start period, leading to significant throughput degradation thus reduced BUE.

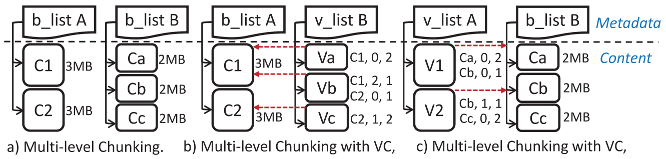
4 System Design

Improving the sync efficiency in wireless networks is important for mobile cloud storage services. In light of various issues that result in sync inefficiency, we propose QuickSync, a novel system which concurrently exploits a set of techniques over current mobile cloud storage services to improve the sync efficiency.

4.1 System Overview

To efficiently complete a sync process, our QuickSync system introduces three key components: the Network-aware Chunker (Section 4.2), the Redundancy Eliminator (Section 4.3), and the Batched Syncer (Section 4.4). The basic functions of the three components are as follows: *1) identifying redundant data through a network-aware deduplication technique; 2) reducing the sync traffic by wisely executing delta encoding between two "similar" chunks;* and *3) adopting a delayed-batched acknowledgement to improve the bandwidth utilization.*

Fig. 7 shows the basic architecture of QuickSync. The sync process begins upon detecting a change (e.g., add or modify a file) in the sync folder. First, the Chunk Selector inside the Network-aware Chunker splits an input file through content defined chunking with the chunk size determined based on the network condition monitored by the Network Monitor. Metadata and contents are then delivered to the Redundancy Eliminator, where redundant chunks are removed and delta



a) Multi-level Chunking. b) Multi-level Chunking with VC, strategy A is selected. The content of Va is stored in C1, while the offset and length are 0 and 2MB respectively. c) Multi-level Chunking with VC, strategy B is selected. The content of V1 is stored in Ca and Cb.

Fig. 8. An example showing how QuickSync generates Virtual Chunks on the server.

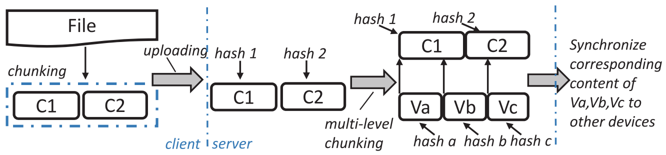


Fig. 9. When the server synchronizes data to the client, the server finds real contents via Virtual Chunks and then delivers data to the client.

encoding is executed between similar chunks to reduce the sync traffic for modification operations. Specifically, QuickSync leverages the Sketch-based Mapper to calculate the similarity of different chunks and identify similar chunks. A database is applied to store metadata of local files. Finally the Batched Syncer leverages a delayed-batched acknowledgement mechanism to synchronize all data chunks continuously to the cloud and conclude the sync process. Like other cloud storage systems, QuickSync separates the control server for metadata management from the storage server for data storage. Metadata and file contents are transferred by meta flow and content flow respectively. Next we describe the detailed design for each component.

4.2 Network-Aware Chunker

To improve the sync efficiency, our first step is to identify the redundant data before the sync process. Although deduplication is often applied to reduce the data redundancy for storage in general cloud systems, extending existing deduplication techniques for personal cloud storage services faces two new challenges. First, previous deduplication techniques mostly focus on saving the storage space [8], improving the efficiency for large-scale remote backup [9], [10], or only optimizing the downlink object delivery [11]. These strategies are difficult to apply for personal cloud storage because they often involve huge overhead and require an important property named “stream-informed” [8], which requires the data segment and their fingerprints to follow the same order as that in a data file or stream. Such a property is not included in a personal scenario. Second, a deduplication scheme should be network-aware in a mobile network with varying topology and channel conditions. A deduplication with aggressive chunking will incur high computational cost for mobile devices, which may degrade the sync performance under good network conditions (Section 2.2).

Generally, the chunking granularity is closely related to the computation overhead and the effectiveness of deduplication. A more aggressive chunking strategy with very small chunk size may allow for more effective deduplication, but would involve higher total computation overhead to identify the duplicated data over a large number of chunks, and vice versa. All previous deduplication systems use a static chunking strategy with a fixed average chunk size. Derived from the basic idea of Dynamic Adaptive Streaming over HTTP (DASH), the basic procedure of our approach is to adaptively select an appropriate chunking strategy according to the real-time network conditions to reduce the total sync time. Intuitively, in slow networks, since the bandwidth is limited, we select aggressive chunking strategy to identify more redundancy and reduce the transmission time. When the bandwidth is sufficient, we prefer larger chunks because of its lower computation overhead. Specifically, our approach consists of two key techniques as we will introduce below.

4.2.1 Network-Aware Chunk Size Selection

Instead, we propose the concept of *Virtual Chunk* that implicitly stores the offset and length to generate the pointers to the real content. For each user file on the server side, QuickSync only stores one copy of all its chunks including real contents, all Virtual Chunks under different chunking strategies, and the metadata. Specifically, the metadata mainly contains a *block list* including all hashes of chunks, and a *vblock list* including all hashes of Virtual Chunks. For a Virtual Chunk, the offset and the length of the corresponding chunks can be calculated based on the knowledge of all previous Virtual Chunk sizes in the *vblock list*. Therefore each Virtual Chunk only needs to store the chunk size of itself. In an uploading process of QuickSync, after receiving all chunks of a file, the server forms the file according to its metadata. It then conducts all other strategies on the *chunking strategy list* to resplit the file and generate the metadata under various strategies.

Fig. 8 gives an example to illustrate how QuickSync generates Virtual Chunks on the server to reduce the storage overhead. Assume that we have two optional chunking strategies to process a 6 MB file. To respond to different chunking requirements of the client, the server can maintain multiple *block_lists* containing all hashes and multiple copies of the same file, as shown in Fig. 8 a at the cost of large storage space. Figs. 8 b and 8 c show the cases when we use Virtual Chunks to save the storage space. For all Virtual Chunks generated by the equal chunking strategy, we add a *vblock_list* including all hashes of these Virtual Chunks to the metadata.

When the server needs to synchronize data to a client, the server first finds the corresponding chunk through the given metadata. If the chunk found is a virtual one, the server fetches the corresponding content based on the offset and length of the chunk recorded. Fig. 9 shows an example. Like all other commercial systems, QuickSync does not transfer contents between two clients directly. A file is split into two chunks and uploaded to the server. Then the server takes other strategies to get three Virtual Chunks that point to the real contents. When the server needs to update or send the Virtual Chunks, it fetches the content from the storage based on its pointer.

4.3 Redundancy Eliminator

The Redundancy Eliminator is designed to eliminate the redundant sync traffic. Ideally, only the modified parts of the file need to be synchronized to the server through a technique such as delta encoding. However the effective function of delta encoding has two requirements. First, the map of the new to the old version must be identified as the input for encoding. Second, the two versions for encoding must be “similar”, otherwise executing the delta encoding will not provide any benefit but only involves additional computation overhead. As discussed in the previous section, all files in current cloud storage systems are stored as independent chunks distributedly, and the delta encoding algorithm is executed between pairs of chunks in the modified and the original file. With the fixed-size chunking, modification on file may lead to a map between two “un-similar” chunks. Also, a chunk in the middle of the uploading process cannot be compared to enable delta encoding. We employ two techniques to alleviate these problems.

4.3.1 Sketch-Based Mapping

In QuickSync, once changes are detected and the modified files are split into chunks, two similar chunks in the original and the modified files are mapped in two steps. We first compare the hashes of the new chunks with those of the original file to identify the unchanged chunks that do not need to be updated. Second, for the chunks without a hash match in the original version, we leverage a technique named *sketch* to estimate the similarity of chunks in the two versions. We only build a map between two similar chunks in the new and old versions to perform delta-encoding. The chunks without either a hash or sketch match are treated as “different” chunks and will be transferred directly. We get the sketch by identifying “features” [9] of a chunk that would not likely change when there are small data variations. In our implementation of QuickSync, we apply a rolling hash function over all overlapping small data regions, and we then choose the maximal hash value seen as one feature. We generate multiple features of the same chunk using different hash functions. Chunks that have one or more features in common are likely to be very similar, but small changes to the data are unlikely to perturb the maximal values. To better represent a chunk, we get the sketch of the chunk by calculating the XOR of four different features.

4.3.2 Buffering Uncompleted Chunks

To take advantage of the chunks transmitted in the air for the delta encoding, we introduce two in-memory queues to buffer the incomplete chunks that have been processed by the Network-aware Chunker. The *uploading queue* temporarily stores all chunks waiting to be uploaded via network communication, with each chunk recorded with three parts: the data content, the hash value and the sketch of it. New chunks from the Network-aware Chunker are pushed into this queue and popped up if they have been completely uploaded. We can thus build a map between a new chunk and the one found in the uploading queue.

To handle modification operations, we create an *updating queue* to buffer a chunk that finds a sketch match with another chunk either on the server or the local uploading queue. Each chunk in the updating queue is tagged with the hash of its matched chunk. Chunks are inserted into the updating queue if a sketch match is found and popped up when the delta encoding for two similar chunks is completed.

Algorithm 1 summarizes how Redundancy Eliminator processes chunks provided by Network-aware Chunker and eliminates redundant data before delivering chunks to the Batched Sync for transmission. Upon file modifications and the triggering of sync, files are first split into chunks by the Network-aware Chunker. Then the Redundancy Eliminator executes the two-step mapping process. The chunk without a sketch or hash match is treated as a new chunk and inserted into the uploading queue directly, while the ones found with sketch match are bundled by the Redundancy Eliminator along with their hashes and put in the updating queue. In Algorithm 1, we include an uploading process that monitors the uploading queue and delivers chunks to Batched Syncer for further uploading. We also provide an independent updating process to continuously fetch chunk from the updating queue, and then calculate the delta between the mapped chunks. The delta will be inserted into the uploading queue. Finally all data in the uploading queue are synchronized to the server by the Batched Syncer.

Algorithm 1. Sync Process at the Redundancy Eliminator

```
/*Assume that files are split as chunk_list first.*/
Two-step mapping process:
for each chunk C_i in chunk_list do
    /*Step 1: check whether C_i is redundant.*/
    if find hash(C_i) in uploading queue or cloud then
        omit redundant C_i, continue;
    end if
    /*Step 2: check whether C_i has a similar chunk.*/
    if find sketch(C_i) in uploading queue or cloud then
        map C_i to the matched one;
        add C_i to updating queue;
    else
        add C_i to uploading queue;
    end if
end for
/*Upload new chunks to the cloud.*/
Uploading process:
for each chunk C_i in uploading queue do
    deliver C_i to Batched Syncer for uploading;
    end for
/*Perform delta-encoding between mapped chunks.*/
Updating process:
for each chunk C_i in updating queue do
    calculate the delta between C_i and the mapped one;
    deliver the delta to Batched Syncer for uploading;
end for
```

4.4 Batched Syncer

The per-chunk sequential acknowledgement from the application layer and the TCP slow start are the main factors that decrease the bandwidth utilization, especially for the sync of multiple small chunks. To improve the sync efficiency, we design the Batched Syncer with two key techniques to improve the bandwidth utilization.

4.4.1 Batched Transmission

Cloud storage services leverage the app-layer acknowledgement to maintain the chunk state. As a benefit, upon a connection interruption, a client only needs to upload the unacknowledged chunks to resume the sync. Dropbox simply bundles small chunks into a large chunk to reduce the acknowledgement overhead. Although this helps improve the sync throughput, when there is a broken connection, the Dropbox client has to retransmit all small chunks if the bundled one is not acknowledged.

Our first basic technique is to defer the app-layer acknowledgement to the end of the sync process, and actively check the un-acknowledged chunks upon the connection interruption. This method on the one hand reduces the overhead due to multiple acknowledgements for different chunks and also avoids the idle waiting for the acknowledgement between two chunk transmissions. On the other hand it avoids the need of retransmitting many chunks upon a connection interruption. The check will be triggered under two conditions. First, the check will be initiated when the client captures a network exception, usually caused by the process shut down or the connection loss at the local side. Second, the failure of the sync process can be also caused by interruption in the network that cannot be easily detected by the local devices. To detect the network failure, we monitor the transmission progress to estimate if there is an exception in the network. Specifically, we monitor the data transfer progress in small time windows (e.g., a second). If there is no progress in several consecutive time windows, the Batched Syncer actively terminates the current connection and checks the control server for the missing chunks.

During the transmission, the Batched Syncer continuously sends chunks in the uploading queue of the Redundancy Eliminator. If the connection is interrupted by network exceptions or the sync process has no progress for a period of time, the client connects to the control server to query the un-acknowledged chunks, and then uploads them after the content flow is re-established.

4.4.2 Reusing Existing Network Connections

The second technique is to reuse the existing network connections rather than making new ones in storing files. While it may be easy and natural to make a new network connection for each chunk, the handshake overhead for establishing a new connection is not negligible, and creating many new connections also extends the period in the slow start state especially for small chunks. The Batched Syncer reuses the storage connection to transfer multiple chunks, avoiding the overhead of duplicate TCP/SSL handshakes. Moreover, cloud storage services maintain a persistent notification flow for capturing changes elsewhere. Hence we reuse the notification flow for both requesting notification and sending file data to reduce the handshake overhead and the impact of slow start. Specifically, both the request and data are transferred over HTTP(S), so we use the `Content-Type` field in the HTTP header to distinguish them in the same TCP connection.

5 System Implementation

To evaluate the performance of our proposed schemes, we build the QuickSync system over both Dropbox and Seafile platforms.

Implementation Over Dropbox. Since both the client and server of Dropbox are totally closed source, we are unable to directly implement our techniques with the released Dropbox software. Although Dropbox provides APIs to allow user program to synchronize data with the Dropbox server, different from the client software, the APIs are RESTful and operate at the full file level. We are unable to get the hash value of a certain chunk, or directly implement delta-encoding algorithm via the APIs.

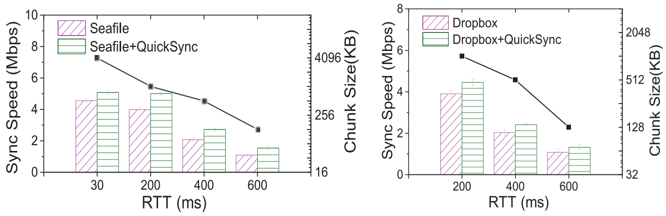
To address this problem, we leverage a proxy in Amazon EC2 which is close to the Dropbox server to emulate the control server behavior. The proxy is designed to generate the Virtual Chunks, maintain the map of file to the chunk list and calculate the hash and the sketch of chunks. During a sync process, user data are first uploaded to the proxy, and then the proxy updates the metadata in the database and stores the data to the Dropbox server via the APIs. Since the data storage server of Dropbox is also built on Amazon EC2, the bandwidth between our proxy and Dropbox is sufficient and not the bottleneck.

To make our Network-aware Chunker efficient and adjustable, we use the SAMPLEBYTE [11] as our basic chunking method. Like other content defined chunking methods, the sample period p set in SAMPLEBYTE also determines both the computation overhead and deduplication ratio. We leverage the adjustable property of p to generate a suite of chunking strategies with various deduplication ratio and computation overhead, including the chunk-based deduplication with the average chunk size set to 4, 1 MB, 512 and 128 KB. Each Virtual Chunk contains a 2-byte field for chunk length.

We use `librsync` [13] to implement delta encoding. We use a tar-like method to bundle all data chunks in the sync process, and a client communicates with our proxy at the beginning of a sync process to notify the offset and length of each chunk in the sync flow. The timer of our Syncer is set to 60 s. We write the QuickSync client and proxy in around 2,000 lines of Java codes. To achieve efficiency, we design two processes to handle chunking and transmission tasks respectively in the client. The client is implemented on a Galaxy Nexus smartphone with a 1.2 GHz Dual Core CPU, 1 GB memory and the proxy is built on an Amazon EC2 server with a 2.8 GHz Quad Core CPU and 4 GB memory.

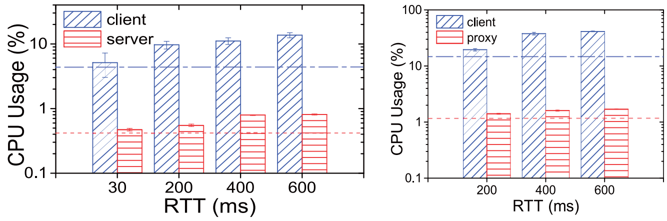
Implementation Over Seafile. Although we introduce a proxy between the client and the Dropbox server, due to the lack of full access of data on the server, this implementation suffers from the performance penalty. For instance, to perform delta encoding, the proxy should first fetch the entire chunk from the Dropbox server, update its content and finally store it back to Dropbox. Even though the bandwidth between the proxy and the Dropbox server is sufficient, such an implementation would inevitably involve additional latency during the sync process.

In order to show the gain in the sync efficiency when our system is fully implemented and can directly operate over the data, we further implement QuickSync with Seafile [14], an open source cloud storage project. The implementation is similar to that using Dropbox but without the need of a proxy. Specifically, we directly modify source codes at both the client and server sides. We modify the client in a Linux laptop with a 2.6 GHz Intel Quad Core CPU and 4 GB memory. We build the server on a Linux machine with a 3.3 GHz Intel Octal Core CPU and 16 GB memory, as only the Seafile software on Linux platform is open source. Techniques in QuickSync can also be implemented in the similar way on other mobile platforms.



(a) Seafile platform (b) Dropbox platform

Fig. 10. Speed improved by network-aware Chunker.



(a) QuickSync on Seafile (b) QuickSync on Dropbox

Fig. 11. CPU overhead of network-aware Chunker.

6 Performance Evaluation

To evaluate the performance of our schemes, we first investigate the throughput improvement of using the Network-aware Chunker, and then show that the Redundancy Eliminator is able to effectively reduce the sync traffic. We further evaluate the capability of the Batched Syncer in improving the bandwidth utilization efficiency. Finally, we study the overall improvement of the sync efficiency using real-world workloads. In each case, we compare the performances of the original Seafile and Dropbox clients with those when the two service frameworks are improved with QuickSync.

6.1 Impact of the Network-Aware Chunker

We first evaluate how the Network-aware Chunker improves the throughput under various network conditions. We collect about 200 GB data from 10 cloud storage services users, and randomly pick about 50 GB as the data set for uploading. The rest about 150 GB data are pre-stored on the server for deduplication purpose. We repeat the sync process under various RTT to measure the sync speed, defined as the ratio of the original data size to the total sync time, and the average CPU usage of both the client and server. The minimal RTT from our testbed to the Seafile and Dropbox server is 30 and 200 ms respectively.

In Fig. 10a, when the RTT is very low (30 ms), since the bandwidth is sufficient, the client selects the un-aggressive chunking strategy with low computation overhead to split files, and the sync speed outperforms the original one by 12 percent. In Fig. 10b, the Network-aware Chunker is shown to adaptively select smaller average chunk size in a poor network condition to eliminate more redundancy and reduce the total sync time. Thus, although the sync speed decreases at higher RTT, our scheme can still achieve a higher total sync speed by selecting a smaller average chunk size with the aggressive chunking strategies to eliminate more redundancy and thus reduce the transmission time. Overall, our implementations can dynamically select an appropriate chunking strategy for deduplication, which leads up to about 31 percent increase of the sync speed under various network conditions.

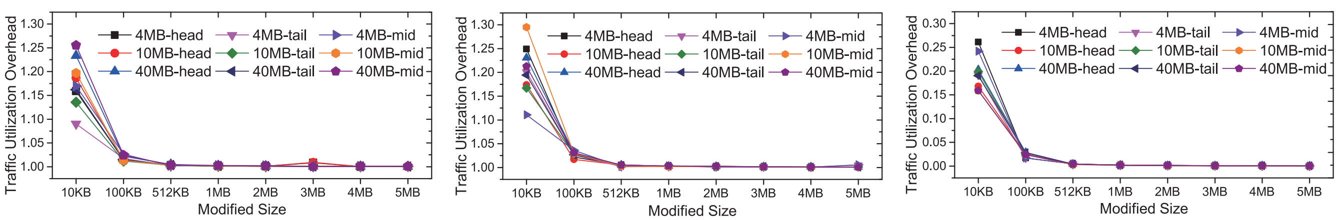
We plot the CPU usages of QuickSync client and server in Fig. 11. Since the original systems do not change their chunking strategies based on network conditions, we also plot their constant CPU usages as the baseline. As RTT increases, the CPU usages for both the client and server of QuickSync increase, as more aggressive chunking strategy is applied to reduce the redundant data. The CPU usage for Seafile is lower because of more powerful hardware. The CPU usage of client reaches up to 12.3 and 42.7 percent in two implementations respectively which is still within the acceptable range.

6.2 Impact of the Redundancy Eliminator

Next we evaluate the sync traffic reduction of using our Redundancy Eliminator with the average chunk size set to 1 MB to exclude the impact of adaptive chunking. We conduct the same set of experiments for modify operation as shown in Fig. 2, and measure the sync traffic size to calculate their TUO.

In Fig. 12, for both flip and insert operations, the TUO of our mechanism for all files in any position is close to 1, indicating that our implementation only synchronizes the modified content to server. The TUO results for flip or insert operation on small files (≤ 100 KB) have reached 1.3, where the additional traffic is due to the basic overhead of delta encoding. The TUO results for delete operation are close to 0 because the client does not need to upload the delta besides performing the delta encoding. The results of Dropbox modification are similar and omitted due to the page limit.

Furthermore, to evaluate the traffic reduction for synchronizing changes of file whose corresponding chunks are on their way to the server, we conduct the same set of experiments as those in Fig. 3 with the results shown in Table 3. The TUO results in each case are close to 1. Our scheme only needs to synchronize the new contents under arbitrary number of modifications and any RTT, with our in-memory uploading queue to buffer files in the middle of transmissions to facilitate the delta encoding.



(a) flip (b) insert (c) delete

Fig. 12. Traffic utilization overhead reduction of Seafile modification.

TABLE 3
TUO of Sync Process

RTT (ms)	Seafile+QuickSync			Dropbox+QuickSync		
	# = 1	# = 3	# = 5	# = 1	# = 3	# = 5
30	1.2306	1.1795	1.1843	-	-	-
200	1.1152	1.2742	1.1834	1.1067	1.1777	1.2814
400	1.2039	1.2215	1.2420	1.1783	1.1585	1.2978
600	1.2790	1.1233	1.2785	1.2268	1.2896	1.1865

During the uploading process, modifications are performed in the being synced files.

6.3 Impact of the Batched Syncer

6.3.1 Improvement of BUE

To examine the performance of the Batched Syncer in improving the bandwidth utilization, we set the average chunk size to 1 MB to exclude the impact of adaptive chunking. In Section 2.4, we observe that cloud storage services suffer low BUE, especially when synchronizing a lot of small files. We conduct the same set of experiments with use of our proposed schemes.

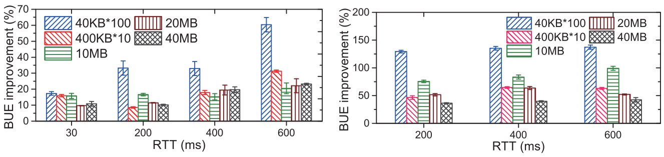
Fig. 13 shows the level of BUE improvement under different network conditions. When synchronizing a batch of chunks, the reduction of the acknowledgement overhead helps improve the bandwidth utilization efficiency up to 61 percent. The improvement is more obvious in high RTT environment where the throughput often experiences big reduction especially when the acknowledgements are frequent.

6.3.2 Overhead for Exception Recovery

The per chunk acknowledgement is designed to reduce the recovery overhead when the sync process is unexpectedly interrupted. In our Batched Syncer, the client will not wait for an acknowledgement for every chunk. Now we examine whether this design will cause much more traffic overhead for exception recovery. We upload a set of files with different sizes, and close the TCP connection when half of the file has been uploaded. After the restart of the program, the client will create a new connection to finish the sync. We record the total sync traffic and calculate the TUO in Fig. 14. Our results show that in each case, the TUO of QuickSync is close to 1, and the highest TUO is only about 1.5, indicating that our implementations will not cause very high overhead for exception recovery. In our design, before resuming the sync, the client communicates with the server first to check the chunks that are not received and need to be transferred.

6.4 Performance of the Integrated System

Now we assess the overall performance of our implementation using a series of representative workloads for cloud storage services on Windows or Android. Each workload combines a set of file operation events, including file



(a) Seafile+QuickSync (b) Dropbox+QuickSync

Fig. 13. BUE improvement.

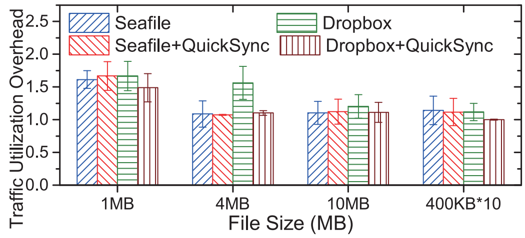


Fig. 14. Recovery overhead.

creation, modification or deletion, which will trigger corresponding events in the local file system. The event number in each workload and performance results are shown in Table 4. We compare the sync performance of QuickSync with other two alternatives. LBFS [7] is a low-bandwidth remote file system that leverages the fine-granularity content-defined chunking to identify and reduce the redundant sync traffic. EndRE [11] is an end-system redundancy elimination service. We also show the performance results of the original system as the baseline in our evaluation.

We first generate the workloads on Windows platform based on Seafile and its modification. The QuickSync Paper workload is resulted from uploading the files of this paper, and the Seafile Source generates load by storing all the source codes of the Seafile. Both types of workload contain a lot of small files and do not contain the file modification or deletion. Compared to the original system, although the traffic size reduction for the two workloads are small (7.5 and 8.9 percent), our implementation reduces the total sync time by 35.1 and 51.8 percent respectively. The reduction is mainly caused by bundling the small files to improve the bandwidth utilization, as the Seafile Source contains 1,259 independent files. The Document Editing workload on Windows is generated when we naturally edit a PowerPoint file in the sync folder from 3 to 5 MB within 40 min. We capture many creation and deletion events during the editing process, as temporary files whose sizes are close to that of the original .ppt file are created and deleted. Changes are automatically synchronized. Our solution significantly reduces the traffic size, with QuickSync to execute the delta encoding on the temporary files in the middle of the sync process to reduce the traffic. The Data Backup workload on Windows is a typical usage for large data backup. This workload contains 37,655 files, with various file types (e.g., PDF or video) and sizes (from 1 KB to 179 MB). Our QuickSync achieves 37.4 percent sync time reduction by eliminating the redundancy and reducing the acknowledgement overhead to improve the bandwidth utilization.

We also play the workload on Android platform. The Document Editing workload on Android is similar to that generated in the above experiment but contains fewer modifications. Our implementation reduces 41.4 percent of the total sync time. The Photo Sharing is a common workload for mobile phones. Although the photos are often in the encoded format and hard to be deduplicated, our implementation can still achieve 24.1 percent time saving through the batched transmission scheme. The System Backup workload is generated to back up all system settings, app binaries together with app configurations in a phone via a slow 3G network. As our implementation adaptively selects aggressive chunking strategy to eliminate larger amount of the backup traffic

TABLE 4
Practical Performance Evaluation for QuickSync Using a Series of Real World Representative Workloads

Workload (Platform)	# of Events			Traffic Size				Sync Time			
	C	M	D	Origin	QSync	LBFS	EndRE	Origin	QSync	LBFS	EndRE
QuickSync Paper (W)	74	0	0	4.67 MB	4.32 MB	4.18 MB	4.47 MB	27.6 s	17.9 s	31.4 s	19.8 s
Seafile Source (W)	1,259	0	0	15.6 MB	14.2 MB	13.7 MB	14.9 MB	264.1 s	127.3 s	291.8 s	174.1 s
Document Editing (W)	12	74	7	64.3 MB	12.7 MB	57.3 MB	60.2 MB	592.0 s	317.3 s	514.8 s	488.2 s
Data Backup (W)	37,655	0	0	2 GB	1.4 GB	1.1 GB	1.6 GB	68.7 m	43.1 m	83.4 m	55.6 m
Document Editing (A)	1	4	0	4.1 MB	1.5 MB	3.7 MB	3.9 MB	24.4 s	14.3 s	46.8 s	21.9 s
Photo Sharing (A)	11	0	0	21.1 MB	20.7 MB	20.2 MB	20.6 MB	71.9 s	54.6 s	133.6 s	65.2 s
System Backup (A)	66	0	0	206.2 MB	117.9 MB	96.4 MB	136.9 MB	612.3 s	288.7 s	762.4 s	402.8 s
App Data Backup (A)	17	0	0	66.7 MB	36.6 MB	34.9 MB	41.3 MB	465.7 s	125.0 s	271.4 s	247.9 s

We compare the sync performance with the original system, LBFS [7], and EndRE [11]. W: Windows platform. A: Android platform. Event C: Creation. Event M: Modification. Event D: Deletion.

and bundles chunks to improve the bandwidth utilization, 52.9 percent sync time saving is achieved. App Data Backup is a workload generated when we walk in the outdoor environment while using a phone in a LTE network to back up the data and configurations of several specified applications. As the network condition changes during our movement, QuickSync dynamically selects the proper chunking strategy to eliminate the redundant data, which reduces 45.1 percent sync traffic and 73.1 percent total sync time.

Interestingly, for most workloads in our experiment LBFS achieves the lowest traffic size in the sync process, but the total sync time of LBFS is larger than other solutions. This is because LBFS leverages a very aggressive deduplication strategy that chops files into small chunks and identifies redundant data by checking hash values. However, the aggressive strategy does not always improve the sync efficiency since it is computation-intensive in the resource-constraint mobile platform. In addition, the effectiveness of deduplication degrades for compressed workloads (e.g., photo sharing). QuickSync outperforms LBFS and EndRE by adaptively selecting the proper chunking strategy according to current network conditions, and wisely executing delta encoding during file editing.

7 Related Work

Measurement Study. Recently a large number of measurement research efforts have been conducted on enterprise cloud storage services [15], [16], [17], [18] and personal cloud storage services [2], [19], [20], [21], [22], [23], [24], [25].

Focusing on the enterprise cloud storage services, CloudCmp [15] measures the elastic computing, persistent storage, and networking services for four major cloud providers. The study in [16] provides a quantitative analysis of the performance of the Windows Azure Platform. Works in [17] perform an extensive measurement against Amazon S3 to elucidate whether cloud storage is suitable for scientific grids. Similarly, [18] presents a performance analysis of the Amazon Web Services. However these studies provide no insights into personal cloud storage services, while our measurement study focuses on the emerging personal cloud storage services in mobile/wireless environments.

Some literature studies also attempt to analyze the performance of personal cloud storage services. To our best knowledge, Hu et al. first analyze personal cloud storage services by comparing the performance of Dropbox, Mozy, Carbonite and CrashPlan [24]. However, they only provide an incomplete analysis on the backup/restore time for several types of files. Gracia-Tinedo et al. study the REST interfaces provided by three big players in the personal cloud storage arena [22], and more recently they conduct a measurement study of the internal structure of UbuntuOne [21]. Drago et al. give a large-scale measurement for Dropbox [19], and then compare the system capabilities for five popular cloud storage services in [2]. However, all these previous studies only focus on the desktop services based on black-box measurement. Li et al. give the experimental study of the sync traffic, demonstrating that a considerable portion of the data sync traffic is wasteful [20]. Our work steps closer to reveal the root cause of inefficiency problem from the protocol perspective, and we are the first to study the sync efficiency problem in mobile/wireless networks where the network condition (e.g., RTT) may change significantly.

System Design. There are many studies about the system design for cloud storage services [26], [27] but they mostly focus on enterprise backup instead of the personal cloud. UniDrive [28] is designed to boost the sync performance of personal cloud storage services by leveraging multiple available clouds to maximize the parallel transfer opportunities. However, relying on existing cloud storage platforms, UniDrive is not able to address the sync inefficiency problems we identified in existing personal cloud storage services. An adaptive sync defer (ASD) mechanism is proposed to adaptively defer the sync process to follow the latest data update [29]. The bundling idea of our Batched Syncer is similar to ASD, but ASD incurs much more recovery overhead when the sync is interrupted. Moreover, as a middleware solution, ASD can not avoid the incremental sync failure described in Section 2.3. QuickSync addresses the sync failure problem by applying our sketch-based redundancy elimination. ViewBox [30] is designed to detect the corrupted data through the data checksum and ensure the consistency by adopting view-based synchronization. It is complemented with our QuickSync system.

CDC and Delta Encoding. QuickSync leverages some existing techniques, such as content defined chunking [7], [8], [9], [10], [11], [14], [31], [32] and delta encoding [4]. Rather than directly using these schemes, the aim of QuickSync is to design best strategies to adjust and improve these techniques for better supporting mobile cloud storage services. In all previous systems using CDC, both the client and server use the fixed average chunk size. In contrast, QuickSync utilizes CDC addressing for a unique purpose, adaptively selecting the optimized average chunking size to achieve the sync efficiency. Delta encoding is also not a new idea but it poses big challenge when implemented with the cloud storage system where files are split into chunks and stored distributedly. The techniques proposed in our Redundancy Eliminator leverage the sketch of chunks to address the limitation and wisely perform delta encoding on similar chunks to reduce the sync traffic overhead.

8 Discussion

In this section, we discuss other issues in deploying and using QuickSync to improve the sync efficiency for mobile cloud storage services.

Why QuickSync Focuses on Upload Traffic. In our current design of QuickSync, we mainly focus on improving the sync efficiency of the upload transmission for two key reasons. First, the dominant traffic of most traditional mobile applications, such as web browser, streaming application, or news reader incur the download traffic. Hence a number of previous efforts have studied on the download transmission optimization in mobile/wireless environments [10], [11], [31]. However as an emerging and popular services, mobile cloud storage generates significant upload traffic which is rarely studied in previous works. Second, typically in an LTE/3G network, the upload throughput is much less than the download throughput [3]. Therefore, it is necessary and more important to improve the sync efficiency for the upload traffic of cloud storage services in a mobile environment.

Energy Consumption. In this paper we mostly focus on the sync efficiency of mobile cloud storage services. Due to the limited battery drain of mobile devices, energy consumption is another important performance metric for the mobile sync services [33]. It is hard to give a conclusion whether QuickSync will cause additional energy consumption for mobile devices. This is because QuickSync improves the sync efficiency by increasing the bandwidth utilization and reducing the volume of sync traffic. Although our techniques may cause additional computation overhead in certain scenarios, these techniques also effectively reduce the data transmission time as well as the energy caused by network interfaces. Generally, the transmission energy consumption is more significant. However, we would not claim that QuickSync reduce the energy consumption, but will study the energy problem in the future.

Deployment of QuickSync. QuickSync can be deployed in current cloud storage services by adding a QuickSync proxy between the client and the server, or updating the existing server-side infrastructure to incorporate these new techniques provided by QuickSync. A proxy-based implementation is easier for deployment but involves more computation and storage overhead since it requires the proxy to temporarily store the intermediate state of the sync process, while a full implementation of QuickSync can achieve better performance but needs to update the server. Besides, a proxy-based implementation is also complemented with multi-cloud system [28] which is built on multiple existing cloud providers to obtain better reliability and security.

9 Conclusion

Despite their near-ubiquity, mobile cloud storage services fail to efficiently synchronize data in certain circumstance. In this paper, we first study four popular cloud storage services to identify their sync inefficiency issues in wireless networks. We then conduct the in-depth analysis to give the root causes of the identified problems with both trace studies and data decryption. To address the inefficiency issues, we propose QuickSync, a system with three novel techniques. We further implement QuickSync to support the sync operation with Dropbox and Seafile. Our extensive evaluations demonstrate that QuickSync can effectively save the sync time and reduce the significant traffic overhead for representative sync workloads.

Acknowledgments

We acknowledge the support of NSFC project (no. 61422206) and National 863 project (no. 2015AA015701). We also acknowledge the support of NSF CNS 1526843.

References

[1] Y. Cui, Z. Lai, and N. Dai, “A first look at mobile cloud storage services: Architecture, experimentation and challenge,” [Online]. Available: http://www.4over6.edu.cn/others/technical_report.pdf
[2] 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.
[3] A. Balasubramanian, R. Mahajan, and A. Venkataramani, “Augmenting mobile 3G using WiFi,” in *Proc. 8th Int. Conf. Mobile Syst. Appl. Services*, 2010, pp. 209–222.
[4] A. Tridgell and P. Mackerras, “The rsync algorithm,” 1996.
[5] D. Kholia and P. Wegrzyn, “Looking inside the (drop) box,” in *Proc. 7th USENIX Workshop Offensive Technol.*, 2013, pp. 9–9.
[6] Dynamorio. [Online]. Available: http://dynamorio.org
[7] A. Muthitacharoen, B. Chen, and D. Mazieres, “A low-bandwidth network file system,” in *Proc. 18th ACM Symp. Operating Syst. Principles*, 2001, pp. 174–187.
[8] 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, Art. no. 18.
[9] P. Shilane, M. Huang, G. Wallace, and W. Hsu, “WAN-optimized replication of backup datasets using stream-informed delta compression,” *ACM Trans. Storage*, vol. 8, 2012, Art. no. 13.
[10] Y. Hua, X. Liu, and D. Feng, “Neptune: Efficient remote communication services for cloud backups,” in *Proc. IEEE INFOCOM*, 2014, pp. 844–852.
[11] B. Agarwal, et al., “EndRE: An end-system redundancy elimination service for enterprises,” in *Proc. 7th USENIX Conf. Netw. Syst. Des. implementation*, 2010, pp. 28–28.
[12] S.-H. Shen and A. Akella, “An information-aware QoE-centric mobile video cache,” in *Proc. 19th Annu. Int. Conf. Mobile Comput. Netw.*, 2013, pp. 401–412.
[13] librsync. [Online]. Available: http://librsync.sourceforge.net/
[14] Seafile source code. [Online]. Available: https://github.com/haiwen/seafile
[15] A. Li, X. Yang, S. Kandula, and M. Zhang, “CloudCmp: Comparing public cloud providers,” in *Proc. 10th ACM SIGCOMM Conf. Internet Meas. Conf.*, 2010, pp. 1–14.
[16] Z. Hill, J. Li, M. Mao, A. Ruiz-Alvarez , and M. Humphrey, “Early observations on the performance of Windows Azure,” in *Proc. 19th ACM Int. Symp. High Perform. Distrib. Comput.*, 2010, pp. 367–376.
[17] M. R. Palankar, A. Iamnitchi, M. Ripeanu, and S. Garfinkel, “Amazon S3 for science grids: A viable solution?” in *Proc. Int. Workshop Data-Aware Distrib. Comput.*, 2008, pp. 55–64.
[18] A. Bergen, Y. Coady, and R. McGeer , “Client bandwidth: The forgotten metric of online storage providers,” in *Proc. IEEE Pacific Rim Conf. Commun. Comput. Signal Process.*, 2011, pp. 543–548.
[19] 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.
[20] Z. Li, et al., “Towards network-level efficiency for cloud storage services,” in *Proc. Conf. Internet Meas. Conf.*, 2014, pp. 115–128.
[21] R. Gracia-Tinedo, et al., “Dissecting UbuntuOne: Autopsy of a global-scale personal cloud back-end,” in *Proc. Conf. Internet Meas. Conf.*, 2015, pp. 155–168.

[22] R. Gracia-Tinedo, M. Sanchez Artigas, A. Moreno-Martinez, C. Cotes, and P. Garcia Lopez, "Actively measuring personal cloud storage," in *Proc. IEEE 6th Int. Conf. Cloud Comput.*, 2013, pp. 301–308.

[23] T. Mager, E. Biersack, and P. Michiardi, "A measurement study of the Wuala on-line storage service," in *Proc. IEEE 12th Int. Conf. Peer-to-Peer Comput.*, 2012, pp. 237–248.

[24] W. Hu, T. Yang, and J. N. Matthews, "The good, the bad and the ugly of consumer cloud storage," *ACM SIGOPS Operating Syst. Rev.*, vol. 44, pp. 110–115, 2010.

[25] Z. Li, et al., "An empirical analysis of a large-scale mobile cloud storage service," in *Proc. Conf. Internet Meas. Conf.*, 2016, pp. 287–301.

[26] M. Vrable, S. Savage, and G. M. Voelker, "BlueSky: A cloud-backed file system for the enterprise," in *Proc. 10th USENIX Conf. File Storage Technol.*, 2012, pp. 19–19.

[27] B. Calder, et al., "Windows azure storage: A highly available cloud storage service with strong consistency," in *Proc. 23rd ACM Symp. Operating Syst. Principles*, 2011, pp. 143–157.

[28] H. Tang, F. Liu, G. Shen, Y. Jin, and C. Guo, "UniDrive: Synergize multiple consumer cloud storage services," in *Proc. 16th Annu. Middleware Conf.*, 2015, pp. 137–148.

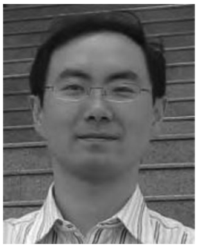
[29] Z. Li, et al., "Efficient batched synchronization in dropbox-like cloud storage services," in *Proc. 14th ACM/IFIP/USENIX Annu. Middleware Conf.*, 2013, pp. 307–327.

[30] Y. Zhang, C. Dragga, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "ViewBox: integrating local file systems with cloud storage services," in *Proc. 6th USENIX Conf. File Storage Technol.*, 2014, pp. 119–132.

[31] N. T. Spring and D. Wetherall, "A protocol-independent technique for eliminating redundant network traffic," in *Proc. Conf. Appl. Technol. Architectures Protocols Comput. Commun.*, 2000, pp. 87–95.

[32] A. Anand, A. Gupta, A. Akella, S. Seshan, and S. Shenker, "Packet caches on routers: The implications of universal redundant traffic elimination," in *Proc. ACM SIGCOMM Conf. Data Commun.*, 2008, pp. 219–230.

[33] Y. Cui, S. Xiao, X. Wang, M. Li, H. Wang, and Z. Lai, "Performance-aware energy optimization on mobile devices in cellular network," in *Proc. IEEE INFOCOM*, 2014, pp. 1123–1131.



Yong Cui received the BE and PhD degrees both on computer science and engineering from Tsinghua University. He is currently a full professor in the Computer Science Department, Tsinghua University. Co-chairing an IETF WG, he served or serves on the editorial boards on the *IEEE Transactions on Parallel and Distributed Systems*, the *IEEE Transactions on Cloud Computing*, the *IEEE Network*, and the *IEEE Internet Computing*. He published more than 100 papers with several Best Paper Awards and seven Internet standard documents (RFC). His research interests include mobile cloud computing and network architecture.

Zeqi Lai received the bachelor's degree in computer science from the University of Electronic Science and Technology of China, in 2009. He is now working toward the PhD degree of the Department of Computer Science and Technology, Tsinghua University, China. His supervisor is Prof. Yong Cui. His research interests include networking and cloud computing.

Xin Wang received the BS and MS degrees in telecommunications engineering and wireless communications engineering, respectively, from the Beijing University of Posts and Telecommunications, Beijing, China, and the PhD degree in electrical and computer engineering from Columbia University, New York. She is currently an associate professor in the Department of Electrical and Computer Engineering, State University of New York at Stony Brook, Stony Brook, New York. She has served in many reputed conferences and journals, and is currently the associate editor of the *IEEE Transactions on Mobile Computing*.

Ningwei Dai received the BEng degree in communication engineering from the Beijing University of Posts and Telecommunications, Beijing, China, in 2015. She is currently working toward the master's degree in the Department of Computer Science and Technology, Tsinghua University, Beijing, China. Her research interests include networking and cloud computing.