QuickSync: Improving Synchronization Efficiency for Mobile Cloud Storage Services

Yong Cui
Department of Computer Science and Technology
Tsinghua University
Beijing, China
cuiyong@tsinghua.edu.cn

Zeqi Lai
Department of Computer Science and Technology
Tsinghua University
Beijing, China
laizq13@mails.tsinghua.edu.cn

Xin Wang
Department of Electrical and Computer Engineering
Stony Brook University
Stony Brook, New York, USA
xwang@ece.sunysb.edu

Ningwei Dai
Department of Computer Science and Technology
Tsinghua University
Beijing, China
lemondnw@gmail.com

Congcong Miao
Department of Computer Science and Technology
Tsinghua University
Beijing, China
mccmiao@163.com

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 circumstance even though the incremental sync is implemented. 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 52.9% sync time in our experiment settings.

Categories and Subject Descriptors: C.4 [Performance of Systems]

Keywords: Mobile Cloud Storage, Measurement, Performance.

1. INTRODUCTION

Cloud storage services, such as Dropbox [13], OneDrive [4] (used to be SkyDrive) and GoogleDrive [2], are expanding their mobile market by enabling users to conveniently synchronize files across multiple mobile devices (laptops, tablets or smartphones) and back up data. These services are gaining tremendous popularity in recent years, and have attracted a huge number of users.

As a primary technique for 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 other devices with low traffic overhead.

However, efficient sync of data is more challenging for mobile cloud storage services as the client often suffers higher delay and loss caused by the mobility and varying channel conditions. The sync process may also be interrupted due to the intermittent connections. Therefore, the sync efficiency of popular services is still far from being satisfactory, and in certain circumstances, the sync time may be much longer than expected. As commercial storage services are mostly closed source with data encrypted, their designs and operational processes remain unclear to the public. It is important but a big challenge to identify the performance bottlenecks and address the issues accordingly.

The aim of this paper is to identify and address the sync inefficiency problem of 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 wireless environment, 2) providing detailed analysis on the problems identified, and 3) proposing a new mobile cloud storage service framework 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 wireless networks (§3). 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 synchronizing multiple small files. Furthermore, although some services, e.g. Drop-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
MobiCom'15, September 7–11, 2015, Paris, France.
© 2015 ACM. ISBN 978-1-4503-3619-2/15/09…$15.00.
DOI: http://dx.doi.org/10.1145/2789168.2790094.

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	√	√	×	×	×	×	×	×

Table 1: Capability implementation of four popular cloud storage services. The var. refers to variable chunk size.

box, have implemented incremental sync to reduce traffic size [11, 17], 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 find the root cause of the inefficiency identified in the sync protocol (§4). Our studies indicate that the inherent limitations of the sync protocol and the distributed architecture are two major factors that cause the inefficiency. Specifically, effective 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 and when the network is slow.

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 (§5). To reduce the sync time, we introduce *Network-aware Chunker* to adaptively select the proper deduplication strategy based on real-time network conditions. To reduce 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 [5], an popular open source personal cloud system (§6). Collectively, these techniques achieve significant improvement in the sync latency for cloud storage services. Evaluation results (§7) show that the QuickSync system is able to significantly improve sync efficiency, reducing up to 51.8% 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. BACKGROUND

Before analyzing 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 [10]: the *client*, the *control server* and the *data storage server*. The file system on the server side has an abstraction different from that of the client. Metadata (including the hashes, modified time etc.) and contents (often split into chunks) of user files are separated and stored in the control and data storage servers respectively. The key operation of the cloud storage services is *data sync*, which automatically maps the changes in users' local file system to the cloud via a series of network communications. 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. 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*. Previous work [11] shows how the capabilities have been implemented on the desktop client. We further follow the methods in [11] to analyze the capabilities that have been implemented on the mobile client. 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 the following sections, we will show that these capabilities also make a strong side impact on the sync efficiency.

3. SYNCHRONIZATION INEFFICIENCY

As discussed previously, sync efficiency indicates how fast client can update changes to the cloud. In this section, we conduct a series of experiments to investigate the sync inefficiency issue existing in four most popular commercial cloud storage service systems in wireless/mobile environments. We will further analyze the observations and explain the root cause in Section 4.

3.1 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 identified 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 wireless/cellular networks [8]. We only perform measurement on the Windows platform

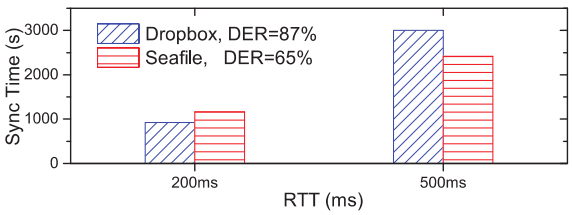


Figure 1: Lower DER does not always make efficient.

because most services did not implement the deduplication on the Android platform. We collect about 500MB 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.

Figure 1 shows that the DER for Dropbox and Seafile are 87% and 65% respectively under each RTT setting. Intuitively, a higher DER would take more time for sync. However, when the RTT is 200ms, it costs more time for Seafile to complete the sync as compared to Dropbox.

3.2 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 [24]) to achieve *incremental sync* instead of *full-file sync*. However, next we will show that the incremental sync *is not* always available, and the client software may synchronize much more data than expected. We use the metric *Traffic Utilization Overhead (TUO)* [17], defined as the ratio of *generated traffic size* to *expected traffic size*, to evaluate how much additional traffic is incurred. We conduct two sets of experiments to find out when the claimed incremental sync mechanism fails.

In experiment 1, we perform three types of basic operation in typical real-world usage patterns: *flip bits*, *insert* and *delete* over continuous w 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. Thereby, we have $TUO = \frac{SyncTrafficSize}{w}$. Since 10KB is the recommended default window size in the original delta encoding algorithm [24], we vary w from 10KB to 5MB 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. Operation in each case is performed 10 times to get the average result. Because Seafile, 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.

Figure 2 displays our results. Interestingly, the three types of operation result in totally different traffic size for Dropbox. For the flip operation, in most cases the TUO is close to 1. Even when the modification window is 10KB, the TUO is less than 1.75, indicating that incremental sync works well for flip operations in any position. The sync traffic of insert operation is closely related to the position of the modification. The TUO is close to 1 when inserting 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 3MB data at the

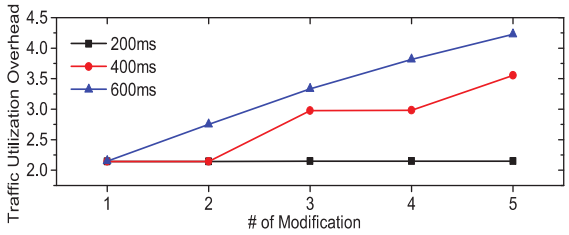


Figure 3: TUO of synchronizing modification in the middle of sync process.

head or random position of a 40MB file results in nearly 40MB 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. 40MB (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 4MB, where the TUO is close to 1 for the insert operation and close to 0 for the delete operation.

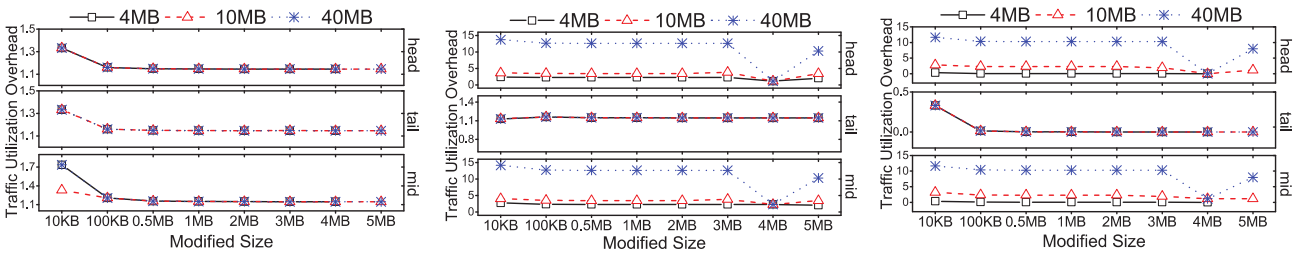
In experiment 1 all operations are performed on synchronized files (both the metadata and contents are completely updated to the cloud). In experiment 2, we investigate the sync traffic of performing modification on being synchronized files, i.e., the files in the middle of the transmissions to the cloud. We first create a 4MB fresh file in the sync folder, and perform the same flip operation as that in experiment 1 at a random position with the modification window $w = 512KB$ in every 20s. Note that the TUO of such an operation is close to 1 in the experiment 1, and in the experiment 2, the flip operation is performed *immediately* after 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 modify 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.

Figure 3 shows the sync traffic for periodic flip on 4MB file with various RTT. Interestingly, for all cases the TUO is larger than 2, indicating that at least 8MB data are synchronized. Moreover, we observe that the TUO is affected by the RTT. When the RTT is 600ms, surprisingly the TUO raises with the increase of the modification times. The sync traffic researches up to 28MB, 448% of the new content size when the modifications are performed five times.

In summary, 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 on the being synchronized files, causing undesired traffic overhead.

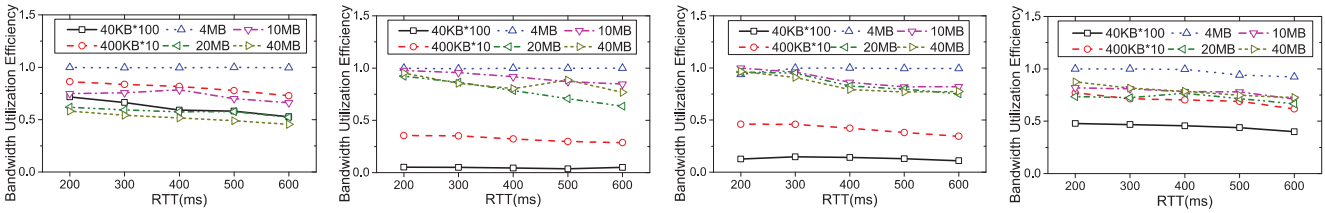
3.3 Bandwidth Inefficiency

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) flip (b) insert (c) delete

Figure 2: Traffic utilization overhead of Dropbox generated by a set of modifications.



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

Figure 4: Bandwidth utilization efficiency of 4 cloud storage services in various network conditions

a storage service is used by directly measuring the throughput or latency [11]. To characterize the network usage of sync protocol, we introduce a novel metric, *Bandwidth Utilization Efficiency (BUE)* to characterize the network usage of sync protocol. We define BUE 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. Compared with the throughput, BUE better reveals the essential bandwidth utilization capability of cloud storage services.

To investigate how the sync protocol utilizes the underlying network bandwidth, we conduct experiments over Dropbox, GoogleDrive, OneDrive and Seafile with their Windows and Android clients running 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 get 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 is measured for all services in WiFi networks with different RTT, as shown in Figure 4. For each service, the BUE of synchronizing 4MB file is close to 1. The traffic size of syncing 40KB*100 files is close to that of 4MB file, but we observe that the BUE slumps significantly when synchronizing multiple files. This degradation is more serious for GoogleDrive and OneDrive, with their BUE dropping under 20% when syncing 40KB*100 files. For all services, BUE decreases for large files such as 20MB or 40MB 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 be omitted due to the page limit.

4. ROOT CAUSE ANALYSIS

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

4.1 Pinning Down the Sync Protocol

To understand the reasons of sync inefficiency, it is difficult to directly analyze the sync protocol of commercial services such as Dropbox, as they are close source and most of the network traffic is encrypted. In our work, we exploit both measurement and decryption to understand the sync protocol. Specifically, we first analyze the network traces of all services studied in Section 3 to show the general sync process, and then we hijack the encrypted traffic of Dropbox so that we can understand the details of the sync protocol.

Commonality analysis: Although all tested services encrypt their sync traffic and we are unable to directly obtain the protocol details, we still can get some high-level knowledge of the protocol by analyzing the packet-level network traces. Our analyses on the traces in Section 3 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.

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 [15] to hi-

jack the SSL socket by DynamoRIO [1]. Although we only decrypt the Dropbox protocol, combining the commonality analysis we believe the other three services may follow a sync protocol similar to that of Dropbox.

Figure 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 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 refresh the metadata.

4.2 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 extensive, and the time cost and effectiveness of deduplication are strongly impacted by the chunking method. Fixed-size chunking used by Dropbox is simple and fast, but is less effective in deduplication. Content defined chunking [20] used by Seafile is more complex and computation extensive, but can identify a larger amount of redundancy. In our experiment in Section 3.1, when RTT is 200ms, Seafile uses the content defined chunking to achieve 65% 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 500ms 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.

4.3 Why the Traffic Overhead Increases

Although delta encoding is a mature and effective method, it is not implemented in all cloud storage services. One possible reason is that most delta encoding algorithms work at the granularity of file, while to save the storage space thus reducing the 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.

Instead, 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 chunks, and delta encoding is adopted between the two chunks. Figure 6 shows how Dropbox performs delta encoding at the granularity of chunk when inserting 2MB data at the head of a 10MB file. When the file is modified, the client follows the fixed-size chunking method to split and re-index the file. After re-indexing, the chunks without hash change are not processed further, so the TUO results of 4MB window size in Figure 2 are all close to 1. Otherwise, a map is built based on the relative locations of the original and modified versions, and delta encoding is executed between mapped chunk. Thus the delta of C1' and C1 is 2MB and the total delta is 6MB, 3 times of the insertion size. Therefore, now we can clearly understand our observations in Figure 2. Inserting 3MB data at the head of 40MB file causes nearly 40MB total sync traffic, because after the re-indexing, all chunks are mapped to different parents. In this case, the incremental sync fails to only update the changed content.

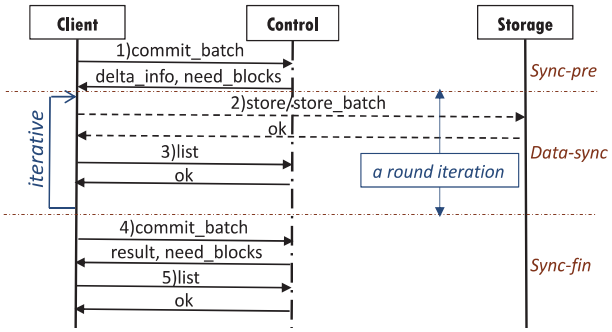


Figure 5: A typical sync process of Dropbox

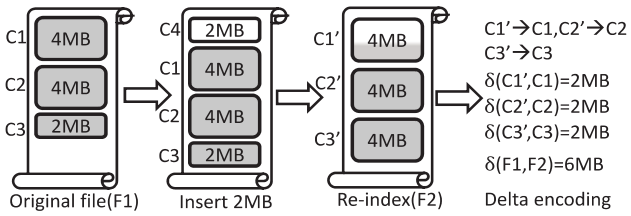


Figure 6: An example to explain why the incremental sync fails in Dropbox.

As discussed in Section 4.1, 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 experiment 2 in Section 3.2, 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=600ms, every modification is performed during the uploading process, and each modified version has to be uploaded entirely.

4.4 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, as the client has to wait for an acknowledgement from the server before transferring the next chunk, 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 4MB) to mitigate the problem, we can still see the throughput slumps between two iterations when synchronizing large files (e.g. 40MB). Different from other storage services, when transferring multiple big chunks at 4MB, Dropbox opens up to 4 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, we observe that 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.

5. 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.

5.1 System Overview

To efficiently complete a sync process, our QuickSync system introduces three key components: the Network-aware Chunker (§5.2), the Redundancy Eliminator (§5.3), and the Batched Syncer (§5.4). The basic functions of the three components can be summarized 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.*

Figure 7 shows the basic architecture of QuickSync. The sync process begins upon detecting a change in the sync folder (e.g. add or modify a file). First, the Network-aware Chunker splits an input file through content defined chunking with the chunk size determined based on the network condition. Metadata and contents are then delivered to the Redundancy Eliminator, where redundant chunks are removed and delta encoding is executed between similar chunks to reduce the sync traffic for modification operations. A database is applied to store metadata of local files. Finally the Batched Syncer leverages a delay-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.

5.2 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 [28], improving the efficiency for large-scale remote backup [22, 14], or only optimizing the downlink object delivery [6]. These strategies are difficult to apply for personal cloud storage because they often involve huge overhead and require an important property named "stream-informed" [28] not included for personal scenario. Second, a deduplication scheme with aggressive chunking will incur high computational cost for mobile devices, which may degrade the performance under good network conditions (Section 3.1).

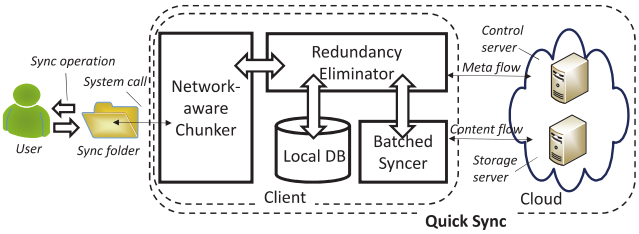


Figure 7: System overview.

Generally, the chunking granularity is closely related to the computation overhead and the effectiveness of deduplication. A more aggressive chunking scheme 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. Deriving from the basic idea of Dynamic Adaptive Streaming over HTTP (DASH), the basic procedure of our approach is to adaptively select an appropriate deduplication scheme 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 strategies as we will introduce below.

5.2.1 Network-aware Chunk Size Selection

The chunking method in our system is based on the content defined chunking (CDC), which introduces *cut-points* to split an input file into chunks based on contents. When the number of operations for insertion, deletion or recording is small, the set of representative hashes for chunks remain mostly the same, this method helps to improve the deduplication efficiency. Moreover, as the average chunk size can be changed to adapt the computational overhead and thus the effectiveness of deduplication, we can select an appropriate chunk size to trade off between computational time and deduplication effectiveness to reduce the total sync time.

To achieve multi-level chunking based on different sizes, both the client and server of QuickSync store *a list of chunking strategies*, along with their *deduplication capacity*, *average chunk size* and *computation cost*. Selecting strategy i means to split the input data into chunks with average size S_i by content defined chunking. At the beginning of a sync process, a client first collects current TCP-level information and follow the method in [21] to estimate the available bandwidth by $Available_BW = \frac{Segment_size * cwnd}{RTT}$, where both the `RTT` and `cwnd` can be observed in the persistent notifi-

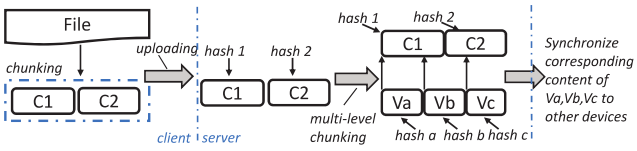


Figure 8: Virtual Chunks in the server.

cation flow. Assume the data size for sync is C bytes, the client will select the chunking strategy i based on the following estimation to minimize the total sync latency:

$$i = \arg \min_{i \in S} \{T_i * C + \frac{(1-\beta_i) * C + Meta_Size_i}{Available_BW}\} \quad (1)$$

where T_i (seconds per byte) and β_i are the corresponding computation time and deduplication ratio of the strategy i respectively. $Meta_Size_i$ is the size of the metadata of the strategy with chunk size S_i. In runtime, a client selects the chunking strategy that provides the minimum sync latency corresponding to its network condition.

5.2.2 Virtual Chunks in the Server

When synchronizing data from the server to devices, the server needs to fetch the chunk content in the storage according to the metadata given by the client. In order to respond to different chunking requirements of the client, the server needs to store the hashes under different strategies. However, it would cost a large amount of storage space if simply executing all chunking strategies in the list and storing all their corresponding metadata and chunks.

To support multi-granularity deduplication, we propose the concept of *Virtual Chunk* that only stores the offset and length which can be used to generate the pointers to the content instead of multiple copies of contents. In an uploading process, after receiving all chunks of a file, the server forms the file according to its metadata. It then conducts all other strategies in the *chunking strategy list* to resplit the file and generate metadata under various strategies. Instead of storing several copies of the same file processed by multiple strategies, the server keeps the Virtual Chunk points to the corresponding chunk in the original file. 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. For all Virtual Chunks generated by a chunking strategy, we add a *vblock list*, which includes all hashes of these Virtual Chunks to the metadata. As a result, the offset of a chunk can be implicitly stored as the sum of all previous chunk sizes in the *vblock list*. Therefore, each Virtual Chunk size only needs to store the length of its corresponding recorded chunk.

Figure 8 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.

5.3 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, in the current cloud storage systems, because all files are stored as independent chunks distributedly, the delta encoding algorithm is executed between pairs of chunks in the modified and the original file. Therefore, with the fixed-size chunking, modification on file may lead to a map between two "un-similar" chunks. Moreover, a chunk in the middle of the uploading process cannot be compared to enable delta encoding. We employ two techniques to alleviate the above problems.

5.3.1 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. Further, 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" [22] of a chunk that would not likely change when there are small data variations. As one typical approach, a rolling hash function can be applied over all overlapping small data regions, and we 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.

5.3.2 Buffering Uncompleted Chunks

To take advantage of the chunks in the air for delta encoding, we introduce two in-memory queues to buffer the uncomplete 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 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 in 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 updating queue if a sketch match is found and popped up when delta encoding for two similar chunks is completed.

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 Eliminator along with their hashes and put in the updating queue. An independent updating process is designed 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. All data in the uploading queue are synchronized to the server by the Batched Syncer.

5.4 Batched Syncer

As discussed in the previous section, 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 chunks. To improve the sync efficiency, we design the Batched Syncer with two key techniques to improve the bandwidth utilization.

5.4.1 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 un-acknowledged 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, and on the other hand 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. Therefore we design a timer for the sync process. If the sync process gets stuck for a long time and timeout, the Syncer will actively terminate the current connection and check the control server for the missing chunks.

During the transmission, the Syncer continuously sends chunks in the uploading queue of the Redundancy Eliminator. If the connection is interrupted by network exceptions or the sync gets stuck 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.

5.4.2 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.

6. SYSTEM IMPLEMENTATION

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

6.1 Implementation over Dropbox

Since both the client and server of Dropbox are totally close 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, 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 hash the chunk to sketch. 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 built on Amazon EC2, the throughput between our proxy and Dropbox is sufficient and not the bottleneck.

To make our Network-aware Chunker efficient and adjustable, we use the SAMPLEBYTE [6] 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 capacity and computation overhead, including the chunk-based deduplication with the average chunk size set to 4MB, 1MB, 512KB and 128KB. Each Virtual Chunk contains a 2-byte field for chunk length.

We use `librsync` [3] to implement delta encoding. We use a tar-like method to bundle all data chunks in 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 60s. We write the QuickSync client and proxy in around 2000 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 in Galaxy Nexus with a 1.2GHz Dual Core CPU, 1GB memory and the proxy is built on an Amazon EC2 server with a 2.8GHz Quad Core CPU and 4GB memory.

6.2 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, an open source cloud storage project. The implementation is similar to that with use of Dropbox but without the need of a proxy. We only need to directly modify both the client and server side source codes to implement our system. Because only the client software on Linux is open source, we implement the modified client in a laptop with a 2.6GHz Intel Quad Core CPU and 4GB memory. and build the server on a machine with a 3.3GHz Intel Octal Core CPU, 16GB memory. We believe our QuickSync can also be implemented in the similar way on other mobile platforms.

7. 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.

7.1 Impact of the Network-aware Chunker

We first evaluate how the Network-aware Chunker improves the throughput under various network conditions. We collect about 200GB data from 10 cloud storage services users, and randomly pick about 50GB as the data set for uploading. The rest about 150GB data are pre-stored in the server for deduplication purpose. We repeat the sync process under various RTT to measure the 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 30ms and 200ms respectively.

Figure 9 shows the results. When the RTT is very low (30ms), 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%. As the RTT increases, the sync speed decreases, but our implementations can still achieve higher total sync speed by taking advantage of 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% increase of the sync speed under various network conditions.

We plot the CPU usages of QuickSync client and server in Figure 10. 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% in two implementations respectively which is still within the acceptable range.

7.2 Impact of the Redundancy Eliminator

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

In Figure 11, 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 synchronize the modified content to server. Note that the TUO results for flip or insert operation on small files ($\leq 100KB$) have reached 1.3. 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 Figure 3. The TUO results in each case are close to 1, showing that our scheme only needs to synchronize the new contents under arbitrary number of modifications and any RTT, because the in-memory uploading queue buffers files in the middle of transmissions to facilitate delta encoding.

7.3 Impact of the Batched Syncer

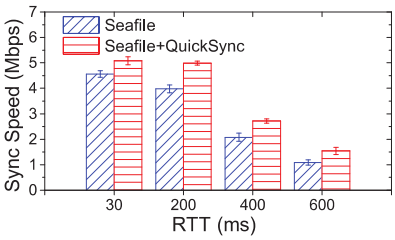
7.3.1 *Improvement of BUE*

To examine the performance of the Batched Syncer in improving the bandwidth utilization, we set the average chunk size to 1MB to exclude the impact of adaptive chunking. In Section 3.3, 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.

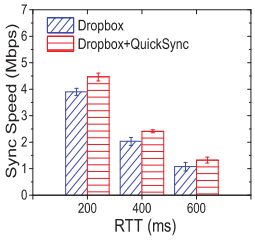
Figure 12 shows the level of BUE improvement under different network conditions. Our mechanism can improve up to 61% the bandwidth utilization efficiency for synchronizing a batch of chunks by reducing the acknowledgement overhead. The improvement is more obvious in high RTT environment where the throughput often experiences big reduction especially when the acknowledgements are frequent.

7.3.2 *Overhead for Exception Recovery*

The per chunk acknowledgement is designed to reduce the recovery overhead when the sync process is unexpected 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

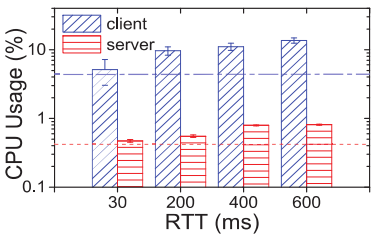


(a) Seafile

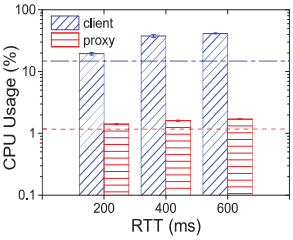


(b) Dropbox

Figure 9: Speed improved by Network-aware Chunker.

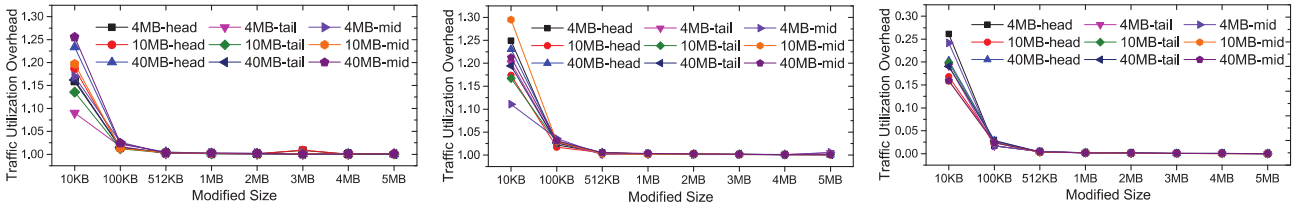


(a) Seafile+QuickSync



(b) Dropbox+QuickSync

Figure 10: CPU overhead of Network-aware Chunker



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

Figure 11: Traffic utilization overhead reduction of Seafile modification.

will create new connection to finish the sync. We record the total sync traffic and calculate the TUO in Figure 13. 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.

7.4 Performance of the Integrated System

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 operations, including creation, modification or deletion, which will trigger corresponding events in the local file system. The performance results are shown in Table 2.

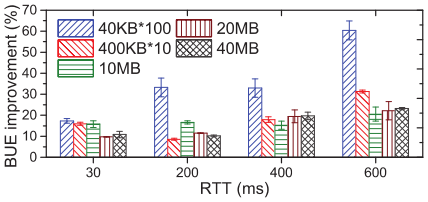
We first generate the workloads on Windows platform based on Seafile and its modification. The QuickSync Paper workload is resulted from uploading the files of this paper, and the Seafile Source generates load by storing all the source codes of the Seafile. Both types of workload contain a lot of small files and do not contain file modification or deletion. Although the traffic size reduction for the two workloads are small (7.5% and 8.9%), our implementation reduces the total sync time by 35.1% and 51.8% respectively. The reduction is mainly caused by bundling the small files to improve the bandwidth utilization, as the Seafile Source contains 1259 independent files. The Document Editing workload on Windows is generated when we naturally edit a PowerPoint file in the sync folder from 3MB to 5MB within 40min. We capture many creation and deletion events because during the editing process, 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, as QuickSync can execute 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 37655 files, with various file types (e.g. PDF or video) and sizes (from 1KB to 179MB). Our QuickSync achieves 37.4% 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 the editing workload generated in the above experiment but contains fewer modifications. Our implementation reduces 41.4% 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% time saving through the batched transmission scheme. The System Backup workload is generated to back up all the app binaries and their data in a phone via a slow 3G network. As our implementation adaptively selects aggressive chunking strategy to eliminate larger amount of the backup traffic and bundles chunks to improve the bandwidth utilization, 52.9% sync time saving is achieved.

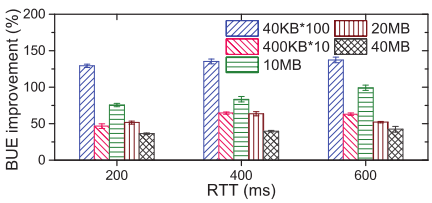
7.5 Server-side Storage Overhead

QuickSync leverages content defined chunking with smaller chunk size and dynamically selects an appropriate deduplication scheme based on real-time network conditions to improve the sync efficiency. Compared with other cloud storage systems using larger fixed-size chunk, our methods may add the metadata overhead on the server side. Now we analyze the impact that our design may have on a Dropbox-like system.

Using the method in Section 4.1, we first measure and analyze the sync flow of uploading a set of files via Dropbox. The metadata of Dropbox contains a number of fields to describe a file, such as modified time, revision number, file path and a list of chunk hashes. The dynamical chunking



(a) Seafile+QuickSync



(b) Dropbox+QuickSync

Figure 12: BUE improvement.

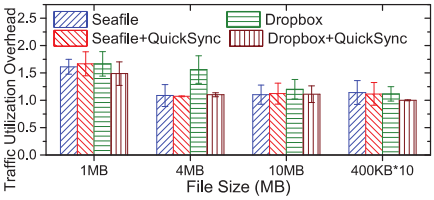


Figure 13: Recovery overhead.

Workload (Platform)	# of Events (C/M/D)	Traffic Size (Origin/Ours/Reduction%)	Sync Time (Origin/Ours/Reduction%)
QuickSync Paper (W)	74/0/0	4.67MB/4.32MB/7.4%	27.6s/17.9s/35.1%
Seafile Source (W)	1259/0/0	15.6MB/14.2MB/8.9%	264.1s/127.3s/51.8%
Document Editing (W)	12/74/7	64.3MB/12.7MB/80.3%	592.0s/317.3s/46.4%
Data Backup (W)	37655/0/0	2GB/1.4GB/30.6%	68.7m/43.1m/37.4%
Document Editing (A)	1/4/0	4.1MB/1.5MB/63.4%	24.4s/14.3s/41.4%
Photo Sharing (A)	11/0/0	21.1MB/20.7MB/1.9%	71.9s/54.6s/24.1%
System Backup (A)	66/0/0	206.2MB/117.9MB/42.8%	612.3s/288.7s/52.9%

Table 2: Practical performance evaluation for QuickSync using a series of real world representative workloads. W: Windows platform. A: Android platform. Event C: Creation. Event M: Modification. Event D: Deletion.

Data set	Hash (byte)	% in Metadata	% in Overall Sync Traffic
1 MB	43[1]	2%	0.004%
10×100 KB	430	13%	0.037%
10 MB	129	7%	0.001%
10×1 MB	430	11%	0.004%

Table 3: The volume of chunk hash and its occupancy in metadata and overall sync traffic of Dropbox.

scheme in QuickSync mainly increases the volume of hashes in the metadata and will not affect other fields. Table 3 shows the size of hashes and its occupancy in the metadata and the total sync traffic of Dropbox. We find that hashes occupy a very small fraction in the metadata and the overall sync traffic.

Further, we analyze and estimate the additional storage overhead incurred by QuickSync. If we only consider the hash values in metadata and use the SHA-1 hashing for each chunk, the storage overhead of a 4MB file in Dropbox is the sum of contents and metadata, i.e. (4MB+20B). The additional storage overhead of QuickSync is caused by two factors: 1) the additional hashes incurred by multi-level chunking (Section 5.2.1) and 2) the storage of Virtual Chunks (Section 5.2.2). Because our implementation of QuickSync uses 4MB, 1MB, 512KB and 128KB chunking, the hash size of a 4MB in QuickSync is 20B*(1+4+8+32)=900B. Moreover, a 4MB file has at most (4+8+32)=44 Virtual Chunks and the rest one contains the real contents. Since a Virtual Chunk occupies a 2-byte field for the chunk length, the maximal cost of Virtual Chunks is (2B*44)=88B. Therefore the total storage cost of a 4MB file in QuickSync is (4MB+988B). In summary, QuickSync adds 968B additional storage cost, which is only 0.02% of the overall storage cost. Consider that in practice the hash values are only a very small fraction of the overall storage cost, QuickSync may not incur much storage overhead on the server side.

[1]The chunk hash is salted in practice.

8. RELATED WORK

Measurement study. Recently a large number of measurement research efforts have been conducted on cloud storage services [16, 12, 11, 17, 19]. CloudCmp [16] measures the elastic computing, persistent storage, and networking services for four major cloud providers. Focusing on personal cloud, Drago *et al.* give a large scale measurement for Dropbox [12], and then compare the system capabilities for five popular cloud storage services in [11]. However, all these previous studies only focus on the desktop services, and all of them are based on black-box measurement. Li *et al.* give the experimental study of the sync traffic, demonstrating that considerable portion of the data sync traffic is wasteful [17]. Our work steps closer to reveal the root cause of inefficiency problem from the protocol view, and we are the first to study the sync efficiency problem in wireless networks.

System design. There are also many studies about the system design for cloud storage services [26, 25, 9]. However they are mostly focusing on enterprise backup instead of personal cloud. Li *et al.* propose an adaptive sync defer (ASD) mechanism, which adaptively tunes its sync deferment to follow the latest data update [18]. The bundling idea of our Batched Syncer is similar to ASD, but ASD will incur 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 3.2. ViewBox [27] is designed to detect corrupt data through the data check sum 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 (CDC) [20, 28, 22, 14, 5, 6, 23, 7] and delta encoding [24]. 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 s-

electing 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 in Redundancy Eliminator address the limitation and wisely use delta encoding to reduce the sync traffic overhead.

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 in-depth analysis to give the root causes of the identified problems concurrently exploiting trace study and data decryption. To address the inefficiency issues, we propose QuickSync, a system with three novel techniques. We further implement QuickSync to support sync with Dropbox and Seafile. Our extensive evaluations demonstrate that QuickSync can effectively save the sync time and reduce significant traffic overhead for representative sync workloads.

10. ACKNOWLEDGMENTS

We sincerely thank Matt Welsh for shepherding our paper, as well as the anonymous reviewers for their valuable comments and feedback. This research was supported by National Natural Science Foundation of China (no. 61120106008 and no. 61422206), National High Technology Development 863 Program of China (no.2013AA010401). The research of Wang is supported by NSF 1408247 and NSF 1247924.

11. REFERENCES

[1] Dynamorio. http://dynamorio.org.

[2] Googledrive. http://www.google.com/drive/index.html.

[3] librsync. http://librsync.sourceforge.net/.

[4] Onedrive. https://onedrive.live.com.

[5] Seafile source code. https://github.com/haiwen/seafile.

[6] B. Agarwal, A. Akella, A. Anand, A. Balachandran, P. Chitnis, C. Muthukrishnan, R. Ramjee, and G. Varghese. Endre: An end-system redundancy elimination service for enterprises. In *NSDI*, pages 419–432. USENIX, 2010.

[7] A. Anand, A. Gupta, A. Akella, S. Seshan, and S. Shenker. Packet caches on routers: the implications of universal redundant traffic elimination. In *SIGCOMM*. ACM, 2008.

[8] A. Balasubramanian, R. Mahajan, and A. Venkataramani. Augmenting mobile 3g using wifi. In *MobiSys*, pages 209–222. ACM, 2010.

[9] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, et al. Windows azure storage: a highly available cloud storage service with strong consistency. In *SOSP*, pages 143–157. ACM, 2011.

[10] Y. Cui, Z. Lai, and N. Dai. A first look at mobile cloud storage services: Architecture, experimentation and challenge. http://www.4over6.edu.cn/others/technical_report.pdf.

[11] I. Drago, E. Bocchi, M. Mellia, H. Slatman, and A. Pras. Benchmarking personal cloud storage. In *IMC*, pages 205–212. ACM, 2013.

[12] I. Drago, M. Mellia, M. M Munafo, A. Sperotto, R. Sadre, and A. Pras. Inside dropbox: understanding personal cloud storage services. In *IMC*, pages 481–494. ACM, 2012.

[13] Dropbox. https://www.dropbox.com.

[14] Y. Hua, X. Liu, and D. Feng. Neptune: Efficient remote communication services for cloud backups. In *INFOCOM*. IEEE, 2014.

[15] D. Kholia and P. Wegrzyn. Looking inside the (drop) box. In *7th USENIX Workshop on Offensive Technologies (WOOT), Washington, DC, USA*, pages 1–7, 2013.

[16] A. Li, X. Yang, S. Kandula, and M. Zhang. Cloudcmp: comparing public cloud providers. In *IMC*, pages 1–14. ACM, 2010.

[17] 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 *IMC*. ACM, 2014.

[18] 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 *Middleware 2013*, pages 307–327. Springer, 2013.

[19] T. Mager, E. Biersack, and P. Michiardi. A measurement study of the wuala on-line storage service. In *Peer-to-Peer Computing (P2P), 2012 IEEE 12th International Conference on*, pages 237–248. IEEE, 2012.

[20] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In *SIGOPS*. ACM, 2001.

[21] S.-H. Shen and A. Akella. An information-aware qoe-centric mobile video cache. In *MobiCom*, pages 401–412. ACM, 2013.

[22] P. Shilane, M. Huang, G. Wallace, and W. Hsu. Wan-optimized replication of backup datasets using stream-informed delta compression. *TOS*, 8(4):13, 2012.

[23] N. T. Spring and D. Wetherall. A protocol-independent technique for eliminating redundant network traffic. *SIGCOMM*, 2000.

[24] A. Tridgell, P. Mackerras, et al. The rsync algorithm, 1996.

[25] M. Vrable, S. Savage, and G. M. Voelker. Cumulus: Filesystem backup to the cloud. *TOS*, 5(4):14, 2009.

[26] M. Vrable, S. Savage, and G. M. Voelker. Bluesky: a cloud-backed file system for the enterprise. In *FAST*, page 19. USENIX, 2012.

[27] Y. Zhang, C. Dragga, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Viewbox: integrating local file systems with cloud storage services. In *FAST*, pages 119–132. USENIX, 2014.

[28] B. Zhu, K. Li, and R. H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *FAST*, volume 8, pages 1–14. USENIX, 2008.