Improving Cloud Storage Usage Experience for Mobile Applications

Yongshu Bai Xin Zhang Yifan Zhang

Binghamton University, Binghamton, New York

{ybai4, xzhang99, zhangy}@binghamton.edu

Abstract

Cloud storage services are becoming increasingly popular in mobile apps. Through comprehensive real app studies, we reveal that many mobile apps making use of cloud storage services provide poor usage experience, such as unnecessary energy consumption, extended synchronization response time, and redundant network traffic. The root cause of these problems stems from the way that current commercial cloud storage providers choose to deploy their services to mobile platforms: in order to have fast and easy deployment, they totally avoid client-side system-level file operation monitoring and servicing, which has been an essential part of the successfulness of traditional distributed file systems, and leave the implementation of the important client-side operations like caching, consistency assurance totally to mobile app developers. We propose `StoArranger`, a user-space system-wide service aiming to improve cloud storage usage experience for exiting mobile applications. We briefly present the design, the associated challenges, and our ongoing implementation of `StoArranger`.

1. Introduction

Personal cloud storage services, such as Dropbox [17], Google Drive [18] and OneDrive [26], have been gaining fast-increasing popularity in recent years [15, 16, 25]. Meanwhile, as smartphones and tablets are becoming more powerful and prevalent, we are seeing the same trend of cloud storage services gaining popularity on mobile devices.

The general problem of efficient, secure and scalable access to cloud-sourced data over networks of limited bandwidth and reliability has been the subject of distributed file systems research over the past three decades. During the couse, many effective solutions and useful insights have been developed, such as improving system scalability [19, 30], gracefully dealing with network failures and user mobility [23, 31], improving network communication efficiency [27], and ensuring data consistency [19, 23, 30, 31]. However, today's commercial services for mobile access to cloud storage have ignored some useful insights and practical experience of this multi-decade research. Most notably, they choose to avoid client side OS-level monitoring and support, in exchange for fast and easy service deployment. As the result of this implementation and deployment strategy, many of the existing mobile apps fall short of using cloud storage service efficiently, and thus leading to poor usage experience, such as unnecessary energy consumption, extended folder synchronization time, and redundant network transmission traffic. Here we summarize our findings as follows.

• *First*, many mobile apps generating user data (e.g., photos, videos, notes and documents) are using cloud storage services for backing up the data. We find that these apps often perform cloud backups in an uncoordinated and energy-agnostic way, causing unnecessary energy consumption. This problem can be exacerbated when there are multiple mobile apps performing cloud backups in the devices, which is not uncommon in practice. We refer this problem as *uncoordinated cloud backup* by mobile apps (Section 3.1).

• *Second*, there are a growing number of mobile apps that enable synchronization of folders across different mobile devices by using cloud storage services. We find that most of these apps fail to properly use cloud storage APIs to provide good usage experience for users. We refer this problem as *inefficient cloud folder sync* (Section 3.2).

• *Third*, many mobile apps dealing with user files, such as file managers/browsers/editors, have been integrating in a new feature that allows for access to files stored on the cloud. We find that, most of these cloud file managers/editors either do not implement caching (in which case the whole file is downloaded every time when it is accessed) or provide weak consistency with caching implemented. Most mobile apps also do not implement metadata caching for cloud folders. We refer this problem as *inefficient cloud file accesses* by mobile apps (Section 3.3).

• *Fourth*, we find that all the mobile apps we tested down-

APSys 2016, August 4–5, 2016, Hong Kong, China.
Copyright © 2016 held by owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-4265-0/16/08…$15.00
DOI: http://dx.doi.org/10.1145/2967360.2967378

load or upload the whole file from cloud storage even there is only a single byte of change. We refer this problem as *inefficient whole file transmission* problem (Section 3.4).

The *direct* causes of the previous problems, as suggested by our experimental results, are lack of cloud upload coordination by the device OS and errors or bad choices made by the app developers due to carelessness or inexperience. The deeper *root* cause is the choice made by commercial cloud storage providers, for the purpose of easy and fast service deployment, to avoid client-side system-level file operation monitoring and servicing, which have been proved to be critical to the correct functioning of traditional distributed file systems [19, 23, 27, 30, 31]. We will discuss this aspect in more details in Section 2, and present our app studies showing the outcomes of this choice in Section 3.

Motivated by our findings, we propose `StoArranger`, a system that provides a system-wide support to intercept and arrange (i.e., apply the correct logic to) the interaction between mobile apps and cloud storage services. With our design, the `StoArranger` device component runs as a user level service, and requires no change to the mobile apps and the underlying mobile OS. Therefore, `StoArranger` can effectively improve cloud storage usage experience for *existing* mobile applications. We will briefly present the design, the associated challenges, and our on-going implementation of `StoArranger` in Section 4.

2. Background: mobile data accessing and cloud storage services on mobile devices

Client-side file operation awareness Common to many of these exiting solutions, *awareness of client-side file operations*, which means the solutions need to be aware of the occurrences of file operations, is essential for providing the correct functionalities, For example, traditional distributed file systems, such as AFS [19, 30], Coda [23, 31], LBFS [27], need to know when a client opens a file (such that the intended file can be fetched from the remote server or from the local cache), and when the client closes the file (such that the file can be written back). This knowledge is also essential to implementing file caching and metadata caching, which are important to system performance improvement.

Many well-known distributed file system solutions enable their awareness of client-side file operations through interposing on file system system calls or device drivers, and achieve ideal system performances. For example, AFS [19, 30] intercepts file open and close system calls on the client side and forward them to a client-side cache-management process for processing. Coda [23, 31] inherited AFS's client-side design and added more functionalities to support disconnected operations and server replication. LBFS clients [27] resort to interposing on XFS [11] device driver to obtain notifications about file opens, closes and modifications, and to achieve the content-based breakpoint chunking.

Cloud storage services on mobile devices Modern cloud storage services are distributed file systems in nature. Commercial cloud storage providers, however, adopt a way different from exiting distributed file system solutions to deploy their services onto mobile devices. Unlike traditional distributed file systems, which have their own client implementations including the file system call or device driver interposition to enable client-side file operation awareness, cloud storage services provide no client implementation. Instead, they usually provide development APIs [5, 6, 9] for mobile apps to make use their services. These development APIs are usually RESTful APIs[34] over HTTP. They allow apps to request or update resources (i.e., user files in the case of cloud storage) on the servers by making HTTP requests. For example, suppose a mobile app is managing a folder on both the local device and the cloud. To synchronize the folder (i.e. to make the folder content consistent on both sides), the mobile app can call the API responsible for retrieving folder metadata on the intended folder. In response, the server returns the folder entries and folder metadata. The app then compares the metadata returned from the server and that on the local device to decide if any entry in the folder has been changed locally or remotely. If so, the app calls the corresponding API to download/upload the changed file(s) from/to the sever.

The adoption of RESTful APIs allows cloud storage providers to deploy their services to third-party mobile apps easily and quickly. The reason is that now the cloud storage services do not need to provide the client implementation, which is now left to app developers. However, this deployment choice causes three notable problems:

• Since these RESTful APIs do not contain any OS-level mechanisms, such as system call/device driver interposition, to enable client-side file operation awareness for the client implementation, we are now totally relying on app developers to correctly implement the client logic, and to implement those practices that are not necessary but extremely beneficial, such as caching. As we will show later, poor cloud storage usage experience in many mobile apps is due to carelessness and/or inexperience of app developers.
• Also, since there is no OS-level support about file operations on the client side, it is impossible to implement those advanced techniques that require tight integration with the OS, such as the content-based file chunking and incremental data transmission [27].
• Lastly, because of the lack of client-side OS-level support from the cloud storage providers, it is hard to implement system-wide cloud storage requests coordination, which we will later show is helpful to improve cloud storage usage experience on mobile devices.

3. Inefficient cloud storage usage by mobile apps

In this section, we present our mobile app studies aiming to understand the inefficient cloud storage usage in current mobile apps. Though the study was performed on Android

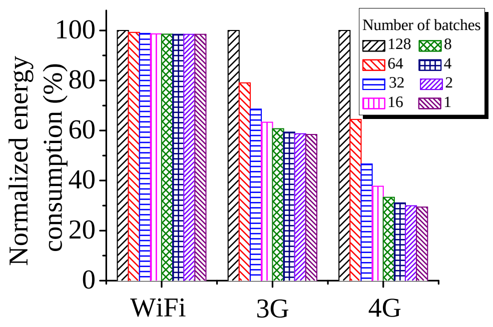

Figure 1. Energy consumption comparison for different cloud backup request batching schedules.

apps from Google play, we believe that the findings are likely to be applied to other major mobile platforms.

3.1 Uncoordinated cloud backups by mobile apps

Many mobile apps that deal with user content (e.g., photos, videos, notes and documents) now offer user an option to automatically back up files to a cloud storage chosen by the user when file is generated or changed. The timing of these backups are determined by the policies set by individual apps. We surveyed 25 popular Android apps that perform automatic photo/video cloud backup. Among these 25 apps, 18 back up new photo/video to the cloud immediately after photo is taken or video is shot. These apps include those very popular cloud storage service apps, such as Dropbox and OneDrive, and social network apps, such as Flickr. Five of the 25 apps wait a fixed amount of time (usually between 5 to 30 seconds), and then upload the newly generated content to the cloud. Two apps perform photo/video backup at the time when user quits from the camera app.

Form the above we can see that most automatic photo/video backup apps upload the new content right after it is generated. This policy can lead to a substantial amount of energy waste due to accumulated promotion and tail energy [12, 21] consumption. More specifically, to transmit data when the wireless interface (e.g., WiFi, 3G or 4G) is idle, certain amount of energy will be spent in bringing the interface from idle state to active state (i.e., promotion energy). After the data is is transmitted, before the interface goes back to lower power idle state, it stays in high power state for a fixed amount of time, during which the energy consumption is called tail energy. Both Promotion and tail energy consumption do not contribute to the actual data transmission. If an app uploads new photos at the times immediately after each of them is take, it's likely that every photo upload suffers from promotion/tail energy waste.

An approach to solve the above energy waste problem is to schedule the cloud upload requests from mobile apps, such that multiple requests can be served in a single burst. This way, the multiple sessions of promotion/tail energy waste can be reduced to one. We conducted a simulation experiment to study what extent this idea can save energy consumption for cloud backups, when using WiFi, 3G and 4G as the wireless transmission method. In this experiment, we assume a user takes 128 pictures along the day, each picture is 1 MB in size. These photos are backed up to a cloud storage in batches. The intervals between two uploads are longer than wireless interface's tail time (meaning each upload has its own promotion/tail energy consumption). We compared the cases of batch count being 128, 64, 32, 16, 8, 4, 2 and 1 (batch count of 128 meaning all the 128 photos are uploaded individually, and batch count of 1 meaning all the 128 photos are uploaded in one burst). The transmission throughput of WiFi, 3G and 4G were set to 12 Mbps, 2 Mbps and 6 Mpbs respectively (which are the U.S. average values to recent studies [1, 7]). We adopted the power models presented in work [21] to calculate the energy consumption of the transmissions, and compared the energy consumed per byte of traffic for the cases of different batch count. Figure 1 shows the result of this experiment. We can see that the batching approach saves a significant amount of energy for the cellular cases (i.e., 4G and 3G). When using 4G/3G as the wireless transmission method, uploading all the photo in one batch saves about 70%/42% energy of the case of uploading them individually. This number of the WiFi case is 2%. This is because 4G and 3G both have longer tail time and the higher tail power than WiFi.

From the above experiment we can see that coordinating cloud upload requests from mobile apps is promising in saving network transmission energy consumption, especially for the cellular cases. As cellular data becomes more affordable and unlimited data plan gains popularity, it is highly worthwhile to investigate saving cloud backup energy for cellular cases.

3.2 Inefficient cloud folder sync

Another useful functionality made possible by cloud storage services is folder synchronization, where user can synchronize a local folder (called sync folder) on the device with the cloud. Any changes made to the sync folder will be automatically synchronized to the cloud. With this functionality, it is easy for a user to keep her data synchronized across the multiple devices she owns.

Currently there is no major personal cloud storage service provide official app to support folder sync on Android. However, due to the usefulness of folder sync, many third-party apps have been developed to provide Android users with this functionality. For example, Autosync-Dropbox [2] is an app providing folder sync for Dropbox users on Android, and is now reaching 5 million installs on Google Play. We believe these third-party folder sync apps using cloud storage services will remain popular even after cloud storage providers provide their official folder sync apps. This is because third-party developers can provides all sorts of customization wanted by users. For example, many folder-sync app allow user to link to different cloud storage services in just one app.

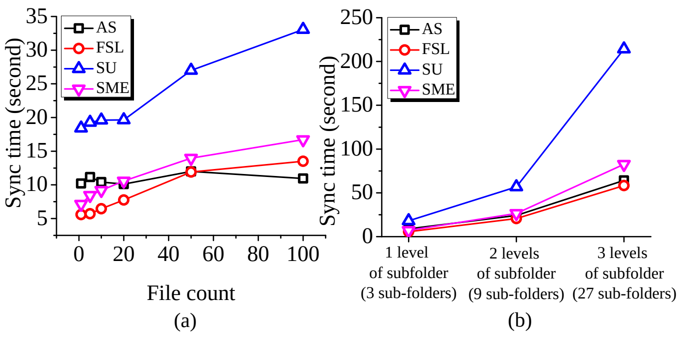

Figure 2. Folder sync turnaround time: OneDrive. (a): fixed sub-folder number; varied file count in each sub-folder. (b): fixed file count in each sub-folder; varied sub-folder number.

The real app study We have performed an real app study to understand how folder sync apps perform. In the following, we explain several key points in this study:

- We tested 20 apps offering cloud folder sync. Here we choose four popular ones to study in details: Autosync (`AS`), FolderSync Lite (`FSL`), Synchronize Ultimate (`SU`) and SME Cloud File Manager (`SME`), all of which support folder sync using different cloud storage services (e.g., Dropbox, Google Drive and OneDrive).
- Since cloud storage services all use HTTPS for transmission, we connect the test phone to Internet via a Charles HTTP proxy [3], through which we could capture and analyze HTTPS traffic.
- We tested two cloud storage services: OneDrive and Goolge Drive. We didn't test Dropbox at network flow level, because Dropbox has integrated SSL pinning [28] into its third-party APIs, and thus does not recognize the proxy certificate even it is already installed on the phone.
- For folder sync performance metrics, we measured folder sync *turnaround time*, which is the time duration of whole sync, and folder sync *traffic*, which is total amount of traffic generated during the sync.
- We wanted to examined how the number of files and the number of sub-folders in the sync folders affect the sync performance. To this end, we separated the test of each app/cloud storage combination in to two parts. In the first part, the sync folder contains 3 sub-folders, and the number of files in each sub-folder is chosen from 1, 5, 10, 20, 50, 100. In the second part, each leaf folder of the sync folder contains just one file, and each non-leaf folder contains three sub-folders. The number of sub-folder levels of the sync folder is chosen from 1, 2 and 3, in which cases the total sub-folders in the sync folder is 3, 9, and 27.
- Before each sync operation, the local folder is already consistent with the cloud folder. The sync operation just confirms local and cloud have already been consistent. No file download/upload actually happened.
- Each app/cloud-storage-service/sync-folder-configuration combination was repeated five times, and the average results are reported here.

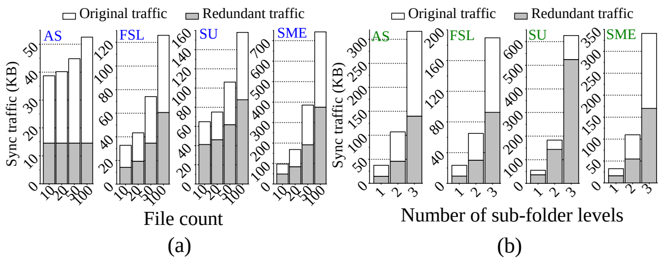

Figure 3. Folder sync traffic: OneDrive. (a): fixed sub-folder number; varied file count in each sub-folder. (b): fixed file count in each sub-folder; varied sub-folder number.

We can have the following observation from Figure 2 and Figure 3, which show the experiment result for OneDrive.

- (Figure 2 (a)) When the number of sub-folders is fixed, the sync turnaround time of `AS` does not depend on the file count in the sub-folders. It remains around 10 seconds as the file count increases. For `FSL` and `SME`, the sync turnaround time becomes longer (ranging from 5 seconds to 15 seconds) as file count increases. `SU` has the same behavior as `FSL` and `SME`, but has longer turnaround time (ranging from 18 seconds to 33 seconds).
- (Figure 2 (b)) When increasing sub-folder number in the sync folder, the sync turnaround time increases in a roughly *proportional* way. When there are 3 levels of sub-folders in the sync folder, the turnaround times of `FSL`/`AS`/`SME`/`SU` are around 65/59/82/215 seconds, which are unacceptably long given that there was no actual file download/upload.
- (Figure 3 (a)) When the number of sub-folders is fixed, the sync traffic of all four apps increases as the file count increases. But there is no fixed relationship between the two. For example, the sync traffic of `FSL` and `SME` is more obvious to be proportional to file count in each sub-folder, but that of `AS` and `SU` is less so. Also we can see that the amount of sync traffic is high given that no actual file download/upload is incurred. The highest cases we saw in this experiment were `FSL` and `SU` working with Google Drive, in which cases the sync traffic were as high as 1.8 MB.
- (Figure 3 (b)) The sync traffic of all apps increases proportionally to the number of sub-folders in the sync folder.
- (Figure 3 (a) & (b)) We find that an app can request the same resource (e.g., user's info, folder metadata) multiple times in the same sync operation, causing redundant network traffic. All the apps have this problem for the OneDrive case.

In summary, there are two takeaways from the results:

- First, current mobile apps offering cloud folder sync incur high sync turnaround time and sync traffic, both of which seem to be proportional to the number of sub-folders in the sync folder.
- Second, there is a substantial amount of redundant traffic generated by the sync.

The Google Drive experiment of `AS`/`FSL`/`SU`/`SME` also indicate the similar results. We've also tested another 10 mobile apps that offer folder sync with OneDrive and/or Google

Drive. All of them have the problem of proportional time and traffic with folder count, and most of them have the redundant traffic problem. In addition, we tested 6 apps doing folder sync with Dropbox. Although we could not see the content of the sync traffic, we could still judge if sync traffic is proportional to number of sub-folders. The result was that 5 of the 6 had the problem.

Understanding the study results To understand the above results, we analyzed the sync traffic captured by the Charles proxy. We found that the way that most of these apps performing folder sync is to request metadata of *every* subfolder of the sync folder. For example, if a folder A contains three files/directories, the metadata of the folder A has three entries, with each entry describing the metadata of each file/directory, such as name, size, last-modified-time, and many other fields related to cloud file management. These apps compare the metadata obtained from the cloud with the metadata of the local file system to determine (for example, via the last-modified-time field) if a file or a directory in the sync folder needs to be downloaded from or uploaded to the cloud. This explains why folder sync turnaround time and traffic are proportional to sub-folder number in the folder.

To explain why some apps' sync traffic are proportional to file count in each sub-folder when the sub-folder number in the sync folder is fixed, but others are not, it depends on what content is redundantly requested and the response's content encoding. If metadata of sub-folders are redundantly requested and the responses from server are not compressed, the sync traffic would be proportional to the file count in the sub-folders.

As for the cause of redundant traffic, we conjecture that it is because developers' error as not all apps have the problem.

How to improve The direct cause of the above problems is mobile apps track metadata for the whole folder hierarchy. In fact, most cloud storage services already provide an efficient way to implement folder sync, which is a set of APIs to get changes on cloud folder incrementally. For example, Dropbox provides a `/delta` API [4] to allow developers to get the list of changes that have been made on the cloud folder since the last time the `/delta` API is called. If none has been changed between the two calls, the latest call will be responded with a empty list. Using the `/delta` API is a more preferable than tracking metadata of the entire sync folder hierarchy at the time of syncing. However, using `/delta` API requires app developer to maintain a state of the local sync folder, and keep track of local changes between syncs[1]. By contrast, using metadata tracking at the time of syncing requires app developer to maintain or track nothing. What the developer needs to do is just to get metadata for all the sub-folders so that she knows what has been changed on both side since last sync, which is much easier and less error-prone than using the `/delta` API. This can explain why almost all the current mobile apps offering cloud-based folder sync adopt the whole folder metadata tracking method. Note that both Google Drive and OneDrive provide similar mechanisms allowing developers to implement more efficient folder sync.

Given the above observation, an possible way to improve the inefficiency folder sync problem is to convert the whole folder metadata tracking way used by the apps to the `/delta` way. We will explore this option next in Section 4.

3.3 Inefficient cloud file accesses

In addition to cloud storage backup and cloud storage based folder sync, another popular cloud storage related functionality is accessing to cloud file remotely via mobile apps. Many file manager/explorer/editor apps have in integrated this functionality. We have conducted another real app study to understand the cloud file access efficiency of these apps. In this study, we chose 20 file manger/explorer apps and 5 file editor apps, most of which have accumulated at least 1 million installs from Google Play. We found the following inefficiency regarding cloud file accesses:

• Many of the apps do not implement caching for downloaded cloud files. Therefore, they re-download the whole file every time user opens the cloud file via the app. 10 of the 20 file manger apps and 2 of the 5 file editor app have this problem. A more efficient implementation is to cache a cloud file at the first time that user opens it. Check with the cloud to see if the file has been updated on the cloud side since last the open before user reopen it. The local cached copy is used to served the reopen requests if there is no change.

• Some of the apps implementing cloud file caching, but do not provide the correct consistency guarantee. For example, 6 of the 20 file manager apps get the metadata of files only at the time when user open their parent folder, but not when the files are opened by user. This could lead to user open dated files with these apps.

• Most of the apps do not implement metadata caching, meaning every time user navigates to a folder, these apps request the full metadata of the folder even there has been no change of the folder since it was last accessed. This will cause unnecessary metadata traffic, especially for those folders that contain many items or those that are frequently accessed. 21 of all the 25 apps have this problem.

Given that many existing apps have the above problem, an efficient way to improve is to provide a system-wide service to perform the cloud file caching and cloud metadata caching for the apps. We will explore this option next in Section 4.

3.4 Inefficient whole file transmission

Related to all the previous inefficiency problems, we also find that when mobile apps download/upload a file from/to the cloud storage, whole file download/upload is always using even there is a very small change in the file. This is consistent with the observations in recent studies [13, 16, 25]. A more efficient implementation would be to enable a

[1] The mobile apps we tested provide different options of sync timing: sync at the time when there is any change on the local folder; sync after some delay after any change on the local folder; fixed interval chosen by user.

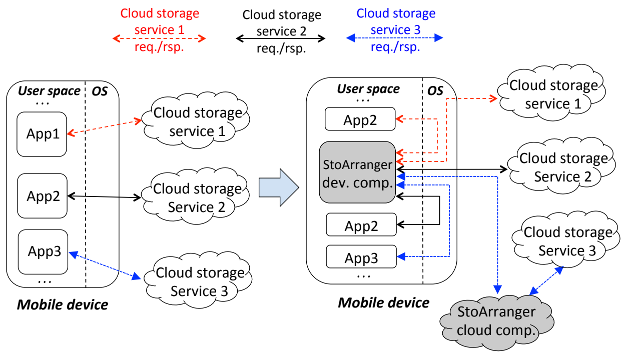

Figure 4. `StoArranger` work context example.

system-wide incremental-file-sync service in the device. We will briefly explore this option next in Section 4.

To sum up, the direct causes of the above cloud storage usage inefficiency problems (Section 3.1-3.4) are:

• There is lack of system level support for cloud storage request scheduling or incremental file sync (Section 3.1, 3.4).

• Some cloud storage APIs, although useful for improving cloud storage usage efficiency, are not adopted by app developers due to their difficulty to use (Section 3.2).

• Developer's inexperience or/and carelessness (Section 3.2, 3.3). Another good example to support his aspect is that in our folder sync app study (Section 3.2), we saw two apps that do not request gzip encoding for server response, which can cause significant transmission bandwidth waste. One app (i.e., `SME`) does this for both OneDrive and Google Drive sync, which case we can attribute to developer's inexperience. Another app (i.e., `FSL`) requests gzip encoding for OneDrive, but not for Google Drive. This is because to receive gzip-encoded response, Google Drive requires developers to add the "gzip" string not only in the "Accept-Encoding" filed, but also in the "User-Agent" field, whereas most other storage services only require the "gzip" string in the "Accept-Encoding" field. Therefore, we can attribute this efficiency of `FSL` to developer's carelessness.

4. `StoArranger`: improving cloud storage usage experience for exiting mobile apps

Motivated by these observations, we design and implement `StoArranger`, a system aiming to improve cloud storage usage experience for mobile applications. As we analyzed previously, although the cause of poor cloud storage usage experience in many mobile apps is poor coding by app developers at first glance, the deeper root cause is the way that the cloud storage services are deployed onto mobile devices lacks OS-level support to enable client-side file operation awareness, to enable client-side cloud storage requests coordination, and to integrate advanced techniques requiring tight OS integration. Since we are seeking to improve mobile app cloud storage usage experience in a scalable and practical way, one of our top design goals is to support exiting mobile apps without modifying the OS and the apps. Therefore, instead of interposing on system call/device driver boundaries in the OS, we choose to interpose on the network transmissions made by the apps. The full version of `StoArranger` consists of a device component and a cloud component. Generally speaking, the `StoArranger` device component is a proxy that intercepts and arranges (i.e., applies the correct logic to) the interaction between mobile apps and cloud storage services. With our design and implementation, the `StoArranger` device component is running as a user level service, and requires no change to the mobile apps and the underlying mobile OS. The `StoArranger` cloud component works with the device component to solve mainly the *inefficient whole file transmission* problem introduced previously. Figure 4 shows an example of the work context of `StoArranger`: suppose in a mobile device there are three apps working with three different cloud storage services. Without `StoArranger`, the three apps communicate directly with their cloud storage services independently, in which case poor cloud storage experience can happen. With `StoArranger`, all the communications between app and cloud storage services are redirected to the `StoArranger` device component, which arranges and transform these communications to achieve better cloud storage usage experience for mobile apps. Depending on different goals, `StoArranger` device component can communicate either directly with the intended storage services, or via the `StoArranger` cloud component.

Addressing inefficient cloud storage usage problems We summarize how the `StoArranger` device component works to address the issues identified in Section 3, and discuss the associated challenges as follows.

• For the *uncoordinated cloud backup* problem, our observation is that, with common mobile workloads, it is often not necessary to back up the content to the cloud at the time when they are generated or changed. Therefore, `StoArranger` tries to delay and batch cloud backup requests from apps to minimize the impact of transmission tail energy.

There are two main challenges in solving this problem. *First*, deciding the batching size and the right timing of uploading the batches is not trivial, and is critical to the performance of `StoArranger`. *Second*, `StoArranger` must not perform batching on cloud uploading requests if they were explicitly requested by the user. However, without changing mobile OS or apps (which is one of our key design goals), it is difficult to determine whether a uploading request is generated due to user's explicit intention or automatic schedule made by the apps.

• To solve the *inefficient cloud folder sync* problem, `StoArranger` first detects problematic folder sync activity before they are actually carried out, and takes advantage of the delta folder sync feature available in most of the cloud storage APIs to minimize sync traffic and turnaround time.

There are also two associated challenges. *First*, detecting the onset of problematic folder sync is difficult. This is because problematic folder sync uses the same cloud storage requests (e.g, folder metadata requests) as other cloud storage operations triggered by users or by automatic job schedule of other apps. Without changing mobile OS or apps, it is difficult to tell if a request comes from problematic folder sync or normal cloud storage operations. *Second*, using the delta folder sync APIs requires `StoArranger` to remember and manage the local state of sync folders, which is not only error-prone, but can also incur non-negligible system overhead for `StoArranger`, which intends to manages all the cloud storage services being used by the apps.

• To solve the *inefficient cloud file accesses* problem, `StoArranger` performs cloud file caching for apps making access to cloud storage. With this functionality, `StoArranger` can serve the download requests from apps with the locally cached copies, and issue file download requests to the cloud only when necessary.

Designing and implementing the cloud file caching are not trivial because `StoArranger` is intended to serve all the apps and all the cloud storage services being used in the device, an inefficient design or implementation can lead to high degree of memory overhead and storage duplication.

Ongoing implementation We are currently undergoing the implementation and evaluation process of our `StoArranger` system. To satisfy our goal of addressing the inefficiency problems without changing the apps or the OS, we have implemented an basic user-space daemon framework where the `StoArranger` device component is running. We leveraged the WiFi and Cellular proxy settings, which are readily available in both Android and iOS, to redirect all the HTTP traffic to the `localhost` interface. The user-space daemon monitors the `localhost` interface using the `libevent` library [8]. We have also successfully integrated the `openssl` [10] library in to the `StoArranger` device component to make it work with HTTS library.

We also aim to enable our solution to be scalable to different cloud storage services. Since different cloud storage services have different communication flows, hard-coding these flow logic is not a good option. Otherwise the need of supporting a new cloud storage service, or a single change in communication flow made by the supported service provider, can disrupt `StoArranger`, and requires new development and installation of `StoArranger`. We have completed the design and implementation of using xml file to describe the communication flows of cloud storage services. The `StoArranger` device component interprets the xml file to understand the communication flow of a cloud storage service, and acts accordingly to bridge the mobile app and the cloud server. This way, any change of communication flow in existing cloud storage services or adding any new cloud storage services would be a matter of updating the xml file.

5. Related Work

Several recent studies put their focuses on **understanding the characteristics of personal cloud storage services** [15, 16, 20, 25]. Hu et al. conducted an early study on comparing cloud storage services' backup and restoration performances [20]. Drago et al. [15] presented a comprehensive measurement and characterization study on Dropbox. The same group later conducted another study to compare the service capabilities (e.g, chunking, client-side deduplication, data compression) of five popular cloud storage services: Dropbox, OneDrive, Google Drive, Wuala, and Cloud Drive [16]. Li et al. performed a measurement study to specifically understand data synchronization efficiency of six cloud storage services: Dropbox, Google Drive, OneDrive, Box, Ubuntu One, and SugarSync [25].

Different from the above studies, which aim to understand the behaviors of cloud storage services, our focus is to understand how mobile apps make use of cloud storage services, and further to improve the cloud storage usage experience for mobile apps.

There exist works on **minimizing mobile traffic and energy consumption through transmission scheduling** [12, 14, 22, 29, 32]. Different from the above studies, which mainly target network traffic/energy consumption optimization, our focus is to study and improve the impact of cloud storage requests from apps on the apps' cloud storage usage experience (e.g., response time, traffic/energy consumption).

There have been several **client-side middleware systems targeting client networking performance improvement**. QuickSync [13] is system that optimizes cloud storage synchronization performance in wireless networks based on network conditions. Li et al. proposed a middleware system to reduce session maintenance traffic generated by cloud storage applications [24]. Unidrive [33] is a client-side middleware system bringing multi-cloud capability to client devices. CacheKeeper [35] performs web caching at system level for mobile applications.

Our `StoArranger` system can also be viewed as a middleware system acting between mobile apps and cloud storage services. Our system coordinates and rectifies cloud storage requests originated from the apps such that the apps' cloud storage usage experience can be greatly improved.

6. Conclusion

In this paper, through comprehensive app studies, we show that currently many mobile apps incur poor cloud storage usage experience. While the direct cause of many of these problems is poor app implementation, the deeper root cause is that the way current cloud storage services are deployed onto mobile devices lacks OS-level support. We propose `StoArranger`, a system to interpose on network transmissions to address the problems.

Acknowledgments

We thank our shepherd, Dr. Mahadev Satyanarayanan, and the anonymous reviewers for their tremendously valuable feedbacks. This work was supported in part by NSF Award #1566375.

References

[1] 3G and 4G Wireless Speed Showdown: Which Networks Are Fastest? `http://www.pcworld.com/article/253808/3g_and_4g_wireless_speed_showdown_which_networks_are_fastest_.html`.

[2] Autosync-Dropbox. `https://play.google.com/store/apps/details?id=com.ttxapps.dropsync`.

[3] Charles Web Debugging Proxy. `http://www.charlesproxy.com`.

[4] The new /delta API call (beta). `https://blogs.dropbox.com/developers/2012/02/the-new-delta-api-call-beta/`, .

[5] Dropbox core API. `https://www.dropbox.com/developers-v1/core`, .

[6] Google Drive APIs - REST v2. `https://developers.google.com/drive/v2/reference/`.

[7] Average Internet Speeds Up, But U.S. Still Has Work to Do. `http://www.pcmag.com/article2/0,2817,2496861,00.asp`.

[8] libevent an event notification library. `http://libevent.org/`.

[9] Develop with the OneDrive API. `https://dev.onedrive.com/readme.htm`.

[10] OpenSSL, howpublished = `https://www.openssl.org/`.

[11] XFS. `http://xfs.org/index.php/Main_Page`.

[12] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. Energy consumption in mobile phones: a measurement study and implications for network applications. In *ACM IMC*, 2009.

[13] Y. Cui, Z. Lai, X. Wang, N. Dai, and C. Miao. Quicksync: Improving synchronization efficiency for mobile cloud storage services. In *ACM MobiCom*, 2015.

[14] S. Deng and H. Balakrishnan. Traffic-aware techniques to reduce 3g/lte wireless energy consumption. In *ACM CoNEXT*, 2012.

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

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

[17] Dropbox INC. Dropbox. `https://www.dropbox.com`.

[18] Google Inc. Google Drive. `https://www.google.com/drive/`.

[19] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. West. Scale and performance in a distributed file system. In *ACM SOSP*, 1987.

[20] W. Hu, T. Yang, and J. N. Matthews. The good, the bad and the ugly of consumer cloud storage. In *ACM SIGOPS Operating Systems Review*, 2010.

[21] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. A close examination of performance and power characteristics of 4g lte networks. In *ACM MobiSys*, 2012.

[22] J. Huang, F. Qian, Z. M. Mao, S. Sen, and O. Spatscheck. Screen-off traffic characterization and optimization in 3g/4g networks. In *ACM IMC*, 2012.

[23] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. In *ACM SOSP*, 1991.

[24] 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 *ACM Middleware*, 2013.

[25] 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 *ACM IMC*, 2014.

[26] Microsoft corporation. OneDrive. `https://onedrive.live.com`.

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

[28] M. Oltrogge, Y. Acar, S. Dechand, M. Smith, and S. Fahl. To pin or not to pinhelping app developers bullet proof their tls connections. In *USENIX Security*, 2015.

[29] F. Qian, Z. Wang, Y. Gao, J. Huang, A. Gerber, Z. Mao, S. Sen, and O. Spatscheck. Periodic transfers in mobile applications: network-wide origin, impact, and optimization. In *WWW*, 2012.

[30] M. Satyanarayanan, J. H. Howard, D. A. Nichols, R. N. Sidebotham, A. Z. Spector, and M. J. West. The itc distributed file system: Principles and design. In *ACM SOSP*, 1985.

[31] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on computers*, 39(4):447–459, 1990.

[32] A. Schulman, V. Navda, R. Ramjee, N. Spring, P. Deshpande, C. Grunewald, K. Jain, and V. N. Padmanabhan. Bartendr: a practical approach to energy-aware cellular data scheduling. In *ACM MobiCom*, 2010.

[33] H. Tang, F. Liu, G. Shen, Y. Jin, and C. Guo. Unidrive: Synergize multiple consumer cloud storage services. In *ACM Middleware*, 2015.

[34] Wikipedia. Representational state transfer. `https://en.wikipedia.org/wiki/Representational_state_transfer`.

[35] Y. Zhang, T. Chiu, and Q. Li. Cachekeeper: a system-wide web caching service for smartphones. In *ACM UbiComp*, 2013.