StoArranger: Enabling Efficient Usage of Cloud Storage Services on Mobile Devices

Yongshu Bai
Department of Computer Science
SUNY Binghamton
Binghamton, NY, USA

Yifan Zhang
Department of Computer Science
SUNY Binghamton
Binghamton, NY, USA

***Abstract*—Cloud storage usages are becoming increasingly popular on mobile devices. Through an extensive motivation study, we find that cloud storage accesses from mobile apps suffer from several notable problems that undermine usage experiences. The root cause is that the way of cloud storage providers deploying their services onto mobile devices relies on app developers for the correct and appropriate implementations and lacks the ability of monitoring and servicing client-side cloud storage accesses. We propose `StoArranger`, a practical system framework that solves the problems by coordinating, rearranging, and transforming cloud storage communications on mobile devices. We have prototyped the proposed system using two different implementation approaches. We discuss our experiences of the implementations in the paper. The real-app evaluation experiments show that `StoArranger` can significantly improve mobile cloud storage access efficiency with little overheads.**

***Index Terms*—Mobile cloud storage; Mobile device; Mobile apps; Middleware; Traffic reduction.**

I. Introduction

Personal cloud storage services, such as such as Dropbox [1], Google Drive [2] and OneDrive [3], have been widely used in our daily lives. In the meantime, with the trend that smart mobile devices are gradually replacing desktop computers to serve as users' main computing devices [4]–[7], cloud storage service usage on mobile devices is becoming popular rapidly. However, we find in our motivation study [8] that many of the existing mobile applications (*apps* for short hereafter) fall short of using cloud storage service efficiently, thus leading to poor usage experience. We identified four groups of inefficiency problems, which are briefly summarized as follows:

• *Uncoordinated cloud backup*: we find that mobile apps performing *automatic data backup* (§II) have their own policies on backup timing, and data backups are totally uncoordinated. Uncoordinated data backups can cause frequent and short live uplink data transmissions, which further lead to unnecessary energy consumption due to the accumulated 4G/3G/WiFi promotion/tail energy [9], [10] consumption.

• *Inefficient cloud folder synchronization*: we find that *folder synchronizations* (§II) implemented by many third-party mobile apps are highly inefficient. This is mainly because many third-party mobile app developers fail to properly use the suitable cloud storage development APIs to provide good usage experience (due to the reasons such as coding/maintenance complexity and developer experience).

• *Inefficient cloud file accesses*: we find that, for many apps that support *manual data browsing and backup* (§II), they often neglect supporting those mechanisms that are beneficial to cloud file access efficiency, such as caching and compression. For example, we find many apps either do not implement caching or provide weak/incorrect consistency with caching implemented. Most mobile apps also do not implement metadata caching for cloud folders.

• *Inefficient whole file transmission*: we find that, when synchronizing files between mobile device and storage server, mobile apps almost always perform whole-file transmission to/from the storage server even when the differences between the local copy and the storage server's copy are small. On Android, no app supports advanced file storage/transmission techniques, such as chunking, deduplication and delta-encoding.

The *direct causes* of the above inefficiency problems are mainly mobile app implementation related issues, such as programming proficiency, experience, carefulness of mobile app developers. However, we show that the *root causes* stem from the way that cloud storage providers choose to deploy their services onto mobile devices. Specifically, cloud storage services are deployed to mobile devices mainly through the usage of development APIs in mobile apps. This deployment approach entirely relies on app developers to properly use the suitable APIs in different usage scenarios, and to correctly implement/enable the practices that are beneficial to cloud storage usage experience (e.g., caching and compression). Moreover, this deployment approach also leads to the *inability of cloud storage services to monitor and service file operations on mobile devices* and the *loose coupling between mobile apps and storage servers*, both of which pose great difficulty in realizing system-level cloud storage service access optimization on mobile devices. More detailed discussion is deferred to §IV.

Motivated by the above findings, we propose `StoArranger`, a system that can rearrange, coordinate, and transform cloud storage accesses from mobile apps. The foundation of `StoArranger` is the enabling of cloud storage accesses monitoring and servicing on mobile devices, based on which a series of scheduling and adaptation optimizations can be done. For example, to solve the uncoordinated data backup problem, we should be aware if a file has been generated/changed and is being backed up to the cloud storage. Another example is that to enable caching for cloud storage backed files on

mobile devices, we need to monitor the file open requests and service them with the cached copies. One major *challenge* to achieve client-side file operations monitoring and servicing is the need of designing practical and scalable solutions suitable for existing mobile devices. Current solutions usually achieve the goal by customizing OS kernels (e.g., by intercepting file-related system calls or interposing on device drivers, more details in §IV), which is not practical for solving the problems that exist on millions of mobile devices already in use. Thus, one of our main goals is to effectively solve the problems *without changing either the apps or the underlying mobile OSes*. To this end, we have explored two design and implementation approaches (details later in §VI):

The *first* approach we try is to interpose on network transmissions caused by cloud storage accesses from mobile apps. In this approach, we redirect all network traffic through a user-level service, which performs scheduling/adaptation optimizations on cloud storage accesses related network transmissions. The advantage of this approach is that it naturally enables cloud storage accesses optimization in a system-wide manner. However, this approach has higher implementation complexity and tends to incur higher time overhead, when compared to the second approach below. Moreover, this approach relies on a system feature of network proxying, which may not be available in some platforms, despite the fact that it is offered in all major mobile OSes, such Android, iOS, and Windows.

The *second* approach is to interpose on app's critical library function call path related to cloud storage accesses. With this approach, we dynamically instrument optimization code into a running app's runtime environment (e.g., Dalvik [11], [12], ART [13], [14]), such that they will be invoked when the app accesses cloud storage services. Compared to the previous way, this approach has lower implementation and maintenance complexity. But to allow the optimization to have a system-wide effect, extra effort needs to be taken.

With the ability of monitoring and servicing cloud storage related operations on mobile devices, we design a series of solutions to solve the inefficiency problem identified in our motivation study. We discuss the associated challenges and the details of these solutions in §V. Our main contribution is that, to the best of our knowledge, we are the first to study and improve cloud storage usage experience on mobile devices from the perspective of individual mobile apps. Specifically,

• we experimentally reveal a series of common and notable inefficient cloud storage usage problems on mobile devices;
• we analytically investigate the causes of the problems, and give our insights of designing fundamental solutions;
• we design effective solutions for addressing the problems;
• we explore practical ways of designing the solutions such that the resulting system can be easily scaled to existing mobile devices without changing apps or OSes;
• we fully implement the proposed `StoArranger` system with two different implementation strategies, and provide our implementation experiences; and
• we evaluate the proposed system with real-app experiments, which show that our system can achieve its goals with little system overheads.

II. Background

Cloud storage usage classification. Cloud storage usages on PC or mobile devices can be generally classified into the following four categories:

• *Automatic data backup*. In this usage scenario, applications automatically backup data to cloud storage when the data is generated or changed. For instance, many mobile apps generating user files (e.g., photos, videos, notes and documents) now allow the data to be automatically backed up to cloud storage chosen by the user when when the data are generated. Examples include Facebook [15], Flickr [16], WPS Office [17], Evernote [18], and many others. Automatic cloud backup also allows mobile app developers to store/update their app data, such as configurations and databases, using cloud storage, so that their apps can seamlessly work on different devices of the same user without the need of reinventing the wheel of data synchronization [19]–[21].
• *Manual data browsing and backup*. In this usage scenario, users can manually browse their data stored on cloud storage via one of the access methods described next, and also can upload file(s) to the cloud storage.
• *Folder synchronization*. In this usage scenario, users can link a folder on their local devices (called the *local sync folder*) to an online storage folder (called the *online sync folder*), such that all changes made in either sync folder and all its sub-folders will be synchronized to their counterparts automatically [22]–[24]. This functionality allows users to have their data seamlessly synchronized across all their devices in real-time manner, and is thus one of the most important reasons for cloud storage's surging popularity.
• *Cloud-storage-backed online document editing*. In this usage scenario, users edit their documents (e.g., texts, spreadsheets, presentations, etc.) on their local devices. All the edits will be automatically saved to the cloud storage. Examples include document editor apps (e.g., Android apps [17], [25]–[27]) and pure online document editors (e.g., Google Docs [28] and Microsoft Office Online Apps [29]).

Cloud storage access methods. Users can access cloud storage through the following three methods:

• *Official proprietary apps*. Cloud storage providers usually provide their own proprietary applications for major operating systems, such as Windows, OSX, Android, and iOS. These official proprietary apps may support some or all the usage scenarios described previously. The advantage of official proprietary apps is that cloud storage providers can implement advanced file storage/transmission techniques, such as chunking/deduplication [30]–[35] and delta-encoding [36]–[40], in their apps to seamlessly work with their cloud storage servers, therefore achieving good performances and usage experience. However, deploying cloud storage services onto mobile devices using this approach is slow because of the wide range of application possibilities.
• *Third-party apps*. To overcome the slow service deployment problem with official proprietary mobile apps, cloud storage

providers offer their *development APIs* [41]–[43] to allow outside app developers to develop third-party apps making use of their services. These APIs are usually RESTful APIs [44] over HTTP. They allow apps to request or update resources (i.e., user files in cloud storage's case) on the servers by making HTTP requests (e.g., GET, PUT, POST, DELETE).

The adoption of RESTful APIs allows cloud storage providers to deploy their services to application-rich platforms, such as Android and iOS, easily and quickly, since they do not need to provide the concrete implementations of third-party/web cloud storage apps, which are now totally left to app developers. However, this deployment choice directly leads to the inefficient cloud storage usage problems on mobile devices (detailed analysis in §IV). It is worth noting that, since the broad diversity of mobile apps is the key reason for the success of mobile platforms, the usage experience of cloud storage services on mobile devices is largely determined by third-party apps that make use of cloud storage services.

III. The Motivation Study

In our motivation study, we performed a series of extensive real app experiments to evaluate cloud storage usage experience on mobile devices. Here we briefly introduce the study and the findings (please refer to [8] for the full details).

Finding 1: automatic data backups from mobile apps are uncoordinated. In this experiment, we studied 25 popular Android apps that perform automatic photo/video cloud backup. Most of them upload new/changed data immediately or shortly after the data are generated/changed. This policy can lead to a substantial amount of *energy waste* due to accumulated promotion and tail energy consumption [9], [10]. We then performed an experiment to study how the idea of coordinating and batching cloud upload requests (such that multiple requests can be served in a single burst) can save energy consumption for cloud backups, when using WiFi, 3G and 4G as the wireless transmission method. The result showed that the coordination/batching idea can save a significant amount of energy for the cellular cases: about 42% and 70% energy consumption can be saved for the 3G and 4G cases respectively. Therefore, 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.

Finding 2: folder sync implemented by mobile apps are highly inefficient. We also aimed to investigate the efficiency of the folder sync usage on mobile devices. To this end, we experimentally evaluated 20 Android apps supporting folder sync. Our observations are threefold. *First*, most apps incurred *long folder synchronization turnaround time* (i.e., the time needed to finish a folder sync operation) and *high synchronization traffic*, both of which were proportional to the number of sub-folders in the sync-folder. *Second*, there is a substantial amount of redundant sync traffic generated by many of the apps tested. *Third*, apps from the same developer that support different cloud storage service can have stark differences in terms of sync time and traffic.

Finding 3: manual data browsing operations are inefficient and sometimes incorrect. In this experiment, we examined mobile apps that support the manual data browsing and backup usage scenario. To do so, we chose 20 file manager/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 problems. *First*, many of the apps tested do not implement caching for files downloaded from the cloud. As a result, they re-download the whole file every time when user browses the cloud file via the app. *Second*, Some of the apps implement cloud file caching, but do not provide the correct consistency guarantee, causing user to access outdated files. *Third*, most of the apps do not implement metadata caching, meaning every time user browses 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 causes unnecessary metadata traffic, especially for folders that contain many items or those that are frequently accessed.

Finding 4: file transmissions from/to cloud storage on Android are always whole file transmissions. In all our experiments, we found that when mobile apps download/upload a file from/to the cloud storage, whole file transmission always happens even there is a very small change in the file, causing unnecessary network traffic. This observation is consistent with the observations in recent studies [45]–[47].

IV. Cause Analysis of the Inefficiency Problems

Although our real app studies above were performed on Android, we believe that the findings are likely to be applied to other mobile platforms, because the causes of the problems, as we analyze next, are not platform-dependent. There are two main causes for the problems we saw in our motivation studies.

Cause 1: inexperience and carelessness of mobile app developers. Since cloud storage providers rely on app developers to correctly implement cloud storage accesses and optimize usage experience, inexperience and carelessness of mobile app developers directly contribute to the poor mobile cloud storage experience we observed. For example, in our motivation study, we observed that most of the apps supporting folder synchronization we tested incurred long sync time and high traffic, both of which were proportional to the number of sub-folders in the sync-folder. This could be fixed by using a different set of cloud storage development APIs, which are, however, more difficult to use. Specifically, we found that the way that most of the apps performing folder sync is whole-hierarchy metadata tracking, which is to request metadata of every sub-folder of the sync folder, and compare them to the ones of local file system to determine if any files have been changed. 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 [48]–[50]. However, these *incremental sync APIs* require app developer to maintain a state of the local sync folder, and keep track of local changes between synchronizations, which is tedious and error-prone. By contrast, using whole-hierarchy 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 incremental sync APIs. This can explain why almost all the current mobile apps supporting folder sync adopt the whole folder metadata tracking method. Another example is that some apps we tested did not request compression encoding when communicating with the cloud storage server, which caused significant transmission bandwidth waste. According to our experiment results, this can be attributed to either inexperience of the developers, who do not realize the importance of compression, or carelessness of the developers, who fail to correctly follow the specific requirements made by different storage providers to request compression encoding.

Cause 2: lack of the ability of client-side cloud storage related file operations monitoring and servicing , and loose coupling between client and server. Modern cloud storage services essentially implement distributed file system (DFS) functionalities for individual users. In conventional DFS solutions, two key properties enable their correct functionalities and good performances.

• The first is the ability to monitor and service client side file operations. For example, traditional DFS solutions, such as AFS [51], [52], Coda [53], [54], LBFS [30], need to know when a client opens a file (so that the intended file can be fetched from the remote server or from the local cache), and when the client closes the file (so 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.

Existing DFS solutions usually enable the monitoring and servicing of client-side file operations by interposing on file system system calls or device drivers. For example, AFS [51], [52] intercepts file open and close system calls on the client side and forward them to a client-side cache-management process for processing. Coda [53], [54] inherited AFS's client-side design and added more functionalities to support disconnected operations and server replication. LBFS clients [30] resort to interposing on XFS [55] device driver to obtain notifications about file opens, closes and modifications, and to achieve the content-based breakpoint chunking.

• The second property is tight coupling between server and client. Many useful techniques in existing DFS solutions require close collaboration between server and client. For example, in AFSv2 [52] and Coda [53], [54], the callback-based cache coherence protocol requires both server and client to work together to ensure coherent client-side caching. In LBFS [30], the content-based chunking technique requires to apply the same chunking algorithm on both server and client.

However, mobile apps developed using cloud storage development (RESTful) APIs do not have the above two properties, which makes it hard to fundamentally solve the inefficiency problems. For example, since these RESTful APIs do not contain any OS-level mechanisms, such as system call/device driver interposition, to enable client-side file operation monitoring and servicing, 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 (e.g., caching, compression). Also, because of the lack of client-side OS-level support from the cloud storage providers, it is hard to coordinate cloud storage access request from mobile apps at system level. Lastly, 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 chunking/deduplication [30]–[35] and delta-encoding [36]–[40], without the cooperation from the server side.

V. Solutions Design

Driven by the findings and insights obtained in the motivation study, we propose `StoArranger`, a system aiming to enable efficient cloud storage usage on mobile devices by properly coordinating, rearranging, and transforming cloud storage accesses from mobile apps. `StoArranger` is founded on the ability of monitoring and servicing cloud storage accesses on mobile devices. In this section, we focus on discussing the designs of our solutions, assuming the availability of such ability. We describe how we achieve practical client-side cloud storage accesses monitoring and servicing next in §VI.

A. Cloud backups coordination

For solving the *uncoordinated cloud backup problem*, our observation is that cloud storage backup requests generated automatically by mobile apps are less delay-sensitive than those generated by explicit user intention. In other words, 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, the basic idea is to *delay and batch cloud backup requests from apps to minimize the impact of transmission promotion/tail energy while maintaining a normal user experience*. To achieve this goal, we need to address the following two main challenges (§V-A1 and §V-A2).

1) Batch size and upload timing determination: The first challenge is to determine how much cloud upload traffic to delay and the right timing of uploading the batched requests, which is critical to the performance of our system. Using the power models presented in work [10], we performed a simulation experiment to study the impact of upload traffic batch size on energy savings when using WiFi, 3G, and 4G for communication. We skip the details of this experiment here due to the space limit. The result suggests that it is not always optimal to batch as many backup request as possible. This is because the energy saving is not proportional to batch size. When the ratio between promotion+tail energy consumption and the transmission energy consumption decreases to certain value as the batch traffic increases, further increasing batch size would bring very marginal improvement.

Energy saving growth rate. In our solution, `StoArranger` keeps monitoring upload requests issued by the apps, and calculates the *energy saving growth rate*, based on which

the decision whether the current batched requests should be *flushed* (i.e., sent to their destinations) or not is made. The energy saving growth rate caused by batching a new upload request (notated as q) is determined as follows:

Suppose after batching a new upload request q, the size of all the batched upload requests is s bytes. According to the power model presented in [10], the *energy of transmitting all the batched requests in a single burst* can be calculated as :

$$E = (\alpha \cdot H + \beta) \cdot t_x + \rho \cdot T_{pt} \quad (1)$$

where H is upload throughput, α is a constant reflecting the impact of uplink throughput of the underlying wireless interface, β is the base power when throughput is zero, t_x is the time needed to transmitted the s bytes, ρ is the the average power during the promotion/tailing stage, and T_{pt} is the time duration of the promotion/tailing stage. Since $t_x = \frac{s}{H}$, the above can be rewritten as:

$$E = \alpha \cdot s + \beta \cdot \frac{s}{H} + \rho \cdot T_{pt} \quad (2)$$

Because the energy saving achieved by transmitting the new upload request q together with the exiting batched requests in a single burst (comparing to transmitting q separately) is $\rho \cdot T_{pt}$. We define the *energy saving ratio* by batching q as:

$$R = \frac{\rho \cdot T_{pt}}{E} = \frac{1}{(A + \frac{B}{H}) \cdot s + 1} \quad (3)$$

where $A = \frac{\alpha}{\rho \cdot T_{pt}}$ and $B = \frac{\beta}{\rho \cdot T_{pt}}$. Finally, we define the *energy saving growth rate* caused by batching the new upload request q as *the derivative of R with respect to s*:

$$G = \frac{dR}{ds} = \frac{M}{(M \cdot s + 1)^2} = \frac{1}{Ms^2 + 2s + \frac{1}{M}} \quad (4)$$

where $M = A + \frac{B}{H}$. As we can see, although keeping batching new upload requests can always lead to energy saving, the energy saving growth rate decreases exponentially as the batch grows. Next, we discuss how the energy saving growth rate is used in our upload timing policy.

Upload timing determination. Figure 1 shows the decision flow of how the timings of uploading batched requests are determined. The idea is to balance the energy saving achieved by delaying cloud upload requests from apps and data freshness on the cloud. Specifically, the existing batched requests are flushed to cloud storage if one of the following conditions (Ⓒ1-Ⓒ4) is true. Ⓒ1-when intercepting a new upload request q, which causes the energy saving growth rate G (Formula 4) to become smaller than the predefined threshold. Ⓒ2-when intercepting a new upload request q, which was explicitly initiated by the user (described next in §V-A2). Ⓒ3-when the wireless interface is awaken by incoming/outgoing traffic. Ⓒ4-when the system has been idle longer than a predetermined threshold.

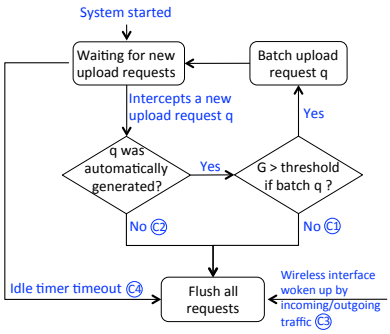

Fig. 1. Upload timing decision flow.

2) Identifying automatic cloud uploads by mobile apps: Another difficulty is that, to preserve the correct user experience, `StoArranger` needs to distinguish cloud upload requests initiated explicitly by users in the foreground from those automatically scheduled by mobile apps in the background. However, without changing mobile OS or apps, it is difficult to do so, because all cloud upload requests would look the same to the `StoArranger`. We make two observations that can help address this issue. First, when a user explicitly selects a file to upload, he needs to interact with an app that has the cloud upload functionality. Therefore, when `StoArranger` intercepts a cloud upload request, it checks whether the foreground app allows user to explicitly request such an upload. If not, we can safely assume the upload request being handled was automatically generated by a background app and thus can be delayed. However, it is possible that automatic backups by background apps happen when user is interacting with a foreground app, which also allows user to initiate such backups manually. For example, in our motivation study we observe that some photo/video backup apps do not start the backup until user quits from the camera app. In this case, if user opens a cloud storage capable app after quitting from the camera app, we will not be able to correctly identify the automatic backups and therefore miss the optimization opportunity. Through extensive app study, we make another observation, which can help mitigate this problem. We find that most apps allowing users to explicitly select and upload files to cloud storage use a system service named "Document" to allow user to select the files to upload. As a result, suppose user uses such an app, notated as "A", to select a file to upload, we will see the foreground app changes from "A" to "Document" and then back to "A" right before the upload request is generated. We named this foreground app change pattern as "*ADA*" pattern. `StoArranger` also monitors foreground app history and looks for such an *ADA* foreground app pattern to determine whether a cloud upload is explicitly requested by the user.

B. Efficient cloud-based folder synchronization

As we analyzed in Section IV, the direct cause of the long sync turnaround time and high sync traffic of many folder sync app is because of the whole-hierarchy metadata tracking approach they adopt. Developers choose such a way instead of the preferable incremental sync APIs provided by most cloud storage providers [48]–[50], because the whole-hierarchy metadata tracking way is easier to use. Motivated by these insights, to solve the *inefficient cloud folder sync problem*, our design is essentially to transform the efficient whole hierarchy metadata tracking to the use of incremental sync APIs. More specifically, upon detecting the problematic folder sync activity (i.e., whole-hierarchy cloud storage metadata tracking by apps), `StoArranger` takes advantage of the incremental sync APIs provided by most of the cloud storage APIs to minimize sync traffic and turnaround time.

Besides efficiently implementing the folder sync functionality using the more complicated incremental sync APIs,

the major challenge of realizing this transformation is to correctly detect the onset of problematic folder sync. 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 apps/OSes, it is difficult to tell if a request comes from problematic folder sync or normal cloud storage operations.

The straightforward way to solve the above challenge is to first mirror all of the cloud storage folder meta-data locally. Then `StoArranger` can carry out the synchronization between the `StoArranger`-maintained cloud folder metadata and its counterpart on the cloud using the efficient delta folder sync APIs. By doing so, there is no need to distinguish problematic folder sync from normal cloud storage operations, because all metadata requests intercepted can be served by the device component from the cloud folder metadata it maintains. However, the above approach can trigger a substantial amount of overhead for the `StoArranger` system, because every cloud storage folder and all of their sub-folders needed to managed by the system. Therefore, it is impractical to mirror the metadata of the entire cloud storage hierarchy locally.

Our solution is based on the following two observations. One is that folder syncs based on whole-hierarchy metadata tracking request the metadata of all the folders in a depth-first or width-first order within the root sync folder, while metadata requests caused by other usage scenarios do not have specific order. The other observation is that the variances of the intervals between metadata requests from a folder sync are more stable than metadata requests caused by explicit user operations (e.g., in the manual data browsing scenario). As a result, to determine whether a given set of metadata requests are caused by whole-hierarchy metadata tracking based folder sync, `StoArranger` monitors all the intercepted folder metadata requests, and checks whether these requests meet the above observations. If so, we can determine that those metadata requests are part of a problematic folder sync, and therefore transform only those metadata requests using the efficient delta folder sync APIs.

C. Efficient cloud file access

To solve the *inefficient cloud file accesses problem*, the `StoArranger` system performs cloud file caching and metadata caching for apps making access to cloud storage. Specifically, `StoArranger` monitors all the outgoing requests for downloading cloud storage files or obtaining cloud folder metadata, and manages to cache the files or metadata returned by the server. Each cached item is uniquely identified by the URL used to request it. With this functionality, `StoArranger` can serve most of 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/metadata 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 processing overhead and high degree of volatile/permanent storage duplication. We present our way of efficient implementation next in §VI, followed by the evaluation in §VII.

VI. System implementation

To allow the `StoArranger` system to be practical and easily scalable to existing mobile devices, the central objective for the `StoArranger` implementation is to have a fully working system addressing the inefficiency problems without changing either the mobile OSes or the apps. We explored two approaches to achieve this objective.

A. Implementation based on network traffic redirection

The first approach we tried for achieving client-side cloud storage accesses monitoring and servicing is to interpose on the network traffic caused by cloud storage related operations. This choice was motivated by the following two observations. *1)* Every cloud storage operation (e.g., getting file/folder metadata, downloading/uploading resources) eventually leads to an HTTP transaction between the mobile device and the server. *2)* All the major mobile OSes (e.g., Android, iOS and Windows) support a system network proxying feature, which allows users to redirect network traffic (WiFi or cellular) to a specified proxy server without changing OS kernels [56]–[59]. In this implementation (Figure 2), network traffic of all the apps is redirected to a user-level `StoArranger` service, which implements the solutions described in §V. The traffic redirection was achieved by configuring the Android proxy server [56], [57] to an IP loopback address (127.0.0.1, or the `localhost`), on which the user-level service is listening.

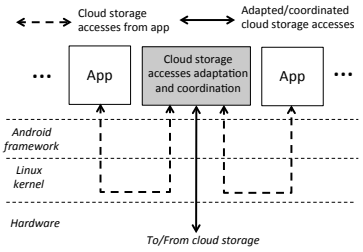

Fig. 2. Network traffic redirection based implementation overview.

There are two main challenges in this implementation. One is that, since almost all cloud storage related traffic are wrapped by HTTPS, correctly and efficiently implementing the SSL/TLS proxying in the user-level `StoArranger` service is important. In our implementation, we use the SSL library provided by the well-known OpenSSL project [60] to implement the SSL/TLS management component for the user-level service. The other challenge is the need of achieving high runtime efficiency for the `StoArranger` service. This is because with this implementation, network packets need to travel double the distance of the original system in the device. An inefficient implementation causing long time overhead would notably compromise the user experience when running the `StoArranger` service. Our experience of addressing this issue is that the runtime efficiency of the `StoArranger` service is largely affected by the way how the service is monitoring the inbound/outbound traffic. For example, an implementation based on polling can result in noticeable app sluggishness and high computation resource usage. Our implementation is based on the framework of

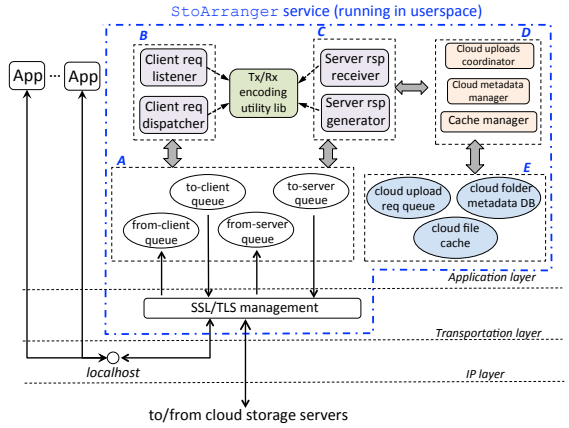

Fig. 3. Implementation of the user-level `StoArranger` service.

SSLsplit, a tool for man-in-the-middle attacks against SSL/TLS encrypted network connections [61]. We realized an event-driven model for handling the network events delivered to the `localhost` interface using the *libevent* API [62], which enables asynchronous event notification by allowing our handing callback functions to be invoked when the corresponding network events happen.

Figure 3 shows the implementation architecture of the user-level `StoArranger` service. There are five parts related to the design introduced previously. *1*) The first part (part A in Figure 3) is a set of four network message buffer queues that are directly interfaced with the apps on the device and the remote storage server. These queues are managed using the *libevent* library [62] API. Among them, the *from-client* queue is used to receive network traffic originated from the apps; the *to-server* queue stores the traffic to be sent out to the server; the *from-server* queue stores the traffic received from the server; and the *to-client* queue stores the network traffic to be sent back to the apps. *2*) The second part (part B) contains the *client request listener*, which implements the handling logic for event happening at the *from-client* queue; and the *client request dispatcher*, which is responsible for relaying the unchanged client requests (e.g., those not related to cloud storage) or putting the adapted client requests to the *to-server* queue. *3*) Similarly, in the third part (part C), the *server response receiver* handles events from the *from-server* queue; and the *server response generator* is interfaced with the *to-client* queue, and handles the cases where client requests can be served locally without contacting the storage server (e.g., serving with the cached content). The components in both part B and C use the transmission/receiving utility library to organize the traffic content such that the `StoArranger` implementation can better scale to different cloud storage services and is resilient to communication flow format changes in exiting services (details later in §VI-C). *4*) The fourth part (part D) contains the three decision engines that manages the three main functionalities of our system: cloud upload coordination, folder sync, and system-wide cloud file/metadata caching. The interfaces implemented in these three engines are used by the components in part B and C to achieve the designs described in §V. *5*) The fifth part (part E) is three memory-based buffers that are managed by the three decision engines in part D.

Next, we make two examples to demonstrate how these components work together. The first example is that for network traffic that are unrelated to cloud storage, they are directly relayed from the *from-client* queue to the *to-server* queue (for outbound traffic), and from the *from-server* queue to the *to-client* queue (for inbound traffic). Another example is that for cloud backups coordination, all backup requests originating from the apps are rerouted to the `localhost` interface, from which the `StoArranger` service fetches the rerouted traffic for processing upon being notified about the events. After the decryption process by the SSL/TLS management component, the backup requests are put into the *from-client-queue*, which automatically triggers the *client request listener* to process these backup requests. The *client request listener* hands over the backup requests to the *cloud upload coordinator*, which runs the algorithm described in §V-A to determine the best timing for flushing the batched requests stored in the *cloud upload request queue*. Once such decision is made, the batched requests are given to the *client request dispatcher* to send to the server.

B. Implementation based on app runtime instrumentation

The advantage of the previous implementation is that it naturally enables system-wide optimization for cloud storage accesses because of the system network traffic proxying feature. However, the need of handling HTTPS traffic introduces two limitations. The first is that it requires a system SSL certificate to be installed on the device for the `StoArranger` service, with which some users may not feel comfortable. The second limitation is that the solution cannot work with apps implementing SSL pinning [63] without the cooperation from the service providers. Moreover, as we can see, the implementation complexity is relatively high.

We aimed to design another way to implement the `StoArranger` functionalities while addressing the above limitations. To avoid dealing with SSL/TLS traffic, we need to have a way to handle the traffic before/after they are encrypted/decrypted. Our observation is that, take Android as an example, calls to cloud storage APIs will eventually cause calls to Android HTTP library functions (e.g., the `HttpURLConnection` class or the `Apache HTTP Client` class [64]). Therefore, we can hook Android HTTP library calls to intercept cloud storage API calls made by apps. Figure 4 shows the high level idea of this mechanism. In Android, the runtime system (i.e., Dalvik [11], [12] or ART [13], [14]) uses a data structure called "`vtable`" to bridge virtual methods and the actual method instances. By

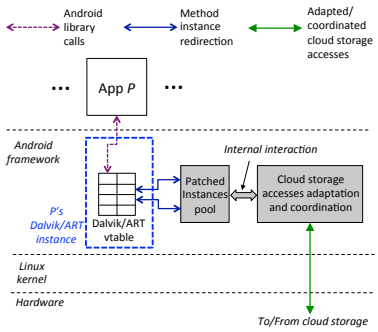

Fig. 4. App runtime instrumentation based implementation overview.

manipulating the `vtable` of the runtime system, we can hook Android library function calls in both Dalvik (using techniques like Dynamic Dalvik Instrumentation [65]) and ART (using a similar technique [66]). In this implementation, by instrumenting the app runtime system, we redirect those HTTP library related virtual methods to our own patched method instances. These patched method instances work with our middleware framework runtime component, which implements the solutions described in §V.

C. Scalability to different cloud storage services and resilience to communication format changes

Mobile apps communicate with the cloud storage servers usually need to conform with certain communication formats defined by the service provider. For example, cloud storage messages exchanged between app and server contain information describing different properties of the communication, such as request ID, response ID, session ID, resource size, etc. Different service providers define their own formats regarding how these information should be organized or parsed by the apps. Moreover, even for the same provider, its communication format may be updated over time. Hard-coding the `StoArranger`'s optimization logic conforming to an existing communication format is not a good option, since a small change in the format would render it unusable, requiring recoding and reinstallation. Therefore, it is desirable to have a solution for `StoArranger` to easily scale to many different cloud storage services, or adapt to frequent changes in communication formats. To address this challenge, we design a mechanism to describe cloud storage communication flow formats. With this mechanism, how different types of cloud storage requests/responses should be transformed while conforming to the communication format is describe using XML. Our implementation of `StoArranger` is essentially an interpreter of these XML description. By doing so, both dealing with changes of communication flow format in existing cloud storage services and adding any new cloud storage services would be a matter of updating the XML description.

We were actually benefited from this design: during our implementation, Google Drive added a new request/response pair at the beginning of the whole flow, which caused the mobile apps ceased to respond to the `StoArranger`. With the XML description interpretation functionality, we simply added this new request/response pair to the XML description to make the whole process work again, without reprogramming/reinstalling the `StoArranger` system.

D. The prototype system

We have prototyped the proposed `StoArranger` system on two Android platforms: Samsung Galaxy S4 running Android 4.4.4 and Motorola Moto G2 running Android 5.1.1. In our prototype system, all the designs and implementations introduced previously are packed in to an Android application package (APK), which can be installed onto Android devices via the normal installation mechanism. For the network traffic redirection based approach, the installation process automatically configures the system proxy server, installs the `StoArranger` system SSL certificate, and the `StoArranger` service. For the app runtime instrumentation based approach, the installation procecss installs a background daemon, which monitors app launches and instruments the newly launched app's runtime (Dalvik or ART) dynamically.

VII. System Evaluation

We used our prototype system on Samsung Galaxy S4 running Android 4.4.4 in the evaluation experiments.

Cloud backup coordination. As discussed previously, properly determining batch size and upload timing of cloud upload requests is critical to achieving good performance for cloud backup coordination performance with `StoArranger`. In this experiment, we designed a program to simulate the cloud upload scenario on mobile device and evaluate our approach. In each round of testing, the program sequentially generated 50 upload requests. The size of each upload was randomly decided from the range between 20 KB to 1 MB. For a new upload request, the program ran `StoArranger`'s upload timing determination algorithm to decide whether the request should be added to the exiting batch, or be sent to the storage server along with the existing batched requests. If the latter is true, we mark this upload request as a "trigger request". For each wireless communication interface (4G, 3G, and WiFi), we tested two uplink bit rates: the base bit rate of 1 Mbps, and the average bit rate in practice according to the literature. For 4G/3G/WiFi, this average uplink bit rate was set to 6/2/12 Mbps respectively [67], [68]. For each bit rate, two rounds of evaluation were performed.

Figure 5 (a), (b) and (c) show the results for 4G, 3G and WiFi. The x-value of each point in the figures represents the index of an upload request, and the y-value is the energy saving ratio if the upload request is chosen as the trigger request. The energy saving ratio is calculated as $\frac{E'-E}{E}$, where E is the energy consumption (calculated using the power models [10]) of uploading the trigger request along with the batched requests in one burst, and E' is the energy consumption of sending all the requests separately. The arrows in each figure point to the actual trigger requests selected by `StoArranger`. From the results we can see that `StoArranger` can achieve significant energy consumption savings for the uploads, especially for the 4G case (about 58% to 75% energy reduction) and the 3G case (about 45% to 55% reduction). Also, `StoArranger` strikes a good balance between energy saving and upload freshness: further batching of upload requests in all test rounds would only bring very marginal gain on energy reduction.

Cloud-based folder synchronization. To evaluate how `StoArranger` can improve performances of cloud-based folder synchronization for mobile apps, we performed tests for four popular apps offering the functionality on our prototype systems (i.e., the implementation based on network traffic redirection, short as NTR below; and the implementation based on app runtime instrumentation, short as ARI below). Due to the space limit, we report the results for the OneSync app [69]. All other apps we tested have the similar results.

In the evaluation, we used the similar setting as in our motivation study: i) Two metrics are evaluated: *folder sync*

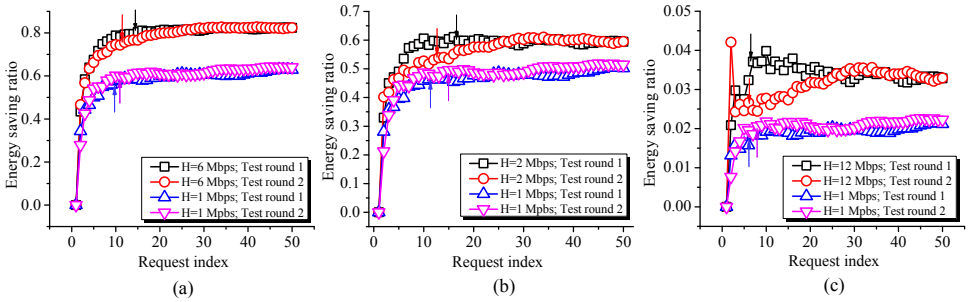

Fig. 5. Cloud backup coordination upload timing determination with three different wireless interfaces - (a) 4G, (b) 3G, and (c) WiFi.

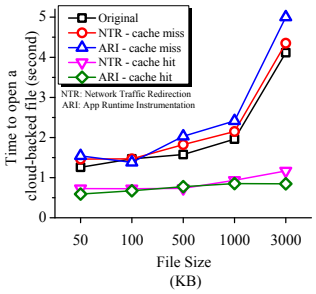

Fig. 6. Caching performance for cloud file accesses.

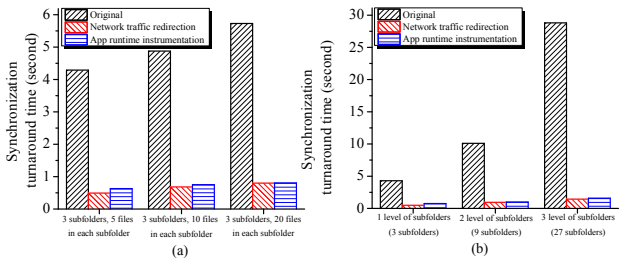

Fig. 7. Folder synchronization turnaround time.

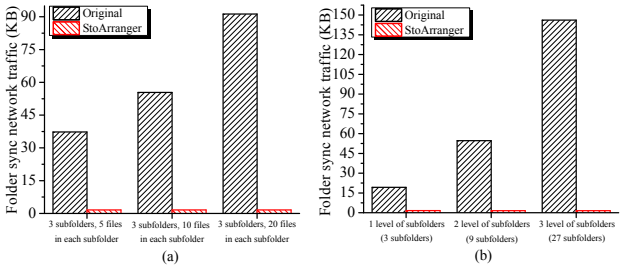

Fig. 8. Folder synchronization network traffic

turnaround time, which is the time duration of whole sync (Figure 7); and *folder sync traffic*, which is total amount of traffic generated during the sync (Figure 8). ii) For each metric's evaluation, the root sync folder has two configurations: one is that the root folder has a fixed number of 3 sub-folders, the number of files contained in each sub-folder varies (sub-figures (a) in Figure 7 and 8); the other is that the root folder contains a varied number of levels of sub-folders, each level has 3 sub-folders, and the leaf sub-folder has 5 files (sub-figures (b) in Figure 7 and 8). iii) Before each sync operation, the local sync folder is already consistent with the cloud folder. The sync operation just confirms local has been synced with the cloud. No file download/upload actually happens. From the results we can see that without running `StoArranger`, the app incurred a long turnaround time and high sync traffic. For example, the sync turnaround time and network traffic for the configuration of 3 sub-folders with 20 files each are 6 seconds and 91 KB. Similarly, these values for the configuration of 3 levels of sub-folders are 29 seconds and 146 KB. As a comparison, with `StoArranger`(both the NTR and ARI implementations) and for all the cases, the sync turnaround time ranged between 0.4 to 1.6 seconds, and the traffic generated was around 1.5 KB.

Cloud storage file/metadata caching. We evaluate how cloud file/metadata caching provided by `StoArranger` can improve file access performance using our prototype systems. In this experiment, we opened files of different sizes stored on Microsoft OneDrive [3] using the official OneDrive app [70], and recorded the time needed and network traffic generated. We performed the tests in five rounds, and report the averages here. Figure 6 shows the results of the time metric. With the original system, files are fetched from storage server every time time when they are opened. With `StoArranger`, in the case of cache miss, files are fetched from cloud server and stored into the cloud file cache; in the case of cache hit, the file open operations by the app are serviced by `StoArranger` with the local cached data.

From the results we can see that `StoArranger` can notably improve file access performances for the cache hit case: on average, it can save about 60% of the time needed to open a file for both implementation approaches when compared to the original system (and 100% of network traffic). In the case of cache miss, `StoArranger` imposes about 15% of time overhead for opening cloud-stored files. To our surprise, the ARI implementation incurs more time overhead than the NTR approach on cache misses, which is contrary to the fact that the NTR approach has longer data travel path than ARI. Our conjecture is that, with our current implementation of ARI, scheduling and adaptation operations are implemented using Android framework APIs, which can cause overhead due to the interpretation execution nature of Dalvik [11]. We are now working on translating the ARI implementation to using native code to improve the ARI approach's performance on Dalvik.

System overhead for non-cloud-storage traffic. Our system can significantly improve cloud storage accesses performance. However, since non-cloud-storage network traffic also needs go through `StoArranger` (i.e., the `StoArranger` service with NTR, and the instrumented component with ARI), in which case `StoArranger` just simply relays them between the server and the traffic-originating app, they can suffer from certain performance degradation. To evaluate this aspect, we designed an app that uploads/downloads files to/from a HTTP server. Figure 9 shows the time needed to transfer the non-cloud-storage files of different sizes using this app. In the meantime, to evaluate how our system is resilient to background computation load, we ran a computation-intensive program that imposed different background loads on CPU. Due to the space limit, we report the case when the background CPU usage is 75%, which is shown in Figure 10. All the data in Figure 9 and 10 are the average of five different test rounds.

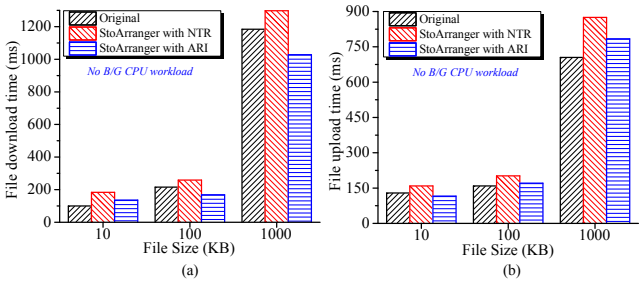

Fig. 9. Time overhead for non-cloud-storage traffic (without background CPU load).

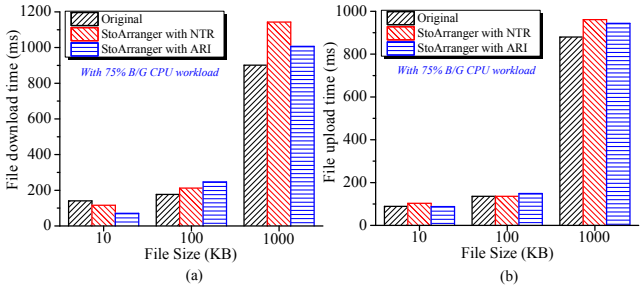

Fig. 10. Time overhead for non-cloud-storage traffic (with 75% background CPU load).

From the results we can see that StoArranger causes only a small amount of time overhead for transmitting non-cloud-storage traffic. For example, even when the background CPU usage was 75%, the NTR approach added about an average of 15%/10% of extra time for downloading/uploading the files, the values of the ARI approach were around 8%/6%. It is worth noting that the results show the ARI implementation approach generally incurred less time overhead than the NTR approach for non-cloud-storage traffic, which supports our motivation of exploring the ARI method in the first place. Moreover, like imposing background CPU load, we have also added different background memory usage and I/O activities to the experiments, where similar results have been obtained.

TABLE I
POWER CONSUMPTIONS (UPLOADING CLOUD-STORAGE FILES WITH NTR)

File size (KB)	5	10	50	100	500	1000
Original (mW)	532.3	541.1	563.1	610.5	629.6	647.6
With SA (mW)	551.8	561.4	599.7	617.1	639.4	660.9
Overhead	3.6%	3.7%	3.5%	1.0%	1.5%	2.0%

Power consumption overhead. We have also evaluated the power consumption overhead of our system with real-app experiments. We used a power monitor [71] to measure the real power consumption when using the official OneDrive app to upload/download files. Table I shows the power consumptions for uploading case, which are calculated as the difference between the system power when uploading and the baseline system power. The result suggests that StoArranger incurs a very small amount of power overhead (around 1% to 3%).

VIII. Related Work

Cloud storage services measurement and characterization (e.g., [45], [46], [72], [73]). Hu et al. conducted an early study on comparing cloud storage services' backup and restoration performances [72]. Drago et al. [73] 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 [45]. 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 [46].

Minimizing mobile traffic and energy consumption through transmission scheduling (e.g., [9], [74]–[77]). TailEnder [9] schedules transmissions to minimize tail energy consumption while meeting the deadlines of transmission requests. Bartendr [74] is a system performing energy-aware cellular data scheduling base on the prediction of cellular signal. Deng and Balakrishnan proposed to turn on/off the cellular radio interface base on the traffic pattern history and prediction [75]. Qian et al. performed a large-scale measurement study [76] on the impact of cellular network periodic transfers, which can incur significant energy overhead due to accumulative promotion/tail energy. Huang et al. [77] proposed to treat traffic generated when screen is off differently from those generated when screen is on, based on the observation that screen off traffic is more delay-tolerant than screen-on traffic.

Client-side middleware systems targeting client networking performance improvement (e.g., [47], [78], [79]). QuickSync [47] 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 [80]. Unidrive [78] is a client-side middleware system bringing multi-cloud capability to client devices. CacheKeeper [79] performs web caching at system level for mobile applications.

IX. Conclusion and Future Work

We have presented StoArranger, a user-space middleware framework that can significantly improve cloud storage usage experience for mobile devices. StoArranger addresses the problems of inefficient mobile cloud storage accesses, which were identified in our extensive motivation study, by coordinating, rearranging, and transforming cloud storage communications on mobile devices. We have implemented StoArranger with two different implementation approaches. The real-app evaluation experiments suggest that StoArranger can effectively achieve its goals with small system overheads.

Our current system has not yet solved the fourth problem identified in our motivation study, which is the always-whole-file-transmission problem. The existing solutions, such as chunking, deduplication, and delta-encoding are not suitable for solving the problem because of low deduplication ratio of these methods on many popular document formats [81] and resource constraints of mobile devices [47]. Meanwhile, solutions for this problem require collaboration from the storage infrastructure, which is beyond the focus of this paper. We are now researching on a solution suitable for mobile devices with the help from edge equipments, and hope to report our experience about it in the near future.

ACKNOWLEDGEMENT

We thank the anonymous reviewers for their tremendously valuable feedbacks. This work was supported in part by NSF Award #1566375.

REFERENCES

[1] Dropbox Inc., "Dropbox," https://www.dropbox.com.
[2] Google Inc., "Google Drive," https://www.google.com/drive/.
[3] Microsoft Corporation, "OneDrive," https://onedrive.live.com.
[4] D. Gilbert, "Microsoft wants to replace your PC with your smartphone," http://www.ibtimes.co.uk/microsoft-wants-replace-your-pc-your-smartphone-1499042.
[5] L. Eadicicco, "HP's New Smartphone Wants to Replace Your Work Computer," http://time.com/4227980/hp-elite-x3-windows-10-continuum/.
[6] J. Valcarcel, "In Less Than Two Years, a Smartphone Could Be Your Only Computer," http://www.wired.com/2015/02/smartphone-only-computer/.
[7] S. J. Purewal, "10 ways your smartphone has already replaced your laptop," http://www.greenbot.com/article/3006339/smartphones/10-ways-your-smartphone-has-already-replaced-your-laptop.html.
[8] Y. Bai, X. Zhang, and Y. Zhang, "Improving cloud storage usage experience for mobile applications," in *ACM APSys*, 2016.
[9] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani, "Energy consumption in mobile phones: a measurement study and implications for network applications," in *ACM IMC*, 2009.
[10] 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.
[11] Wikipedia, "Dalvik (software)," https://en.wikipedia.org/wiki/Dalvik_(software).
[12] D. Ehringer, "The dalvik virtual machine architecture," *Techn. report (March 2010)*, vol. 4, p. 8, 2010.
[13] "Android runtime (art)," https://en.wikipedia.org/wiki/Android_Runtime.
[14] A. Frumusanu, "A closer look at android runtime (art) in android l," *AnandTech: http://www.anandtech.com/show/8231/a-closer-look-at-android-runtime-art-in-android-l/*, 2014.
[15] Facebook, "Android app: Facebook(version 28.0.0.20.16) ," https://play.google.com/store/apps/details?id=com.facebook.katana&hl=en.
[16] Yahoo, "Android app: Flickr (version 3.2.2) ," https://play.google.com/store/apps/details?id=com.yahoo.mobile.client.android.flickr&hl=en.
[17] Kingsoft Office Software Corporation Limited, "Android App: WPS Office + PDF," https://play.google.com/store/apps/details?id=cn.wps.moffice_eng&hl=en.
[18] Evernote Corporation, "Android app: Evernote," https://play.google.com/store/apps/details?id=com.evernote&hl=en.
[19] Y. Go, N. Agrawal, A. Aranya, and C. Ungureanu, "Reliable, consistent, and efficient data sync for mobile apps," in *USENIX FAST*, 2015.
[20] D. Perkins, N. Agrawal, A. Aranya, C. Yu, Y. Go, H. V. Madhyastha, and C. Ungureanu, "Simba: tunable end-to-end data consistency for mobile apps," in *ACM EuroSys*, 2015.
[21] B.-G. Chun, C. Curino, R. Sears, A. Shraer, S. Madden, and R. Ramakrishnan, "Mobius: unified messaging and data serving for mobile apps," in *ACM MobiSys*, 2012.
[22] Dropbox Inc., "What is the Dropbox desktop application?" https://www.dropbox.com/en/help/65.
[23] Google Inc., "How Google Drive for Mac/PC works," https://support.google.com/a/answer/2490101?hl=en.
[24] Microsoft Corporation, "OneDrive desktop app for Windows 8.1," https://support.office.com/en-us/article/OneDrive-desktop-app-for-Windows-8-1-850703dd-ea56-4c7a-bff5-6c2e4da227cf.
[25] MobiSystems, "Android app: OfficeSuite + PDF Editor," https://play.google.com/store/apps/details?id=com.mobisystems.office.
[26] Microsoft Corporation, "Android app: Microsoft Word," https://play.google.com/store/apps/details?id=com.microsoft.office.word.
[27] Microsoft , "Android app: Microsoft Excel," https://play.google.com/store/apps/details?id=com.microsoft.office.excel.
[28] Google Inc., "Google Docs," https://www.google.com/docs/about/.
[29] M. Corporation, "Microsoft office online apps," https://products.office.com/en-us/office-online/documents-spreadsheets-presentations-office-online.
[30] A. Muthitacharoen, B. Chen, and D. Mazieres, "A low-bandwidth network file system," in *ACM SOSP*, 2001.
[31] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer, "Farsite: Federated, available, and reliable storage for an incompletely trusted environment," in *USENIX OSDI*, 2002.
[32] L. P. Cox, C. D. Murray, and B. D. Noble, "Pastiche: Making backup cheap and easy," in *USENIX OSDI*, 2002.
[33] S. Quinlan and S. Dorward, "Venti: A new approach to archival storage." in *USENIX FAST*, 2002.
[34] N. Tolia, M. Kozuch, M. Satyanarayanan, B. Karp, T. C. Bressoud, and A. Perrig, "Opportunistic use of content addressable storage for distributed file systems." in *USENIX ATC*, 2003.
[35] S. Annapureddy, M. J. Freedman, and D. Mazieres, "Shark: Scaling file servers via cooperative caching," in *USENIX NSDI*, 2005.
[36] A. Tridgell, P. Mackerras *et al.*, "The rsync algorithm," 1996.
[37] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy, "Potential benefits of delta encoding and data compression for http," in *ACM SIGCOMM Computer Communication Review*, 1997.
[38] D. Rasch and R. C. Burns, "In-place rsync: File synchronization for mobile and wireless devices." in *USENIX ATC*, 2003.
[39] F. Douglis and A. Iyengar, "Application-specific delta-encoding via resemblance detection." in *USENIX ATC*, 2003.
[40] T. Knauth and C. Fetzer, "dsync: Efficient block-wise synchronization of multi-gigabyte binary data," in *USENIX LISA*, 2013.
[41] Dropbox Inc., "Dropbox core API," https://www.dropbox.com/developers-v1/core.
[42] Google Inc., "Google Drive APIs - REST v2," https://developers.google.com/drive/v2/reference/.
[43] Microsoft Corporation, "Develop with the OneDrive API," https://dev.onedrive.com/readme.htm.
[44] Wikipedia, "Representational state transfer," https://en.wikipedia.org/wiki/Representational_state_transfer.
[45] I. Drago, E. Bocchi, M. Mellia, H. Slatman, and A. Pras, "Benchmarking personal cloud storage," in *ACM IMC*, 2013.
[46] 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.
[47] Y. Cui, Z. Lai, X. Wang, N. Dai, and C. Miao, "Quicksync: Improving synchronization efficiency for mobile cloud storage services," in *ACM MobiCom*, 2015.
[48] Dropbox Inc., "The new /delta API call (beta)," https://blogs.dropbox.com/developers/2012/02/the-new-delta-api-call-beta/.
[49] Google Inc., "Synchronize Resources Efficiently," https://developers.google.com/google-apps/calendar/v3/sync.
[50] Microsoft Corporation, "View changes for a OneDrive Item and its children," https://dev.onedrive.com/items/view_delta.htm.
[51] 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.
[52] 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.
[53] 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*, vol. 39, no. 4, pp. 447–459, 1990.
[54] J. J. Kistler and M. Satyanarayanan, "Disconnected operation in the coda file system," in *ACM SOSP*, 1991.
[55] "XFS," http://xfs.org/index.php/Main_Page.
[56] Google Inc., "Connect to Wi-Fi networks-Advanced Wi-Fi settings-Configure proxy settings," https://support.google.com/nexus/answer/2819519?hl=en.
[57] Techverse, "Configure a Proxy Server with 3G or 4G Data Connection on Android," http://www.techverse.net/how-to-setup-proxy-server-3g-4g-data-connection-android-phone/.
[58] Apple Inc., "Set advanced network settings in the iOS Setup Assistant," https://support.apple.com/en-us/HT202693.
[59] Microsoft Corporation, "How to configure proxy server settings in Windows 8," https://support.microsoft.com/en-us/kb/2777643.
[60] "OpenSSL: Cryptography and SSL/TLS Toolkit," https://www.openssl.org/.
[61] "SSLsplit - transparent SSL/TLS interception," https://www.roe.ch/SSLsplit.

[62] "libevent - an event notification library," http://libevent.org/.
[63] M. Oltrogge, Y. Acar, S. Dechand, M. Smith, and S. Fahl, "To pin or not to pin - helping app developers bullet proof their tls connections," in *USENIX Security*, 2015.
[64] Jesse Wilson, "Android's HTTP Clients," http://android-developers.blogspot.com/2011/09/androids-http-clients.html.
[65] Collin Mulliner, "Android DDI: Dynamic Dalvik Instrumentation," http://www.mulliner.org/android/feed/mulliner_ddi_30c3.pdf.
[66] V. Costamagna and C. Zheng, "Artdroid: A virtual-method hooking framework on android art runtime," in *The Workshop on Innovations in Mobile Privacy and Security (IMPS) at ESSoS*, 2016.
[67] M. Sullivan, "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.
[68] D. Murphy, "Average Internet Speeds Up, But U.S. Still Has Work to Do," http://www.pcmag.com/article2/0,2817,2496861,00.asp.
[69] MetaCtrl, "Android app: Autosync OneDrive - OneSync," https://play.google.com/store/apps/details?id=com.ttxapps.onesyncv2.
[70] Microsoft Corporation, "Official OneDrive Android app," https://play.google.com/store/apps/details?id=com.microsoft.skydrive.
[71] MonSoon Solutions Inc., "Monsoon Power Monitor," https://www.msoon.com/LabEquipment/PowerMonitor/.
[72] 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.
[73] 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.
[74] 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.
[75] S. Deng and H. Balakrishnan, "Traffic-aware techniques to reduce 3g/lte wireless energy consumption," in *ACM CoNEXT*, 2012.
[76] 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.
[77] 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.
[78] H. Tang, F. Liu, G. Shen, Y. Jin, and C. Guo, "Unidrive: Synergize multiple consumer cloud storage services," in *ACM Middleware*, 2015.
[79] Y. Zhang, C. Tan, and L. Qun, "Cachekeeper: a system-wide web caching service for smartphones," in *ACM UbiComp*, 2013.
[80] 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.
[81] D. Meister and A. Brinkmann, "Multi-level comparison of data deduplication in a backup scenario," in *ACM SYSTOR*, 2009.