UniDrive: Synergize Multiple Consumer Cloud Storage Services

Haowen Tang[1], Fangming Liu[1]*, Guobin Shen[2], Yuchen Jin[1], Chuanxiong Guo[2]
[1]Huazhong University of Science and Technology
[2]Microsoft Research Asia
[1]{cstang02, fangminghk, dianjinyuchen}@gmail.com
[2]{jacky.shen, chguo}@microsoft.com

ABSTRACT

Consumer cloud storage (CCS) services have become popular among users for storing and synchronizing files via apps installed on their devices. A single CCS, however, has intrinsic limitations on networking performance, service reliability, and data security. To overcome these limitations, we present UniDrive, a CCS app that synergizes multiple CCSs (multi-cloud) by using only few simple public RESTful Web APIs. UniDrive follows a *server-less, client-centric* design, in which synchronization logic is purely implemented at client devices and all communication is conveyed through file upload and download operations. Strong consistency of the metadata is guaranteed via a quorum-based distributed mutual-exclusive lock mechanism. UniDrive improves reliability and security by judiciously distributing erasure coded files across multiple CCSs. To boost networking performance, UniDrive leverages all available clouds to maximize parallel transfer opportunities, but the key insight behind is the concept of *data block over-provisioning* and *dynamic scheduling*. This suite of techniques masks the diversified and varying network conditions of the underlying clouds, and exploits more the faster clouds via a simple yet effective in-channel probing scheme. Extensive experimental results on the global Amazon EC2 platform and a real-world trial by 272 users confirmed significantly superior and consistent sync performance of UniDrive over any single CCS.

Categories and Subject Descriptors

D.4.8 [**Performance**]: Measurements; C.2.4 [**Distributed Systems**]: Distributed applications

General Terms

Measurement, Design, Performance

*The Corresponding Author is Fangming Liu, from Services Computing Technology and System Lab, Cluster and Grid Computing Lab in the School of Computer Science and Technology, Huazhong University of Science and Technology.

Middleware '15, December 07-11, 2015, Vancouver, BC, Canada
 ISBN 978-1-4503-3618-5/15/12 ...$15.00
DOI: http://dx.doi.org/10.1145/2814576.2814729.

Keywords

Consumer Cloud Storage, Erasure Codes, Synchronization, Networking Performance

1. INTRODUCTION

Consumer cloud storage (CCS) services like Dropbox and Google Drive provide a convenient way for users to store and sync files across their multiple devices. With CCS apps installed on user's devices, local file updates are monitored and automatically synchronized with the cloud service which propagates the updates to all user's devices. This is considered to be a great advance compared to conventional file hosting services like RapidShare and Megaupload (shut down in 2012) with storage only. As a result, we have witnessed a rapid adoption of this new generation of cloud storage services in recent years. For example, Dropbox has gained 300 million users by May 2014 [47]. An increasing number of companies (e.g., Microsoft, Google, Baidu), are also speeding up their CCS offerings.

Despite the conveniences provided by the CCS, our online user survey (with 594 participants) and active measurements (over 13 globally distributed PlanetLab nodes) reveal multiple user concerns and performance issues for any single CCS.

Performance. Our measurement results show that the upload and download performance of a CCS vary drastically in both spatial and temporal dimensions due to fluctuating and dynamic network conditions. For instance, as we measured in Dec. 2013, the average time Dropbox takes to upload an 8 MB file in Los Angeles can be up to 2.76 times that in Princeton; and even at the same location (e.g., Princeton), the difference between the max and min upload duration within the same day can be as large as 17 times. Our user survey results also reveal that users have common concerns (reported by 69.62% of survey participants) about the sync performance and the unpredictability of sync time for the CCSs they use.

Reliability. A service may experience temporal outages [48] or spatial outages (i.e., it is available in some regions but not in others). For example, Dropbox and Google Drive are blocked by the Great Firewall in China. Users who live in or travel to such a region will suffer from spatial outages.

Security and Vendor Lock-in. CCSs have full access to user data. Thus, user data is at risk when the CCS is attacked, or when authorities require the CCS provider to expose their data. When users invest heavily in placing their data into a single CCS, they become locked-in to that particular CCS. The more data they store, the more difficult it is to switch to another CCS, even if more favorable CCSs are available.

There are works [5, 9] that help enterprises to move their storage from single to multi-cloud to get better reliability, higher security, and avoid vendor lock-in. This is achieved with a proxy system

that stripes data with redundancy to multiple enterprise cloud storage (ECS) services (such as Amazon S3 and Windows Azure Storage), while providing the same GET/PUT interfaces as the underlying clouds. Since multiple CCSs are readily available (Dropbox, OneDrive, and Google Drive, etc.) and over 70% of CCS users already own multiple CCS accounts according to our survey, we wonder if such multi-cloud approach is also viable in the field of consumer cloud storage. However, CCS is very different from ECS for the provision of automatic multi-device synchronization rather than simple GET/PUT interfaces, which requires far more complicated coordination between client apps and cloud services. Further, existing CCSs do not provide a public execution environment, nor can they directly communicate with each other. Unlike their native client apps that use private APIs for advanced control and efficient data transfer, third-party applications can use only few simple public RESTful Web APIs sufficient for simple file access to existing user accounts. Therefore, the first question we ask is: *can we realize a CCS app in the multi-cloud while avoiding the reliability, security, and vendor lock-in issues of a single CCS?*

Existing multi-cloud solutions cannot well address the performance issue, their networking performance is degraded by the slower clouds, thus they can only achieve a medium level of performance compared to the underlying clouds (i.e., unable to outperform the fastest cloud) [9]. However, our measurement result shows that the network bandwidth of different CCSs is diverse, highly fluctuating, and difficult to predict in both spatial and temporal dimensions, and the performance disparity among CCSs (up to 60 times for average upload speed of different clouds) is much worse than that among ECSs [25], thus traditional multi-cloud approaches are deemed to suffer severe performance degradation for CCS. This challenge leads to our second question: *can we overcome the limitation of multi-cloud and achieve superior networking performance?*

We provide affirmative answers through the design and implementation of UniDrive, a CCS app that synergizes multiple CCSs into a multi-cloud with better sync performance, reliability, and security. We follow a *server-less, client-centric* system design to address the first challenge. UniDrive introduces no additional server, and assumes only the minimum set of public data access Web APIs from CCS providers. All message exchanges are performed via file upload and download between client and the clouds. In UniDrive, files are used in three different ways: metadata carrier, synchronization signal, and normal data file. To realize efficient file synchronization in a distributed way, we store all the metadata — the file hierarchy image, the mapping between local files and their counterparts in cloud, etc. — into a file. With the metadata, the exact sync folder structure and pointers to actual data content in the multi-cloud can always be restored. The metadata file is replicated to all clouds and is sync'ed to all clients. Strong consistency of the metadata file (hence all content data) is achieved through a quorum-based distributed locking protocol, in which empty flag files are used to implement a mutual exclusive lock.

To address the second challenge, we employ content-based segmentation [33] to divide files into segments that are further divided into blocks and apply erasure coding [27,36]. This not only enables fine-grained control of block distribution into multiple CCSs for better reliability and security, but also limits the impact of file editing and hence reduces network traffic in the synchronization. To overcome the performance limitation in traditional multi-cloud approach and boost networking performance, UniDrive always uses all available clouds and maximizes parallel transfer opportunities. This is achieved via *data block over-provisioning* and *dynamic scheduling* techniques that mask the diversified and varying network conditions of the underlying clouds. More data blocks are scheduled to faster clouds so that the network utilization of a cloud is in proportion to its networking performance. With a simple yet effective in-channel probing scheme that determines the performance of clouds in a timely manner, UniDrive is able to benefits more from the faster clouds while avoiding being degraded by slower clouds.

We have implemented UniDrive as an app on the Windows platform [3], and thoroughly evaluated its performance (upload and download speed, sync time) and system overhead. We deployed UniDrive on Amazon EC2 instances distributed in 7 different data centers in 6 countries across 5 continents, and constructed a multi-cloud using five popular CCSs (Dropbox, OneDrive, Google Drive, BaiduPCS and DBank from Huawei). We compare UniDrive against their native apps and two baseline multi-cloud designs. Experimental results show that UniDrive achieves superior and stable sync performance, as well as enhanced reliability and security, across all locations: the average speed improvement is about 2.64× for upload, 1.49× for download, and 1.33× for synchronization over the best of the five CCSs. It incurs about 1% sync overhead, which is similar to other CCSs. We also studied the performance of UniDrive "in the wild". The real-world usage data from 272 pilot users worldwide confirmed that UniDrive delivered fast and consistent access experiences all the time.

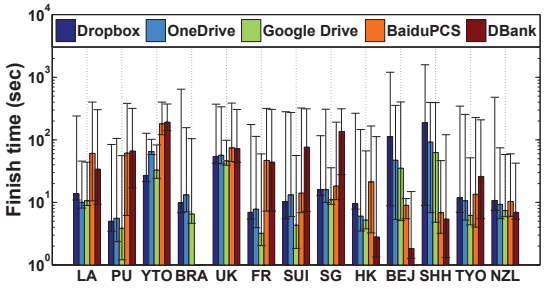
UniDrive uses the available quotas of a user's multiple CCS accounts, free or paid, at no extra cost as CCSs do not charge for bandwidth (even though the storage quota may be charged). Further, our design leads to more effective use of existing quotas. For instance, assuming a user has 100 GB on three vendors, respectively, under the requirement of tolerating unavailability of one vendor, UniDrive provides 200 GB of storage space while a conventional replication-based scheme would provide at most 150 GB.

This paper makes two key contributions. First, it demonstrates that it is possible yet non-trivial to use simple RESTful Web APIs to build a more reliable and secure CCS app in the multi-cloud. Second, it is the first to address the performance issue in multi-cloud and boost networking performance via the proposed data block over-provisioning and dynamic scheduling techniques.

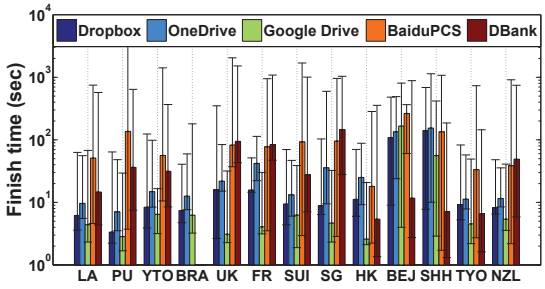
2. RELATED WORK

Many storage systems have been built using public cloud services. For example, [15,30] exploited rented virtual machines (VMs), while backup [44], file [45] and database [11] systems have been built using public cloud storage services (e.g., Amazon S3 or Windows Azure Storage). Some systems improve integrity and security via auditing and encryption [31,39,42]. However, these systems are within the domain of a single cloud and suffer from reliability and vendor lock-in issues. UniDrive instead focuses on the use of multiple CCSs to overcome the issues associated with using a single cloud provider.

We are not the first to propose combining multiple cloud services. Recent work has proposed to improve data availability and security by striping data across multiple service providers with redundancy [10, 22, 23]. Other systems further optimize striping to minimize monetary cost in terms of bandwidth and storage [35], switching provider [5], or repairing lost data [21]. By using the multi-cloud, DepSky [9] and ICStore [12] share a similar goal of building a dependable client-centric storage system, and MetaSync [18] realizes secure and reliable file synchronization. However, Depsky requires global clock synchronization among clients and incurs an extra lock/release process in each write operation, while MetaSync relies on a specific API called append-only list to implement the synchronization barrier. Basescu et al. [7] address this limitation by proposing a wait-free algorithm which supports concurrent multi-writer. In contrast, UniDrive supports concurrent



(a) Upload



(b) Download

Figure 1: Average upload/download time for an 8 MB file, to/from different CCSs on 13 PlanetLab nodes.

updates to the same file without explicit coordination using immutable data blocks and conflict detection, and provides efficient multi-device file synchronization with basic file access APIs from CCSs. Further, UniDrive improves the networking performance by its data block over-provisioning and dynamic scheduling techniques, which we believe are UniDrive's novel and unique contributions.

Previous work on distributed file systems (DFSs) [38,43,50] provides many inspiring techniques to the UniDrive design. Examples include content-based segmentation [33] to minimize update traffic, and splitting metadata [40] to reduce sync overhead. While many DFSs [6, 24, 34] share a similar server-less client-centric design, UniDrive differs from them in that the storage clouds it incorporates are 'non-cooperative' — they cannot communicate with each other, nor do they provide an execution environment such as a VM. UniDrive also does not assume direct communication between clients. In contrast, DFSs usually assume autonomous participating nodes that can execute programs customized for the service [16, 49].

Besides building a Dropbox-like elastic file synchronization service [29] involving both cloud servers and desktop clients, there are also works based on the native client app of existing CCS to reduce its network traffic via bundling frequent short data updates [26], or to prevent data corruption and inconsistency with customized local file system [51]. Instead of developing proprietary cloud services or enhancing existing native CCS client apps [28], UniDrive utilizes public Web APIs to synergize multiple CCSs, and provides higher reliability, security, and better networking performance in the multi-cloud.

3. UNDERSTANDING CCS

As a motivation to this work, we conducted an online user survey regarding the usage of consumer cloud storage services in Dec. 2013. We also performed in-depth measurement study on the performance of five representative CCSs. Unlike previous measurement work on the performance [13, 20], architecture design [46], application protocol [14], and security issues [32] using native CCS apps, we have focused on the network characteristics of their *Web APIs* that are essential for synergizing multiple cloud storage services, as a third-party application.

3.1 User Perceptions: A Survey Study

A total number of 594 valid questionnaires are collected worldwide (mainly China and U.S.), with 68.35% from students and professors in colleges, and the rest from IT and information workers. The questionnaires consist of multiple-choice questions[1]. We summarize major findings below.

Basic Statistics: First of all, CCS indeed sees great penetration. Around 80% of all participants (474 users) use CCSs, and over 70% of them (347, with reference to all the 474 CCS users, same hereafter) own multiple CCS accounts, mainly propelled by the sharing needs. Second, the main criterion that users choose or switch to a particular CCS is that it is free (63.08% of users), followed by large storage space (42.41%) and fast upload/download speed (33.97%). Third, the top three functions of CCSs are file backup (used by 86.71% of users), file sharing (47.26%) and multi-device synchronization (44.3%).

Major Concerns: The top three concerns are the slow upload/download speed (69.62%), limitation on the file size and quota (41.56%), service unavailability (31.43%), respectively. Somewhat surprisingly, the top three reasons if a user would pay for a cloud storage service are higher security (58.08%), better performance (54.13%), and more storage space (33%). Significant portion (60.55%) of surveyors expressed the concern of vendor lock-in threat when asked what if ultra-large space (e.g., 1 TB) is provided for free.

3.2 Performance: A Measurement Study

Measurement Methodology: We selected top five representative CCSs worldwide, three from U.S. (Dropbox, OneDrive, and Google Drive) and two from China (BaiduPCS and DBank from Huawei). They are all among the most popular CCSs and provide open RESTful Web APIs. We implemented a client software that is able to upload/download files to/from all these five CCSs using their own Web APIs. To gain a comprehensive view of their performance over time and across locations, we periodically (every half an hour) measured their performance for over one month from 13 geographically distributed PlanetLab nodes in 10 countries across 5 continents. In each experiment, we upload or download a file with known size to or from the five CCSs back to back to ensure fair network conditions. We repeat the experiments using different file sizes (0/0.5/1/2/4/8 MB). Although there may exist background traffic or bandwidth restrictions on PlanetLab, we believe the results still statistically reflect the performance variations from end users' viewpoint, since the performance varies dramatically from every node to all CCSs. Our evaluation on Amazon EC2 in Section 7.2 confirms our observation.

Networking Performance – Spatial Dimension: Figure 1 shows the average (over one month), min and max time (plotted in log scale) for uploading or downloading an 8 MB file to the five CCSs from geographically distributed PlanetLab nodes. Similar observations can be made for other file sizes, and are omitted for sake of space. From the figure, we observe that, firstly, the upload and download performance of any CCS vary significantly across different locations. For instance, Dropbox takes 2.76 times longer

[1]http://www.sojump.com/jq/3036565.aspx

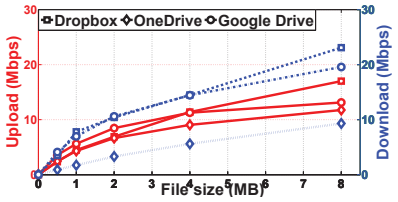


Figure 2: Impact of different file sizes on the throughput, on Princeton node.

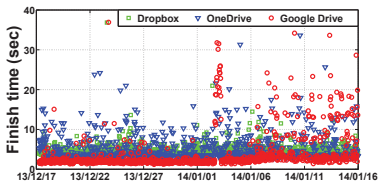


Figure 3: Daily upload time for 8 MB file over a month, on Princeton node.

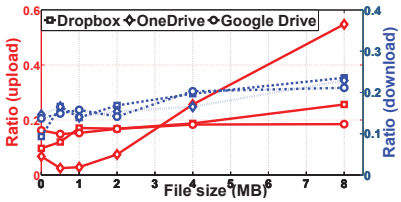


Figure 4: Impact of different file sizes on the failure rate, on Princeton node.

to upload an 8 MB file on Los Angeles node than on Princeton node. Secondly, there is *no* always winner across different locations. Some CCSs outperform others at certain locations but underperform at others. For example, the upload time (8 MB file) of Dropbox is only half that of OneDrive on Princeton node, whereas their roles reverse on Beijing node. Thirdly, the upload and download performance are weakly correlated. The correlation between the two subfigures is about 0.41. That is, clouds with fast upload speed are likely to have fast download speed as well. Finally, higher throughput is achieved as file size increases for both upload and download, as shown in Figure 2. However, the increase tends to diminish when the file size is larger than 4 MB.

Cloud data center locations are of great importance to consumer cloud storage services, involving both policy and performance implications. In our measurement, we observed a total of 554 unique IP addresses from pinging service names of different clouds. We found that Dropbox hosts its Web APIs on two Amazon data centers in USA, while BaiduPCS and DBank utilize geo-distributed servers for their API services. In contrast, OneDrive and Google Drive use globally distributed data centers and Edge POPs [17], respectively, to serve API requests near end users. Besides different API protocol designs, their different data center locations and serve strategies result in the performance disparity of different CCSs at different locations.

Networking Performance – Temporal Dimension: Figure 3 shows the upload time of the three U.S. CCSs (we omit the similar results for the other two China CCSs for clarity) for an 8 MB file on Princeton node in one month. We can easily observe that the fluctuation is indeed high without predictable pattern (e.g., the difference between the max and min upload time of Dropbox within the same day can be up to 17 times). But one interesting finding is that the performance variations of different CCSs are largely *independent* with no obvious pattern along the time. These results indicate that the temporal variation in performance is not resulted from the last mile, but caused by network fluctuations across Internet paths and clouds.

Service Availability: In addition to service outages [48], there exist frequent transient accessibility issues: not every Web API request is always successful. We collected the statistics of success rate of all Web API requests in our experiments. We found that when accessing U.S. CCSs from PlanetLab nodes in U.S. or Canada, the success rate is around 99%, whereas the rate drops to around 90% from nodes in China. Accessing BaiduPCS from all but one nodes typically sees about 95% success rate, whereas that of accessing DBank sees much larger fluctuation. There is no report of severe service outages during the period of our experiments, thus the access failures are likely caused by transient network or server failures.

We studied the transient failure behavior of the three U.S. CCSs. We found that: 1) their failures have *negative* correlation, as shown in Table 1. This means that different CCSs rarely experience outages at the same time; statistically, it is expected that while some clouds are experiencing issues, others might be working normally. 2) Larger files are more likely to experience transmission failures, as shown in Figure 4, which plots the percentage of different file sizes among all the failed cases. We can see that when the file size is less than 2 MB, there is no obvious increase in failure rate.

Table 1: Correlation between failed Web API requests among three U.S. CCSs. Italic numbers correspond to the download case

Upload/*Download*	Dropbox	OneDrive	Google Drive
Dropbox	–	*-0.5064*	*-0.4601*
OneDrive	-0.1161	–	*-0.5326*
Google Drive	-0.9714	-0.123	–

4. UniDrive: DESIGN OVERVIEW

The user survey and measurement results indicate that using a single CCS has potential issues in performance, reliability, security, and vendor lock-in. In this paper, we hope to overcome these intrinsic problems of single CCS through the design of UniDrive that synergizes multiple CCSs (multi-cloud for short) with a suite of techniques to boost networking performance.

Challenges: To realize efficient file synchronization in multi-cloud, there are two major challenges. Firstly, unlike previous systems that rely on running processes on server [4, 30] or communication between servers [8, 19], existing CCSs do not provide any execution environment, nor direct communication capabilities to other CCSs. Worse even, different from a native CCS app that may use private, possibly stateful, APIs to run complex control logic and efficient file transfer, we can use only the few public RESTful (i.e., stateless) APIs (i.e., file `upload`, `download`; directory `create`, `list`; and `delete`) that are solely for data access. Such constraints make it a challenging task to effectively sync files across devices and multiple CCSs and to ensure consistency among them. Secondly, network conditions and accessibilities to different CCSs are fluctuating across locations and along time, and are hard to predict. The presence of slow clouds can severely degrade the networking performance of the multi-cloud systems, making them unable to outperform the underlying fastest cloud [9]. Thus, how to accommodate such diversities and achieve superior performance than any single cloud becomes another challenge.

Solution Idea: Given aforementioned challenges, our basic solution idea is to go after a *server-less, client-centric* design that distributes the coordination tasks to users' devices, and rely on the basic file upload/download operation to convey messages for locking and notification in multi-cloud multi-device synchronization. No additional server beyond those servers that are part of the cloud storage provider system are involved. To boost performance, we may stripe user data into smaller blocks, perform erasure coding to add redundancy, then carefully schedule the distribution of these blocks to the multi-cloud to fulfill reliability and security require-

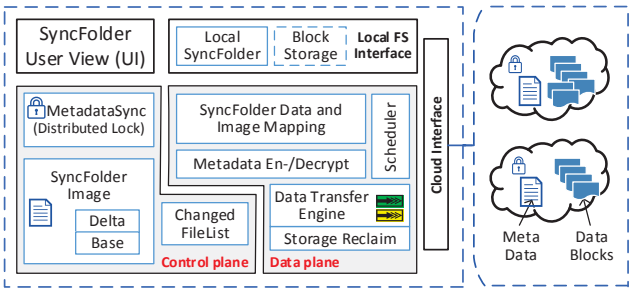


Figure 5: Overall architecture of UniDrive.

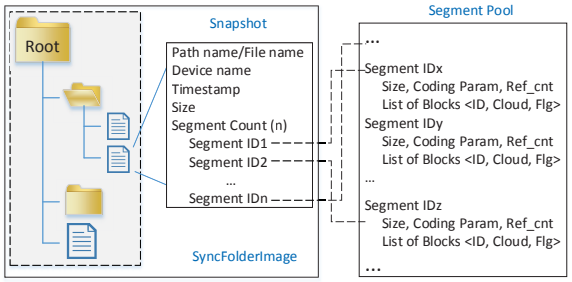


Figure 6: Illustration of SyncFolderImage content structure. The hierarchy in the shaded box reflects the actual file hierarchy of the local sync folder in user's view.

ments, e.g. being able to recover user data during outages of a fraction of clouds, and ensuring no single provider can have user data in integrity. Networking performance can be improved through parallel transferring to multi-cloud and exploring faster clouds.

Architectural Overview: Figure 5 depicts the overall architecture of UniDrive. As typical CCS apps do, UniDrive employs a local sync folder and exposes normal folder view to the user so that the user will not be aware of the complexity of managing multiple clouds.

Under such *server-lesss, client-centric* architecture, center to UniDrive is the metadata design. Different from existing multi-cloud systems (such as DepSky [9] and MetaSync [18]) that maintain an individual metadata file for each file/directory entry, we instead adopt a single metadata file – SyncFolderImage that captures all the metadata: the image of file hierarchy, the mapping between local files and their counterparts in multi-cloud, etc. With SyncFolderImage, clients can recover the exact sync folder. This design eliminates the maintenance of massive tiny metadata files and thus drastically reduces the metadata overhead in the multi-cloud multi-device synchronization.[2] Metadata is DES encrypted and replicated to all clouds and is sync'ed to all clients. Comparing local and sync'ed versions of SyncFolderImage, the clients can autonomously take proper actions to reach a sync'ed state. Function-wise, UniDrive consists of the following three high-level modules:

Interfaces: UniDrive assumes the minimum set of RESTful data access Web APIs to ensure the ability of integrating any potential CCSs. The cloud interface abstracts these basic functionalities, so as to hide the disparities in the APIs of different clouds and ensures the extensibility of UniDrive. When a new cloud is to be added, the developer only needs to implement this interface. The local file system interface monitors file system changes arisen from local editing and also commits file updates (write, delete, etc.) from the cloud.

Control plane: Control plane modules are responsible for the replication of SyncFolderImage to all the clouds and clients. Strong consistency (via MetadataSync) of metadata in the multi-cloud is achieved by implementing a quorum-based distributed mutual-exclusive lock mechanism. We use an empty file to serve as the lock signal. To reduce the overhead in MetadataSync, we design a Delta-sync mechanism where a log-structured file is adopted to accumulate incremental changes to the SyncFolderImage.

Data plane: Data plane modules handle all the tasks of data transfer, including file segmentation and assembling, encryption and decryption of metadata, scheduling and actual transfer of data blocks. Reliability and security are enhanced through scheduling different numbers of data blocks to physically separated multiple clouds. Eventual consistency across devices and the multi-cloud is guaranteed through file synchronization, i.e., user files in different devices and data stored in the multi-cloud will eventually reach a consistent state. Sync speed is improved by 1) maximizing parallel transfer opportunities, which is achieved via erasure coding and the two-phase (availability-first and reliability-second) transferring strategy, and 2) exploiting more the faster clouds by over-provisioning and dynamic scheduling (i.e., transferring more parity blocks to faster clouds than its fair share) via an in-channel probing scheme.

[2]Transferring a large number of tiny files consumes much more network traffic and time than that of a single file with the same amount of data, e.g., about 1.89 MB traffic to upload 1024 files each of 100 bytes to Dropbox, while that for a single 100 KB file is only 102 KB.

5. CONTROL PLANE DESIGN

In this section, we present the data model and the synchronization protocol to achieve metadata synchronization with limited Web APIs.

5.1 Data Model

In UniDrive, there are two types of data, namely *content data* and *metadata*. The content data are the actual file content. It may be divided into multiple segments, each segment is further chunked into fixed-sized *data blocks* (or blocks for short). In UniDrive, a block is the basic unit for file transfer and also the data storage in the cloud. The block is immutable once created.

The metadata captures the status of the sync folder and all the updates occurred to it. It consists of three parts: a *SyncFolderImage*, a *segment pool* and a *ChangedFileList*. SyncFolderImage (or image for short) maintains the file system hierarchy of the local sync folder and all its files. For each file in the sync folder, an entry exists in the image, containing the *snapshots* of the corresponding file. Each snapshot summarizes all the metadata of a file, including its full path, timestamp, file size, segment count and a list of segment IDs referring to concrete segments in the segment pool, as shown in Figure 6.

In UniDrive, a segment is identified by the hash of its content and represented as a list of blocks <Block-ID, Cloud-ID>, where Block-ID is its sequence number in the scope of the segment (the actual filename of a block is thus the segment ID concatenated with its sequence number), Cloud-ID denotes on which cloud the block is stored. It mandates to upload data blocks before updating the metadata, and is set asynchronously via callback. ChangedFileList records all the changes performed in the local sync folder, such as adding, editing, or deleting files or folders, etc., since the last synchronization. These records enable UniDrive to aggregate and commit series of changes to the image at once. ChangedFileList will be cleared after each successful synchronization.

5.2 Metadata Synchronization

The separation of content data from metadata enables asynchronous update of them. In UniDrive, data blocks are uploaded freely (e.g., upon created) and multiple devices can upload their data blocks concurrently. The file consistency is ensured through the consistency of metadata. Such a design allows maximum efficiency as transferring of data blocks usually takes most of the sync time.

There exist two types of metadata updates, namely *local update* and *cloud update*. A local update is generated locally, due to file edits in local sync folder, and to be propagated to the cloud and then to other devices. It is signaled by a non-empty ChangedFileList. The cloud update is the pending update (from other devices) a device needs to sync up with. It occurs when a device committed its local update to the multi-cloud, and is signaled by a newer version of metadata in the clouds. It is periodically checked at a time interval τ. To reduce the overhead of checking cloud update, we use a small *version file* that contains only a device name and timestamp. There is no need of global clock synchronization. As long as the version file is different from the local one, a client can conclude a pending cloud update.

Algorithm 1 Metadata synchronization.

Input:
Multi-cloud: $C = \{c_1, c_2, \ldots, c_n\}$
Original metadata: v_o
ChangedFileList: ϵ
Output:
Strong consistency of metadata across devices and multi-cloud

```
1: function SyncMetadata
2:     if check_local_update(ε) then
3:         local metadata v_l ← apply_local_update(v_u, ε)
4:         acquire_lock(C)
5:         if check_cloud_update(C) then
6:             cloud metadata v_c ← download_metadata(C)
7:             updated metadata v_u ← merge(v_o, v_l, v_c)
8:             upload_metadata(v_u, C)
9:             v_o ← v_u
10:        else
11:            upload_metadata(v_l, C)
12:            v_o ← v_l
13:        end if
14:        release_lock()
15:    else if check_cloud_update(C) then
16:        cloud metadata v_c ← download_metadata(C)
17:        apply_cloud_update(v_o, v_c)
18:        v_o ← v_c
19:    end if
20: end function
```

Algorithm 1 outlines the logic of metadata sync of UniDrive. To commit a local update, a client must first acquire the lock (line 4), make the metadata up-to-date by downloading and merging with cloud update if it exists (line 6-7), and then commit the latest metadata (line 8, 11). Normally, it can sync up cloud updates lazily e.g., periodically, and merge to local metadata (line 17). Two cases need special handling, namely concurrent local updates from multiple devices and conflicting local and cloud updates. We elaborate our design as below.

Handling Concurrent Local Updates: To ensure the consistency of metadata, all attempts from different devices to commit local updates to the multi-cloud have to be serialized. This is achieved by using a mutual-exclusive lock mechanism. We have designed a *quorum-based distributed locking protocol* on top of multiple CCSs, which only involves a special *lock file* with *empty* content.

Concretely, the protocol works as follows: firstly, the attempting device d generates a lock file with name $lock\_d\_t$ (t denotes the local timestamp on d when locking) to identify itself, and uploads the lock file to a specific *lock directory* across all the clouds.[3] Secondly, d calls `list` on each cloud to list all files stored in the lock directory, and is considered to acquire the lock of a cloud if there exists only its own lock file on that cloud. To handle the potential contention (due to different network latency), quorum number of locks needs to be acquired, i.e., succeed in locking down majority of clouds. Only in this case, it can proceed to upload the metadata (after ensuring metadata is up-to-date). Attempts failed to acquire enough locks are treated as failures, then d needs to withdraw its own request by deleting its lock files from all the clouds, and executes a random backoff – waiting for a random time before trying to acquire the metadata lock again. A device releases the lock after finishing updates by deleting all its lock files from all the clouds.

The locking protocol does not requires a cloud to guarantee sequential concsistency for file operations, but only assumes the commonly provided read-after-write consistency, i.e., after uploading a file to the cloud, the file can be listed and read. When a client `list` and sees file A, then all subsequent `lists` from any clients will also contain file A. Hence, once a client acquires the lock of a cloud, all other clients will see its lock file and deem that cloud is blocked. Furthermore, a *lock-breaking* mechanism, with no requirement on global clock synchronization among clients, is devised to tolerate potential faulty clients. After acquiring the lock, the client is required to periodically refresh its lock files to prevent them from being obsolete, i.e., seen by other clients over a time threshold $\triangle T$ such as 120 seconds. Failed to do so will prevent the client from continuing the metadata update process, its lock will be revoked since other clients will check (by recording the first time it sees each lock file) and delete those obsolete lock files after calling the `list`. This will ensure that other devices will be able to make process when a device crashes while holding the lock, and that the crashed device will no longer hold the lock when it recovered after a long delay (i.e., $\triangle T$).

Conflicting Local and Cloud Updates: When both the local update and the cloud update exist, conflicts may happen. Inspired by the resolution policies in SVN and GIT, we propose a practical conflict detection and resolution mechanism. Specifically, we first record the original metadata v_o, and handle ChangedFileList to obtain the latest metadata v_l. We then download the cloud metadata v_c and try to merge v_l and v_c. We proceed to detect potential conflicts. To this end, we de-serialize v_c and perform a tree-comparison against v_o, and obtain their difference ΔC. We further obtain the difference ΔL between v_l and v_o in a similar way. We compare ΔC and ΔL. For entries without coincidental updates, we directly merge v_l and v_c to obtain up-to-date metadata v_u by applying ΔC to v_l or equivalently ΔL to v_c. For entries do have both local and cloud updates, we retain both updates in the merged metadata v_u, and prompt the user of the conflict. The user can resolve the conflict later. Note that file content data corresponding to conflict entries are also retained to facilitate future conflict resolution.

Delta-sync for Efficiency: The gross metadata file grows linearly with the number of files, and can become large when there are many files in the sync folder. To avoid repeated transferring of unchanged part of the whole metadata, we adopt the metadata design principle of HDFS [40] and divide the metadata into a *base* file and a *delta* file. The base file is the snapshot of the metadata at a specific time, and the delta file records all the updates since then with a log

[3]Using the special lock directory instead of an existing one that may contain many data files is to ensure small traffic when listing. The lock directory contains at most the same number of files as that of devices.

structure [37]: updates are always appended to the delta file. Normally, only delta is transferred to the cloud. The size of delta file grows with time. When its size reaches a threshold λ (e.g., 0.25 of the total base file size or 10 KB), it is merged with the base (by client who acquires the lock) to restore the up-to-date metadata and form a new base. The new base file will be transferred to cloud and sync'ed to other devices later on. Meanwhile, the delta file is cleared.

6. DATA PLANE DESIGN

Data plane handles data block generation, scheduling, and transferring. It seeks to improve sync performance by minimizing the traffic, maximizing parallel transferring opportunities, and exploring faster clouds.

6.1 Segment and Block Generation

Content-based File Segmentation: In UniDrive, to minimize data traffic, we adopt the *content-based segmentation* method [33] to divide a file into segment(s), which are indexed by the SHA-1 hash of all their content. This ensures that segments with same content, even from different files, will have the same file name. Thus, effective content deduplication, and hence the suppression of redundant transmissions, are enabled. Deduplication is achieved via *reference counting* [41]. Recall that we keep both versions of conflicting files who partially share the same content, thus deduplication is especially helpful in reducing the storage and network traffic. Content-based segmentation will result in segments of varying sizes. Considering the impact of file size on transfer efficiency, we constrain the size of final segments to a range, say (0.5θ, 1.5θ), where θ is a tunable parameter. This is achieved by merging small neighboring segments or splitting large segments.

Data Block Generation: Packet-based coding schemes are typically used in file transfer over the Internet. It simplifies the scheduling task especially when there are multiple paths/connections between the two transferring entities. In UniDrive, we follow these practices and generate (parity) data blocks via erasure coding [27, 36]. Note that data deduplication is performed at the segment level, and data blocks are immutable for consistency and efficient concurrent access. When a new segment is formed, new data blocks are generated; and similarly, when a segment is deleted, all its data blocks are deleted.

Security is reinforced by applying non-systematic [2] Reed-Solomon codes to generate parity data blocks, which removes their semantics and thus prevents the providers from inferring the original contents. Assume a user enrolls N clouds, and imposes the security requirement of preventing unauthorized access to original files if less than K_s clouds are breached by either malicious insiders or outside attackers (i.e., no $K_s - 1$ providers can recover the data), and the reliability requirement of tolerating up to $N - K_r$ clouds being not accessible (i.e., at least K_r clouds are accessible). Obviously, we need to have $1 \leq K_s \leq K_r \leq N$. Assume a file segment is divided into k data blocks. From reliability requirement, we need to put at least $\lceil \frac{k}{K_r} \rceil$ blocks (termed fair share afterwards) to each cloud, whereas security requirement dictates that we can put at most $\lceil \frac{k}{K_s - 1} \rceil - 1$ blocks (or k blocks if $K_s = 1$). Therefore, we should generate a minimum of $\lceil \frac{k}{K_r} \rceil N$ and a maximum of $(\lceil \frac{k}{K_s - 1} \rceil - 1)N$ data blocks. For simplicity, we adopt non-systematic RS codes with pre-fixed parameters, i.e., $(\lceil \frac{k}{K_s} \rceil N, k)$, and generate in advance $\lceil \frac{k}{K_r} \rceil N$ normal parity blocks. Additional parity blocks are called over-provisioned parity blocks. They can be generated either in advance (consuming more memory or temp storage) or on demand (incurring additional coding latency).

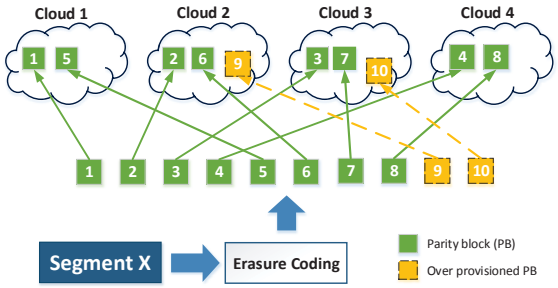


Figure 7: Scheduling for normal parity blocks, and exploration of faster clouds with over-provisioned parity blocks ($N = 4$, $k = 4$, $K_r = 2$, $K_s = 2$).

6.2 Data Block Scheduling

The scheduler determines how to distribute data blocks to the multi-cloud while fulfilling the reliability and security requirements. In UniDrive, files are uploaded to (or downloaded from) multiple clouds in parallel. A file is said to become *available* when each of its segment has k data blocks uploaded to the multi-cloud altogether; and it is said to become *reliable* when each cloud received at least its fair share. We design different strategies for files to become available or reliable.

Basic Upload Scheduling: The $\lceil \frac{k}{K_r} \rceil N$ normal parity data blocks are scheduled in a *deterministic* way to achieve the reliability goal. As multi-cloud has fluctuating network conditions as well as uncorrelated access failures, we simply schedule them *evenly* to all the available clouds. This ensures other devices to have more sources to download from in parallel. However, as clouds may have diversified and varying networking performance, this even assignment strategy is likely to end up with uneven completion time and render idle those faster clouds, while waiting for slower clouds to complete.

Over-Provisioning: To mask the performance disparity of the underlying clouds and better leverage faster clouds, we temporarily *over-provision* parity blocks, by continuing to send extra parity blocks to faster clouds even if they have received their fair share. Clearly, these over-provisioned parity blocks maximally leverage the parallel uploading opportunities and make the network utilization of a cloud in proportion to its performance. This not only ensures quicker availability of files, but also helps to accelerate the downloading as they are likely to be faster clouds in the download time and they have more blocks to supply. Figure 7 depicts the process where the normal and over-provisioned parity blocks are in green and yellow colors, respectively. In the figure, the slower Cloud 1 and 4 received only the fair share of 2 blocks while the faster Cloud 2 and 3 received additional over-provisioned parity blocks besides their fair share.

Over-provisioned parity blocks are *not* scheduled in advance. Rather, they are assigned on the fly to those clouds finished transferring their fair share. The over-provisioning process will stop when the slowest cloud finishes uploading its fair share or when the maximally allowed blocks (due to security requirement) are transferred. Since a faster cloud becomes "unavailable" for upload when its storage quota is used up, over-provisioned parity blocks will be cleaned to reclaim storage space when the corresponding file is sync'ed to all devices.

Dynamic Scheduling for Batch Uploading: When performing batch uploading, data blocks are scheduled following a two-phase – *availability-first, reliability-second* – principle. That is, when

a file becomes available (to the multi-cloud), all networking resources are immediately assigned to the next file. Only when all files become available, we then start to transfer blocks to fulfill the reliability requirement to those clouds that have not received their fair shares. Note that over-provisioned parity blocks may be used, as described above. Whenever a block is successfully uploaded or downloaded, it notifies the scheduler via callbacks (to update the Cloud-ID field in the metadata). The scheduler hence tracks the progress of each file transfer. Evidently, this principle ensures all files become available in the shortest time. It weighs more the availability of all files over the reliability of a portion of files. The rationale lies in the fact that people often want to get files uploaded or sync'ed as soon as possible.

Dynamic Scheduling for Download: Only k data blocks are needed to recover an original file segment, no matter they are normal or over-provisioned parity blocks, from whichever clouds. Thus, the download scheduling is simple: eligible clouds (i.e., those have data to supply, as indicated by the Cloud-ID field) are kept sorted according to their connection speed; and then request of the next block is always assigned to the idle connection of the fastest clouds. This not only considers the transient performance of each cloud, but also benefits from over-provisioning as faster clouds are likely to contribute more blocks for downloading besides their fair share. When k blocks are downloaded, all networking resources are assigned to the next file.

In-Channel Bandwidth Probing: Essentially, we intend to put the next uploading or downloading request to the fastest available cloud. Instead of betting on predicting which clouds are faster or via explicit probing that would incur overheads and latency, we employ a simple yet effective in-channel bandwidth probing scheme that uses the last transmission as probes. Concretely, we monitor the throughput of all the currently transferred data blocks to each cloud, compute the average *per-connection* throughput of all clouds and sort them accordingly. We use average per-connection throughput instead of average throughput due to multiple concurrent HTTP(s) connections to the same cloud and the scheduling is done on a per-block basis.

Adding or Removing CCSs: A user may add or remove CCSs. This can be effectively handled in UniDrive as the client has a full copy of all the files. It only needs to rebalance the content distribution to fulfil the reliability and security requirement (no more than allowed blocks will ever be stored). In particular, to remove a CCS, we only need to redistribute its fair share (i.e., normal parity data blocks identifiable from the metadata) to other available CCSs. To add a CCS, its fair share is first calculated and then uploaded. Other CCSs' fair share are updated simply by deleting some data blocks.

7. EXPERIMENTS AND REAL-WORLD TRIALS

Implementation: We have implemented UniDrive as an app on Windows platform with over 40,000 lines of C# codes (over 7,000 lines for the core logic). Its most recent release is available at `https://code.google.com/p/unidrive-hust/`. The system is decoupled into three layers: the cloud interface layer, the core layer and the local interface layer. At the back-end, the cloud interface layer hides the complexity of managing the underlying diverse CCSs by abstracting each CCS as a *storage cloud object* with five basic file access interfaces (i.e., upload, download, create, list and delete). The core layer implements all the control plane and data plane logic and carries out file synchronization. It realizes the sync logic using priority queuing (a high priority and a low priority queue for each cloud) and uses multi-threaded file transfer to each cloud to make good use of the available bandwidth. The local interface layer not only monitors changes in sync folders via local file system interface, but also provides a unified GUI for multi-cloud configuration and management.

Evaluation Overview: In UniDrive, data security requirement is guaranteed by storing no more than allowed data blocks on each cloud and is always met. Therefore, we focus our experimental evaluation on the networking performance, overhead and reliability. The same set of five commercial CCSs as in our measurement study (Section 3.2) are employed. Their native apps are deployed on 7 geographically distributed Amazon EC2 instances covering 6 countries (U.S., Brazil, Ireland, Singapore, Japan and Australia) across 5 continents. Each instance is running Windows Server 2012 with a single "virtual CPU" and 1.7 GB memory, which is sufficient for hosting these client applications. AWS is used instead of PlanetLab because most official native CCS apps do not support Linux platform. Since standard benchmark tool like SPEC SFS 2014 [1] for measuring file server throughput and response time are not quite suitable for CCS, we draw lessons from previous CCS measurement works [13, 14] and devise our own evaluation methodology to comprehensively compare different performance metrics of these CCS apps. Note that, without access to native APIs of those CCSs, we infer their performance by capturing their network traffic (using Wireshark) and analyzing the traces. We also measure UniDrive's real-world performance from traces of 272 pilot users worldwide.

7.1 Experimental Setup

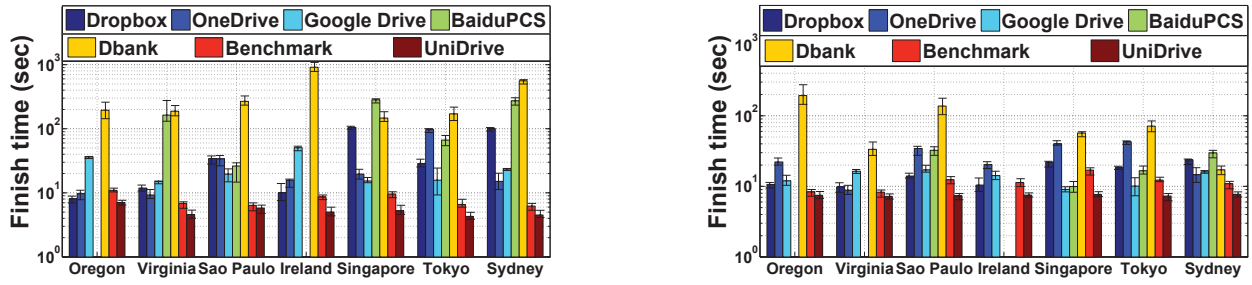
Approaches in Comparison: In addition to official native apps of commercial CCSs, we also compare UniDrive against an *intuitive multi-cloud* solution where a file is chunked into blocks and uniformly distributed to the local sync folders of CCSs' native apps (i.e., using their own proprietary sync logic), and a *multi-cloud benchmark* which resembles traditional multi-cloud solutions like RACS and DepSky, i.e., without over-provisioning and dynamic scheduling but still uniformly distributes parity data blocks to the multi-cloud with reliability and security features. This is to verify the effectiveness of the performance enhancing techniques in UniDrive.

System Metrics: The major performance metrics we are interested in are the upload/download time and also end-to-end sync time. We focus on the *available time*, i.e., the time it takes for a file to become available to the user. Another metric to examine is the *system overhead* of UniDrive, which is defined as the ratio of additional network traffic to the actual sync'ed data size.

Parameter Configurations: Given that we have 5 clouds, we set $K_r = 3$ and $K_s = 2$ for moderate reliability and security requirement. We note that the performance of UniDrive will be better if smaller K_r and K_s are specified. We set segment size θ to 4 MB and k to 3 so that the final block size is around 1-2 MB, which strikes a good balance between throughput and failure rate (see Section 3.2). Different CCSs' native apps allow different numbers of concurrent connections, e.g., Dropbox allows 8 connections while OneDrive allows only 2. We use up to 5 connections to each cloud for fair comparisons.

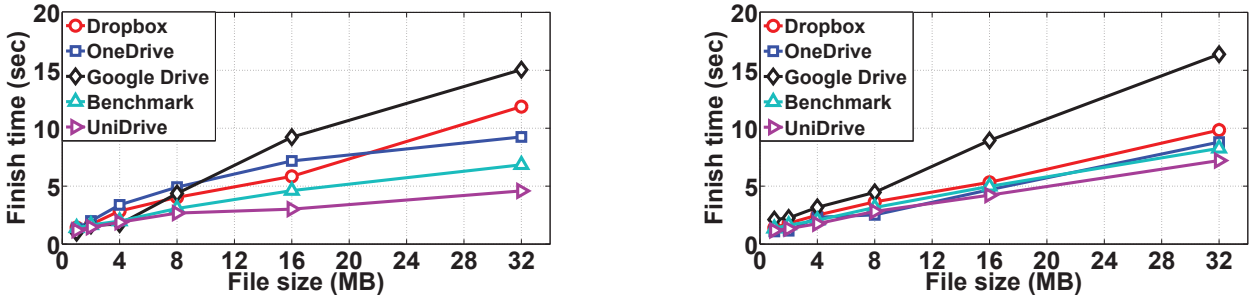
7.2 Evaluation Results

Micro-benchmark: We begin by measuring the upload and download time of files with different sizes. Considering the spatial and temporal variations in network condition, we follow the similar methodology as in Section 3.2. We focus on comparing networking performance, and only account for successful upload/download operations. While single CCS experiences availability issues during



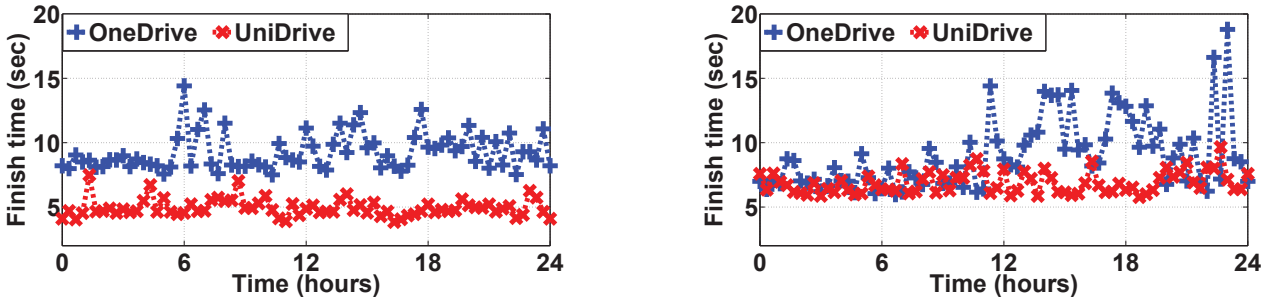
(a) Upload (b) Download

Figure 8: Average time to transfer 32 MB file to/from different CCSs.



(a) Upload (b) Download

Figure 9: Average time to transfer different sized files, on Virginia node.



(a) Upload (b) Download

Figure 10: Hourly variation in a day, transferring 32 MB file (Virginia).

our experiments[4], UniDrive always completes the operation due to its multi-cloud design.

Figure 8 shows the average time (with min/max value, plotted in log scale) each CCS app and UniDrive take to upload or download a 32 MB file (with randomly generated contents to avoid deduplication and transfer suppression of segments) on geographically distributed EC2 nodes. We see that UniDrive achieves the best performance. Compared to the respective fastest CCS at each location, UniDrive improves the upload and download time by 2.64× and 1.49×, respectively. The improvement over the multi-cloud benchmark is about 1.5×, which indicates the effectiveness of overprovisioning and dynamic scheduling. In addition, UniDrive is the most stable one: the gap between min and max time is much smaller for all locations. We notice that some CCSs' performance is very poor, e.g., the two CCSs from China are not even accessible from certain locations. In the rest, we only compare against the three U.S. CCSs.

[4]OneDrive is found to be inaccessible on Virginia node for a whole day on December 20, 2013

Results for different file sizes are plotted in Figure 9 for the EC2 node in Virginia. We see that UniDrive and even the multi-cloud benchmark outperform all native CCS apps for almost all file sizes. Same phenomenon is observed on other EC2 nodes. Figure 10 further compares the performance variation of UniDrive and OneDrive (the fastest CCS there) for repeatedly uploading/downloading a 32 MB file during a one day period on the Virginia node. We can clearly see that UniDrive achieves a higher and more stable networking performance over time, whereas OneDrive varies significantly. We notice that while UniDrive still remains the best for the download, the performance improvement is not as good as the upload. We found the reason being that the downlink bandwidth is capped to 40 Mbps on our rented VMs.

End-to-End Time for Batch Sync: We continue to measure the sync performance for batch file sync. We conduct many experiments on different file sizes and file quantities, and report the representative results on syncing a large number of small files (100 × 1 MB). Again, we have created random content to avoid segment deduplication.

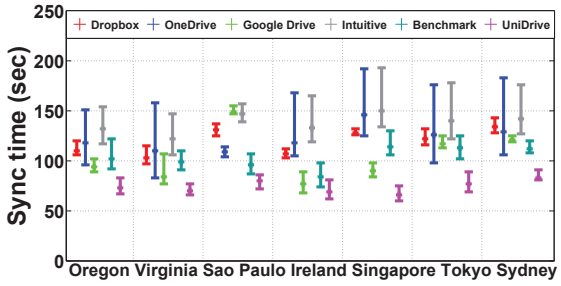


Figure 11: End-to-end sync time for batch file sync across EC2 nodes.

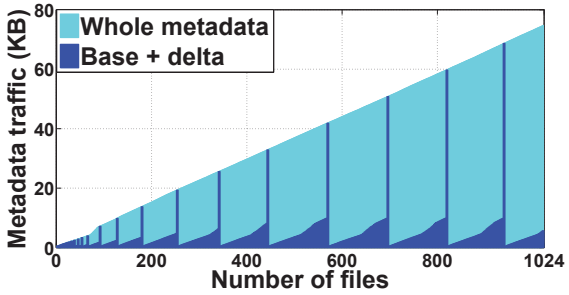


Figure 13: Metadata sync traffic.

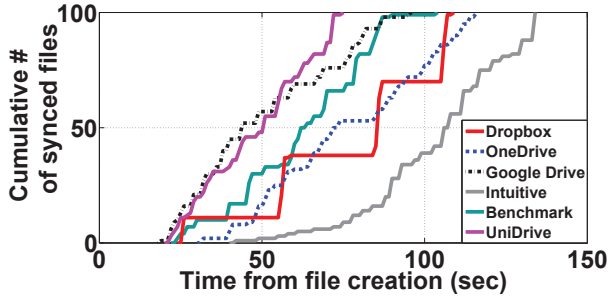


Figure 12: Cumulative plot of # of sync'ed files over time (Virginia).

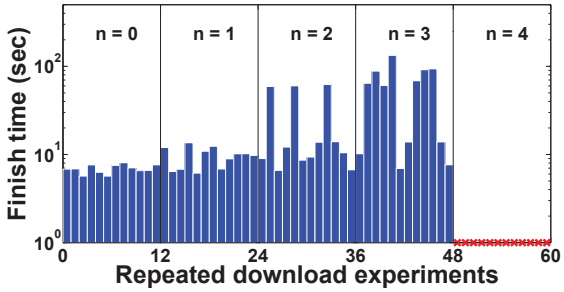


Figure 14: Availability and download performance under different numbers (n) of unavailable clouds.

Figure 11 shows the average and min/max end-to-end (from each uploading node to all the rest 6 downloading nodes) sync time for a batch of 100 $\times$ 1 MB files. From the figure, we observe: firstly, huge performance disparities can be found between different CCSs and among different clients of the same CCS, due to their different application protocol designs and network conditions. Secondly, UniDrive always achieves *best and consistent* performance across all nodes. More concretely, the average (across all locations) sync time speedups of UniDrive, compared with the top 3 fastest CCSs at each location are 1.33$\times$, 1.61$\times$ and 1.75$\times$, respectively. Thirdly, the multi-cloud benchmark achieves a medium level of performance thanks to the aggregation of multiple clouds via erasure coding and parallel file transfer; whereas the intuitive multi-cloud solution usually experiences the longest average sync time as its performance is dominated by the slowest CCS. Finally, there is a huge performance gap (1.4$\times$ on average) between UniDrive and the multi-cloud benchmark, as compared to Figure 8 and Figure 9. This confirms the effectiveness of data block over-provisioning and dynamic scheduling techniques.

Stability and File Available Rate: Recall that dynamic scheduling follows the availability-first and reliability-second principle, we take a closer look at its effect by inspecting the number of sync'ed files over time. Figure 12 plots the cumulative number of sync'ed files along the time when syncing from Oregon node to the Virginia node. We can see that UniDrive readies the files at a fast and steady speed. The curves of different solutions have varying slopes and may cross each other, indicating their varying performance during the batch sync. The almost constant slope of UniDrive curve indicates a stable performance. We further examine the variation of average sync time across different locations for all solutions. Table 2 shows the variance of average sync time in seconds over 7 EC2 nodes. We see that UniDrive is remarkably more stable (by several folds) than any single CCS.

System Overhead: The system overhead is resulted from using Web APIs (i.e., HTTP(s) connections), and the sync traffic of metadata. We compare the overhead of all solutions, using the captured traces in the batch sync experiments described above. The results are shown in Table 3. We see that UniDrive has a similar network overhead (around 1%) as that of most native apps. The intuitive solution incurs the highest network overhead as it involves all the 5 CCSs for each file sync. UniDrive also involves 5 CCSs but the overhead is suppressed with Delta-sync and the use of small version file for the cloud update check. We further conduct experiments to examine the effectiveness of Delta-sync in detail. We use UniDrive to sync a total number of 1024 $\times$ 100 KB files, one after another at every minute, and measure original metadata size at the sending Oregon node and the actual metadata traffic (after Delta-sync) at the receiving Virginia node. Figure 13 shows the results. We can see that the metadata size grows linearly with the number of updated files. With Delta-sync, the metadata traffic is drastically reduced, by a factor of 13.1 from 74.7 KB to 5.7 KB, including some sparse peaks corresponding to the transfer of new base files when the delta is merged.

Table 2: Variance of average sync time across locations

	Dropbox	OneDrive	GoogleDr.	UniDrive
Variance	134.2	140.9	558.0	33.1

Table 3: Overall sync overhead

	Dropbox	OneDrive	GoogleDr.	BaiduPCS
Overhead(%)	7.07	2.04	1.89	0.70
	DBank	Intuitive	Benchmark	UniDrive
Overhead(%)	0.96	14.93	1.01	1.04

Reliability: Complete CCS outage, though is rare, does happen from time to time. To test the reliability of UniDrive and to study the impact on the networking performance, we emulate service outage by randomly disabling n ($n \in [0, 4]$) of all the five CCSs. We pre-upload a 32 MB file to multi-cloud with the reliability requirement fulfilled. We then repeatedly download the file on the Tokyo node every 5 minutes for 12 times for a specific n. Figure 14 shows the result. As expected, UniDrive achieves the desired reliability and security, i.e., unable to restore file content when only one CCS

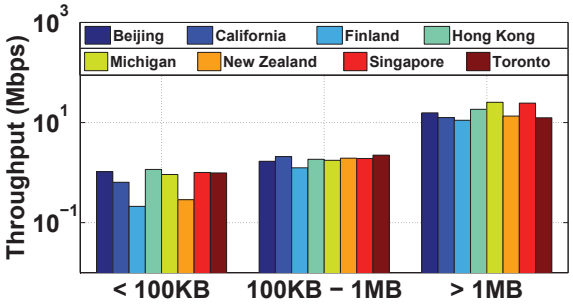


Figure 15: Average upload throughput of UniDrive at different locations.

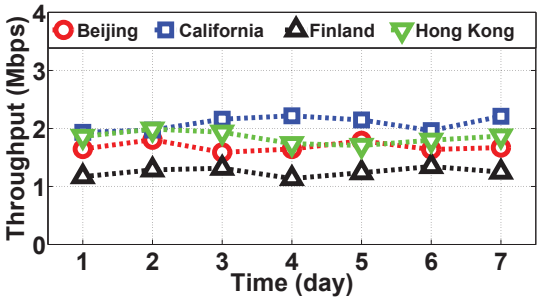


Figure 16: Daily average upload throughput of UniDrive for medium sized files over one week.

is available ($n = 4$), since K_s is set to 2. Interestingly, we found that the over-provisioning strategy may also contribute to reliability. We set $K_r = 3$, but the content can still be restored when only 2 CCSs are available (i.e., $n = 3$).[5] Figure 14 further shows the impact of the failure of CCSs on the downloading speed: with fewer CCSs available, the performance drops and is more heavily impaired by the slower CCSs.

7.3 Real World Usage

Deployment Statistics: We advertise UniDrive to CCS users with different backgrounds (thus under different network conditions, e.g., residential network, universities, companies), and collect traces from 272 users worldwide. Results from reverse IP lookup reveal that the installed devices spread over 21 sites across America, Europe, Asia and Australia. Over 500 GB data consisting of 96,982 files, of which more than a half are documents (28.3%) and multimedia contents (30.5%), are uploaded to the multi-cloud using UniDrive. The total volume of metadata traffic is reduced from 3,955 MB to 141 MB via Delta-sync.

System Performance: During the user trial, the success rate of the Web API requests from the underlying storage cloud objects is only 82.5% (much worse than the results from PlanetLab experiment in Section 3.2), while UniDrive still achieves a high reliability of 98.4% in terms of each complete file upload and download operations. Since not every user is using all the 5 clouds (some may use only 3 clouds), the 1.6% failed operations occur when API calls from more than $N-3$ clouds fail at the same time. However, all file operations will eventually complete when at least 3 clouds become available. Figure 15 shows the average network throughput (plotted in log scale) of file uploading at different geo-locations, grouped by file size range. Results for downloading are omitted since they exhibit similar phenomena. Figure 16 shows a close-up to reveal more temporal performance: it plots the daily average upload throughput for medium sized files (ranging from 100 KB to 1 MB) collected over a one week period from Sept. 14 to Sept. 20, 2014. To avoid being overly busy, we show only the performance at 4 locations. From these two figures, we observe that the throughputs at different locations are very close to each other within each given file size range, and that the performance is also stable in temporal dimension. These results confirm that UniDrive delivers *consistent access experience* across different geo-locations and over time. We also see that larger files achieve higher and more stable networking performance than smaller ones (below 100 KB) due to long (setup) latency in the underlying Web APIs, which is consistent with our experiments on EC2. UniDrive achieves an average throughput of over 10 Mbps for files above 1 MB at almost all locations.

[5]In this particular case, one CCS is much slower than the rest four. When it achieves its fair share, all other CCSs are over-provisioned.

8. CONCLUSION

In this paper, we conduct online user survey and global measurement study on five representative consumer cloud storage services, revealing their potential issues and diversified networking performance. These insights motivate the design and implementation of UniDrive, a CCS app that delivers superior networking performance in the multi-cloud, while avoiding the reliability and security risks of using a single cloud. UniDrive enhances reliability and security via careful placement of partial contents to different clouds. A suite of networking performance enhancing techniques, including block over-provisioning and dynamic scheduling with in-channel probing, are proposed to mask network fluctuations and boost networking performance by maximizing parallel transfer opportunities and exploiting more the faster clouds. Extensive experiments and real-world trials across geographically distributed locations demonstrate that UniDrive produces much improved and consistent sync performance. Although some of the individual techniques we use are not technically all that novel or surprising, the combination of the whole is interesting, deep, and non-trivial enough to make UniDrive a solid system overall. We gain the insight that, through a careful design, using a few simple public RESTful Web APIs of multiple CCSs can indeed outperform native CCS apps that use advanced private APIs, in addition to enhanced reliability and security.

9. ACKNOWLEDGEMENTS

The research was supported by grants from the National Basic Research Program (973 Program) under grant NO.2014CB347800 and NSFC under grant NO. 61520106005.

10. REFERENCES

[1] Spec sfs 2014. `https://www.spec.org/sfs2014/`.

[2] Systematic code. `http://en.wikipedia.org/wiki/Systematic_code`.

[3] Unidrive. `https://code.google.com/p/unidrive-hust/`.

[4] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable byzantine fault-tolerant services. *ACM SIGOPS Operating Systems Review*, 39(5):59–74, 2005.

[5] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon. Racs: a case for cloud storage diversity. In *SoCC*, 2010.

[6] 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 *OSDI*, 2002.

[7] C. Basescu, C. Cachin, I. Eyal, R. Haas, A. Sorniotti, M. Vukolic, and I. Zachevsky. Robust data sharing with key-value stores. In *DSN*, 2012.

[8] N. M. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. Practi replication. In *NSDI*, volume 6, pages 5–5, 2006.

[9] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. Depsky: dependable and secure storage in a cloud-of-clouds. In *EuroSys*, 2011.

[10] K. D. Bowers, A. Juels, and A. Oprea. Hail: a high-availability and integrity layer for cloud storage. In *CCS*, 2009.

[11] M. Brantner, D. Florescu, D. Graf, D. Kossmann, and T. Kraska. Building a database on s3. In *SIGMOD*, 2008.

[12] C. Cachin, R. Haas, and M. Vukolic. Dependable storage in the intercloud. *IBM Research*, 2010.

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

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

[15] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. Sporc: Group collaboration using untrusted cloud resources. In *OSDI*, 2010.

[16] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *SOSP*, 2003.

[17] Google. Peering & content delivery. `https://peering.google.com/about/delivery_ecosystem.html`.

[18] S. Han, H. Shen, T. Kim, A. Krishnamurthy, T. Anderson, and D. Wetherall. Metasync: File synchronization across multiple untrusted storage services. *University of Washington Technical Report*, 2014.

[19] J. Hendricks, S. Sinnamohideen, G. R. Ganger, and M. K. Reiter. Zzyzx: Scalable fault tolerance through byzantine locking. In *DSN*, 2010.

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

[21] Y. Hu, H. C. Chen, P. P. Lee, and Y. Tang. Nccloud: Applying network coding for the storage repair in a cloud-of-clouds. In *FAST*, 2012.

[22] J. S. P. Jason K. Resch. Aont-rs: Blending security and performance in dispersed storage systems. In *FAST*, 2011.

[23] R. Kotla, L. Alvisi, and M. Dahlin. Safestore: a durable and practical storage system. In *USENIX ATC*, 2007.

[24] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *ASPLOS*, 2000.

[25] A. Li, X. Yang, S. Kandula, and M. Zhang. CloudCmp: comparing public cloud providers. In *IMC*, 2010.

[26] Z. Li, C. Wilson, Z. Jiang, Y. Liu, B. Y. Zhao, C. Jin, Z.-L. Zhang, and Y. Dai. Efficient batched synchronization in dropbox-like cloud storage services. In *Middleware*. 2013.

[27] F. Liu, S. Shen, B. Li, B. Li, and H. Jin. Cinematic-quality vod in a p2p storage cloud: Design, implementation and measurements. *JSAC*, 31(9):214–226, 2013.

[28] F. Liu, Y. Sun, B. Li, B. Li, and X. Zhang. Fs2you: Peer-assisted semipersistent online hosting at a large scale. *TPDS*, 21(10):1442–1457, 2010.

[29] P. G. Lopez, M. Sanchez-Artigas, S. Toda, C. Cotes, and J. Lenton. Stacksync: Bringing elasticity to dropbox-like file synchronization. In *Middleware*, 2014.

[30] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. In *OSDI*, 2010.

[31] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. *TOCS*, 2011.

[32] M. Mulazzani, S. Schrittwieser, M. Leithner, M. Huber, and E. Weippl. Dark clouds on the horizon: Using cloud storage as attack vector and online slack space. In *SEC*, 2011.

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

[34] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *OSDI*, 2002.

[35] T. G. Papaioannou, N. Bonvin, and K. Aberer. Scalia: an adaptive scheme for efficient multi-cloud storage. In *SC*, 2012.

[36] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *JACM*, 1989.

[37] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *SOSP*, 2003.

[38] F. B. Schmuck and R. L. Haskin. Gpfs: A shared-disk file system for large computing clusters. In *FAST*, 2002.

[39] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket. Venus: Verification for untrusted cloud storage. In *CCSW*, 2010.

[40] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *MSST*, 2010.

[41] K. Srinivasan, T. Bisson, G. R. Goodson, and K. Voruganti. idedup: latency-aware, inline data deduplication for primary storage. In *FAST*, 2012.

[42] E. Stefanov, M. van Dijk, A. Juels, and A. Oprea. Iris: A scalable cloud file system with efficient integrity checks. In *ACSAC*, 2012.

[43] J. Stribling, Y. Sovran, I. Zhang, X. Pretzer, J. Li, M. F. Kaashoek, and R. Morris. Flexible, wide-area storage for distributed systems with wheelfs. In *NSDI*, 2009.

[44] M. Vrable, S. Savage, and G. M. Voelker. Cumulus: Filesystem backup to the cloud. In *FAST*, 2009.

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

[46] H. Wang, R. Shea, F. Wang, and J. Liu. On the impact of virtualization on dropbox-like cloud file storage/synchronization services. In *IWQoS*, 2012.

[47] T. N. Web. Dropbox reaches 300m users, adding on 100m users in just six months. `http://thenextweb.com/insider/2014/05/29/`.

[48] T. N. Web. Microsoft apologizes for three-day outlook.com outage, says caching issue was to blame. `http://thenextweb.com/microsoft/`.

[49] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *OSDI*, 2006.

[50] B. Welch, M. Unangst, Z. Abbasi, G. A. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable performance of the panasas parallel file system. In *FAST*, 2008.

[51] Y. Zhang, C. Dragga, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Viewbox: Integrating local file systems with cloud storage services. In *FAST*, 2014.