HyCloud: Tweaking Hybrid Cloud Storage Services for Cost-Efficient Filesystem Hosting

Jinlong E[†‡], Yong Cui[†*], Mingkang Ruan[§], Zhenhua Li[§], Ennan Zhai[¶]
[†] Department of Computer Science and Technology, Tsinghua University, China
[‡] School of Computer Science and Engineering, Nanyang Technological University, Singapore
[§] School of Software and KLISS MoE, Tsinghua University, China
[¶] Department of Computer Science, Yale University, USA * *Corresponding Author*
ejinlong@ntu.edu.sg, {cuiyong, lizhenhua1983}@tsinghua.edu.cn, brmk@vip.qq.com, ennan.zhai@yale.edu

***Abstract*—Today's cloud storage infrastructures typically provide two distinct types of services for hosting files: *object storage* like Amazon S3 and *filesystem storage* like Amazon EFS. The former supports simple, flat object operations with a low unit storage price, while the latter supports complex, hierarchical filesystem operations with a high unit storage price. In practice, however, a cloud storage user often desires the advantages of both—efficient filesystem operations with a low unit storage price. An intuitive approach to achieving this goal is to combine the two types of services, *e.g.,* by hosting large files in S3 and small files together with directory structures in EFS. Unfortunately, our benchmark experiments indicate that the clients' download performance for large files becomes a severe system bottleneck.**

In this paper, we attempt to address the bottleneck with little overhead by carefully tweaking the usages of S3 and EFS. This attempt is enabled by two key observations. First, since S3 and EFS have the same unit network-traffic price and the data transfer between S3 and EFS is free of charge, we can employ EFS as a relay for the clients' quickly downloading large files. Second, noticing that significant similarity exists between the files hosted at the cloud and its users, in most times we can convert large-size file downloads into small-size file synchronizations (through delta encoding and data compression). Guided by the observations, we design and implement an open-source system called HyCloud. It automatically invokes the data APIs of S3 and EFS on behalf of users, and handles the data transfer among S3, EFS and the clients. Real-world evaluations demonstrate that the unit storage price of HyCloud is close to that of S3, and the filesystem operations are executed as quickly as in EFS in most times (sometimes even more quickly than in EFS).

I. Introduction

Recent years have witnessed phenomenal successes of cloud storage in hosting data with the economies of scale. Specifically, today's cloud storage infrastructures have provided a spectrum of services exemplified by Amazon S3 (Simple Storage Service), EBS (Elastic Block Storage), EFS (Elastic File System), Glacier (Archive Storage), and so forth. As the most basic data-organization form and the most user-friendly information carrier, files are typically hosted by two types of cloud storage services with distinct design principles at the moment: *object storage* (*e.g.,* Amazon S3, OpenStack Swift and Aliyun OSS [1]) and *filesystem storage* (*e.g.,* Amazon EFS, Azure File Storage and Aliyun NAS [2]).

Object storage services have experienced the highest growth amongst the spectrum of cloud storage services, due to their simple, flat data interfaces (like PUT, GET and DELETE an object/file) and the extremely low unit storage price (*e.g.,* ∼$0.02/GB/month in S3). As a result, they have been widely used by various popular applications (*e.g.,* Dropbox, Netflix, and Airbnb). On the other hand, the simple, flat data interfaces also become a weakness when the upper-layer applications wish to support POSIX-like [3] file and directory operations (*e.g.,* MKDIR, RMDIR, MOVE, COPY and LIST). Consequently, the concerned applications (*e.g.,* Dropbox) have to maintain a separate index cloud, which incurs considerably additional costs and complexities.

More recently, filesystem storage services, an alternative type of cloud storage services, were provided to natively support complex, hierarchical filesystem operations, particularly those operations involving directory structures. Third parties can thus directly build upper-layer applications wishing to support POSIX-like operations on top of this type of services. Nevertheless, such services are found to have a much higher unit storage price than object storage services. For example, the unit storage price of Amazon EFS (∼$0.3/GB/month) is over 10× higher than that of Amazon S3.

In practice, a cloud storage user is often concerned with price and efficiency, and desires the advantages of both cloud storage services, *i.e.,* efficient filesystem operations with a low unit storage price. An intuitive approach to achieving this goal is to combine the two types of services. For example, we can host large files in S3 to achieve low storage costs, and meanwhile host small files and the metadata of all files (mainly directory structures) in EFS to achieve efficient filesystem operations. Additionally, by maintaining "link files" in EFS that refer to the large files in S3, we can easily handle those directory-related operations such as MOVE, LIST and COPY.

To examine the practical performance of the intuitive approach, we made a real-world deployment using S3, EFS, and EC2 (note that the EFS service should be accessed through an EC2 VM instance to which the EFS filesystem is mounted [4]). Our benchmark experiments show that storing and accessing small files (of several KBs to several MBs) substantially benefit from the stably high performance of EFS. Unfortunately, the clients' download performance for large files (of course from S3) often becomes a severe system bottleneck. For instance, a client with a 100-MBps Internet access bandwidth can spend up to 16 minutes in downloading

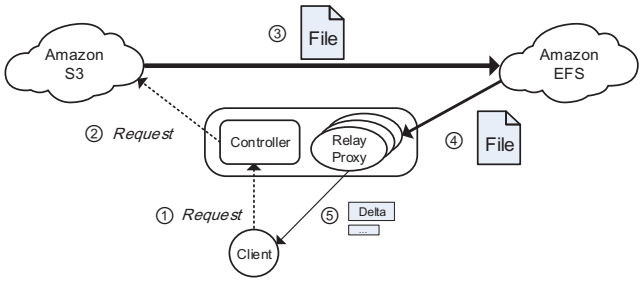

Fig. 1. Architectural overview of HyCloud, and a typical process for a client to download a large file from the hybrid cloud storage services.

a 100-MB file (hence the download speed is merely 0.1 MBps). In essence, the highly unstable performance of S3 stems from its relatively simple implementation, which cannot effectively tackle possible transfer congestions incurred by numerous concurrent data requests [5], [6]; in contrast, the mature load balance support of EFS can well cope with bursty data requests [4]. Still worse, the download bottleneck would also hold off relevant filesystem operations (*e.g.,* COPY) and thus essentially undermine the user experiences.

In this paper, we attempt to address the bottleneck with little overhead by carefully tweaking the usages of S3 and EFS to improve interoperability. This attempt is enabled by our two key observations. First, since S3 and EFS have the same unit network-traffic price for clients (∼$0.05/GB for outbound traffic and free for inbound traffic) and the data transfer between S3 and EFS within the same AWS (Amazon Web Serivces) region is not only rapid but also free of charge, we can always employ EFS as a relay for the clients' quickly downloading large files. Second, noticing that significant similarity exists between the files hosted at the cloud and its users [7], [8], [9], in most times we can convert large-size file downloads into small-size file synchronizations (through delta encoding and data compression).

Guided by the observations, we design and implement a system called HyCloud. As demonstrated in Fig. 1, HyCloud utilizes a centralized *controller* to receive all clients' filesystem operation requests, and automatically invokes the data APIs of S3 and EFS to execute the filesystem operations on behalf of the users. In order to accelerate the clients' downloading large files, HyCloud leverages EFS to forward the file content from S3 to the clients (EC2 VM instances further work as *relay proxies* between EFS and the clients given that they are necessary for executing POSIX-like filesystem operations on EFS). Moreover, whenever a user requests to download a large file (say f) from the hybrid cloud storage services, the client first checks whether there is already an old version of the file (say f') locally stored. If yes, the client will interact with an assigned relay proxy to calculate the differences between f and f' (the so-called "delta encoding"); afterwards, the relatively small-size differences are returned to the client in their compressed form for generating f. In addition, we devise filesystem operation control mechanisms to balance operation execution timeliness and system overhead, as well as eliminate redundant filesystem operation requests.

With all the above efforts, HyCloud achieves cost-efficient filesystem hosting atop S3 and EFS in a scalable manner. All the source code is publicly available at *https://github.com/iHyCloud/hycloud-demo*. Comprehensive real-world evaluations demonstrate the efficacy of our design. Under typical workloads, the overall unit storage price is quite close to that of S3 (with only a 0.43% increase). The filesystem operations are executed as quickly as in EFS in most times. For example, downloading a 100-MB file with HyCloud takes at most 15 seconds, approximately $5\times$ faster than that of S3 (when the client-side access bandwidth is not a bottleneck). When there is already an old-version file locally stored, downloading a 100-MB file costs less than 8 seconds, even exceeding the performance of EFS.

In summary, this paper makes the following contributions:

- A combinatory use of two major, heterogeneous types of cloud storage services (Amazon S3 and EFS) is implemented for whole filesystem hosting, and opportunities to address its performance bottleneck of large-file downloads are observed (§II).
- Based on the observations, several enabling mechanisms are designed to carefully tweak the usages of S3 and EFS, including relay-based downloading, adaptively-adjusted delta encoding, and filesystem operation control (§III).
- An open-source cost-efficient filesystem hosting system called HyCloud is finally implemented to embody the above enabling mechanisms. Extensive real-world evaluations demonstrate its high efficiency and low cost (§IV).

II. Motivation

To fulfill cloud storage users' desires for both low unit storage price and high filesystem operation efficiency, this section presents our first endeavor towards a hybrid architecture that makes a combinatory use of object storage and filesystem storage services in an intuitive manner. We describe the design and implementation of the intuitive approach on top of Amazon S3 and EFS (§II-A), followed by real-world measurements of filesystem hosting performance with various benchmark experiments (§II-B).

A. The Intuitive Approach

As the representatives of two distinct types of cloud storage services, Amazon S3 and EFS have highly heterogeneous pricing models for hosting files, briefly quantified in Table I. Most notably, despite the pricing disparities due to the usage amount and region, the unit storage price of Amazon EFS (∼$0.3/GB/month) is over 10 times higher than (on average nearly 15 times as) that of Amazon S3 (∼$0.02/GB/month) [10], [11], [12]. On the other hand, S3 and EFS have the same unit network-traffic price for clients[1], and the data transfer between them (in the same region) is

[1]It is necessary for clients to access EFS through EC2 VM instances, and thus the corresponding network-traffic price and request price in Table I are actually those of EC2 (as accessing EFS from EC2 is free of charge [11]).

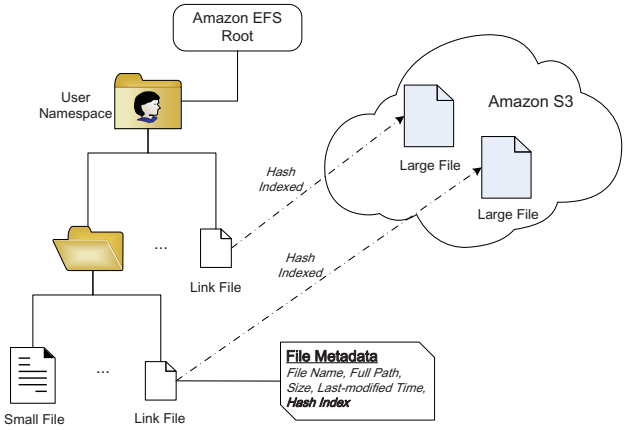

Fig. 2. The hybrid system design using the intuitive approach (large files stored in S3 are indexed by link files).

TABLE I
PRICING MODELS OF AMAZON S3 AND EFS

Cloud Service	Storage ($/GB/Month)	Network Traffic (Outbound, $/GB)	Requests ($)
Amazon S3	$0.021 \sim 0.023$	$0.05 \sim 0.09$ (Inbound free)	PUT 5×10^{-6} GET 4×10^{-7}
EFS via EC2	$0.3 \sim 0.36$	$0.05 \sim 0.09$ (Inbound free)	Free

free of charge [11], [12]. Also, we find that for a large file, the request price is almost negligible compared to the storage and network transfer price.

The above three findings directly motivate us to the intuitive approach that hosts large files in S3 to achieve an overall low unit storage price. In contrast, small files as well as directory structures are hosted in EFS to take advantage of the efficiency of EFS, given that small files have little impact on the overall unit storage price and directory structures are frequently accessed and updated. On this basis, we embody the intuitive approach into a real-world system as depicted in Fig. 2. In the cloud, a *namespace* path is allocated to each user, in which a small file is directly stored in the same path as in the user's client-side filesystem. For a large file stored in S3 (as an object), a *link file* with the same file name plus a special extension is maintained in the corresponding path of the EFS filesystem. The link file includes a series of metadata, particularly a hash index calculated based on the file content and used as the name of the corresponding S3 object.

With such implementation, all those directory-related filesystem operations (*e.g.,* MKDIR, RMDIR, LIST, MOVE, COPY and DELETE) can be easily handled with the EFS filesystem. For example, we can directly read some metadata of large files from their link files when listing files is requested. Likewise, we just need to move or copy the corresponding link file to another path for an operation to a file stored in S3. In detail, a required filesystem operation is first encapsulated in an HTTP POST/GET/PUT/DELETE request by the client, and then sent to the controller working on an EC2 VM instance.

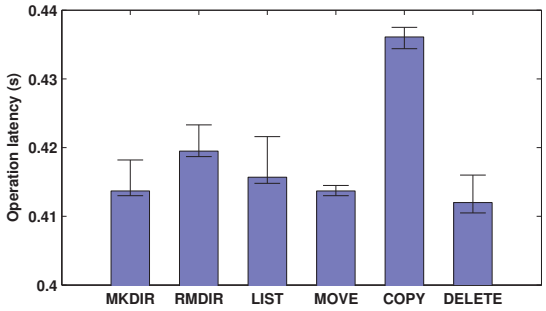

Fig. 3. Operation latencies of EFS on common directory-related operations.

On receiving the HTTP request, the controller first extracts the required filesystem operation, and then executes it in EFS using the NFS (Network File System) protocol [4].

To save storage space for redundant large files, we make multiple link files (linking to the same content) located in different paths refer to a single object in S3. At the same time, a global *link number* is marked in the object's name to easily count such link files (hence, the link number does not need to be maintained in each link file). Accordingly, we rename the object by increasing the link number for a file copy, and decreasing the link number for a file deletion.

B. Measurements and Key Observations

Although the above described intuitive approach appears to have offered a moderate balance between the working efficiency and monetary cost for filesystem hosting, its real-world performance has to be carefully examined to meet the requirements of practical usages. Besides, we need a quantitative understanding on some key system parameters, *e.g.,* the threshold between small and large files. Thus, we conduct measurements on a real-world deployment concerning the basic performance of S3 and EFS.

Specifically, by using S3, EFS and EC2 services all located in the AWS Oregon region (where all the services have the lowest price), we first measure the upload and download latencies of S3 through S3 data API requests (PUT and GET), and then measure the operation latencies of EFS through various HTTP requests (LIST, COPY, MOVE, DELETE, *etc.*). To comprehensively evaluate the performance, each kind of requests are issued for files in 6 typical sizes exponentially increasing from 1 KB to 100 MB, from three geo-distributed DigitalOcean [13] VM nodes located at Singapore, Toronto and London. Each of the DigitalOcean clients possesses a 100-MBps Internet connection (so that the client-side bandwidth would not become a bottleneck). To grasp the stability of the performance, each experiment is executed for 100 times over a whole week.

Pros and Cons. As depicted in Fig. 3, all common directory-related operations can be quickly executed in 0.44 seconds. If the network-level round trip time is excluded, a single directory operation can be finished in 0.1 seconds. Next, the

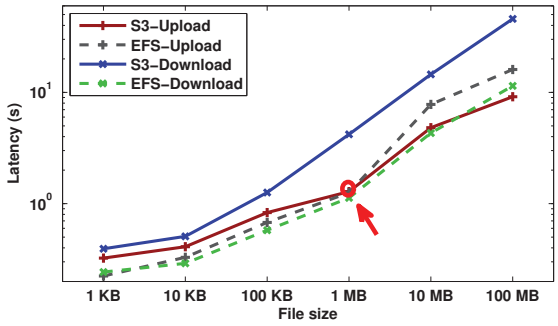

Fig. 4. Average upload/download latencies for files in different sizes to/from S3 and EFS.

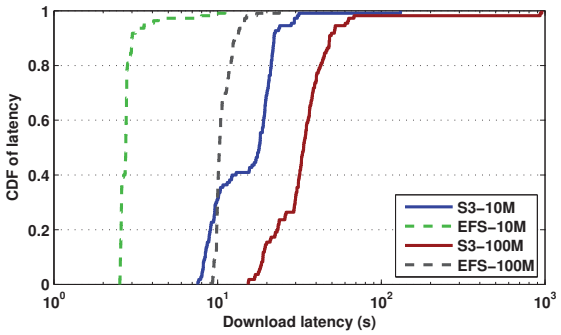

Fig. 5. CDF of download latencies for 10-MB and 100-MB files to/from S3 and EFS.

measurement results of average upload/download latencies (in log-log scale) for files are shown in Fig. 4. By comparing the upload latencies of S3 and EFS, we find that EFS outperforms S3 for small files while its performance falls behind S3 for relatively large files. Quantitatively, the file-size threshold between small and large files can be roughly taken as 1 MB or several MBs according to the intersection point (marked in Fig. 4) of the "S3-Upload" and "EFS-Upload" curves. These findings confirm the efficacy of the intuitive approach in handling both directory operations and file uploads.

On the other hand, Fig. 4 indicates that EFS substantially outperforms S3 in terms of download latency for the files of *all* sizes. For instance, a client needs an average of 10 seconds to download a 100-MB file from EFS, but nearly 50 seconds from S3. Still worse, we notice that the performance variance (or says the instability) of S3 is much larger than that of EFS, thus further aggravating the inferiority of S3 in handling file downloads. To quantify this, we plot in Fig. 5 the CDF of download latencies for 10-MB and 100-MB files. Obviously, S3 exhibits a much higher tail latency than EFS—a client can spend up to 16 minutes in downloading a 100-MB file (hence the download speed is as low as 0.1 MBps).

On the whole, while the attractive pricing makes S3 quite suitable for large-file storage, the implementation defect (*i.e.,* no mature load balance mechanism to tackle bursty data requests) largely impairs its performance for large-file delivery. If we adopt the intuitive approach for filesystem hosting, the download performance of large files (of course from S3) will become a severe system bottleneck. In addition, a long-lasting download process may influence the execution efficiency of other filesystem operations. For example, the COPY operation (on a large file f) would be held off while f is being downloaded. This is because in the intuitive approach, the COPY operation need increase the link number of f and thus the name of f is changed accordingly. Hence, the system has to wait until f is totally downloaded to start the COPY operation.

Opportunities. During our experiments, we also notice some opportunities to potentially address the system bottleneck unravelled above. First, we notice that the data transfer between S3 and EFS is quite rapid. For instance, when we transfer a 100-MB file between S3 and EFS in the same AWS region (with the help of EC2) for 100 times, the average transfer latencies in both directions are quite short: ∼4.5 seconds for S3→EFS and ∼2.5 seconds for EFS→S3. It is also worth mentioning that the data transfer between S3 and EFS in the same AWS region is free of charge [11], [12]. Besides, EFS has the same unit network-traffic price with S3, as listed in Table I. Given these three factors, *we can always employ EFS as a relay for the clients' quickly downloading large files.*

In addition, we notice that significant similarity exists between the files hosted at the cloud and its users. A comprehensive, real-world dataset of cloud storage usages [7] indicates that the majority (84%) of files hosted by cloud storage services are modified by users for at least once; moreover, over half (52%) of files can be effectively compressed. Thus, adopting data sync techniques such as delta encoding and data compression is expected to reduce the WAN traffic (between the cloud and its users) to a large extent, *e.g.,* 76% saving between adjacent versions of Emacs source codes [8], and 26% saving among the network traffic from 11 enterprise sites [9]. Unfortunately, the data APIs of S3 do not support any of these data sync techniques. As a result, whenever possible, *we can convert the full-content download of a large file into the transmission of fine-grained sync data (in a relatively small size) with some effective data sync techniques.*

III. HyCloud Design

Guided by the above two key observations, we design a cost-efficient filesystem hosting service named HyCloud by carefully tweaking the usages of S3 and EFS. In this section, we first describe the system framework, followed by an *advanced file transfer scheme* for boosting the file transfer speed, especially the large-file download speed. After that, we present *filesystem operation control mechanisms* which balance operation execution timeliness and system overhead.

A. System Framework

HyCloud fulfills all common filesystem operations based on the interactions among *Client*, *Controller* and *Relay Proxy*, as demonstrated in detail in Fig. 6. The functionality of each building component is outlined as follows.

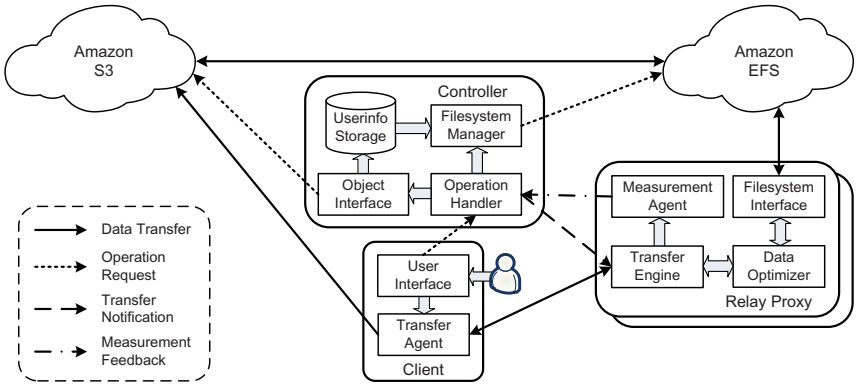

Fig. 6. System framework of HyCloud (including interaction among components and their internal module calls).

Client: HyCloud adopts a lightweight client-side implementation. On behalf of a registered user, *User Interface* keeps sending filesystem operation requests to the controller. In particular, for a file upload request *Transfer Agent* helps upload the file content to either S3 or EFS according to the file size. Besides, it is also responsible for downloading files hosted at the cloud by interacting with the relay proxies and managing the corresponding metadata.

Controller: As the command center of HyCloud, the controller handles filesystem operation requests from all user clients with *Operation Handler*, which assigns relay proxies according to their available bandwidths. *Object Interface* and *Filesystem Manager* are two cloud interfaces to S3 and EFS respectively, handling data transfer between S3 and EFS as well as maintaining link files in EFS. Moreover, *Userinfo Storage* is a small database that stores infomation of all registered users.

Relay Proxy: *Filesystem Interface* on each relay proxy interacts with EFS to forward large files from S3. *Data Optimizer* further conducts delta encoding and data compression on the file when there is an old-version file locally stored. In addition, *Measurement Agent* periodically measures link bandwidths from connected clients and feeds back the available bandwidth to the controller, and *Transfer Engine* forwards files to the clients finally.

B. Advanced File Transfer Scheme

Aimed at the goal of cost-efficient filesystem hosting, we have proposed an intuitive approach that hosts large files in S3 and small files together with directory structures in EFS. With the elaborate design of hybrid storage and directory-based maintenance, it proves to be cost-efficient for most filesystem operations. Therefore, HyCloud inherits the basic storage structure as well as all directory-related filesystem operations (*e.g.,* LIST, MOVE, COPY). On this basis, we design an advanced file transfer scheme to address the large-file download bottleneck of the intuitive approach.

Relay-Based Downloading. According to our first key observation, in addition to storing small files, HyCloud also adopts EFS as a relay to accelerate the download of large files in S3. Specifically, when a user requests HyCloud service to download a large file, the file is firstly transferred from S3 to EFS by the controller invoking their data APIs. Thereafter, an assigned relay proxy (on an EC2 VM instance to which EFS is mounted) forwards the file content to the client. Note that such a file should not be stored in the corresponding filesystem path in EFS (where its link file is), or it will inevitably influence other directory-related filesystem operations (*e.g.,* LIST, MOVE, COPY) to the path. Instead, all these files are temporarily stored in a special caching path. To make relay proxies easily forward them, they can be named by their original full paths in which the separators are replaced with a special character (*e.g.,* "#"). Given the high performance of both transfer periods (shown in §II-B), this mechanism is supposed to largely relieve the download bottleneck.

Adaptively-Adjusted Delta Encoding. In practice, for a large proportion of files, there exists significant similarity between the versions hosted at the cloud and its users (just as our second key observation illustrates). To further improve transfer efficiency, HyCloud conducts *delta encoding* for large-file download when there is already an old-version file stored locally. Concretely, to download a file f in S3, a *checksum list* of the local old-version file f' is generated by the client firstly. Once the file f is acquired from S3 or has been cached in EFS, the assigned relay proxy calculates the differences (*i.e.,* a *delta* file) between two file versions based on the uploaded checksum list. According to a large number of tests, we observe that most delta files have an over 10 times *compression ratio* (= *file size before compression* / *file size after compression*) and the computation is also quite rapid. Thus an effective compression algorithm (*e.g.,* gzip, bzip2) can be further adopted to reduce the overall download time. After the compressed delta file is returned to the client, it is decompressed and finally applied on the old-version file f' to generate the requested file f.

It is worth noting that the *delta* size Δ is determined by both file attributes (size, type, and modification scale) and the rolling chunk size. It is impossible to obtain an optimal chunk size for a file unless the file is actually transferred. Here we try to predict the best value based on the observation

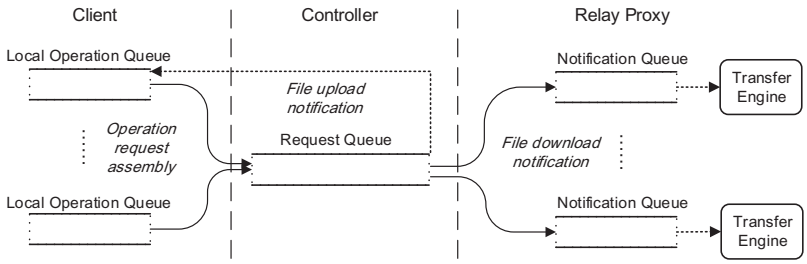

Fig. 7. The complete filesystem operation control process of HyCloud (operation requests from different clients are handled in the controller).

that the optimal chunk size is highly consistent among file versions, and design an *adaptively-adjusted delta encoding* mechanism. Specifically, we pick a small collection of typical chunk sizes $\vec{c}$ in advance, from hundreds of bytes to tens of kilobytes. For a file of size S_f, we define *elimination ratio* $\gamma = 1 - \Delta/S_f$. Every time a client finishes downloading a large file, it conducts delta encoding on the file *locally* with each chunk size c_i to get elimination ratio γ_i. Note that the process can be done in background when there is enough CPU resource. Next the local chunk-size selection probability vector are defined as the normalized elimination ratios $\vec{P}_{loc} = \{\gamma_i / \sum_i \gamma_i\}$. The correlation between newly predicted selection probability vector $\vec{P}_{new}$ and previously recorded selection probability vector $\vec{P}_{pre}$ is

$$\vec{P}_{new} = \vec{P}_{pre} * \lambda + \vec{P}_{loc} * (1 - \lambda), \tag{1}$$

where λ is a decay factor and is typically set as $n/(n+1)$ for the n-th adjustment. The chunk size corresponding to the *highest probability* in $\vec{P}_{new}$ (recorded as the new file's metadata) will be adopted next time.

File Migration in Background. It is also worth mentioning that some files are small initially but their sizes grow beyond the file-size threshold $\bar{S}_f$ due to a series of modification operations. If such files are migrated to S3 in time, we can not only save total storage cost (given the per size and time unit pricing model shown in Table I), but also improve transfer efficiency (as the designed delta encoding mechanism could be adopted then)[2]. Accordingly, a message is added into a queue named *Migration Queue* every time the size of a file uploaded to EFS exceeds the size threshold. A background thread is in charge of monitoring the queue, and conducts the real file migration offline when there are enough CPU and network resources and no delta application in progress. After the file is successfully migrated to S3, a link file with all the aforementioned metadata is built to replace it in EFS. Note that the above process should be an atomic operation, *i.e.,* any request of accessing the file cannot be responded until the two steps are both finished. Similar to handling redundant large files in §II-A, we can avoid the real migration to further reduce the transfer overhead, if there has been an object with the same hash index (object name) in S3.

[2]As file size decrease happens at a very low frequency and amount for most users, file migration from S3 to EFS can hardly bring benefits, and thus it is not considered in our design.

C. Filesystem Operation Control Mechanisms

In practice, a filesystem operation is expected to be completed as soon as it is requested by a user. However, massive workloads induced by a large scale of filesystem operation requests (especially those cannot be well optimized with the above transfer scheme) may bring heavy overhead to HyCloud proxies. Therefore, we next design filesystem operation control mechanisms in both the controller and the client, to guarantee well acceptable user experience while reducing the system overhead. Fig. 7 depicts the whole filesystem operation control process of HyCloud.

Operation Request Handling. When operation requests arrive at *Request Queue* of the controller, they are added to a priority-based message queue named *Request Queue*. Each filesystem operation request includes file metadata like operation type and arrival timestamp. As the directory-related operations do not involve data transfer, HyCloud sets a higher priority for them, which will be handled before file transfer operations. Among operations of the same type, FCFS (First Come First Served) is adopted. Particularly, operations to one file (*e.g.,* download and then delete) should be executed based on the request order to avoid potential logic errors. Meanwhile, the available transfer bandwidth in each relay proxy is measured by a monitor thread in background, and periodically fed back to the controller. On this basis, the controller assigns relay proxies with top transfer bandwidth to tackle file operations by sending notifications to their *Notification Queues*.

Redundant Operation Elimination. To further boost the efficiency of filesystem operation executions, additional control mechanisms are adopted in the HyCloud client. Firstly, it is manifest that too many frequent operation requests from a large number of users can severely influence the performance of controller. Therefore, whenever the controller feeds back its performance reduction or a client itself encounters network bandwidth bottleneck, we cache filesystem operation requests in *Local Operation Queue*, and then send them to the controller as soon as there is no bottleneck. Moreover, during the caching interval, some redundant requests can be eliminated locally. For instance, a file upload request is removed once a deletion

or another upload operation to the same file appears subsequently. Through the above mechanisms, we can guarantee the system scalability as well as reduce the real filesystem operation workloads to a large extent.

IV. Performance Evaluation

In this section, we first briefly present the implementation of HyCloud prototype. On the basis of real-world deployment, we conduct measurements on the effectiveness of the designed file transfer scheme in different scenarios with a variety of typical workloads. Finally, we evaluate the system's storage cost-effectiveness with a large-scale data trace.

A. HyCloud Prototype

We have implemented a prototype of HyCloud framework in approximately 3500 lines of codes for all three platforms (*client*, *controller* and *relay proxy*). The prototype can provide a cost-efficient filesystem hosting service atop S3 and EFS in a scalable manner. The source code is available at *https://github.com/iHyCloud/hycloud-demo*.

Particularly, we implement rsync [14]-like delta encoding without invoking rsync libraries, thus conveniently adding transfer optimization mechanisms while avoiding extra overhead. Meanwhile, a moderate amount (typically 1 GB) of EFS storage capacity is used to cache the forwarded files, which are recycled periodically in idle time following LRU (Least Recently Used) caching scheme. Both the controller and relay proxies interact with S3 and EFS by invoking their data APIs respectively. When a number of operation requests arrive simultaneously, HyCloud will start multiple threads to accelerate API upload or download correspondingly under the designed transfer scheme. In addition, the bandwidth feedback as well as workload dispatch rely on the interaction between the controller and a relay proxy with Apache MINA [15].

B. Experiment Setup

We adopt both S3 and EFS in the AWS Oregon region for all the experiments, because the region is overall the best among all AWS regions on the aspects of both performance and cost. Based on our first key observation, we deploy the controller and three relay proxies on Amazon EC2 *t2.micro* instances (the configuration is low and cheap) in the same region, where EFS is able to be directly mounted and S3 can be also efficiently accessed. According to the requirement of experiments, we adopt DigitalOcean VM node in Singapore, Toronto and London (the same as in §II-B) as HyCloud clients with the lowest configuration but unlimited network bandwidth, which can well simulate user PC performance and avoid local network congestion. As the comprehensive cost and performance efficacy of the intuitive approach is proven to be better than only storing files in S3 or EFS in §II-B, next we evaluate it as a baseline for performance comparison.

In addition, to calculate a convincing overall unit storage price of HyCloud, we utilize a large-scale data trace of Xuanfeng [16] cloud service. This trace was collected from 742,064 users with 3,412,827 files over a week. Specifically, we select the records on Feburary 22, 2015 for storage cost comparison (with S3 and EFS), which has totally 514,095 files ranging from 5 KB to approximately 4 GB.

C. Effectiveness of Advanced File Transfer Scheme

Relay-based downloading is a basic mechanism that HyCloud adopts to tackle the large-file download bottleneck. We first evaluate it on a number of typical large files (the size ranges from 10 MB to 100 MB). The overall download time of each size is shown in Fig. 8. We also show the performance of the intuitive approach (directly downloading the large files from S3) for contrast. As the figure illustrates, the file download in all sizes experience performance promotion, with time reduction up to 83.9% and 64.5% on average. Especially, downloading a 100-MB file only takes at most 15 seconds with HyCloud, which indicates the effectiveness of the relay-based downloading mechanism.

In addition, HyCloud adopts adaptively-adjusted delta encoding in the relay-based download process for files with old versions locally stored. Accordingly, we conduct evaluation on download time of source codes, which are updated frequently in general. Specifically, four common source code tar files in different sizes are adopted (Redis $\sim$7 MB, Tomcat $\sim$25 MB, FFmpeg $\sim$58 MB, Hadoop $\sim$100 MB). We download 10 latest versions of codes sequentially based the adaptively-adjusted delta encoding mechanism in HyCloud. Fig. 9 describes the transfer time of each multi-version code file, in which the intuitive approach (downloading the source codes from S3) also serves as a comparison. HyCloud can bring quite large efficiency promotion, reducing the download time up to 81.9% (67.3% on average). It is also worth mentioning that the average download time of 100-MB Hadoop code files is less than 8 seconds, even exceeding the performance of EFS ($\sim$10 seconds for the 100-MB file download as shown in Fig. 4).

We further evaluate the effectiveness of adaptively-adjusted delta encoding mechanism by measuring the network traffic incurred among the above 10 different versions of FFmpeg code files. Note that the overall data transfer latency is positively related to network traffic, as computation time of delta sync varies little with different chunk sizes. The metric *transfer traffic ratio* is defined as the ratio of the transfer traffic with a chunk size to the theoretically optimal one. Fig. 10 shows the traffic ratio of HyCloud in contrast with that of several typical chunk sizes. We observe that the transfer traffic of HyCloud converges to the optimal curve much faster than all fixed chunk sizes, and its steadiness among different versions also shows robustness of the mechanism. As the computation is mainly conducted in the idle time, the mechanism brings little overhead in practice.

D. Storage Cost-Effectiveness of HyCloud Service

In addition to the above experiments on the main transfer optimization mechanisms, we also evaluate two overall indicators of HyCloud service—storage cost and scalability. Before that, we first determine the file-size threshold $\bar{S}_f$ for HyCloud filesystem hosting by testing a small collection of

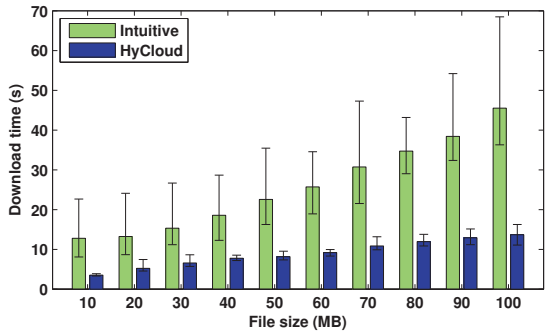

Fig. 8. Download time reduction with the relay-based mechanism for large files in different sizes.

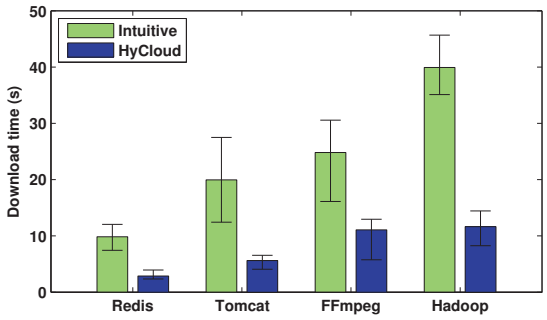

Fig. 9. Download time reduction for different versions of four common source codes.

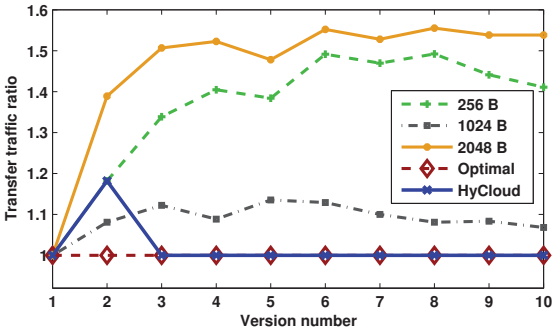

Fig. 10. Performance of adaptively-adjusted delta encoding mechanism (using one FFmpeg version at a time).

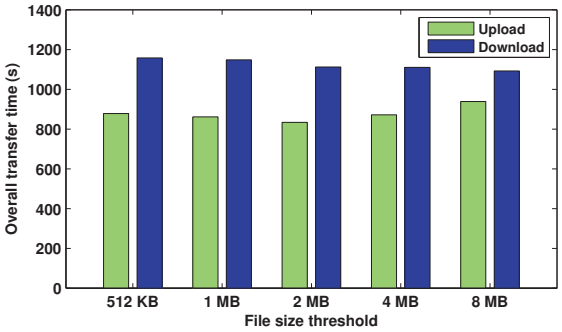

Fig. 11. Impact of file size threshold on the overall upload and download time of a number of files.

TABLE II
STORAGE COSTS OF CLOUD SERVICES UNDER TYPICAL WORKLOADS

Cloud Service	File Data Cost ($/Month)	Metadata Cost ($/Month)	Unit Storage Price ($/GB/Month)
Amazon S3	11712.496	0.154	0.02278
Amazon EFS	167321.366	0	0.3
HyCloud	11717.66	0.144	0.02279

typical file sizes (512 KB∼8 MB) around the rough threshold (1 MB) for the intuitive approach (in §II-B). Specifically, we generate 100 files in different sizes which are randomly selected from the data trace (file sizes over 1GB are filtered out), and then evaluate their overall transfer performance by uploading and downloading them to/from S3 or EFS according to each threshold. As the results depicted in Fig. 11, we can set the file-size threshold as 2 MB for HyCloud service due to its overall best performance.

On this basis, we evaluate the storage cost of HyCloud through overall unit storage price comparison. For files in the selected one-day data trace, the costs ($) of storing their file data and metadata in the three cloud services are shown in Table II. According to the latest prices of S3 and EFS, the overall unit storage price *(including metadata storage)* of HyCloud is ∼$0.02279/GB/month for the selected typical file workloads, which is much cheaper than that of EFS and quite close to that of S3 (the increase is only 0.43% if S3 metadata is similarly stored in EFS). The evaluation well shows the cost-effectiveness of HyCloud as a filesystem hosting service.

V. RELATED WORK

There has been a quantity of work on the topic of cloud storage service, which our work is mainly related to in the following three aspects.

Cloud Measurement Studies. A dozen of research papers measure and benchmark performance of both public clouds [5], [17] and personal cloud services [18], [7]. Especially, some papers elaborately study the well-performed cloud services like Azure [19], Dropbox [20], UbuntuOne [21] and OpenStack Swift [22] by pinning their inside architectures. In contrast with them, we are the first to measure the performance of the newly-launched Amazon EFS on all common filesystem operations. More importantly, we further observe that EFS can work as a relay to accelerate the large-file downloads from S3.

Multiple Cloud Storage. Some previous work has applied multiple public or personal cloud services for client-central redundant data backup (*e.g.,* DepSky [23], MetaSync [24], CYRUS [25]), as well as enabled efficient cross-cloud file collaboration based on cloud web APIs (*e.g.,* CoCloud [26]). Besides, some other studies leverage different clouds to reduce transfer latency [6] or provide cost-effective data placement

[27]. Unlike those studies binding multiple object storage services for redundant chunk-based storage, HyCloud provides a cost-efficient filesystem hosting service on top of two distinct types of cloud storage services by carefully tweaking the usages of them.

Optimization in Data Transfer. There have been a number of relevant cloud storage techniques these years, like chunking [8], [9], delta encoding and deduplication [14], [28] as well as bundling [29], [30]. However, the APIs provided by Amazon S3 support none of the above techniques. Fortunately, EFS and HyCloud controller can be deployed near enough to S3 storage servers to overcome their inefficacy, and thus file transfer operations can be very efficient in virtue of our advanced file transfer scheme.

VI. Conclusion

Motivated by the fact that object storage services like Amazon S3 support simple, flat operations with a low price and filesystem storage services like Amazon EFS support complex, efficient operations with a much higher price, this paper has presented a cost-efficient filesystem hosting service through carefully tweaking the usages of Amazon S3 and EFS.

To achieve the above goal, we first reveal two key observations to address the large-file download bottleneck: 1) data transformation between S3 and EFS is cheap and 2) large-size file downloads can be coverted into small-size file synchronizations. Guided by the observations, we design enabling mechanisms of relay-based downloading, adaptively-adjusted delta encoding, and filesystem operation control. Finally we put the above proposed techniques together to develop an open-source HyCloud prototype, which can offer cost-efficient filesystem hosting in a scalable way. Our real-world evaluations demonstrate that the unit storage price of HyCloud is close to that of Amazon S3, and the filesystem operations are executed as quickly as in EFS in most times (sometimes even more quickly than in EFS).

Acknowledgment

This research is supported by the National Key R&D Program of China under grant 2017YFB1010002, the National Natural Science Foundation of China under grants 61872211 and 61822205, and the Tsinghua University Initiative Scientific Research Program under grant 2014Z09103.

References

[1] "Aliyun OSS (Object Storage Service)," https://www.aliyun.com/product/oss.
[2] "Aliyun NAS (Network Attached Storage)," https://www.aliyun.com/product/nas.
[3] "The Open Group Base Specifications Issue 7," *IEEE Std 1003.1*, 2008, http://pubs.opengroup.org/onlinepubs/9699919799.
[4] "Amazon Elastic File System (EFS)," https://aws.amazon.com/efs/.
[5] A. Li, X. Yang, S. Kandula, and M. Zhang, "CloudCmp: Comparing Public Cloud Providers," in *Proc. of ACM Internet Measurement Conference (IMC)*, 2010, pp. 1–14.
[6] Z. Lai, Y. Cui, M. Li, Z. Li, N. Dai, and Y. Chen, "TailCutter: Wisely Cutting Tail Latency in Cloud CDN under Cost Constraints," in *Proc. of IEEE International Conference on Computer Communications (INFOCOM)*, 2016, pp. 1845–1853.
[7] 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 *Proc. of ACM Internet Measurement Conference (IMC)*, 2014, pp. 115–128.
[8] A. Muthitacharoen, B. Chen, and D. Mazières, "A Low-bandwidth Network File System," in *Proc. of ACM Symposium on Operating Systems Principles (SOSP)*, 2001, pp. 174–187.
[9] B. Aggarwal, A. Akella, A. Anand, A. Balachandran, P. V. Chitnis, C. Muthukrishnan, R. Ramjee, and G. Varghese, "EndRE: An End-System Redundancy Elimination Service for Enterprises," in *Proc. of USENIX Symposium on Network System Design and Implementation (NSDI)*, 2010, pp. 419–432.
[10] "Amazon S3 Pricing," https://aws.amazon.com/s3/pricing/.
[11] "Amazon EFS Pricing," https://aws.amazon.com/efs/pricing/.
[12] "Amazon EC2 On-Demand Pricing," https://aws.amazon.com/ec2/pricing/on-demand/.
[13] "DigitalOcean," https://www.digitalocean.com/.
[14] A. Tridgell and P. Mackerras, "The Rsync Algorithm," *Joint Computer Science Technical Report Series, Australian National University*, 1996.
[15] "Apache MINA," http://mina.apache.org.
[16] "Xuanfeng Downloading Service," http://xf.qq.com.
[17] K. He, A. Fisher, L. Wang, A. Gember, A. Akella, and T. Ristenpart, "Next Stop, the Cloud: Understanding Modern Web Service Deployment in EC2 and Azure," in *Proc. of ACM Internet Measurement Conference (IMC)*, 2013, pp. 177–190.
[18] I. Drago, E. Bocchi, M. Mellia, H. Slatman, and A. Pras, "Benchmarking Personal Cloud Storage," in *Proc. of ACM Internet Measurement Conference (IMC)*, 2013, pp. 205–212.
[19] B. Calder, J. Wang, A. Ogus, and N. Nilakantan, "Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency," in *Proc. of ACM Symposium on Operating Systems Principles (SOSP)*, 2011, pp. 143–157.
[20] I. Drago, M. Mellia, M. M. Munafò, A. Sperotto, R. Sadre, and A. Pras, "Inside Dropbox: Understanding Personal Cloud Storage Services," in *Proc. of ACM Internet Measurement Conference (IMC)*, 2012, pp. 481–494.
[21] R. Gracia-Tinedo, Y. Tian, J. Sampé, H. Harkous, J. Lenton, P. Garcá-López, M. Sánchez-Artigas, and M. Vukolic, "Dissecting UbuntuOne: Autopsy of a Global-scale Personal Cloud Back-end," in *Proc. of ACM Internet Measurement Conference (IMC)*, 2015, pp. 155–168.
[22] M. Ruan, T. Titcheu, E. Zhai, Z. Li, Y. Liu, J. E, Y. Cui, and H. Xu, "On the Synchronization Bottleneck of OpenStack Swift-Like Cloud Storage Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 9, pp. 2059–2074, 2018.
[23] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa, "DepSky: Dependable and Secure Storage in a Cloud-of-Clouds," in *Proc. of ACM European Conference on Computer Systems (EuroSys)*, 2011, pp. 31–46.
[24] S. Han, H. Shen, T. Kim, A. Krishnamurthy, T. Anderson, and D. Wetherall, "MetaSync: File Synchronization Across Multiple Untrusted Storage Services," in *Proc. of USENIX Annul Technical Conference (ATC)*, 2015, pp. 83–95.
[25] J. Y. Chung, C. Joe-Wong, S. Ha, J. W.-K. Hong, and M. Chiang, "CYRUS: Towards Client-Defined Cloud Storage," in *Proc. of ACM European Conference on Computer Systems (EuroSys)*, 2015, pp. 1–16.
[26] J. E, Y. Cui, P. Wang, Z. Li, and C. Zhang, "CoCloud: Enabling Efficient Cross-Cloud File Collaboration based on Inefficient Web APIs," in *Proc. of IEEE International Conference on Computer Communications (INFOCOM)*, 2017, pp. 28–36.
[27] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha, "SPANStore: Cost-Effective Geo-Replicated Storage Spanning Multiple Cloud Services," in *Proc. of ACM Symposium on Operating Systems Principles (SOSP)*, 2013, pp. 292–308.
[28] Y. Hua, X. Liu, and D. Feng, "Neptune: Efficient Remote Communication Services for Cloud Backups," in *Proc. of IEEE International Conference on Computer Communications (INFOCOM)*, 2014, pp. 844–852.
[29] 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 *Proc. of ACM International Middleware Conference (Middleware)*, 2013, pp. 307–327.
[30] Y. Cui, Z. Lai, X. Wang, N. Dai, and C. Miao, "QuickSync: Improving Synchronization Efficiency for Mobile Cloud Storage Services," in *Proc. of ACM International Conference on Mobile Computing and Networking (MobiCom)*, 2015, pp. 592–603.