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

Jinlong E, Yong Cui, *Member, IEEE*, Zhenhua Li, *Member, IEEE*, Mingkang Ruan, and Ennan Zhai

***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. In practice, 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 article, we attempt to address the bottleneck with little overhead by carefully tweaking the usages of S3 and EFS. Guided by two key 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 intelligently schedules the data transfer among S3, EFS and the clients in a distributed manner. 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).**

***Index Terms*— Hybrid cloud storage, filesystem hosting, cloud computing, transfer efficiency.**

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]).

Manuscript received June 19, 2019; revised June 3, 2020; accepted July 30, 2020; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor Y. Guan. Date of publication September 11, 2020; date of current version December 16, 2020. This work was supported in part by the National Key Research and Development Program of China under Grant 2018YFB1800303 and Grant 2017YFB1010002 and in part by the National Natural Science Foundation of China (NSFC) under Grant 61822205 and Grant 61872211. *(Corresponding author: Yong Cui.)*

Jinlong E is with the School of Software, Tsinghua University, Beijing 100084, China (e-mail: ejinlong@tsinghua.edu.cn).

Yong Cui is with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China (e-mail: cuiyong@tsinghua.edu.cn).

Zhenhua Li and Mingkang Ruan are with the School of Software and KLISS MoE, Tsinghua University, Beijing 100084, China (e-mail: lizhenhua1983@gmail.com; brmk@vip.qq.com).

Ennan Zhai is with the Network Research Group, Alibaba Group Inc., Seattle, WA 98004 USA (e-mail: ennan.zhai@alibaba-inc.com).

Digital Object Identifier 10.1109/TNET.2020.3019571

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, especially those operations involving directory structures. Third parties can thus directly build upper-layer applications to support POSIX-like operations atop 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 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

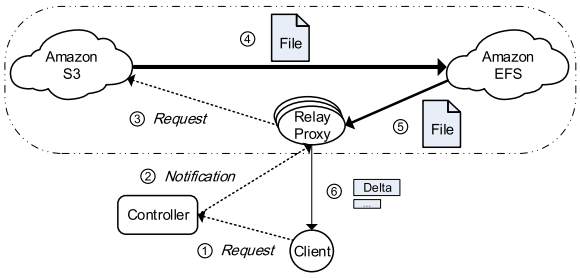

Fig. 1. Architectural overview of HyCloud, and a typical process of a client downloading a large file from the hybrid cloud storage services (Step ①-⑥).

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 article, we attempt to address the bottleneck with little overhead by carefully tweaking the usages of S3 and EFS. 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 the users [7]–[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 notifies *relay proxies* to execute the filesystem operations on behalf of a user through invoking the data APIs of S3 and EFS (EC2 VM instances work as relay proxies given that they are necessary for executing POSIX-like filesystem operations on EFS). In order to accelerate the client's downloading large files, HyCloud leverages EFS and a relay proxy to forward the file content from S3 to the clients. 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.

To boost efficiency of all file transfer operations, we devise additional traffic-aware mechanisms including large-file incremental upload and small-file bundle transfer. Still, when deployed within a single AWS region, HyCloud could produce suboptimal performance when serving geo-distributed clients. To address this, we strategically deploy multiple HyCloud *instances* in different AWS regions, and accordingly design dedicated mechanisms such as erasure-coding storage and multi-source file download to optimize the distributed data transfer performance and the data storage overhead.

Furthermore, we devise an online dataflow scheduling algorithm to moderately balance the timeliness of filesystem operation executions and the bandwidth overhead of relay proxies. Specifically, based on the link bandwidths between the clients and relay proxies, the algorithm dynamically adjusts the execution priorities assigned to the clients' filesystem operation requests. When the timeliness falls below a threshold thus causing considerable degradation in user experience, more relay proxies will be added to the system to deal with bursty workloads. We also design additional control mechanisms to eliminate redundant operation requests, guarantee the filesystem consistency, and tackle controller failures.

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× 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.

This article makes the following contributions:

- We propose a combinatory use of two major types of cloud storage services (Amazon S3 and EFS) to provide users efficient filesystem operations with a low price, and observe opportunities to address the performance bottleneck of large-file downloads in the intuitive approach (§II).
- Based on the observations, we carefully tweak the usages of S3 and EFS with an advanced file transfer scheme consisting of relay-based file download and adaptively-adjusted delta encoding, and design dedicated multi-instance deployment and transfer optimization for geo-distributed clients (§III-B).
- Additionally, we devise an online dataflow scheduling algorithm to boost the timeliness of operation executions while reducing system overhead, and design a bundle of control mechanisms to maintain operational reliability (§III-C).
- We implement an open-source cost-efficient filesystem hosting system called HyCloud to embody the above enabling solutions. Extensive real-world evaluations demonstrate its efficiency, cost-effectiveness and scalability (§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 ∼15 times as) that of Amazon S3 (∼$0.02/GB/month)

TABLE I
PRICING MODELS OF AMAZON S3 AND EFS

Cloud Service	**Storage** ($/GB/Month)	**Network Traffic** (Outbound, $/GB)	**Requests** ($)
Amazon S3	0.023 ~ 0.025	0.05 ~ 0.09 (Inbound free)	PUT 5×10^{-6} GET 4×10^{-7}
EFS via EC2	0.3 ~ 0.39	0.05 ~ 0.09 (Inbound free)	Free

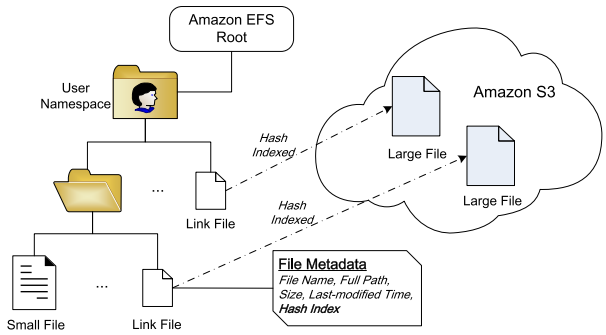

Fig. 2. The hybrid system design using the intuitive approach (large files stored in S3 are indexed by link files).

[10]–[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 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 path known to the user's client. 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. 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, inspired by *copy-on-write* adopted in Linux [13], 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 [14] VM nodes located at Singapore (SGP), London (LON) and Toronto (TOR). 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. We take the measurement results over all three geolocations into account when calculating the average or CDF of latencies.

Pros and Cons: As depicted in Fig. 3, all common directory-related operations can be quickly executed in 0.44 seconds. Except for the network-level round trip time between the clients and EC2, a single directory operation's execution time is 0.1 seconds at most (on an EC2 VM instance to which the EFS filesystem is mounted). Next, the 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

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

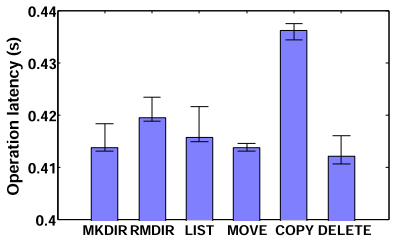

Fig. 3. Operation latencies of EFS on common directory-related operations (including the round trip time).

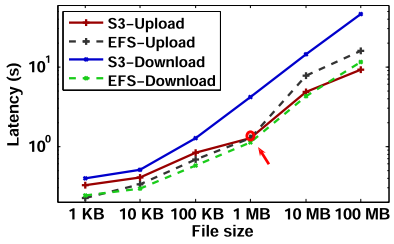

Fig. 4. Average upload/download latencies for files in different sizes to/from S3 and EFS.

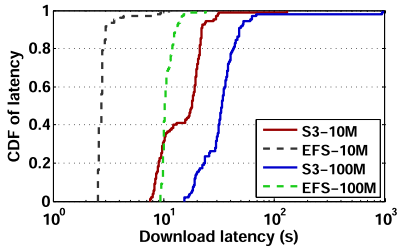

Fig. 5. CDF of download latencies for 10-MB and 100-MB files from S3 and EFS.

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

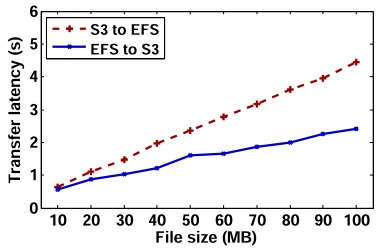

Fig. 6. Average data transfer latencies for large files in different sizes between S3 and EFS.

for S3→EFS and ∼2.5 seconds for EFS→S3 (as depicted in Fig. 6). The download latencies from S3 and EFS together with the S3→EFS transfer latency (denoted as t_{SC}, t_{EC}, and t_{SE}, respectively) comply with the phenomenon of Triangle Inequality Violation (TIV) (*i.e.,* $t_{SE}+t_{EC} < t_{SC}$), and thus a relay mechanism is able to reduce its download latency [15]. It is also worth mentioning that the data transfer between S3 and EFS within 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 all above 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 the 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*.

Geo-Distributed Cloud Instances: It is worth noting all the above measurements are on the premise that only S3 and EFS in the AWS Oregon region are deployed. Indeed, when the users' geolocations are relatively centralized (and the number of users is not too large), deploying the hybrid clouds in an AWS region with the lowest average access latency can provide each user well acceptable file transfer performance. However, the latencies of most cloud storage services (especially object storage services like S3) show significant spatial and temporal variances [6], [16], and thus the transfer performance of a *centralized* cloud would not fulfill users of all geographic regions in any period of time. As shown in Fig. 7, when downloading a 10-MB file from S3 instances in AWS Oregon (OR) and Ireland (IRL) regions, both median and tail latencies of the Singapore client are much higher than those of the Toronto and London clients (close to Oregon and Ireland respectively).

An intuitively feasible solution is to deploy distributed cloud instances serving their nearby clients in different regions. However, as the performance of S3 is highly unstable, even nearby clients may still encounter the *high tail latency* problem when downloading large files. As illustrated by the file download comparisons in Fig. 8 (three clients download

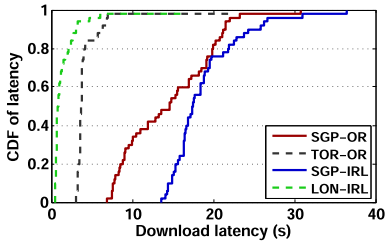

Fig. 7. CDF of download latencies for a 10-MB file from S3 with clients in different regions.

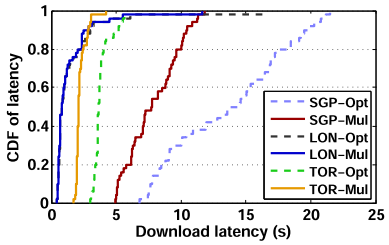

Fig. 8. CDF of download latencies for a 10-MB file from optimal and multiple cloud instance(s).

the same file concurrently from the above two S3 instances versus only from the optimal instance), leveraging multiple cloud instances could be an effective approach to reduce the user-perceived latency as well as latency variances. In addition, reducing the data storage and transfer overhead should also be considered, given that intuitive multi-replica storage and inter-instance transfer are costly. Therefore, it is necessary to deploy multiple cloud instances and well design optimization mechanisms for distributed data transfer.

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* to boost the file transfer speed, especially for the large-file download and geo-distributed data storage. After that, we present a *filesystem operation control scheme* which balances the timeliness of filesystem operation executions and the 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. 9. The functionality of each building component is outlined as follows.

Client: HyCloud adopts a lightweight client-side implementation. On behalf of a registered user, *Operation Handler* 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 in multiple HyCloud instances 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 geo-distributed user clients. The work is mainly scheduled by the

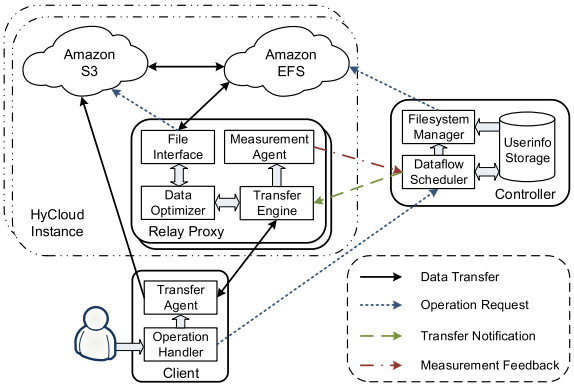

Fig. 9. System framework of HyCloud (a centralized controller interacts with HyCloud instances in different AWS regions and geo-distributed clients).

core *Dataflow Scheduler*, which adaptively adjusts the number of in-used relay proxies in each instance according to the scale of workloads and their available bandwidths. *Filesystem Manager* accesses EFS of all instances to execute directory-based filesystem operations. Moreover, *Userinfo Storage* is a small database that stores infomation of all registered users.

Relay Proxy: HyCloud adopts an adaptive number of relay proxies in multiple instances together with S3 and EFS. *File Interface* on each relay proxy executes file transfer operations by invoking data APIs, and meanwhile it handles data transfer between S3 and EFS and maintains link files. *Measurement Agent* periodically measures link bandwidths from connected clients and feeds back the available bandwidth to the controller for file workload dispatch. *Transfer Engine* is scheduled by the controller to forward a file to the client. *Data Optimizer* further conducts delta encoding and data compression on the file when there is an old-version file stored on the client.

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, and further boost efficiency of all file transfer operations. Besides, the scheme also optimizes data transfer performance and storage overhead for geo-distributed HyCloud instance deployment.

Relay-Based File Download: 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 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) but temporarily stored in a special caching path, to avoid influencing other directory-related filesystem operations (*e.g.,* LIST, MOVE,

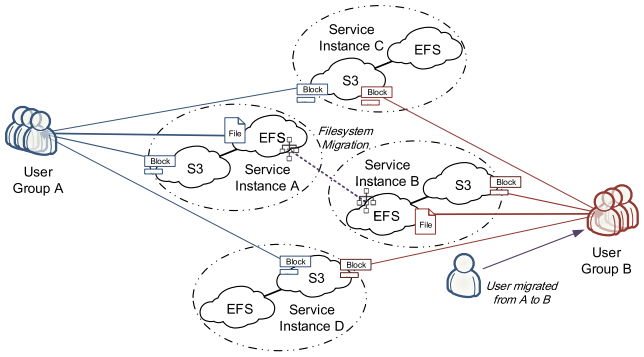

Fig. 10. An example of geo-distributed instance deployment (users are divided into groups based on the clients' optimal service instances).

COPY) to the 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) as well as data caching, 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 program (*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. As a result, it seems impossible to obtain an optimal chunk size for a file unless the file has already been transferred. Instead, given our observation that the optimal chunk size is highly consistent among versions of a file, here we design an *adaptively-adjusted delta encoding* mechanism to adjust the chunk size based on historical ones. 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 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 to gradually reduce the influence of previous probabilities, and it 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.

Additional File Transfer Optimization: In addition to addressing the large-file download bottleneck, we also try to optimize other file transfer operations. According to the performance measurements in §II-B, the S3 upload is efficient enough for typical large files (*e.g.,* uploading a 100-MB file only takes ∼10 seconds on average). Thus a HyCloud client generally uploads a large file to S3 directly. However, there are still a number of clients that experience client-side bandwidth bottlenecks. In this case, the upload of large files may also benefit from delta encoding.

A typical process of delta-encoding upload is that the client generates a delta file every time a large file is modified and uploads it to EFS as a small file, and then the delta file is applied to the old-version file acquired from S3 to generate a new file. The corresponding link file is renewed thereafter, in which the checksum list is replaced with the latest one. This process is only conducted when its overall time (including the time of delta computation and data upload, as well as the time for data transfer between S3 and EFS in both directions) is less than that of the direct upload approach. Accordingly, when uploading a large file with an old version at the cloud, HyCloud compares data upload rate r_u with delta computation rate r_c, which can be inferred from the client's real-time outbound network bandwidth and its computation capacity, respectively. The correlation between the above two rates r_u and r_c should fulfill

$$r_u < r_c r_t \gamma_{max} / (2r_c + r_t), \tag{2}$$

where the average data transfer rate between S3 and EFS r_t is acquired by periodical measurements (refer to Fig. 6), and the currently maximum elimination ratio γ_{max} is calculated based on the above adaptively-adjusted delta encoding mechanism.

Besides, for small files stored in EFS, we supplement a bundling mechanism to further boost their transfer efficiency. Instead of one HTTP connection per file upload/download, a persistent network connection is set up between HyCloud client and a special relay proxy for uploading or downloading all available small files to/from EFS. On this basis, asynchronous application-layer *file transfer status* acknowledgments are adopted instead of stop-and-wait ACK mode given that these files are not related to each other. A failed file transfer is tackled by putting the request into *Request Queue* for retransmission (shown in Fig. 11).

Geo-Distributed Instance Deployment and Optimization: Given the analysis in §II-B, we can deploy multiple HyCloud instances in different AWS regions to preserve sound performance when serving geo-distributed clients. Initially, each registered user's client uploads and downloads an enough-large test file (a typical size is 10 MB) to/from all HyCloud instances, and latency values are fed back to the controller in groups. The instances with access latency less than $+10\%$ the lowest value for a client are selected as its service instances [17], among which the one with lowest access latency is denoted as the *optimal* service instance. As illustrated in Fig. 10, while instances A and B are two user groups' optimal service instances separately, instances C and D are also service instances for both user groups. All clients' current service instances and the corresponding latencies are

maintained in *UserInfo Storage* database of the controller (as shown in Fig. 9), which will be updated periodically and inquired every time a file transfer request arrives.

On the basis of the above instance deployment mechanism as well as the aforementioned file transfer optimization, we next further consider effective storage and transfer optimization for the distributed service instances. On the one hand, HyCloud can still host a user's filesystem (including the directory structures, small files, and links files of large files) in EFS, which is deployed in the client's optimal service instance (*e.g.,* instance A serves for user group A and instance B serves for user group B in Fig. 10). The design is based on the observations in §II-B that small-file transfer as well as directory-related operations generally has stably high performance over time. On the other hand, large files should be stored in S3 of multiple service instances (*e.g.,* instances C and D also store files of user groups A and B) to further tackle the temporal large-file performance bottleneck, especially for the clients without service instances nearby.

Accordingly, erasure coding is wisely applied to distribute large files to multiple service instances, which provides high data availability and reliability while introducing low storage overhead in contrast with storage replication. Before a file is uploaded, the client segments it into fixed chunks, where the chunk size is set to be the file-size threshold $\bar{S}_f$. Then for each chunk, n blocks are generated with Reed-Solomon erasure coding [18], among which there are k data blocks and $n-k$ parity blocks (denoted as $RS(n,k)$). By this means, any k blocks can restore the original chunk (*i.e.,* download failures of up to $n-k$ blocks are tolerated). As each block is then uploaded to a distinct service instance, the block number n should be no more than the number of the client's service instances N (fed back by the controller); to support erasure coding, n should also be no less than 3, *i.e.,* $3 \leq n \leq N$. Note that when $N < 3$, the erasure coding is not applicable and HyCloud adopts the replication mechanism instead.[2] Further, the data block number k should fulfill the constraint $2 \leq k < n$. While the upload and download mechanisms below are applicable to all possible (n,k) values, it is better to set them as the minimum values to reduce data transfer and computation overhead.

To balance the loads of HyCloud instances, a *hash ring* is built to maintain all N service instances of a client, and each time n of them are selected for file upload based on the consistent hashing algorithm [19]. Note that when a service instance is lack of available bandwidth, the process can be asynchronously executed before the file download starts. Once the file upload is completed, all blocks' hash values and locations are recorded in the corresponding link file together with other metadata. Given the n service instances with blocks of a file, we next dispatch their download workloads with a *multi-source file download* algorithm to minimize the file download time. Suppose the file is segmented into m chunks, and the block size of chunk i $(i = 1, 2, \cdots, m)$ is b_i. Moreover, d_{ri} indicates if a block of chunk i is downloaded from service instance r $(r = 1, 2, \cdots, N)$, and s_{ri} indicates if the block is stored in service instance r ($1 =$ *yes* and $0 =$ *no* for both indicators). For each chunk, k of n blocks are required to be downloaded, and thus d_{ri} fulfills the constraint

$$\sum_r d_{ri} = k, d_{ri} \leq s_{ri} \in \{0,1\}, \forall i. \tag{3}$$

It is also worth mentioning that the relay-based file download mechanism is still adopted here. For a given service instance r, both the S3→EFS transfer bandwidth BW_r^t and the EFS→client download bandwidth BW_r^d should be estimated in advance. To boost real-time optimal workload allocation, here we induce an online bandwidth estimation mechanism. Based on the observation that the network conditions are usually stable on short timescales [20], it is possible to obtain reasonably accurate bandwidth prediction for a short horizon to the future. Therefore, we collect historical relay-based download latencies for each service instance, and calculate both bandwidth sets $\{BW_r^t\}$ and $\{BW_r^d\}$ of the latest K files (up to 5 in the same *period*). The harmonic means of these bandwidths can be regarded as the two real-time bandwidths [21],

$$\tilde{BW}_r^x = K/(\sum_{i=1}^{K} 1/BW_{ri}^x), \forall r \ (x = t, d). \tag{4}$$

When there is no file download for a service instance during a period, the bandwidths are estimated by the last periodical test-file measurements instead.

In addition, the adaptively-adjusted delta encoding mechanism is still applicable, *i.e.,* only relevant blocks are transferred based on the mechanism when a file is updated. Thus, we also define the delta computation rate C_r and the elimination ratio γ, which are forecasted by file history traits provided by the client ($\gamma = 0$ if no delta encoding is conducted). Then the workload download time of service instance r is calculated by

$$t_r = \sum_i b_i d_{ri} * [1/\tilde{BW}_r^t + (1-\gamma)/\tilde{BW}_r^d + 1/C_r]. \tag{5}$$

Given that the file download time is the maximum workload download time from n service instances, our objective can be expressed as $\arg\min_{d_{ri}}\{\max\{t_r\}\}$. For such a 0-1 programming problem, we adopt a near-optimal solution with low running time. Firstly, we compute a convex approximation that does not consider the integer constraint on d_{ri} (replacing d_{ri} with the closest linear estimator $\hat{d_{ri}} = (3^{1/4} d_{ri} + 3^{-1/4})/2$). Then we can solve the converted linear integer optimization problem for the optimal $\{d_{ri}\}$ and $\{t_r\}$ with the standard branch-and-bound algorithm [22].

C. Filesystem Operation Control Scheme

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 which cannot be well optimized with the above transfer scheme) may bring heavy overhead to HyCloud proxies. Therefore, we next design a filesystem operation control scheme including an online dataflow scheduling algorithm and additional control mechanisms in both controller and client sides, to guarantee well acceptable timeliness of filesystem operation executions while reducing the system overhead as much as possible. Fig. 11 depicts the whole filesystem operation control process of HyCloud.

Dataflow Scheduling Algorithm: We first design an online scheduling algorithm including priority assignment and workload dispatch for filesystem operation requests before their

[2]In this case, the client uploads/downloads the whole file to/from the service instance(s) based on the aforementioned mechanisms; and when there are two instances, their download workloads are allocated proportionally according to the client-instance link bandwidths.

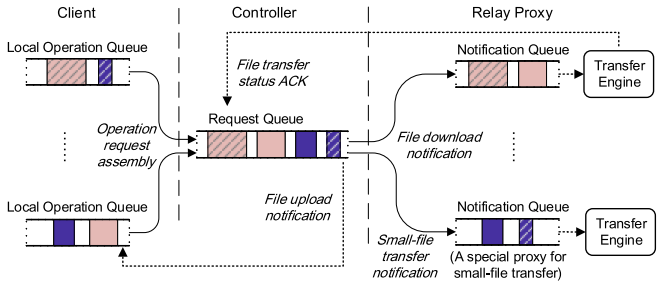

Fig. 11. The complete filesystem operation control process of HyCloud (operation requests from different clients are scheduled in the controller).

real executions. Each operation request includes file metadata like file size (0 for a directory), operation type and arrival timestamp. They are added to a priority-based message queue named *Request Queue* when arriving at the controller. We next optimize the overall performance of filesystem operations by adjusting the orders in which the requests are handled.

In general, users are less sensitive to the timeliness of large file transfer than that of small file transfer and other directory-related operations. Therefore, we take both arrival time of a request and the file size (together with the operation type) into account. While the execution of a large-file operation may be postponed to when some small-file operations have been handled, starvation of large-file operations could be prevented given their enough early arrival time. Accordingly, we define *Filesystem Operation Priority* (FOP) as the following virtual completion time metric (the operation request with the *smallest* FOP value will be handled first).

$$FOP(x) = t_A^x + t_T^x, \forall\ operation\ x, \tag{6}$$

where t_A corresponds to the *arrival timestamp* attached in the operation request. Besides, t_T represents the file transfer time, which can be calculated with the estimated real-time bandwidths and other factors (*e.g.,* elimination ratio, delta computation rate) according to the mechanisms in §III-B. Note that the directory-related operations do not involve data transfer, so they can be simply and quickly finished by the controller. Correspondingly, HyCloud sets a fixed small t_T value for them, and thus these low-cost operations are most likely to be handled before file transfer operations. Among these operations, an FCFS (First Come First Served) priority assignment is adopted then.

In particular, common operations to the same file or relevant files/directories (*e.g.,* download and then delete a file, make a directory and then upload a file into it) are executed based on their request sequence to avoid potential logic errors or filesystem inconsistencies. An operation is executed only when the acknowledgment of its previous one is received by the controller (*i.e.,* the stop-and-wait ACK mode is adopted to these files and directories). Similarly, the controller can easily support more user-specified orderings (especially those crafted to conserve crash consistency [23]) by adding them as special cases of the priority assignment.

Based on the above execution priority assignment, relevant clients then adopt multiple threads for uploading large files to S3, and handles small files by one thread that successively transfers them to a special relay proxy. On the other hand, the controller schedules a relay proxy to tackle file download by sending notifications to its *Notification Queue*. Then the relay proxy adopts multiple handling threads to download the files, making the best of its available bandwidth. In both cases, the filesystem consistency can be guaranteed because multiple threads simultaneously accessing one file will be naturally prevented by the priority assignment of the above scheduling algorithm. It is worth noting that in the distributed cloud storage scenario, only relay proxies deployed in a client's service instance(s) are scheduled by the controller.

Adaptive Relay-Proxy Adoption: To balance the timeliness of bursty workloads and system overhead, we further design an adaptive relay-proxy adoption mechanism. A background thread monitors the real-time available bandwidth BW_a in each proxy and periodically fed back to the controller, to avoid congestion caused by too many concurrent file workloads. Relay proxies will be added for load balance when bandwidth shortage occurs in the working proxies. Specifically, whenever $BW_a \leq BW_o * \theta_c$ (where BW_o represents the initially measured overall bandwidth of the proxy and θ_c is a congestion threshold), another proxy is added to balance the load. Suppose that there are currently n relay proxies ($R = \{r_i\}$, $i = 1, 2, \cdots, n$), BW_o^i represents the initially measured overall bandwidth of each proxy and θ_c is a congestion threshold. To fully utilize the available bandwidth of the existing proxies, only when

$$BW_a^i \leq BW_o^i * \theta_c,\ \forall r_i \in R, \tag{7}$$

the proxy number increases to $n+1$. In addition, we also set a leisure threshold θ_l to save proxy resource. Here only when

$$BW_a^i \geq BW_o^i * \theta_l,\ \forall r_i \in R, \tag{8}$$

the proxy with lowest BW_o will be recycled after finishing the current file transfer (but not considered for workload dispatch of new files), and then the number of proxies decreases to $n-1$. We adopt such a lazy decrease mechanism since it can reduce the possibility of proxy number fluctuation. On this basis, the workload dispatch can be quite simple. The relay proxy with the highest BW_a among all in-use proxies is selected for the next file transfer workload. Note that the above proxy selection process is conducted by the controller based on relay proxies' available bandwidths piggybacked in *file transfer status* application-layer acknowledgments.

Local Redundant Request Elimination: To further boost the execution efficiency of filesystem operations, 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 controller's performance. 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.

Filesystem Consistency Maintenance: We need maintain filesystem consistency between a HyCloud instance and its clients in case of system failures. While a user's namespace path is hosted in EFS of the client's optimal service instance,[3]

[3]The client accesses the EFS filesystem through a special relay proxy deployed in the optimal service instance.

the client maintains the metadata of all files and directories in the path locally. In addition, a user may migrate to another geographic region (identified by the client's IP address [24]) and thus the optimal service instance changes. Migrating small files and directories from the original service instance to the new optimal one in the idle time can help reduce the user's access latency thereafter. For example, in Fig. 10 a user originally in user group A migrates to user group B, and thus filesystem migration is conducted between instances A and B. To support breakpoint-continued filesystem migration among instances, we should also maintain filesystem consistency between any two instances. To check the filesystem consistency in either case, we organize hash values of a user's all files and directories with a Merkle tree [25] and hierarchically exchange the hash values between the two sides.

Specifically, we generate an MD5 hash value for each file based on its file content and full path. On this basis, a directory's hash value is calculated by concatenating all the hash values of non-cascaded files and subdirectories. For instance, there are n files $\{f_i\}(i = 1, 2, \cdots, n)$ and m subdirectories $\{d_j\}(j = 1, 2, \cdots, m)$ in the directory d, and the hash values for each file and subdirectory are $Hash(f_i)$ and $Hash(d_j)$, respectively. Then the directory d's hash value can be recursively calculated by

$$H(d) = H(H(d_0) + \sum_i H(f_i)) + \sum_j H(d_j)), \qquad (9)$$

where "+" means concatenation and $H(d_0)$ is generated by the directory d's full path name ($H(d) = H(d_0)$, if d is an empty directory). When erasure coding is conducted to generate multiple blocks $\{b_k\}(k = 1, 2, \cdots, p)$ for a large file f, all the blocks' hash values instead of the file's hash value serve as leaves of the Merkle tree. In this case, each block's hash value $H(b_k)$ is concatenated to the large file's MD5 hash value $H(f_0)$ by sequence, *i.e.,* the file's hash value is specially calculated by

$$H(f) = H(H(f_0) + \sum_k H(b_k)). \qquad (10)$$

We denote either a client and an instance or two counterpart instances as A and B (*e.g.,* instances A and B in Fig. 10), and the hash value exchange process between them could be: A sends the root hash value to B, which compares the received value with that of its own Merkle tree. If the two values match, the two filesystems are consistent and thus the process terminates. Otherwise, B in turn sends the hash values of subdirectories to A for further checking. The similar steps are repeated until reaching the leaves of the Merkle trees. By this means, we can check out the inconsistent files within h steps, where h is the maximum height of the two Merkle trees and it is generally small for most users. For any inconsistent counterpart elements (directories, files or blocks), we further compares the respective *last-modified time* attributes, and then replace the old-version elements with the new-version ones or create/delete elements accordingly (facilitated by file deletion marking). Note that the above process is conducted in the idle time or when a system failure occurs, which would not affect the performance of filesystem operations.

Controller Selection and Maintenance: HyCloud utilizes a centralized controller to simplify the consistency maintenance, and we demonstrate by evaluation (refer to §IV-D) that a single controller is sufficient to tackle a large quantity of filesystem operation requests arriving in a short time. As the controller need execute filesystem operations on behalf of the users, it is expected to access the EFS filesystem of each HyCloud instance with low latency. Accordingly, we measure the variation of round trip time between each pair of HyCloud instances in n different AWS regions over a whole week and calculate the average results, denoted as $\{RT_{ij}\}$ $(i, j = 1, 2, \cdots, n)$, and then select a region to deploy the controller by calculating $\arg\min_i\{\max\{RT_{ij}\}\}$. To enhance the reliability of HyCloud, we also prepare an alternative controller and adopt it when failures occur to the in-used controller. We can deploy it as a *reserved instance* in another AWS region with the minimum round trip time among the rest regions (the two controllers are deployed in different regions to avoid simultaneous failures). The alternative controller also utilizes a *UserInfo Storage* database to store information of all clients.

When a controller failure (*e.g.,* power loss or kernel panic) occurs and thus the controller cannot be connected, the above filesystem consistency maintenance mechanism is adopted between each client and its optimal service instance's EFS, examining the inconsistencies by hierarchically exchanging hash values and then updating the old-version files and directories. By this means, there is no need to periodically conduct request backups to the alternative controller, and HyCloud is tolerant of the request loss caused by the controller failure. Afterwards, the clients' new operation requests are sent to the alternative controller and still handled by the aforementioned dataflow scheduling algorithm. When the suspended original controller can be reused, it notifies the in-used alternative controller, which then forwards the message to clients by attaching it to the responses. To avoid filesystem operation conflicts induced by simultaneous usage of both controllers, once receiving the message, a client caches the user's new operation requests in its *Local Operation Queue* until the previous requests are all acknowledged. Then the client starts sending operation requests to the original controller, and the cached requests are sent in a batch. The alternative controller is suspended after it finishes handling the existing requests.

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 HyCloud's storage cost-effectiveness as well as scalability (by adopting the above control scheme) with a large-scale data trace.

A. HyCloud Prototype

We have implemented a prototype of HyCloud framework in approximately 5000 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 and distributed manner. The source code is available at *https://github.com/iHyCloud/hycloud-demo*.

Particularly, we implement rsync [26]-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 the idle time following LRU (Least Recently Used) caching scheme. We implement erasure coding

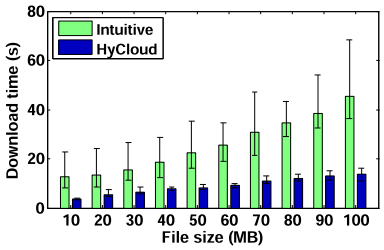

Fig. 12. Download time reduction with the relay-based mechanism for large files in different sizes.

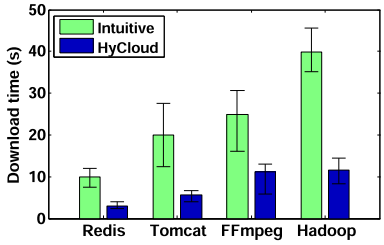

Fig. 13. Download time reduction for different versions of four common source codes.

with the zfec library [27] based on the $RS(3, 2)$ configuration. The controller and relay proxies interact with S3 and EFS by invoking their data APIs. 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, bandwidth feedback as well as workload dispatch relies on the interaction between the controller and a relay proxy with Apache MINA [28].

B. Experimental Setup

We deploy HyCloud instances in geo-distributed AWS regions with S3 and EFS. Relay proxies on Amazon EC2 *t2.micro* instances with 1 vCPU @2.5 GHz and 1 GB of memory (the configuration is low and cheap) are adaptively adopted in each region, where EFS is able to be directly mounted and S3 can be also efficiently accessed. According to our designed selection criterion, EC2 *t2.micro* instances in the AWS Oregon and Virginia regions serve as the in-used and alternative controllers respectively. We evaluate the file transfer mechanisms not involving geo-distribution based on the Oregon instance, which is overall the best among all instances on the aspects of both performance and cost.

According to the requirement of experiments, we adopt DigitalOcean VM nodes with the lowest configuration but unlimited network bandwidth in Singapore (SGP), London (LON) and Toronto (TOR) as HyCloud clients (the same as in §II-B), which can well simulate user PC performance and avoid local network congestion. Here the average bandwidth between a client and a relay proxy is ∼50 Mbps according to the real-world measurement. 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, we next take it as a baseline for most performance comparisons.

In addition, to calculate a convincing overall unit storage price and evaluate the scalability of HyCloud, we utilize a large-scale data trace of a former cloud service Xuanfeng. This trace was collected from 742,064 users with 3,412,827 files over a week, involving all kinds of filesystem operations. The major fields in each record include file metadata like hash value, file size, operation type, request and execution time, and the corresponding user client's IP address. 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. We further select records in the period [0:00, 0:10] of that day for the scheduling scalability evaluation. This 10-minute subset has totally 2002 files, which corresponds to a download burst.

C. Effectiveness of Advanced File Transfer Scheme

Relay-based file download 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). Note that the download data were not cached locally and do not contain any duplicated chunks here. The overall download time of each size is shown in Fig. 12 (their average S3-to-EFS transfer latencies are all shown in Fig. 6). We also show the performance of the intuitive approach (directly downloading the large files from S3) for contrast. As the figure illustrates, downloading files 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 file download 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 ∼7 MB, Tomcat ∼25 MB, FFmpeg ∼58 MB, Hadoop ∼100 MB). We download 10 latest versions of codes sequentially based the adaptively-adjusted delta encoding mechanism in HyCloud. Fig. 13 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 (∼10 seconds for the 100-MB file download as shown in Fig. 4).

We then 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. 14 shows the 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.

The transfer optimization of large-file upload is also evaluated with three source code files used above, *i.e.,* Redis, Tomcat, and FFmpeg. Here the intuitive approach (uploading the full file to S3) and the basic delta encoding (immediately applying delta to the old version file) serve as two comparisons. Fig. 15 shows their transfer time of 10 uploaded

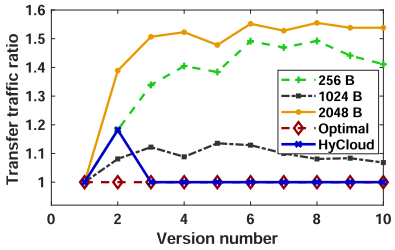

Fig. 14. Performance of adaptively-adjusted delta encoding mechanism (using one FFmpeg version at a time).

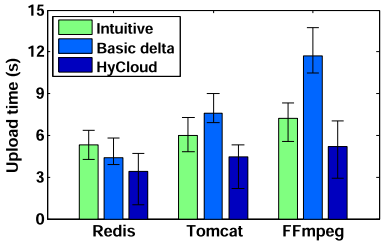

Fig. 15. Upload time reduction among versions of three common source codes.

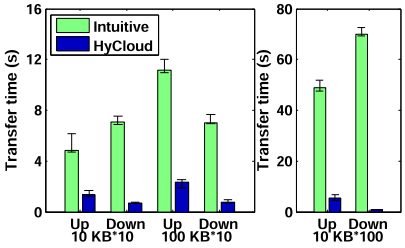

Fig. 16. Overall upload and download time of different batches of small files.

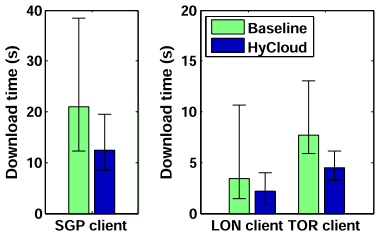

Fig. 17. Download time reduction with the multi-source download optimization algorithm.

versions. As shown in the figure, the transmission optimization mechanism of HyCloud has a significant improvement in the upload time of files. Compared with the best of the two comparisons, there is an average time reduction of 25.7%.

Besides large files, sometimes a user may intend to upload or download a folder with a number of small files. We next evaluate our small-file bundling mechanism by uploading and downloading typical batches of small files ($10 * 10$ KB, $10 * 100$ KB, and $100 * 10$ KB). Likewise, the intuitive approach (uploading or downloading small files to/from EFS successively) is used as a contrast. As shown in Fig. 16, despite performance disparities among different transfer scenarios, HyCloud always outperforms the contrastive scheme, with transfer time reduced 86.3% on average. The comparison confirms the effectiveness of the bundling mechanism in our transfer scheme.

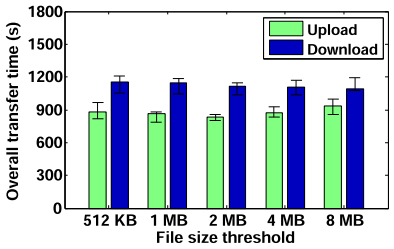

Fig. 18. Impact of file size threshold on the overall upload and download time of a number of files.

We further evaluate the multi-source file download algorithm designed for geo-distributed service instances. For the three testing clients mentioned in §IV-B, HyCloud instances in three AWS regions are selected as their service instances according to the deployment mechanism in §III-B. We measure the download time of a 100-MB file for each client (the generated blocks have been stored in the service instances in advance). Fig. 17 illustrates the performance of HyCloud in comparison with that of the *baseline* approach (downloading the whole file from the optimal service instance). The download time reduction is observed on all the three clients, among which the SGP client with high download latencies from all service instances experiences the most obvious reduction (up to 77.9% and 40.7% on average). The reduction mainly comes from HyCloud's higher overall transfer rate than that of the baseline given that only approximately twice the file workload (two of three blocks for each chunk) is downloaded from three sources in parallel, and both indicator computation and block decoding in the mechanism bring little extra time (less than 0.5 seconds in total). On the other hand, as the optimal service instance has to be determined based on the instances' link bandwidths before file download, its performance may not keep optimal throughout the download process, especially in the case that a large file is downloaded from a remote client.

D. Overall Indicators 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 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 1 GB are filtered out), and then evaluate their overall transfer performance by uploading and downloading them to/from S3 or EFS according to each threshold for 10 times. According to the results depicted in Fig. 18, we can set the file-size threshold as 2 MB for the 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 (described in §IV-B), 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 metadata is stored in EFS in all cases). The evaluation well shows the cost-effectiveness of HyCloud as a filesystem hosting service.

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

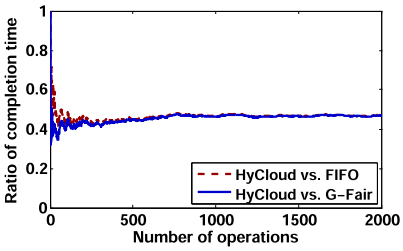

Fig. 19. Performance of HyCloud scheduling in comparison with typical scheduling algorithms.

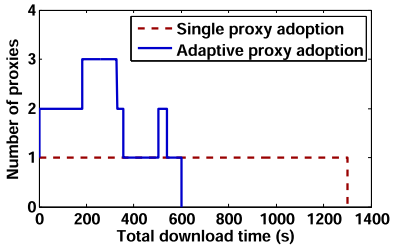

Fig. 20. Proxy number variation with the adaptive relay-proxy adoption mechanism.

On the other hand, we evaluate the scalability of HyCloud scheduling algorithm by executing a large number of file download operations selected from the data trace (described in §IV-B). To guarantee timeliness of operation executions, the average file transfer completion time is an important metric. Thus we focus on the ratios of HyCloud's average completion time to that of two contrastive algorithms FIFO and group-based fair sharing (*G-Fair*). Note that the G-Fair algorithm schedules a group of files (up to 10) concurrently to share the bandwidth resource, and the transfer bandwidth is around 50 Mbps as mentioned in §IV-B. Fig. 19 depicts the variation trends of two ratios as the number of file download operations increases. The ratios of HyCloud to both algorithms converge to around 50% soon and keep steady thereafter. The obvious promotion indicates that HyCloud can well guarantee the timeliness of a burst of file transfer requests.

According to the bandwidth measurement and data trace analysis, we set the congestion threshold θ_c and leisure threshold θ_l to be 20% and 60%, respectively. On this basis, we also evaluate the effectiveness of the adaptive relay-proxy adoption mechanism with the above data trace. Fig. 20 depicts how the number of relay proxies varies in a HyCloud instance as the file download operations are scheduled, which can well reflect the system overhead (the available bandwidth at a proxy is 100 Mbps to the maximum). In contrast with adopting a single relay proxy, the adaptive adoption mechanism can achieve 53.9% reduction in the total download time. More importantly,

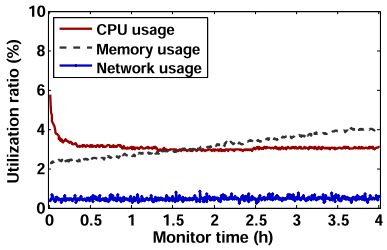

Fig. 21. Variations of the controller's resource utilization during the monitoring period.

HyCloud keeps quite low number of relay proxies adopted for workloads in the one-day data trace during the whole process (less than 2 on average and up to 3 only for some short bursty periods). Given the low price of EC2's compute resources ($0.0116/hour for each deployed VM instance), these relay proxies bring a marginal extra cost to the whole system – merely 0.21% of the storage cost.

Finally, we monitor the in-used controller (with 1 vCPU @2.5 GHz, 1 GB of memory and 500 Mbps of maximum bandwidth) when executing filesystem operations of 4 peak hours ([19:00, 23:00]) in the selected one-day data trace (refer to §IV-B). Specifically, there are 73,754 filesystem operations requested from 22,675 users, each of which is emulated by a distinct process on a nearby DigitalOcean or Aliyun ECS VM node with the lowest configuration (through identifying geolocation with the user client's IP address), and the concurrent user number is around 3,000 (up to 3,144) during the monitoring period. We measure variations of the controller's CPU, memory and network usages (sampling every 30 seconds). As shown in Fig. 21, the CPU, memory and network utilization ratios[4] are quite low and steady (all below 6%) confronted with intensive arrival of operation requests. Given the controller involves no data transfer and wisely adopts a priority queue for scheduling, deploying one controller is enough to handle concurrent operation requests from a large scale of users.

V. DISCUSSION

In addition to the above system design and implementation, we also discuss a few more relevant issues as follows.

System Generality and Productization: HyCloud achieves cost-efficient filesystem hosting based on hybrid cloud storage services, and we have implemented it atop Amazon S3 and EFS. It is worth mentioning that the system framework can also be applied to hybrid storage services of other popular public clouds, such as Azure Blob Storage and File Storage, Aliyun OSS and NAS. Especially, as the latency TIV phenomenon (illustrated in §II-B) universally exists, HyCloud's relay-based file download mechanism is suitable to the large-file download of these clouds. For instance, according to our real-world measurements, downloading a 100-MB file from Azure Blob Storage takes nearly 40 seconds on average; in contrast, when Azure File Storage works as a relay, the overall download latency (from Blob Storage to File Storage and then to the client) is only approximately 8 seconds. In addition, to be productized, HyCloud should be both cost-efficient and easy to use, to attract a multitude of users that overwhelms the overhead of storage cost and proxy maintenance.

[4]We combine the inbound and outbound traffic (21.9 and 9.5 KB/s on average during the monitoring period) for the network utilization calculation.

Security and Privacy Issues: Apart from the architecture, security and privacy are also often concerned for a filesystem hosting service, as data confidentiality of cloud storage becomes increasingly significant [29]. Currently, public cloud storage services such as AWS and Azure require users or third-party services to register and obtain a unique token before invoking various data APIs. In reality, a considerable number of users would like to thoroughly prevent the third parties like HyCloud or even cloud providers like Amazon from accessing their confidential data. An intuitive approach is that a user directly conducts asymmetric encryption with the RSA algorithm locally before uploading a file. However, this approach will largely increase the total upload time and impose significant computation overhead on the client, especially for large files. A seemingly feasible approach is to encrypt the file content with a symmetric encryption key and then encrypt the key with an asymmetric algorithm like RSA, which helps avoid performance penalty of encryption while achieving the same level of security. We will further explore more effective mechanisms to balance the security and computation overhead in the future work.

VI. 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], [30] and personal cloud services [7], [24]. Especially, some papers elaborately study the well-performed cloud services (like Azure [31], Dropbox [32], UbuntuOne [33] and OpenStack Swift [34]) by pinning their inside architectures, and even localize performance anomalies in Infrastructure-as-a-Service clouds [35]. 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 [36], MetaSync [37], CYRUS [38], UniDrive [39]), as well as enabled efficient cross-cloud file collaboration (*e.g.,* CoCloud [17]). Besides, some other studies leverage different cloud CDNs to reduce transfer latency [6], [40], [41] or provide cost-effective data placement [42], [43]. Unlike those studies binding multiple object storage services for distributed storage, HyCloud provides a cost-efficient filesystem hosting service by combining the advantages of both object storage and filesystem storage, and carefully tweaking their usages in a distributed manner.

Optimization in Data Transfer: There have been a number of relevant cloud storage techniques these years, like chunking [8], [9], [44], delta encoding and deduplication [26], [45], [46] as well as bundling [47], [48]. However, the APIs provided by Amazon S3 support none of the above techniques. Fortunately, EFS and the 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. In addition, by leveraging priority assignment that involves virtual completion time [49], [50], an online dataflow scheduling algorithm is devised to well balance their timeliness and system overhead.

VII. Conclusion

This article presents a cost-efficient filesystem hosting service through carefully tweaking the usages of Amazon S3 and EFS. We first reveal two key observations to address the large-file download bottleneck when combining the two storage services in an intuitive manner. Guided by the observations, we design enabling mechanisms of relay-based file download, adaptively-adjusted delta encoding, as well as additional file transfer optimization and geo-distributed instance deployment. We also devise an online dataflow scheduling algorithm and additional filesystem operation control mechanisms to boost timeliness while reducing system overhead. We put the above proposed techniques together to develop an open-source HyCloud prototype, which can offer low unit storage price (close to that of S3) and efficient filesystem operations (as quickly as in EFS) in a scalable way.

References

[1] *Aliyun OSS (Object Storage Service)*. Accessed: Sep. 3, 2020. [Online]. Available: https://www.aliyun.com/product/oss

[2] *Aliyun NAS (Network Attached Storage)*. Accessed: Sep. 3, 2020. [Online]. Available: https://www.aliyun.com/product/nas

[3] *The Open Group Base Specifications*, IEEE Standard 1003.1, 2008, no. 1. [Online]. Available: http://pubs.opengroup.org/onlinepubs/9699919799

[4] *Amazon Elastic File System (EFS)*. Accessed: Sep. 3, 2020. [Online]. Available: https://aws.amazon.com/efs/

[5] A. Li, X. Yang, S. Kandula, and M. Zhang, "CloudCmp: Comparing public cloud providers," in *Proc. 10th ACM SIGCOMM Conf. Internet Meas.*, 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. IEEE 35th Annu. IEEE Int. Conf. Comput. Commun. (INFOCOM)*, Apr. 2016, pp. 1845–1853.

[7] Z. Li *et al.*, "Towards network-level efficiency for cloud storage services," in *Proc. Conf. Internet Meas. Conf. (IMC)*, 2014, pp. 115–128.

[8] A. Muthitacharoen, B. Chen, and D. Mazières, "A low-bandwidth network file system," in *Proc. 18th ACM Symp. Operating Syst. Princ. (SOSP)*, 2001, pp. 174–187.

[9] B. Aggarwal *et al.*, "EndRE: An end-system redundancy elimination service for enterprises," in *Proc. USENIX NSDI*, 2010, pp. 419–432.

[10] *Amazon S3 Pricing*. Accessed: Sep. 3, 2020. [Online]. Available: https://aws.amazon.com/s3/pricing/

[11] *Amazon EFS Pricing*. Accessed: Sep. 3, 2020. [Online]. Available: https://aws.amazon.com/efs/pricing/

[12] *Amazon EC2 On-Demand Pricing*. Accessed: Sep. 3, 2020. [Online]. Available: https://aws.amazon.com/ec2/pricing/on-demand/

[13] *Copy-on-Write*. Accessed: Sep. 3, 2020. [Online]. Available: https://wikipedia.org/wiki/Copy-on-write

[14] *DigitalOcean*. Accessed: Sep. 3, 2020. [Online]. Available: https://www.digitalocean.com/

[15] C. Lumezanu, R. Baden, N. Spring, and B. Bhattacharjee, "Triangle inequality and routing policy violations in the Internet," in *Proc. Int. Conf. Passive Active Netw. Meas.*, 2009, pp. 45–54.

[16] G. Wu *et al.*, "On the performance of cloud storage applications with global measurement," in *Proc. IEEE/ACM 24th Int. Symp. Qual. Service (IWQoS)*, Jun. 2016, pp. 31–40.

[17] J. E., Y. Cui, P. Wang, Z. Li, and C. Zhang, "CoCloud: Enabling efficient cross-cloud file collaboration based on inefficient Web APIs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 1, pp. 56–69, Jan. 2018.

[18] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *J. Soc. for Ind. Appl. Math.*, vol. 8, no. 2, pp. 300–304, Jun. 1960.

[19] D. Karger *et al.*, "Web caching with consistent hashing," in *Proc. World-Wide Web Conf. (WWW)*, 1999, pp. 1203–1213.

[20] Y. Zhang and N. Duffield, "On the constancy of Internet path properties," in *Proc. 1st ACM SIGCOMM Workshop Internet Meas. (IMW)*, 2001, pp. 197–211.

[21] J. Jiang, V. Sekar, and H. Zhang, "Improving fairness, efficiency, and stability in HTTP-based adaptive video streaming with FESTIVE," *IEEE/ACM Trans. Netw.*, vol. 22, no. 1, pp. 326–340, Feb. 2014.

[22] J. Clausen, "Branch and bound algorithms-principles and examples," Univ. Copenhagen, Copenhagen, Denmark, Tech. Rep., 1999. [Online]. Available: https://imada.sdu.dk/~jbj/DM85/TSPtext.pdf
[23] J. Mohan, A. Martinez, S. Ponnapalli, P. Raju, and V. Chidambaram, "Finding crash-consistency bugs with bounded black-box crash testing," in *Proc. USENIX OSDI*, 2018, pp. 33–50.
[24] I. Drago, E. Bocchi, M. Mellia, H. Slatman, and A. Pras, "Benchmarking personal cloud storage," in *Proc. Conf. Internet Meas. Conf. (IMC)*, 2013, pp. 205–212.
[25] R. C. Merkle, "Protocols for public key cryptosystems," in *Proc. IEEE Symp. Secur. Privacy*, Apr. 1980, p. 122.
[26] A. Tridgell and P. Mackerras, "The Rsync algorithm," Joint Comput. Sci. Ser., Austral. Nat. Univ., Canberra, NSW, Australia, Tech. Rep. TR-CS-96-05, 1996.
[27] *Zfec 1.5.3–An Efficient, Portable Erasure Coding Tool*. Accessed: Sep. 3, 2020. [Online]. Available: https://pypi.org/project/zfec/
[28] *Apache MINA*. Accessed: Sep. 3, 2020. [Online]. Available: http://mina.apache.org/
[29] J. Tang, Y. Cui, Q. Li, K. Ren, J. Liu, and R. Buyya, "Ensuring security and privacy preservation for cloud data services," *ACM Comput. Surv.*, vol. 49, no. 1, p. 13, 2016.
[30] 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. ACM Conf. Internet Meas. Conf.*, 2013, pp. 177–190.
[31] B. Calder, J. Wang, A. Ogus, and N. Nilakantan, "Windows Azure Storage: A highly available cloud storage service with strong consistency," in *Proc. ACM SOSP*, 2011, pp. 143–157.
[32] I. Drago, M. Mellia, M. M. Munafo, A. Sperotto, R. Sadre, and A. Pras, "Inside Dropbox: Understanding personal cloud storage services," in *Proc. ACM IMC*, 2012, pp. 481–494.
[33] R. Gracia-Tinedo *et al.*, "Dissecting UbuntuOne: Autopsy of a global-scale personal cloud back-end," in *Proc. ACM IMC*, 2015, pp. 155–168.
[34] M. Ruan *et al.*, "On the synchronization bottleneck of OpenStack swift-like cloud storage systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 9, pp. 2059–2074, Sep. 2018.
[35] D. J. Dean, H. Nguyen, P. Wang, X. Gu, A. Sailer, and A. Kochut, "PerfCompass: Online performance anomaly fault localization and inference in infrastructure-as-a-service clouds," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 6, pp. 1742–1755, Jun. 2016.
[36] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa, "DepSky: Dependable and secure storage in a cloud-of-clouds," in *Proc. ACM EuroSys*, 2011, pp. 31–46.
[37] S. Han, H. Shen, T. Kim, A. Krishnamurthy, T. Anderson, and D. Wetherall, "MetaSync: File synchronization across multiple untrusted storage services," in *Proc. USENIX ATC*, 2015, pp. 83–95.
[38] J. Y. Chung, C. Joe-Wong, S. Ha, J. W.-K. Hong, and M. Chiang, "CYRUS: Towards client-defined cloud storage," in *Proc. ACM EuroSys*, 2015, pp. 1–16.
[39] H. Tang, F. Liu, G. Shen, Y. Jin, and C. Guo, "UniDrive: Synergize multiple consumer cloud storage services," in *Proc. ACM Middleware*, 2015, pp. 137–148.
[40] Z. Wu, C. Yu, and H. V. Madhyastha, "CosTLO: Cost-effective redundancy for lower latency variance on cloud storage services," in *Proc. USENIX NSDI*, 2015, pp. 543–557.
[41] F. Chen, K. Guo, J. Lin, and T. La Porta, "Intra-cloud lightning: Building CDNs in the cloud," in *Proc. IEEE INFOCOM*, Mar. 2012, pp. 433–441.
[42] 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. ACM 24th ACM Symp. Operating Syst. Princ.*, 2013, pp. 292–308.
[43] G. Liu and H. Shen, "An economical and SLO-guaranteed cloud storage service across multiple cloud service providers," in *Proc. IEEE 35th Annu. IEEE Int. Conf. Comput. Commun. (INFOCOM)*, Apr. 2016, pp. 2689–2697.
[44] N. T. Spring and D. Wetherall, "A protocol-independent technique for eliminating redundant network traffic," in *Proc. Conf. Appl., Technol., Architectures, Protocols Comput. Commun. (SIGCOMM)*, 2000, pp. 87–95.
[45] Y. Hua, X. Liu, and D. Feng, "Neptune: Efficient remote communication services for cloud backups," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Apr. 2014, pp. 844–852.
[46] Q. Zhang *et al.*, "DeltaCFS: Boosting delta sync for cloud storage services by learning from NFS," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jun. 2017, pp. 264–275.
[47] Z. Li *et al.*, "Efficient batched synchronization in dropbox-like cloud storage services," in *Proc. ACM Middleware*, 2013, pp. 307–327.
[48] Y. Cui, Z. Lai, X. Wang, N. Dai, and C. Miao, "QuickSync: Improving synchronization efficiency for mobile cloud storage services," in *Proc. ACM MobiCom*, 2015, pp. 592–603.
[49] A. Ghodsi, V. Sekar, M. Zaharia, and I. Stoica, "Multi-resource fair queueing for packet processing," in *Proc. ACM SIGCOMM Conf. Appl., Technol., Architectures, Protocols Comput. Commun. (SIGCOMM)*, 2012, pp. 1–12.
[50] C. Zhang, Y. Cui, R. Zheng, J. E, and J. Wu, "Multi-resource partial-ordered task scheduling in cloud computing," in *Proc. IEEE/ACM 24th Int. Symp. Qual. Service (IWQoS)*, Jun. 2016, pp. 1–6.

Jinlong E received the B.E. and M.Sc. degrees in computer software from Nankai University in 2011 and 2014, respectively, and the Ph.D. degree in computer science and technology from Tsinghua University in 2018. Then, he worked as a Research Fellow with Nanyang Technological University until 2019. He is currently a Post-Doctoral Researcher with the School of Software, Tsinghua University. His research interests include cloud storage, mobile/edge computing, and big data analysis.

Yong Cui (Member, IEEE) received the B.E. and Ph.D. degrees from Tsinghua University, China in 1999 and 2004, respectively. He is currently a Full Professor with the Department of Computer Science and Technology, Tsinghua University. He has published more than 100 academic articles in refereed journals and conferences. He has authored a number of RFCs. His major research interests include mobile/wireless Internet and network architecture. He received two best paper awards. He also serves as the Co-Chair of *IETF IPv6 Transition Softwire WG*.

Zhenhua Li (Member, IEEE) received the B.Sc. and M.Sc. degrees from Nanjing University in 2005 and 2008, respectively, and the Ph.D. degree from Peking University in 2013, all in computer science and technology. He is currently an Associate Professor with the School of Software, Tsinghua University. His research interests include cloud computing/storage, big data analysis, content distribution, and mobile Internet.

Mingkang Ruan received the B.Sc. degree from Sun Yat-sen University, Guangzhou, China, in 2014, and the M.Eng. degree from Tsinghua University, Beijing, China, in 2018, all in software engineering. His research interests include cloud computing/storage, big data analysis, and natural language processing.

Ennan Zhai received the Ph.D. degree from Yale University in 2015. He is currently a Staff Engineer and a Researcher with the Alibaba Group. Prior to joining Alibaba, he was a Research Scientist and a Lecturer with Yale University. Specifically, his work takes advantage of an interdisciplinary approach, integrating areas including verification, programming languages, and security. He has published more than 20 articles in top-tier conferences, such as SIGCOMM, OSDI, NSDI, CAV, and VLDB. His research interests include building reliable and secure networking systems.