DCStore: A Deduplication-Based Cloud-of-Clouds Storage Service

***Abstract*—The increasing popularity of cloud storage is leading many organizations to move their data into the cloud. However, putting all data in one cloud causes problems such as vendor lock-in, increased service costs, and data availability. In this paper, we introduce DCStore, a Cloud-of-Clouds storage service designed for an organization to outsource their data into the clouds. To achieve the goal of cost-efficient and high-available, we combine three key techniques. First, DCStore eliminates the redundant data at client-side to save storage cost via application-aware chunking method. Second, DCStore uses an inner-chunk based erasure coding scheme to distribute unique chunks across multiple clouds for high availability. Finally, a container-based share management strategy is used for performance optimization. Our experimental evaluations show that DCStore can improve the performance and cost efficiency significantly, compared with existing Cloud-of-Clouds storage systems.**

***Index Terms*—Cloud Storage Service; Data Deduplication; Erasure Coding; High Availability; Cost Efficiency**

I. Introduction

The increasing popularity of cloud storage is leading many organizations to move their own data into the cloud. Typical usage examples include off-site backup, storing digital media and content distribution. Even organizations that handle critical data, such as medical record and financial data, are adopting cloud computing as a way to reduce costs [1]. With such large amounts of critical data stored in the cloud, data availability and storage cost have become two major concerns for an organization. Although cloud service providers abide by strict service-level agreements (SLAs) with impressive up-times and response delays, experience tells us that even the best providers suffer from occasional outages, which can result in severe financial losses [2]. Besides, the cloud service providers always offer different prices for market competition [3]. Therefore, it is wise for customers to leverage multiple cloud storage services, called Cloud-of-Clouds, to improve data availability and minimize cost.

In a Cloud-of-Clouds storage system, data is distributed across multiple clouds to achieve the goal of high-available and cost-efficient. Reliability-enhanced technologies (e.g., replication or erasure coding) are first used so that data can be recovered from a subset of clouds even if the remaining clouds are unavailable. For example, DuraCloud [4] utilizes replication to achieve the goal of availability at very high storage overhead. Some recent studies (e.g., RACS [5] and NCCloud [6]) leverage erasure coding to tolerate cloud failures with much less storage overhead. After that, some algorithms leveraging the cloud pricing policies and workload characteristics are applied for cost optimization [7], [8]. Besides, previous studies also have shown that data redundancy is moderate to high in the storage system for an organization. For instance, about 69%-97%, 42%-68%, and 20%-30% of the data in secondary storage, primary storage, and HPC data centers, respectively, are redundant [9]. Since those unnecessary redundancies incur additional cost, organizations can benefit a lot from client-side data deduplication before outsourcing their data into the clouds.

However, it is not easy in real life. Several factors impede this goal. First, data deduplication technology splits large files into a sequence of small chunks, which leads to degraded Gets/Puts performance and additional expense due to the extra network I/Os. For example, Amazon S3 has both a per-request and a per-byte cost when storing a file. If an 8MB file was split into chunks with an average size of 8KB, the request costs will increase 1,000 times. Our experiments in Section IV-D also show that the access latency for an 8MB file will increase nearly 20 times due to the extra network I/O.

Second, Data deduplication reduces the reliability of the storage systems because the loss of a few critical data chunks can lead to many referenced files being lost [10], [11]. While applying erasure coding after deduplication improves data availability, existing works [12], [13] that operate erasure encoding on multiple fixed-size objects are inefficient. For example, current approaches first pack varied-size chunks into fixed-size objects, then encode k objects to generate r additional objects. If a chunk is missing, reconstruction using parities incurs huge bandwidth overheads [14]. This problem will be specified in Section II-B. In addition, a cloud failure is transient [15]. Repairing all the data during failures may waste money since the failed cloud will rejoin the system with contents intact sooner or later.

To handle the above-stated two challenges, we design and implement a Cloud-of-Clouds storage service called DCStore. DCStore is designed for an organization to outsource a large group of users' data into multiple clouds with cost efficiency and availability guarantees. It combines three main techniques: client-side data deduplication, inner-chunk encoding, and container-based share management strategy. Our experimental evaluations show that DCStore can improve the performance and cost efficiency significantly, compared with some existing Cloud-of-Clouds storage systems (i.e., DAC [12] and CYRUS [16]).

In designing and evaluating this system we make three contributions. First, through trace-driven simulation, we show that it is possible for an organization to minimize cost and tolerate outages using data deduplication and erasure coding

techniques. Second, we propose an inner-chunk based erasure coding scheme and use on-demand chunk reconstruction to reduce storage overhead upon failures. Third, we design and implement a container-based share management strategy to aggregate small data shares into a single larger unit for performance optimization.

The remainder of this paper is organized as follows. In the next section, we provide the necessary background to motivate our study. In Section III, we describe the system architecture and design details of DCStore. We evaluate DCStore on its prototype implementation with real datasets, by comparing it with the existing state-of-the-art schemes in terms of performance and storage cost in Section IV. We present the related work in Section V and conclude this paper in Section VI.

II. Background and Motivation

In this section, we present some necessary background, including data deduplication and erasure coding, to motivate our study.

A. Data deduplication

Data deduplication is an efficient data reduction technology to reduce network bandwidth and storage overhead by eliminating duplicate data. As shown in Figure 1, a typical data deduplication system splits the input data stream into multiple chunks identified with a unique hash identifier. Deduplication systems then remove duplicate data chunks by those identifiers and store or transfer only one copy of them to achieve the goal of saving storage space or network bandwidth.

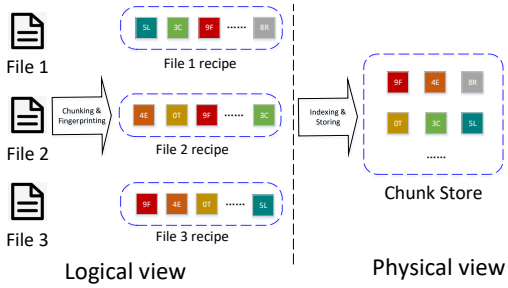


Fig. 1. Overview of deduplication processing

Two general approaches can be used to split the input file. The first one is called static chunking and splits the data into chunks of a fixed size. The main advantage of this technique lies in the fact that it offers a very high throughput and is easy to implement. While being successfully used by Dropbox [17], this approach comes with a major drawback: when bytes are added at the beginning of a file, all the following chunk boundaries are shifted [18], i.e., all chunks after the addition will have different hash sums and will become obsolete for that file. The second chunking approach is called content-defined chunking (CDC) and eliminates this problem. It was proposed by Muthitacharoen et al. for the Low Bandwidth File System (LBFS) [19]. With content-defined chunking, the system moves a sliding window over the data and hashes the windowed data in each step, using Rabins fingerprinting method. A new chunk boundary is found if the hash value satisfies some pre-defined condition.

However, the chunking methods discussed above do not utilize file characteristics of the underlying data. If the chunking method understands the data stream of the file (format of the file), the deduplication method can provide the best redundancy detection ratio and throughput because this method could set boundaries more natural than other algorithm methods [20]. Thus, DCStore uses an application-aware chunking method to make a better trade-off between deduplication ratio and deduplication throughput.

B. Erasure Coding

Erasure coding has been applied in many large-scale distributed storage systems, including storage systems at Facebook and Google. In a Cloud-of-Clouds storage services, erasure coding allows client to mask temporary outages and get higher data availability by adding chunk redundancies across multiple providers. As shown in Figure 2, a (k, r) erasure coding encodes k data blocks and generates r parity blocks such that any k of the (k + r) total blocks are sufficient to decode the original k data blocks. In many storage systems [13], [21]–[23], erasure coding is applied across fixed-size objects: k objects are encoded to generate r additional objects. Read requests to an object are served from the original object unless it is missing. If the object is missing, reconstruction using parities incurs huge bandwidth overheads [14]. Besides, deduplicated storage systems need pack varied-size chunks into fixed-size objects, which may cause zero-byte padding (e.g., to pad zero-bytes at the end of each object to make everyone equally-sized) thereby making the space utilization low [24].

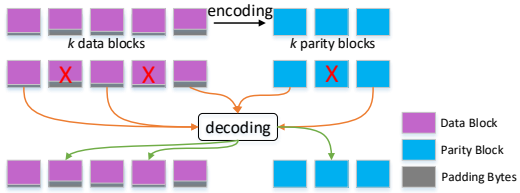


Fig. 2. Overview of Erasure Coding

In contrast, DCStore uses a inner-chunk based erasure coding scheme, which divides individual chunks into k fixed-sized data blocks and creates r additional data blocks like Per-file RAID [15]. Read requests to an chunk are served by reading any k of the (k+r) splits and decoding them to recover the desired chunk. This approach provides two benefits which help customers save unnecessary expenses. First, the space overhead consumed by the zero-byte padding is reduced. Then, decoding the chunk using the parities upon a cloud failure does not incur any additional bandwidth overhead. It is also worth noting that the computational overhead of erasure coding is negligible compared to the overhead of reading, writing or sending data in the WAN.

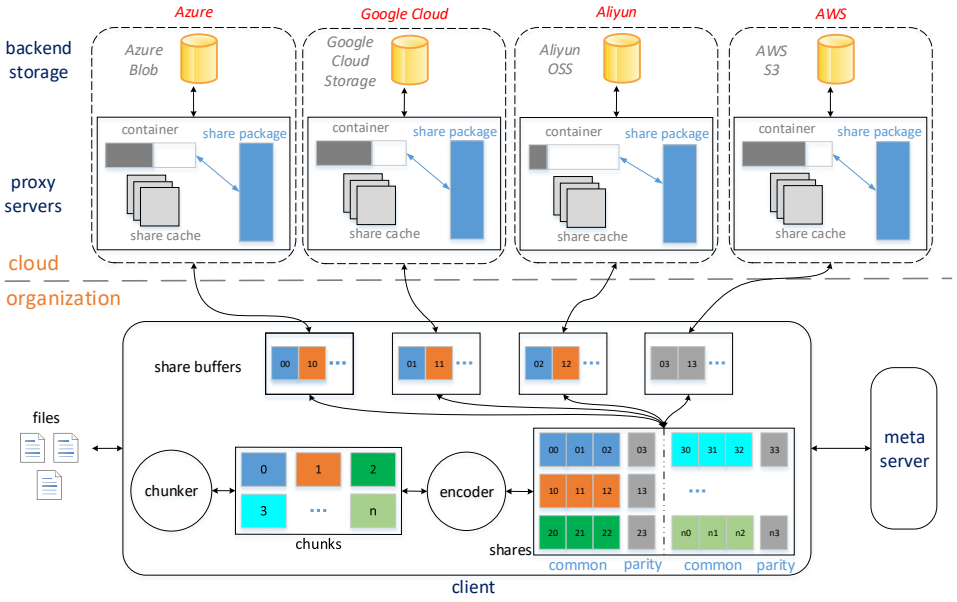


Fig. 3. System Architecture

III. The Design of DCStore

In this section, we introduce the DCStore architecture overview, some design details and the prototype implementation.

A. Architecture Overview

We follow a modular approach to implement DCStore, whose client and server architectures are shown in Figure 3. On the organization side, the DCStore client is responsible for data chunking, chunk encoding/decoding, and file recipe collection. The file recipe from the client will serve as input for the metadata server to maintain index information in a LevelDB. On the cloud side, a co-locating proxy server in each clouds is in charge of data transferring between the DCStore client and the corresponding cloud storage backend.

When uploading a file, the DCStore client first splits the file into chunks via the chunker module using an application-aware chunking method and avoid uploading redundant chunks by checking the metadata server. New Chunks are divided into shares through (n, k) erasure coding via the encoder module. Then, the DCStore client batches the shares in a client buffer and uploads it to the corresponding proxy server as a whole package. Upon receiving the shares package, each proxy server packs the unique shares into an open container. Finally, the container module writes the container to the cloud storage backend through the internal network when it is full.

Downloading a file is much easier. The DCStore client first retrieves file recipe from the metadata server. Then the client selects any k of n cloud storage services and downloads the missing shares through proxy servers. In DCStore, upon receiving client requests, each proxy server retrieves the corresponding containers and returns all required shares together. Finally, the DCStore client decodes the shares and assembles the chunks back to the file.

B. Application-Aware Chunking

The chunking module is implemented in the DCStore client. To make a better trade-off between deduplication ratio and deduplication throughput, we choose application-aware chunking. Currently, our implementation divided files into two main categories: static files and dynamic files. The dynamic files are always editable, while the static files are uneditable in common. We split static files into fix-sized chunks with ideal chunk size, and break dynamic files into variable-sized chunks with optimal average chunk size using CDC based on the Rabin fingerprinting.

C. Inner-Chunk Based Erasure Coding

After chunking, the encoder module divides each unique chunk into k same-sized data shares and encode them into n-k parity shares through (n, k) erasure coding. These n shares are stored on different cloud service providers so that we can achieve high availability and resist against possible failures of particular providers. Whenever the DCStore client requests a missing chunk, we only need to get any k parts shares concurrently and reconstructs the chunk in the client.

We define r = n/k as the redundancy rate. A higher degree of redundancy rate comes with a higher cost. A recent survey of the cloud service outages indicates that two concurrent cloud outages are extremely rare [25], so we fix n - k = 1. On the other hand, increasing k leads to large numbers of small shares. Since the average chunk size is always several KB (e.g., 8KB), after k increase beyond a threshold, the large number of concurrent share retrieval processes becomes the bottleneck. So we employ Reed-Solomon (RS) code, with n = 4 and k = 3 as a default setting. Nevertheless, it must be noted that

the degree of redundancy rate is configurable to satisfy the requirements of different organizations.

D. Container-Based Share Management

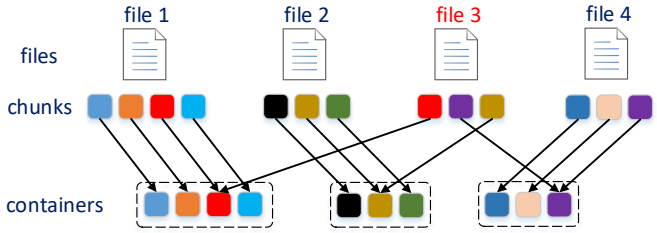


Fig. 4. Network traffic Overhead

As described before, data deduplication technology splits large files into small chunks, which leads to degraded Gets/Puts performance and additional request cost. Some recent studies pack those small data pieces into larger storage units at client-side could avoid high I/O overhead [16]. But it is still not enough. Since cloud storage services today only support limited operations (e.g., download the whole storage object, cannot download part of it), this may incur much more network traffic for a client to get a file. For example, as shown in Figure 4, file 3 has many redundant chunks which stored in containers generated by other files. To get file 3, the client has to get all the containers from the clouds while most of the chunks in those containers are totally unnecessary.

So we employ a co-locating virtual machine at each cloud as a proxy server. To upload a file, a DCStore client first batches shares in a client buffer and uploads the buffer to the proxy server when it is full. An open share container is maintained in each proxy server. When a container fills up with a predefined fixed size (e.g., 8MB), the container module writes it to the cloud storage backend through the internal network. This process reduces the extra network I/Os and takes advantage of share spatial locality so that the data restoration performance will be reasonably good.

When downloading a file, a client must choose k of the n cloud service providers from which to download shares. Since there is a huge variance among the performance and the cost of different cloud providers, random picking clouds may lead to longer file retrieve time and incur more cost. So we sort the clouds based on the access monetary cost and try to access them in parallel. This optimization further decreases the monetary cost of DCStore. Then, each proxy server retrieves the corresponding containers, and only returns the required shares to the client. Finally, the DCStore client decodes the shares and assembles the chunks back to the file. We also maintain a least-recently-used cache for the most recently accessed shares to optimize performance further.

Since there is no billing for the high-speed network between the co-locating proxy server and the cloud backend storage service, we argue that our approach incurs limited overhead in expense and performance compared with the benefits it brings.

E. Metadata Management

Currently, the metadata module is located in a dedicated server owned by the organization for security and performance reason. It includes four file-specific tables:

1) file_info_table: This table records the metadata of a file and a reference to the file recipe, which describes the complete details of a file.

2) chunk_info_table: This table describes chunks' information, such as chunks' fingerprint, size, and referenced shares.

3) share_info_table: This table maps each share's fingerprint to the container which contains the share.

4) container_info_table: This table stores each container's size and the information of stored shares.

In addition to those file-specific tables, a global chunk index is maintained. To support chunk deduplication, this chunk index holds the entries for all unique shares of different files. When uploading a file, metadata module will use the information from the DCStore client to update the corresponding tables.

F. Failure Detection and Recovery

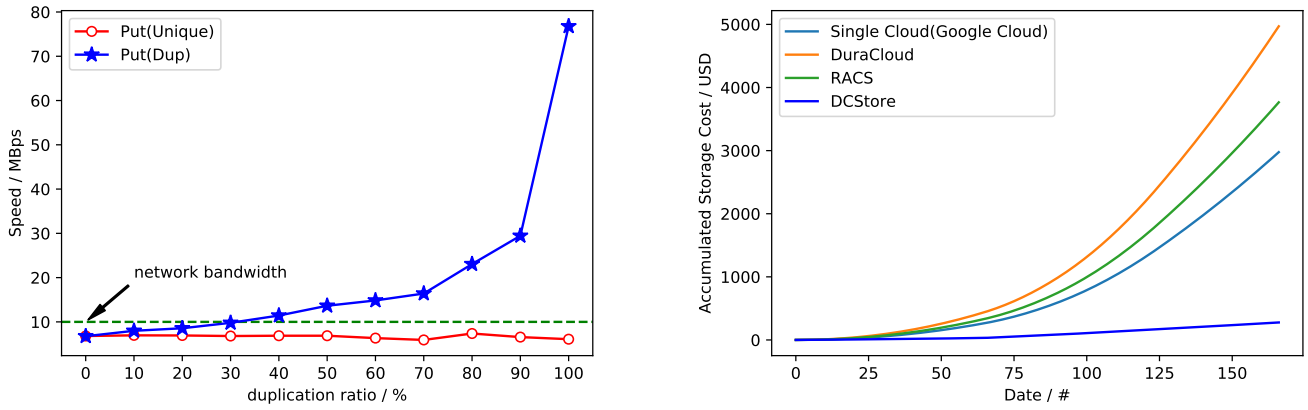
A cloud failure is quite different from a disk failure, so a lazy way of recovery is applied in DCStore. First of all, the proxy server detects a cloud failure if it fails to upload containers. Once this happens, the proxy server periodically checks if the failed cloud comes back. During the failure period, all the get/put operations are performed as usual. For the get operations, the missing shares will be reconstructed using the erasure coding redundancy while the unrequested shares remain unchanged. As we use an inner-chunk based erasure coding scheme, decoding the chunk using the parities upon a cloud failure does not incur any additional bandwidth overhead. Those reconstructed shares are cached in the corresponding proxy server using an LRU algorithm. For the put operations, the client will succeed as long as it has written to the majority of cloud storage services successfully. But this failure will be recorded in a recover log. Upon the cloud recovers, the proxy server will reconstruct the missing shares and write it back to the recovered provider. When the logs are completely processed, the recovery process completes.

G. Prototype Implementation

The DCStore prototype is written in 2500 *LOC* python on Linux. We use *fastchunking* [26] and *zfec* [27] for generating file chunks and encoding chunks into shares, respectively, and SHA-256 for the fingerprints in deduplication. We use *multiprocessing* [28] lib to accelerate uploading and downloading of the shares.

IV. Evaluation

We evaluate the performance and the cost advantages of DCStore by both running real-world dataset and simulating on trace dataset.



(a) Performance Improvement

(b) Storage Cost Saving

Fig. 5. Deduplication Benefits.

A. Experimental Setup

Client. Our experimental client runs in a virtual machine on *Aliyun*, which has a dual-core 2.50GHz Intel Xeon Platinum 8163 processors, 8GB RAM and a 40GB HDD. This virtual machine's bandwidth is limited to 100Mbps to simulate the network bandwidth between an organization to the clouds. The machine runs *Ubuntu 16.04*. Noted that the DCStore client could be deployed on any Linux-based machines, and it does not matter where the machine is.

Cloud-Side Proxy Servers. The proxy server in the cloud-side have the similar configuration as the client machine, which has a dual-core processor, 8G RAM and a 400GB HDD.

Cloud Storage Services. In our experiments, we choose *Amazon S3*, *Aliyun OSS*, *Microsoft Azure Blob* and *Google Cloud Storage* as the backend storage services.

Datasets. We use both real-world benchmark files and open-source trace datasets to evaluate DCStore.

1) Random Generated Benchmark Files: We generate eleven sets of files with different duplication ratio from *0%* to *100%*. All files have 2.5e7 lines and each line has ten integer numbers which follow a Gauss distribution. For files with a duplication ratio *r%*, they share exactly the same $r\% \times 2.5e7$ lines. We also generate a set of smaller files ranging from *1MB* to *10MB* for the performance evaluation, whose usage will be detailed in Section IV-D.

2) FSL Trace Dataset: We use the FSL dataset [29] to simulate DCStore's cost-saving effect. This dataset is published by the File systems and Storage Lab at Stony Brook University. Due to the large dataset size, we use the *Fslhomes* dataset in 2014, containing daily snapshots of 17 students' home directories from a shared network file system. The dataset is represented in 48-bit chunk fingerprints and corresponding chunk sizes obtained from content-defined chunking.

B. Benefits of Deduplication

In this section, we evaluate both the performance and cost benefits brought by deduplication.

1) Performance Improvement: *Put* and *Get* are the two main operations in DCStore, among which *Put* is the one that is relevant to deduplication while *Get* is not. This is because the corresponding content must be fetched from the clouds whether the redundant content is removed or not. The benchmark files with different duplication ratio mentioned in Section IV-A1 is used in this experiment. We first upload a file *a.r.csv* and then upload a file *b.r.csv* as the *r* goes from *0%* to *100%* (*r* represents the duplication ratio). We repeat this process 10 times and take the average throughput as the final metrics. Noted that after the uploading of a pair of *r%*-dup files, we clear the metadata and the shares stored in the clouds to make sure there is no interference between files with different duplication levels.

The experimental result is shown as Figure 5(a). As for a redundant level *r%*, we call the first-uploaded file the unique-file and the second-uploaded file the dup-file. The green horizontal line in the figure represents the upper bound of the network bandwidth, which is about 100Mbps. The red line represents the upload speed of unique files, which can achieve about 70Mbps, *70%* of the bandwidth. As the duplication ratio goes higher, the uploading of the dup-file becomes faster. When the duplication ratio comes to *30%* or higher, the speed can become faster than the native bandwidth. When the duplication ratio is *100%*, which means uploading two files with identical content, the speed can achieve nearly 800Mbps. Since no new content needs to be delivered to the cloud, the transfer speed at the *100%* point can represent the speed of the client-side chunking and encoding process.

2) Storage Cost Saving: To evaluate the cost-saving benefits brought by deduplication, we run a simulation based on the FSL trace mentioned in Section IV-A2. We combine all the users' trace data together, which is about 1.41TB in 168 days totally. From day #1 to day #168, we simulate the upload process for the modified files according to four different strategies, which are used by Single Cloud, DuraCloud, RACS, and DCStore, respectively. The first three strategies simply

upload the whole file when its modify-timestamp changes while DCStore only uploads the changed chunks. We choose Google Cloud as the Single Cloud instance, which is the cheapest among the four cloud vendors. DuraCloud simply stores two copies of a single file. RACS do a (3,1) erasure-encoding for all the files without chunk-level deduplication.

The simulation result is shown in Figure 5(b). The vertical axis is the accumulated cost. DuraCloud stores the most data so it costs the most, about 4970USD on the last day. RACS also provides high availability but still costs too much, about 3765USD on the last day. Storing all the data in the cheapest cloud costs lower than the former two, about 2975USD on the last day. But it may suffer from cloud outages. The blue line at the bottom is DCStore's cost. DCStore removes redundant data via chunk-level deduplication and provide high availability by performing erasure-encoding on chunks. The cost of DCStore is the lowest among these four strategies, about only 277 USD on the last day.

When the size of total stored data fixed to 1.41TB, DCStore incurs additional VM costs of around 1000USD in 168 days but reduces the storage cost to around 277USD and thus achieving about 57% of cost savings as a whole. Noted that the trace dataset on which we perform the simulation comes from a small research group while the real world data size can be rather huge. The cost-saving effect of DCStore can be more significant as the data size goes larger.

C. Benefits of Inner-Chunk Based Erasure Coding

As mentioned in Section II-B, we adopt an inner-chunk based erasure coding scheme to take advantage of its small storage and recovery overhead. Existing methods usually pack the chunks into fixed-size data blocks in client-side and apply erasure coding across data blocks to generate the parity blocks (We called inter-encoding). In contrast, DCStore divides individual chunks into k fixed-sized data blocks and creates r additional parity blocks. When the cloud fails, the client only needs to get the corresponding shares to decode the missing chunks while inter-encoding method has to retrieve larger data blocks. Besides, assume the file size, and the average chunk size are represent as $file\_size$ and $chunk\_size$. If we use inner-chunk encoding, and suppose there are n shares after each chunk is encoded, the zero-padding size can be estimated approximately according to (1). Let $chunk\_size$ and n be 128KB and 4, a 1GB file will be padding about 16KB useless zeros using inner-chunk encoding.

$$zero\_padding = \frac{file\_size}{chunk\_size} \times \frac{n}{2} \tag{1}$$

Apart from the above mathematical analysis, we also run a simulation based on the FSL trace dataset. We select a user's backup data trace in a single data, which is about 22GB totally. We let the size of client-side data blocks be 3 common-used values which are 4MB, 8MB and 16MB. And we set the encoding parameter n as 4. The simulation result is shown in Figure 6. The inter-encoding method incurs 374MB, 209MB and 115MB zero-padding overhead as the container size varies from 4M to 16MB while the inner-encoding method incurs only 520KB zero-padding. Thus the inner-chunk encoding method can be much more storage-saving than the existing approach.

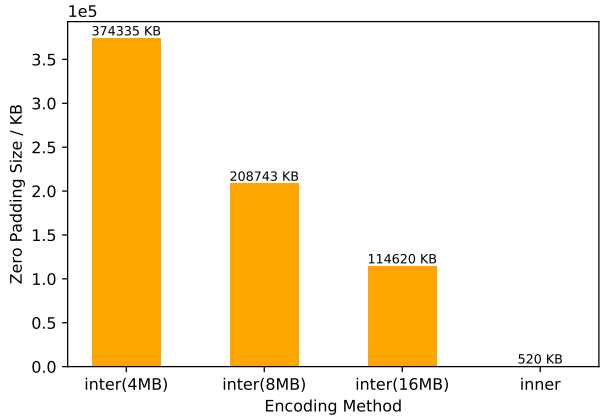


Fig. 6. Zero Padding Overhead Comparison of Inter/Inner Encoding.

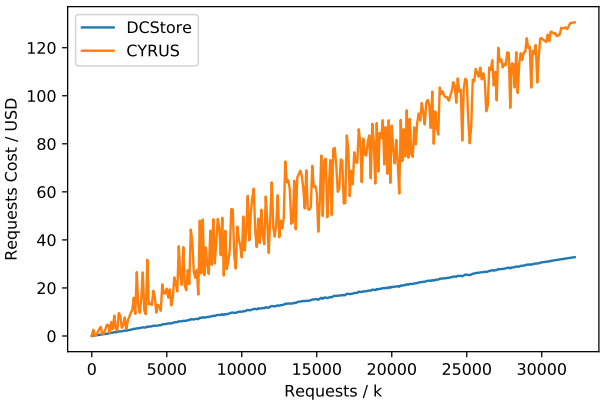
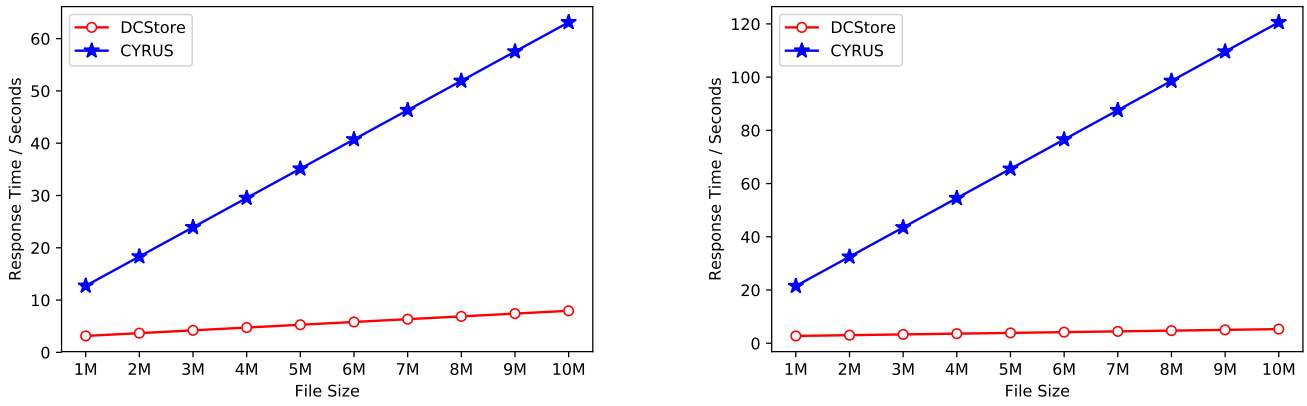


Fig. 7. Request Cost Saving by Container Strategy

D. Benefits of Container Strategy

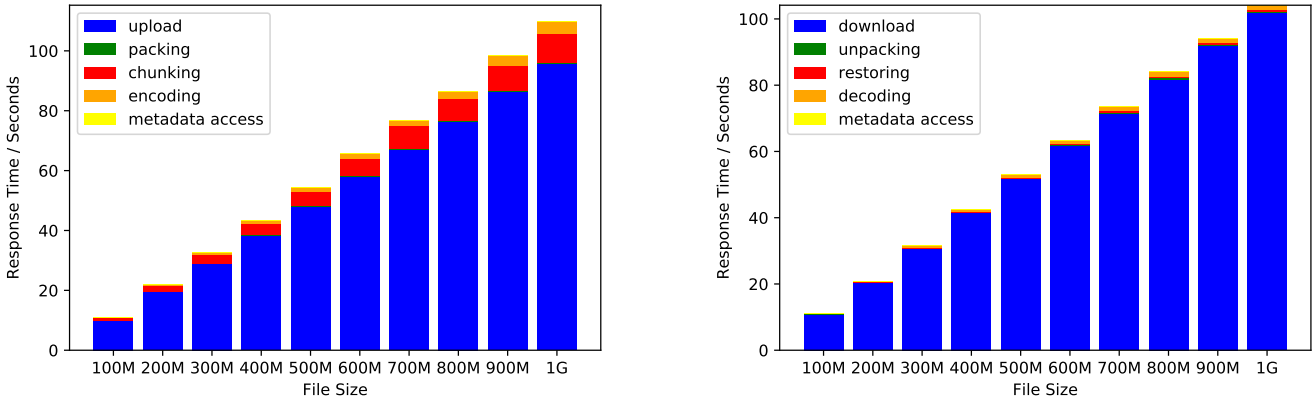
In this section, we evaluate the benefits of container strategy detailed in Section III-D by running two experiments. We implement CYRUS's strategy for comparison, which transfers all the shares to or from the clouds without packing them together during the *Put* or *Get* operation. In fact, many other storage systems adopt the same policy as CYRUS, such as DAC, etc.

1) Put/Get Performance Improvement: First, we test the performance improvement for the *Put*/*Get* operation. We use the smaller files mentioned in Section IV-A1. For each file, we successively run a *Put* and a *Get* using DCStore and CYRUS strategies respectively 10 times. When running *Put*, DCStore waits until the client-side buffer is full and packs the buffer to a container while CYRUS simply sends all the shares to the cloud one by one. When running *Get*, the proxy servers fetch



(a) Put (b) Get

Fig. 8. Response time Comparison with/without Container Strategy.



(a) Put (b) Get

Fig. 9. Response Time Distribution of Put/Get.

all the containers that contain the needed shares and send the shares back to the client together while CYRUS simply fetches all the needed shares without packing. As shown in Figure 8, the response time of CYRUS is much longer than DCStore, which is mostly because of the frequent HTTP requests for small chunks. For example, when the file is 5M and the chunk size is set to 128K, over 40 chunks and 160 shares will be generated by the (3, 1) erasure coding. If there is no container strategy, over 160 HTTP requests will be launched. But with the container strategy, four HTTP requests at the client will be enough to transfer all the shares to or from the clouds.

2) Request Cost Saving: Second, we evaluate the request cost saving benefits of DCStore by running a simulation on the FSL trace. We randomly choose a certain number of files from all the users' data of a single day, and perform 1k requests to them. As for CYRUS, all the shares is requested. While for DCStore, the shares in the same container are requested only once by the proxy server.

The simulation result is shown in Figure 7. The orange line has a significant higher cost over the blue line. With the request number comes to 30000k, CYRUS costs about 131 USD while DCStore only costs 33 USD. In the figure we can observe that the orange line is fluctuating up and down. The reason is that the requested files are randomly chosen and the chunk number of them varies a lot. But the blue line stays quite smooth, because the chunks are packed together to a number of containers such that the gap is reduced.

E. System Overhead

In this section, we evaluate the system overhead by analyzing the time cost distribution for a single *Put*/*Get* operation. We perform a *Put* and a *Get* operation on a set of files ranging from *100M* to *1G* 10 times and record the time cost of every stage during the whole process. In the *Put* process, the stages include chunking, encoding, packing, upload and metadata accessing. In the *Get* process, the stages include download, unpacking, decoding, restoring and metadata accessing.

As shown in Figure 9(a), the upload stage takes up most of the time for the *Put* operation. Taking the *1G* file for instance, the whole process costs 109 seconds in average while the upload stage costs 95 seconds, about *87%* of the total time, in average. The system overhead of *Get* operation is much smaller than that of *Put* operation because it does not need to re-calculating the hash value of all the chunks. Besides, decoding is much faster than encoding in erasure coding. As shown in Figure 9(b), the overhead of the additional stages can be neglected compared with the network transferring time.

V. Related Work

There are several systems proposed for the Cloud-of-Clouds (e.g. [5]–[7], [12], [13]). Many of them focus on data availability in the presence of cloud failures and vendor lock-ins. For example, RACS [5] uses erasure coding to improve availability and tolerate provider price hikes, thus reducing cost of data migrations and vendor lock-in. SPANStore [7] is another recent multi-cloud storage system that seeks to minimize the cost of storage, leveraging a centralized cloud placement manager. On the other hand, data deduplication is a widely-used technique is storage system. DCStore is not the first which applies data deduplication before erasure coding in a Cloud-of-Clouds storage system to reduce the total storage cost. DAC [12] uses fixed-size chunk algorithm while RAMDA [13] packs varied-size chunks into fixed-size data blocks to perform erasure coding, since erasure coding requires fixed-size data blocks. In contrast to the above systems, DCStore uses an inner chunk based erasure coding scheme and a container based share management strategy to distribute chunks across multiple clouds, thus improving the cost-efficiency and performance of cloud storage services from the user's perspective.

VI. Conclusion

In this paper, we present DCStore, a deduplication-based Cloud-of-Clouds storage service that practically addresses the reliability of cloud storage service. DCStore not only achieves fault tolerance of storage, but also achieves cost savings via client-side data deduplication, inner-chunk encoding, and container-based share management strategy. Our lightweight prototype implementation of DCStore shows that DCStore improve the performance and cost efficiency significantly compared with existing Cloud-of-Clouds storage system.

References

[1] M. Greer, "Survivability and information assurance in the cloud," in *Dependable Systems and Networks Workshops (DSN-W), 2010 International Conference on*. IEEE, 2010, pp. 194–195.

[2] "Serious cloud outages," https://www.bvoip.com/blog/the-10-biggest-cloud-outages-of-2018-so-far, 2018.

[3] M. Naldi and L. Mastroeni, "Cloud storage pricing: a comparison of current practices," in *Proceedings of the 2013 international workshop on Hot topics in cloud services*. ACM, 2013, pp. 27–34.

[4] "Duracloud project," http://www.duracloud.org/, 2019.

[5] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon, "RACS: a case for cloud storage diversity," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 229–240.

[6] Y. Hu, H. C. Chen, P. P. Lee, and Y. Tang, "NCCloud: applying network coding for the storage repair in a cloud-of-clouds." in *FAST*, 2012, p. 21.

[7] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha, "Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 292–308.

[8] Z. Wu, C. Yu, and H. V. Madhyastha, "Costlo: Cost-effective redundancy for lower latency variance on cloud storage services." in *NSDI*, 2015, pp. 543–557.

[9] W. Xia, H. Jiang, D. Feng, F. Douglis, P. Shilane, Y. Hua, M. Fu, Y. Zhang, and Y. Zhou, "A comprehensive study of the past, present, and future of data deduplication," *Proceedings of the IEEE*, vol. 104, no. 9, pp. 1681–1710, 2016.

[10] D. Bhagwat, K. Pollack, D. D. Long, T. Schwarz, E. L. Miller, and J.-F. Paris, "Providing high reliability in a minimum redundancy archival storage system," in *null*. IEEE, 2006, pp. 413–421.

[11] X. Li, M. Lillibridge, and M. Uysal, "Reliability analysis of deduplicated and erasure-coded storage," *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, no. 3, pp. 4–9, 2011.

[12] S. Wu, K.-C. Li, B. Mao, and M. Liao, "DAC: improving storage availability with deduplication-assisted cloud-of-clouds," *Future Generation Computer Systems*, vol. 74, pp. 190–198, 2017.

[13] C. Liu, Y. Gu, L. Sun, B. Yan, and D. Wang, "R-ADMAD: High reliability provision for large-scale de-duplication archival storage systems," in *Proceedings of the 23rd international conference on Supercomputing*. ACM, 2009, pp. 370–379.

[14] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the facebook warehouse cluster." in *HotStorage*, 2013.

[15] B. Fan, W. Tantisiriroj, L. Xiao, and G. Gibson, "Diskreduce: Replication as a prelude to erasure coding in data-intensive scalable computing," *SC11*, 2011.

[16] J. Y. Chung, C. Joe-Wong, S. Ha, J. W.-K. Hong, and M. Chiang, "CYRUS: Towards client-defined cloud storage," in *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015, p. 17.

[17] I. Drago, M. Mellia, M. M Munafo, A. Sperotto, R. Sadre, and A. Pras, "Inside Dropbox: understanding personal cloud storage services," in *Proceedings of the 2012 Internet Measurement Conference*. ACM, 2012, pp. 481–494.

[18] C. Policroniades and I. Pratt, "Alternatives for detecting redundancy in storage systems data." in *USENIX Annual Technical Conference, General Track*, 2004, pp. 73–86.

[19] A. Muthitacharoen, B. Chen, and D. Mazieres, "A low-bandwidth network file system," in *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5. ACM, 2001, pp. 174–187.

[20] Y. Fu, H. Jiang, N. Xiao, L. Tian, F. Liu, and L. Xu, "Application-aware local-global source deduplication for cloud backup services of personal storage," *IEEE transactions on parallel and distributed systems*, vol. 25, no. 5, pp. 1155–1165, 2014.

[21] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, S. Yekhanin *et al.*, "Erasure coding in windows azure storage." in *Usenix annual technical conference*. Boston, MA, 2012, pp. 15–26.

[22] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang *et al.*, "f4: Facebooks warm blob storage system," in *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*. USENIX Association, 2014, pp. 383–398.

[23] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A hitchhiker's guide to fast and efficient data reconstruction in erasure-coded data centers," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 331–342, 2015.

[24] W. Chen, Y. Hu, S. Yin, and W. Xia, "EEC-Dedup: Efficient erasure-coded deduplicated backup storage systems," in *Ubiquitous Computing and Communications (ISPA/IUCC), 2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on*. IEEE, 2017, pp. 251–258.

[25] O. Khan, R. C. Burns, J. S. Plank, W. Pierce, and C. Huang, "Rethinking erasure codes for cloud file systems: minimizing i/o for recovery and degraded reads." in *FAST*, 2012, p. 20.

[26] "Fastchunking," https://pypi.org/project/fastchunking/, 2019.

[27] "Zfec," https://pypi.org/project/zfec/, 2019.

[28] "Multiprocessing," https://pypi.org/project/multiprocessing/, 2019.

[29] "Fsl traces and snapshots public archive," http://tracer.filesystems.org/, 2019.