On the Synchronization Bottleneck of OpenStack Swift-like Cloud Storage Systems

Mingkang Ruan, Thierry Titcheu, Ennan Zhai, Zhenhua Li, *Member, IEEE,*
Yao Liu, Jinlong E, Yong Cui, *Member, IEEE,* and Hong Xu, *Member, IEEE*

Abstract—As one type of the most popular cloud storage services, OpenStack Swift and its follow-up systems replicate each object across multiple storage nodes and leverage *object sync protocols* to achieve high reliability and *eventual consistency*. The performance of object sync protocols heavily relies on two key parameters: r (number of replicas for each object) and n (number of objects hosted by each storage node). In existing tutorials and demos, the configurations are usually $r = 3$ and $n < 1000$ by default, and the sync process seems to perform well. However, we discover in data-intensive scenarios, *e.g.*, when $r > 3$ and $n \gg 1000$, the sync process is significantly delayed and produces massive network overhead, referred to as the *sync bottleneck problem*. By reviewing the source code of OpenStack Swift, we find that its object sync protocol utilizes a fairly simple and network-intensive approach to check the consistency among replicas of objects. Hence in a sync round, the number of exchanged hash values per node is $\Theta(n \times r)$. To tackle the problem, we propose a lightweight and practical object sync protocol, *LightSync*, which not only remarkably reduces the sync overhead, but also preserves high reliability and eventual consistency. LightSync derives this capability from three novel building blocks: 1) *Hashing of Hashes*, which aggregates all the h hash values of each data partition into a single but representative hash value with the Merkle tree; 2) *Circular Hash Checking*, which checks the consistency of different partition replicas by only sending the aggregated hash value to the clockwise neighbor; and 3) *Failed Neighbor Handling*, which properly detects and handles node failures with moderate overhead to effectively strengthen the robustness of LightSync. The design of LightSync offers provable guarantee on reducing the per-node network overhead from $\Theta(n \times r)$ to $\Theta(\frac{n}{h})$. Furthermore, we have implemented LightSync as an open-source patch and adopted it to OpenStack Swift, thus reducing the sync delay by up to 879$\times$ and the network overhead by up to 47.5$\times$.

Index Terms—Cloud storage, OpenStack Swift, object synchronization, performance bottleneck.

1 Introduction

Today's cloud storage services, *e.g.*, Amazon S3, Google Cloud Storage, Windows Azure Storage, Aliyun OSS, and Rackspace Cloud Files, provide highly available and robust infrastructure support to upper-layer applications [1]–[10]. As one type of the most popular open-source cloud storage services, OpenStack Swift and its follow-up systems such as Riak S2 and Apache Cassandra (called *OpenStack Swift-like systems*) have been used by many organizations and companies like Rackspace, UnitedStack, Sina Weibo, eBay, Instagram, Reddit, and AiMED Stat. In order to offer high data reliability and durability, OpenStack Swift-like systems typically replicate each data object across multiple storage nodes, thus leading to the need of maintaining consistency among the replicas. Almost all existing OpenStack Swift-like systems employ the *eventual consistency* model [11] to offer consistency guarantees to the hosted data objects' replica versions. Here, eventual consistency means that if no new update is made to a given object, eventually all read/write accesses to that object would return the last updated value. For OpenStack Swift-like systems, the eventual consistency model is embodied by leveraging an *object sync(hronization) protocol* to check different replica versions of each object.

While OpenStack Swift-like systems have been widely used, we still hope to deep understand how well they achieve the consistency in practice. To this end, the first part of our work is to make a lab-scale case study based on OpenStack Swift. In our realistic deployment and experiments, we observe that OpenStack Swift indeed performs well (with just a few seconds of sync delay and a few MBs of network overhead) for regular configuration (as proposed in most existing tutorials and demonstrations [12]–[14]), *i.e.*, $r = 3$ and $n < 1000$. Here r denotes the number of replicas for each object, and n denotes the number of objects hosted by each storage node. Nevertheless, we find that in data-intensive scenarios, *e.g.*, when $r > 3$ and $n \gg 1000$, the object sync process is significantly delayed and produces massive network overhead. [1] For example, when $r = 5$ and $n = 4M$, the sync delay is as long as 58 minutes and there are 3.63 GB of network messages exchanged *by every node in a single sync round.*

The exposed phenomenon is referred to as the *sync bottleneck problem* of OpenStack Swift, which also occurs

- *Mingkang Ruan and Zhenhua Li (corresponding author) are with the School of Software, Tsinghua University, Beijing, China. Email: brmk@vip.qq.com, lizhenhua1983@tsinghua.edu.cn*
- *Thierry Titcheu is with the Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg. Email: thierry_tct@yahoo.com*
- *Ennan Zhai is with the Department of Computer Science, Yale University, New Haven, CT, US. Email: ennan.zhai@yale.edu*
- *Yao Liu is with the Department of Computer Science, Binghamton University, NY, US. Email: yaoliu@binghamton.edu*
- *Jinlong E and Yong Cui are with the Department of Computer Science and Technology, Tsinghua University, Beijing, China. Email: ejl14@mails.tsinghua.edu.cn, cuiyong@tsinghua.edu.cn*
- *Hong Xu is with the Department of Computer Science, City University of Hong Kong. Email: henry.xu@cityu.edu.hk*

1. On the other hand, although both the CPU and memory usages increase as r and n increase, they generally stay at an affordable level.

in Riak S2 and Cassandra. Moreover, the problem is considerably aggravated in the presence of data updates (*e.g.*, object creations and deletions) and node failures (the worst case). In particular, when node failures occur, the failed node needs multiple (typically 3 to 4) sync rounds to converge, *i.e.*, to re-enter a stable state. Furthermore, our experiments show that this problem cannot be fundamentally addressed by employing parallelism techniques, *i.e.*, by increasing the number of sync threads N_{thread} (detailed in §3.7).

Therefore, the sync bottleneck problem would easily lead to negative influences because many of today's data-centric applications have to configure their back-ends with $r > 3$ and $n \gg 1000$ while still desiring for quick (eventual) consistency and low overhead. Such kinds of applications are pretty common in practice: first, in a realistic object storage system the number of objects is typically far more than 1000; second and more importantly, a larger r (exceeding 3) is often adopted by systems that require faster access to numerous small objects [15], a higher level of fault tolerance [16], or better geo-distributed availability [17].

Driven by the above observations, the second part of our work is to investigate the source code of OpenStack Swift, so as to thoroughly understand why the sync bottleneck problem happens. In particular, we find that during each sync round, the storage node for each *data partition* (say P) compares its local *fingerprint* of P with the fingerprints of all the other $r-1$ replicas of P. This sync process introduces network overhead of $r(r-1)$ sync messages. Specifically, as a typical storage technique, partitioning allows the entire object storage space to be divided into smaller pieces, where each piece is called a (data) partition. The fingerprint of a partition is denoted by a file which records the hash values of all the h suffix directories included in this partition. Therefore, each sync message contains h hash values.

More in detail, as one storage node can host multiple ($\approx \frac{n}{h}$) partitions, the number of exchanged hash values by each storage node is as large as $\Theta(n \times r)$ in a single sync round. This brings about considerable unnecessary network overhead. In addition, the aforementioned shortcomings are also found in other OpenStack Swift-like systems such as Riak S2 (the active anti-entropy component [18]) and Cassandra (the anti-entropy node repair component [19]).

To tackle the sync bottleneck problem, we propose a lightweight and practical sync protocol, called *LightSync*. At the heart of LightSync lie three novel building blocks:

- HoH aggregates all the h hash values of each data partition (in one sync message) into a single but representative hash value by using the Merkle tree structure. Thus, one sync message contains only one hash value.
- CHC is responsible for reducing the number of sync messages exchanged in each sync round. Specifically, CHC organizes the r replicas of a partition with a small ring structure. During a certain partition's object sync process, CHC only sends the aggregated hash value to the clockwise neighbor in the small ring.
- FNH properly detects and handles node failures with moderate overhead, so as to effectively strengthen the robustness of LightSync. Also, FNH helps a failed node quickly rejoin the system with a consistent, latest state.

With the above design, the per-node network overhead for OpenStack Swift object sync is provably reduced from $\Theta(n \times r)$ to $\Theta(\frac{n}{h})$ hash values. Besides, the performance degradation incurred by node failures is substantially mitigated. To evaluate the real-world performance, we have implemented LightSync as an open-source patch to OpenStack Swift, which is also applicable to Riak S2 and Cassandra in principle. The patch can be downloaded from *https://github.com/lightsync-swift/lightsync*. In both lab-scale (including 5 physical servers) and large-scale (including 64 Aliyun ECS virtual servers) deployments, we observe that LightSync remarkably reduces the sync delay by up to 879× and the network overhead by up to 47.5× . We also compare LightSync with existing object sync protocols using other topologies (*e.g.*, Primary/Backup using Star [20], [21] and Chain Replication using Chain [22], [23]), and find that the sync delay of LightSync is obviously shorter by 2–8 times.

This paper makes the following contributions:

- We (are the first to) discover the sync bottleneck problem of OpenStack Swift-like systems through comprehensive experiments (§3). In particular, this problem is considerably aggravated in the presence of data updates (§3.5) and node failures (§3.6), and cannot be fundamentally solved by increasing the number of sync threads (§3.7).
- We reveal the key factors that lead to the problem by investigating the source code of OpenStack Swift (§4).
- We propose an efficient and practical object sync protocol, named LightSync, to address the problem (§5).
- We implement an open-source LightSync patch which is suited to general OpenStack Swift-like systems (§5.5).
- After the patch is applied to realistic deployments, both lab-scale and large-scale testbed results illustrate that LightSync is capable of significantly improving the object sync performance. (§6) Also, LightSync essentially outperforms its counterparts in terms of sync delay. (§6.4).

2 Background

OpenStack Swift is a well-known open-source object storage system. It is typically used to store diverse unstructured data objects, such as virtual machine (VM) snapshots, pictures, audio/video volumes, and various backups. Many existing cloud storage systems are designed and implemented by (partially) following the paradigm of OpenStack Swift.

2.1 Design Goals of OpenStack Swift

OpenStack Swift offers each data object *eventual consistency*, a well-studied consistency model in the area of distributed systems. Compared with the *strong consistency* model, the eventual consistency model can achieve better data availability but may lead to a situation where some clients read an old copy of the data object [24]. Besides, OpenStack Swift provides reliability (and durability) by replicating each object across multiple (3 by default) storage nodes. .

2.2 OpenStack Swift Architecture

As demonstrated in Fig. 1, there are two types of nodes in an OpenStack Swift cluster: *storage nodes* and *proxy nodes*. Storage nodes are responsible for storing objects while proxy

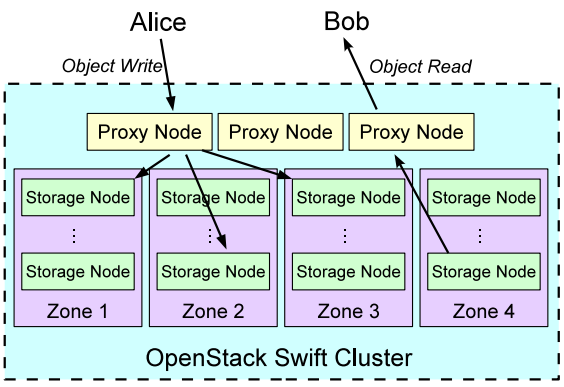


Fig. 1: OpenStack Swift architecture.

nodes—as a bridge between clients and storage nodes—communicate with clients and allocate requested objects on storage nodes. On receiving a client's read request on an object o, the proxy node first searches for the storage nodes hosting the replicas of o and then sends requests to all the replica nodes of the object o. We use r to denote the number of replicas for each object throughout the paper. By default, OpenStack Swift utilizes a quorum-based voting mechanism for replica control [25]. Once a valid number of ($\geq \lfloor r/2 \rfloor + 1$) responses are received, the proxy node selects the best response (*i.e.*, the one with the latest version of o) and then redirects the response to the client. On the other side, for a given write request on o, the proxy node sends the request to all the r storage nodes hosting o. As long as a certain number ($\geq \lfloor r/2 \rfloor + 1$) of them reply with "successful write," the update is taken as successful.

2.3 Partition and Synchronization

Like many popular storage systems, OpenStack Swift organizes data partitions through consistent hashing (or says DHT, distributed hash table) [26], [27]. Specifically, OpenStack Swift constructs a logical ring (called the *object ring* or *partition ring*) to represent the entire storage space. This logical ring is composed of many equivalent subspaces. Each subspace represents a partition and includes a number of $(h)^2$ objects belonging to the partition. According to the working principle of consistent hashing, h dynamically changes with the system scale.

Each partition is replicated r times on the logical ring, physically mapped to r different storage nodes. If all the N storage nodes in the logical ring are homogeneous, the number of partitions hosted by each node is $\frac{r \times p}{N}$, where p denotes the total number of unique partitions.

Each object is assigned a unique identifier, *i.e.*, an MD5 hash value of the object's path. Further, objects in the same partition are split into multiple subdirectories (*suffix directories*) according to the suffixes of their hash values. For Example in Fig. 2, one suffix in the directory 25 is 882, so the last three characters of all the hash values located in this suffix directory are exactly 882.

For a given partition, its fingerprint is denoted by the *hashes.pkl* file. Each line of the *hashes.pkl* file contains at least

2. Mostly each suffix directory contains only one object, *i.e.*, we may assume $h \approx$ the number of objects in a partition.

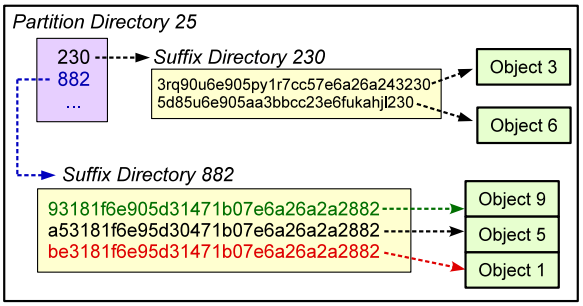


Fig. 2: An example for a data partition's structure.

35 hex characters: 3 for the hash suffix and 32 for the MD5 hash value. The corresponding sync message of a partition mainly contains its *hashes.pkl* file.

3 Case Study

To deep understand how well OpenStack Swift-like systems achieve consistency, this section presents a lab-scale case study on the object sync performance of OpenStack Swift. We first conduct experiments to understand OpenStack Swift's sync delay (§3.2), network overhead (§3.3), and CPU & memory usages (§3.4) in a *stable state*. Here a stable state means very few to no data updates (*e.g.*, object creations or deletions) occur to the OpenStack Swift system. Then, on the contrary, we examine the sync delay and network overhead of OpenStack Swift in the presence of bursty data updates (§3.5) and node failures (§3.6). We finally summarize our OpenStack Swift case study in §3.8.

3.1 Experimental Setup

We make a lab-scale OpenStack Swift deployment for the case study. The deployment involves five Dell PowerEdge T620 servers, each equipped with 2×8-core Intel Xeon CPUs@2.0 GHz, 32-GB 1600-MHz DDR3 memory, 8×600-GB 15K-RPM SAS disk storage, and two 1-Gbps Broadcom Ethernet interfaces. The operating system of each server is Ubuntu 14.04 LTS 64-bit. All these servers, as well as the client devices, are connected by a commodity TP-LINK switch with 1-Gbps wired transmission rate.

One of these servers (*Node-0*) is used to run the Openstack Keystone service for account/data authentication, and meanwhile plays the roles as both a proxy node and a storage node in the OpenStack Swift system. The other servers (*Node-1*, *Node-2*, *Node-3*, and *Node-4*) are only used as storage nodes. In this lab-scale OpenStack Swift system, the max number of partitions is fixed to $2^{18} = 262144$ (as recommended in the official OpenStack installation guide [14]), and the number of replicas for each data object is configured as $r = 2, 3, 4, 5$, respectively.

In addition, we employ multiple common laptops as the client devices. They are responsible for sending both object read and write requests through ssbench (SwiftStack Benchmark Suite [28]), a benchmarking tool for automatically generating intensive OpenStack Swift workloads. Each data object is filled with random bytes between 6 KB and 10 KB (we will prove in §4 that the object sync performance of OpenStack Swift is generally irrelevant to the concrete content and size of each data object).

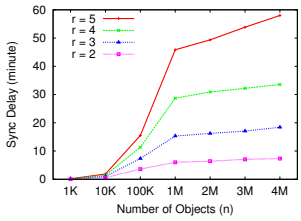


Fig. 3: Sync delay in a stable state.

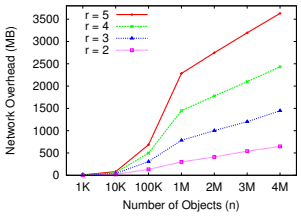


Fig. 4: Network overhead in a stable state.

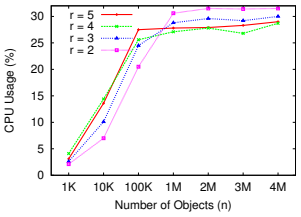


Fig. 5: CPU usage in a stable state.

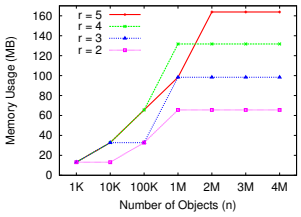


Fig. 6: Memory usage in a stable state.

3.2 Sync Delay in a Stable State

First of all, we want to understand the impact of the two key parameters, *i.e.*, r and n, on the running time of a sync round (called the sync delay). To this end, we conduct multiple experiments with increasing $n = 1K, 10K, 100K, 1M, 2M, 3M, 4M$ and $r = 2, 3, 4, 5$, respectively, and measure the sync delay when the system enters a stable state. In an OpenStack Swift system, the sync delay is recorded in its log file, *i.e.*, */var/log/syslog*.

As shown in Fig. 3, when $n \leq 1K$, the sync delay is merely a few seconds. However, when n reaches several million, it sharply increases to tens of minutes. Meanwhile, the sync delay increases with a larger r. The above phenomena are not acceptable in practical data-intensive scenarios, since they may well influence the desired availability and consistency of OpenStack Swift. An interesting finding is when $n > 1M$, the sync delay increases quite slowly (for a fixed r). This can be explained by the number of partitions (p) illustrated in Fig. 7. As mentioned in §3.1, the max number of partitions is fixed to $2^{18} = 262144$. When n grows, p is automatically increased by OpenStack Swift. But when $n > 1M \gg 262144$, p stays close to (but no more than) 262144. Hence, the number of sync messages exchanged per node (heavily depending on the value of p) keeps stable while the size of each sync message is enlarged, which will be thoroughly explained in §4.

3.3 Network Overhead in a Stable State

Next, we aim at understanding the network overhead in a sync round, which might be an essential factor that determines the sync delay. For this purpose, we measure the size of network messages exchanged within the OpenStack Swift system during the object sync process in a stable state. The measurement results in Fig. 4 show that the network overhead increases with larger n and/or r. More importantly, the four curves in Fig. 4 are basically consistent with those in Fig. 3 in terms of variation trend. For example, when $n = 4M$ and $r = 5$, the sync delay reaches the maximum 58 minutes, and meanwhile the network overhead reaches the maximum 3.63 GB. When $n > 1M \gg 262144$ (for a fixed r), although the number of sync messages keeps stable, the size of each sync message still grows with n since each sync message contains more hash values (of more data objects). This is why the network overhead continues growing with n when $n > 1M$.

3.4 CPU and Memory Usages in a Stable State

In addition to sync delay and network overhead, we wish to know the computation overhead of the object sync process. We, therefore, measure the CPU and memory usages of OpenStack Swift in a sync round. The CPU usage per storage server is plotted in Fig. 5 and the memory usage per storage server is plotted in Fig. 6. As shown in these two figures, we have the following two findings. First, both the CPU and memory usages increase as the number of objects (n) and/or the number of replicas for each object (r) increase. Second, even for the largest deployment (where $n = 4M$ and $r = 5$), the CPU usage is close to 30% and the memory usage is close to 160 MB. Since each storage server has 32 GB of memory, the highest memory usage rate is merely 0.5% ($= \frac{160\ MB}{32\ GB}$). Thus, both the CPU and memory usages are affordable for the OpenStack Swift system.

3.5 Sync Performance in the Presence of Data Updates

We now examine the sync delay and network overhead of OpenStack Swift in the presence of data updates. Specifically, we generate two types of data updates. First, we create a certain portion (10%) of data objects relative to the existing n objects, and then record the sync performance right after the new objects are successfully created. Second, we delete a certain portion (10%) of objects, and then record the sync performance right after the 10% objects are successfully deleted. The sync delay corresponding to the two types of data updates is shown in Fig. 8 and Fig. 10, respectively. The two figures consistently illustrate that the sync delay is increased when data updates happen to the system. Compared with the stable state, object creations and deletions lead to around 34.4% and 40.2% addition to the sync delay, respectively. This is because OpenStack Swift needs to recalculate the hash values of the modified data partitions, which takes extra time [3]. Differently, we observe that the network overhead in the presence of data updates is comparable to that in a stable state, indicated by the measure results in Fig. 9 and Fig. 11.

3.6 Sync Performance in the Presence of a Node Fault

In addition to data updates, we study the performance degradation caused by a node failure in OpenStack Swift. We conduct the study in the following two steps. First, we remove a storage node from the system by disconnecting it with the switch, and then record the sync performance of the remaining live nodes. Next, we *modify* a certain portion (10%) of data objects in the system, and then add the failed node into the system. At the same time, we record the

3. Caching such hash values in memory using systems such as Memcached or Mbal [29], however, would considerably increase the system complexity, and thus cannot fundamentally address the issue.

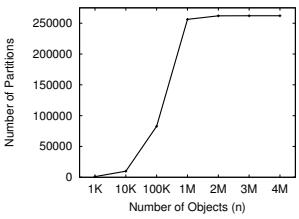


Fig. 7: Number of partitions (p).

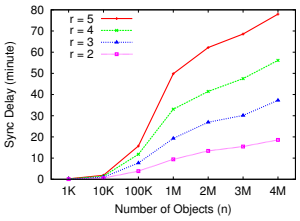


Fig. 8: Sync delay right after 10% object creations.

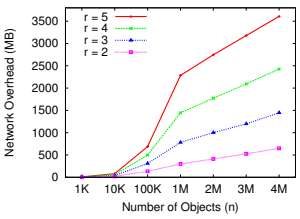


Fig. 9: Network overhead right after 10% object creations.

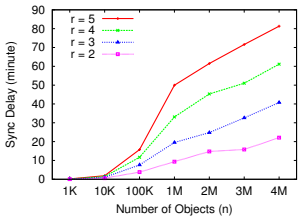


Fig. 10: Sync delay right after 10% object deletions.

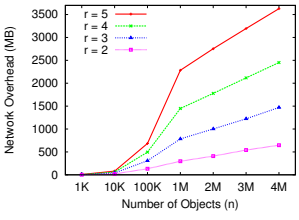


Fig. 11: Network overhead right after 10% object deletions.

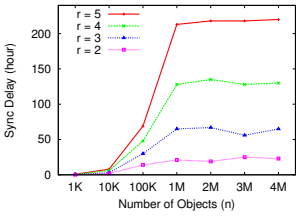


Fig. 12: Sync delay in the presence of a node failure.

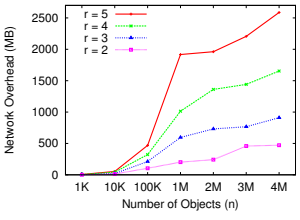


Fig. 13: Network overhead in the presence of a node failure.

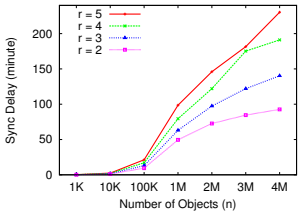


Fig. 14: Sync delay during a failed node's rejoining process.

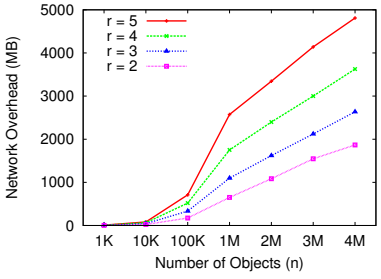


Fig. 15: Network overhead during a failed node's rejoining process.

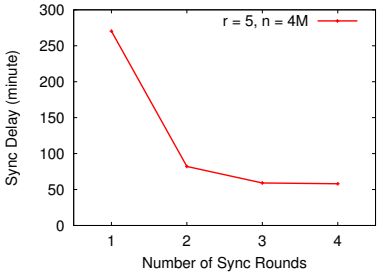


Fig. 16: Sync delay of a failed node in different sync rounds right after it rejoins the system.

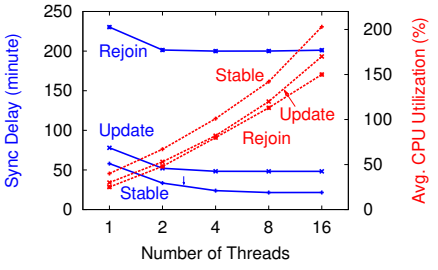


Fig. 17: Sync delay and CPU utilization in a stable state, in the presence of data updates, and during a node's rejoining process when $r = 5$ and $n = 4M$.

sync performance of the live nodes during the failed node's rejoining process. Here modify means that we first delete a certain portion of objects and then create the same number of new objects using the previous object paths.

The sync delay in the presence of a node failure is presented in Fig. 12. When $r \geq 2$ and $n \gg 10000$, the sync delay is extremely long (*e.g.*, 220 hours when $r = 5$ and $n = 4M$). The corresponding network overhead is shown in Fig. 13, which is however less than that in a stable state because many messages cannot be sent to or received from the failed node. Next, the sync delay and network overhead corresponding to the failed node's rejoining process are shown in Fig. 14 and Fig. 15, respectively. Compared with the case in a stable state, the rejoining process incurs about 297% and 32.5% addition to the sync delay and network overhead, respectively. In addition, we observe that the failed node itself needs multiple (typically 3 to 4) sync rounds to converge (*i.e.*, to re-enter a stable state), as shown in Fig. 16. When the failed node rejoins the system, the sync processes of the other live nodes are at different levels of completion, therefore some partitions on the failed node have to be updated (by the live nodes) in the subsequent sync rounds. In general, a node failure leads to the worst case in the sync performance of OpenStack Swift.

3.7 Using Multiple Sync Threads

OpenStack Swift provides an optimization option to accelerate the object sync process by increasing the number of sync threads (N_{thread}) (*i.e.*, the so-called "parallelism"). By default, $N_{thread} = 1$, which is the configuration adopted in §3.2 and §3.3. To understand the influence of N_{thread} on the sync delay, we make every storage node host $n = 4M$ objects with $r = 5$ replicas, and run experiments with different N_{thread}. As demonstrated in Fig. 17, the sync delay in a stable state can be considerably reduced from 58 minutes to 21 minutes when N_{thread} increases from 1 to 8, but cannot be further reduced when $N_{thread} > 8$. Differently, N_{thread} does not impact the network overhead of the sync process.

Even worse, increasing parallelism contributes little to the reduction of sync delay, especially in scenarios when hash values have to be recalculated and rewritten on disk

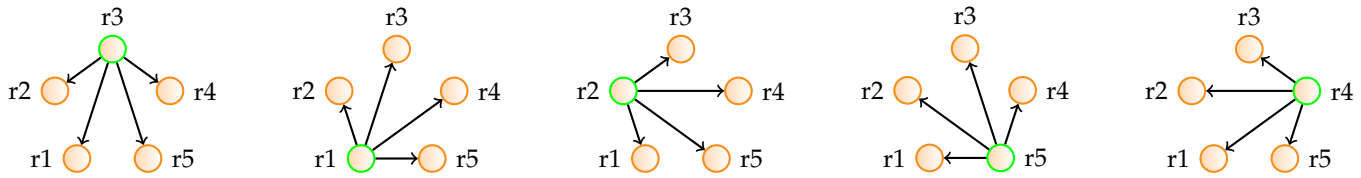


Fig. 18: An example for a complete object sync process. The five sub-processes run in parallel rather than in sequence.

(*e.g.*, in the presence of data updates or during a rejoining process). This is because the object sync process is an I/O-bound process. The disk bandwidth has become the major performance bottleneck when N_{thread} reaches a certain level (*e.g.*, $N_{thread} > 2$ during a rejoining process). Simply increasing the level of parallelism may even aggravate the problem (by investing more CPU resources).

On the other hand, as N_{thread} increases, the average CPU utilization (in unit of CPU core) also increases. For instance, the average CPU utilization of a storage node reaches 100% (200%) if we use 4 (16) sync threads. In conclusion, parallelism could *partially* reduce the sync delay (since the sync delay can hardly be reduced when $N_{thread} > 8$) but at the cost of higher computation overhead. Further, we observe the similar phenomenon in the presence of data updates and node failures.

3.8 Summary: The Sync Bottleneck Problem

Based on all the revealed results in this section, we discover an important phenomenon: the object sync performance can be badly influenced once the data intensity of OpenStack Swift becomes higher than a certain level, *e.g.*, $r > 3$ and $n \gg 1000$. This phenomenon is referred to as the *sync bottleneck problem* of OpenStack Swift, which also occurs in the follow-up systems like Riak S2 and Cassandra, where similar benchmark experiments illustrate similar situations. In particular, our experimental results demonstrate the following four features of OpenStack Swift. First, common configurations, *e.g.*, $r = 3$ and $n < 1000$, make OpenStack Swift work well. Second, data-intensive configurations, *e.g.*, $r > 3$ and $n \gg 1000$, would lead to the sync bottleneck problem. Third, the sync bottleneck problem is considerably aggravated in the presence of data updates and node failures (the worst case). Finally, the optimization mechanism by increasing N_{thread} cannot fundamentally solve the sync bottleneck problem in practice. Thus, we will only use the default $N_{thread} = 1$ in the remainder of the paper.

4 Root Cause Analysis

To explore the root cause of the sync bottleneck problem, we investigate the source code of OpenStack Swift. This section presents our investigation results about: 1) how the object sync process works in OpenStack Swift; and 2) how expensive the current object sync protocol is.

4.1 Object Sync Process in OpenStack Swift

The relevant source code of OpenStack Swift (the Icehouse version [4]) is mainly included in the files listed in Table 1.

4. We have also examined the latest Liberty version of OpenStack Swift and find that the concerned source code is generally unchanged.

TABLE 1: Source code of OpenStack Swift relevant to its object sync process.

Path	File
python2.7/dist-packages/swift/obj	diskfile.py mem_diskfile.py replicator.py server.py
python2.7/dist-packages/swift/proxy	server.py
python2.7/dist-packages/swift/proxy/controller	base.py obj.py
python2.7/dist-packages/swift/common	bufferedhttp.py http.py
python2.7/dist-packages/swift/common/ring	$*$.py

Through the source code review, we find that OpenStack Swift is currently using a fairly simple and network-intensive approach to check the consistency among replicas of a data partition, where a partition consists of h objects. Fig. 18 depicts an example for a complete OpenStack Swift object (partition) sync process with $r = 5$. For a given partition P, in each sub-process, all the nodes hosting the r replicas will randomly elect one node as the leader, but different sub-processes must generate different leaders. The leader sends one sync message to each of the other $r - 1$ nodes to check the statuses of P on them. When all the r sub-processes finish, we say an object sync process (*i.e.*, a sync round) of the partition P is completed.

In addition, by examining the relevant source code, we find the above approach is also adopted by other OpenStack Swift-like systems, such as Riak S2 (the active anti-entropy component [18]) and Cassandra (the anti-entropy node repair component [19]).

4.2 Network Overhead Analysis

While we have observed the sync bottleneck problem from our case study (§3), we hope to quantitatively understand how expensive the current object sync protocol of OpenStack Swift is in principle/theory. As §3 has clearly illustrated that it is the enormous network overhead that leads to the sync bottleneck problem, we focus on analyzing the network overhead. It is straightforward to deduce from Fig. 18 that for a given partition P, the total number of sync messages exchanged during an entire object sync process is $2C_r^2 = r(r-1)$, assuming no message loss. Besides, the size of each sync message depends on the size of the *hashes.pkl* file (see Fig. 2), which contains h hash values. Furthermore, as one storage node can host multiple ($\frac{n}{h}$) partitions, the number of exchanged hash values by each node is around $\frac{n}{h} \times \frac{r(r-1)\times h}{r} = n(r-1)$ in a sync round. Finally, taking the other involved network overhead (*e.g.*, HTTP/TCP/IP packet headers for delivering the hash values) into account, we conclude that the per-node per-round

network overhead of OpenStack Swift is in $\Theta(n \times r)$. Apart from the hash values, OpenStack Swift needs to push the content of corresponding objects to remote nodes if there is any inconsistency.

5 LightSync: Design and Implementation

Guided by the thorough understanding of the object sync process of OpenStack Swift, we design a lightweight and practical object sync protocol, called *LightSync*, to tackle the sync bottleneck problem. LightSync not only significantly reduces the sync overhead, but also is applicable to general OpenStack Swift-like systems.

5.1 LightSync Overview

LightSync is designed to replace the original object sync protocols in current OpenStack Swift-like systems. Besides reducing the sync overhead, it can also ensure high reliability and eventual consistency. LightSync derives the desired properties from the following three novel building blocks.

First, LightSync employs the Hashing of Hashes (HoH) mechanism (§5.2) to reduce the size of each sync message. The basic idea of HoH is to aggregate all the h hash values in each partition into a single but representative hash value by using the Merkle tree data structure. HoH replaces the original approach to generating the fingerprint file *hashes.pkl* by generating a much smaller fingerprint file *changed.pkl*.

Second, LightSync leverages the Circular Hash Checking (CHC) mechanism (§5.3) to reduce the number of sync messages exchanged in each sync round. CHC organizes all the replicas of a partition with a ring structure. During a certain partition's object sync process, CHC only sends the aggregated hash value (by HoH) to the clockwise neighbor in the ring (instead of the original all-to-all manner).

Third, LightSync utilizes the Failed Neighbor Handling (FNH) mechanism (§5.4) to properly detect and handle node failures with moderate overhead, so as to effectively strengthen the robustness of LightSync.

Finally, §5.5 describes how we implement LightSync as an open-source patch to OpenStack Swift.

5.2 Hashing of Hashes (HoH)

Preliminary: Merkle tree. A Merkle tree [30] is a tree structure for organizing and representing the hash values of multiple data objects. The leaves of the tree are the hash values of data objects. Nodes further up in the tree are the hash values of their respective children. For example, in Fig. 19, Hash 0 is the hash value of concatenating Hash 0-0 and Hash 0-1, *i.e.*, Hash 0 = Hash(Hash 0-0 + Hash 0-1), where "+" means concatenation. In practice, Merkle tree is mainly used to reduce the amount of data transferred during data checking. Suppose two storage nodes A and B use a Merkle tree to check the data stored by each other. First, A sends the root-layer hash value of its Merkle tree to B. Then, B compares the received hash value with the root-layer hash value of its local Merkle tree. If the two values match, the checking process terminates; otherwise, A should send the lower-layer hash values in its Merkle tree to B for further checking. However, finding inconsistent data objects in the above-mentioned way needs multiple rounds of data transmission. The cost of long round-trip time (RTT) outweighs the benefits of reduced network traffic. Therefore, LightSync discards the intermediate hash values but stores only the root-layer hash values and the leaves of the Merkle tree. Once the root-layer hash values do not match each other, LightSync will directly compare the hash values in the leaves of the Merkle tree.

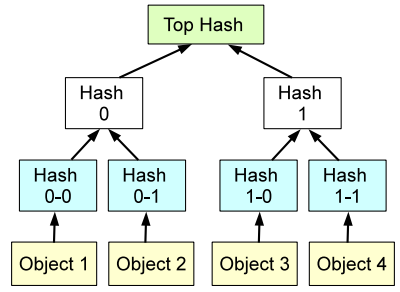


Fig. 19: An example for the Merkle tree.

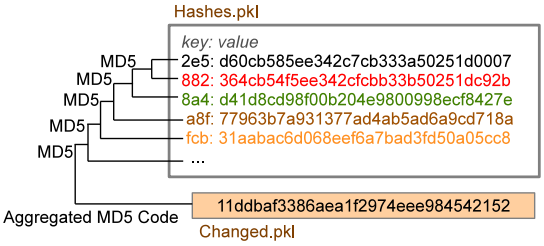


Fig. 20: Hashing of hashes for a data partition.

Generation of the aggregated hash value. We now describe how HoH generates the aggregated hash value that represents a given partition P. First, HoH extracts the *hashes.pkl* file of P (*i.e.*, the fingerprint of P). Then, HoH computes the MD5 hash values of all the suffix hashes in P one by one (as demonstrated in Fig. 20). This process constructs the Merkle tree structure. Finally, HoH stores the aggregated MD5 hash value, *i.e.*, the root-layer hash value of the Merkle tree, in a file named *changed.pkl* (also stored in the partition's directory). So far, when a storage node wants to send a sync message (for a partition P) to another storage node, it only needs to "envelop" a single hash value, *i.e.*, the aggregated hash value in *changed.pkl*, into each sync message.

Consistency checking. If an aggregated hash value of a data partition is found inconsistent between two storage nodes, the local node should first determine which suffix directory is inconsistent and then which version of that corresponding suffix directory is more up-to-date.

Firstly, the inconsistent suffix directory is sought out by comparing the s leaf-layer hash values received with the local ones within $O(s)$ steps, where s is the number of suffix directories in the corresponding data partition. Once the inconsistent suffix directory is found, the local node actively pushes the corresponding data chunks to the remote node, which will later determine which version is newer by checking timestamps recorded as file names of data chunks. Finally, the stale data chunks will be deleted.

Compared with the original design of OpenStack Swift, HoH uses a single but representative hash value to replace a

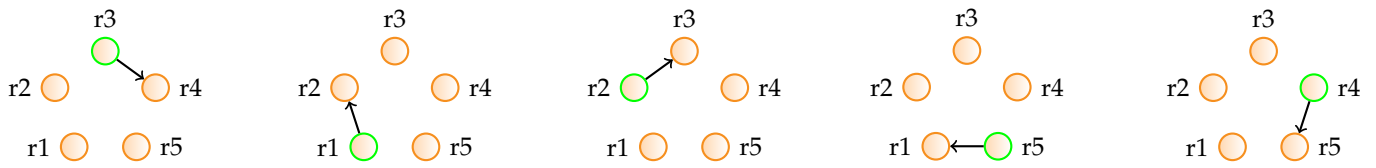


Fig. 21: An example for a complete LightSync process. The five sub-processes run in parallel rather than in sequence.

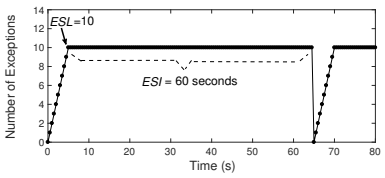


Fig. 22: An exemplified procedure of our failure detection mechanism.

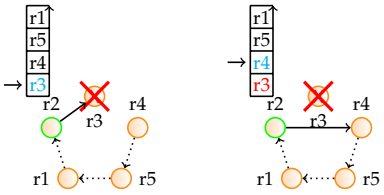


Fig. 23: An example of how FNH works.

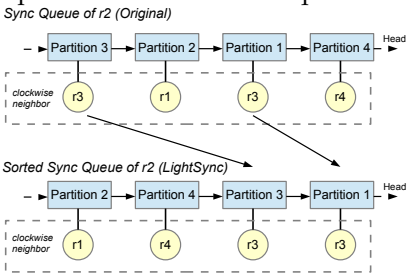


Fig. 24: An example for the sync queue maintained by r2.

Algorithm 1: Circular Hash Checking

Input: A set R_P containing all the replica nodes' IDs for a given data partition P;

```
while R_P ≠ ∅ do
    Randomly pick out a replica node's ID from R_P;
    r_P ← the picked replica node's ID;
    Remove r_P from R_P;
    The replica node (with ID =) r_P sends a sync message to r_P's clockwise neighbor;
    if r_P's version of P is different from the version held by its clockwise neighbor then
        r_P pushes its hosted data of P at local version to its clockwise neighbor;
```

large collection of hash values, thus effectively reducing the size of each sync message by nearly h times.

5.3 Circular Hash Checking (CHC)

CHC is responsible for enabling different replicas of the same partition to achieve consistency more efficiently. Specifically, during a circular hash checking process, the storage nodes hosting the r replicas of a given partition P form a small logical ring, called the *replica ring* of P. This small replica ring is easy to form as it already exists inside the large *object ring* (refer to §2.3).

Suppose P has 5 replicas, and ri denotes the storage node hosting the i-th replica for P. When a storage node wants to check the consistency of P with the other replica nodes, it only sends a sync message (generated by HoH) to the successor node clockwise on the replica ring of P—this successor replica node is referred to as its *clockwise neighbor*. For example in Fig. 21, when r3 wants to check the consistency of P, it only sends a sync message to r4 rather than r1, r2, r4 and r5 (as in Fig. 18). After each replica node finishes sending a sync message to its clockwise neighbor, we say a CHC process (or a CHC sync round) is completed. Formally, Algorithm 1 describes how CHC works.

TABLE 2: An example of the failure table maintained by r2.

Node ID	$n_{exception}$	$t_{exception}$
r1	0	–
r3	10	5/5/2017 1:12:30 PM
r4	9	5/5/2017 1:12:29 PM
r5	0	5/5/2017 1:11:00 PM

5.4 Failed Neighbor Handling (FNH)

As illustrated in §3.6, node failures significantly degrade the sync performance of OpenStack Swift. For LightSync, this problem remains since neither HoH nor CHC could ever alleviate it (sometimes they can even aggravate the problem). We therefore propose a mechanism called *Failed Neighbor Handling* (abbreviated as FNH) to effectively address the problem with moderate overhead. Specifically, our solution consists of three successive procedures: failure detection (§5.4.1), failure handling (§5.4.2) and failed node's rejoining (§5.4.3). We detail each of them as follows.

5.4.1 Failure Detection

In a distributed storage system like OpenStack Swift, the failure of any node is a complicated phenomenon which can hardly be accurately determined. On one hand, when a node fails to send heartbeat messages in a heartbeat period (referred to as an *exception*), we cannot simply determine whether this node fails, because this exception might be owing to a temporary network problem. In other words, we should not determine a node's failure in an aggressive manner; otherwise, the overhead of failure handling would be enormous and mostly unnecessary.

On the other hand, when a node has lost contact with its neighbor(s) in a number of heartbeat periods, we have to look these consecutive exceptions as a node failure and then take efforts to handle the failure, rather than waiting for the potential recovery to this node for infinite time. In other words, we should not determine a node's failure in a conservative manner; otherwise, the working efficiency of the system would be substantially impaired.

Guided by the above insights, we devise a practical failure detection procedure to reasonably determine a node's

failure. First, we use a *failure table* for each storage node to record the statistics of the other storage nodes' exceptions, as exemplified in Table 2. Each row of the failure table corresponds to one of the other storage nodes and contains two variables: $n_{exception}$ (*i.e.*, the number of exceptions that have occurred to that node) and $t_{exception}$ (*i.e.*, the happening time of the last exception). When $n_{exception}$ reaches a threshold $error\_suppression\_limit$ (ESL), the corresponding node is considered failed. Then, the current node would stop syncing with the failed node for a specific period of time $error\_suppression\_interval$ (ESI). In our implementation, $ESL = 10$ and $ESI = 60$ seconds; both configurations are derived from OpenStack Swift statistics [31]. After the ESI, $n_{exception}$ will be reset to 0. The whole procedure is demonstrated in Fig. 22.

5.4.2 Failure Handling

The failure of a node would impair the clockwise propagation of latest data and state information along the replica ring (constructed by CHC, refer to §5.3). In other words, a failed node would make CHC inefficient or even ineffective, which would further undermine the whole object sync process of LightSync. To this end, once a failed node is detected, we leverage a simple yet practical mechanism to handle the failure, which actively eliminates a failed node from the current replica ring and reorganizes the healthy nodes into a new replica ring. For example in Fig. 23, when a healthy node r2 determines that its clockwise neighbor r3 has failed, it needs to skip r3 and connects to r3's clockwise neighbor r4 using its failure table. Likewise, if r4 has also failed, r2 will skip both r3 and r4 and then connect to r5. When the new replica ring is successfully formed, r2 needs to send a notification message to all the other healthy nodes (in the new replica ring) so that they can update their failure tables into a consistent state.

5.4.3 Failed Nodes' Rejoining

In practice, a failed node can be recovered in a certain period of time, and then rejoin the system by merging into its originally residential replica ring. A strawman solution for a failed node's rejoining process is: when an existing node in the replica ring detects the recovery of the failed node (now becoming a *rejoining node*), it will introduce the rejoining node to all the other existing nodes. This process can work in a reverse manner compared to failure handling. Nevertheless, if there are data updates happening after the failed node (temporarily) leaves the system and before the failed node rejoins the system, the original rejoining mechanism implemented by OpenStack Swift would incur considerably longer delay and larger traffic overhead than necessary. Specifically, once data inconsistency is detected, the original rejoining mechanism lets each node push its hosted data objects at their locally latest versions (which might not be the globally latest ones) to the other node(s) in the replica ring. Obviously, the network traffic caused by the failed node's pushing its obsolete data to the other node(s) is unnecessary and introduces more delay.

In order to figure out how to reduce such a kind of unnecessary data pushes, we first revisit the sync process of OpenStack Swift. At first, a storage node adds its hosted data partitions to a sync queue in a random order. During a sync round, the current node extracts partitions from the queue serially, generates their corresponding replica rings (one for each partition , as described in §5.3), checks their consistency with respective clockwise neighbors and initiates data pushes if inconsistency is detected. The same process is run by other storage nodes concurrently and independently, so that some obsolete partitions in the local sync queue will be updated by other storage nodes (*i.e.*, inconsistency is eliminated) before the current node checks their consistency.

Guided by the above insights, we optimize the original rejoining mechanism of OpenStack Swift by dynamically adjusting the order of partitions in the sync queue of a healthy node. First, FNH synchronizes those partitions whose replica rings use the rejoining nodes as the current node's clockwise neighbors. In this way, their corresponding replicas on the rejoining nodes (which are likely to be obsolete) can be updated earlier. For example in Fig. 24, the current healthy node r2 maintains a sync queue of its hosted data partitions and r3 is a rejoining node. Then, FNH moves partitions whose replica rings use r3 as r2's clockwise neighbor to the head of the sync queue. Hence, the corresponding replicas of the same partitions (*i.e.*, Partition 1 and 3) on r3 are likely to be updated before r3 checks their consistency. The same process will be done in parallel by other healthy nodes (*i.e.*, r1, r4, r5), which provides even faster recovery of r3 with lower network overhead.

5.5 Implementation

We implement LightSync for OpenStack Swift (the Icehouse version) in Python, without introducing any additional library. Specifically, we develop HoH + CHC + FNH by adding, deleting, or modifying over 500 lines of Python codes. We have published LightSync as an open-source patch to benefit the community via the following link: *https://github.com/lightsync-swift/lightsync*.

6 Evaluation

In this section, we first analyze the theoretical network overhead of LightSync in §6.1. Then, to evaluate the real-world performance of LightSync, we conduct both lab-scale and large-scale experiments on top of OpenStack Swift equipped with LightSync. The lab-scale and large-scale experiment results are presented in §6.2 and §6.3, respectively. The goal of our evaluation is to explore how well LightSync improves the object sync process of OpenStack Swift-like systems, mainly in terms of sync delay and network overhead. Some other performance metrics, including CPU usage, memory usage are also examined during our evaluations.

6.1 Theoretical Analysis

Network overhead. By comparing Fig. 18 and Fig. 21, we discover that for a given partition P, the total number of sync messages exchanged during a sync round is reduced from $2C_r^2 = r(r-1)$ to r by CHC. Further, with respect to each sync message, the number of its delivered hash values is reduced from h to 1 by HoH. As one storage node can host $\frac{n}{h}$ partitions, the number of exchanged hash values by each node is around $\frac{n}{h} \times \frac{r}{r} \times 1 = \frac{n}{h}$ with LightSync. It is worth

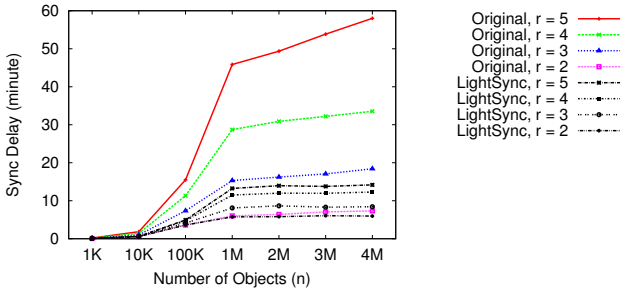


Fig. 25: Sync delay of LightSync in a stable state in lab-scale experiments. The sync delay with the original design is also plotted for comparison.

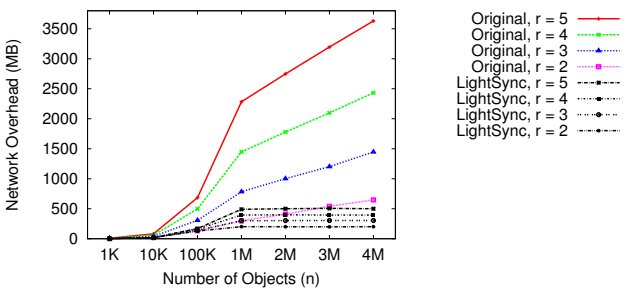


Fig. 26: Network overhead of LightSync in a stable state in lab-scale experiments. The network overhead with the original design is also plotted for comparison.

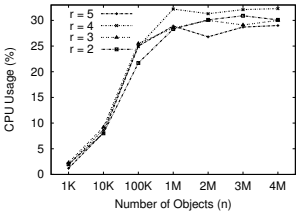


Fig. 27: CPU usage of LightSync in a stable state.

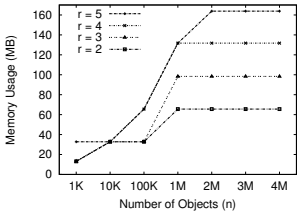


Fig. 28: Memory usage of LightSync in a stable state.

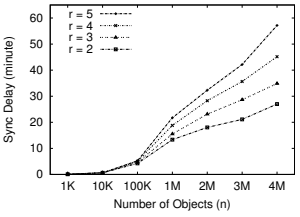


Fig. 29: Sync delay right after 10% object creations.

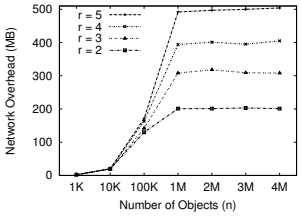


Fig. 30: Network overhead after 10% object creations.

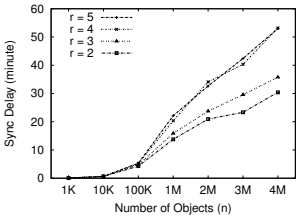


Fig. 31: Sync delay right after 10% object deletions.

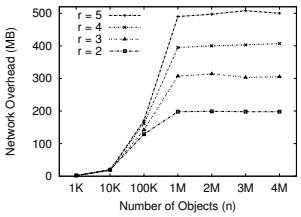


Fig. 32: Network overhead right after 10% object deletions.

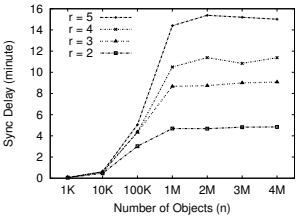


Fig. 33: Sync delay in the presence of a node failure.

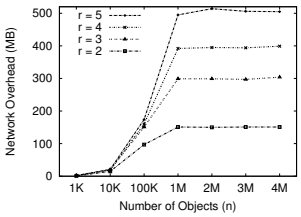


Fig. 34: Network overhead in the presence of a node failure.

mentioning that h dynamically changes with the system scale and we have $\frac{n}{h} = \frac{r \times p}{N}$, as described in §2.3. Finally, taking the other involved network overhead into account, we conclude that LightSync significantly reduces the per-node per-round network overhead of OpenStack Swift from $\Theta(n \times r)$ to $\Theta(\frac{n}{h})$.

Eventual Consistency guarantee. At each sync round, replicas are updated by its anti-clockwise neighbor (refer to §5.2). Let Ω_t^i denote the replica's timestamp of node i at sync round t, then we have: $\Omega_{t+1}^i = \max(\Omega_t^{(i-1)\%r}, \Omega_t^i)$. Applying this chain rule for multiple times, we have $\Omega_{t+4}^i = \max(\Omega_t^{(i-4)\%r}, \Omega_t^{(i-3)\%r}, \Omega_t^{(i-2)\%r}, \Omega_t^{(i-1)\%r}, \Omega_t^i)$ Suppose there are five replicas configured. In the worst case, if the updated version of data exists on only one of the five replica nodes, the other four nodes can obtain the newest version in no more than 1, 2, 3, 4 rounds, respectively. However, in our case where the quorum-based replica control mechanism is enabled, a successful write requires that more than half of replica nodes should have the updated version, *i.e.*, 3 if there are five replicas. On this condition, in no more than 2 rounds (mostly 1 round is enough), all replica nodes will have the updated version. Further experiments in §6.2 show that LightSync converges quickly.

6.2 Lab-scale Experiments

To comprehensively understand the performance of LightSync, we still conduct experiments on our lab-scale OpenStack Swift deployment (refer to §3.1 for details).

Sync delay and network overhead in a stable state. As in Fig. 25, LightSync remarkably decreases the sync delay of OpenStack Swift—the four curves of LightSync are almost always below their counterparts of Original. Here "Original" denotes the original object sync protocol. More importantly, it is validated that the sync delay with LightSync is positively related to the number of partitions hosted by each storage node (*i.e.*, $\frac{r \times p}{N}$). Owing to the notable power of CHC and HoH in avoiding unnecessary sync messages, the sync delay keeps stable (for a fixed r and N) when $n \geq 1M$ and p reaches its maximum 262144. LightSync also effectively decreases the network overhead of OpenStack Swift, as shown in Fig. 26. Note that the four curves of LightSync are below their counterparts of Original, and the network overhead with LightSync is basically consistent with the sync delay in

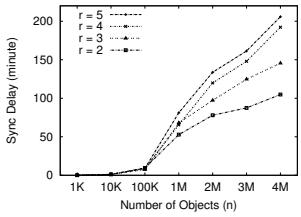


Fig. 35: Sync delay during a failed node's rejoining process.

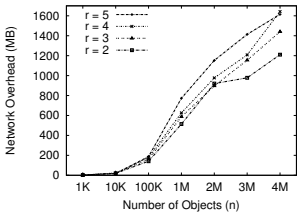


Fig. 36: Network overhead in a failed node's rejoining process.

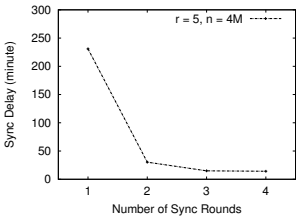


Fig. 37: Sync delay in different sync rounds during the rejoining process.

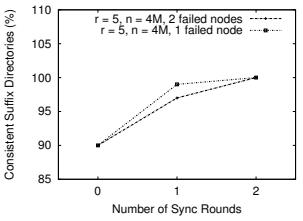


Fig. 38: Accuracy of CHC in different sync rounds during the rejoining process.

Fig. 25 in terms of variation trend. Quantitatively, LightSync reduces the sync delay by $1.0 \sim 4.09$ (2.23 on average) times, and the network overhead by $1.0 \sim 7.26$ (3.50 on average) times. In particular, with regard to the largest configuration (*i.e.*, $r = 5$ and $n = 4M$), the sync delay is reduced from 58 to 14.18 minutes, and the network overhead is reduced from 3630 to 500 MB.

CPU and memory usages in a stable state. The CPU and memory usages of LightSync (per storage server) in a stable state are shown in Fig. 27 and Fig. 28. By comparing Fig. 27 with Fig. 5 and comparing Fig. 28 with Fig. 6, we observe that the computation overhead of LightSync is comparable to that of the original object sync protocol of OpenStack Swift. Namely, although LightSync can significantly reduce the sync delay and network overhead, it does not save on computation resources.

Sync delay and network overhead in the presence of data updates. Sync delay of LightSync right after 10% object creations and deletions is presented in Fig. 29 and Fig. 31, respectively. By comparing the two figures with Fig. 8 and Fig. 10, we observe that LightSync greatly reduces the sync delay of OpenStack Swift. For example, when $r = 5$ and $n = 4M$, LightSync reduces the sync delay right after 10% object creations and deletions by around 27% and 35%, respectively. Accordingly, the network overhead of LightSync right after 10% object creations and deletions is shown in Fig. 30 and Fig. 32, respectively. By comparing the two figures with Fig. 9 and Fig. 11, we find that LightSync significantly reduces the network overhead of OpenStack Swift. For example, when $r = 5$ and $n = 4M$, LightSync reduces the network overhead right after 10% object creations and deletions by around 86% and 86.2%, respectively.

Sync delay and network overhead in the presence of a node failure. When one of the five storage nodes fails in the system, both the sync delay and network overhead of LightSync are comparable to those in a stable state, as indicated in Fig. 33 and Fig. 34. They consistently illustrate the robustness of LightSync in the presence of a node failure. On the contrary, the original object sync protocol of OpenStack Swift does not function well in this situation (*e.g.*, its sync delay reaches 220 hours when $r = 5$ and $n = 4M$). When the failed node is recovered and then rejoins the system, the sync delay and network overhead introduced by the rejoining process are shown in Fig. 35 and Fig. 36, respectively. By comparing the two figures with Fig. 14 and Fig. 15, we find that the sync delay of LightSync is slightly shorter than that of Original while the network overhead is substantially reduced. For example, when $r = 5$ and $n = 4M$, the sync delay of LightSync during a failed node's rejoining process is 205.89 minutes while that of Original is 230.31 minutes. However, the network overhead is greatly reduced from 4811 MB to 1614 MB.

Convergence time of a failed node. We record sync delay of a failed node in several sync rounds right after it rejoins the system and find that LightSync also needs multiple sync rounds to converge. However, each sync round takes less time than that of Original and therefore a failed node can get back to work sooner. For example in Fig. 37, the sync delay of the second and third sync rounds is very close to that in a stable state.

Accuracy of CHC. We define the *accuracy* of CHC as the percentage of suffix directories in a storage node that are consistent with the updates. we conduct experiments under conditions where no more than half of the replica nodes fail. As shown in Fig. 38, LightSync converges in no more than two sync rounds (mostly a single sync round is enough).

6.3 VM-based Large-scale Experiments

To construct a large-scale experimental environment for the performance evaluation of LightSync (as well as the original design of OpenStack Swift), we launch 64 VMs on top of the Aliyun.com ECS (Elastic Compute Service) platform. Each VM is equipped with a 2-core Intel Xeon CPU@2.3 GHz, 4-GB memory, and 600-GB disk storage. The operating system of each VM is Ubuntu 14.04 LTS 64-bit. All these VMs are connected by a local area network (LAN) or VLAN inside the Aliyun.com ECS cloud.

In the large-scale deployment, we directly conduct our experiment with the largest configuration $r = 5$ and $n = 4M$, and the major performance results are listed in TABLE 3. In a stable state, the sync delay is considerably reduced by LightSync from 4.92 minutes to 1.19 minutes, and the network overhead is substantially reduced from 1758 MB to 37 MB. Also, in the presence of data updates and node failures, we observe obvious reductions in both sync delay and network overhead brought by LightSync. On the other hand, we notice that in most cases LightSync incurs more CPU and memory usages, although both usages are quite low and thus well affordable. To facilitate the comparison of key performance in lab-scale and large-scale experiments, we visualize them in Fig. 39, Fig. 40, Fig. 41 and Fig. 42. Surprisingly, we find that the sync performance with many relatively weak VMs is considerably better than that with 5 powerful physical servers. The reason can be

TABLE 3: Performance of large-scale experiments when $r = 5$ and $n = 4M$.

Experimental Scenario	Sync Delay (minute)	Network Overhead (MB)	CPU (%)	Memory (MB)
In a stable state	Original: 4.92 LightSync: 1.19	Original: 1758 LightSync: 37	Original: 7.3 LightSync: 11.5	Original: 36.86 LightSync: 49.15
Right after 10% object creations	Original: 96.20 LightSync: 68.95	Original: 1709 LightSync: 38	Original: 4.0 LightSync: 6.1	Original:33.8 LightSync: 49.1
Right after 10% object deletions	Original: 90.2 LightSync: 76.53	Original: 1720 LightSync: 40	Original: 2.5 LightSync: 3.1	Original:33.8 LightSync: 49.1
In the presence of a node failure	Original: 1680 (28 hours) LightSync: 2	Original: 1680 LightSync: 34	Original: 1 LightSync: 10.4	Original:41 LightSync: 41
During a failed node's rejoining process	Original: 47.62 LightSync: 30	Original: 2115 LightSync: 153	Original: 5.4 LightSync: 9.3	Original:41 LightSync: 41

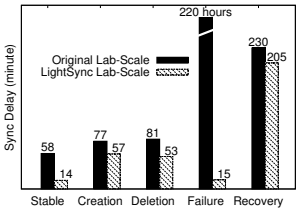


Fig. 39: Sync delay in lab-scale experiments.

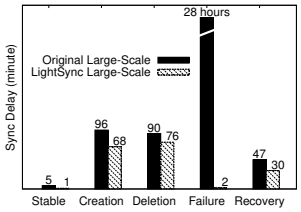


Fig. 40: Sync delay in large-scale experiments.

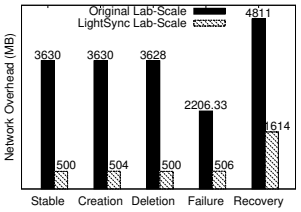


Fig. 41: Network overhead in lab-scale experiments.

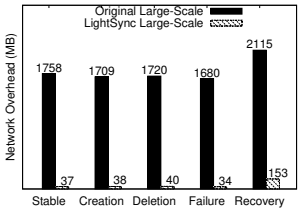


Fig. 42: Network overhead in large-scale experiments.

traced back to the composition of a data partition again (similar to the reason in §3.2). Specifically, in the large-scale deployment the number of partitions is $p \approx 262144$. Compared with the case of lab-scale deployment (when $r = 5$ and $n = 4M$), the same number (≈262144) of partitions have to store much more ($64/5$ times) objects. Thus, the size of each sync message is either enlarged (for the Original object sync protocol) or unchanged (for LightSync), but the number of sync messages exchanged per node (*i.e.*, $\frac{r \times p}{N}$) definitely decreases.

6.4 Object Sync Protocols using Other Topologies

To conduct a proper and fair comparison between LightSync and its counterparts, we first investigated existing object sync protocols using other topologies (i.e., the Primary/Backup protocol [32] using the Star topology and the Chain Replication protocol [33] using the Chain topology) adopted by relevant distributed systems or storage systems (e.g., Zookeeper [20], WheelFS [21], and Windows Azure Storage [23]). Then, we re-implemented them in realistic OpenStack Swift systems and compared them with our proposed LightSync solution (using the Ring topology) in terms of sync delay, network overhead, CPU usage, and memory usage. The experiment results (Fig. 43, Fig. 44, Fig. 45 and Fig. 46) show that LightSync essentially outperforms its counterparts in terms of sync delay. Quantitatively, sync delay of LightSync is obviously shorter by 2–8 times. Besides, its CPU usage, memory usage and network overhead are still affordable, though not always the most desirable.

7 Related Work

In recent years, numerous cloud storage systems have been designed and implemented with a variety of consistency models and object sync protocols. For almost every imaginable combination of features, certain object-based or key-value stores exist, and thus they occupy every point in the space of consistency, reliability, availability, and performance trade-offs. These stores include Amazon S3, Windows Azure Storage [23], OpenStack Swift, Riak S2, Cassandra, and so forth. In this paper, we focus on improving the sync performance of achieving eventual consistency—the most widely adopted consistency model at the moment, based on a preliminary conference version [34].

Generally speaking, eventual consistency is a catch-all phrase that covers any system where replicas may diverge in short term as long as the divergence is eventually repaired [24]. In practice, systems that embrace eventual consistency have their specific advantages and limitations. Some systems aim to improve efficiency by waiving the stable history properties, either by rolling back operations and re-executing them in a different order at some of the replicas [35], or by resorting to a last-writer-wins strategy which often results in loss of concurrent updates [36]. Other systems expose multiple values from divergent branches of operation replicas either directly to the client [37] or to an application-specific conflict resolution procedure [24].

Particularly, efforts have been made to improve the working efficiency of OpenStack Swift's object sync process, *e.g.*, by computing hash values of objects in real time and deploying an agent to check the logs for PUT and DELETE operations [38]. Compared with LightSync, they fail to provide quantitative evaluation results in data-intensive deployments. Besides, some other studies [39]–[44] reveal that the node placement/organization strategy in object storage services may lead to data availability bottlenecks, but they do not dive deeper into the sync bottleneck problem.

8 Conclusion

OpenStack Swift-like cloud storage systems have been widely used in recent years. In this paper, we study the object sync protocol that is fundamental to their performance, particularly the key parameters r (number of replicas for each object) and n (number of objects hosted by each storage

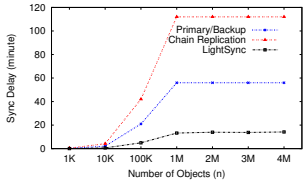


Fig. 43: Sync delay of protocols using different topologies.

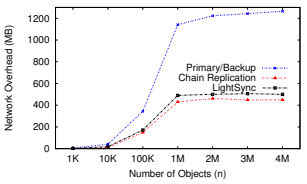


Fig. 44: Network overhead of protocols using different topologies.

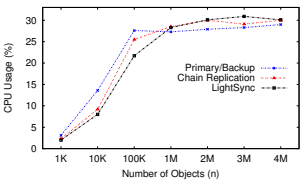


Fig. 45: CPU usage of protocols using different topologies.

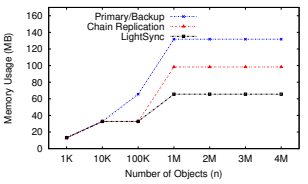


Fig. 46: Memory usage of protocols using different topologies.

node). Our measurement study reveals that the original object sync protocol of OpenStack Swift is not well suited to data-intensive scenarios. In particular, when $r > 3$ and $n \gg 1000$, the object sync delay is unacceptably long and the network overhead is unnecessarily high. This phenomenon is called the sync bottleneck problem. In addition, this problem is considerably aggravated in the presence of data updates and node failures, and cannot be fundamentally solved by increasing the number of sync threads. Guided by an in-depth investigation into the source code of OpenStack Swift-like systems, we design and implement a novel protocol, named LightSync, to practically solve the sync bottleneck problem. Both theoretical analysis and real-world experiments confirm the efficacy of LightSync.

ACKNOWLEDGMENTS

This work is supported by the High-Tech R&D Program of China ("863–China Cloud" Major Program) under grant 2015AA01A201, and the NSFC under grants 61471217, 61432002, 61632020 and 61472337. Thierry Titcheu is supported by the AFR PhD Grant of the National Research Fund, Luxembourg. Ennan Zhai is partly supported by the NSF under grants CCF-1302327 and CCF-1715387.

REFERENCES

[1] N. Bonvin, T. G. Papaioannou, and K. Aberer, "A Self-organized, Fault-tolerant and Scalable Replication Scheme for Cloud Storage," in *Proc. of SoCC*. ACM, 2010.

[2] I. Drago, M. Mellia, M. Munafò, A. Sperotto, R. Sadre, and A. Pras, "Inside Dropbox: Understanding Personal Cloud Storage Services," in *Proc. of IMC*. ACM, 2012.

[3] I. Drago, E. Bocchi, M. Mellia, H. Slatman, and A. Pras, "Benchmarking Personal Cloud Storage," in *Proc. of IMC*. ACM, 2013.

[4] Z. Li, C. Wilson, Z. Jiang, Y. Liu, B. Zhao, C. Jin, Z.-L. Zhang, and Y. Dai, "Efficient Batched Synchronization in Dropbox-like Cloud Storage Services," in *Proc. of Middleware*. ACM, 2013.

[5] Z. Li, Z.-L. Zhang, and Y. Dai, "Coarse-grained Cloud Synchronization Mechanism Design May Lead to Severe Traffic Overuse," *Journal of Tsinghua Science and Technology*, vol. 18, no. 3, pp. 286–297, 2013.

[6] Z. Li, C. Jin, T. Xu, C. Wilson, Y. Liu, L. Cheng, Y. Liu, Y. Dai, and Z.-L. Zhang, "Towards Network-level Efficiency for Cloud Storage Services," in *Proc. of IMC*. ACM, 2014.

[7] Z. Li, X. Wang, N. Huang, M. A. Kaafar, Z. Li, J. Zhou, G. Xie, and P. Steenkiste, "An Empirical Analysis of a Large-scale Mobile Cloud Storage Service," in *Proc. of IMC*. ACM, 2016.

[8] Q. Zhang, Z. Li, Z. Yang, S. Li, S. Li, Y. Guo, and Y. Dai, "DeltaCFS: Boosting Delta Sync for Cloud Storage Services by Learning from NFS," in *Proc. of ICDCS*. IEEE, 2017.

[9] H. Xiao, Z. Li, E. Zhai, T. Xu, Y. Li, Y. Liu, Q. Zhang, and Y. Liu, "Towards Web-based Delta Synchronization for Cloud Storage Services," in *Proc. of FAST*. USENIX, 2018.

[10] E. Jinlong, Y. Cui, P. Wang, Z. Li, and C. Zhang, "CoCloud: Enabling Efficient Cross-Cloud File Collaboration based on Inefficient Web APIs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 1, pp. 56–69, 2018.

[11] S. Gilbert and N. A. Lynch, "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services," *SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002.

[12] "OpenStack Swift Tutorial SAIO - Swift All In One," http://docs.openstack.org/developer/swift/development_saio.html.

[13] "OpenStack Storage Tutorial," http://storageconference.us/2011/Presentations/Tutorial/4.McKenty.pdf.

[14] "OpenStack Installation Guide for Ubuntu 14.04," http://docs.openstack.org/icehouse/install-guide/install/apt/content.

[15] "Storage Policies-SwiftStack Documentation," https://www.swiftstack.com/docs/admin/cluster_management/policies.html.

[16] M. Zhong, K. Shen, and J. I. Seiferas, "Replication Degree Customization for High Availability," in *Proc. of EuroSys*. ACM, 2008.

[17] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan, "Availability in Globally Distributed Storage Systems," in *Proc. of OSDI*. USENIX, 2010.

[18] "The Active Anti-entropy Component of Riak," http://docs.basho.com/riak/latest/theory/concepts/aae.

[19] "The Anti-entropy Node Repair Component of Cassandra," http://docs.datastax.com/en/cassandra/2.1/cassandra/operations/ops_repair_nodes_c.html.

[20] F. J. Reed and Benjamin, *ZooKeeper : Distributed Process Coordination*. O'Reilly Media, Inc., 2013.

[21] J. Stribling, Y. Sovran, I. Zhang, X. Pretzer, J. Li, M. F. Kaashoek, and R. Morris, "Flexible, Wide-Area Storage for Distributed Systems with WheelFS," in *Proc. of NSDI*. USENIX, 2009.

[22] R. Guerraoui, D. Kostic, R. R. Levy, and V. Quema, "A High Throughput Atomic Storage Algorithm," in *Proc. of ICDCS*. IEEE, 2007.

[23] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. Mckelvie, Y. Xu, S. Srivastav, J. Wu, and H. Simitci, "Windows Azure Storage: a Highly Available Cloud Storage Service with Strong Consistency," in *Proc. of SOSP*. ACM, 2011.

[24] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser, "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System," in *Proc. of SOSP*. ACM, 1995.

[25] D. K. Gifford, "Weighted Voting for Replicated Data," in *Proc. of SOSP*. ACM, 1979.

[26] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web," in *Proc. of STOC*. ACM, 1997.

[27] G. Chen and Z. Li, *Peer-to-Peer Network: Structure, Application and Design*. Tsinghua University Press, 2007.

[28] "SwiftStack Benchmark Suite (ssbench) Project," http://github.com/swiftstack/ssbench.

[29] Y. Cheng, A. Gupta, and A. Butt, "An In-memory Object Caching Framework with Adaptive Load Balancing," in *Proc. of EuroSys*. ACM, 2015.

[30] R. C. Merkle, "Protocols for Public Key Cryptosystems," in *Proc. of S&P*. IEEE, Apr. 1980.

[31] "Proxy Configuration," https://docs.openstack.org/icehouse/config-reference/content/proxy-server-configuration.html.

[32] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "Distributed Systems (2Nd Ed.)." ACM Press/Addison-Wesley Publishing Co., 1993, ch. The Primary-backup Approach, pp. 199–216.

[33] R. Van Renesse and F. B. Schneider, "Chain Replication for Supporting High Throughput and Availability." in *Proc. of OSDI*. USENIX, 2004.

[34] T. T. Chekam, E. Zhai, Z. Li, Y. Cui, and K. Ren, "On the Synchronization Bottleneck of OpenStack Swift-like Cloud Storage Systems," in *Proc. of INFOCOM*. IEEE, 2016.
[35] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis, "Zeno: Eventually Consistenct Byzantine-Fault Tolerance," in *Proc. of NSDI*. USENIX, 2009.
[36] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS," in *Proc. of SOSP*. ACM, 2011.
[37] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish, "Depot: Cloud Storage with Minimal Trust," *ACM Transactions on Computer Systems*, vol. 29, no. 4, p. 12, 2011.
[38] "OpenStack Swift Improved Object Replicator," http://wiki.openstack.org/wiki/Swift-Improved-Object-Replicator.
[39] A. Corradi, M. Fanelli, and L. Foschini, "VM Consolidation: A Real Case based on OpenStack Cloud," *Future Generation Computer Systems*, vol. 32, pp. 118–127, 2014.
[40] E. Zhai, R. Chen, D. I. Wolinsky, and B. Ford, "Heading Off Correlated Failures through Independence-as-a-Service," in *Proc. of OSDI*. USENIX, 2014.
[41] ——, "An Untold Story of Redundant Clouds: Making Your Service Deployment Truly Reliable," in *Proc. of HotDep*. ACM, 2013.
[42] Q. Zhang, S. Li, Z. Li, Y. Xing, Z. Yang, and Y. Dai, "CHARM: A Cost-efficient Multi-cloud Data Hosting Scheme with High Availability," *IEEE Transactions on Cloud Computing*, vol. 3, no. 3, pp. 372–386, 2015.
[43] Z. Lai, Y. Cui, M. Li, Z. Li, N. Dai, and Y. Chen, "TailCutter: Wisely Cutting Tail Latency in Cloud CDN under Cost Constraints," in *Proc. of INFOCOM*. IEEE, 2016.
[44] G. Wu, F. Liu, H. Tang, K. Huang, Q. Zhang, Z. Li, B. Y. Zhao, and H. Jin, "On the Performance of Cloud Storage Applications with Global Measurement," in *Proc. of IWQoS*. IEEE/ACM, 2016.

Mingkang Ruan is an M.Eng. student at the School of Software, Tsinghua University, Beijing, China. He received the B.Sc. degree in Software Engineering from Sun Yat-sen University, Guangzhou, China in 2014. His research areas mainly include cloud computing/storage, big data analysis, and natural language processing.

Thierry Titcheu is a Ph.D. student at the Interdisciplinary Centre for Security, Reliability and Trust, the University of Luxembourg. He received the B.Sc. degree in Computer Science and Technology from the University of Science and Technology of China in 2013, and the M.Eng. degree in Software Engineering from the School of Software, Tsinghua University in 2015. His research areas comprise cloud computing/storage, distributed systems, and so forth.

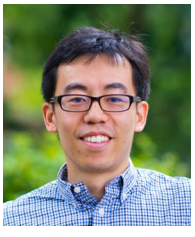
Ennan Zhai is currently an associate research scientist at the Computer Science Department of Yale University. He received the Ph.D. and M.Phil. degrees from Yale University in 2015 and 2014. His research interests mainly include distributed system, applied cryptography, and software verification.

Zhenhua Li is an assistant professor at the School of Software, Tsinghua University. He received the B.Sc. and M.Sc. degrees from Nanjing University in 2005 and 2008, and the Ph.D. degree from Peking University in 2013, all in computer science and technology. His research areas cover cloud computing/storage/download, big data analysis, content distribution, and mobile Internet.

Yao Liu is an assistant professor at the Department of Computer Science, Binghamton University. She received the B.S. degree in computer science from Nanjing University and the Ph.D. degree in computer science from George Mason University. Her research areas include Internet mobile streaming, multimedia computing, Internet measurement and content delivery, and cloud computing.

Jinlong E received the B.E. and M.Sc. degrees in computer software from Nankai University, Tianjin, China, in 2007 and 2011, respectively. He is currently working towards the Ph.D. degree in computer science and technology at Tsinghua University. His current research interests include cloud storage, content distribution, scheduling, and mobile cloud computing.

Yong Cui is a professor at the Department of Computer Science and Technology, Tsinghua University. He received the B.Eng. and Ph.D. degrees both in Computer Science and Technology from Tsinghua University, respectively in 1999 and 2004. He served or serves at the editorial boards on IEEE TPDS, TCC, and Internet Computing. His major research interests include mobile cloud computing and network architecture.



Hong Xu is an assistant professor at the Department of Computer Science, City University of Hong Kong. He received the M.A.Sc. and Ph.D. degrees from the Department of Electrical and Computer Engineering, University of Toronto. His research interests include data center networking, NFV, and cloud computing. He was the recipient of an Early Career Scheme Grant from Hong Kong Research Grants Council in 2014.