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

Thierry Titcheu Chekam [1,2], Ennan Zhai [3], Zhenhua Li [1*], Yong Cui [4], Kui Ren [5]
[1] School of Software, TNLIST, and KLISS MoE, Tsinghua University
[2] Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg
[3] Department of Computer Science, Yale University
[4] Department of Computer Science and Technology, Tsinghua University
[5] Department of Computer Science and Engineering, SUNY Buffalo
{*tctthierry, ennan.zhai, lizhenhua1983*}*@gmail.com, cuiyong@tsinghua.edu.cn, kuiren@buffalo.edu*

***Abstract*—As one type of the most popular cloud storage services, OpenStack Swift and its follow-up systems replicate each data object across multiple storage nodes and leverage *object sync protocols* to achieve high availability 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 object sync process seems to perform well.**

To deep understand object sync protocols, we first make a lab-scale OpenStack Swift deployment and run experiments with various configurations. We discover 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. This phenomenon is referred to as the *sync bottleneck problem*.

Then, to explore the root cause, we review the source code of OpenStack Swift and find that its object sync protocol utilizes a fairly simple and network-intensive approach to check the consistency among replicas of objects. In particular, each storage node is required to periodically multicast the hash values of all its hosted objects to all the other replica nodes. Thus in a sync round, the number of exchanged hash values per node is $\Theta(n \times r)$.

Further, to tackle the problem, we propose a lightweight object sync protocol called *LightSync*. It remarkably reduces the sync overhead by using two 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. Its design provably reduces the per-node network overhead from $\Theta(n \times r)$ to $\Theta(\frac{n}{h})$. In addition, we have implemented LightSync as an open-source patch and adopted it to OpenStack Swift, thus reducing sync delay by up to 28.8× and network overhead by up to 14.2×.

I. Introduction

Today's cloud storage services, *e.g.*, Amazon S3 [1], Google Cloud Storage [2], Windows Azure Blob [3], and Rackspace Cloud Files [4], provide highly available and robust infrastructure support to upper-layer applications [5]–[11]. As one type of the most popular open-source cloud storage services, OpenStack Swift [12] and its follow-up systems such as Riak S2 [13] and Apache Cassandra [14] (called *OpenStack Swift-like systems*) have been used by many organizations and companies like Rackspace, UnitedStack, Sina Weibo, eBay, Instagram, Reddit, and AiMED Stat [15].

In order to offer high data availability 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 [16] 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 the regular configuration (as proposed in most existing tutorials and demonstrations [17]–[19]), *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.

On the other hand, 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. For example, when $r = 5$ and $n = 4M$, the sync delay is as long as 58 minutes and there are 1.53 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 in Riak S2 and Cassandra. This 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 [20], [21].

Driven by the 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

* Corresponding author.

978-1-4673-9953-1/16/$31.00 ©2016 IEEE

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 data objects included in this partition. Therefore, each sync message contains h hash values and each hash value corresponds to a data object.

More in detail, as one storage node can host multiple ($\frac{n}{h}$) 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 [22]) and Cassandra (the anti-entropy node repair component [23]).

To tackle the sync bottleneck problem, we propose a lightweight sync protocol called *LightSync*. At the heart of LightSync lie two novel building blocks: *Hashing of Hashes* (HoH) and *Circular Hash Checking* (CHC).

- 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.

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.

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 *http://github.com/LightSync/patch-openstack-swift*. Real-world experimental results illustrate that LightSync remarkably reduces the sync delay by up to 28.8 times and the network overhead by up to 14.2 times.

To summarize, this paper makes the following contributions:

1) We (are the first to) discover the sync bottleneck problem of OpenStack Swift-like systems by conducting various experiments on our lab-scale testbed (§III).
2) We reveal the key factors that lead to the problem by investigating the source code of OpenStack Swift (§IV).
3) We propose an efficient and practical object sync protocol, named LightSync, to address the problem (§V).
4) We implement an open-source LightSync patch which is suited to general OpenStack Swift-like systems (§V-D).
5) After the patch is applied to real-world deployments, the results illustrate that LightSync is capable of significantly improving the object sync performance (§VI).

II. 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.

A. Design Goals of OpenStack Swift

Eventual consistency. 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].

High availability. OpenStack Swift provides availability (and durability) by replicating each object across multiple (3 by default) storage nodes. To ensure high availability, OpenStack Swift places each replica of a given object "as unique as possible", *i.e.*, at best effort replicating each object across *independent* nodes. OpenStack Swift achieves this goal by introducing *zones*. A zone is a collection of many (physical) nodes, and there are no such correlations as shared devices or software-level dependencies among different zones.

B. 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 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 o's locations on the storage nodes, and then randomly forwards the request to one of the storage nodes hosting o. 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 $\lfloor r/2 \rfloor + 1$ of them reply with "successful write", the update is taken as successful.

C. Partition and Synchronization

Like many popular storage systems, OpenStack Swift organizes data partitions through consistent hashing (or says DHT, distributed hash table) [25], [26]. 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) objects belonging to the partition. According to the working principle of consistent hashing, h dynamically changes with the system scale, particularly the number of storage nodes, the number of data objects, and the max number of partitions.

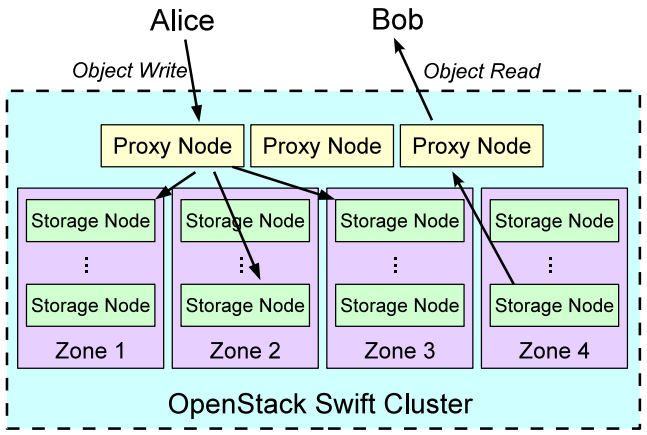

Fig. 1: OpenStack Swift architecture. Alice sends an object write request to a proxy node, and then all the replicas of the target object are (eventually) updated. When Bob wants to read an object, he first sends an object read request to a proxy node. Then, one of the storage nodes hosting Bob's requested object responds.

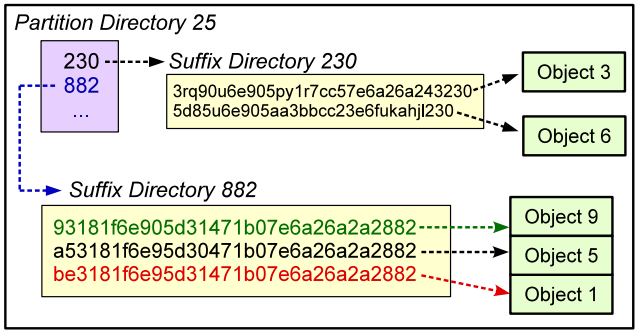

Fig. 2: An example for a data partition's structure. Each suffix directory includes all the hash values (of data objects) whose last three characters are equal to the suffix.

Each partition is replicated r times on the logical ring, physically mapped to r different locations in the 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 an unique identifier, *i.e.*, an MD5 hash value of the object's path. An MD5 hash value is made up of 32 hex(adecimal) characters. Each partition has a directory (file) called *hashes.pkl*, which is used to list the hash values of all the objects in this partition. Further, a directory contains multiple *suffix directories*. Each suffix directory includes all the hash values (of data objects) whose last three characters are equal to the suffix. 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 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, from which one can easily figure out which replica(s) have a different version status.

III. 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.

A. 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 (called *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 (called *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 [19]), 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 [27]), a benchmarking tool for automatically generating intensive OpenStack Swift workloads. Each data object is filled with random bytes between 64 KB and 128 KB (we will prove in §IV that the object sync performance of OpenStack Swift is generally irrelevant to the concrete content and size of each data object).

B. Sync Delay

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 $r = 2, 3, 4, 5$. 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, the sync delay 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 that 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. 4.

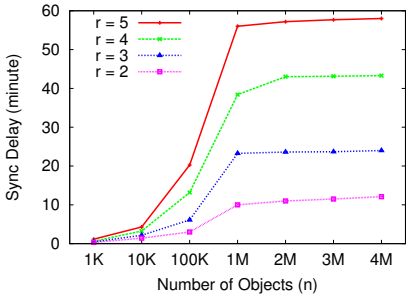

Fig. 3: Sync delay.

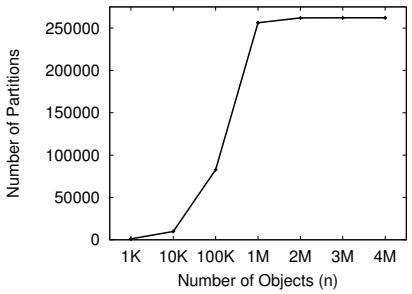

Fig. 4: Number of partitions (p).

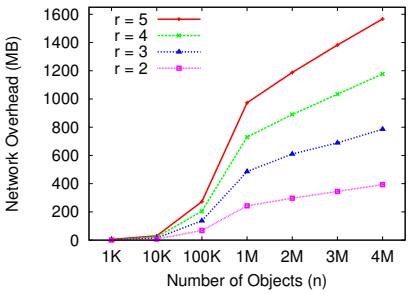

Fig. 5: Network overhead.

As mentioned in §III-A, 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 §IV.

C. Network Overhead

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.

The measurement results in Fig. 5 show that the network overhead (per node in a sync round) increases with larger n and/or r. More importantly, the four curves in Fig. 5 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 1.53 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 $n > 1M$.

D. The Sync Bottleneck Problem

Based on the above results, 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).

IV. Root Cause

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.

A. Object Sync Process in OpenStack Swift

The relevant source code of OpenStack Swift (the Icehouse version [1]) is mainly included in the following files:

Path	File
/usr/lib/python2.7/dist-packages/swift/obj	diskfile.py mem_diskfile.py replicator.py server.py
/usr/lib/python2.7/dist-packages/swift/proxy	server.py
/usr/lib/python2.7/dist-packages/swift/proxy/controller	base.py obj.py
/usr/lib/python2.7/dist-packages/swift/common	bufferedhttp.py http.py
/usr/lib/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. 6 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 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. If any node (including the leader) is not having the latest version of P, it would try to update its version by sending a request to the other side. 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 [22]) and Cassandra (the anti-entropy node repair component [23]).

B. Network Overhead Analysis

While we have observed the sync bottleneck problem from our case study (§III), we hope to quantitatively understand how expensive the current object sync protocol of OpenStack Swift is in principle/theory. As §III has clearly illustrated that it is the enormous network overhead that leads to the sync bottleneck problem, we focus on analyzing the network overhead.

[1]We have also examined the latest Kilo version of OpenStack Swift and find that the concerned source code is generally unchanged.

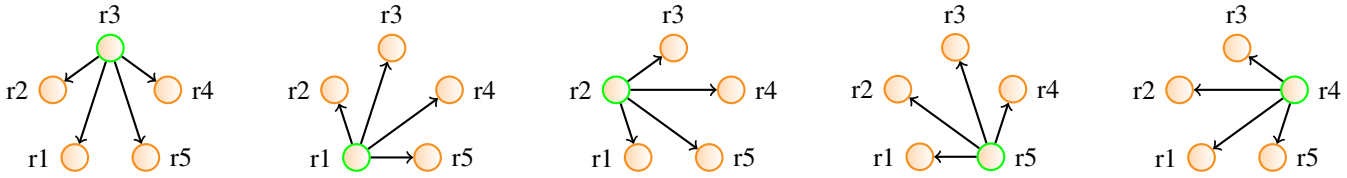

Fig. 6: An example for a complete object sync process. Note that the five sub-processes run in parallel rather than in sequence.

It is straightforward to deduce from Fig. 6 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 corresponding to the h data objects included in P. 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 $\Theta(n \times r)$.

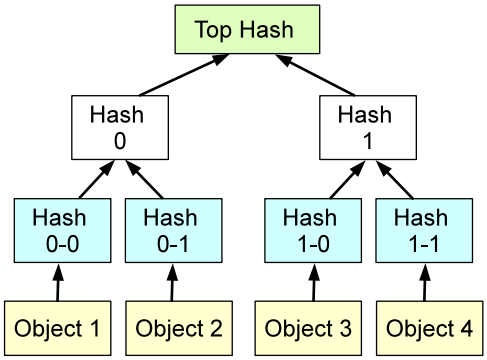

Fig. 7: An example for the Merkle tree data structure.

V. LightSync: Design and Implementation

Guided by the thorough understanding of the object sync process of OpenStack Swift in §IV, we design a lightweight 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.

A. LightSync Overview

LightSync is designed to replace the original object sync protocols in current OpenStack Swift-like systems, so as to significantly reduce the sync delay and network overhead. It derives the desired property from the following two novel building blocks.

First, LightSync employs the Hashing of Hashes (HoH) mechanism (§V-B) 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 (§V-C) 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).

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

B. Hashing of Hashes (HoH)

Preliminary: Merkle tree. A Merkle tree [28] 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. 7, 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. The above steps have to be repeated between A and B until the leaves of the trees are reached. Consequently, the network complexity of data checking using the Merkle tree is in $\Theta(\log N)$, where N is the number of nodes in the Merkle tree.

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), which records the hash values of all the objects in P. Then, HoH computes the MD5 hash values of all the suffix hashes in P one by one (as demonstrated in the Fig. 8 example). 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).

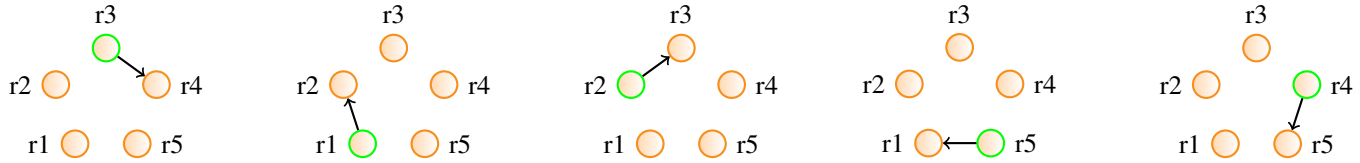

Fig. 9: An example for a complete LightSync working process. The five sub-processes run in parallel rather than in sequence.

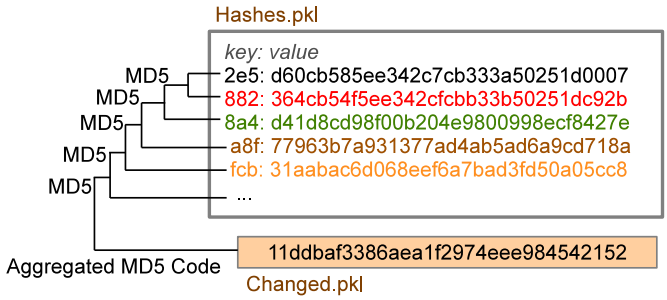

Fig. 8: Hashing of hashes for a data partition.

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 is found inconsistent between two storage nodes, both of them need to determine which one is storing an older version by checking the timestamp recorded in each *changed.pkl* file. For the storage node storing an older version, it needs to locally locate which suffix directory is inconsistent. Since we use the Merkle tree structure, this storage node can easily find the older suffix directory within $O(\log s)$ steps, where s is the number of suffix directories in the corresponding data partition.

Compared with the original design of OpenStack Swift, HoH uses a single but representative hash value to replace a large collection of hash values, thus effectively reducing the size of each sync message by nearly h times.

C. 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 §II-C).

Suppose P has five replicas, and r_i 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 replicas of P, 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. 9, when r_3 wants to check the consistency of P, it only sends a sync message to r_4 rather than r_1, r_2, r_4 and r_5 (as in Fig. 6). After each replica node finishes sending a sync message to its clockwise neighbor, we say a CHC object sync process (or a CHC sync round) is completed. Formally, Algorithm 1 describes how CHC works.

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 synchronizes with its clockwise neighbor by
        comparing the timestamps of their respective
        versions of P;
```

D. Implementation

We implement LightSync for OpenStack Swift (the Icehouse version) in Python, without introducing any additional library. Specifically, we develop HoH + CHC by adding, deleting, or modifying over 500 lines of Python codes mostly located in two files: `.../swift/obj/replicator.py` and `.../swift/obj/diskfile.py`. Here we use '`...`' to represent `/usr/lib/python2.7/dist-packages`.

In detail, regarding the implementation of HoH, we disable the partition hash function for generating the original *hashes.pkl* file, and make our designed *changed.pkl* file effective by modifying the function `invalidate_hash()` located in `.../swift/obj/diskfile.py`. Besides, we recompute the partition hash and update the corresponding *changed.pkl* file when the suffix hashes are computed (refer to §II-C). This is achieved by modifying the function `get_hashes()` in `.../swift/obj/diskfile.py`. In addition, regarding the implementation of CHC, we modify the function `update()` in `.../swift/obj/replicate.py`, so as to enable the object sync process of CHC.

We have published LightSync as an open-source patch to benefit the community and meanwhile request for peer researchers' comments. It can be downloaded via the following link: *http://github.com/LightSync/patch-openstack-swift*.

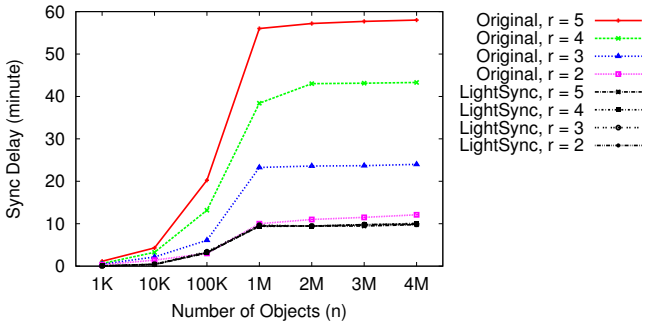

Fig. 10: Sync delay with LightSync in lab-scale experiments. The sync delay with the Original design of OpenStack Swift is also plotted for comparison.

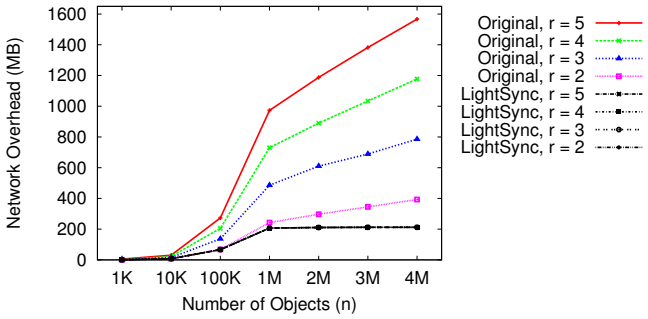

Fig. 11: Network overhead with LightSync in lab-scale experiments. The network overhead with the Original design of OpenStack Swift is also plotted for comparison.

VI. Evaluation

In this section, we first analyze the theoretical network overhead of LightSync based on its design principle in §V. Then, to evaluate the real-world performance of LightSync, we conduct experiments on top of OpenStack Swift equipped with LightSync. Our goal 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.

A. Theoretical Analysis

By comparing Fig. 6 and Fig. 9, 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. 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 $\Theta(\frac{n}{h})$.

B. Experimental Results

To understand the real-world performance of LightSync, we still conduct experiments on our lab-scale OpenStack Swift deployment: five Dell PowerEdge T620 servers (refer to §III-A for each server's detailed configuration).

First, as illustrated in Fig. 10, LightSync remarkably decreases the sync delay of OpenStack Swift — the four curves of LightSync are almost always below those of Original. Here "Original" denotes the original object sync protocol. More importantly, we observe that the sync delay with LightSync is generally regardless of r, owing to the notable power of CHC in avoiding unnecessary sync messages.

Second, as shown in Fig. 11, LightSync also effectively decreases the network overhead of OpenStack Swift. Once again, the four curves of LightSync are below those of Original, and the network overhead with LightSync is regardless of r.

Quantitatively, LightSync reduces the sync delay by $1.0 \sim 28.8$ (5.7 on average) times, and the network overhead by $1.0 \sim 14.2$ (3.6 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 9.8 minutes, and the network overhead is reduced from 1567 to 212 MB.

VII. 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, availability, and performance trade-offs. These stores include Amazon S3, Windows Azure Blob, OpenStack Swift, Riak S2, Cassandra, BigTable [29], Dynamo [30], SimpleDB [31], 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.

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 [32], or by resorting to a last-writer-wins strategy which often results in loss of concurrent updates [33]. Other systems expose multiple values from divergent branches of operation replicas either directly to the client [30], [34] 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 [35]. The agent is responsible for computing all the hash values and coordinating the sync process. Compared with LightSync, the above effort fails to provide quantitative evaluation results in data-intensive deployments. Besides, some other case studies [36], [37] reveal that the VM placement strategy of OpenStack may lead to data availability bottlenecks, but they do not dive deeper into the sync bottleneck problem.

VIII. Conclusion and Future Work

OpenStack Swift-like cloud storage systems have been widely used and studied in recent years. In this paper, we investigate 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 node).

First, we conduct a lab-scale case study which 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, which also occurs in Riak S2 and Cassandra.

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.

There are several important future works that deserve further investigations. In particular, this paper studies the object sync protocol of OpenStack Swift-like systems in a relatively small scale (*i.e.*, lab-scale) with only two key performance metrics (*i.e.*, sync delay and network overhead). We plan to make a large-scale study with comprehensive performance metrics in the near future, including not only sync delay and network overhead, but also CPU utilization, memory consumption, disk I/O burden, data availability, time to consistency, and so forth. Besides, we note that OpenStack Swift provides a parallel optimization option to accelerate the object sync process by increasing the number of sync threads. Nevertheless, throughout this paper, a single sync thread is used during the object sync process. It will be interesting to clarify the influence of parallelism (by using multiple sync threads) on the object sync performance. Moreover, the design of LightSync has to take the issues of possible node failures and object updates into account, which are not addressed in this paper. Specifically, we need to further examine the object sync performance in the presence of node failures or object updates (after all objects are initially created).

Acknowledgment

This work is supported by the High-Tech Research and Development Program of China ("863 – China Cloud" Major Program) under grant 2015AA01A201, the China NSF (National Science Foundation) under grants 61432002 (State Key Program), 61471217, 61422206 and 61120106008, and the CCF-Tencent Open Fund under grant IAGR20150101. Prof. Kui Ren's research is supported in part by the US NSF under grant CNS-1262277.

In addition, we would like to thank Prof. Yunhao Liu at Tsinghua University for his valuable resource support and constructive suggestion.

References

[1] "Amazon S3 (Simple Storage Service)," http://aws.amazon.com/s3.
[2] "Google Cloud Storage Service," http://cloud.google.com/storage.
[3] "Microsoft Windows Azure Blob Storage Service," http://azure.microsoft.com/en-us/services/storage/blobs.
[4] "Rackspace Cloud Files Service," http://www.rackspace.com/cloud/files.
[5] N. Bonvin, T. G. Papaioannou, and K. Aberer, "A Self-organized, Fault-tolerant and Scalable Replication Scheme for Cloud Storage," in *Proc. of the 1st ACM Symposium on Cloud computing (SoCC)*, 2010, pp. 205–216.
[6] I. Drago, M. Mellia, M. Munafò, A. Sperotto, R. Sadre, and A. Pras, "Inside Dropbox: Understanding Personal Cloud Storage Services," in *Proc. of the 12th ACM SIGCOMM/SIGMETRICS Internet Measurement Conference (IMC)*, 2012, pp. 481–494.
[7] I. Drago, E. Bocchi, M. Mellia, H. Slatman, and A. Pras, "Benchmarking Personal Cloud Storage," in *Proc. of the 13th ACM SIGCOMM/SIGMETRICS Internet Measurement Conference (IMC)*, 2013, pp. 205–212.
[8] 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 the 14th ACM/IFIP/USENIX International Middleware Conference (Middleware)*, 2013, pp. 307–327.
[9] 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 the 14th ACM SIGCOMM/SIGMETRICS Internet Measurement Conference (IMC)*, 2014, pp. 115–128.
[10] Y. Cui, Z. Lai, X. Wang, N. Dai, and C. Miao, "QuickSync: Improving Synchronization Efficiency for Mobile Cloud Storage Services," in *Proc. of the 21st Annual International Conference on Mobile Computing and Networking (MobiCom)*. ACM, 2015, pp. 592–603.
[11] Y. Cui, Z. Lai, and N. Dai, "A First Look at Mobile Cloud Storage Services: Architecture, Experimentation and Challenge," *IEEE Network*, 2016.
[12] "OpenStack Swift Object Storage Service," http://swift.openstack.org.
[13] "Riak S2 Object Storage Software," http://basho.com/products/riak-s2.
[14] "Apache Cassandra Database Service," http://cassandra.apache.org.
[15] "AiMED Stat Uses Riak," http://basho.com/posts/business/aimed-stat-uses-riak-to-increase-their-competitive-advantage.
[16] 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.
[17] "OpenStack Swift Tutorial SAIO - Swift All In One," http://docs.openstack.org/developer/swift/development_saio.html.
[18] "OpenStack Storage Installing Tutorial," http://storageconference.us/2011/Presentations/Tutorial/4.McKenty.pdf.
[19] "OpenStack Installation Guide for Ubuntu 14.04," http://docs.openstack.org/icehouse/install-guide/install/apt/content.
[20] 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 the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2010, pp. 61–74.
[21] M. Zhong, K. Shen, and J. I. Seiferas, "Replication Degree Customization for High Availability," in *Proc. of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys)*, 2008, pp. 55–68.
[22] "The active anti-entropy component of Riak," http://docs.basho.com/riak/latest/theory/concepts/aae.
[23] "The anti-entropy node repair component of Cassandra," http://docs.datastax.com/en/cassandra/2.1/cassandra/operations/ops_repair_nodes_c.html.
[24] D. B. Terry, M. Theimer, K. Petersen, A. J. Demers, M. Spreitzer, and C. Hauser, "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System," in *Proc. of the 15th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 1995, pp. 172–182.
[25] 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 the 29th Annual ACM Symposium on Theory of Computing (STOC)*, 1997, pp. 654–663.
[26] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4, pp. 149–160, 2001.

[27] "SwiftStack Benchmark Suite (ssbench) project," http://github.com/swiftstack/ssbench.
[28] R. C. Merkle, "Protocols for Public Key Cryptosystems," in *Proc. of the 1st IEEE Symposium on Security and Privacy (S&P), Oakland, CA, US*, 1980.
[29] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber, "BigTable: A Distributed Storage System for Structured Data," in *Proc. of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006, pp. 205–218.
[30] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-Value Store," in *Proc. of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, 2007, pp. 205–220.
[31] "Amazon SimpleDB," http://aws.amazon.com/simpledb.
[32] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis, "Zeno: Eventually Consistenct Byzantine-Fault Tolerance," in *Proc. of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009, pp. 169–184.
[33] 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 the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011, pp. 401–416.
[34] 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 (TOCS)*, vol. 29, no. 4, p. 12, 2011.
[35] "OpenStack Swift Improved Object Replicator," http://wiki.openstack.org/wiki/Swift-Improved-Object-Replicator.
[36] 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.
[37] E. Zhai, R. Chen, D. I. Wolinsky, and B. Ford, "Heading Off Correlated Failures through Independence-as-a-Service," in *Proc. of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 317–334.