Recent Activities Involving OpenStack Swift

Kota Tsuyuzaki and Masahiro Shiraishi

Abstract

OpenStack Swift is popular open source software used to build very large-scale storage systems. OpenStack Swift was originally released by Rackspace, and it has been developed with community effort over the last five years by developers from all over the world. The OpenStack Swift community recently introduced some major features such as global cluster management and erasure code capability. NTT has also contributed to the OpenStack Swift community. In addition, NTT developed a proprietary secret sharing engine called Super High-speed Secret Sharing, which achieves data encryption that is compatible with erasure code. In this article, we introduce these developments and discuss NTT's activities in the Swift community.

Keywords: OpenStack Swift, distributed storage system, object storage

1. Introduction

OpenStack Swift (hereinafter referred to as either Swift or OpenStack Swift) [1] is one of the most common types of open source software (OSS) used to build very large-scale storage systems with Hypertext Transfer Protocol (HTTP) based application programming interfaces (APIs). OpenStack Swift was originally released by Rackspace [2], and developers all over the world have been collaborating on it for five years with great community effort. OpenStack Swift is now used in production in various ways. For example, Rackspace and HP [3] are using OpenStack Swift for their own public cloud storage services. Additionally, NTT DOCOMO is using OpenStack Swift [4] as a 7-petabyte private storage system for its cloud mail backup system.

Swift's key features for production use cases have three main characteristics.

(1) HTTP based APIs

Swift supports HTTP based APIs using HTTP verbs such as PUT, GET, and DELETE for uploading/downloading and deleting data. This way of using the cloud storage system is an easy way to share data among cloud systems because developers do not have to worry about the actual data location, and they can retrieve their own data whenever they want. Furthermore, the HTTP based APIs are quite useful for handling binary large objects because in recent commonly used Internet technologies, clients such as web browsers and smartphones transfer content via HTTP on the Internet.

(2) High reliability

Swift has the capability to ensure that data are stored with high reliability and to prevent significant data loss caused by various events (e.g., disk failure). To prevent the data loss, Swift has a consistency engine called an object-replicator that works to find a lack of data redundancy (three replicas in default) and consistency. Then it copies valid data if some replicas are lost or mismatched in the cluster.

Swift also has the capability to store each replica in as unique a failure domain as possible. For example, Swift never allows more than one replica to be placed on the same disk to prevent a failure of one device resulting in reduced data redundancy. These mechanisms ensure that Swift has high reliability and high durability.

(3) Scale out

An important item to consider when deploying a large system is scalability. Swift can scale with no single point of failure. A typical example of a Swift

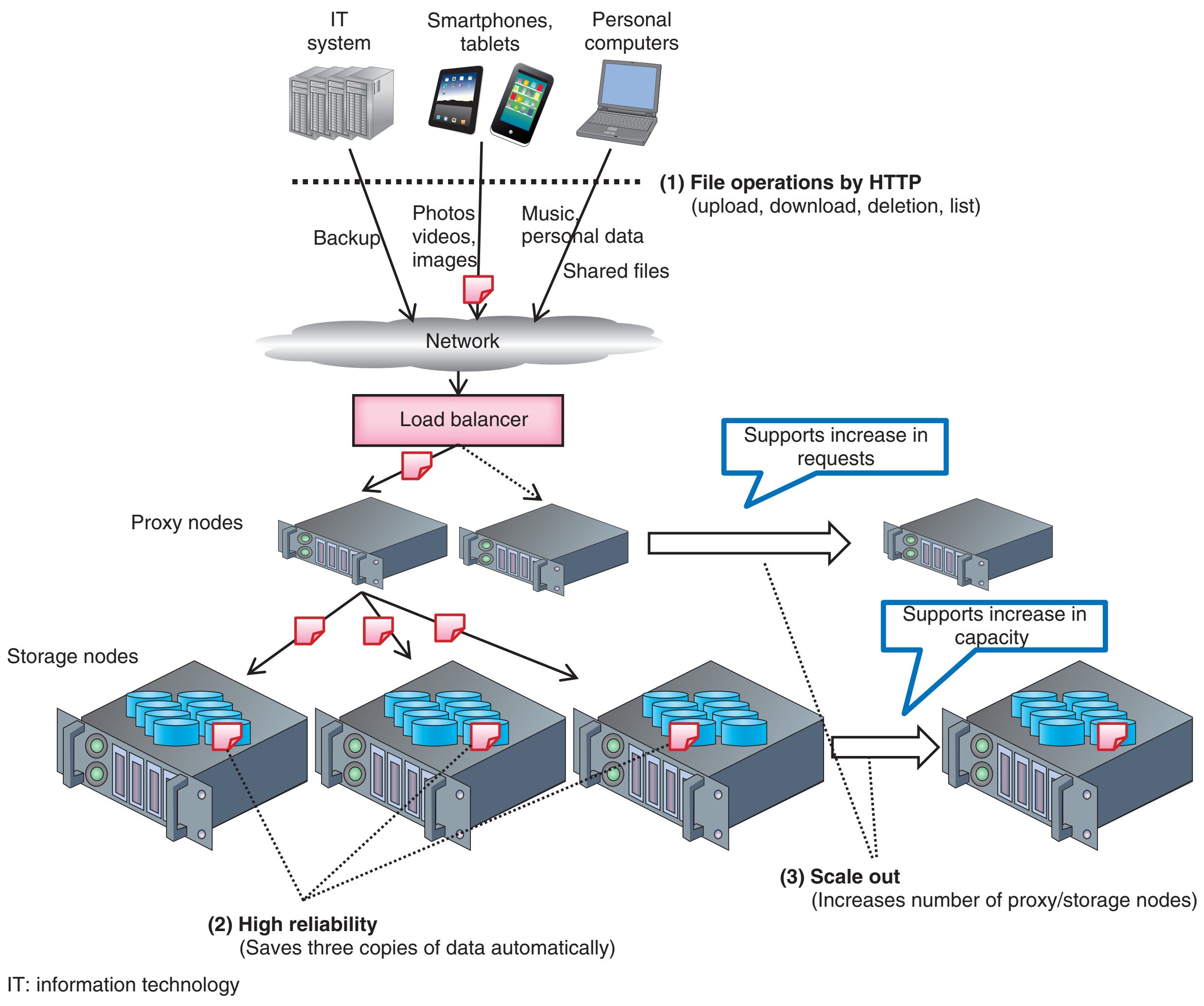

IT system
Smartphones, tablets
Personal computers
(1) File operations by HTTP
(upload, download, deletion, list)
Backup
Photos videos, images
Music, personal data
Shared files
Network
Load balancer
Supports increase in requests
Proxy nodes
Supports increase in capacity
Storage nodes
(3) Scale out
(Increases number of proxy/storage nodes)
(2) High reliability
(Saves three copies of data automatically)
IT: information technology

Fig. 1. Example of OpenStack Swift cluster configuration.

cluster configuration is shown in **Fig. 1**. In this example, the system has proxy nodes that receive requests from clients and storage nodes that actually store data. This makes for a highly extendible cluster architecture since proxy nodes can be added if the number of requests becomes excessive, while storage nodes can be added if storage capacity becomes insufficient.

In addition to these characteristics, the Swift community has been working on building some major features, and some of them were achieved in recent releases. In this article, we introduce two of these features: global cluster management and erasure code capability. We also describe the secret sharing engine and how it is used with Swift. This secret sharing engine makes it possible to store data with encryption; it was developed by NTT to be compatible with OpenStack Swift erasure code capability.

2. Global cluster management

Some companies may want to deploy OpenStack Swift in more than one datacenter to prevent a large data loss caused by a disaster such as an earthquake, fire, or tsunami. Our customers must also consider their requirements for disaster recovery. However, geographically distributed clusters sometimes have physical issues with network latency. In the worst case, the network latency will degrade the input/output performance of the storage system.

The Swift community has been working to improve

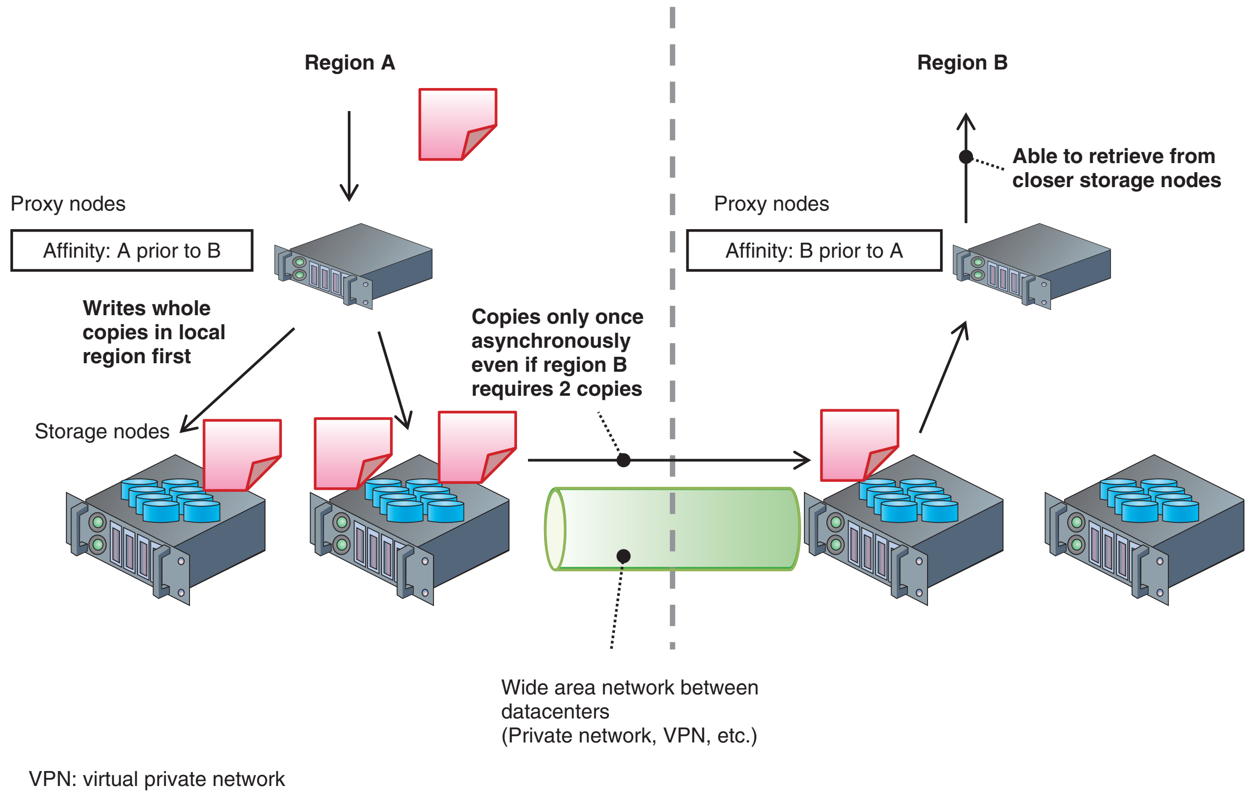

Region A
Region B
Proxy nodes
Affinity: A prior to B
Proxy nodes
Affinity: B prior to A
Able to retrieve from closer storage nodes
Writes whole copies in local region first
Copies only once asynchronously even if region B requires 2 copies
Storage nodes
Wide area network between datacenters (Private network, VPN, etc.)
VPN: virtual private network

Fig. 2. Global cluster.

the inner architecture in order to reduce the effect of network latency. This is the feature known as a global cluster.

Swift employs the concepts of *region* and *affinity* to achieve the global cluster. A region is a domain that defines which datacenters the actual hard drives belong to. Affinity is an attribute that defines the priority among regions seen by the proxy-server.

By defining these two factors, Swift can access regions that are as local as possible. In an uploading sequence, Swift will write all of the replicas into unique devices in the closest region, and the replicator will then copy the replicas asynchronously to nodes in another region. In a downloading sequence, Swift will read the object from nodes in the closest region first. If all nodes are offline in the region, Swift will try to retrieve the object from another region.

For example, when we define two regions A and B and define B prior to A by its affinity at the proxy-server in region B, Swift will try to get the object from the devices in region B prior to region A, as shown in **Fig. 2**. As described in section 1, Swift stores three copies of replicated data in devices in domains (regions, zones, Internet protocol addresses, and devices) that are as unique as possible so that the global cluster mechanism efficiently retrieves data from the device in the closest unique domain.

Furthermore, Swift now has the ability to reduce the number of data transfers among regions by copying the replica only once between regions. This feature was developed mainly by NTT with the Swift community.

3. Erasure code capability

Since the first major release of OpenStack Swift, Swift has employed the replicated model to protect stored objects with high availability and durability. However, nowadays we need a scheme that is more efficient than the replicated model in order to reduce the amount of hardware (especially the number of hard disks) and the associated costs. Erasure code is a way to reduce the volume of hardware by creating parity fragments, which refers to smaller amounts of redundant data than the replicated model. This scheme uses redundant arrays of inexpensive disks

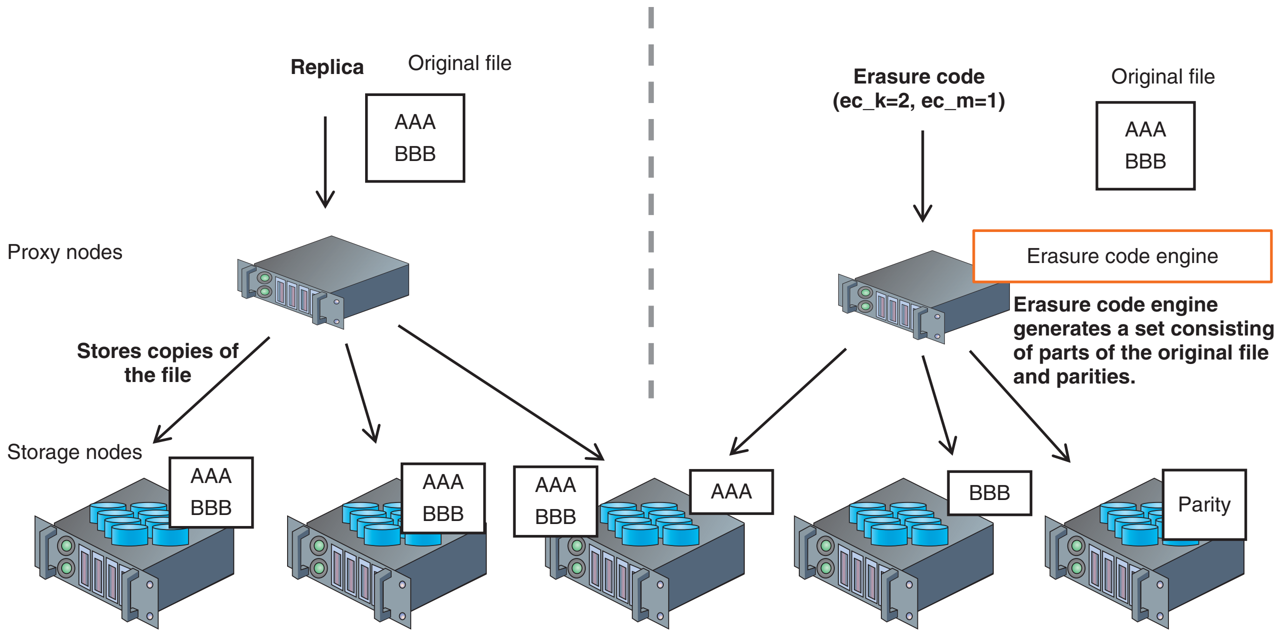

Replica
Original file
AAA
BBB
Proxy nodes
Stores copies of the file
Storage nodes
AAA
BBB
AAA
BBB
AAA
BBB
Erasure code (ec_k=2, ec_m=1)
Original file
AAA
BBB
Erasure code engine
Erasure code engine generates a set consisting of parts of the original file and parities.
AAA
BBB
Parity

Fig. 3. Example of Swift erasure code for a PUT request.

(RAID).

In an erasure code scheme as shown in **Fig. 3**, Swift slices the original data into "ec_k" data fragments that consist of an aggregation of split original data. Swift also creates "ec_m" parity fragments, which are mathematically redundant data of the data fragments. In the erasure code scheme, it is possible to rebuild the original data from any "ec_k" fragments among all the fragments.

For example, we use ec_k=2 and ec_m=1 parameters for Swift erasure code; Swift will create two data fragments and one parity fragment and store a total of three fragments in three unique devices. When a user requests Swift to retrieve original data, Swift responds by rebuilding the original data from any two of the stored fragments.

For the erasure code feature, the Swift community added a new consistency engine called object-reconstructor. It works almost the same as the object-replicator to maintain high durability of data redundancy, but the difference is that it enables the reconstruction of a unique fragment (not a copy) for the node.

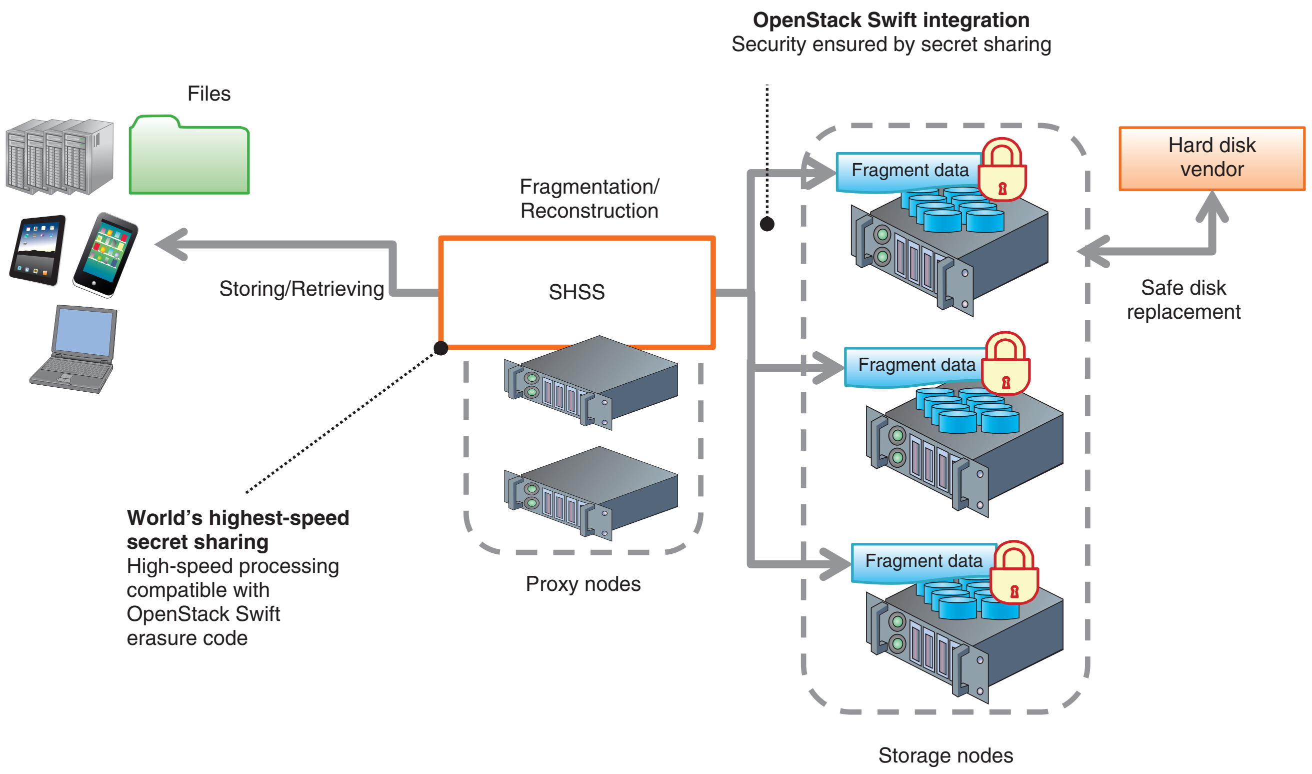
The major players in the Swift community have been making an effort to implement this feature for approximately a year and a half, and it was finally published as a beta version in the most recent OpenStack release. We are hoping that Swift erasure code will be ready to release as a production-ready version with the code name Liberty in the next release.

4. Secret sharing

Solutions for security, data durability, and increasing data volume are more important than ever in the information technology business market, since the amount of highly confidential information will continue to grow. To manage confidential data, NTT developed a proprietary secret sharing engine called Super High-speed Secret Sharing (SHSS) [5] as a pluggable backend for the OpenStack Swift erasure code, and it is expected to be used in secure storage products as the amount of confidential data increases.

The SHSS engine enables OpenStack Swift to encrypt fragments and to reconstruct the plain data from the encrypted fragments (**Fig. 4**). The reconstruction requires a number of fragments, and OpenStack Swift stores the fragments to unique disks in the same way as erasure code. This mechanism prevents the system from reconstructing original data from insufficient fragments, and it reduces the risk of information leakage when broken physical drives are replaced by hard drive vendors.

In addition, the main advantage of SHSS is that it has the world's fastest fragmentation and reconstruction performance, which allows OpenStack Swift to


OpenStack Swift integration
Security ensured by secret sharing
Files
Fragmentation/
Reconstruction
Storing/Retrieving
SHSS
Fragment data
Fragment data
Fragment data
Hard disk vendor
Safe disk replacement
World's highest-speed secret sharing
High-speed processing compatible with OpenStack Swift erasure code
Proxy nodes
Storage nodes

* SHSS is configured in Swift as the only available erasure code engine that achieves security.

Fig. 4. Swift secret sharing.

quickly store/retrieve files. Previously, secret sharing processing for fragmentation and reconstruction was much slower than erasure code's encoding and decoding; therefore, it was difficult to apply secret sharing to storage systems. To improve the performance, NTT developed a new high-performance 64-bit processing technique that is faster than the 8-bit process used in previous mechanisms. This makes it possible to increase the processing speed so that SHSS can fragmentize and reconstruct data at about 20 Gbit/s in the case of 24 total fragments, with 20 fragments used for reconstruction.

5. Future work

For three years, NTT has been working to develop the Swift features as described in this article. In the future, we will primarily focus on developing two new features for Swift.

One is a combination of global cluster and erasure code. As explained, they both have substantial advantages. However, erasure code cannot currently be applied for global cluster use cases because the lower data redundancy (in particular, < two times data redundancy) leads to the possibility of data loss if a region goes completely offline. In addition, we noticed from parity calculation constraints that a large number of parity fragments (i.e., ec_m) for increasing data redundancy reduced the PUT/GET performance in our experiment. To achieve the benefits of both global cluster and erasure code, we are now attempting to develop a new scheme called global EC (erasure code) cluster, with the Swift community.

The other concept we are trying to develop is storage tiering. Automated tiering has recently become a popular feature in storage system products. It makes it possible to connect two or more storage tiers together that are basically different in performance and cost. With tiering, we can use actual hardware more efficiently according to the user's own data access pattern. Swift currently supports the static deployment of certain kinds of storage definitions called storage policies, but we are now researching a way to dynamically place each object among the storage policies in the Swift cluster to achieve greater efficiency.

6. Conclusion

Working with Swift software and the Swift community has improved our software development and enabled us to focus on which areas we should develop as part of the OSS community. The Swift production development cycles and community activities provide many opportunities for developers to contribute, and the Swift community has recently finished developing notable features such as global cluster and erasure code. We believe that working together with the OSS community is a great way to improve our products. To further speed up our research and development, we will continue to work closely with the OSS community.

References

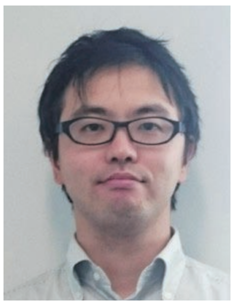
[1] Website of OpenStack Swift, Documentation.
http://docs.openstack.org/developer/swift/

[2] Website of Rackspace, Knowledge Center, Cloud Files: How to articles & other resources.
http://www.rackspace.com/knowledge_center/article/cloud-files-how-to-articles-other-resources

[3] HP, "Maintaining and Operating Swift at Public Cloud Scale," OpenStack Summit, Vancouver, Canada, May 2015.
https://www.openstack.org/summit/vancouver-2015/summit-videos/presentation/maintaining-and-operating-swift-at-public-cloud-scale

[4] NTT DATA press release issued on January 15, 2015 (in Japanese).
http://www.nttdata.com/jp/ja/news/release/2015/011500.html

[5] NTT press release issued on May 18, 2015.
http://www.ntt.co.jp/news2015/1505e/150518a.html



Kota Tsuyuzaki
Software Engineer, NTT Software Innovation Center.
He received an M.E. in information engineering from Waseda University, Tokyo, in 2010. Since joining NTT in 2010, he has been engaged in developing distributed storage systems. He has been working on OpenStack Swift for approximately three years and has been a member of the Swift Core Team since June 2015.

Masahiro Shiraishi
Senior Research Engineer, Supervisor, NTT Software Innovation Center.
He received an M.E. in mathematics from Kagoshima University in 1991. He joined NTT in 1991 and studied and developed operating system platforms. He is currently studying distributed storage systems.