Swift-X: Accelerating OpenStack Swift with RDMA for Building an Efficient HPC Cloud

Shashank Gugnani, Xiaoyi Lu, and Dhabaleswar K. (DK) Panda
Department of Computer Science and Engineering, The Ohio State University
Email: {gugnani.2, lu.932, panda.2}@osu.edu

***Abstract*—Running Big Data applications in the cloud has become extremely popular in recent times. To enable the storage of data for these applications, cloud-based distributed storage solutions are a must. OpenStack Swift is an object storage service which is widely used for such purposes. Swift is one of the main components of the OpenStack software package. Although Swift has become extremely popular in recent times, its proxy server based design limits the overall throughput and scalability of the cluster. Swift is based on the traditional TCP/IP sockets based communication which has known performance issues such as context-switch and buffer copies for each message transfer. Modern high-performance interconnects such as InfiniBand and RoCE offer advanced features such as RDMA and provide high bandwidth and low latency communication. In this paper, we propose two new designs to improve the performance and scalability of Swift. We propose changes to the Swift architecture and operation design. We propose high-performance implementations of network communication and I/O modules based on RDMA to provide the fastest possible object transfer. In addition, we use efficient hashing algorithms to accelerate object verification in Swift. Experimental evaluations with microbenchmarks, Swift stack benchmark (ssbench), and synthetic application workloads reveal up to 2x and 7.3x performance improvement with our two proposed designs for put and get operations. To the best of our knowledge, this is the first work towards accelerating OpenStack Swift with RDMA over high-performance interconnects in the literature.**

***Keywords*-OpenStack, Swift, RDMA, High-performance interconnects**

I. Introduction

Cloud computing has become a novel computing paradigm that has changed the way enterprise or Internet computing is seen. The public cloud market is expected to grow by more than 17% by the end of 2016 to a total of over $208 billion, up from $178 billion in 2015, according to new projections from Gartner [1]. The success story of cloud computing as a technology is credited to the long term efforts of the computing research community and industry companies across the globe. SaaS (Software as a Service), PaaS (Platform as a Service), and IaaS (Infrastructure as a Service) are the three major cloud product sectors. IaaS supports easy and scalable resource management and a better overall utilization of clusters when compared to dedicated clusters. IaaS also provides high-performance sharing of critical cluster resources among multiple jobs using the system.

*This research is supported in part by National Science Foundation grants #CNS-1419123, #IIS-1447804, #ACI-1450440, #CNS-1513120, and #IIS-1636846.

The cloud computing paradigm motivates more and more users to move their applications to the cloud or build private clouds inside their own organizations. OpenStack [2] is one of the most popular open-source solutions to build clouds and manage cloud computing, storage, and networking resources. OpenStack can be used to build efficient HPC clouds to support running various applications. In order to run data-intensive applications in the cloud efficiently, it is a must to enable cloud-based distributed storage solutions for these applications. OpenStack Swift is an object storage service which is widely used for such purposes. Swift is one of the main components of the OpenStack software package.

According to the latest OpenStack user survey [3], more than 53% of all OpenStack deployments use Swift. In addition, large Swift deployments are becoming more common with 24% of deployments having a total storage capacity of more than 100 TB and 32% of deployments having more than 10,000 objects. It also reports that the primary use cases for Swift include backup/archiving and storing Docker/VM images, application data, and Big Data.

Although OpenStack Swift has become extremely popular in recent times, Swift is still using traditional TCP/IP sockets based communication which has known performance issues such as context-switch and buffer copies for each message transfer [4], [5]. Modern high-performance interconnects such as InfiniBand [6] and RoCE [7] offer advanced features such as Remote Direct Memory Access (RDMA) and provide high bandwidth and low latency communication. InfiniBand has been widely used in modern HPC clusters. Based on the November 2016 TOP500 [8] ranking, 37% clusters in the top 500 supercomputers are using InfiniBand technology. Due to its high-performance and advanced features (e.g. RDMA), many recent studies [9], [10], [4], [11], [12] have re-designed popular Big Data stacks such as Hadoop, Spark, and Memcached, with native RDMA operations to achieve the huge benefits compared to the default sockets based designs with IP-over-IB protocol.

This trend motivates us to explore possible options to make the OpenStack Swift design efficiently take advantage of high-performance interconnects such as InfiniBand and its associated advanced features such as RDMA. In the process of accelerating OpenStack Swift with RDMA, we also try to exploit more opportunities to enhance the Swift architecture for building efficient HPC clouds. All these issues lead us to the following broad challenges:

1) What are the performance characteristics and bottlenecks of the current Swift design?
2) How can high-performance and scalable RDMA-based communication schemes be designed to reduce the network communication time for Swift operations and further improve overall Swift performance?
3) In addition to communication, what else can be done to further accelerate Swift performance and scalability? For example, how can the I/O, computation, and architecture designs in Swift be enhanced?

To address these challenges, in this paper, we first start with understanding the performance characteristics of the current OpenStack Swift design. Through the breakdown analysis of Swift operations, we identify three major bottlenecks inside current Swift design, namely communication, I/O, and hashsum computation. From the architecture perspective, OpenStack Swift operations are heavily using the proxy server based design, which significantly limits the overall throughput and scalability of the Swift cluster.

Based on these observations, we propose a high-performance design and implementation of OpenStack Swift, called Swift-X, for building efficient HPC clouds. Swift-X has two new designs to improve the performance and scalability of Swift applications in its two typical usage scenarios. One design is client-oblivious where users can benefit from our proposed designs without the need for any modification in the client library or any need of RDMA-capable networking devices on the client node. The second design is a metadata server-based design, which completely overhauls the existing design of put and get operations in Swift. Instead of using the proxy server for routing requests, we propose to reuse it as a metadata server instead.

In both these two designs, we propose high-performance implementations of network communication and I/O modules based on RDMA to provide the fastest possible object transfer. We also explore different hashing algorithms in the community to further improve the object verification performance in Swift.

Experimental evaluations with microbenchmarks, Swift stack benchmark (ssbench) [13], and synthetic application workloads reveal up to 2x and 7.3x performance improvement with our two proposed designs for put and get operations. The overall communication time is reduced by up to 4x, while the I/O time is reduced by up to 2.3x.

To summarize, the main contributions of this paper are as follows:

1) Identifying the performance bottlenecks inside default Swift architecture and designs
2) Re-desiging the Swift architecture to improve scalability and performance
3) Proposing RDMA-based communication framework for accelerating networking performance
4) Proposing high-performance I/O framework to provide maximum overlap between communication and I/O
5) Exploiting benefits from different hashing algorithms for improving the object verification performance
6) Introducing new operation mode in Swift to take advantage of our proposed designs

To the best of our knowledge, this is the first work towards accelerating OpenStack Swift with RDMA over high-performance interconnects in the literature.

The rest of this paper is organized as follows. Section II discusses the background for our work and Section III presents the motivation behind our work. Section IV presents our proposed designs to accelerate Swift and Section V demonstrates a performance evaluation of our proposed design. Section VI discusses related work and Section VII concludes the work.

II. Background

A. OpenStack Swift

Swift [14] is a distributed cloud-based object storage service. It is one of the main components of the OpenStack [2] software family. Usually, Swift is deployed as part of an OpenStack deployment. However, it may also be deployed as an independent storage solution. Swift is not a filesystem-based solution, but provides access to data using standard HTTP calls. This is one of the biggest advantages of Swift, because it allows data to be accessed from anywhere in the world, as long as there is an Internet connection available. Swift stores files in the form of *objects* inside *containers*. Containers in Swift are the equivalent of folders in a filesystem. For object verification while uploading and downloading, Swift computes the hashsum of each object.

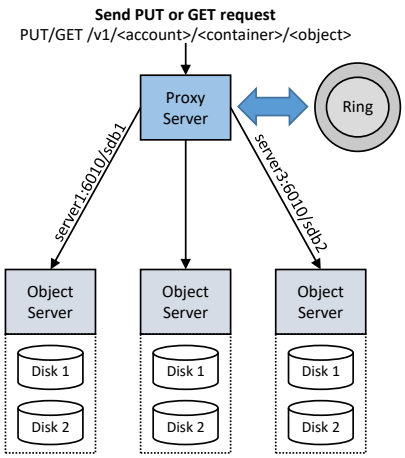


Figure 1. Swift Architecture

The Swift architecture consists of the following components:

1) Proxy Server: The proxy server ties the entire swift architecture together. It handles all requests made to Swift and routes it to the appropriate server.
2) Account Server: Server that maintains account information and handles account related requests
3) Container Server: Server that handles information and requests related to containers
4) Object Server: The object server is a blob storage server that handles upload, download, and deletion of objects.

Each object is stored as a binary file with metadata stored as the file's extended attributes. It employs various auditing procedures to recover from and avoid certain error conditions.

5) The Ring: It provides a mapping between the names of entities and their locations. There are seperate rings for containers, accounts, and objects.

Each storage node is typically deployed with one instance of account, container, and object server.

B. Python/ctypes

Python [15] is a general purpose high-level programming language. Python is an object-oriented and dynamically interpreted language. Owing to its minimal syntax design and programmability, it allows programmers to express concepts in relatively fewer lines of code compared to other common languages. This has made it really popular as a programming language in recent times. However, because it is high-level and dynamic in nature, it suffers in performance.

Ctypes [16] is a library in the Python programming language that provides with C compatible datatypes along with the ability to call shared libraries and DLLs directly from Python code. For any C function call, ctypes automatically maps the Python datatypes to C datatypes. Using this library, C-based DLLs and shared libraries can be wrapped in pure Python.

C. InfiniBand

InfiniBand [6] is a computer-networking communication standard used in high-performance computing to achieve high throughput and low latency. This high speed, general purpose I/O interconnect is widely used in supercomputers worldwide. According to the latest TOP500 [8] rankings released in November 2016, more than 37% of the top 500 supercomputers use InfiniBand as their networking interconnect. One of the key features of InfiniBand is Remote Direct Memory Access (RDMA). RDMA can be used by a process to remotely read or update memory contents of another remote process without any involvement at the remote side. InfiniBand offers data transfer in a complete OS bypassed manner, i.e the communication is processed in userspace and carried out in a zero-copy manner. InfiniBand uses hardware offload for all protocol processing, resulting in high-performance communication. InfiniBand also features Internet Protocol over InfiniBand (IPoIB) protocol that can be used to run traditional socket-based applications over InfiniBand hardware.

III. Motivation

Swift is typically used by users for uploading/downloading software, simulation input files, experimental results, large datasets, VM images, and configuration files. Based on where the cluster is accessed from, its usage can be classified into two scenarios, as shown in Figure 2. The two scenarios are highlighted in red (dashed) and green (solid). The first scenario consists of a user accessing the cluster from outside of the local cluster network via the Internet. The second usage scenario consists of the user accessing the cluster from within the local network. This is usually from a bare-metal OpenStack compute node or a virtual machine running on one of the compute nodes. The second scenario is more likely to happen since most users use Swift for storing and retreiving files for running experiments on the compute nodes. From the figure it is clear that the proxy server is a bottleneck for all requests and limits the throughput of the cluster.

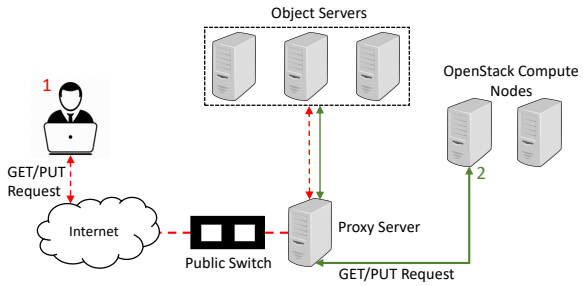


Figure 2. Swift Usage Scenarios

Apart from uploading and downloading objects, other Swift requests involve manipulating containers, accounts, and objects. All of these operations do not incur any significant network communication or I/O since they do not involve any object transfer. Our evaluations reveal sub-second latency for such operations. However, uploading and downloading objects incurs significant network and I/O overhead, especially for large objects. The Swift code is written in Python with network communication implemented using TCP sockets-based communication. As we already know, Python performance is lower than other common languages. In addition, TCP communication has serveral known performance bottlenecks, such as context-switch and extra buffer copies for each message communication. Thus, it is important to analyze the performance of upload and download operations with the default Swift implementation and come up with ideas to improve the performance of these operations.

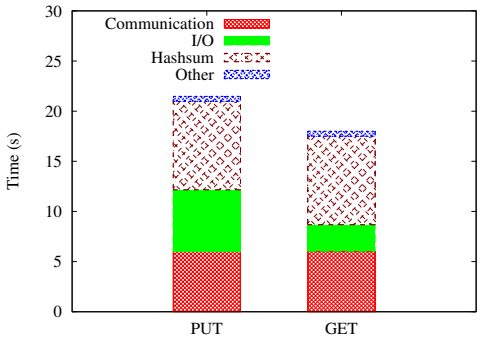


Figure 3. Breaking down GET and PUT latency into different components

Figure 3 shows the breakup of get and put operations into different components for a 5 GB object. The surprising result here is that computing the hashsum of the object takes a good chunk of time. This is a result of the slow performance of the md5 hashing algorithm which is used by default in Swift.

For put, 41.1%, 28.1%, and 28.5% of the total time is spent in hashsum computation, network communication, and I/O, while for get this breakup is 49%, 33.5%, and 14.6%. It is evident that computing hashsum, network communication, and I/O take up a big chunk of the total operation count. Thus, it is only natural to ask whether the performance of these three main components can be enhanced in some manner to improve the overall performance of the cluster.

The benefits of high-performance networking interconnects such as InfiniBand have been extracted by the HPC community for a long time. They provide advanced features such as RDMA-based communication, which provides for low latency and high-bandwidth communication. As we have already seen, network communication and I/O constitute a big portion of get and put operations. This provides an opportunity to use RDMA-based semantics to accelerate get and put Swift operations. The challenge here is to design a scalable communication framework based on RDMA which can not only speed up the network communication but also provide overlap with I/O. Our primary motivation in this paper is to reduce the communication, I/O, and hashsum components of get and put operations while improving the scalability and throughput of the Swift cluster, and maintaining the same level of fault-tolerance.

IV. Proposed Designs

In this paper, we propose a high-performance implementation of the OpenStack Swift Object Storage, called Swift-X. We propose designs to accelerate the network, I/O, and object verification (hashsum) components of get and put operations. We present our proposed designs in this section.

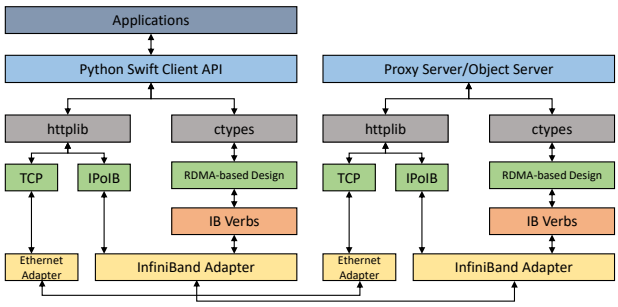


Figure 4. Technology Overview

A. Swift-X Overview

Figure 4 presents an overview of our proposed design. We propose extensions and modifications to the command-line Swift client library, object server, and proxy server. Figure 5 shows the architecture overview of our design. We introduce an RDMA-based communication module in the client, object server and proxy server for low latency communication. We also introduce a dedicated I/O module in the client and object server for object file related operations. The communication and I/O modules have been designed to work in an integrated manner and are written in C for performance. It is important to mention here that we do not change the default Swift client API. Thus, existing applications can transparently run over Swift-X without any code modification. Since the Swift code is written in Python, we cannot directly call our C modules from Python. To solve this issue, we use the *ctypes* Python library which allows shared C libraries to be loaded and called from Python code. Thus, we compiled our C modules as shared libraries and used ctypes to integrate the proposed modules with Swift code. The default implementation uses the *httplib* Python library [17] for HTTP-based communication. This communication can either go through the Ethernet adapter using standard TCP or through the InfiniBand adapter using the IPoIB protocol. Our implementation uses the *ctypes* Python library to make calls to our communication module which is built on top of the InfiniBand verbs interface allowing for native communication over InfiniBand adapters.

B. Client-Oblivious Design

As explained in Section III, Swift typically has two usage scenarios - one where the cluster is accessed from within the local network, and the other where it is accessed from an external network. Access from external networks is typically through the user's personal computer which unlike datacenter and cluster servers typically do not have RDMA-capable network devices. For this usage scenario, we propose a client-oblivious design where users can still benefit from our proposed designs without the need for any modification in the client library or need for RDMA-capable network devices on the client node. Figure 6(a) shows how this design works. The overall communication semantics of get and put operations are preserved with no changes to the communication between the client and proxy server. However, communication between the proxy and object servers is via RDMA using our proposed design changes to the proxy and object servers. This design works as follows. The client sends requests and data to the proxy server over TCP using the default client implementation. The proxy server sends requests and data to object servers in parallel using RDMA communication. It then waits for responses from the object servers, before returning the final response to the client. For this design, we inherit the replication semantics and design from the default implementation and provide the same fault-tolerance level.

C. Metadata Server-based Design

The default Swift design routes all requests and data through the proxy server adding additional latency to each operation. Swift provides fault-tolerance by replicating objects to multiple servers. The proxy server is responsible for handling replication in the default implementation. While multiple proxy servers can be deployed, the number of proxy servers usually ranges from 1-4. Thus, the proxy server becomes a bottleneck for multiple get and put operations. For the second usage scenario, where the Swift cluster is accessed from within the local network, the object servers are directly accessible from the client nodes. Thus, there is no need to route all requests through the proxy server. For this scenario, we propose a

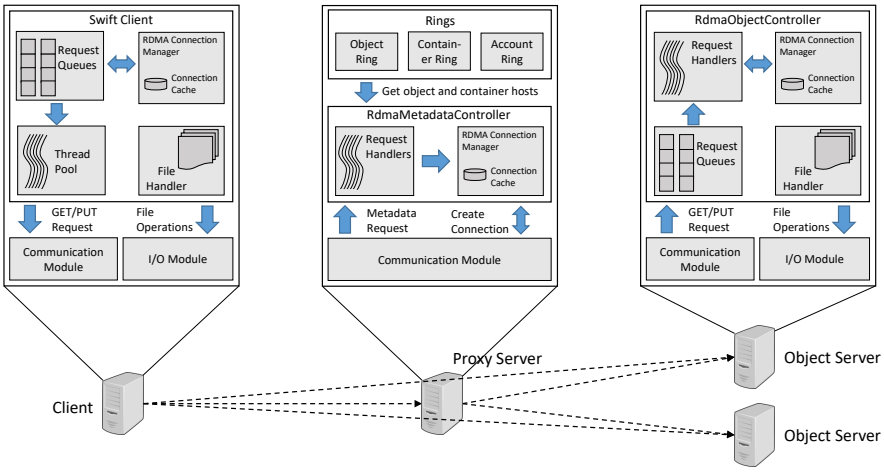
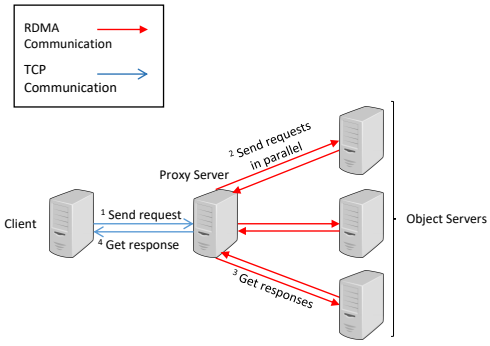
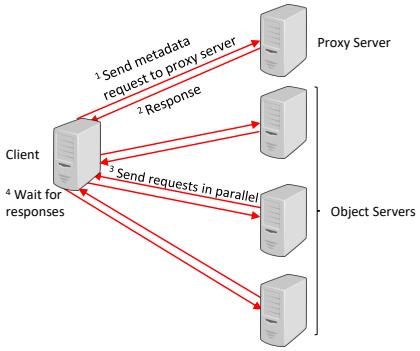


Figure 5. Swift-X Architecture Overview



(a) Client-Oblivious Design



(b) Metadata Server-based Design

Figure 6. Proposed Designs for Swift-X Operations

metadata server-based design which completely overhauls the existing design of put and get operations in Swift. Instead of using the proxy server for routing requests, we propose to reuse it as a metadata server instead. The clients will use the proxy server to obtain object metadata for get and put operations. This design will work as follows. The client will send a get or put request to the proxy server using the RDMA communication module. The proxy server will then get the required metadata for the object including the locations of object servers where the object needs to be sent to or gotten from. The client then parallely sends the request and data to all object servers. Thus, replication is done in a parallel manner using RDMA. We make sure that the semantics of replication is exactly the same as the default design and there is no change in the fault-tolerance of the cluster. For get operations, we get the object from the first object server which indicates that it has an uncorrupted copy of the object. By eliminating the need to route data through the proxy server and handling replication in the client itself, this design offers siginificant scalability over the original design. Figure 6(b) shows how this design works.

D. Object Verification

For object verification while uploading and downloading objects, Swift computes the md5 hashsum of each object. While md5 is a popular hashing algorithm and provides high quality hashing, it suffers from poor performance. As our evaluations in Section III show, 41% of the total time for put and 49% of the total time for get is spent in computing the md5 hashsum of an object. Thus, this calls for a re-evaulation of the decision to use md5 for object verification.

Name	Speed	Quality
xxHash [18]	5.4 GB/s	10
MurmurHash 3a [19]	2.7 GB/s	10
SBox [20]	1.4 GB/s	9
Lookup3 [21]	1.2 GB/s	9
CityHash64 [22]	1.05 GB/s	10
FNV [23]	0.55 GB/s	5
CRC32	0.43 GB/s	9
MD5-32	0.33 GB/s	10
SHA1-32	0.28 GB/s	10

Table I
HASHING ALGORITHMS

After a thorough survey of state-of-the-art hashing algorithms, we found one which delivers the best performance while providing high hashing quality - xxHash. The SMHasher test [24] is a good benchmark to measure the quality of hashing algorithms. Xxhash scores a perfect 10 on this benchmark

while delivering the best performance as shown in Table I. Thus, in our designs, we modify the existing verification schemes to use xxHash instead of md5.

E. Design Implementation

In this subsection, we present the different components of our proposed design.

1) Proxy Server: As shown in Figure 5, we introduce dedicated communication and I/O modules in the proxy server. We add a RDMAMetadataController class which provides with request handlers and an RDMA connection manager for handling RDMA object requests. The RDMA connection manager builds connections on demand and caches connections for performance. For metadata requests, the proxy server uses the object and container rings to get host information about the request object and returns this information to the client. For the client-oblivious design, we made changes to the ObjectController class in the proxy server to use our RDMA communication module for communication with the object servers.

2) Object Server: Each object server has a RDMAObjectController class which provides with request handlers and a connection manager for handling object get and put requests. This connection manager is similar to the connection manager proposed in the proxy server. Each request handler has a dedicated request queue. All requests received by the object server are placed in the request queue of a handler selected in a round-robin manner. The request handlers poll the queues for requests and then process them. File operations are processed using the dedicated file handler which uses the I/O module underneath.

3) Client: Our client implementation is based on the Python command-line Swift client. This version of the Swift client is the most popular among users. Our client implementation uses the communication module also used by the object and proxy servers. For sending requests to the proxy and object servers, we use a dedicated fixed size thread pool. This allows us to reuse the spawned threads and send requests to servers in parallel.

4) Object Transfer: Our proposed I/O and communication modules work in an integrated manner to transfer objects to the object server. The semantics of the object transfer are as follows. The object contents are read and transfered in a chunked manner. We read the object contents chunk by chunk directly into the pre-allocated RDMA communication buffers. This prevents the need for any extra buffer copies for each network transfer. We then send each chunk using RDMA and wait for acknowledgement from the receiver. While waiting for the acknowledgement, we read the next chunk of data from the object to overlap communication with I/O as much as possible. On the receiver side, upon receipt of a object data chunk, we directly write the data from the RDMA communication buffer to the object file, again ensuring maximum overlap between communication and I/O. By using acknowledgements and object hashsum, our design ensures lossless transfer of object data while delivering high-performance.

F. Usage Modes

Based on our two proposed designs, we propose two usage modes for Swift-X, as shown in Figure 7. The first mode (highlighted in red dashed) is for use from external networks over the Internet. This mode is the same as default Swift design, except that the communication between the proxy server and object servers is using our modified RDMA-based designs and modules. The second mode (highlighted in green solid) is for use from within the OpenStack cluster network. In this mode, our metadata server-based design is utilized to allow direct communication between the client and object servers. This mode uses our modified client designs while the first mode uses the default client design. Swift-X can operate in both modes simultaneously. Based on the client implementation, the appropriate mode is automatically selected and then used.

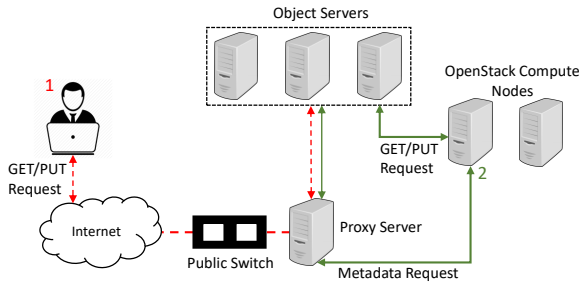


Figure 7. Swift-X Usage Scenarios

V. Performance Evaluation

A. Experimental Testbed

Our testbed consists of 18 physical nodes on the Chameleon Cloud [25], which is an OpenStack deployment. Each compute node has a 24-core 2.3 GHz Intel Xeon E5-2670 (Haswell) processor with 128 GB main memory and is equipped with Mellanox ConnectX-3 FDR (56 Gbps) HCAs and PCI Gen3 interfaces. We use CentOS Linux 7.1.1503 (Core) with kernel 3.10.0-229.el7.x86_64. In addition, we use the Mellanox OpenFabrics Enterprise Distribution (OFED) [26] MLNX_OFED_LINUX-3.0-1.0.1, Python 2.7.5, Swift 2.8.0, and Python Swiftclient 3.0.0.

We deployed a Swift cluster with 16 object servers and 1 proxy server. Each object server also runs an account and container server. We use the standard replication policy for each object with a replication factor of 3.

Table II provides a description of the terms we have used in all graphs as well as the remaining text. We use M1 to signify the operation mode in our implementation where the client-oblivious design is triggered, while M2 is used for the mode where the metadata server-based design is triggered.

B. Microbenchmarks

We first evaluated our designs with basic put and get microbenchmarks. We used our implementation of the Swift client and measured the overall latency of each operation. The

Term	Description
Swift	Swift v2.8.0
Swift-X	Proposed design implementation on Swift v2.8.0
M1	Client-oblivious design
M2	Metadata server-based design

Table II
GRAPH LEGEND

	Number of Objects				
Scenario	**Small (10 MB)**	**Medium (100 MB)**	**Large (1 GB)**	**Huge (5 GB)**	**CRUD Profile**
Scenario1	114	57	29	0	[4 4 2 0]
Scenario2	0	23	13	4	[4 5 1 0]

Table III
SSBENCH SCENARIOS

object size is varied from 1 MB to 5 GB and each object is uploaded using a seperate put or get operation. Each object is a binary file that contains randomly generated data. We go up to 5 GB because that is the maximum size of an object that Swift can support. Uploading objects larger than 5 GB is handled by splitting the object into multiple chunks of size 5 GB or less and uploading them as seperate objects. This is automatically done by the client code and is supported in our implementation as well. Downloading of large objects is handled similarly by the client. Figures 8(a) and 8(b) show the latency of get and put operations. Overall, we see up to 40% and 47% improvement with M1 for put and get operations, respectively. With M2 this improvement is 55% and 66%, respectively. It can also be observed that the latency for objects of sizes 64 MB or less is similar for all three implementations. This is because operations on small object sizes do not involve significant network or I/O. We also did a time breakup comparison to understand where the performance benefits are coming from. This analysis is shown in Figure 8(c). It can be observed that the hashsum time is reduced by a huge margin (15x). This is due to the fast performance of the xxHash algorithm. In addition, both communication and I/O times are reduced for both designs in Swift-X. These improvements can be attributed to RDMA-based communication, efficient I/O implementation, and overlap between I/O and communication. The performance of design M2 is much better than that of design M1. This is expected as the TCP communication in M1 between the client and proxy server limits the overall performance of the operation. This design also suffers from the need to route all requests and data through the proxy server. With these limitations solved in M2, we observe much more improvement as compared to M1, demonstrating scalability improvement in the cluster. Overall, the communication time is reduced by 36% and 3.8x for put and 36% and 2.8x for get with M1 and M2, respectively. While the overall I/O time is reduced by 16.3% and 2.3x for put with M1 and M2, respectively. We do not see any significant improvement in I/O for get operations. This is because the I/O read code-path is quite similar for all cases, while the default write code-path involves additional memcopies which our design avoids by directly writing to the object file from the communication buffers.

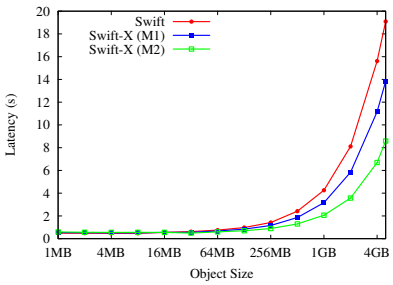
C. Evaluation with ssbench

Swift Stack Benchmark or ssbench [13] is a flexible and scalable benchmark for evaluating Swift performance. Ssbench allows for testing the Swift cluster under different *scenarios*. Each *scenario* is defined by a configuration file which includes information such as the number and size of objects to test with, the ratio of create, read, update, and delete operations (also known as CRUD profile), total operation count, etc. Create and update involve a put operation, the only difference being that for update, the object already exists in the cluster. Read involves a simple get operation. Ssbench also supports distributed multi-client evaluation. Its architecture consists of one master process and several worker processes. The worker processes actually execute the operations, while the master process co-ordinates all worker processes. Ssbench does not directly use the Python Swift client API, but uses a modified version of it. Thus, for evaluating Swift-X with ssbench, we brought our client side changes to ssbench as well. Our modifications are based on ssbench 0.3.9, and we use our modified ssbench implementation for all experiments. After looking at Swift usage reports, we came up with two scenarios to evaluate our cluster (Table III). Since we did not modify the delete operation code-path, we do not evaluate the performance of delete operations. For all evaluations we test with 8 client workers, each with a concurrency of 1.

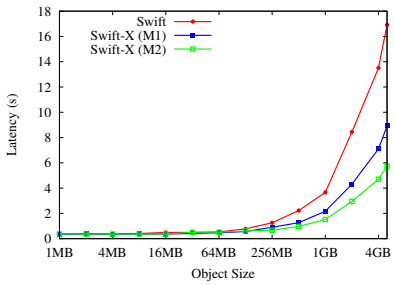
Figure 9 shows the number of requests per second for the complete benchmark run for both scenarios. Figure 10 shows latency figures for Scenario 1 while Figure 11 shows figures for Scenario 2. We observe 77.4% and 2.8x improvement in the total operations per second for Scenario 1 and 2x and 3.5x improvement for Scenario 2 with M1 and M2, repectively. For overall latency figures, we see 2.1x and 2.68x improvement for create over Swift, 2x and 6.25x improvement for read, and 27% and 23% for update with M1 and M2, respectively. For Scenario 2, the improvement is 2x and 2.6x for create, 21% and 7.3x for read, and 42.4% and 2.72 for update. Overall M2 performs much better than M1, however for small object sizes we observe that M1 performs slightly better than M2. This is because for small object sizes, the connection initialization and metadata request overhead cannot be compensated by improvements in the small amount of network and I/O involved. It can also be observed that there is much more improvement with ssbench for M2 than what we saw with our microbenchmarks. This can be attributed to our metadata server-based design because of which the 8 clients running in parallel can process more requests per second since the proxy server is no longer the bottleneck.

D. Synthetic Application Benchmark

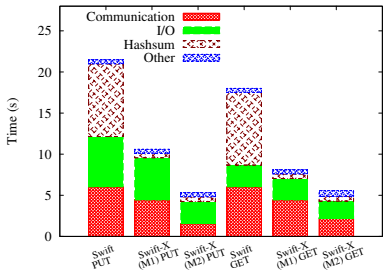
Swift is often used for storing input files for Big Data applications to be run in a cloud environment. According to the official OpenStack user survey [3], 58% of Swift deployments



(a) PUT Latency Evaluation

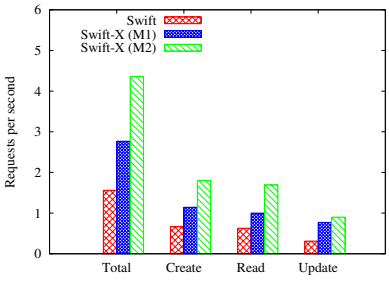


(b) GET Latency Evaluation

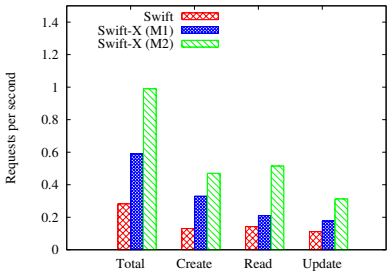


(c) Time Breakup of GET and PUT operations for a 5 GB object

Figure 8. GET and PUT Microbenchmark evaluation

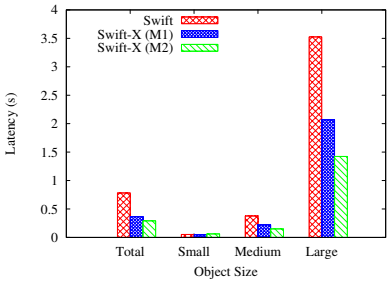


(a) Scenario 1

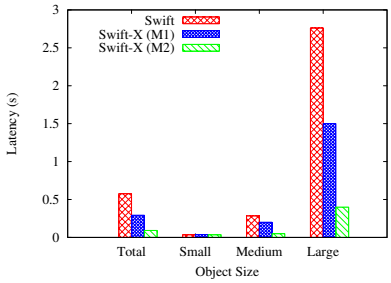


(b) Scenario 2

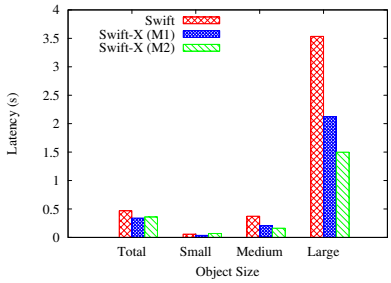
Figure 9. Evaluation of total requests per second with ssbench



(a) Create Latency

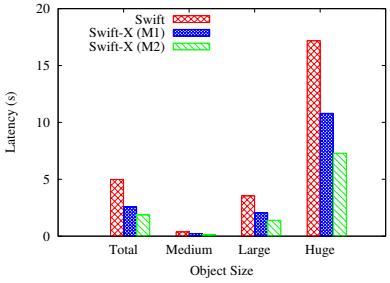


(b) Read Latency

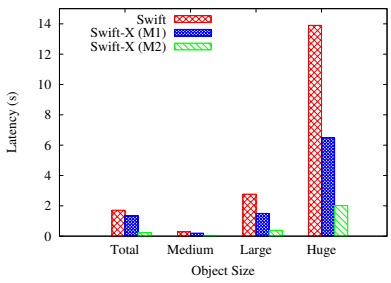


(c) Update Latency

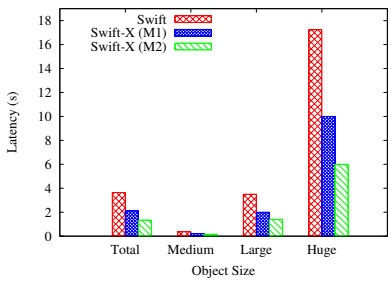
Figure 10. Latency of different operations for ssbench scenario 1



(a) Create Latency

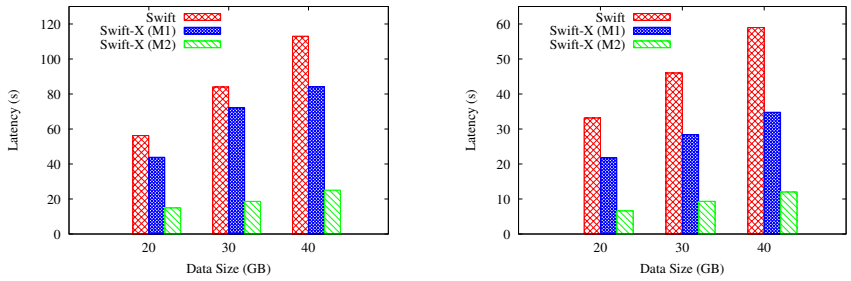


(b) Read Latency



(c) Update Latency

Figure 11. Latency of different operations for ssbench scenario 2



(a) Upload (b) Download

Figure 12. Application evaluation with e-book dataset

store user application data while 32% of deployments store Big Data. Thus, it is important to analyze the performance of storing and retreiving this data from the cluster. Usually, input files are uploaded to Swift from the user's personal computer or from a server on a cluster. These files are then downloaded from the virtual cloud cluster and the experiment is run there. Apache Hadoop [27] is a popular Big Data stack based on the MapReduce [28] framework. To evaluate our designs with this use case, we designed a synthetic application benchmark which uses an e-book dataset [29], which can be used as input for the Hadoop WordCount application. The benchmark uploads and downloads the input dataset from one of the compute nodes in the OpenStack cluster. We create three datasets of sizes 20, 30, and 40 GB, each consisting of multiple large e-books in text format. We evaluated both uploading and downloading the input dataset with our proposed designs. Results for this evaluation are presented in Figure 12. We see up to 27% and 4.5x improvement for uploading and 41% and 5x improvement for downloading with M1 and M2, respectively. This demonstrates the feasibility and application of our design in real-life scenarios.

VI. Related Work

There have been several publications which propose modifications to Swift or present use cases and case studies of Swift usage. In this section, we discuss the ones most related to our work.

Yokoyama et al. [30] propose an intercloud object storage service called Colony. Their design allows object storage services on different clouds to access each others data using the same client API. They implement their design on top of Swift. They stress on inter-organization research collaboration as the motivation behind their work. Authors in [31] present an approach to allow for content level access control in Swift. While the default Swift implementation uses an *all or nothing* approach, their implementation allows specifying which user can access which part of an object. In [32], authors present a middleware package built on top of Swift, called ZeroVM. Their middleware allows users to run containerized applications directly on the object servers. Their main goal is to bring computation to data rather than the other way around. Another work [33] proposes a client-based deduplication scheme for securely storing data in Swift. Their system generates a key for each data object and ensures that only the user with the correct private key can decrypt object data. In [34], authors propose a network-aware inter-cloud object storage service based on Swift. Their approach uses topology-aware operations and asynchronous-replication to improve network communication time. However, their design suffers from the TCP communication bottleneck and reduced fault-tolerance.

Poat et al. [35] provide a performance comparison of Swift and Ceph with real-life scenarios. Their results indicate that Swift performs better for single file writes, but falls short of Ceph for I/O concurrency and multi-client tests. Another case study [36] by CERN [37] presents results for using Swift for handling data from CERN experiments. Their results indicate that Swift could fulfill requirements by the CERN scientific community. Authors in [38] propose a smart cloud seeding system for BitTorrent [39] which uses Swift for data storage and reliability. They modify Swift to support the BitTorrent protocol. The Swift proxy server handles all incoming requests and upon detecting a certain mass for specific content, switches to the BitTorrent protocol. Community clouds are usually more distributed, diverse and less reliable than data center clouds. In [40], the authors evaluate the performance and sensitivity of Swift in a typical community cloud setup. Through their evaluation results, they establish a relationship between the performance of Swift and the various environmental factors in a community cloud.

There has also been a lot of work on using RDMA to accelerate Big Data stacks. Authors in [11] use RDMA to improve the performance of Spark, while in [9], the authors present an RDMA-enhanced HDFS design. Shankar et al. [12] propose to accelerate Memcached using RDMA. They also present non-blocking extensions and designs with SSD for Memcached.

Although there has been a lot of research on modifying Swift to introduce new functionality, most of the works do not focus on performance. There has also been a lot of research on using RDMA to improve the performance of Big Data middleware, however no such work has been done for Swift. Moreover, most papers focus on improving the performance of Big Data stacks, while such a direction for cloud computing middleware is relatively unexplored. This makes our work unique and our contributions significant.

VII. Conclusion and Future Work

In this paper, we proposed a high-performance design and implementation of OpenStack Swift, called Swift-X, for building efficient HPC clouds. We first analyzed the Swift architecture and its common usage scenarios and identified major bottlenecks. We also conducted a comprehensive performance evaluation of get and put operations and identified the components contributing the most to the overall latency of the operation. We identified hashsum computation, communication, and I/O as the main factors affecting performance. Based on our analysis, we proposed two designs, namely the client-oblivious design and the metadata server-based design, for accelerating Swift performance for the two common use cases. We also proposed designs to accelerate network communication, I/O, and object verification components of put and get operations. We introduced new operation modes in Swift to take advantage of our proposed designs. We presented a comprehensive evaluation of our proposed design with microbenchmarks, ssbench, and synthetic application benchmarks. Our evaluation reveals that our designs can deliver up to 2x performance improvement for the client-oblivious design and up to 7.3x improvement for the metadata server-based design.

In the future, we plan to modify the S3 and HDFS Swift clients to work with our designs. We also plan to evaluate with additional benchmarks and application scenarios. Evaluation with multiple proxy servers, SSDs, and other deployment scenarios is also left as future work.

References

[1] "Gartner Says Worldwide Public Cloud Services Market to Grow 17 Percent in 2016." [Online]. Available: http://www.gartner.com/newsroom/id/3443517

[2] "OpenStack," http://openstack.org/.

[3] "OpenStack User Survey," https://www.openstack.org/assets/survey/April-2016-User-Survey-Report.pdf.

[4] X. Lu, N. S. Islam, M. W. Rahman, J. Jose, H. Subramoni, H. Wang, and D. K. Panda, "High-Performance Design of Hadoop RPC with RDMA over InfiniBand," in *42nd International Conference on Parallel Processing (ICPP)*. IEEE, 2013, pp. 641–650.

[5] X. Lu, D. Shankar, S. Gugnani, H. Subramoni, and D. K. Panda, "Impact of HPC Cloud Networking Technologies on Accelerating Hadoop RPC and HBase," in *8th IEEE International Conference on Cloud Computing Technology and Science*. IEEE, 2016.

[6] "InfiniBand Trade Association," http://www.infinibandta.com.

[7] I. T. Association *et al.*, "Supplement to Infiniband Architecture Specification Volume 1, Release 1.2. 1: Annex A16: RDMA over Converged Ethernet (RoCE)," 2010.

[8] "TOP500 Supercomputing Sites," http://www.top500.org/.

[9] N. S. Islam, X. Lu, M. W. Rahman, and D. K. Panda, "SOR-HDFS: a SEDA-based approach to maximize overlapping in RDMA-enhanced HDFS," in *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2014, pp. 261–264.

[10] M. W. Rahman, X. Lu, N. S. Islam, and D. K. Panda, "HOMR: A Hybrid Approach to Exploit Maximum Overlapping in MapReduce over High-performance Interconnects," in *Proceedings of the 28th ACM International Conference on Supercomputing*. ACM, 2014, pp. 33–42.

[11] X. Lu, D. Shankar, S. Gugnani, and D. K. Panda, "High-Performance Design of Apache Spark with RDMA and Its Benefits on Various Workloads," in *IEEE International Conference on Big Data*. IEEE, 2016.

[12] D. Shankar, X. Lu, N. Islam, M. W. Rahman, and D. K. Panda, "High-Performance Hybrid Key-Value Store on Modern Clusters with RDMA Interconnects and SSDs: Non-blocking Extensions, Designs, and Benefits," in *IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2016, pp. 393–402.

[13] "ssbench," https://github.com/swiftstack/ssbench.

[14] "OpenStack Swift," swift.openstack.org/.

[15] "Python," https://www.python.org/.

[16] "Python ctypes," https://docs.python.org/2/library/ctypes.html.

[17] "Python httplib," https://docs.python.org/2/library/httplib.html.

[18] "xxHash," https://github.com/Cyan4973/xxHash.

[19] A. Appleby, "MurmurHash," https://sites.google.com/site/murmurhash/.

[20] A. Webster and S. E. Tavares, "On the Design of S-boxes," in *Conference on the Theory and Application of Cryptographic Techniques*. Springer, 1985, pp. 523–534.

[21] "Lookup3," http://www.burtleburtle.net/bob/c/lookup3.c.

[22] G. Pike and J. Alakuijala, "The CityHash Family of Hash Functions," 2010.

[23] L. C. Noll, "Fowler/Noll/Vo (FNV) Hash," *Accessed Jan*, 2012.

[24] "SMHasher," https://github.com/aappleby/smhasher.

[25] "Chameleon," http://chameleoncloud.org/.

[26] Open Fabrics Enterprise Distribution, http://www.openfabrics.org/.

[27] "Apache Hadoop," http://www.hadoop.apache.org.

[28] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. USENIX Association, 2004, pp. 10–10.

[29] "Project Gutenberg," https://www.gutenberg.org/.

[30] S. Yokoyama, N. Yoshioka, and M. Ichimura, "Intercloud Object Storage Service: Colony," *Cloud Computing*, pp. 95–98, 2012.

[31] P. Biswas, F. Patwa, and R. Sandhu, "Content Level Access Control for Openstack Swift Storage," in *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*. ACM, 2015, pp. 123–126.

[32] P. Rad, V. Lindberg, J. Prevost, W. Zhang, and M. Jamshidi, "ZeroVM: Secure Distributed Processing for Big Data Analytics," in *2014 World Automation Congress (WAC)*. IEEE, 2014, pp. 1–6.

[33] N. Kaaniche and M. Laurent, "A Secure Client Side Deduplication Scheme in Cloud Storage Environments," in *2014 6th International Conference on New Technologies, Mobility and Security (NTMS)*. IEEE, 2014, pp. 1–7.

[34] S. Yokoyama, N. Yoshioka, and M. Ichimura, "A Network-aware Object Storage Service," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*. IEEE, 2012, pp. 556–561.

[35] M. Poat, J. Lauret, and W. Betts, "POSIX and Object Distributed Storage Systems Performance Comparison Studies With Real-Life Scenarios in an Experimental Data Taking Context Leveraging OpenStack Swift & Ceph," in *Journal of Physics: Conference Series*, vol. 664, no. 4. IOP Publishing, 2015, p. 042031.

[36] S. Toor, R. Töebbicke, M. Z. Resines, and S. Holmgren, "Investigating an Open Source Cloud Storage Infrastructure for CERN-specific Data Analysis," in *7th IEEE International Conference on Networking, Architecture and Storage (NAS)*. IEEE, 2012, pp. 84–88.

[37] "CERN," https://home.cern/.

[38] X. León, R. Chaabouni, M. Sanchez-Artigas, and P. Garcia-Lopez, "Smart Cloud Seeding for BitTorrent in Datacenters," *IEEE Internet Computing*, vol. 18, no. 4, pp. 47–54, 2014.

[39] "BitTorrent," http://www.bittorrent.com/.

[40] Y. Liu, V. Vlassov, and L. Navarro, "Towards a Community Cloud Storage," in *2014 IEEE 28th International Conference on Advanced Information Networking and Applications*. IEEE, 2014, pp. 837–844.