



C o m m u n i t y E x p e r i e n c e D i s t i l l e d

OpenStack Object Storage (Swift) Essentials

Design, implement, and successfully manage your cloud storage
using OpenStack Swift

Amar Kapadia
Sreedhar Varma

see more please visit: <https://homeofpdf.com>

Kris Rajana

[PACKT] open source*
PUBLISHING community experience distilled

OpenStack Object Storage (Swift) Essentials

Design, implement, and successfully manage your
cloud storage using OpenStack Swift

Amar Kapadia

Kris Rajana

Sreedhar Varma



BIRMINGHAM - MUMBAI

OpenStack Object Storage (Swift) Essentials

Copyright © 2015 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: May 2014

Second edition: May 2015

Production reference: 1270515

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78528-359-8

www.packtpub.com

Credits

Authors

Amar Kapadia
Kris Rajana
Sreedhar Varma

Reviewers

Steve Martinelli
Juan J. Martínez
Christian Schwede

Commissioning Editor

Kartikey Pandey

Acquisition Editor

Harsha Bharwani

Content Development Editor

Akashdeep Kundu

Technical Editors

Prajakta Mhatre
Tanmayee Patil

Copy Editor

Vikrant Phadke

Project Coordinator

Milton Dsouza

Proofreaders

Stephen Copestake
Safis Editing

Indexer

Rekha Nair

Graphics

Jason Monteiro

Production Coordinator

Manu Joseph

Cover Work

Manu Joseph

About the Authors

Amar Kapadia is a storage technologist and blogger based in the San Francisco Bay Area. He is currently the senior director of product marketing for Mirantis, the #1 pure-play OpenStack company. Prior to Mirantis, he was the senior director of strategy for EVault's Long-Term Storage Service, a public cloud storage offering based on OpenStack Swift. He has over 20 years of experience in storage, server, and I/O technologies at Emulex, Philips, and HP. Amar's current passion is in cloud and object storage technologies. He holds a master's degree in electrical engineering from the University of California, Berkeley.

When not working on OpenStack Swift, Amar can be found working on technologies such as Kubernetes, MongoDB, PHP, or jQuery. His blogs can be found at www.buildcloudstorage.com.

I would like to thank my wife for tolerating my late-night and weekend book-writing sessions.

Kris Rajana is a technologist and serial entrepreneur, passionate about building globally distributed teams to deliver innovative infrastructure solutions. His areas of interest include data infrastructure and fast-emerging open source cloud storage technologies, such as OpenStack, Cloud Foundry, Dockers/Containers, and big data. As the CEO of Vedams and Biarca (an offshoot of Vedams), he takes immense pride in his team and its development, which leads to excellence in execution. Kris has over 20 years of experience in managing engineering teams in fields such as space, aviation, storage at BFGoodrich Aerospace, Snap Appliance (currently Overland Storage), Adaptec, Xyratex, and Sullego. His current passion is DevOps, and he likes to leverage leading open source cloud technologies to make enterprises more agile, speed up the development and deployment of modern enterprise applications, and make IT operations more efficient. Kris earned his doctorate in engineering science from Pennsylvania State University.

He is a member of the board of the Pratham Bay Area Chapter. Along with the Vedams team, he is a sponsor of an urban learning center in Hyderabad. He is a student and sevak of the San Jose Chinmaya mission.

I would like to thank my family for their patience and support.
I would also like to thank all my mentors and teachers over the years.

Sreedhar Varma has more than 15 years of experience in the storage industry, and has worked on various storage technologies such as SCSI, SAS, SATA, FC HBA drivers (Adaptec, Emulex, Qlogic, Promise, and so on), RAID, storage stacks of various operating systems, and system software for fault-tolerant and high-availability systems. He has good experience with SAN, NAS, and iSCSI networks; various storage arrays (Dothill, IBM, EMC, Netapp, Oracle Pillar, and so on); object storage implementations (Swift and Ceph); and software development using the corresponding REST APIs.

Sreedhar is currently working for Vedams software providing storage engineering services. In the past, he has worked for Stratus Technologies, Compaq, Digital Equipment Corp, and IBM. He has a master's degree in computer science from the University of Massachusetts.

About the Reviewers

Steve Martinelli is a software developer for IBM, and has been involved with OpenStack since the Grizzly release. He is a core contributor to OpenStack's Identity Service—Keystone. He primarily focuses on enabling Keystone to better integrate into enterprise environments. Steve has helped federated identity, auditing, and OAuth support to Keystone. In his spare time, he contributes to OpenStackClient, PyCADF, and Oslo Policy as a core contributor.

Juan J. Martínez is an experienced software developer with a strong open source background. He has been involved in OpenStack object storage since the Bexar release. His work related to Swift includes the customization and deployment of an award-winning cloud storage solution. He currently maintains a number of open source projects that provide access to the storage using common protocols (FTP, SFTP, and NBD). Juan is currently employed by Memset, a British cloud provider based in Cranleigh.

Christian Schwede is a principal software engineer working at Red Hat. He started working on OpenStack Swift in 2012. He is a core reviewer and contributor to Swift.

Christian's main interests are open source software, storage systems, cloud computing, and software-defined infrastructure.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<https://www2.packtpub.com/books/subscription/packtlib>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	vii
Chapter 1: Cloud Storage – Why Can't I Be Like Google?	1
What constitutes cloud storage?	2
Reduced TCO	2
Unlimited scalability	2
Elastic	2
On-demand	2
Universal access	3
Multitenancy	3
Data durability and availability	3
Limitations of cloud storage	3
Performance	3
New APIs	4
Object storage	5
The importance of being open	6
OpenStack Swift	7
Summary	8
Chapter 2: OpenStack Swift Architecture	9
Logical organization of objects	9
Swift implementation and architecture	10
Key architectural principles	10
Physical data organization	11
Data path software servers	13
A day in the life of a create operation	14
A day in the life of a read operation	15
A day in the life of an update operation	16
A day in the life of a delete operation	16

Post-processing software components	16
Replication	16
Updaters	17
Auditors	17
Other processes	17
Inline middleware options	18
Authentication	18
Other modules	19
Additional features	20
Large object support	20
Metadata	21
Multirange support	21
CORS	21
Server-side copies	21
Cluster health	21
Summary	22
Chapter 3: Installing OpenStack Swift	23
Hardware planning	23
Server setup and network configuration	24
Pre-installation steps	25
Downloading and installing Swift	26
Setting up storage server nodes	27
Installing services	27
Formatting and mounting hard disks	29
RSYNC and RSYNCd	30
Setting up the proxy server node	31
The Keystone service	32
Installing MariaDB	32
Installing Keystone	33
The ring setup	37
Multiregion support	39
Finalizing the installation	40
Storage policies	40
Implementing storage policies	41
Applying storage policies	43
Summary	44
Chapter 4: Using Swift	45
Installing clients	45
Creating a token using Keystone authentication	46
Displaying metadata information for an account, container, or object	47
Using the Swift client CLI	47

Using cURL	48
Using the specialized REST API client	48
Listing containers	49
Using the Swift client CLI	49
Using cURL	50
Listing objects in a container	50
Using the Swift client CLI	50
Using cURL	51
Using the REST API	51
Updating the metadata for a container	52
Using the Swift client CLI	52
Using the REST API	52
Environment variables	52
The pseudo-hierarchical directories	53
Container ACLs	54
Transferring large objects	56
Amazon S3 API compatibility	57
Accessing Swift using S3 commands	59
Accessing Swift using client libraries	60
Java	60
Python	61
Ruby	61
Summary	61
Chapter 5: Additional Swift Interfaces	63
Using Swift for virtual machine storage	63
Swift in Sahara	65
Hadoop Cluster with Sahara	66
Using Swift with Sahara	66
Running a job in Sahara	67
Authenticating with Swift proxy	67
Summary	68
Chapter 6: Monitoring and Managing Swift	69
Routine management	69
Swift cluster monitoring	70
Swift Recon	70
Swift Informant	72
Swift dispersion tool	73
StatsD	73
Swift metrics	74
Tulsi – a Swift health monitoring tool	75

Architecture of Tulsi	76
Deploying Tulsi	76
Running Tulsi	77
Anomaly detection in Tulsi	80
Logging using rsyslog	81
Failure management	81
Detecting drive failures	82
Handling drive failure	82
Handling node failure	83
Proxy server failure	83
Zone and region failure	84
Capacity planning	84
Adding new drives	84
Adding new storage and proxy servers	85
Migrations	85
Summary	87
Chapter 7: Docker Intercepts Swift	89
Swift with Docker	90
Installation of Docker	91
Basic commands for the Docker user	91
Setting up a Swift proxy container using the Docker image	93
Setting up the storage container using the Docker image	94
Setting up a Swift cluster using a Dockerfile	96
Creating a proxy container using a Dockerfile	96
Creating a storage container using a Dockerfile	97
Summary	98
Chapter 8: Choosing the Right Hardware	99
The hardware list	99
The hardware selection criteria	101
Choosing the storage server configuration	101
Determining the region and zone configuration	103
Choosing the account and container server configuration	104
Selecting the proxy server configuration	104
Choosing the network hardware	105
Choosing the ratios of various server types	106
Heterogeneous hardware	107
Choosing additional networking equipment	107
Selecting a cloud gateway	107
Additional selection criteria	108

The vendor selection strategy	109
Branded hardware	109
Commodity hardware	109
Summary	110
Chapter 9: Tuning Your Swift Installation	111
Performance benchmarking	111
Hardware tuning	117
Software tuning	117
Ring considerations	117
Data path software tuning	118
Post-processing software tuning	119
Additional tuning parameters	120
Summary	121
Chapter 10: Additional Resources	123
Use cases	123
Archival	124
Backup	124
Content repository	124
Collaboration	125
Data lakes	125
Operating systems used for OpenStack implementations	125
Virtualization used for OpenStack implementations	127
Provisioning and distribution tools	128
Monitoring and graphing tools	130
Additional information	130
Summary	131
Appendix: Swift CLI Commands	133
Commands	133
list	133
Examples	134
stat	134
Examples	135
post	135
Examples	136
upload	137
Examples	138
download	139
Examples	139
delete	140
Examples	140
Index	143

Preface

CIOs around the world are asking their teams to take advantage of cloud technologies as a way to cut costs and improve usability. OpenStack is a piece of fast-growing open source cloud software with a number of projects, and OpenStack Swift is one such project that allows users to build cloud storage. With Swift, users can not only build storage using inexpensive commodity hardware, but also use the public cloud storage built using the same technology. Starting with the fundamentals of cloud storage and OpenStack Swift, this book will provide you with the skills required to build and operate your own cloud storage or use a third-party cloud. This book is an invaluable tool if you want to get a head start in the world of cloud storage using OpenStack Swift. You will be equipped to build an on-premise private cloud, manage it, and tune it.

What this book covers

Chapter 1, Cloud Storage – Why Can't I Be Like Google?, introduces the need for cloud storage, the underlying technology of object storage, and an extremely popular open source object storage project called OpenStack Swift.

Chapter 2, OpenStack Swift Architecture, discusses the internals of the Swift architecture in detail, and shows you how elegantly Swift converts commodity hardware into reliable and scalable cloud storage.

Chapter 3, Installing OpenStack Swift, walks you through all the necessary steps required to perform a multinode Swift installation, and show you how to set it up along with the Keystone setup for authentication.

Chapter 4, Using Swift, describes the various ways in which you can access Swift object storage. This chapter also provides examples for the various access methods.

Chapter 5, Additional Swift Interfaces, describes the interfaces available for using Swift object storage as data stores (block storage), as well as the Swift interface within Sahara.

Chapter 6, Monitoring and Managing Swift, provides details on the various options that are available for monitoring and managing a Swift cluster. Some of the topics covered in this chapter are StatsD metrics, handling drive failures, node failures, and migrations.

Chapter 7, Docker Intercepts Swift, describes dockerization of Swift services and how to deploy a dockerized Swift image.

Chapter 8, Choosing the Right Hardware, provides you with the information necessary to make the right decision in selecting the required hardware for your cloud storage cluster.

Chapter 9, Tuning Your Swift Installation, walks you through a performance benchmarking tool and the basic mechanisms available for tuning a Swift cluster. Users utilizing Swift will need to tune their installation to optimize performance, durability, and availability, based on their unique workload.

Chapter 10, Additional Resources, explores several use cases of Swift and provides pointers on operating systems, virtualization, and distribution tools used across various Swift installations.

Appendix, Swift CLI Commands, provides details on various commands that can be run from a Swift CLI session.

What you need for this book

The various software components required to follow the instructions in the chapters are as follows:

- Ubuntu operating system 12.04, which can be downloaded from the following sites:
 - <http://www.ubuntu.com/download/server>
 - <http://releases.ubuntu.com/12.04/>
- OpenStack Swift Juno release
- The python-swiftclient Swift CLI
- cURL

- Swift tools such as Swift-Recon, Swift-Informant, and Swift-Dispersion
- A StatsD server from <https://github.com/etsy/statsd/>

Who this book is for

This book is targeted at IT and storage administrators who want to enter the world of cloud storage using OpenStack Swift. It also targets anyone who wishes to understand how to use OpenStack Swift, and developers looking to port their applications to OpenStack Swift.

This book also provides invaluable information for IT management professionals trying to understand the differences between traditional and cloud storage.

Basic knowledge of Linux and server technology will be beneficial if you want to get the most out of the book.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The name of the container should be provided after the `stat` command to get the container information."

A block of code is set as follows:


```
import org.jclouds.ContextBuilder;
import org.jclouds.blobstore.BlobStore;
import org.jclouds.blobstore.BlobStoreContext;
import org.jclouds.openstack.swift.CommonSwiftAsyncClient;
import org.jclouds.openstack.swift.CommonSwiftClient;
```


When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
France.txt segment 0
France.txt segment 1
France.txt segment 2
France.txt
```

Any command-line input or output is written as follows:

```
swift post -r tenant1:user1 cities
```

 Warnings or important notes appear in a box like this.

 Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

1

Cloud Storage – Why Can't I Be Like Google?

If you could build your IT systems and operations from scratch today, would you recreate what you have? That's the question Geir Ramleth, CIO of the construction giant Bechtel, asked himself in 2005. The answer was obviously not, and Bechtel ended up using the best practices from four Internet forerunners of that time – YouTube, Google, Amazon, and Salesforce – to create their next set of data centers.

This is exactly the same question CIOs and IT administrators around the world are asking themselves! In this book, you will learn about a revolutionary new storage system called **cloud storage** that uses the best practices (though not the exact technologies) of these web giants. This will cut the **total cost of ownership (TCO)** of storage by more than 10 times compared to traditional enterprise block or file storage.

This book will show you how you can implement cloud storage using a leading open source storage software stack called OpenStack Swift. Let's first explore some key elements that constitute cloud storage:

- Dramatic reduction in TCO
- Unlimited scalability
- Elasticity achieved by virtualization
- On-demand; that is, pay for what you use
- Universal, that is, access from anywhere
- Multitenancy, which means sharing storage hardware with other departments or companies
- Data durability and availability, even with partial failures of the storage system

What constitutes cloud storage?

Let's review each of these elements of cloud storage in more detail.

Reduced TCO

Reduced TCO is the crux of cloud storage. Unless this new storage cuts storage cost by more than 10 times, it is not worth switching from block or file storage and dealing with something new and different. By total cost of ownership, we mean the total of **capital expenditure (CAPEX)** which involves equipment and **operational expenditure (OPEX)** in the form of IT storage administrators, electricity, power, cooling, and so on. This TCO reduction must be achieved without sacrificing durability (keeping data intact) or availability.

Unlimited scalability

Whether the cloud storage offering is public (that is, offered by a service provider) or private (that is, offered by central IT), it must have unlimited scalability. As we will see, cloud storage is built on distributed systems, which means that it scales very well. Traditional storage systems typically have an upper limit, making them unsuitable for cloud storage.

Elastic

Storage virtualization decouples and abstracts the storage pool from its physical implementation. This means that you can get an elastic (grow and shrink as required) and unified storage pool, when in reality, the underlying hardware is neither. IT professionals who have spent endless hours forecasting data growth and then waiting for their equipment will appreciate the magnitude of this benefit.

On-demand

Consumers do not reserve blocks of electricity and pay for it upfront, yet we routinely pay for storage upfront, whether we use it or not. Cloud storage uses a pay-as-you-go model, where you pay only for the data stored and the data accessed. For a private cloud, there is a minimal cluster to start with, beyond which it is on-demand. This can result in huge cost savings for the storage user.

Universal access

Existing enterprise storage has limitations in terms of access. Block storage is very limiting; a server has to be on the same storage area network, and storage volumes cannot be shared. **Network-attached-storage (NAS)** must be *mounted* to access it. This creates limitations on the number of clients and requires LAN access.

Cloud storage is extremely flexible — there is no limit on the number of users or from where you can access it. This is possible since cloud storage systems usually use a REST API over HTTP (GET, PUT, POST, and DELETE) instead of the traditional SCSI or CIFS/NFS protocols.

Multitenancy

Cloud storage is typically multi-tenant. The tenants may be different organizations in a public cloud or different departments in a private cloud. The benefit is centralized management and higher storage utilization, which reduces costs. Security, often an issue with multi-tenant systems, is addressed comprehensively in cloud storage through strong authentication, access controls, and various encryption options.

Data durability and availability

Cloud storage is able to run on commodity hardware, yet it is highly durable and available. This is even more impressive in that durability and availability is maintained in the face of a partial system failure. As with many modern distributed systems, the burden of data durability and availability is on the software layer rather than the underlying hardware layer.

Limitations of cloud storage

While cloud storage has numerous benefits, there are some limitations in the areas of performance and new APIs.

Performance

Storage systems have struggled to balance reliability, cost, and performance. Generally, you can get two out of these three aspects. Cloud storage optimizes reliability and cost, but not performance. In fact, as we will see later, reliability in cloud storage is better than the traditional RAID when you reach a large scale. By the way RAID works, you are at a very high risk of getting an unrecoverable failure during a RAID rebuild when operating at-scale. Cloud storage uses different techniques such as replication or erasure coding to provide high reliability.

This means that cloud storage is well suited for applications such as web servers and application servers, but not for databases or high-performance computing. It is also suitable for tier 2/3 storage, for example, backup, archival (photos, documents, videos, logs, and so on), and creating an additional copy for disaster recovery.

New APIs

Cloud storage affects applications in two ways: its interface with storage and its behavior. Firstly, applications need to port to a new and different storage interface utilizing HTTP instead of SCSI or CIFS/ NFS. Secondly, applications need to handle an eventually consistent storage system. The second part requires explanation.

Cloud storage is built using distributed systems that are governed by a theorem called the **CAP theorem**, which states that out of the following three points, it is impossible to guarantee more than two:

- **Consistency:** For cloud storage, this means that a request to any region or node returns the same data
- **Availability:** For cloud storage, this signifies that a request is successfully acknowledged with a response other than no response or an error
- **Tolerance to partial failures:** For cloud storage, this implies that the architecture is able to withstand failures in connectivity or parts of the system

Most cloud storage systems guarantee availability and tolerance to partial failures at the expense of consistency, making the system eventually consistent. This means that an operation such as an update may not be reflected to all nodes at the same time. Traditional applications expect strict consistency and may need to be modified.

If an application has not ported to cloud storage, is that a dead end? Fortunately not. There is a class of devices called cloud gateway that provides file or block interfaces to an application (for example, CIFS, NFS, iSCSI, or FTP/SFTP) and performs protocol conversion on the cloud. These gateways provide other functions as well, such as caching, WAN optimization, optional compression, encryption, and deduplication. They also eliminate the need for an application to handle the eventual consistency problem.

Object storage

How do you build a cloud storage system? The most suitable underlying technology is object storage.

Object storage is different from block or file storage as it allows a user to store data in the form of objects (essentially files) in a flat namespace using REST HTTP APIs. Object storage completely virtualizes the physical implementation from the logical presentation. It is similar to check-in luggage versus carry-on luggage, where once you put your check-in luggage in the system, you really don't know where it is. You simply get it back at your destination. With carry-on luggage, you have to know exactly where you have kept it at all times.

Object storage is built using scale-out distributed systems. Each node, most often, actually runs on a local filesystem. As we will see, object storage architectures allow for the use of commodity hardware, as opposed to specialized, expensive hardware used by traditional storage systems. The most critical tasks of an object storage system are as follows:

- Data placement
- Automating management tasks, including durability and availability

Typically, a user sends their HTTP `GET`, `PUT`, `POST`, `HEAD`, or `DELETE` request to any one node out from a set of nodes, and the request is translated to physical nodes by the object storage software. The software also takes care of the durability model by doing any one of the following: creating multiple copies of the object, chunking it, creating erasure codes, or a combination of these.

The durability model is not RAID because, as discussed earlier, RAID simply does not scale beyond hundreds of terabytes. The second critical task deals with management, such as periodic health checks, self-healing, and data migration. Management is also made easy by using a single flat namespace, which means that a storage administrator can manage the entire cluster as a single entity.

Let's evaluate through the following table how object storage meets the aforementioned cloud storage benefits:

Criteria	Ability to meet
Low TCO	Storage nodes have no special requirements such as high availability, management, or special hardware such as RAID. This means that commodity hardware can be used to cut capital expenses (CAPEX). A single flat namespace with automated management features allows you to cut operational expenses (OPEX). A full analysis of how this cuts the TCO by 10 times or more is beyond the scope of this book.
Unlimited scalability	A distributed architecture allows capacity and performance to scale.
Elasticity	A fully virtualized approach allows data to grow and shrink as necessary.
On-demand	A fully virtualized approach with centralized management allows storage to be offered as an on-demand self-service resource.
Universal access	REST HTTP APIs provide access from wherever the user is, with no restriction on the number of users.

The importance of being open

Although the need for software to be open is not a technical requirement, it is increasingly becoming a business requirement. Open means three things:

- **Open source:** While there are numerous benefits of open source software, the key advantages are the users' ability to influence the direction of the project, the velocity of innovation, reduced license fees, and the ability to switch vendors.
- **Open APIs:** To avoid vendor lock-in, the APIs must be open. Often, proprietary APIs are enticing upfront but lock users in.
- **Agnostic to underlying hardware choices:** To reduce hardware costs and maintain users' preferences, the software needs to be hardware agnostic.

OpenStack Swift

OpenStack Swift is a leading open source object storage project that meets the mentioned object storage and open technology requirements, and is the topic of this book. Let us first look at what the OpenStack project is about, and then specifically what OpenStack Swift (also referred to as just Swift) is.

OpenStack, a project launched by NASA and Rackspace in 2010, is currently the fastest growing open source project, and its mission is to produce a cloud computing platform useful for both public and private implementations. Its two core principles are simplicity and scalability. OpenStack has numerous subprojects under its umbrella, ranging from computing and storage to networking, among others. The object storage project is called Swift and is a highly available, durable, distributed, masterless, and eventually consistent software stack.

The Swift project, in particular, came out of Rackspace's cloud files platform. The project was unique because it utilized a DevOps methodology, where the engineers and ops professionals worked together to create and operate it. This resulted in a very powerful storage system that is simple, yet easy to manage. Rackspace made Swift open source in 2010, and the leading contributors include SwiftStack, Rackspace, Red Hat, HP, Intel, IBM, and others.

In addition to sharing the mentioned generic object storage characteristics, OpenStack Swift has some unique additional functionality, as follows:

- **Open source:** Comes with no license fees, as mentioned previously.
- **Open standards:** Using HTTP REST APIs with SSL for optional encryption. The combination of open source and open standards eliminates any potential vendor lock-in.
- **Account container object structure:** OpenStack Swift incorporates rich naming and organization capacity, unlike a number of object storage systems that offer a primitive interface, where the user gets a key upon submitting an object. In these other systems, the burden of mapping names to keys and organizing them in a reasonable manner is left to the user. Swift, on the other hand, handles the organization of data along with multitenancy.
- **Global cluster capability:** This allows replication and distribution of data around the world. This functionality helps with disaster recovery, distribution of hot data, and so on.

- **Storage policies:** This feature allows sets of data (stored in separate containers) to be optionally stored on different types of underlying storage using different durability models. For example, a valuable set of digital assets can be stored on high-quality hardware using triple replication, while less important assets can be stored on lower quality hardware with a lower level of replication. Hot data could be stored on SSDs.
- **Partial object retrieval:** For example, you want just a portion of a movie object or a TAR file.
- **Middleware architecture:** This allow users to add functionality. A great example of this is integrating with an authentication system.
- **Large object support:** Objects of any size can be stored.
- **Additional functionality:** This includes object versioning, causing objects to expire, rate limiting, temporary URL support, CNAME lookup, domain remap, account-to-account data copy, quota support, and static web mode. This list is constantly growing as a consequence of Swift being an open source project.

Summary

In this chapter, we saw why cloud storage is a new way of building storage systems that cuts the total cost of ownership significantly. It uses a technology called object storage. A high-quality and open source object storage software stack to consider is OpenStack Swift. OpenStack Swift uses a dramatically different architecture from traditional enterprise storage systems by using a distributed architecture on commodity servers. The next chapter explains this architecture in detail.

2

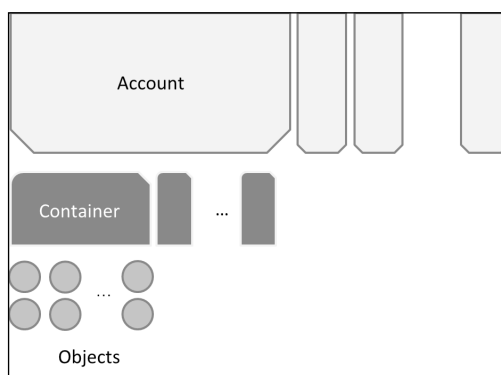
OpenStack Swift Architecture

OpenStack Swift is the magic that converts a set of unconnected commodity storage servers into a scalable, durable, and easy-to-manage storage system. We will look at Swift's architecture (based on the Juno release) in detail by understanding the logical organization of objects and how Swift organizes this data by virtualizing the underlying physical hardware. This chapter then covers data path software servers and contains a walkthrough of the four basic operations (create, read, update, and delete) and post-processing software. The chapter concludes with inline middleware options and additional key features.

Logical organization of objects

First, let's look at the logical organization of objects and then how Swift completely abstracts and maps objects on the physical hardware.

A **tenant** is assigned an **account**. A tenant can be any entity — a person, department, company, and so on. The account holds **containers**. Each container holds **objects**, as shown in the following figure. You can think of objects essentially as files.



Logical organization of objects in Swift

A tenant can associate additional **users** to an account. Users can keep adding containers and objects within a container without having to worry about any physical hardware boundaries, unlike traditional file or block storage. Containers within an account obviously require a unique name, but two containers in separate accounts can have the same name. Containers are flat, and objects are not stored hierarchically, unlike files stored in a filesystem, where directories can be nested. This simplifies the design and removes a number of performance issues with hierarchical filesystems. However, Swift does provide a mechanism to simulate **pseudo-directories** by inserting a forward slash (/) delimiter in the object name.

Swift implementation and architecture

The two key issues that any storage system has to solve are as follows:

- Where to put and fetch data
- How to keep the data durable and available

We will explore these two core issues through the upcoming discussion on architecture and implementation.

Key architectural principles

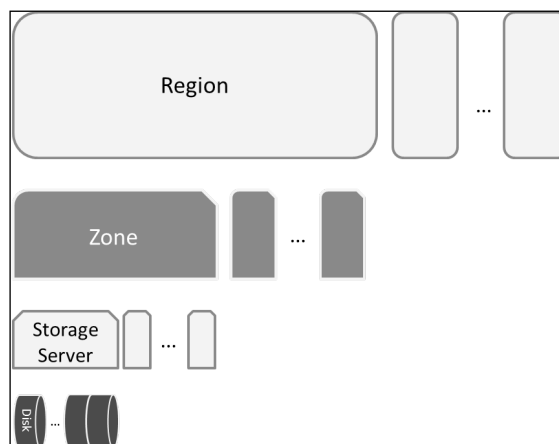
Some key architectural principles behind Swift are as follows:

- **Masterless design:** A master in a system creates both a failure point and a performance bottleneck. Masterless design removes this and also allows multiple members of the cluster to respond to API requests.
- **Loose coupling:** There is no need for tight communication in the cluster. This is also essential for preventing performance and failure bottlenecks.
- **Load spreading:** Unless the load is spread out, performance, capacity, account, container, and object scalability cannot be achieved.
- **Self-healing:** The system must automatically adjust for hardware failures. As per the CAP theorem discussed in *Chapter 1, Cloud Storage – Why Can't I Be Like Google?*, Swift is designed to tolerate partial system failures.
- **Multi-tenancy layer:** A number of object storage systems simply return a hash key for a submitted object and provide a completely flat namespace. The task of creating accounts and containers and mapping keys to object names is left to the user. Swift simplifies life for the user and provides a well-designed data organization layer.

- **Available and eventually consistent:** As discussed in *Chapter 1, Cloud Storage – Why Can't I Be Like Google?*, Swift needs to maximize the availability of data for the user and ensure that data is consistent across all nodes in the shortest possible time.
- **Heterogeneous:** The various storage nodes that make up Swift do not need to be identical. In fact, as we will see, it is possible to set policies to place objects of on specific hardware.

Physical data organization

Swift completely abstracts logical organization of data from the physical organization. At a physical level, Swift classifies the location of data into a hierarchy, as shown in the following figure:



Physical data location hierarchy

The hierarchy is as follows:

- **Region:** At the highest level, Swift stores data in regions that are geographically separated and thus suffer from a high-latency link. A user may use only one region if, for example, the cluster utilizes only one data center.
- **Zone:** Within regions, there are zones. Zones are a set of storage nodes that share different availability characteristics. Availability may be a function of different physical buildings, power sources, or network connections. This means that a zone could be a single storage server, a rack, or a complete data center depending on your requirements. Zones need to be connected to each other via low-latency links. Rackspace recommends having at least five zones per region.

- **Storage servers:** A zone consists of a set of storage servers ranging from one to several racks.
- **Disk (or devices):** Disk drives are part of a storage server. These can be inside the server or connected via a **just a bunch of disks (JBOD)**. The devices can be spinning disks or SSDs.

Swift stores a number of replicas (the default is 3) of an object on different disks. Due to the use an as-unique-as-possible algorithm, these replicas are as "far" away as possible in terms of being in different regions, zones, storage servers, and disks. This algorithm first tries to put data in different regions, then zones, then servers, and finally disks. The algorithm is responsible for the durability aspect of Swift. It also improves data availability since all three nodes would have to be unavailable for data to be unavailable.

Swift uses a semi-static table to determine where to place objects and their replicas. It is semi-static because the look-up table, called a **ring**, in Swift is created by an external process called the **ring builder**. The ring can be modified, but not dynamically and never by Swift. It is not distributed, so every node that deals with data placement has a complete copy of the ring. The ring has entries in it called **partitions** (this term is not to be confused with the more commonly referred-to disk partitions). Essentially, an object is mapped on a partition, and the partition provides the devices where the replicas of an object should be stored. The ring also provides a list of handoff devices should any of the initial ones fail. There are rings for accounts and containers as well, used for the same purpose. There is only one account and container ring for an entire cluster, but there can be more than one object ring to support a very important capability, which described next.

Swift allows objects in different containers to reside on different sets of hardware while still being in the same cluster. This is very useful, since administrators can create policies for data placement based on characteristics such as these:

- **Durability:** Dual replication, triple replication, and so on
- **Hardware type:** Spinning disks, SSD, and so on
- **Geography:** Spread across multiple regions, locked to a specific geography, and so on

Administrators can name these policies as something meaningful for users. Users can then decide which policy to use for a particular container. For example, medical images may be put in a container with a triple replication policy using geo-replication on nodes of high disk density.

Logs may be put in a container with dual replication in a single geography that utilize nodes with SSDs. Policies are enabled by the ability of Swift to support multiple object rings. Each ring corresponds to a policy and includes a specific set of devices that the ring should redirect object requests to.

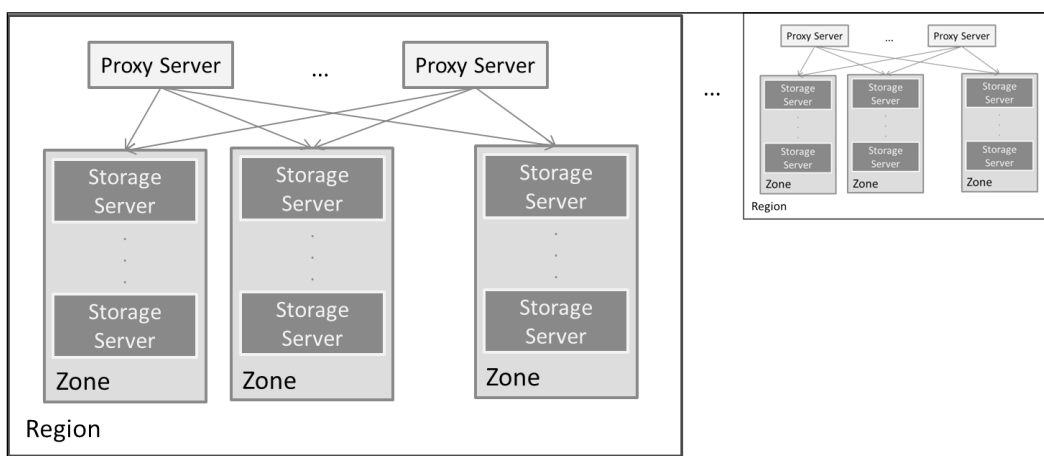
The actual storage of the object is done on a filesystem that resides on the disk, for example, **Extent file system (XFS)**. Account and container information is kept in SQL databases for concurrent access. The account database contains a list of all its containers, and the container database contains a list of all its objects. These databases are stored in single files, and the files are replicated just like any other object.

Data path software servers

The data path consists of the following four software servers. These are technically services but the Swift documentation calls them servers. To be consistent, we will call them servers too:

- Proxy server
- Account server
- Container server
- Object server

Unless you need performance, the account, container, and object servers are often put into one physical server and are called a **Storage server** (or node), as shown in the following figure:



Data path software servers; a storage server includes account, container, and object servers

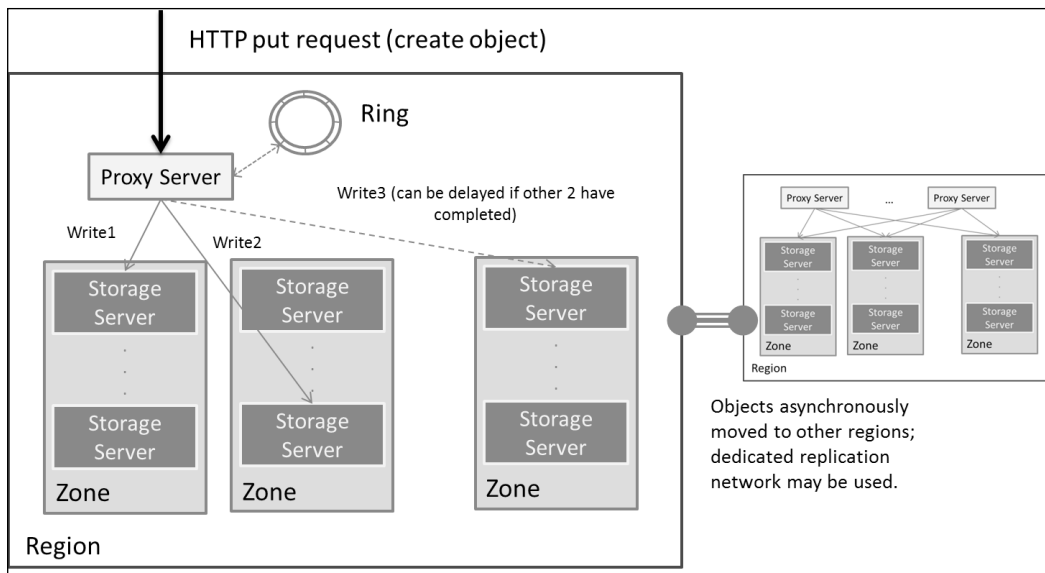
A description of each server type is as follows:

- **Proxy server:** The proxy server is responsible for accepting HTTP requests from a user. It will look for the location of the storage server (or servers) where the request needs to be routed by utilizing the appropriate ring. The proxy server accounts for failures by looking up handoff nodes and performs read/write affinity (by sending writes or reads to the same region; refer to the *A day in the life of a create operation* and *A day in the life of a read operation* sections). When objects are streamed to or from an object server, they are streamed directly through the proxy server as well. Moreover, proxy servers are also responsible for the read/write quorum and often host inline middleware (discussed later in this chapter).
- **Account server:** The account server tracks the names of containers in a particular account. Data is stored in SQL databases; database files are further stored in the filesystem. This server also tracks statistics, but does not have any location information about containers. The proxy server determines the location information based on the container ring. Normally, this server is hosted on the same physical server with container and object servers. However, in large installations, this may need to be on a separate physical server.
- **Container server:** This server is very similar to the account server, except that it deals with object names in a particular container.
- **Object server:** Object servers simply store objects. Each disk has a filesystem on it, and objects are stored in these filesystems.

Let us stitch the physical organization of the data with the various software components and explore the four basic operations: `create`, `read`, `update`, and `delete` (known as CRUD). For simplicity, we will focus on the object server, but it may be further extrapolated to both account and container servers too.

A day in the life of a create operation

A `create` request is sent via an HTTP `PUT` API call to a proxy server. It does not matter which proxy server gets the request because Swift is a distributed system and all proxy servers are created equal. The proxy server interacts with the ring associated with the policy of the container to get a list of disks and associated object servers to write data to. As we covered earlier, these disks will be as unique as possible. If certain disks have failed or are unavailable, the ring provides handoff devices. Once the majority of disks acknowledge the write (for example, two in the case of triple replication), the operation is returned as successful. Assuming that the remaining writes complete successfully, we are fine. If not, the replication process, shown in the following figure, ensures that the remaining copies are ultimately created:



A day in the life of a create operation

The `create` operation works slightly differently in a multi-region cluster. All copies of the object are written to the local region. This is called **write affinity**. The object is then asynchronously moved to another region (or regions). A dedicated replication network may be used for this operation.

A day in the life of a read operation

A `read` request is sent via an HTTP `GET` API call to a proxy server. Again, any proxy server can receive this request. Similar to what happens in the `create` operation, the proxy server interacts with the appropriate ring to get a list of disks and associated object servers. The `read` request is issued to object servers in the same region as the proxy server. This is called **read affinity**. For a multi-region implementation, eventual consistency presents a problem, since different regions might have different versions of an object. To get around this issue, a `read` operation for an object with the latest timestamp may be requested. In this case, proxy servers first request the timestamp from all the object servers and read from the server with the newest copy. Similar to the `write` case, in case of failure, handoff devices may be requested.

A day in the life of an update operation

An `update` request is handled in the same manner as a `create` request. Objects are stored with their timestamp to make sure that when read, the latest version of the object is returned. Swift also supports a versioning feature on a per container basis. When this is turned on, older versions of the object are also made available in a special container called `versions_container`.

A day in the life of a delete operation

A `delete` request sent via an HTTP `DELETE` API call is treated like an update, but instead of a new version, a "tombstone" version with 0 bytes is placed. A delete operation is very difficult in a distributed system, since the system will essentially fight deletion by recreating deleted copies to ensure that an object has the right number of replicas. The Swift solution is indeed very elegant and eliminates the possibility of deleted objects suddenly showing up again.

Post-processing software components

There are three key post-processing software components that run in the background, as opposed to being part of the data path.

Replication

Replication is a very important aspect of Swift. It ensures that the system is consistent; that is, all servers and disks assigned by the ring to hold copies of an object or database indeed have the latest version. This process protects against failures, hardware migration, and ring rebalancing (where the ring is changed and data has to be moved around). This is accomplished by comparing local data with the remote copy. If the remote copy needs to be updated, the replication process "pushes" a copy. The comparison process is efficient and is carried out by simply comparing hash lists rather than each byte of an object (or account or container database). Replication uses `rsync`, a Linux-based remote file synchronization utility, to copy data. Replication alternatives such as `ssync` and Swift primitives for replication are also available.

Updaters

In certain situations, account or container servers may be busy due to heavy load or being unavailable. In this case, the update is queued onto the storage server's local storage. There are two **updaters** that process these queued items. The object updater will update objects in the container database, while the container updater will update containers in the account database. This situation could lead to an interesting eventual consistency behavior, where the object is available but the container listing does not have it at that time. These windows of inconsistency are generally very small.

Auditors

Auditors walk through every object, container, and account to check their data integrity. This is done by computing an MD5 hash and comparing it to the hash stored under the `Etag` metadata key of the object (see the *Metadata* section later). The `Etag` metadata key is created when the object is first written. If the item is found corrupted, it is moved to a quarantine directory, and in time, the replication process will create a clean copy. This is how the system is self-healing. The MD5 hash is also available for the user so that they can perform operations such as comparing the hash in their local database against the one stored in Swift.

Other processes

The other background processes are as follows:

- **Account reaper:** This process is responsible for deleting an entire account once it is marked for deletion in the database.
- **Object expirer:** Swift allows users to set retention policies by providing "delete at" or "delete after" information for objects. This process ensures that expired objects are deleted.
- **Drive audit:** This is another useful background process that looks out for bad drives and unmounts them. It can be more efficient than letting the auditor deal with this type of failure.
- **Container-to-container synchronization:** Using the container-to-container synchronization process, all contents of a container can be mirrored to another container. These containers can be in different clusters, and the operation uses a secret sync key. Before multi-region support, this feature was the only way to get multiple copies of your data in two or more regions. In addition to being an alternative way to perform replication, this feature is useful for hybrid (private-public combination) or community clouds (multiple private clouds).

- **Container reconciler:** The combination of policies and eventual consistency presents the risk of objects being written to the wrong policy for an interim period. This process corrects any such writes.

Inline middleware options

In addition to the aforementioned core data path components, other items may also be placed in the data path to extend Swift's functionality. This is done by taking advantage of Swift's architecture, which allows middleware to be inserted. The following is a non-exhaustive list of various middleware modules. Most of them apply only to the proxy server, while some modules such as logging and recon do apply to other servers as well.

Authentication

Authentication is one of the most important inline functions. All of Swift's middleware is separate and is used to extend Swift. Thus, authentication systems are separate projects and one out of several may be chosen. Keystone authentication is the official OpenStack identity service and may be used in conjunction with Swift, though there is nothing to prevent a user from creating their own authentication system or using others such as Swauth or TempAuth.

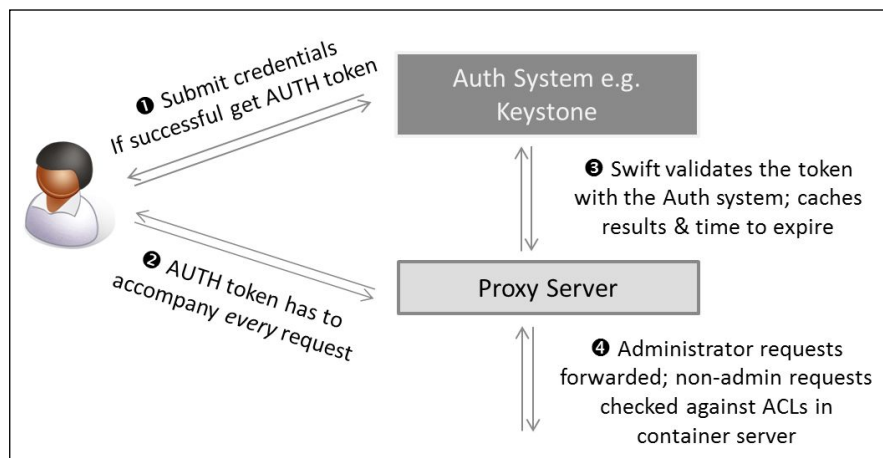
Authentication works as follows:

1. A user presents credentials to the auth system. This is done by executing an HTTP REST API call.
2. The auth system provides the user with an AUTH token.
3. The AUTH token is not unique for every request and expires after a certain duration (in the case of TempAuth, the default time-to-live duration is 86,400 seconds).

Every request made to Swift has to be accompanied by the AUTH token.

4. Swift validates the token with the auth system and caches the result. The result is flushed upon expiration.
5. The auth system generally has the concept of administrator accounts (tenant) and non-admin (user) accounts. Administrator requests are obviously passed through.
6. Non-admin requests are checked against container-level **access control lists (ACL)**. These lists allow the administrator to set read-and-write ACLs for each non-admin user.

7. Therefore, for non-admin users, the ACL is checked before the proxy server proceeds with the request. The following figure illustrates the steps involved when Swift interacts with the auth system:



Other modules

A number of other Swift and third-party middleware modules are available; the following are a few examples:

- **Logging:** Logging is a very important module. A user may insert their custom log handler as well.
- **Health check:** This module provides a simple way to monitor the proxy server to check whether it is alive. Simply access the `/healthcheck` path and the module will respond with OK or whatever the fail case is.
- **Domain remap:** This middleware allows you to remap the account and container name from the path to the host domain name. This allows you to simplify domain names.
- **CNAME lookup:** Using this software, you can create friendly domain names that remap directly to your account or container. CNAME lookup and domain remap may be used in conjunction.
- **Rate limiting:** Rate limiting is used to limit the rate of requests that result in database writes to account and container servers.
- **Container and account quotas:** An administrator can set container or account quotas in terms of bytes using these two middleware modules.
- **Bulk delete:** This middleware allows bulk operations such as deletion of multiple objects or containers.

- **Bulk archive autoextraction:** For bulk expansion of compressed and uncompressed tarball (TAR, `tar.gz`, and `tar.bz2`) files to be performed with a single command, use this software.
- **TempURL:** The TempURL middleware allows you to create a URL that provides temporary access to an object. This access is not authenticated but expires after a certain duration of time. Furthermore, the access is only to a single object, and no other objects can be accessed via the URL.
- **Swift origin server:** This is a module that allows the use of Swift as an origin server to a **content delivery network (CDN)**.
- **Static web:** This software converts Swift into a static web server. You can also provide CSS style sheets to establish full control over the look and feel of your pages. Obviously, requests can be from a nonauthenticated source.
- **Form post:** By using the form post middleware, you get the ability to upload objects to Swift using standard HTML form posts. The middleware converts different POST requests to different PUT requests, and the requests do not go through authentication to allow collaboration across users and non-users of the cluster.
- **Recon:** Recon is a piece of software useful for management. It provides monitoring and returns various metrics about the cluster.
- **Profiler:** This middleware accumulates CPU timing statistics for all incoming requests. It can be very useful for understanding performance issues and for tuning purposes.

Additional features

Swift has additional features that were not covered in the previous sections. The following sections detail some of the additional features.

Large object support

Swift places a limit on the size of a single uploaded object (the default is 5 GB), yet allows storage and downloading of virtually unlimited-size objects. The technique used is segmentation. An object is broken into equal-size segments (except the last one) and uploaded. These uploads are efficient since no one segment is unreasonably large, and data transfers can be done in parallel. Once the uploads are completed, a manifest file, which shows how the segments form a single large object, is uploaded. The download is a single operation where Swift concatenates the various segments to recreate the large object.

Metadata

Swift allows custom metadata to be attached to accounts, containers, or objects that are set and retrieved in the form of custom headers. The metadata is simply a key/value pair (key means name). Metadata may be provided at the time of creating an object (using `PUT`) or updated later (using `POST`). It may be retrieved independently of the object using the `HEAD` method. Examples of metadata range from `ETag`, mentioned in the *Auditors* section, to custom tags such as the patient name, doctor name, x-ray date for a medical image, and so on.

Multirange support

The HTTP specification allows multirange `GET` operations, and Swift supports this by retrieving multiple ranges of an object rather than the entire object.

CORS

Cross-origin resource sharing (CORS) is a specification that allows JavaScript running in a browser to make a request to domains other than where it came from. Swift supports this, and this feature makes it possible for you to host your web pages with JavaScript on one domain and request objects from a Swift cluster on another domain. Swift also supports a broader cross-domain policy file where other client-side technologies such as Flash, Java, and Silverlight can interact with Swift that is in a different domain.

Server-side copies

Swift allows you to copy objects across containers or accounts entirely using server-side copy operations. Since the entire copy operation is performed on the server side, the client is offloaded.

Cluster health

A tool called **swift-dispersion-report** may be used to measure the overall cluster health. It does so by ensuring that the various replicas of an object and container are in their proper places.

Summary

In summary, Swift takes a set of commodity servers and creates a durable and scalable storage system that is simple to manage. In this chapter, we reviewed the Swift architecture and major functionalities. The next chapter will show you how you can install Swift on your own environment using multiple servers.

3

Installing OpenStack Swift

The previous chapter should have given you a good understanding of OpenStack Swift's architecture. This chapter is meant for IT administrators who want to install OpenStack Swift. The version discussed here is the Juno release of OpenStack. Installation of Swift has several steps and requires careful planning before beginning the process.

A simple installation consists of installing all Swift components on a single node, and a complex installation consists of installing Swift on several proxy server nodes and storage server nodes. The number of storage nodes can be in the order of thousands across multiple zones and regions. Depending on your installation, you need to decide on the number of proxy server nodes and storage server nodes that you will configure. This chapter demonstrates a manual installation process; advanced users may want to use utilities such as Puppet or Chef to simplify the process.

This chapter walks you through an OpenStack Swift cluster installation that contains one proxy server and five storage servers. As explained in *Chapter 2, OpenStack Swift Architecture*, storage servers include account, container, and object servers.

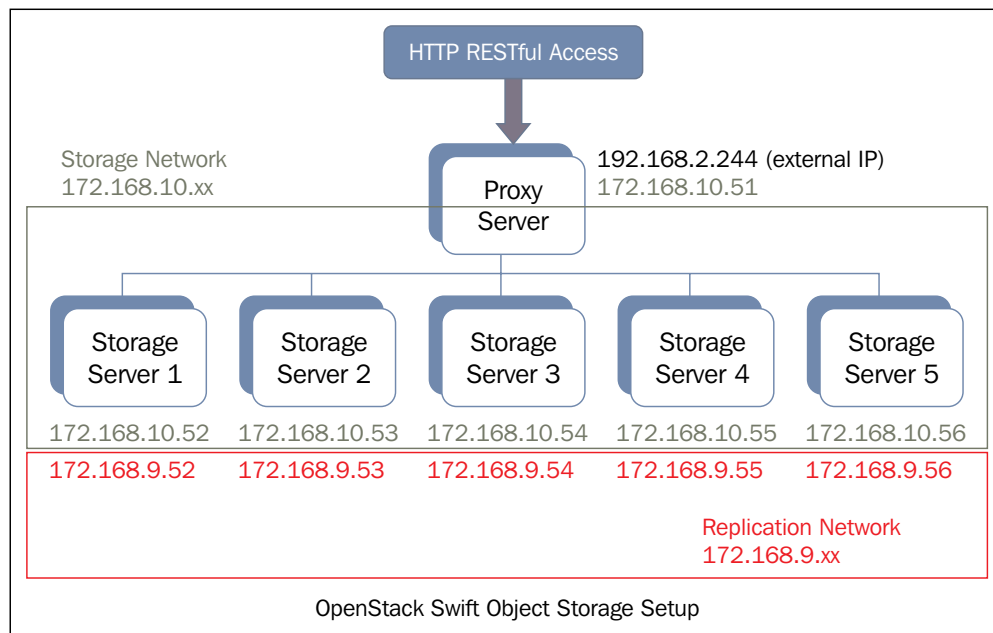
Hardware planning

This section describes the various hardware components involved in the setup (see *Chapter 8, Choosing the Right Hardware*, for a complete discussion on this topic). Since Swift deals with object storage, disks are going to be a major part of hardware planning. The size and number of disks required should be calculated based on your requirements.

Networking is also an important component, where factors such as a public or private network and a separate network for communication between storage servers need to be planned. Network throughput of at least 1 GB per second is suggested, while 10 GB per second is recommended.

The servers we set up as proxy and storage servers are dual quad-core servers with 12 GB of RAM.

In our setup, we have a total of 15 x 2 TB disks for Swift storage; this gives us a total size of 30 TB. However, with in-built replication (with a default replica count of 3), Swift maintains three copies of the same data. Therefore, the effective capacity for storing files and objects is approximately 10 TB, taking filesystem overhead into consideration. This is further reduced due to less than 100 percent utilization. The following figure depicts the nodes of our Swift cluster configuration:



The storage servers have container, object, and account services running in them.

Server setup and network configuration

All the servers are installed with the Ubuntu server operating system (64-bit LTS version 14.04). You'll need to configure three networks, which are as follows:

- **Public network:** The proxy server connects to this network. This network provides public access to the API endpoints within the proxy server.

- **Storage network:** This is a private network and it is not accessible to the outside world. All the storage servers and the proxy server will connect to this network. Communication between the proxy server and the storage servers and communication between the storage servers take place within this network. In our configuration, the IP addresses assigned in this network are 172.168.10.0 and 172.168.10.99.
- **Replication network:** This is also a private network that is not accessible to the outside world. It is dedicated to replication traffic, and only storage servers connect to it. All replication-related communication between storage servers takes place within this network. In our configuration, the IP addresses assigned in this network are 172.168.9.0 and 172.168.9.99. This network is optional, and if it is set up, the traffic on it needs to be monitored closely.

Pre-installation steps

In order for various servers to communicate easily, edit the `/etc/hosts` file and add the host names of each server in it. This has to be done on all the nodes. The following screenshot shows an example of the contents of the `/etc/hosts` file of the proxy server node:

127.0.0.1	localhost
192.168.2.244	swift-proxy
172.168.10.51	s-swift-proxy
172.168.10.52	swift-storage1
172.168.10.53	swift-storage2
172.168.10.54	swift-storage3
172.168.10.55	swift-storage4
172.168.10.56	swift-storage5

Install the **Network Time Protocol (NTP)** service on the proxy server node and storage server nodes. This helps all the nodes to synchronize their services effectively without any clock delays.

The pre-installation steps to be performed are as follows:

1. Run the following command to install the NTP service:

```
# apt-get install ntp
```

Configure the proxy server node to be the reference server for the storage server nodes to set their time from the proxy server node.

2. Make sure that the following line is present in `/etc/ntp.conf` for NTP configuration in the proxy server node:

```
server ntp.ubuntu.com
```

3. For NTP configuration in the storage server nodes, add the following line to `/etc/ntp.conf`. Comment out the remaining lines with server addresses such as `0.ubuntu.pool.ntp.org`, `1.ubuntu.pool.ntp.org`, `2.ubuntu.pool.ntp.org`, and `3.ubuntu.pool.ntp.org`:

```
# server 0.ubuntu.pool.ntp.org
# server 1.ubuntu.pool.ntp.org
# server 2.ubuntu.pool.ntp.org
# server 3.ubuntu.pool.ntp.org
server s-swift-proxy
```

4. Restart the NTP service on each server with the following command:

```
# service ntp restart
```

Downloading and installing Swift

The Ubuntu Cloud Archive is a special repository that provides users with the ability to install new releases of OpenStack.

The steps required to download and install Swift are as follows:

1. Enable the capability to install new releases of OpenStack, and install the latest version of Swift on each node using the following commands. The second command shown here creates a file named `cloudarchive-juno.list` in `/etc/apt/sources.list.d`, whose content is "deb `http://ubuntu-cloud.archive.canonical.com/ubuntu`":

```
# apt-get install ubuntu-cloud-keyring
# echo "deb http://ubuntu-cloud.archive.canonical.com/ubuntu" \
"trusty-updates/juno main" > /etc/apt/sources.list.d/cloudarchive-juno.list
```

2. Now, update the OS using the following command:
3. On all the Swift nodes, we will install the prerequisite software and services using this command:

```
# apt-get install swift rsync memcached python-netifaces
python-xattr python-memcache
```

4. Next, we create a Swift folder under `/etc` and give users the permission to access this folder, using the following commands:

```
# mkdir -p /etc/swift/
# chown -R swift:swift /etc/swift
```

5. Download the `/etc/swift/swift.conf` file from GitHub using this command:


```
# curl -o /etc/swift/swift.conf \
https://raw.githubusercontent.com/openstack/swift/stable/juno/
etc/swift.conf-sample
```
6. Modify the `/etc/swift/swift.conf` file and add a variable called `swift_hash_path_suffix` in the `swift-hash` section. We then create a unique hash string using `# python -c "from uuid import uuid4; print uuid4()"` or `# openssl rand -hex 10`, and assign it to this variable, as shown in the following configuration option:

```
[swift-hash]
# random unique strings that can never change (DO NOT LOSE)
swift_hash_path_prefix = bd08f643f5663c4ec607
swift_hash_path_suffix = f423bf7ab663888fe832
```

7. We then add another variable called `swift_hash_path_prefix` to the `swift-hash` section, and assign to it another hash string created using the method described in the preceding step. These strings will be used in the hashing process to determine the mappings in the ring. The `swift.conf` file should be identical on all the nodes in the cluster.

Setting up storage server nodes

This section explains additional steps to set up the storage server nodes, which will contain the object, container, and account services.

Installing services

The first step required to set up the storage server node is installing services. Let's look at the steps involved:

1. On each storage server node, install the packages for `swift-account` services, `swift-container` services, `swift-object` services, and `xfsprogs` (XFS Filesystem) using this command:


```
# apt-get install swift-account swift-container swift-object
xfsprogs
```
2. Download the `account-server.conf`, `container-server.conf`, and `object-server.conf` samples from GitHub, using the following commands:


```
# curl -o /etc/swift/account-server.conf \
```



```
https://raw.githubusercontent.com/openstack/swift/stable/juno/
etc/account-server.conf-sample

# curl -o /etc/swift/container-server.conf \
https://raw.githubusercontent.com/openstack/swift/stable/juno/
etc/container-server.conf-sample

# curl -o /etc/swift/object-server.conf \
https://raw.githubusercontent.com/openstack/swift/stable/juno/
etc/object-server.conf-sample
```

3. Edit the `/etc/swift/account-server.conf` file with the following section:

```
[DEFAULT]
bind_ip = 172.168.10.52
bind_port = 6002
user = swift
swift_dir = /etc/swift
devices = /srv/node

[pipeline:main]
pipeline = healthcheck recon account-server

[filter:recon]
use = egg:swift#recon
recon_cache_path = /var/cache/swift
```

4. Edit the `/etc/swift/container-server.conf` file with this section:

```
[DEFAULT]
bind_ip = 172.168.10.52
bind_port = 6001
user = swift
swift_dir = /etc/swift
devices = /srv/node

[pipeline:main]
pipeline = healthcheck recon container-server

[filter:recon]
use = egg:swift#recon
recon_cache_path = /var/cache/swift
```

5. Edit the `/etc/swift/object-server.conf` file with the following section:

```
[DEFAULT]
bind_ip = 172.168.10.52
bind_port = 6000
user = swift
swift_dir = /etc/swift
devices = /srv/node

[pipeline:main]
pipeline = healthcheck recon object-server

[filter:recon]
use = egg:swift#recon
recon_cache_path = /var/cache/swift
```

Formatting and mounting hard disks

On each storage server node, we need to identify the hard disks that will be used to store the data. We will then format the hard disks and mount them on a directory, which Swift will then use to store data. We will not create any RAID levels or subpartitions on these hard disks because they are not necessary for Swift. They will be used as entire disks. The operating system will be installed on separate disks, which will be RAID configured.

First, identify the hard disks that are going to be used for storage and format them. In our storage server, we have identified `sdb`, `sdc`, and `sdd` to be used for storage.

We will perform the following operations on `sdb`. These four steps should be repeated for `sdc` and `sdd` as well:

1. Carry out the partitioning for `sdb` and create the filesystem using this command:

```
# fdisk /dev/sdb
# mkfs.xfs /dev/sdb1
```
2. Then let's create a directory in `/srv/node/sdb1` that will be used to mount the filesystem. Give the permission to the `swift` user to access this directory. These operations can be performed using the following commands:

```
# mkdir -p /srv/node/sdb1
# chown -R swift:swift /srv/node/sdb1
```

3. We set up an entry in `fstab` for the `sdb1` partition in the `sdb` hard disk, as follows. This will automatically mount `sdb1` on `/srv/node/sdb1` upon every boot. Add the following command line to the `/etc/fstab` file:

```
/dev/sdb1 /srv/node/sdb1 xfs  
noatime,nodiratime,nobarrier,logbufs=8 0 2
```
4. Mount `sdb1` on `/srv/node/sdb1` using the following command:

```
# mount /srv/node/sdb1
```

RSYNC and RSYNCD

In order for Swift to perform the replication of data, we need to configure `rsync` by configuring `rsyncd.conf`. This is done by performing the following steps:

1. Create the `rsyncd.conf` file in the `/etc` folder with the following content:

```
# vi /etc/rsyncd.conf
```

We are setting up synchronization within the network by including the following lines in the configuration file:

```
uid = swift  
gid = swift  
log file = /var/log/rsyncd.log  
pid file = /var/run/rsyncd.pid  
address = 172.168.9.52  
[account]  
max connections = 2  
path = /srv/node/  
read only = false  
lock file = /var/lock/account.lock  
[container]  
max connections = 2  
path = /srv/node/  
read only = false  
lock file = /var/lock/container.lock  
[object]  
max connections = 2  
path = /srv/node/  
read only = false  
lock file = /var/lock/object.lock
```

172.168.9.52 is the IP address that is on the replication network for this storage server. Use the appropriate replication network IP addresses for the corresponding storage servers.

2. We then have to edit the `/etc/default/rsync` file and set `RSYNC_ENABLE` to `true` using the following configuration option:

```
RSYNC_ENABLE=true
```

3. Next, we restart the `rsync` service using this command:

```
# service rsync restart
```

4. Then we create the `swift`, `recon`, and `cache` directories using the following commands, and then set its permissions:

```
# mkdir -p /var/cache/swift
# mkdir -p /var/swift/recon
```

5. Setting permissions is done using these commands:

```
# chown -R swift:swift /var/cache/swift
# chown -R swift:swift /var/swift/recon
```

Repeat these steps on every storage server.

Setting up the proxy server node

This section explains the steps required to set up the proxy server node, which are as follows:

1. Install the following services only on the proxy server node:

```
# apt-get install python-swiftclient python-keystoneclient
python-keystonemiddleware swift-proxy
```

2. Swift doesn't support HTTPS. OpenSSL has already been installed as part of the operating system installation to support HTTPS. We are going to use the OpenStack Keystone service for authentication. In order to set up the `proxy-server.conf` file for this, we download the configuration file from the following link and edit it:

```
https://raw.githubusercontent.com/openstack/swift/stable/juno/
etc/proxy-server.conf-sample
# vi /etc/swift/proxy-server.conf
```

3. The `proxy-server.conf` file should be edited to get the correct `auth_host`, `admin_token`, `admin_tenant_name`, `admin_user`, and `admin_password` values (refer to the following section about Keystone setup to see how to set up the correct credentials):

```
admin_token = 01d8b673-9ebb-41d2-968a-d2a85daa1324
admin_tenant_name = admin
admin_user = admin
admin_password = changeme
```

4. Next, we create a `keystone-signing` directory and give permissions to the `swift` user using the following commands:

```
# mkdir -p /home/swift/keystone-signing
# mkdir -R swift:swift /home/swift/keystone-signing
```

The Keystone service

We will be using OpenStack's identity service (Keystone) for authentication. The Keystone service exposes an endpoint that a user will connect to using username and password credentials, and the tenant (project) as the scope of the request. After validation by the Keystone identity service, a token that will be cached and used in further API calls to various other OpenStack API endpoints is returned to the user. Within Keystone, a user is defined to have account credentials and is associated with one or more tenants. Also, a user can be given a role such as admin, which entitles that user to more privileges than an ordinary user.

Let's consider the case where a user is connecting to a Swift endpoint to read an object. When a user initiates an API call along with a token to the Swift endpoint, this token is passed by the Swift endpoint back to Keystone for validation. Once validated by Keystone, it returns a success code to the Swift endpoint. The Swift service will then continue processing the API to read the object.

We will now show you the steps necessary to install and configure the Keystone service in the following sections.

Installing MariaDB

We will use MariaDB for the Keystone database. The installation steps are as follows:

1. Install the MariaDB database and client software on the proxy server node using the following command:

```
# apt-get install mariadb-server python-mysqldb
```

 The Python MySQL library is compatible with MariaDB.

2. Edit `/etc/mysql/my.cnf` in the proxy node, assigning the proxy server host name to `Bind-address`, as shown in this command:

```
Bind-address = swift-proxy
```

3. Restart the MySQL service on the proxy node using the following command:

```
# service mysql restart
```
4. Delete anonymous users using the `mysql_secure_installation` command, as follows:

```
# mysql_secure_installation
```
5. Respond with `yes` to delete the anonymous user prompt.

Installing Keystone

Keystone may be installed on dedicated servers for large installations, but for this example we'll install the Keystone service on the proxy node. The following steps describe how to install and set up the Keystone service:

1. Install the Keystone service using the following command:

```
# apt-get install keystone python-keystoneclient
```
2. We have to generate a random token to access the Keystone service, as shown in the following commands:

```
# python -c "from uuid import uuid4; print uuid4()"
# openssl rand -hex 10
```
3. We then edit the `/etc/keystone/keystone.conf` file to perform the following changes:
 - Replace `admin_token` with the random token that gets generated, as shown in this configuration option:

```
admin_token = 01d8b673-9ebb-41d2-968a-d2a85daa1324
```
 - Replace SQLite with a MariaDB database connection using the following configuration option under the database group:

```
connection = mysql://keystone:vedams123@swift-proxy/keystone
```
4. Make sure that the SQLite file has been deleted after configuring MariaDB. Otherwise, you'll need to manually delete the file. Run the following command to list the content of the `/var/lib/keystone` directory, and delete the `keystone.sqlite` file if present:

```
# ls -la /var/lib/keystone/
```

5. We then create the Keystone database user and grant permissions using the commands shown in this screenshot:

```
# mysql -u root -pvedams123
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 317
Server version: 5.5.43-0ubuntu0.14.04.1 (Ubuntu)

Copyright (c) 2000, 2015, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> CREATE DATABASE keystone;
Query OK, 1 row affected (0.07 sec)

mysql> GRANT ALL PRIVILEGES ON keystone.* TO 'keystone'@'localhost' IDENTIFIED BY 'vedams123';
Query OK, 0 row affected (0.00 sec)

mysql> GRANT ALL PRIVILEGES ON keystone.* TO 'keystone'@'%' IDENTIFIED BY 'vedams123';
Query OK, 0 row affected (0.00 sec)

mysql> quit
bye
#
```

6. Next, we check the Keystone database synchronization and restart the Keystone service using the following commands:

```
# keystone-manage db_sync
# service keystone restart
```
7. Export the following environment variables:

```
# export OS_SERVICE_TOKEN=01d8b673-9ebb-41d2-968a-d2a85daa1324
# export OS_SERVICE_ENDPOINT=http://swift-proxy:35357/v2.0
```
8. The service token and service endpoint are used to **bootstrap** keystone when no users exist, and we use the service token to create the initial user/tenant/roles roles and endpoints/services services. Once these have been created, the typical flow is to use the username and password combination for authentication. Once authenticated, access to Swift services and endpoints is permitted. We then create a tenant for an administrator user, an administrator user called admin, and a role for administrative tasks. Next, we add an admin role to the admin user. This is shown in the following screenshot:

```
# keystone tenant-create --name=admin --description="Admin Tenant"
# keystone user-create --name=admin --pass=changeme --email=test@example.com
# keystone role-create admin
# keystone user-role-add --user=admin --tenant=admin --role=admin
```

The following screenshot shows the output of executing the preceding commands:

```

root@swift-proxy:/home/vedams# keystone tenant-create --name=admin --description="Admin Tenant"
+-----+
| Property | Value |
+-----+
| description | Admin Tenant |
| enabled | True |
| id | f570de35b6dc4a4d81a24516d049173a |
| name | admin |
+-----+
root@swift-proxy:/home/vedams# keystone user-create --name=admin --pass=vedams123 --email=test@gmail.com
+-----+
| Property | Value |
+-----+
| email | test@gmail.com |
| enabled | True |
| id | 77461f6a3763462b990cdaceec034afe |
| name | admin |
+-----+
root@swift-proxy:/home/vedams# keystone role-create --name=admin
+-----+
| Property | Value |
+-----+
| id | 814ffecf0bbc4221a9ab98618d159ded |
| name | admin |
+-----+
root@swift-proxy:/home/vedams# keystone user-role-add --user=admin --tenant=admin --role=admin
root@swift-proxy:/home/vedams#

```

9. Next, we will create another user called `swift-user` and add it to the tenant called `swift-tenant`. The user is given the member access role. The following screenshot shows the process of creation:

```

root@swift-proxy:/home/vedams# keystone tenant-create --name=swift-tenant --description="Swift Tenant"
+-----+
| Property | Value |
+-----+
| description | Swift Tenant |
| enabled | True |
| id | bd1e87f876e541a4acc42803430a1b2b |
| name | swift-tenant |
+-----+
root@swift-proxy:/home/vedams# keystone user-create --name=swift-user --pass=vedams123 --email=swiftuser@gmail.com
+-----+
| Property | Value |
+-----+
| email | swiftuser@gmail.com |
| enabled | True |
| id | 0b81ddf04865444bbbdd4be417a392fc |
| name | swift-user |
+-----+
root@swift-proxy:/home/vedams# keystone user-role-add --user=swift-user --tenant=swift-tenant --role=_member_
root@swift-proxy:/home/vedams#

```

10. The Keystone service keeps track of the various OpenStack services that we have installed and where they are in the network. In order to keep track of the services, IDs are created for them using the `keystone service-create` command, as shown in this screenshot:

```

# keystone service-create --name=keystone --type=identity \
--description="Keystone Identity Service"
# keystone service-create --name=swift --type=object-store \
--description="swift Service"

```


The following screenshot shows the output of executing the preceding service-create commands:

```
root@swift-proxy:/home/vedams# keystone service-create --name=keystone --type=identity --description="Keystone Identity Service"
+-----+
| Property | Value |
+-----+
| description | Keystone Identity Service |
| id | a9c2d44442464975bb50e296fcc584b4 |
| name | keystone |
| type | identity |
+-----+
root@swift-proxy:/home/vedams# keystone service-create --name=swift --type=object-store --description="Swift Object storage service"
+-----+
| Property | Value |
+-----+
| description | Swift Object storage service |
| id | a0ab378728b148fd9c9a0534d1d6a227 |
| name | swift |
| type | object-store |
+-----+
root@swift-proxy:/home/vedams#
```

11. We then need to specify the Keystone service endpoints and Swift service endpoints to Keystone using the endpoint-create command. In the following commands, swift-proxy is the hostname of the proxy server:

```
# keystone endpoint-create \
--service-id $(keystone service-list | awk '/ identity / {print $2}') \
--publicurl http://swift-proxy:5000/v2.0 \
--internalurl http://swift-proxy:5000/v2.0 \
--adminurl http://swift-proxy:35357/v2.0 \
--region regionOne
# keystone endpoint-create \
--service-id $(keystone service-list | awk '/ object-store / {print $2}') \
--publicurl 'http://swift-proxy:8080/v1/AUTH_$(tenant_id)s' \
--internalurl 'http://swift-proxy:8080/v1/AUTH_$(tenant_id)s' \
--adminurl http://swift-proxy:8080 \
--region regionOne
```

The following screenshot shows the output of executing the endpoint-create commands:

```
root@swift-proxy:/home/vedams# keystone endpoint-create --service-id $(keystone service-list | awk '/ identity / {print $2}') --publicurl http://swift-proxy:5000/v2.0 --internalurl http://swift-proxy:5000/v2.0 --adminurl http://swift-proxy:35357/v2.0 --region regionOne
+-----+
| Property | Value |
+-----+
| adminurl | http://swift-proxy:35357/v2.0 |
| id | b01b9f70267c4a5faf7f1108cc228d30 |
| internalurl | http://swift-proxy:5000/v2.0 |
| publicurl | http://swift-proxy:5000/v2.0 |
| region | regionOne |
| service_id | da875a654c42416dad400dfbe3321eece |
+-----+
root@swift-proxy:/home/vedams# keystone endpoint-create --service-id $(keystone service-list | awk '/ object-store / {print $2}') --publicurl 'http://swift-proxy:8080/v1/AUTH_$(tenant_id)s' --internalurl 'http://swift-proxy:8080/v1/AUTH_$(tenant_id)s' --adminurl http://swift-proxy:8080 --region regionOne
+-----+
| Property | Value |
+-----+
| adminurl | http://swift-proxy:8080 |
| id | 312e66fc6b5b688aaf5041b78a1b8c |
| internalurl | http://swift-proxy:8080/v1/AUTH_$(tenant_id)s |
| publicurl | http://swift-proxy:8080/v1/AUTH_$(tenant_id)s |
| region | regionOne |
| service_id | c7170057e3404e7982adab6b9ab6a8c |
+-----+
```

12. We will now unset the environment variables that we exported earlier, since we won't need them again. We will be calling the REST APIs and providing the username and password to them along with the endpoint. Unset the environment variables using the following commands:

```
# unset OS_SERVICE_TOKEN
# unset OS_SERVICE_ENDPOINT
```

13. Then we will request an authentication token using the admin user and password. This verifies that the Keystone service is configured and running correctly on the configured endpoint.

We also verify that authentication is working correctly by requesting the token on a particular tenant, as shown in the following command:

```
# keystone --os-username=admin --os-password=changeme --os-tenant-name=admin --os-auth-url=http://swift-proxy:35357/v2.0 token-get
```

14. Finally, test the Keystone service by running the following commands to list the users, tenants, roles, and endpoints:

```
# keystone --os-username=admin --os-password=changeme --os-tenant-name=admin --os-auth-url=http://swift-proxy:35357/v2.0 user-list
# keystone --os-username=admin --os-password=changeme --os-tenant-name=admin --os-auth-url=http://swift-proxy:35357/v2.0 tenant-list
# keystone --os-username=admin --os-password=changeme --os-tenant-name=admin --os-auth-url=http://swift-proxy:35357/v2.0 role-list
# keystone --os-username=admin --os-password=changeme --os-tenant-name=admin --os-auth-url=http://swift-proxy:35357/v2.0 endpoint-list
```

The ring setup

As discussed in *Chapter 2, OpenStack Swift Architecture*, the ring (also called the ring builder, or simply the builder file) contains information required to map the user API request information on the physical location of the account, container, or object. We will have a builder file for accounts, which will contain mapping information for the account. Similarly, we will have a builder file for containers and objects.

Builder files are created using the commands shown in the following screenshot:

```
# cd /etc/swift
# swift-ring-builder account.builder create 18 3 1
# swift-ring-builder container.builder create 18 3 1
# swift-ring-builder object.builder create 18 3 1
```

The 18 parameter indicates that there can be 2 raised to 18 partitions created to store the data. To determine the number of partitions, estimate the maximum number of disks, multiply that number by 100, and then round it off to the nearest power of 2. Picking a number smaller than needed is not catastrophic, it will just result in an unbalanced cluster from a storage capacity point of view. Picking a number larger than needed will impact performance. The 3 parameter indicates that three replicas of data will be stored, and the 1 parameter is set in such a way that we don't move a partition more than once in an hour.

In Swift storage, hard disks are the smallest unit in a ring. They are part of nodes, which are grouped into zones, and rings can be set up according to zones. Zones (and regions) are used to group multiple hosts or nodes into failure domains. Each hard disk in a storage server belongs to a particular zone. This helps Swift replicate the data to different zones in an as-unique-as-possible manner. If there is a failure in a particular zone, data can be fetched from the data copies in other zones. In a multi-region setup, if there is a failure in a particular region, then data can be fetched from other regions.

The following command syntax is used to add storage server hard disk devices to ring builder files. Note that the region and zone the hard disk belongs to are provided as an input parameter. The weight parameter (100) indicates how much data is going to be placed on this disk compared to other disks.

Run the following commands to add the hard disks allocated for storage to the ring. In order to add mapping for the sdb1 device, we run the commands shown in the following screenshot:

```
# swift-ring-builder account.builder add r1z1-172.168.10.52:6002\
R172.168.9.52:6005/sdb1 100
# swift-ring-builder container.builder add r1z1-172.168.10.52:60\
01R172.168.9.52:6004/sdb1 100
# swift-ring-builder object.builder add r1z1-172.168.10.52:6000R\
172.168.9.52:6003/sdb1 100
```

The final step in completing the ring builder process involves creating the ring files that will be used by the Swift processes. This is done using the `rebalance` command, as shown here:

```
# swift-ring-builder account.builder rebalance
# swift-ring-builder container.builder rebalance
# swift-ring-builder object.builder rebalance
```

Upon running the preceding commands, the following files will be created: `account.ring.gz`, `container.ring.gz`, and `object.ring.gz`. Copy these files to the `/etc/swift` directories of all the nodes in the cluster.

We should also restart the `rsyslog` and `memcached` services on the storage servers using the following commands:

```
# service rsyslog restart
# service memcached restart
```

Multiregion support

In a multiregion installation, we place a pool of storage nodes in one region and the remaining in other regions. We can either have a single endpoint for all the regions or a separate endpoint for each region. During the ring builder setup, the region is specified as a parameter. Clients can access any endpoint and perform operations (create, delete, and so on), and they will be replicated across other regions. The proxy server configuration files will contain `read_affinity` and `write_affinity` in a particular region.

Our test configuration had two proxy servers and five storage nodes. Two regions were created by creating two endpoints. A list of the endpoints gives the following output, where `Region1` and `Region2` are the two regions:

```
# keystone endpoint-list
```

id	region	publicurl	
internalurl		adminurl	
service_id			
72c4904fd859403283be7eae5c05245	RegionOne	http://192.168.2.230:5000/v2.0	
http://192.168.2.230:5000/v2.0		http://192.168.2.230:35357/v2.0	e8607a
aea2b54b9d81fbef02af003f54			
7c85bfb57870420fbfadfb934a2565fa	Region1	http://proxy1:8080/v1/AUTH_\$(tenant_id)s	h
http://proxy1:8080/v1/AUTH_\$(tenant_id)s		http://proxy1:8080/	19177f
2c544d4f68bb30f3d967666456			
df4e49fde60d4a4bb8d2133bc2620e8e	Region2	http://proxy2:8080/v1/AUTH_\$(tenant_id)s	h
http://proxy2:8080/v1/AUTH_\$(tenant_id)s		http://proxy2:8080/	399527
d757424da48769bc0eeb5ee1da			

In the preceding output, `Region1` and `Region2` are the two regions.

Finalizing the installation

Let's finalize the installation by ensuring proper ownership and restarting services. This is done by running the following commands:

- Ensure proper ownership of the configuration directory on all nodes:
`# chown -R swift:swift /etc/swift`
- Restart the services on the proxy node:
`# service memcached restart`
`# service swift-proxy restart`
- Start all the services on the storage nodes:
`# swift-init all start`

Storage policies

Storage policies allow the user to create multiple object rings and use them for different purposes based on the level of importance of the data. Some factors that will be considered while implementing storage policies are the number of replicas, performance required from the ring, disk affinity for the ring, and so on.

We are going to explain the concept of storage policies using an example of bank documents. Some documents are critical for the operation of a bank, and others are not so critical. We can use a storage policy with a higher number of replicas to store the critical documents, database, and so on.

We can use SSD disks in the object ring to store documents that need to be retrieved without any delay (performance-oriented policy). This ring can have two replicas if needed.

The capabilities of storage policies make the Swift cluster more adaptable to customer requirements.

Implementing storage policies

Storage policies were introduced in the Juno release of OpenStack Swift. There is a default policy called `policy-0`. It is set up by default, as shown in the following screenshot:

```
[swift-hash]
# random unique strings that can never change (DO NOT LOSE)
swift_hash_path_suffix = 8de2672f68239cd78ea8
swift_hash_path_prefix = da75a2eda3ca4aefb2e0

#storage policy:0
[storage-policy:0]
name = policy-0
default = yes
```

Now we will implement two new policies called `bank` (for critical documents), which will contain three replicas, and `regular` (for storing regular, noncritical documents), which will contain two replicas. For this, perform the following steps:

1. Edit the `swift.conf` file and change the name `policy-0` to `bank`. Create a new storage policy called `policy-1` and name it `regular`. The `swift.conf` file should look like what is shown in the following screenshot:

```
[storage-policy:0]
name = bank
default = yes

[storage-policy:1]
name = regular
```

We should replace the `swift.conf` file with the preceding content in all the proxy nodes.

2. The next step is to create the object-ring file specific to the new policy (with two replicas). This is done using the following command:

```
swift-ring-builder object-1.builder create 10 2 1
```

This contains `object-1.builder`, which contains the policy for creating two replicas instead of three replicas.

3. Now let's create the ring files to implement the new policy. This is done by running the following `swift-ring-builder` commands:

```
# swift-ring-builder object-1.builder add r1z1-
172.168.10.52:6000/sdb1
# swift-ring-builder object-1.builder add r1z1-
172.168.10.53:6000/sdb1
```

- Next, we will rebalance to create object-1.ring.gz. We should then copy this file to all proxy nodes and storage nodes:

```
# swift-ring-builder object-1.builder rebalance
```



Do not delete the existing object.ring.gz ring file.

- We then need to create a file called container-reconciler.conf in the proxy node in the /etc/swift folder. The contents of the file are shown in this screenshot:

```
[DEFAULT]
# swift_dir = /etc/swift
# user = swift
# You can specify default log routing here if you want:
# log_name = swift
# log_facility = LOG_LOCAL0
# log_level = INFO
# log_address = /dev/log
# comma separated list of functions to call to setup custom log handlers.
# functions get passed: conf, name, log_to_console, log_route, fmt, logger,
# adapted_logger
# log_custom_handlers =
# If set, log_udp_host will override log_address
# log_udp_host =
# log_udp_port = 514
# You can enable StatsD logging here:
# log_statsd_host = localhost
# log_statsd_port = 8125
# log_statsd_default_sample_rate = 1.0
# log_statsd_sample_rate_factor = 1.0
# log_statsd_metric_prefix =

[container-reconciler]
# reclaim_age = 604800
# The cycle time of the daemon
# interval = 30
# Server errors from requests will be retried by default
# request_tries = 3

[pipeline:main]
pipeline = catch_errors proxy-logging cache proxy-server

[app:proxy-server]
use = egg:swift#proxy
# See proxy-server.conf-sample for options

[filter:cache]
use = egg:swift#memcache
# See proxy-server.conf-sample for options

[filter:proxy-logging]
use = egg:swift#proxy_logging

[filter:catch_errors]
use = egg:swift#catch_errors
# See proxy-server.conf-sample for options
```

6. Next, we run the following command to fix the user option:

```
$ sed -i "s/# user.*/user=$USER/g" /etc/swift/container-reconciler.conf
```

Applying storage policies

We will now show you some examples of creating containers and uploading objects using storage policies. In order to verify that objects are being created using the desired policies, we enable `list_endpoints` in the `proxy-server.conf` file by modifying the pipeline section, as follows:

```
[pipeline:main]
pipeline = healthcheck list-endpoints cache authtoken proxy-server
```

We now have to restart all the proxy and storage node services.

In order to create a container with a desired policy, we use this command:

```
swift post <container name> -H "X-Storage-Policy: <policy name>"
```

The following is an example of creating a container with the `regular` policy:

```
# swift post mycontainer -H "X-Storage-Policy: regular"
```

In order to add objects to the container, we use the following command:

```
swift upload <<container name>> <<filename>> -H "X-Storage-Policy:
<<policy name>>"
```

Here is an example of uploading an object to the previously created container:

```
# swift upload mycontainer myfile.txt -H "X-Storage-Policy: regular"
```

We use the `stat` command to verify that the container and objects have been created with the `regular` policy. The output of the `stat` command is as follows:

```
# swift stat
Account: AUTH_8e43ee201cbcb70bd8bb2f8ae10f025
Containers: 1
Objects: 1
Bytes: 8
Objects in policy "regular": 1
Bytes in policy "regular": 8
Objects in policy "bank": 0
Bytes in Policy "bank": 0
Content-Type: text/plain; charset=utf-8
X-Timestamp: 1419319875.96594
X-Trans-Id: tx6d690b72f2a1480bb1d71-0054b8b02c
Accept-Ranges: bytes
```


To use the default policy while creating containers and uploading objects, we don't need to specify the `Storage-Policy` metadata in the command.

The following is an example of creating a container and uploading an object using the bank default policy.

```
# swift post mycontainer_1
# swift post mycontainer_1 myfile.txt
```

We use the `stat` command to verify that the container and objects have been created with the bank policy. The output of the `stat` command is shown here:

```
# swift stat
Account: AUTH_8e43ee201cbc4b70bd8bb2f8ae10f025
Containers: 3
Objects: 2
Bytes: 16
Objects in policy "regular": 1
Bytes in policy "regular": 8
Objects in policy "bank": 1
Bytes in policy "bank": 8
Content-Type: text/plain; charset=utf-8
X-Timestamp: 1419319875.96594
X-Trans-Id: tx6d690b72f2a1480bb1d71-0054b8b02c
Accept-Ranges: bytes
```

This section should have given you a good overview of storage policies that will help you create your own storage policies in your environments.

Summary

In this chapter, you learned how to install and set up the OpenStack Swift service to provide object storage, and install and set up the Keystone service to provide authentication for users to access the Swift object storage. You also got an overview on storage policies.

The next chapter provides details on various tools, commands, and APIs that are available for accessing and using the Swift object storage.

4

Using Swift

This chapter explains the various mechanisms that are available for accessing Swift. Using these mechanisms, we will be able to authenticate accounts, list containers, create containers, upload objects, delete objects, and so on.

Tools and libraries such as the Swift client CLI, cURL client, HTTP REST API, JAVA libraries, Ruby OpenStack libraries, and Python libraries use Swift APIs internally to provide access to the Swift cluster. In particular, we will be using the Swift client CLI, cURL, and HTTP REST API to access Swift and perform various operations on containers and objects. Also, we will be using the Vedams internal Swift cluster cloud storage setup to demonstrate the use of Swift.

Installing clients

This section talks about installing cURL and Swift's client CLI command-line tools. In this section, we describe how to install these tools on a 64-bit Ubuntu 14.04 LTS server Linux operating system. Refer to the other Linux distribution command sets to install the clients on those operating systems.

The following commands are used to install cURL and the Swift client CLI:

- **cURL:** This is a command-line tool that can be used to transfer data using various protocols. The following command is used to install cURL:
`# apt-get install curl`
- **Swift client CLI:** This is a tool used to access and perform operations on a Swift cluster. It is installed using the following command:
`# apt-get install python-swiftclient`
- **Specialized REST API client:** To access Swift services via the REST API, we can use third-party tools, such as Fiddler web debugger, that support REST's architecture.

Creating a token using Keystone authentication

The first step in order to access containers or objects is to authenticate the user by sending a request to the authentication service, and thus get a valid token that can then be used in subsequent commands to perform various operations.

We are using Keystone authentication in our configuration and the examples shown in this chapter. There is another method of authentication called **Swauth** that can be used. It works in a slightly different way, but we won't deal with the details of Swauth here. While using cURL, the following command is used to get the valid Keystone authentication token:

```
# curl -X POST -i https://auth.vedams.com/v2.0/tokens -H 'Content-type: application/json' -d '{"auth":{"passwordCredentials":{"username":"user","password":"password"},"tenantName":"tenant1"}}'
```

In the preceding command, `https://auth.vedams.com/v2.0` is the Vedams Keystone authentication endpoint. Along with this, the username, password, and tenant name are also provided.

The token that is generated is shown as follows (it has been truncated for better readability):

```
token = 0a06c208c7a9479b9e21994fe3492802
```

This token is then used as a parameter in the commands that access Swift, for example, in the following command:

```
curl -X HEAD -i http://storage.vedams.com/v1/AUTH_8e43ee201cbc4b70bd8bb2f8ae10f025 -H 'X-Auth-Token: token' -H 'Content-type: application/json'
```

More details on the commands are provided in the upcoming sections.

Displaying metadata information for an account, container, or object

This section describes how we can obtain information about the account, container, or object.

Using the Swift client CLI

The Swift client CLI's `stat` command is used to get information about the account, container, or object. The name of the container should be provided after the `stat` command to get the container information. The name of the container and object should be provided after the `stat` command to get the object information.

Execute the following request to display the account status:

```
# swift --os-auth-token=token --os-storage-url=  
http://storage.vedams.com/v1/AUTH_8e43ee201cbc4b70bd8bb2f8ae10f025  
stat
```

In the preceding commands, `token` is the generated token, as described in the previous section, and `AUTH_8e43ee201cbc4b70bd8bb2f8ae10f025` is the account name. We can use environment variables to initialize the `token` and `storage-url`, and then we will not need to specify the `os-auth-token` and `os-storage-url` parameters in the preceding command. More details are provided in the *Environment variables* section of this chapter.

The response shows some information about the account, which is as follows:

```
Account: AUTH_8e43ee201cbc4b70bd8bb2f8ae10f025  
Containers: 5  
Objects: 6  
Bytes: 46957619  
Objects in policy "regular": 2  
Bytes in policy "regular": 46953998  
Objects in policy "bank": 4  
Bytes in policy "bank": 3621  
Content-Type: text/plain; charset=utf-8  
X-Timestamp: 1419319875.96594  
X-Trans-Id: tx85af67b7302547f38d411-0054b8a301  
Accept-Ranges: bytes
```

Using cURL

The following command shows us how to obtain the same account information using cURL. It shows that the account contains two containers and six objects. The AUTH_8e43ee201cbc4b70bd8bb2f8ae10f025 value is obtained from the output of the previous step.

Execute the following request:

```
-H 'X-Auth-Token: token' -H 'Content-type: application/json'
# curl -X HEAD -i
http://storage.vedams.com/v1/AUTH_8e43ee201cbc4b70bd8bb2f8ae10f025
-H 'X-Auth-Token: token' -H 'Content-type: application/json'
```

The response to the preceding command is as follows:

```
HTTP/1.1 204 No Content
Content-Length: 0
Content-Type: text/plain; charset=utf-8
X-Account-Object-Count: 6
X-Account-Storage-Policy-Regular-Object-Count: 2
X-Account-Storage-Policy-Bank-Object-Count: 4
X-Timestamp: 1419319875.96594
X-Account-Storage-Policy-Bank-Container-Count: 4
X-Account-Storage-Policy-Regular-Bytes-Used: 46953998
X-Account-Bytes-Used: 46957619
X-Account-Container-Count: 5
X-Account-Storage-Policy-Bank-Bytes-Used: 3621
Accept-Ranges: bytes
X-Account-Storage-Policy-Regular-Container-Count: 1
X-Trans-Id: txc018fe80597f4d1d8095a-0054b8a1c5
Date: Fri, 16 Jan 2015 05:29:41 GMT
```

Using the specialized REST API client

Fiddler web debugger, which supports REST, was used to send the request and receive the HTTP response. Execute the following request:

```
Method : HEAD

URL :
http://storage.vedams.com/v1/AUTH_8e43ee201cbc4b70bd8bb2f8ae10f025
Header : X-Auth-Token: token
Data : No data
```

```
HTTP/1.1 204 No Content
Content-Length: 0
Content-Type: text/plain; charset=utf-8
X-Account-Object-Count: 6
X-Account-Storage-Policy-Regular-Object-Count: 2
X-Account-Storage-Policy-Bank-Object-Count: 4
X-Timestamp: 1419319875.96594
X-Account-Storage-Policy-Bank-Container-Count: 4
X-Account-Storage-Policy-Regular-Bytes-Used: 46953998
X-Account-Bytes-Used: 46957619
X-Account-Container-Count: 5
X-Account-Storage-Policy-Bank-Bytes-Used: 3621
Accept-Ranges: bytes
X-Account-Storage-Policy-Regular-Container-Count: 1
X-Trans-Id: tx81b400450db24987af92e-0054b8f39d
Date: Fri, 16 Jan 2015 11:18:53 GMT
```

As you can see, this is a different mechanism of issuing the command, but is very similar to accessing the Swift cluster using cURL.

Listing containers

This section describes how to obtain information about the containers present in an account.

Using the Swift client CLI

Execute the following request:

```
swift --os-auth-token=token --os-storage-url=
http://storage.vedams.com/v1/AUTH_8e43ee201cbc4b70bd8bb2f8ae10f025
list
```

The response is as follows:

```
cities
countries
```

Using cURL

The following command shows you how to obtain the same container information using cURL. It shows that the account comprises of two containers and six objects.

Execute this request:

```
curl -X GET -i
http://storage.vedams.com/v1/AUTH_8e43ee201cbc4b70bd8bb2f8ae10f025 -H
'X-Auth_token: token'
```

The response to the request is as follows:

```
HTTP/1.1 200 OK
X-Account-Container-Count: 2
X-Account-Object-Count: 6

cities
countries
```

Here, we see that the output has a header and a body, whereas in the previous example, we only had a header and no body in the output.

Listing objects in a container

This section describes how to list objects that are present in a container.

Using the Swift client CLI

The following command shows you how to list objects using the Swift client CLI (in this example, we are listing the objects in the `cities` container):

Execute the following request:

```
swift --os-auth-token=token --os-storage-url=
http://storage.vedams.com/v1/AUTH_8e43ee201cbc4b70bd8bb2f8ae10f025
list cities
```

Here is the response to the request:

```
London.txt
Mumbai.txt
NewYork.txt
```

Using cURL

The following command shows you how to list objects using cURL. In this example, we list the objects in the `cities` container. Execute this request:

```
curl -X GET -i
http://storage.vedams.com/v1/AUTH_8e43ee201cbc4b70bd8bb2f8ae10f025/ci
ties
-H 'X-Auth-Token: token '
```

The response is as follows:

```
HTTP/1.1 200 OK
Content-Type: text/plain; charset=utf-8
Content-Length: 34
X-Container-Object-Count: 3

London.txt
Mumbai.txt
NewYork.txt
```

Using the REST API

In this example, we will list the objects in the `countries` container. Execute this request:

```
Method : GET
URL      :
http://storage.vedams.com/v1/AUTH_8e43ee201cbc4b70bd8bb2f8ae10f025/co
untries
Header   : X-Auth-Token: token
Data     : No content
```

The response to the request is as follows:

```
HTTP/1.1 200 OK
Content-Type: text/plain; charset=utf-8
Content-Length: 38
X-Container-Object-Count: 3

France.txt
India.txt
UnitedStates.txt
```


Updating the metadata for a container

This section describes how to add or update metadata for a container. Examples showing how to update X-Container-Meta-Countries are covered in the following sections.

Using the Swift client CLI

In this example, we are adding metadata for the countries that we have visited. Execute the following request:

```
-H "X-Container-Meta-Countries: visited"
swift --os-auth-token=token --os-storage-url=
http://storage.vedams.com/v1/AUTH_8e43ee201cbc4b70bd8bb2f8ae10f025
post countries
-H "X-Container-Meta-Countries: visited"
```

Using the REST API

Here we are adding metadata using the REST API.

Execute this request:

Method : POST

URL :
http://storage.vedams.com/v1/AUTH_8e43ee201cbc4b70bd8bb2f8ae10f025/countries

Header : X-Auth-Token: token
X-Container-Meta-Countries: visited

Data : No content

Environment variables

The following environment variables can be used to simplify the Swift CLI commands:

- OS_USERNAME: This contains the username used to access the account
- OS_PASSWORD: This contains the password associated with the username
- OS_TENANT_NAME: This contains the name of the tenant
- OS_AUTH_URL: This contains the authentication URL

Once these environment variables are exported, we no longer have to pass these values as input parameters when running the Swift CLI tools. The following is an example of setting up an environment variable (`OS_USERNAME`):

```
# export OS_USERNAME=admin
```

The pseudo-hierarchical directories

In OpenStack Swift, object storage can simulate a hierarchical directory structure in containers by including a / (forward slash) character in the object's name.

Let's upload a file (`AMERICA/USA/Newyork.txt`) to the `Continent` container using the following command:

```
# swift upload Continent AMERICA/USA/Newyork.txt
```

Now let's list the `Continent` container, which has a few pseudo-hierarchical folders, using the following commands:

```
# swift list Continent
AMERICA/USA/Newyork.txt
ASIA/ASIA.txt
ASIA/China/China.txt
ASIA/INDIA/India.txt
Australia/Australia.txt
continent.txt
```

We can use / as the delimiter parameter to limit the displayed results. We can also use the prefix parameter along with the delimiter parameter to view the objects in the pseudo-directory along with the pseudo-directories within it. The following are a couple of examples showing the use of these parameters:

```
# swift list Continent --delimiter /
AMERICA/
ASIA/
Australia/
continent.txt

# swift list Continent --delimiter / --prefix ASIA/
ASIA/ASIA.txt
ASIA/China/
ASIA/INDIA/

# swift list Continent --delimiter / --prefix ASIA/INDIA/
ASIA/INDIA/India.txt
```

Container ACLs

As we saw in the previous sections, in order to access containers and objects, a valid auth token has to be sent in the X-Auth-Token header with each request. Otherwise, an authorization failure code will be returned. In certain cases, access needs to be provided to other clients and applications for certain containers. Access can be provided by setting a metadata element for the container, called X-Container-Read. The following paragraphs cover the setting of this **Access Control List (ACL)** for the `cities` container.

First, let's list the container status that shows the lack of ACL. Run the following command with admin privileges (the admin user will have the permissions to run this command):

```
swift stat cities
```

The values of `Read ACL` and `Write ACL` in the following response indicate the lack of ACL:

```
Account: AUTH_8e43ee201cbc4b70bd8bb2f8ae10f025
Container: cities
Objects: 3
Read ACL:
Write ACL:
Sync To:
```

When the `tenant1:user1` user, who does not have access to this container, tries to access it, a forbidden error message is returned.

Execute this request:

```
swift -V 2.0 -A https://auth.vedams.com/v2.0 -U tenant1:user1 -K t1
list cities
```

A forbidden error message is returned as the response. The error is as follows:

```
Container GET failed: 403 Forbidden
Access was denied to this resource
```

In the preceding example, the username is provided using the `-U` option, and the key required to access the account is provided using the `-K` option.

Now, let's set the X-Container-Read metadata element and enable the `READ` access for `tenant1:user1`. This operation can only be done by the admin user, using the following command:

```
swift post -r tenant1:user1 cities
```

To check the ACL permissions, we execute this command:

```
swift stat cities
```

The response to the preceding command is as follows:

```
Account: AUTH_8e43ee201cbc4b70bd8bb2f8ae10f025
Container: cities
Objects: 3
Read ACL: tenant1:user1
Write ACL:
Sync To:
```

Now, when the `tenant1:user1` user tries to access this container, access is granted and the command is successfully executed.

Execute the following request:

```
swift -V 2.0 -A https://auth.vedams.com/v2.0 -U tenant1:user1 -K t1
list cities
```

Here is the response to the request:

```
London.txt
Mumbai.txt
NewYork.txt
```

Since the X-Container-Write ACL is not set for the `tenant1:user1` user for the `cities` container, this user cannot write to the `cities` container. In order to allow write access, let's set the X-Container-Write ACL, as follows:

```
swift post -w tenant1:user1 cities
```

To check the ACL permissions, we execute the following command:

```
swift stat cities
```

The response to the preceding command is this:

```
Account: AUTH_8e43ee201cbc4b70bd8bb2f8ae10f025
Container: cities
Objects: 3
Read ACL: tenant1:user1
Write ACL: tenant1:user1
Sync To:
```

Now the `tenant1:user1` user will be able to write objects to the `cities` container.

If we want to give access to a large number of users, ACLs such as `.r:*` and `.rlistings` can be used. The `.r:*` prefix allows any user to retrieve objects from the container, and `.rlistings` turns on listing for the container.

Transferring large objects

As discussed in *Chapter 2, OpenStack Swift Architecture*, Swift limits a single object upload to 5 GB. Larger objects can be split into 5 GB or smaller segments by specifying the segment size option in the Swift CLI tool command-line argument, and uploaded to a special container (created within the container where the object is being uploaded).

Once the upload has been completed, a manifest object that contains information about the segments has to be created. The manifest file is of zero size, with headers such as `X-Object-Manifest` identifying the special container in which the segments are stored and the name with which all the segments will start. For example, if we have to upload `France.txt` (which is of size 8 GB) to the `countries` container, then the `France.txt` object has to be split into two chunks (5 GB and 3 GB). The chunk objects' names will start with `France.txt` (`France.txt/./00000000` and `France.txt/./00000001`).

If the Swift CLI is being used, a special container called `countries_segments` will be created and the chunks will be uploaded to this container. A manifest object called `France.txt` will be created in the `countries` container. The manifest file will have zero size and will contain the following header (it is not mandatory to have the segments placed in a special container; they might as well exist in the same container):

```
X-Object-Manifest: countries_segments/France.txt
```

When a download request is made for the large object, Swift will automatically concatenate all the segments and download the entire object.

The Swift client CLI has the `-s` flag. It is used to specify the segment size in bytes, which can be used to split a large object into segments and upload. The following command is used to upload a file with a segment size of 5368709120 bytes (5.3 GB):

```
swift upload countries -segment-size 5368709120 France.txt
```

The response to the preceding command is this:

```
France.txt segment 0
France.txt segment 1
```

```
France.txt segment 2
France.txt
```

The following command can be used to list the containers present:

```
swift list
```

The response is as follows:

```
Countries
Countries_segments
cities
```

This command lists the objects in the `countries_segments` container:

```
swift list countries_segments
```

The response to the preceding command is this:

```
France.txt/1385989364.105938/5368709120/00000000
France.txt/1385989364.105938/5368709120/00000001
```

Amazon S3 API compatibility

Users familiar with the Amazon S3 API and accessing S3 buckets and objects can access Swift using S3-compatible APIs with the help of Swift3 middleware.

Here, we will show you the steps required for a method that uses S3 APIs to access Swift's object store. These steps explain how to install the necessary tools and packages, create credentials, and update the configuration files.

The following steps are to be performed on the proxy server node that is running the Ubuntu 14.04 Linux distribution:

1. First, the user requires EC2 credentials (access key and secret key). The `keystone user-list` and `keystone tenant-list` commands can be used to obtain the user ID and tenant ID of the user. The following command can be used to create these keys (these need to be run from the proxy server):

```
keystone ec2-credentials-create --user-id
916673a90b8749e18f0ee3ec5bf17ab9 --tenant-id
6530edfe037242d1ac8bb07b7fd76046
```

The response is as follows:

```
+-----+-----+
| Property | Value |
+-----+-----+
```

	access		1178d235dbd84d48b417170ec9aed72c	
	secret		c4ea0a8fbf7d4a469f6d0fb5cdb47d5b	
	tenant_id		6530edfe037242d1ac8bb07b7fd76046	
	user_id		916673a90b8749e18f0ee3ec5bf17ab9	

2. Install the Swift3 package by running the following commands (these commands require Git to be installed on your system):

```
# sudo git clone https://github.com/fujita/swift3.git
# cd swift3
# python setup.py install
```
3. Install the libdigest-hmac-perl package (used for integrity checking between two entities that share a secret key) by running this command:

```
apt-get install libdigest-hmac-perl
```
4. Edit the `proxy-server.conf` file and make the following changes if you want to use the keystone authentication:
 - Change the pipeline line in the `proxy-server.conf` file to this:

```
[pipeline:main]
pipeline = catch_errors cache swift3 s3token authtoken
keystone proxy-server
```
 - Add a Swift3 WSGI filter to the `proxy-server.conf` file using the following command:

```
[filter:swift3]
use = egg:swift3#swift3
```
 - Add the `s3token` filter as shown in these commands:

```
[filter:s3token]
paste.filter_factory = keystone.middleware.s3_token:filter_
factory
auth_port = 35357
auth_host = 127.0.0.1
auth_protocol = http
```
 - Restart the proxy service:

```
Service swift-proxy restart
```
5. The following steps should be performed on the client that will access Swift object storage:
 - Since we will use `s3curl` to execute the S3 commands, download `s3-curl.zip` from <http://s3.amazonaws.com/doc/s3-example-code/s3-curl.zip>.

- Install the `wget` utility prior to running the following command:
`wget http://s3.amazonaws.com/doc/s3-example-code/s3-curl.zip`
- Unzip `s3-curl.zip` and provide executable access to the `s3curl.pl` file.
- Create a `.s3curl` file. Change the ID and key of the personal account with the EC2 credentials (access and secret keys) that were given to the user. We are using the `vi` editor to create the file, as shown in the following code:

```
#vi ~/.s3curl
%awsSecretAccessKeys = (
# personal account
  personal => {
    id => '1178d235dbd84d48b417170ec9aed72c',
    key => 'c4ea0a8fbf7d4a469f6d0fb5cdb47d5b',
  },
);
```

Accessing Swift using S3 commands

In this section, we will give examples of the S3 commands used to perform various operations.

- **List buckets:** This command lists all the buckets for the particular user. Buckets in S3 are similar to containers in Swift:

```
# ./s3curl.pl --id=personal -- https://auth.vedams.com -v
```

The response is as follows:

```
<?xml version="1.0" encoding="UTF-
8"?><ListAllMyBucketsResult xmlns="http://doc.s3.amazonaws.
com/2006-03-01"><Buckets>
  <Bucket><Name>cities</Name><CreationDate>2009-02-
03T16:45:09.000Z</CreationDate></Bucket>
  <Bucket><Name>countries</Name><CreationDate>2009-02-
03T16:45:09.000Z</CreationDate></Bucket>
</Buckets></ListAllMyBucketsResult>
```

- **List objects in a bucket:** This command lists all the objects present in the specified bucket. Let's list all the objects in the `cities` bucket using the following command:

```
# ./s3curl.pl --id=personal -- https://auth.vedams.com/cities
-v
```


- **Create a bucket:** The following command creates a bucket called `continents`:

```
# ./s3curl.pl --id=personal --createBucket -- -v  
https://auth.vedams.com/continents
```
- **Delete a bucket:** The following command deletes the bucket called `continents`:

```
# ./s3curl.pl --id=personal --delete -- -v  
https://auth.vedams.com/continents
```

Accessing Swift using client libraries

There are several libraries available in Java, Python, Ruby, PHP, and other programming languages for accessing the Swift cluster. Applications can be simplified using these libraries. Let's explore a few libraries.

Java

The Apache jclouds library (<http://jclouds.apache.org/guides/rackspace/>), particularly the `org.jclouds.openstack.swift.CommonSwiftClient` API, can be used to write applications in Java to connect to Swift and perform various operations on accounts, containers, and objects.

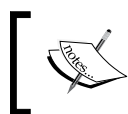
A sample code is shown as follows:

```
import org.jclouds.ContextBuilder;  
import org.jclouds.blobstore.BlobStore;  
import org.jclouds.blobstore.BlobStoreContext;  
import org.jclouds.openstack.swift.CommonSwiftAsyncClient;  
import org.jclouds.openstack.swift.CommonSwiftClient;  
  
BlobStoreContext context = ContextBuilder.newBuilder(provider)  
  
    .endpoint("http://auth.vedams.com/")  
    .credentials(user, password)  
    .modules(modules)  
    .buildView(BlobStoreContext.class);  
  
storage = context.getBlobStore();  
swift = context.unwrap();  
containers = swift.getApi().listContainers();  
objects = swift.getApi().listObjects(myContainer);
```

Python

The `python-swiftclient` library provides Python language bindings for OpenStack Swift. After authentication, the following sample code is used to list containers:

```
#!/usr/bin/env python
http_connection = http_connection(url)
cont = get_container(url, token, container, marker, limit, prefix,
delimiter, end_marker, path, http_conn)
```



More information about this library has been provided at
<https://github.com/openstack/python-swiftclient/>.

Ruby

The `ruby-openstack` library (<https://github.com/ruby-openstack/ruby-openstack>) provides ruby bindings for the OpenStack cloud. The following sample code shows us how to list containers and objects:

```
Lts2 = OpenStack::Connection.create(:username => USER, :api_key =>
API_KEY, :authtenant => TENANT, :auth_url => API_URL,
:service_type => "object-store")

Lts2.containers
=>["cities" , "countries"]

Cont = Lts2.container("cities")
Cont.objects
=>[" London.txt", " Mumbai.txt", " NewYork.txt"]
```

Summary

In this chapter, you learned how to use various Swift clients to interact with Swift clusters and get information on accounts, containers, and objects. You were introduced to ACLs, large object transfers, and also various Swift client libraries that can be used to write applications in your desired language, such as Java, Ruby, and Python.

The next chapter will talk about managing Swift, and the factors to be considered while replacing or expanding disks, nodes, and zones. It will also provide information on various tools that can be used to gather information about object storage behavior.

5

Additional Swift Interfaces

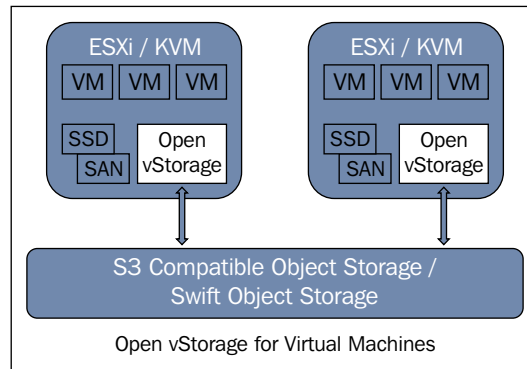
With object storage gaining popularity, its uses are also becoming varied. In several environments where higher-cost block storage devices were used, object storage is being considered as an alternative storage that provides low-cost deployment. However, this requires the use of certain intermediate interfaces or middleware to bind the upper level application or use case with the lower level object storage. In this chapter, we will talk about two such applications where object storage can be used for storage, and where previously block storage was being used.

One such application is the use of Swift object storage for virtual machine storage with the help of Open vStorage. Another application is the use of Swift in Sahara, which provides a Hadoop cluster for big data computations.

Using Swift for virtual machine storage

Typically, direct attach or SAN attach storage is used for data stores that provide block storage for virtual machines. Swift object storage has gained popularity due to its scale-out architecture, reliability, API interface, storage spaces, erasure coding, low cost, and several other benefits. However, it has a couple of problems, that is, high latency and unacceptable performance if it has to be used as a replacement for traditional block storage. Open vStorage is a technology that provides the perfect interface to solve this latency problem and make it possible for Swift object storage to be used as the underlying storage for virtual machines.

Open vStorage sits as a middleware between the hypervisor layer and the object storage, and provides high reliability and performance. Some of the main components are the VFS router, volume driver, file driver, and storage router.



The VFS router intercepts volume data from the virtual machine and sends it to the volume driver, which aggregates the data into chunks within the storage container. The storage router then distributes the storage container data to the object storage backend. A volume created for a virtual machine will be stored as a separate bucket in the S3-compatible object store.

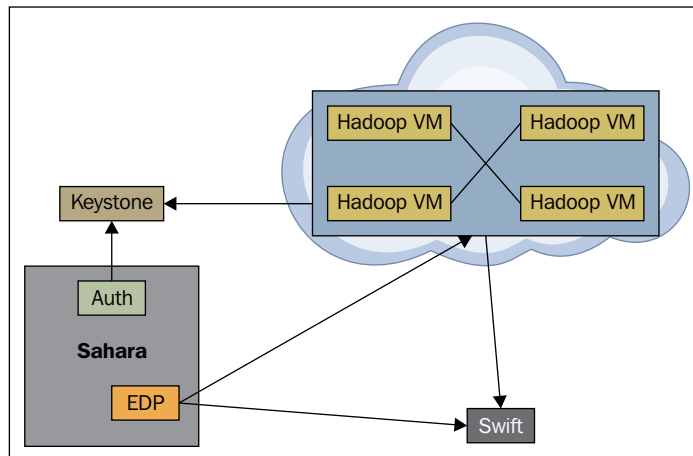
When a write operation is performed by the VM, it gets added to a **Storage Container Object (SCO)**. Typically an SCO consists of 4 MB of data, and the storage router pushes the SCOs to the object storage system when it becomes full. The SCO can be treated as a local cache, and it helps solve the eventual consistency problem present with Swift object storage.

Open vStorage is compatible with major hypervisors such as ESX and KVM. It also supports file-based storage using the file driver. These files can be stored on independent object storage backends. Also, based on the kind of file, storage policies can be set up to provide better reliability via additional replications or better performance via storage on SSDs.

Therefore, Open vStorage is an ideal interface for storing virtual machine data on compatible object storage backends.

Swift in Sahara

Sahara is a project within OpenStack intended for data processing. This is achieved by creating a Hadoop cluster using the various components available in OpenStack. Sahara was formerly known as **Savanna project**. The following diagram shows the architecture of Sahara:



Sahara uses various OpenStack components in order to enable the Hadoop cluster. The Horizon Dashboard service can be used to perform activities such as creation of a cluster, registration of images for the cluster, creation of jobs, and so on. The Keystone service is used to perform authentication for various components. The Glance service is used to register images in Sahara, and Swift is used for object storage.

Sahara uses plugins to create a Hadoop cluster. These plugins include **Vanilla** (to deploy Apache Hadoop), **Hortonworks Data Platform** (to deploy the Hortonworks data platform), **Spark** (to deploy the Cloudera HDFS), **MapR** (to deploy the MapR plugin with the MapR filesystem), and **Cloudera** (to deploy Cloudera Hadoop).

Hadoop Cluster with Sahara

The setup of a Hadoop cluster with Sahara can be done in different ways. As the deployment of a Hadoop cluster needs more than one OpenStack component, it can be simplified using automated deployments such as FUEL, developed by Mirantis. After successfully deploying and setting up the Sahara cluster, the configurations for Sahara can be seen in the `/etc/sahara/sahara.conf` file.

Using the Horizon dashboard, we can provide the data sources to be used with the Hadoop clusters. Swift object storage can be used as the data source, and it is specified using the `<container>/<object path>` format. The `swift://` will automatically get prepended while specifying this path.

Using Swift with Sahara

The main integration of Swift with Sahara happens at the filesystem level. The component responsible for this is the Hadoop Swift filesystem. It is a JAR file located in the `/usr/lib/share/hadoop/lib` folder. This was initially released as a patch called **Hadoop 8545**, which has now been integrated with Hadoop.

Sahara maintains the Hadoop cluster configuration in a file called `core-site.xml`, which contains the following information:

```
<property>
  <name>${name} + ${config}</name>
  <value>${value}</value>
  <description>${not mandatory description}</description>
</property>
```

When we use the Swift filesystem implementation, we specify `fs.swift.impl` for `${name} + ${config}` by editing the `core-site.xml` file, as follows:

```
<property>
  <name>fs.swift.impl</name>
  <value>org.apache.Hadoop.fs.Swift.snative.SwiftNativeFileSystem
</value>
</property>
```

There are some more configuration parameters such as `.auth.url` for the authorization URL, `.connect.timeout` for the connection timeout (the default is 60,000), `.blocksize` for representing the block size (the default is 32 MB), and so on. There are some configuration parameters specific to a particular provider. See the provider documentation to configure such parameters.

The configuration for Swift has to be provided in the map reducer programs of Hadoop for Keystone authentication, as shown in this code:

```
conf.set("Swift.auth.url", "http://auth.vedams.com/v2.0/tokens");
conf.set("Swift.tenant", "myuser");
conf.set("Swift.username", "admin");
conf.set("Swift.password", "password");
conf.setInt("Swift.http.port", 8080);
conf.setInt("Swift.https.port", 443);
```

Running a job in Sahara

Swiftfs (short for Swift filesystem for Hadoop) is a Hadoop filesystem implementation for Swift. It enables MapReduce (for batch processing), Pig (the scripting component of Hadoop), and Hive (the SQL query component of Hadoop) to directly perform file operations such as read and write on Swift containers.

The typical syntax for exporting a file to a Hadoop cluster with Swift is as follows:

```
swift://<<container.service_name>>/<<file_name>>
```

A typical intra-cluster copying of an object in Hadoop with Swift is shown in the following lines. Here, the `myobject` object within the `mycontainer` container is being copied to the `myobject1` object within the `mycontainer1` container:

```
$ hadoop distcp -D fs.swift.service.sahara.username=admin \
-D fs.swift.service.sahara.password=password \
swift://mycontainer.sahara/myobject
swift://mycontainer1.sahara/myobject1
```

Typical Hadoop commands and other shell commands can be found at these links:

- <http://hadoop.apache.org/docs/r2.7.0/hadoop-project-dist/hadoop-common/FileSystemShell.html>
- <http://hadoop.apache.org/docs/stable1/distcp.html>

Authenticating with Swift proxy

Implementing security for a Sahara cluster is a complicated task. Keystone has to store user credentials in order to authenticate users, which is a problem for large-scale batch processing (a large number of users may be running jobs). In order to overcome this, Sahara uses proxy users, which are users created during the start of a job and lasting until the job ends. This avoids storage of user credentials. This is achieved by creating a domain in Keystone that can hold the proxy users. This particular domain contains the details required for creating new user accounts.

In order to enable this usage, we need to change the following parameters in the `sahara.conf` file in `/etc/sahara`, as follows:

```
[DEFAULT]
use_domain_for_proxy_users=True
proxy_user_domain_name=myproxy
proxy_user_role_names=Swift
```

The preceding configuration parameters specify the domain name as `myproxy` and the role as `Swift`. The proxy users then use these roles to get authenticated from Swift. The details given in the preceding code are communicated by proxy users while accessing objects from Swift.

Summary

In this chapter, you learned how Open vStorage is used to interface with OpenStack Swift object storage to provide storage for virtual machines. We described how the Sahara component of OpenStack interfaces uses Swift for the Hadoop clusters.

The next chapter talks about managing Swift and things to consider while replacing or expanding disks, nodes, and zones. It also provides information on various tools that can be used to gather information on object storage behavior.

6

Monitoring and Managing Swift

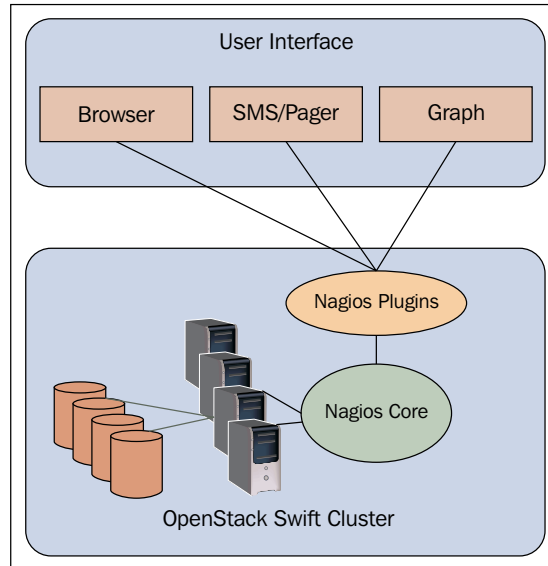
After a Swift cluster has been installed and deployed, it needs to be managed to serve customer expectations and service-level agreements. Since there are several components in a Swift cluster, it is a little different and more difficult to manage compared to traditional storage. There are several tools and mechanisms an administrator can use to effectively manage a Swift cluster. This chapter deals with these aspects in more detail.

Routine management

The Swift cluster consists of several proxy server nodes and a storage server node, and these nodes run many processes and services to keep the cluster up and running and to provide overall availability. Any kind of general server management tools or applications, such as Nagios, can be run to track the state of the general services, CPU utilization, memory utilization, disk subsystem performance, and so on. Looking at the system logs is a great way to detect impending failures. Along with this, there are some tools used to monitor Swift services in particular. Some of them are Swift Recon, Swift StatsD, Swift Dispersion, and Swift Informant.

Nagios is a monitoring framework that comprises several plugins that can be used to monitor network services (such as HTTP and SSH), processor load, performance, and CPU and disk utilization. It also provides remote monitoring capabilities by running scripts, remotely connected to the monitored system, using SSH or SSL. Users can write their own plugins, depending on their requirements, to extend these monitoring capabilities. These plugins can be written in several languages such as Perl, Ruby, C++, and Python. Nagios also provides a notification mechanism, where an administrator can be alerted when problems occur on the system.

The following figure shows you how to integrate a monitoring solution based on Nagios:



More information on Nagios can be found at www.nagios.org. Next, let's look into the details of Swift monitoring tools.

Swift cluster monitoring

In this section, we will describe various tools that are available for monitoring Swift clusters. We will also show you screenshots from the Vedams Swift monitoring application, which integrates data from various Swift monitoring tools.

Swift Recon

Swift Recon is a piece of middleware that is configured on the object server node and sits in the data path. It gets installed as part of the `python-swiftclient` installation. A local cache directory needs to be specified during setup, and it is used to store logs. It also comes with the `swift-recon` command-line tool, which can be used to access and display the various metrics that are being tracked. You can use `swift-recon -h` to get help on how to use the `swift-recon` tool.

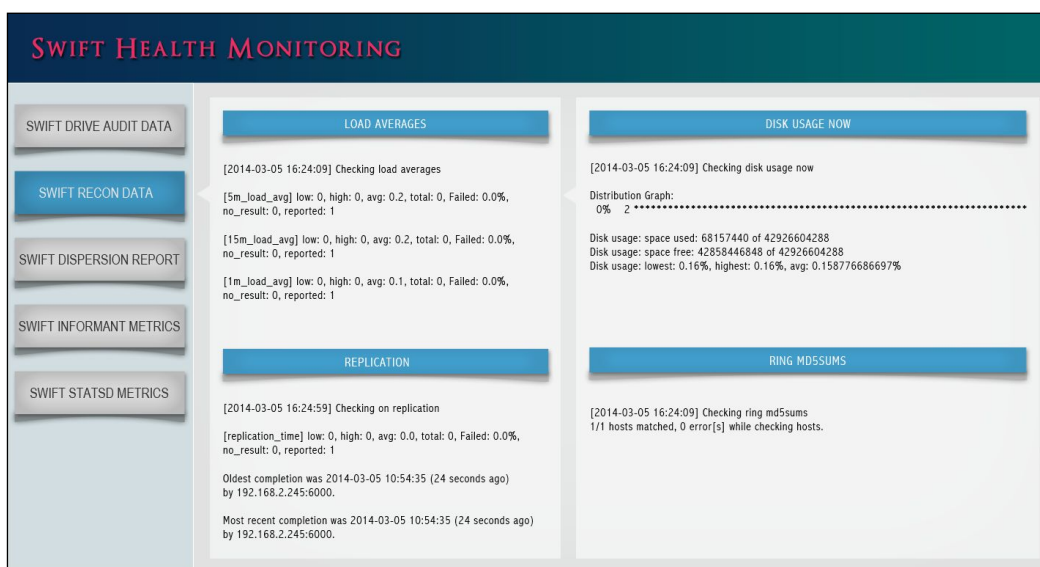
Some general server metrics that are tracked are as follows:

- Load averages
- The `/proc/meminfo` data
- Mounted filesystems
- Unmounted drives
- Socket statistics

Along with these, some of the following Swift stats are also tracked:

- MD5 checksums of the account, container, and object ring
- Replication information
- Number of quarantined accounts, containers, and objects

Vedams Swift health monitoring application is a GUI tool used to display consolidated information from several Swift tools. The following screenshot shows Swift Recon data within the Vedams Swift monitoring application:



Swift Informant

Swift Informant is a piece of middleware that gives an insight into client requests to the proxy server. This software sits in the proxy server's data path, and provides the following metrics to the StatsD server (which is a server that receives the StatsD metrics from the Swift cluster):

- HTTP response code (200, 201, 401, and so on) for requests to the account, container, or object.
- Duration of the request as well as the time taken for the `start_response` metric to appear. By default, this is measured in milliseconds.
- Bytes transferred in the request (in bytes).

Swift Informant can be downloaded from <https://github.com/pandemicsyn/swift-informant>.

The following screenshot displays Swift Informant's data within the Vedams Swift monitoring application:

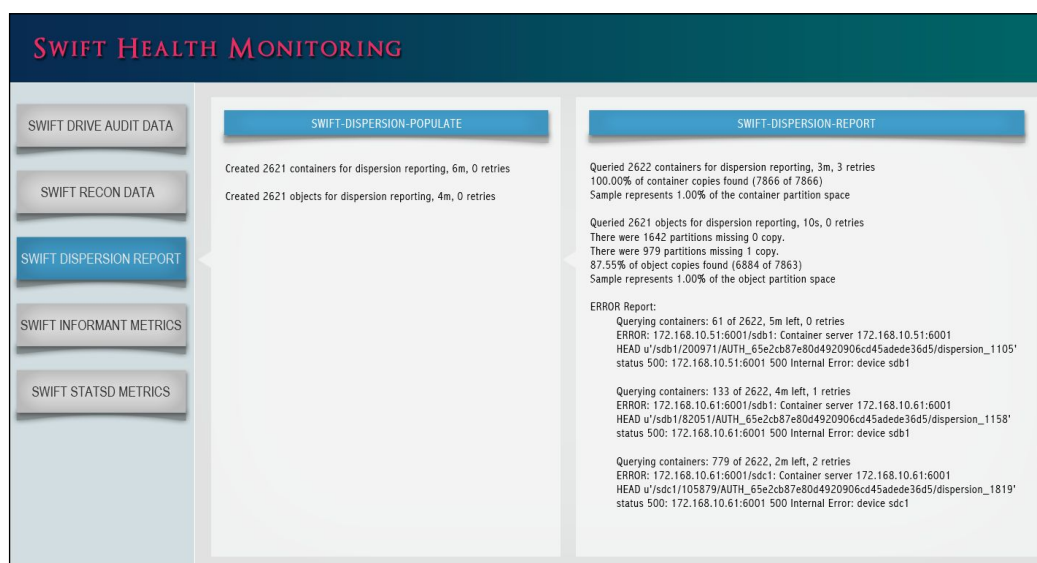
The screenshot displays the 'SWIFT HEALTH MONITORING' interface. On the left, there is a sidebar with several menu items: 'SWIFT DRIVE AUDIT DATA', 'SWIFT RECON DATA', 'SWIFT DISPERSION REPORT', 'SWIFT INFORMANT METRICS' (which is highlighted in blue), and 'SWIFT STATS METRICS'. The main content area shows a table of metrics categorized by 'Account', 'Container', and 'Object'. The table has four columns: 'COMPONENT', 'COUNTER', 'TIMING(ms)', and 'BYTES TRANSFERED'.

COMPONENT	COUNTER	TIMING(ms)	BYTES TRANSFERED
Account			
HEAD.204	1	99	--
GET.200	1	12	--
Container			
HEAD.204	1	59	--
GET.200	1	10	--
PUT.201	1	36	--
POST.204	1	64	--
DELETE.204	1	37	--
PUT.404	1		--
Object			
HEAD.200	1	75	--
GET.200	1	96	--
PUT.201	1	57	--
DELETE.204	1	30	--

Swift dispersion tool

Swift dispersion tool is a post-processing tool and is used to determine the overall health of a Swift cluster. The `swift-dispersion-populate` tool (which comes with `python-swiftclient`) is used to distribute random objects and containers throughout the Swift cluster in such a way that the random objects and containers fall under distinct partitions. Next, the `swift-dispersion-report` tool is run to determine the health of these objects and containers. In the case of objects, Swift creates three replicas for redundancy. If all the replicas of an object are good, then the health of the object is said to be good, and the `swift-dispersion-report` tool helps figure out this health of all objects and containers within the cluster.

The following screenshot displays the Swift dispersion data within the Vedams Swift monitoring application:



StatsD

Swift services have been instrumented to send statistics (counters, logs, and so on) directly to a StatsD server that is configured.

A simple StatsD daemon used to receive the metrics can be found at <https://github.com/etsy/statsd/>. The StatsD metrics are provided in real time and can help identify problems as they occur.



Configuration files (`proxy-server.conf`, `account-server.conf`, `container-server.conf`, and `object-server.conf`) containing the following parameters (default values given in parentheses) should be set in the Swift configuration files to enable StatsD logging:

- `log_statsd_host` (IP address)
- `log_statsd_port` (8125)
- `log_statsd_default_sample_rate` (1.0)
- `log_statsd_sample_rate_factor` (1.0)
- `log_statsd_metric_prefix`

The `statsd_sample_rate_factor` parameter can be adjusted to set the logging frequency. The `log_statsd_metric_prefix` prefix is configured on a node to prepend this prefix to every metric sent to the StatsD server from that node. If the `log_statsd_host` entry is not set, then this functionality will be disabled.

Swift metrics

Swift has the ability to log metrics such as counters, timings, and so on built into it. Some of the metrics sent to the StatsD server from various Swift services are as follows.

 The metrics have been classified based on the Create, Read, Update, and Delete (CRUD) operations. 

Create/PUT	Read/GET	Update/POST	Delete
account-server. PUT.errors.timing	account-server. GET.errors.timing	account-server. POST.errors.timing	account-server. DELETE.errors. timing
account-server. PUT.timing	account-server. GET.timing	account-server. POST.timing	account-server. DELETE.timing
container-server. PUT.errors.timing	container-server. GET.errors.timing	container-server. POST.errors.timing	container-server. DELETE.errors. timing
container-server. PUT.timing	container-server. GET.timing	container-server. POST.timing	container-server. DELETE.timing
object-server. async_pendings	object-server. GET.errors.timing	object-server. POST.errors.timing	object-server. async_pendings
object-server. PUT.errors.timing	object-server. GET.timing	object-server. POST.timing	object-server. DELETE.errors. timing

Create/PUT	Read/GET	Update/POST	Delete
object-server. PUT.timeouts	proxy-server. <type>.client_ timeouts	proxy-server.<type>. <verb>.<status>. timing	object-server. DELETE.timing
object-server. PUT.timing	proxy-server.<type>. <verb>.<status>. timing	proxy-server.<type>. <verb>.<status>. xfer	proxy-server.<type>. <verb>.<status>. timing
object-server. PUT.<device>. timing	proxy-server.<type>. <verb>.<status>. xfer		proxy-server.<type>. <verb>.<status>. xfer
proxy-server. <type>.client_ timeouts			
proxy-server. <type>.client_ disconnects			
proxy-server.<type>. <verb>.<status>. timing			
proxy-server.<type>. <verb>.<status>. xfer			

Tulsi – a Swift health monitoring tool

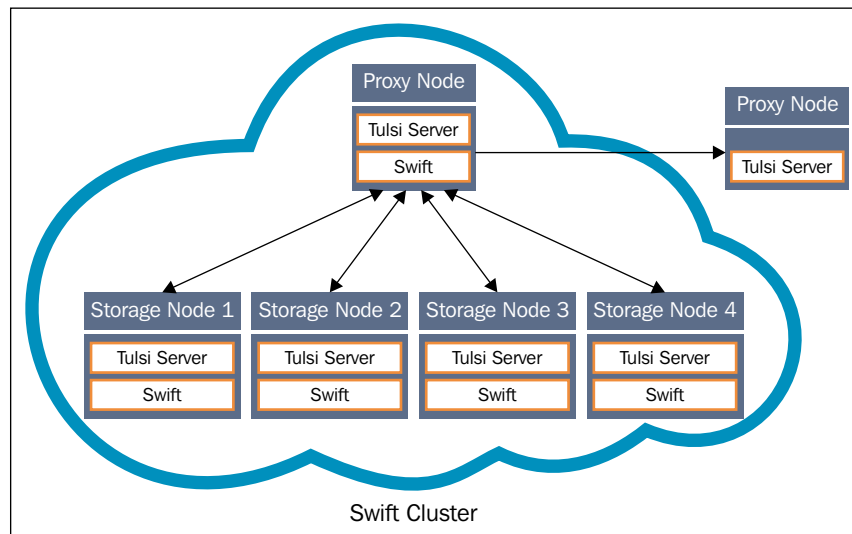
Tulsi is an open source tool used to monitor the health of a Swift cluster. It was developed by Vedams software solutions (www.vedams.com), and the code is provided on GitHub. The main intention of Tulsi is to check the status of drives and services of each node within the Swift cluster. This tool provides a graphical layout of the Swift cluster, along with the status of each node represented using appropriate colors. It also performs and applies an anomaly detection algorithm on top of the StatsD metrics to determine the health of the cluster.

The functions of Tulsi are as follows:

- Monitors the status of drives in the cluster
- Monitors the status of Swift services on each node in the cluster
- Monitors the logs of the StatsD metrics in the system
- Applies an anomaly detection algorithm on the StatsD metrics

Architecture of Tulsi

Tulsi is implemented using standard client-server architecture. It contains the Tulsi server package, which needs to be installed on every node within the Swift cluster, and the Tulsi client package, which needs to be installed on the client machine that will be used to monitor the Swift cluster. The server package sends UDP packets containing the status of the Swift cluster encoded in **JSON** (short for **Java Script Object Notation**) format. The following diagram depicts the Swift cluster with Tulsi deployed in it:



Deploying Tulsi

Deploying Tulsi is a pretty straightforward process. It is intended for Ubuntu deployment right now. The implementation on other platforms can be checked out in the README file of the package. There are two modules you will need to install in this deployment: **Tulsi server** and **Tulsi client**.

Download the Tulsi package by following these steps:

1. The Tulsi package can be downloaded from GitHub by running the following command:

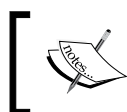
```
# git clone https://github.com/vedgithub/tulsi
```

2. Install the Tulsi server on all the Swift nodes by running these commands from the `tulsi` folder of the downloaded package:

```
# cd tulsi/TulsiServer/  
# sh tulsi.sh
```

3. Edit the `tulsi.conf` file in the `/etc/tulsi` folder and update the host and port parameters as shown here:

```
[tulsi]  
Host = Tulsi_Client_IP_Address  
Port = 5005
```



Try leaving the port as 5005. If you want to change the port, make sure that you assign the same port for both the client and server packages.

4. Next, install the Tulsi client on a system where you will be monitoring the cluster from. The installation procedure for different platforms can be found in the `README.md` file in the package. Install the client using the instructions provided in the `README.md` file.

Running Tulsi

After the installation procedure completes, start the services on all the storage and proxy nodes using this command:

```
# service tulsi start
```

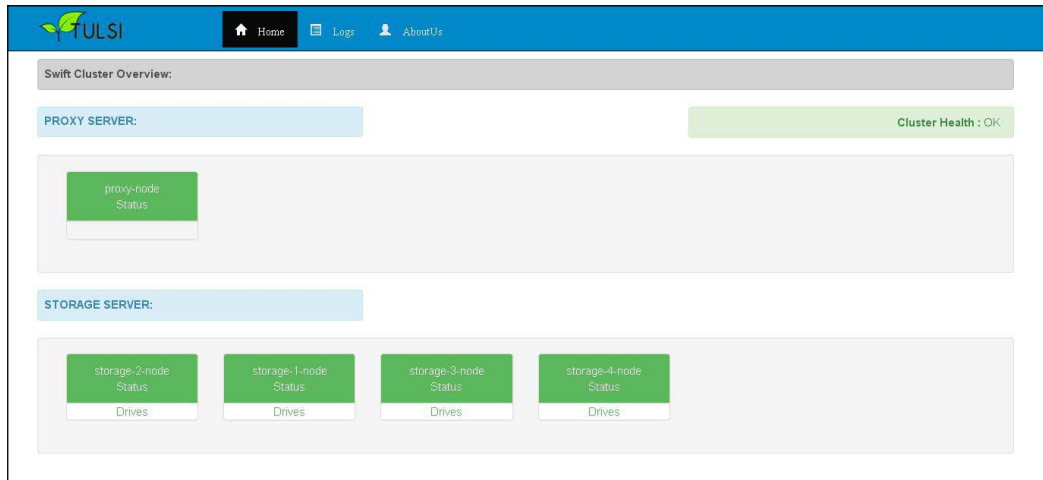
On the monitoring system, open the Tulsi client user interface by typing the following command in a web browser:

```
http://localhost:8080/Tulsi
```

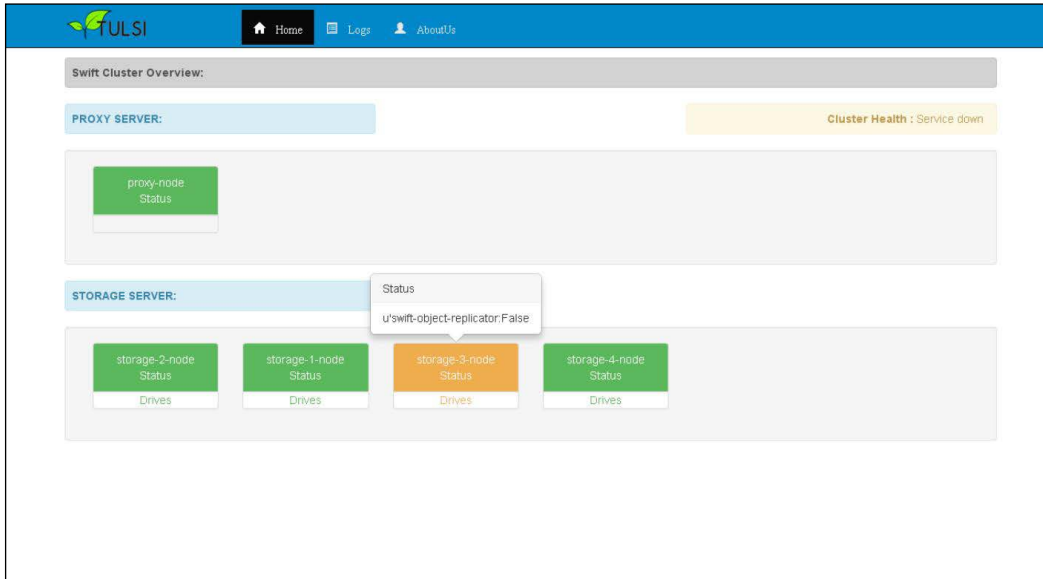
Alternatively, you can use this command:

```
http://<ip_of_tulsi_client_system>:8080/Tulsi
```

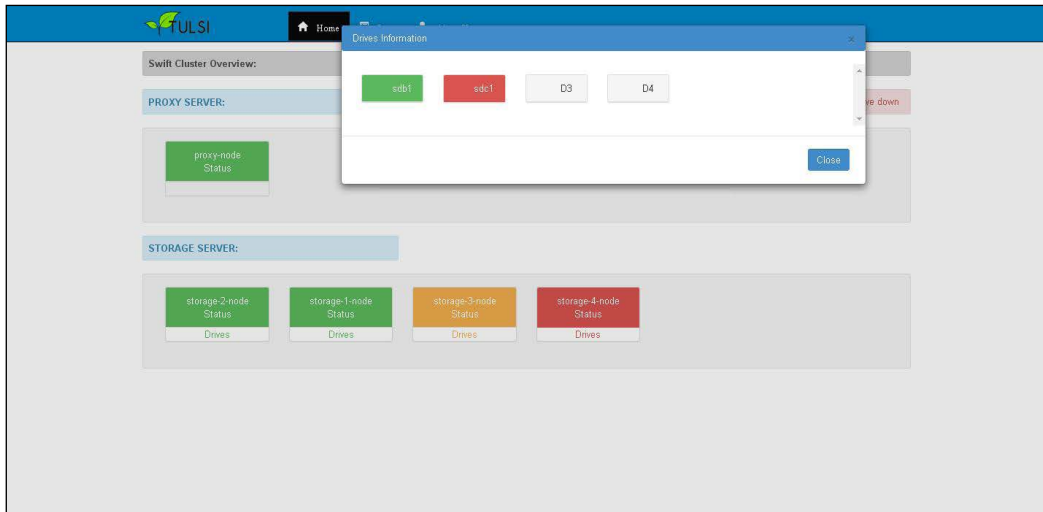
The web browser shows the status of the Swift cluster, as follows:



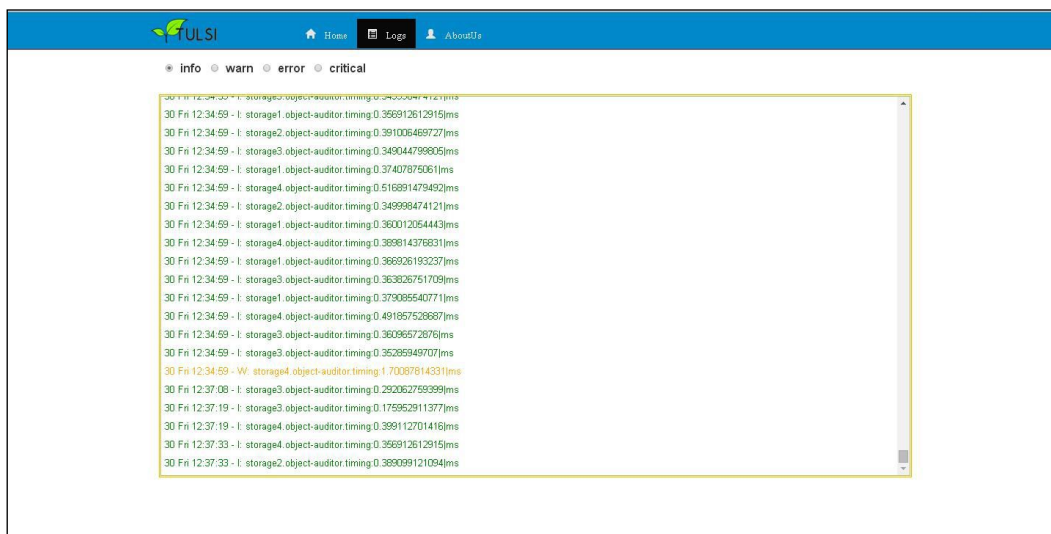
The default page displays the status of the proxy nodes as well as the storage nodes. A green node indicates a healthy status, amber indicates a service down status, and red indicates a problem with the disks on the storage nodes. Here is a screenshot showing a problem with one of the services on a storage node:



The following is a screenshot showing a problem with the disk on one of the storage nodes:

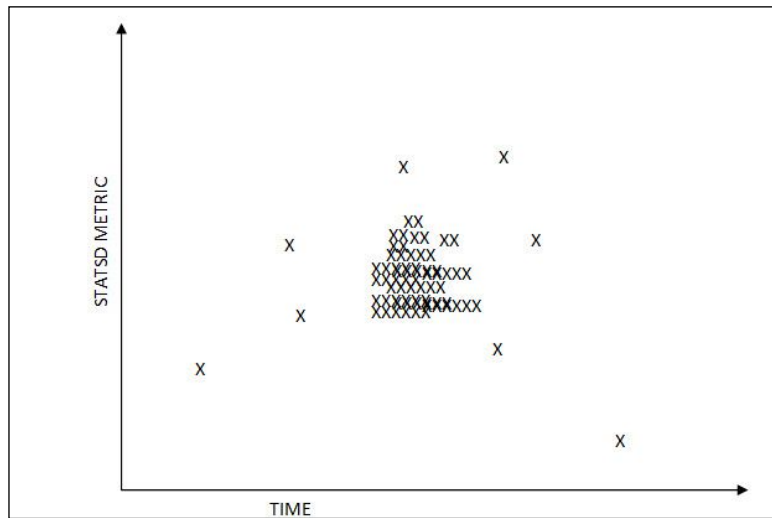


The web page has a **Logs** tab that can be used to show the StatsD metric logs received by the Tulsi client. These logs have been categorized as information, warning, error, and critical, and are represented by the prefixes **I**, **W**, **E**, and **C** in the logs. Here is a screenshot of the logs:

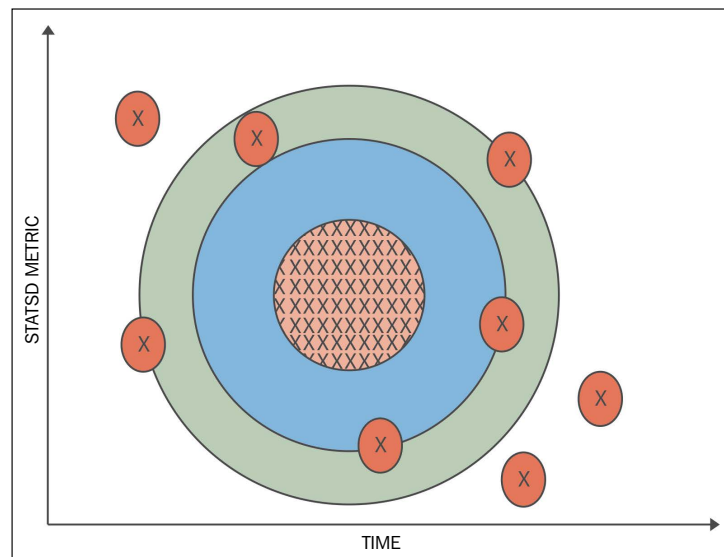


Anomaly detection in Tulsi

Anomaly detection is a machine learning algorithm that is applied to StatsD logs to determine anomalies in the metrics. Here is a diagram that depicts the incoming values of a StatsD metric received by a Tulsi client:



We use standard deviation and normalization techniques to determine the anomalies. The following diagram depicts the values after applying the anomaly detection algorithm:



More detailed information about the Tulsi server and client can be obtained by reading the sources available on GitHub.

Logging using rsyslog

It is very useful to get logs from various Swift services, and this can be achieved by configuring `proxy-server.conf` and `rsyslog`:

1. In order to receive logs from the proxy server, we modify the `/etc/swift/proxy-server.conf` configuration file by adding the following lines:

```
log_name = name
log_facility = LOG_LOCALx
log_level = LEVEL
```

Let's describe these entries. The `name` parameter can be any name that you want to see in the logs. The letter `x` in `LOG_LOCALx` can be any number between 0 and 7. `LEVEL` can be any one of emergency, alert, critical, error, warning, notification, informational, or debug.

2. Next, we modify `/etc/rsyslog.conf` to add the following line of code in the `GLOBAL_DIRECTIVES` section:

```
$PrivDropToGroup adm
```

3. Also, we create a config file called `/etc/rsyslog.d/swift.conf` and add the following line of code to it:

```
local2.* /var/log/swift/proxy.log
```

The preceding line tells `syslog` that any log written to the `LOG_LOCAL2` facility should go into the `/var/log/swift/proxy.log` file. We then give permissions for access to the `/var/log/swift` folder and restart the proxy service and `syslog` service.

Restart the proxy and `rsyslog` services using the following commands:

```
service rsyslog restart
swift-init rest restart
```

Failure management

In this section, we will deal with detecting failures, and actions to be done to rectify failures. There can be drive, server, zone, or even region failures. As described in *Chapter 2, OpenStack Swift Architecture*, during the CAP theorem discussion, Swift is designed for availability and tolerance to partial failure (where entire parts of a cluster can fail).

Detecting drive failures

Kernel logs are a good place to look for drive failures. The disk subsystem will log warnings or errors that can help an administrator determine whether drives are going bad or have already failed. We can also set up a script on storage nodes (explained in the following steps) to capture drive failure information using the drive audit process described in *Chapter 2, OpenStack Swift Architecture*.

1. On each storage node, create a `swift-drive-audit` script in the `/etc/swift` folder with the following content:

```
[drive-audit]
log_facility = LOG_LOCAL0
log_level = DEBUG
device_dir = /srv/node
minutes = 60
error_limit = 2
log_file_pattern = /var/log/kern*
regex_pattern_X = berrorb.*b(sd[a-z]{1,2}d?)b and b(sd[a-z]{1,2}d?)b.*berrorb
```

2. Add the following line of code to `/etc/rsyslog.d/swift.conf`:

```
local0.*      /var/log/swift/drive-audit
```

3. We then restart the `rsyslog` service using this command:

```
service rsyslog restart
```

4. Next, we restart the Swift services using the following command:

```
swift-init rest restart
```

The drive failure information will now be stored in the `/var/log/swift/drive-audit` log file.

Handling drive failure

When a drive failure occurs, we can replace the drive quickly, replace it at a later time, or not replace it at all. In any case, we have to first unmount the drive. If we do not plan to replace the drive immediately, then it is better to remove it from the ring. If we decide to replace the drive, then we take out the failed drive. We replace it with a good drive, format it, and mount it. We will then let the replication algorithm take care of filling this drive with data to maintain consistent replicas and data integrity.

Handling node failure

When a storage server in a Swift cluster is experiencing problems, we have to determine whether the problem can be fixed in a short interval, such as a few hours, or it might take an extended period of time. If the downtime interval is small, we can let Swift services work around the failure while we debug and fix the issue with the node.

Since Swift maintains multiple replicas of data (the default is three), there won't be any problem of data availability, but the time taken for data access might increase. As soon as the problem is found and fixed and the node is brought back up, Swift replication services will take care of figuring out the missing information, update the nodes, and get them in sync.

If the node repair time is extended, then it is better to remove the node and all the associated devices from the ring. Once the node is brought back online, the devices can be formatted, remounted, and added back to the ring.

The following commands are useful for removing devices and nodes from the ring:

- To remove a device from the ring, use this code:

```
swift-ring-builder <builder-file> remove  
<ip_address>/<device_name>
```

An example of use of the preceding code is `swift-ring-builder account.builder remove 172.168.10.52/sdb1`.

- Remove a server from the ring like this:

```
swift-ring-builder <builder-file> remove <ip_address>
```

An example of use of this code is `swift-ring-builder account.builder remove 172.168.10.52`.

Next, the ring needs to be rebalanced (*Chapter 3, Installing OpenStack Swift* mentions how to rebalance the ring).

Proxy server failure

If there is only one proxy server in the cluster and it goes down, then there is a chance that no objects can be accessed (uploaded or downloaded) by the client, so this needs immediate attention. This is why it is always a good idea to keep a redundant proxy server to increase data availability in the Swift cluster. After identifying and fixing the failure in the proxy server, the Swift services are restarted and object store access is restored.

Zone and region failure

When an entire zone fails, it is still possible that the Swift services are not interrupted because of the high-availability configuration containing multiple storage nodes and multiple zones.

The storage servers and drives belonging to the failed node must be brought back to the service if the failure can be debugged quickly. Otherwise, the storage servers and drives belonging to the zone need to be removed from the ring, and the ring needs to be rebalanced. Once the zone is brought back to the service, the drives and storage servers can be added back to the ring, and the ring can be rebalanced.

In general, zone failure should be dealt with as a critical issue. In some cases, the top-of-the-rack storage or network switch can witness failures, thus disconnecting storage arrays and servers from the Swift cluster and leading to zone failures. In these cases, the switch failures have to be diagnosed and rectified quickly.

In a multi-region setup, if there is a region failure, then all requests can be routed to the surviving regions. The servers and drives belonging to the region need to be brought back to service quickly to balance the load, which is currently being handled by the surviving regions. In other words, this failure should be dealt with as a blocker issue.

There can be latencies observed in uploads and downloads due to the requests being routed to different regions. Region failures can also occur due to failures occurring in core routers or firewalls. These failures should also be quickly diagnosed and rectified to bring the region back to the service.

Capacity planning

As more clients start accessing the Swift cluster, there will be an increase in the demand for additional storage. With Swift, this is easy to accomplish: you can simply add more storage nodes and associated proxy servers. This section deals with the planning and addition of new storage drives as well as storage servers.

Adding new drives

Though a straightforward process, adding new drives requires careful planning since it involves rebalancing of the ring. Once we decide to add new drives, we will add these drives to a particular storage server in a zone by formatting and mounting them. Next, we run the `swift-ring-builder add` command to add the drives to the ring.

Finally, we run the `swift-ring-builder rebalance` command to rebalance the ring. The generated `.gz` ring files need to be distributed to all storage server nodes. The commands used to perform these operations have been explained in *Chapter 3, Installing OpenStack Swift*, in the *Formatting and mounting hard disks* section and the *Ring setup* section.

More often than not, we will also end up replacing old drives with larger and better drives. In this scenario, instead of executing an abrupt move, it is better to slowly start migrating data from the old drive to the other drives by reducing the weight of the drive in the ring and repeating this step a few times. Once data has been moved from this drive, it can be safely removed. After removing the old drive, simply insert the new drive and follow the previously mentioned steps to add this drive to the ring.

Adding new storage and proxy servers

Adding new storage and proxy servers is also a straightforward process where new servers need to be provisioned according to the instructions provided in *Chapter 3, Installing OpenStack Swift*. Storage servers need to be placed in the right zones, and drives belonging to these servers need to be added to the ring.

After rebalancing and distributing the `.gz` ring files to the rest of the storage servers, the new storage servers are now part of the cluster. Similarly, after setting up a new proxy server, the configuration files and load balancing settings need to be updated. This proxy server is now part of the cluster and can start accepting requests from users.

Migrations

This section deals with hardware and software updates. The migrations can be to either existing servers or new servers within a zone or a region. As new hardware and software (operating system, packages, or Swift software) become available, the existing servers and software need to be updated to take advantage of faster processor speeds and the latest software updates. It is a good idea to upgrade one server at a time and one zone at a time, since Swift services can deal with an entire zone going away.

The following steps are required for upgrading a storage server node:

1. Execute the following command to stop all the Swift operations running in the background:

```
swift-init rest stop
```

2. Shut down all Swift services using the following command:
`swift-init {account|container|object} shutdown`
3. Upgrade the necessary operating system and system software packages, and install or upgrade the required Swift package. In general, Swift has a 6-month update cycle.
4. Next, create or make the required changes to the Swift config files.
5. After rebooting the server, make sure that the required services are running. If not, restart all the required services by executing the following commands:
`swift-init {account|container|object} start`
`swift-init rest start`

If there are changes with respect to the drives on the storage server, we have to make sure we update and rebalance the ring.

Once we have completed the upgrades, we check the log files for proper operation of the server. If the server is operating without any issues, we then proceed to upgrade the next storage server.

Next, we will discuss how to upgrade proxy servers. We can make use of the load balancer to isolate the proxy server that we plan to upgrade so that client requests are not sent to this proxy server.

We have to perform the following steps to upgrade the proxy server:

1. Shut down the proxy services using the following command:
`swift-init proxy shutdown`
2. Upgrade the necessary operating system and system software packages, and install or upgrade the required Swift package.
3. Next, create or make the required changes to the Swift proxy configuration files.
4. After rebooting the server, restart all the required services by running the following command:
`swift-init proxy start`

We then have to make sure that we add the upgraded proxy server back into the load balancer pool so that this proxy server can start receiving client requests.

After the upgrade, we have to make sure that the proxy server is operating correctly by monitoring the log files.

Summary

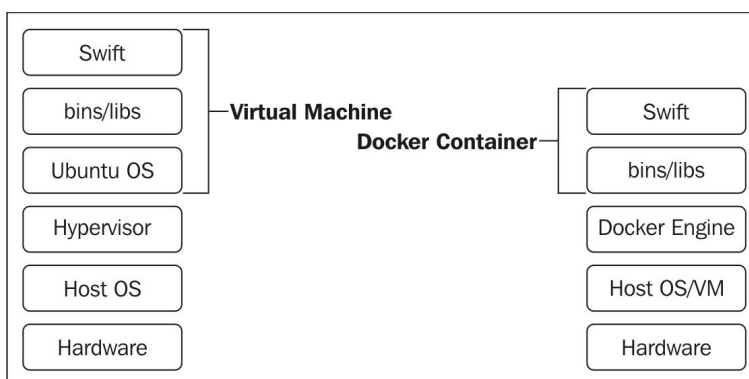
In this chapter, you learned how to manage a Swift cluster, the various tools available for monitoring and managing the Swift cluster, the various metrics used to determine the health of the cluster, and an open source tool called Tulsi. You also learned what actions need to be taken if a component fails in the cluster and how a cluster can be extended by adding new disks and nodes.

7

Docker Intercepts Swift

This chapter explains how to set up a Swift cluster on top of Docker. Let's briefly delve into Docker to make this chapter easier to understand. Docker is all about making it easier to create, deploy, and run applications using containers. Containers allow a developer to package an application with all the parts it needs, such as libraries and other dependencies, and ship it as one package.

Docker is a bit like a virtual machine. However, unlike a virtual machine, where a complete virtual operating system gets installed, Docker allows applications to use the same Linux kernel as the system they're running on, and only requires applications to be shipped with things not already running on the host computer. Refer to the following diagram:



A Docker container versus a virtual machine

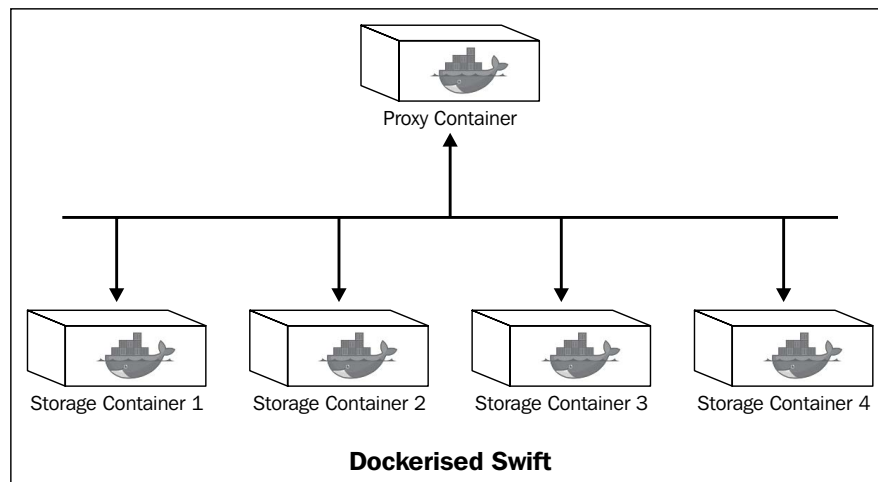
Virtual machines have a full operating system with their own memory management installed with the associated overhead of virtual device drivers. In a virtual machine, valuable resources are emulated for the guest OS and hypervisor, which makes it possible to run many instances of one or more operating systems in parallel on a single machine (or host). Every guest operating system runs as an individual entity from the host system.

On the other hand, Docker containers are executed with the Docker engine rather than the hypervisor. Containers are therefore smaller than virtual machines and enable faster startup with better performance, less isolation, and greater compatibility due to sharing of the host's kernel.

Swift with Docker

Swift with Docker is a way of creating an OpenStack Swift cluster as a Docker container after taking care of all the dependencies. A Docker image of this container can then be created and uploaded to the Docker repository. After that, the image can be pulled down to any other system and used to create a new container.

Any addition of nodes will just be a creation of additional containers using the Docker image. Refer to the following diagram:



Installation of Docker

This section talks about the installation of Docker and the environment for Docker. For the Docker software, we need a 64-bit host operating system. In this section, we will describe how to install Docker on an Ubuntu 14.04 Linux operating system. Refer to the other Linux distribution command sets to install the clients in those operating systems.

Log in to your Ubuntu installation as a user with the `sudo` privileges.

Update the package index of the host using the following command:

```
# apt-get update
```

Verify that you have `wget` installed using this command:

```
# which wget
```

If `wget` isn't installed, install it using the following command:

```
# apt-get install wget
```

Get the latest Docker package using this command:

```
# wget -qO- https://get.docker.com/ | sh
```

Verify that Docker is installed correctly by running the following command:

```
# docker run hello-world
```

The preceding command downloads a test image and runs it in a container.

Basic commands for the Docker user

This section talks about the basic command line for using Docker. In order to handle Docker containers, we need some commands for creating images, running containers, and so on.

In the Docker hub, there are already a number of images uploaded and available. If we need to use an image, we have to pull it to our localhost. The following command can be used to download images from Docker:

```
# docker pull <image_name>
```


For example, suppose you want to download the Ubuntu 14.04 official container, using the following command:

```
# docker pull ubuntu:14.04
```

We need to make sure that we specify the version name. Otherwise, whatever Ubuntu versions are available inside the Docker hub will be downloaded. If you want to download the latest version of any container, then specify `latest` instead of the version, for example, `ubuntu:latest`.

This command is used to list images on your local machine:

```
# docker images
```

Create a container that uses the existing images by using the following command:

```
# docker run -i -t <image_name> /bin/bash
```

We can run multiple containers on a single host. In this case, all the containers will use resources from the host machine. The following are some basic commands that can be used to manage containers:

- Show all created containers on the host machine:

```
# docker ps -a
```
- Show the running container on the host machine:

```
# docker ps
```
- Attach the running container to your prompt for making changes inside the container.

```
# docker attach <container_name>
```

The following commands are used to start, stop, and restart the container.

```
# docker start <container_id>
# docker stop <container_id>
# docker restart <container_id>
```

After installing the required packages within a container and making any required modifications, the following command can be used to create an image of this modified container, which can then be uploaded to a public or private account on the Docker hub:

```
# docker commit <container_id> "image_name"
```

For uploading images to the Docker hub, we need a Docker hub account. Use the Docker login command and provide the username, password, and e-mail ID of your Docker hub account.

Run the following command to log in. After running this command, the Docker hub will ask for the e-mail, password, and username of your Docker repository.

```
# docker login
```

After login, push the image to the Docker hub using the following command:

```
# docker push <image_name>
```

If you want to reuse this image from another host machine, then pull the image to your host machine and create the container using the downloaded image.

Setting up a Swift proxy container using the Docker image

The Swift proxy node is the controller node of the Swift cluster. For the configuration of the proxy node, we have to install the related packages of the proxy node, but here we will show you how to reuse an already configured proxy node Docker image. This image is provided by Vedams.

To download this image, run this command:

```
# docker pull vedams/swift-proxy
```

After downloading the image, we have to create the container for the proxy node. Docker has some limitations when using static IP addresses for the Docker container. We can perform a mapping to the host IP address to enable communication between proxy and storage containers.

Host networking mode is needed for this container to work, so the `--net=host` parameter must be used when starting up an instance of this container.

```
# docker run -i -t --net=host --name="proxy" vedams/swift-proxy /bin/bash
```

Now that we have created the proxy container, we have to do some basic configuration to run the proxy container.

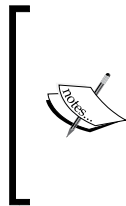
First, create the ring files as per your Swift cluster and copy them to the proxy container and storage container. If any changes (for example, `bind_ip`, `auth_ip`, any `statsd` configurations, and so on) are required to the `/etc/swift/proxy-server.conf` file, make the necessary changes.

Ensure proper ownership of the `/etc/swift` folder by running this command:

```
# chown -R swift:swift /etc/swift
```

Run this command to start all the services of the proxy node:

```
# service mysql restart
# service supervisor restart
```



Docker container runs a single process when it is launched, but in Swift cluster we need more than one process. For that, we are using supervisor. Using supervisor allows us to better control, manage, and restart the processes that we want to run. The configurations of all services are located in `/etc/supervisor/conf.d/supervisord.conf`.

Since the Keystone endpoint in the container image was created using some default values, you need to delete this endpoint and create your endpoint, as described in *Chapter 3, Installing OpenStack Swift*, under the *Installing Keystone* section.

Update the `/home/keystonerc` file to provide the username, password, tenant, endpoint, and so on. Provide the Swift environment variables using this command:

```
# source keystonerc
```

Setting up the storage container using the Docker image

Download the `swift-storage` image using the following command:

```
# docker pull vedams/swift-storage
```

In order for Docker to use entire filesystems from its host OS, those filesystems need to be mounted on the host OS, and they need to be passed to the container using the `-v` parameters. Therefore, create drives, create the filesystem on those drives, and mount it.

The following command is used to install the XFS filesystem:

```
# apt-get install xfsprogs
```

Perform the partitioning for `sdb` and create the filesystem on this partition. We also have to add a line in `fstab` for this partition. The commands used to perform these steps are shown in the following screenshot:

```
# fdisk /dev/sdb
# mkfs.xfs /dev/sdb1
# echo "/dev/sdb1 /srv/node/sdb1 xfs noatime,nodiratime,nobarrier,logbufs=8 0 0" >> /etc/fstab
# mkdir -p /srv/node/sdb1
# mount /srv/node/sdb1
```

Let's now create a container for the storage node; we are using the `-v` parameter to map the host drive on the container:

```
# docker run -i -t --net=host --name="storage" -v /srv/node/sdb1:/srv/node/sdb1:rw vedams/swift-storage /bin/bash
```

Now the storage container is created with a single storage drive. We can map more than one storage drive on the storage container.

Before using the storage container, we need to perform some basic configurations:

- Replace the ring files as per your Swift cluster in the `/etc/swift` directory
- Replace `bind_ip` in the following files with your storage container host IP:


```
# vi /etc/swift/account-server.conf
# vi /etc/swift/container-server.conf
# vi /etc/swift/object-server.conf
```
- Replace the IP address of the `/etc/rsyncd.conf` file with the storage host IP address
- Ensure proper ownership of all folders related to the storage container:


```
# chown -R swift:swift /srv/node
# chown -R swift:swift /etc/swift
```
- Run the following command to start all services related to the storage container:


```
# service supervisor restart
```

Setting up a Swift cluster using a Dockerfile

A Dockerfile is a text document that contains all the commands you normally execute manually in order to build a Docker image. By calling Docker build from your terminal, you can have Docker build your image step by step, executing the instructions successively.

Download the Swift proxy and storage Dockerfile and related files. Vedams has provided a Git repository for all Dockerfiles and files related to Swift. Download the repository using this command:

```
git clone https://github.com/vedams-docker/swift-with-docker.git
```

Creating a proxy container using a Dockerfile

First of all, we need an image to create the proxy container. Here, we will use a Dockerfile to create the image.

Create a folder on the host machine and copy and paste the proxy folder into it. Navigate to the folder and you will find the Dockerfile and files folder.

Inside the files folder, there are ring files and configuration files related to the Swift proxy. Replace the ring files as per your Swift cluster configuration, and make other necessary changes in all configuration files as per your requirements. Change `bind_ip` in the `proxy-server.conf` file:

1. After completing the changes inside the files folder, come back to the proxy directory and run this command:

```
# docker build -t="swift-proxy" .
```
2. After the image-building process completes, run the following command to get the proxy container:

```
# docker run -i -t --net=host --name="proxy" swift-proxy /bin/bash
```
3. Ensure proper ownership for the `/etc/swift` folder:

```
# chown -R swift:swift /etc/swift
```

4. Run this command to start all the services of the proxy node:

```
# service mysql restart
# service supervisor restart
```

Create a database for the Keystone and user, tenant and their role (follow *The Keystone service* section in *Chapter 3, Installing OpenStack Swift*).

Creating a storage container using a Dockerfile

Create an image using the Dockerfile in storage:

1. Copy the storage folder to the host, replace the ring files, and configure all the configuration files.
2. Next, run this command:

```
# docker build -t="swift-storage" .
```
3. After creating an image of the storage node, create the container for storage. Run the following command to create the storage container:

```
# docker run -i -t --net=host --name="storage" -v
/srv/node/sdb1:/srv/node/sdb1:rw swift-storage /bin/bash
```
4. Ensure proper ownership of all folders related to the storage container:

```
# chown -R swift:swift /srv/node
# chown -R swift:swift /etc/swift
```
5. Run the following command to start all services related to the storage container:

```
# service supervisor restart
```

The setup for running your Swift cluster in a Dockerized environment is now complete.

Summary

In this chapter, you learned about Docker and how to set up and run your OpenStack Swift cluster in a Dockerized environment.

The next chapter talks about choosing the right hardware for your OpenStack Swift cluster, and the things you need to consider when doing so.

8

Choosing the Right Hardware

Users who utilize OpenStack Swift in their private cloud will be faced with the task of hardware selection. This chapter walks through all of the hardware you need to select, the criteria to be used, and finally a vendor selection strategy. If you are using a public cloud, the only hardware you might select is the cloud gateway, and you can skip the remaining part of the chapter.

The hardware list

The list of the minimum hardware required to install Swift is as follows:

Item	Description
Storage servers	These are physical servers that run the object server software, and generally also run the account and container server software. Storage servers require disks or solid-state drives (SSDs) to store objects.
Proxy server (or servers)	These are physical servers that run the proxy server software. At least one is required.
Network switch (or switches)	<i>Chapter 3, Installing OpenStack Swift</i> , describes the various networks required. At least one switch is required.

The following is a list of optional hardware that may need to be purchased:

Item	Description
Account servers	For large installations where container listings and updates overwhelm the storage servers, separate account servers may be needed.
Container servers	For large installations where object listings and updates overwhelm the storage servers, separate container servers may be needed.
Auth servers	For large installations where user authentication overwhelms the proxy servers, separate auth servers may be needed.
JBODs	For installations where disk density is important, a storage server may be connected to a JBOD (short for just a bunch of disks) using a SAS connection to increase disk density.
Load balancer / SSL acceleration	This is useful for providing a single IP address for the entire cluster (there are software mechanisms for accomplishing this as well, but these are not covered in this book). The SSL functionality in the load balancer offloads the software SSL in the proxy server.
Firewall and security appliances	For public, community, and some private networks, firewall and security appliances such as intrusion detection/prevention may be required, depending on your company's security policies.
On-premise cloud gateway	To adapt applications that have not ported to the REST HTTP APIs yet, you will need a protocol translation device that converts a familiar file and blocks protocols to REST APIs. This device is called a cloud gateway, and it is the only piece of hardware that you may need, even in the case of a public cloud.

To complicate things even further, each server has numerous design elements to configure:

- **CPU performance:** The CPU's performance is specified in terms of the number of processors and number of cores-per-processors. This has the most direct impact on the server's performance.
- **Memory:** The next important consideration is the amount of DRAM memory, which is specified in GB.
- **Flash memory:** Flash memory is another critical performance consideration, and it is typically in the TB range.

- **Disk/JBOD:** For storage servers, you need to specify the number of disks and types of disks (interface, speed, rating, and so on). Alternatively, you may choose SSDs. For the remaining part of the chapter, the term "disk" includes SSDs. These disks can be in the server, connected via a JBOD, or a combination of the two.
- **Network I/O:** A server needs network I/O connectivity via a **LAN-on-motherboard (LOM)** or an add-on **network interface card (NIC)**. This is typically 1 GB per second or 10 GB per second in terms of speed.
- **Hardware management:** Servers vary widely in hardware management features, starting with rudimentary monitoring only through the operating system and OS-independent IPMI all the way to sophisticated remote KVM and remote storage.

The hardware selection criteria

The permutations of hardware choices and the elements within each server are numerous. Further, the ratio of proxy to account to container to storage servers is yet another complication. Before we go through the systematic selection criteria, we need to determine the following characteristics about our environment:

- **Point of optimization for your environment:** You will need to decide whether you want to optimize for performance or cost. Of course, with storage policies, you can mix both use cases in the same cluster, but for now it is useful to separate the two use cases.
- **Scale:** The scale also has a huge impact on hardware selection. For simplicity, let's say *small* is in the range of hundreds of TB, *medium* is in the PB range, and *large* is in the range of tens of PB and beyond. You will need to determine what range you want to design the cluster for.

The process of choosing hardware is as follows, in stepwise order.

Choosing the storage server configuration

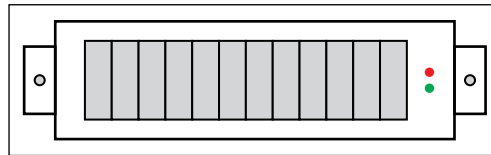
Let's see how various levels of installations affect the selection of the storage server.

- For small and medium installations, the storage server can include the object, account, and container server software.
- For large installations, we recommend a separate account and/or container servers. This limits the use of the storage server as just an object server.

- For performance-optimized clusters, the aggregate disk performance must match the total performance of the other server components (CPU, memory, flash, and I/O).
- For cost-optimized clusters, the disk performance can exceed the performance of other components (in other words, saving money to throttle performance).

In fact, consider attaching JBODs to significantly increase disk density.

A higher disk density also results in slight reliability degradation, since a node failure takes longer to self-heal, and two additional failures (if you have three copies) have a slightly higher probability of occurring during this longer duration. Of course, the probability of two failures occurring in one self-heal window is very low in both cases. The following diagram denotes a storage server with disks (of course, an optional JBOD may also be connected to it):



A storage server and an optional JBOD

The OpenStack configuration guide at <http://docs.openstack.org/juno/install-guide/install/apt/content/object-storage-system-requirements.html> recommends the following server specifications:

- **Processor:** Dual quad-core.
- **Memory:** 8 to 12 GB.
- **Network I/O:** 1 x 1 Gbps NIC. Cost permitting, our recommendation is to go beyond the official recommendation and use 10 Gbps.
- **Disks/JBOD:** Additionally, we need to consider this for direct attached storage. The number of hard drives or SSDs depends on the density-versus-performance trade-off desired. As of 2013, Rackspace used 96 x 3 TB SATA drives per storage server.

For direct attached storage, RAID should not be turned on due to performance degradation. Moreover, RAID may be counterproductive for other reasons too: for example, Swift flushes data to the disk to confirm a write, but a RAID controller may intercept it and not commit the write to the disk immediately.

Next, a key consideration is the type of disk: enterprise or desktop. Within enterprise disks, there are 15,000, 10,000, or 7,200 rotations-per-minute (RPM) drives and a variety of capacity configurations. Although it is prohibitively expensive to populate all storage servers exclusively with SSDs, a small percentage of drives may include SSDs for very high performance, and storage policies can be used to carve up the cluster.

For small and medium installations, your enterprise drives may be most appropriate, as they are more reliable than desktop drives. Most small and medium installations are typically not set up to deal with the higher failure rate of desktop drives. The performance and capacity that you choose for an enterprise drive obviously depends on your specific requirements.

Large installations that are also very cost sensitive may warrant consideration of desktop drives. The density of desktop drives (up to 6 TB at the time of writing this book) also contributes favorably to large installations. In addition to reliability, desktop drives are not specified to be able to run 24x7. This means that your IT staff have to be sophisticated enough to deal with a large number of failures and/or spin down drives to conform to the specification.

Determining the region and zone configuration

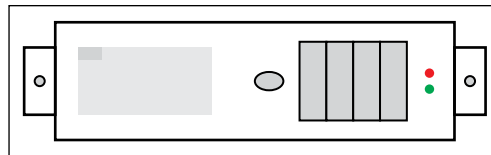
Next, we need to decide on regions and zones. The number of regions stems from the desire to protect data from a disaster or to be closer to the sources consuming data. For example, if the goal is to protect against disasters in the United States, two regions in different Federal Emergency Management Agency (FEMA) regions ought to be sufficient.

Once the number of regions has been determined, we need to choose the number of zones for each region. While this is not a must, we recommend at least as many zones as there are replicas to ensure that the data is available even in the case of a zone being unavailable. We recommend no fewer than three zones, and Rackspace uses five (<http://docs.openstack.org/juno/install-guide/install/apt/content/example-object-storage-installation-architecture.html>).

Small clusters may be fine with four zones. Refer to *Chapter 2, OpenStack Swift Architecture*, for a refresher on the definition of regions and zones.

Choosing the account and container server configuration

As previously mentioned, except for large configurations, separate account and container servers are generally not required, and they can be combined with object servers (see the *Choosing the storage server configuration* section). For separate account and/or container servers, the SQL database performance should be adequate to meet your database listing and update needs by selecting the right amount of memory and flash.



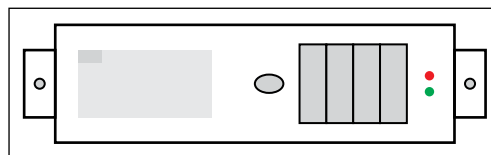
An optional account and a container server

The OpenStack configuration guide recommends the following specifications (you may be able to reduce the requirements based on your cluster's size and performance requirements):

- **Processor:** Dual quad-core.
- **Memory:** 8 to 12 GB.
- **Network I/O:** 1 x 1 Gbps NIC. Cost permitting, our recommendation is to go beyond the official recommendation and use 10 Gbps.
- **SSD/Flash:** Not specified. This depends on the user's performance requirements.

Selecting the proxy server configuration

In general, the proxy server needs to keep up with the number of API requests. As discussed in *Chapter 2, OpenStack Swift Architecture*, additional middleware modules may also be running on the proxy server. Therefore, the proxy server needs performance that can keep up with this workload. Using a few powerful proxy servers as opposed to a large number of **wimpy** servers was proven to be more cost effective by Zmanda (<http://www.zmanda.com/blogs/?p=774>).



A proxy server

The OpenStack configuration guide seems to concur, and recommends the following specifications:

- **Processor:** Dual quad-core.
- **Network I/O:** A 1 x 1 Gbps NIC. Our recommendation is to use at least two NICs, one for internal storage cluster connectivity and one for client (API) facing traffic. Cost permitting, our recommendation is to go beyond the official recommendation and use at least 10 Gbps for internal storage cluster connectivity. Also see the related SSL discussion in the *Choosing additional networking equipment* section that affects network I/O.

If your proxy server is running a lot of middleware modules, consider moving some of them to dedicated servers. The most common middleware to be separated is the auth software.

Choosing the network hardware

There are three networks mentioned earlier — client (API) facing, internal storage cluster, and replication. See *Chapter 3, Installing OpenStack Swift*, for a view of the architecture of these three networks. This might be a combination of 1 GB per second, 10 GB per second, or hybrid 1/10 GB per second Ethernet switches. The following are some performance-related sizing techniques:

- **Client-facing network:** The throughput requirement of the overall cluster dictates the network I/O for this network. For example, if your cluster has 10 proxy servers and is sized to satisfy 10,000 I/O requests per second of 1 MB size each, then clearly each proxy server needs 10 GB per second network I/O capability.
- **Internal storage cluster:** The network requirements depend on the overall cluster throughput and size of the cluster. The size of the cluster matters since it will generate a large amount of post-processing software component traffic (see *Chapter 2, OpenStack Swift Architecture*). As mentioned before, we recommend the use of a 10 GB per second network, cost permitting.
- **Replication network:** This depends on the overall write throughput and the size of the cluster. For example, for 1,000 write requests per second of 1 KB each, a 10 MB per second network might perform adequately.

An additional consideration is the availability model. Network switches take down entire zones or regions, so unless you can service the failed switches rapidly, you might want to consider dual-redundant configurations. The following figure denotes a network switch:



A network switch

Choosing the ratios of various server types

After selecting the individual server configurations, the ratios of different server types have to be chosen. Since most configurations will have only two types, that is, proxy and storage, we will discuss the ratios of only these two. According to work done by Zmanda, the proxy server should neither be underutilized nor overutilized. If the throughput of one storage server is 1 GB per second and that of the proxy server is 10 GB per second for example, then the ratio is 10:1. (This simple calculation applies to large objects dominated by throughput. For smaller objects, the calculation needs to focus on the number of requests.)

Instead of buying hardware piecemeal, this ratio exercise allows a user to define a "unit" of purchase. The unit may be a full rack of hardware, multiple racks, or a few rack units. A unit of hardware is orthogonal to Swift zones, and typically you would want each unit to add capacity to every zone in a symmetrical fashion. Each unit can have a set of proxy servers, storage servers, network switches, and so on defined in detail. Scaling the Swift cluster as data grows becomes a lot simpler using this technique of purchase. As mentioned earlier, you need to start with at least two proxies to provide for adequate availability.

For example, assume you want to grow your cluster in roughly 1 PB raw storage increments, with dense configurations. You might consider a unit of hardware with one proxy server, two 10 Gbps switches, one management switch, and five storage servers with 60 drives of 4 TB each (that is, $240\text{ TB} \times 5 = 1.2\text{ PB}$). Given the previous comment regarding the need for at least two proxy servers, the initial installation would have to be 2.4 PB. With triple replication, the 1.2 PB raw storage translates into 400 TB usable storage. This example is not perfect because it may not fit cleanly within the rack boundaries, but it is meant to illustrate the point.

Heterogeneous hardware

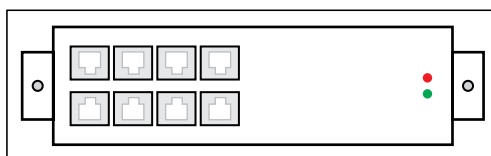
As described in *Chapter 2, OpenStack Swift Architecture*, storage policies allow users to mix heterogeneous hardware into one single cluster. Data is segregated based on these storage policies. While heterogeneous hardware provides more flexibility, it also contributes to complexity. The previous six steps may be used to calculate various hardware types.

Choosing additional networking equipment

The final step is to choose the load balancer, SSL acceleration hardware, and security appliances. A load balancer is required if there is more than one proxy node. Furthermore, you need to ensure that the load balancer is not a performance bottleneck.

SSL hardware acceleration is required if most of the traffic is over secure HTTP (HTTPS) and the software SSL operation is found to overwhelm the proxy servers. In fact, even if you use software SSL, you might find an SSL terminator such as **Pound** to be better than using the Python-based SSL terminator in the proxy server.

Finally, security appliances such as IPS and IDS are required if the cloud is on the public Internet. These requirements vary according to company policies. Similar to the load balancer, these additional pieces of hardware must have enough performance to keep up with the aggregate proxy server's performance. The following diagram denotes additional networking equipment needed for your Swift cluster:



Selecting a cloud gateway

This piece of equipment is different from the rest. It is not required to build an OpenStack Swift cluster. Instead, it is needed on the premises (in the case of a public cloud) or near the application (in the case of a private cloud) if your application has not been ported yet to HTTP REST APIs. In this situation, the application expects a traditional block or file storage, which is the interface exposed by these cloud gateways.

The gateway performs protocol translation and interfaces with the cloud on the other side. In addition to protocol translation, cloud gateways often add numerous other features such as WAN optimization, compression, deduplication, and encryption. Since gateways are not part of the OpenStack cluster, the selection criteria are outside the scope of this book.

While most of this section has dealt with performance, there are other considerations as well, which are covered in the next section.

Additional selection criteria

In addition to the previous criteria, the following also need to be considered before finalizing your hardware selection:

- **Durability:** Durability is a measure of reliability and is defined as 100 percent minus the probability of losing a 1 KB object in 1 year. Therefore, 99.99999999 percent durability (simply stated as 11 x 9 in this case) implies that every year, you statistically lose one object if you have 100 billion 1 KB objects. Alternatively, given 10,000 objects, you expect a loss of one object every 10,000,000 years.



Calculating the durability of a Swift cluster is beyond the scope of this book, but the selected hardware needs to meet your durability requirements. For users requiring a high level of durability, low-density enterprise-class disk drives, servers with dual fans and power supplies, and so on are some considerations.

- **Availability:** Availability is defined as the percentage of time during which requests are successfully processed by the cluster. Availability mostly impacts frontend network architecture in terms of having a single network (that is, a single point of failure) versus dual-redundant networks. As mentioned earlier, networks in a given zone can be designed as single points of failure as long as your IT staff have the ability to troubleshoot them quickly.
- **Serviceability:** The serviceability of various pieces of hardware depends heavily on your strategy. If you choose fail-in-place (typically for large installations), serviceability is not a big concern. If you choose a repair/servicing strategy (typically for small and medium installations), serviceability is a concern. Each device should lend itself to repair or servicing. A smaller scale installation may also force the choice of more expensive hardware in terms of dual-redundant fans, power supplies, and so on. The reason is that, if there is a failure, there simply won't be many back-off devices available for the Swift ring to choose from.

- **Manageability:** As previously discussed, servers come in all different flavors when it comes to hardware management and associated software. You should choose servers with management features that match your overall IT strategy.

The vendor selection strategy

If you really want to be like a web giant, you should buy hardware from ODMs and other commodity hardware manufacturers (either directly or through a systems integrator). However, in reality, the decision is not that simple. The questions you need to ask yourself are as follows:

Question	"Yes" for all questions	"No" for even one question
Can you specify the configuration of each server, taking performance, durability, availability, serviceability, and manageability into account (versus needing vendor sales engineers to help)?	You are ready for commodity hardware!	You should stick to branded hardware
Can you self-support? That is, if you get a call at 2:00 a.m., are you prepared to root-cause what happened as opposed to calling the vendor?		
Are you prepared to accept less sophisticated warranty, lead times, end-of-life policies, and other terms?		
Can you live with minimal vendor-provided hardware management capabilities and software?		

Branded hardware

If you choose branded hardware, the process is fairly simple and involves issuing RFQs to your favorite server manufacturers, such as HP, Dell, IBM, and FTS, to networking manufacturers such as Cisco, Juniper, and Arista, and to JBOD manufacturers such as Seagate, DotHill, and Violin. You can then make the selection based on all the responses.

Commodity hardware

If you go this route, there are numerous manufacturers to consider — Taiwanese ODMs and other storage hardware specialists such as Xyratex and Sanmina. Perhaps the most interesting option to look at is an open source hardware movement called the **Open Compute Platform (OCP)**.

According to their website at <http://www.opencompute.org>, OCP's mission is to design and enable the delivery of the most efficient server, storage, and data center hardware designs for scalable computing. All of OCP's work is open source. A number of manufacturers sell OCP-compliant hardware, and this compliance makes it somewhat simpler for users to choose consistent hardware across manufacturers.

The OCP Intel motherboard hardware version 2.0, for example, supports two CPUs, four channels of DDR3 memory per CPU, a mini-SAS port for potential JBOD expansion, 1 GB per second network I/O, and a number of hardware management features. It can also accept a PCIe mezzanine NIC card for a 10 GB per second network I/O. This server can be suitable for both the proxy and storage server (with different items populated).

The OCP OpenVault JBOD, which is another example, is a 2U chassis that can hold up to 30 drives. This makes it a suitable companion for dense storage servers.

Summary

In this chapter, we looked at the complex process of selecting hardware for an OpenStack Swift installation and the various trade-offs that can be made. In the next chapter, we will look at how to benchmark and tune our Swift cluster.

9

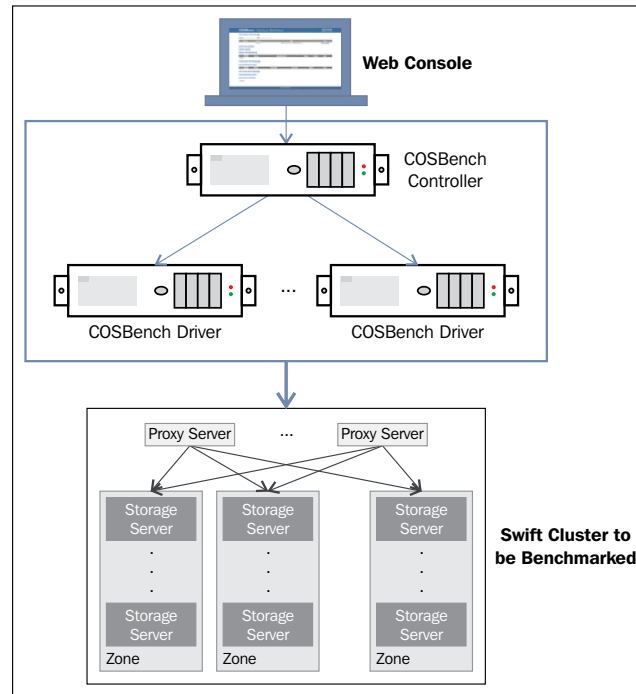
Tuning Your Swift Installation

OpenStack Swift's tremendous flexibility means that it has a very large number of tuning options. Therefore, users using Swift as a private cloud will need to tune their installation to optimize performance, durability, and availability for their unique workload. This chapter walks you through a performance benchmarking tool and the basic mechanisms available for tuning your Swift cluster.

Performance benchmarking

There are several tools that can be used to benchmark the performance of your Swift cluster against a specific workload. **COSBench**, **ssbench**, and **swift-bench** are the most popular tools available. This chapter discusses COSBench, given its completeness and the availability of graphical user interfaces with it.

COSBench is an open source distributed performance benchmarking tool for object storage systems. It has been developed and maintained by Intel. COSBench supports a variety of object storage systems, including OpenStack Swift. The physical configuration of COSBench is shown in the following diagram:



The key components of COSBench are as follows:

- Driver (also referred to as the COSBench driver or load generator):
 - Responsible for workload generation, issuing operations to target cloud object storage, and collecting performance statistics
 - In our test environment, the drivers were accessed via <http://10.27.128.14:18088/driver/index.html> and <http://10.27.128.15:18088/driver/index.html>
- Controller (also referred to as the COSBench controller):
 - Responsible for coordinating drivers to collectively execute a workload, collecting and aggregating runtime status and benchmark results from driver instances, and accepting workload submissions
 - In our environment, the controller was accessed via <http://10.27.128.14:19088/controller/index.html>

A critical thing to keep in mind as we start with COSBench is to ensure that the driver and controller machines do not inadvertently become performance bottlenecks. These nodes need to have adequate resources.

Next, the benchmark parameters are tied closely to your use case, and they need to be set accordingly. *Chapter 10, Additional Resources*, explores use cases in more detail, but a couple of benchmark-related examples are described here:

- **Audio file sharing and collaboration:** This is a use case where data is accessed often after being initially written. This is where you may want to set the ratio of read requests to write requests as relatively high, for example, 80 percent. The access rate for containers and objects may be relatively small (in tens of requests per second) with rather large objects (say, a size of hundreds of MB or larger per object).
- **Document archiving:** This is a somewhat cold data use case, where you may want to set a relatively low read-request-to-write-request ratio, for example, 5 percent. The access rate for containers and objects may be high (in hundreds of requests per second) with medium-size objects (say, a size of 5 MB per object).

Keep these use cases in mind as we proceed.

In our test setup, COSBench was installed on an Ubuntu 12.04.1 LTS operating system. The system also had JRE 1.6, unzip, cvstool, and cURL 7.22.0 installed prior to installing COSBench version 0.3.3.0 (<https://github.com/intel-cloud/cosbench/releases/tag/0.3.3.0>).

TCP ports 19088 on the controller and 18088 on the driver machines need to be free and accessible nonlocally. The installation is very simple; use the following easy steps:

```
cd ~
unzip 2.1.0.GA.zip
ln -s 2.1.0.GA/ cos
cd cos
chmod +x *.sh
```

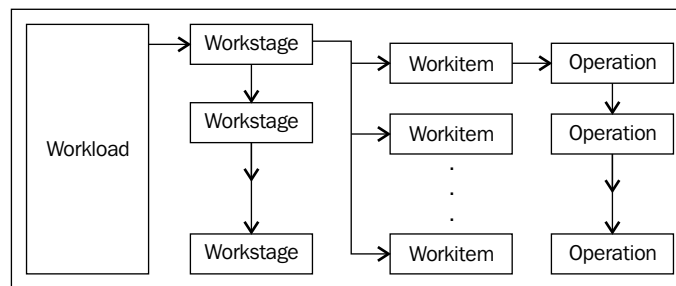
More details on the installation and validation that the software has been installed correctly can be obtained from the COSBench user guide located at <https://github.com/intel-cloud/cosbench>. With the installation of COSBench, the user gets access to a number of scripts. Some of these scripts are as follows:

- `start-all.sh` and `stop-all.sh`: Used to start and stop both the controller and the driver on the current node
- `start-controller.sh` and `stop-controller.sh`: Used to start and stop only the controller on the current node
- `start-driver.sh` and `stop-driver.sh`: Used to start and stop only the driver on the current node
- `cosbench-start.sh` and `cosbench-stop.sh`: These are internal scripts called by the preceding scripts
- `cli.sh`: Used to manipulate workload through the command line

The controller can be configured using the `conf/controller.conf` file (in the `cos` directory), and the driver can be configured using the `conf/driver.conf` file.

Drivers can be started on all driver nodes using the `start-driver.sh` script, while the controller can be started on the controller node using the `start-controller.sh` script.

Next, we need to create workloads. A **workload** can be considered one complete benchmark test. A workload consists of **workstages**. Each workstage consists of **work items**. Finally, work items consist of **operations**. A workload can have more than one workstage that is executed sequentially. A workstage can have more than one work item that is executed in parallel, as shown in the following diagram:



Workload composition

There is one normal type (`main`) and four special types (`init`, `prepare`, `cleanup`, and `dispose`) of work. The `main` type is where we will spend the rest of this discussion; the key parameters for this phase are as follows:

- The `workers` type is used to specify the number of workers used to conduct work in parallel, and thus control the load generated
- The `runtime` parameter (plus `rampup` and `rampdown`), `totalOps`, and `totalBytes` are used to control other parameters of the load generated, including how to start and end the work

The main phase has the `read`, `write`, and `delete` operations. You will typically want to specify the number of containers and objects to be written and the object sizes. Numbers and sizes are specified as expressions, and a variety of options, such as `constant`, `uniform`, and `sequential`, are available.

The workload is specified as an XML file. We will now create a workload that is fashioned after the document archiving use case discussed earlier. It uses a workload ratio of 95 percent writes and 5 percent reads. The drivers will spawn 128 workers for a duration of one hour. The object size is static at 5 MB, and 100 containers will be created. The workload is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<workload name="VEDAMS-UAT-V1-128W-5MB-Baseline"
description="VEDAMS UAT workload configuration">
<auth type="swauth" config=" ;password=xxxx;url= >username=8016-
2588:vedams-user@evault.com https://auth.vedams.com/v1.0"/
<storage type="swift" config=""/>
<workflow>
  <workstage name="init" closedelay="0">
    <work name="init" type="init" workers="16" interval="20"
division="container" runtime="0" rampup="0" rampdown="0"
totalOps="1" totalBytes="0" config="containers=r(1,32)">
      <operation type="init" ratio="100" division="container"
config="objects=r(0,0);sizes=c(0)B;containers=r(1,32)"
id="none"/>
    </work>
  </workstage>
  <workstage name="prepare" closedelay="0">
    <work name="prepare" type="prepare" workers="16"
interval="20"
division="object" runtime="0" rampup="0" rampdown="0"
totalOps="1" totalBytes="0"
config="containers=r(1,32);objects=r(1,50);
sizes=u(5,5)MB">
      <operation type="prepare" ratio="100" division="object"
config="createContainer=false;containers=r(1,32);
objects=r(1,50);sizes=u(5,5)MB" id="none"/>
    </work>
  </workstage>
```



```
<workstage name="normal" closedelay="0">
  <work name="normal" type="normal" workers="128"
    interval="20"
    division="none" runtime="300" rampup="100" rampdown="0"
    totalOps="0" totalBytes="0">
    <operation type="read" ratio="5" division="none"
      config="containers=u(1,32);objects=u(1,50);" id="none"/>
    <operation type="write" ratio="95" division="none"
      config="containers=u(1,32);objects=u(51,100);
        sizes=u(5,5)MB"
      id="none"/>
    </work>
  </workstage>
  <workstage name="cleanup" closedelay="0">
    <work name="cleanup" type="cleanup" workers="16"
      interval="20"
      division="object" runtime="0" rampup="0" rampdown="0"
      totalOps="1" totalBytes="0"
      config="containers=r(1,32);objects=r(1,100);">
    <operation type="cleanup" ratio="100" division="object"
      config="deleteContainer=false;containers=r(1,32);
        objects=r(1,100);" id="none"/>
    </work>
  </workstage>
  <workstage name="dispose" closedelay="0">
    <work name="dispose" type="dispose" workers="16"
      interval="20"
      division="container" runtime="0" rampup="0" rampdown="0"
      totalOps="1" totalBytes="0" config="containers=r(1,32);">
    <operation type="dispose" ratio="100"
      division="container"
      config="objects=r(0,0);sizes=c(0)B;containers=r(1,32);"
      id="none"/>
    </work>
  </workstage>
</workflow>
</workload>
```

The result of this workload is a series of reported metrics: throughput as measured by operations per second, response time measured by the average duration between operation start and end, bandwidth as measured by MB per second, success ratio (percentage successful), and other metrics. A sample COSBench report is shown in the following screenshot:

Final Result						
General Report						
Op-Type	Op-Count	Byte-Count	Avg-ResTime	Throughput	Bandwidth	Succ-Ratio
init-write	0 ops	0 B	N/A	0 op/s	0 B/S	N/A
prepare-write	1.6 kops	102.4 MB	142.26 ms	7.02 op/s	449.5 KB/S	100%
read	4.21 kops	269.31 MB	20.84 ms	14.03 op/s	898.03 KB/S	100%
write	16.84 kops	1.08 GB	137.17 ms	56.16 op/s	3.59 MB/S	100%
cleanup-delete	3.2 kops	0 B	89.5 ms	11.17 op/s	0 B/S	100%
dispose-delete	0 ops	0 B	N/A	0 op/s	0 B/S	N/A

[show performance details](#)

A sample of the performance result

If the Swift cluster under test stands up to your workload, you are done. You may want to perform some basic tuning, but this is optional. However, if the Swift cluster is unable to cope with your workload, you need to perform tuning.

The first step is to identify bottlenecks. See *Chapter 6, Monitoring and Managing Swift*, for tools used to find performance bottlenecks. Nagios or Swift Recon may be particularly well suited to this. Of course, simple tools such as **top** may be used as well. Once you isolate the bottleneck from a particular server (or servers) and the underlying components such as CPU performance, memory, I/O, disk bandwidth, and response times, you can move to the next step, which is tuning.

Hardware tuning

Chapter 8, Choosing the Right Hardware, discusses the hardware considerations in great detail. It is sufficient to say that choosing the right hardware can have a huge impact on your performance, durability, availability, and cost.

Software tuning

In *Chapter 2, OpenStack Swift Architecture*, we talked about Swift using two types of software modules—data path (referred to as **WSGI** servers in Swift documentation) and post-processing (referred to as **background daemons**). In addition, there is the ring. All three merit different considerations in terms of software tuning. Also, we will briefly look at some additional tuning considerations.

Ring considerations

The number of partitions in a ring affects performance, and it needs to be chosen carefully because it cannot be changed easily. Swift documentation recommends a minimum of 100 partitions per drive to ensure even distribution across servers.

Taking the maximum anticipated number of drives multiplied by 100 and then rounded off to the nearest power of 2 provides the minimum number of total partitions. Using a number higher than needed will mean extremely uniform distribution, but at the cost of performance, since more partitions put a higher burden on replicators and other post-processing jobs. Therefore, users should not overestimate the ultimate size of the cluster.

For example, let's assume that we expect our cluster to have a maximum of 1,000 nodes, each with 60 disks. This gives us $60 \times 1,000 \times 100 = 6,000,000$ partitions. Rounded off to the nearest power of two, we get $2^{23} = 8,388,608$. The value that will be used to create the ring will therefore be 23. We did not discuss disk size in this computation, but note that a cluster with smaller/faster disks (for example, a 2 TB SAS drive) will perform better than a cluster with larger/slower disks (for example, a 6 TB SATA drive) with the same number of partitions.

Data path software tuning

The key data path software modules are proxy, account, container, and object servers. There are literally dozens of tuning parameters, but the four most important parameters in terms of performance impact are as follows:

Parameter	Proxy server	Storage server
workers (auto by default)	Each worker can handle <code>max_clients</code> number of simultaneous requests. Ideally, more workers means more requests can be handled without being blocked. However, there is an upper limit dictated by the CPU. Start by setting <code>workers</code> as 2 multiplied by the number of cores. If the storage server includes account, container, and object servers, you may have to perform some experiments.	
<code>max_clients</code> (1024 by default)	Since we want the most effective use of network capacity, we want a large number of simultaneous requests. You probably won't need to change the default setting.	In data published by Red Hat, filesystem calls were found to block an entire worker. This means that a very large setting for <code>max_clients</code> is not useful. Experiment with this parameter, and don't be afraid to reduce this number all the way down to match <code>threads_per_disk</code> or even to 1.

Parameter	Proxy server	Storage server
<code>object_chunk_size</code> (64 KB by default)	Given that this data is flowing over an internal Swift network, a larger setting may be more efficient. Red Hat found 2 MB to be more efficient than the default size when using a 10 GB per second network.	N/A
<code>threads_per_disk</code> (0 by default)	N/A	This parameter defines the size of the per-disk thread pool. A default value of 0 means a per-disk thread pool will not be used. In general, the Swift documentation recommends keeping this small to avoid large queue depths, which result in high read latencies. Try starting with four threads per disk.

Post-processing software tuning

The impact of tuning post-processing software is very different from data path software. The focus is not so much on servicing API requests but rather on reliability, performance, and consistency.

Increasing the rate of operations for replicators and auditors makes the system more durable, since this reduces the time required to find and fix faults, but at the expense of increased server load. Also, increasing the auditor rate reduces consistency windows by putting a higher server load. The following are the parameters to consider:

- **Concurrency:** Swift documentation (http://docs.openstack.org/developer/swift/deployment_guide.html) recommends setting the concurrency of most post-processing jobs at 1, except for replicators, where they recommend 2. If you need a higher level of durability, consider increasing this number. Again, durability is measured by $1-P$ (object loss in a year), where the object size is typically 1 KB.
- **Interval:** Unless you want to reduce the load on servers, increase reliability, or reduce consistency windows, you will probably want to stick to the default value.

Considering post-processing softwares, the object auditor is the most expensive background job. The following parameters are also worth tuning:

- `bytes_per_second`: This defines the maximum number of bytes audited per second (0 means unlimited). The default is set to 10,000,000 bytes. As per other parameters, this should be increased for higher durability but reduced for busy clusters where you do not want the auditor to kill your I/O performance.
- `files_per_second`: This defines the maximum number of files audited per second (0 means unlimited). The default is set to 20 files, which means that Swift assumes that your files are 500 KB on an average. If you change `bytes_per_second` or if your average file size is different, it may be worth tuning this parameter.
- `disk_chunk_size`: This defines the size of chunks read during auditing. It is set to 65,536 by default. Increasing this number will reduce the total CPU load at the expense of larger reads.

Additional tuning parameters

A number of additional tuning parameters are available for the user. The important parameters are listed here:

- **Memcached**: A number of Swift services rely on memcached for caching lookups, since Swift does not cache any object data. While memcached can be run on any server, it should be turned on for all proxy servers. If memcached is turned on, ensure that adequate RAM and CPU resources are available.
- **System time**: Given that Swift is a distributed system, the timestamp of an object is used for a number of reasons. Therefore, it is important to ensure that time is consistent between servers. Services such as NTP must be used for this purpose.
- **Filesystem**: Swift is filesystem agnostic; however, XFS is the one tested by the Swift community. It is important to keep a high `inode` size, 1024 for example, to ensure that the default and some additional metadata can be stored efficiently. Other parameters should be set as described in *Chapter 3, Installing OpenStack Swift*.

- **Operating system:** General operating system tuning is beyond the scope of this book. However, the Swift documentation suggests disabling `TIME_WAIT` and `syn cookies` and doubling the amount of `conntrack` allowed in `sysctl.conf`. Since the OS is usually installed on a disk that is not part of the storage drives, you may want to consider a small SSD to get short boot times.
- **Network stack:** Network stack tuning is also beyond the scope of this book. However, there may be some obvious tuning, for example, enabling jumbo frames for the internal storage cluster network. Jumbo frames may also be enabled on the client-facing or replication network if this traffic is over the LAN (in the case of private clouds).
- **Logging:** Unless custom log handlers are used, Swift logs directly to `syslog`. Swift generates a large amount of log data, and so managing logs correctly is extremely important. Setting logs appropriately can impact both performance and your ability to diagnose problems. You may want to consider high performance variants such as **rsyslog** (<http://www.rsyslog.com/>) or **syslog-ng** (<http://www.balabit.com/network-security/syslog-ng/opensource-logging-system>).

Summary

In this chapter, we reviewed how to benchmark a Swift cluster and tune it for a specific use case for private cloud users. The next, and final, chapter covers use cases that are appropriate for OpenStack Swift and additional resources.

10

Additional Resources

Having acquired the know-how on building, managing, and tuning OpenStack clusters by reading the preceding chapters, you are now ready to join the global elite group of OpenStack Swift experts and take your career to the next level. Let's now explore a few use cases of OpenStack Swift and get pointers to useful resources.

Use cases

There are five major use cases for OpenStack Swift:

- Archival
- Backup
- Content repository
- Collaboration
- Data lakes

The first three use cases cut capital and operational costs, while the last two use cases improve the business outcome.

Archival

The default answer for archival has been **tapes**. However, the availability of data on tapes is poor, from minutes to days. For data that needs to be made available to users instantly, Swift is a great archival solution. With the right hardware choice, the cost of a Swift cluster can be substantially lower than block or file storage, and yet the data can be made available at any given time.

Swift also provides greater durability through background data integrity checks. Over time, more and more archival tools are interfacing directly with REST APIs such as Swift. These tools can scan data on primary storage and, based on policies, move them to Swift. Sophisticated archival tools can leave stubs as well so that users get a seamless experience.

Backup

Large backup users find that a full backup is difficult to complete in a meaningful window of time with tapes. Users find themselves adding more libraries with more drives just to get performance, which becomes expensive. Furthermore, it is not possible to dedupe data on tapes. Finally, it is not possible to ensure durability of data on tapes. For these reasons, backup users are increasingly looking at object storage such as in Swift, where costs, performance, durability, and dedupe can all be optimized to the users' needs.

Content repository

Object storage as a content repository was an early use case of Amazon S3, where S3 was used to serve web and streaming content. Swift can serve this very purpose for on-premise cloud storage. Any file type can be stored on Swift, and web applications can serve content from Swift given the easy-to-use REST APIs. With middleware such as the Swift origin server and the static web server described in *Chapter 2, OpenStack Swift Architecture*, this operation becomes even easier. Swift is less expensive than traditional file or block storage. It is also more scalable and compatible with web applications.

Collaboration

Network-attached storage (NAS) has traditionally been used to share data. However, NAS is difficult to use with a large number of users, especially if they are spread out geographically. If there are users outside the organization, it becomes even more difficult to use NAS to share data. All of these problems are solved with Swift. Swift scales elegantly to handle a large number of users. Supporting users around the world is not a problem, even if they are outside the organization. The collaboration use case helps bring products to the market faster, as opposed to the first three where cost savings are the main benefit.

Data lakes

Data lakes are another emerging use case, where a large amount of data can be stored in Swift for analytics purposes and a Hadoop cluster can run jobs against this data. Imagine a hospital storing medical images in a data lake. Now the hospital can run analytics against this data, for example, correlating all images of a certain type.

Operating systems used for OpenStack implementations

OpenStack supports a variety of operating systems, and we have compiled a table listing the operating systems used in some OpenStack implementations.

The following table provides information on organizations using these operating systems in their implementations:

Operating system	Implementation/organization	Links
Ubuntu	NeCTAR, MercadoLibre, Intel, Opscode, Liveperson, NTT, NEC, Time Warner Cable, and Telekom Deutschelands	<ul style="list-style-type: none">• https://www.openstack.org/user-stories/nectar/• https://www.openstack.org/user-stories/mercadolibre/• https://www.openstack.org/summit/openstack-summit-hong-kong-2013/session-videos/presentation/openstack-deployment-with-chef-workshop• https://www.openstack.org/summit/openstack-summit-hong-kong-2013/session-videos/presentation/is-open-source-good-enough-a-deep-study-of-swift-and-ceph-performance• https://www.openstack.org/summit/openstack-summit-hong-kong-2013/session-videos/presentation/liveperson-openstack-case-study-from-0-to-100-in-1-year• https://www.openstack.org/summit/openstack-summit-atlanta-2014/session-videos/presentation/canonical-keynote
Red Hat	CERN	<ul style="list-style-type: none">• https://www.openstack.org/user-stories/cern/
CentOS	Workday	<ul style="list-style-type: none">• https://www.openstack.org/summit/openstack-summit-hong-kong-2013/session-videos/presentation/workday-on-openstack
HP Cloud OS	HP	<ul style="list-style-type: none">• https://www.openstack.org/summit/openstack-summit-hong-kong-2013/session-videos/presentation/is-open-source-good-enough-a-deep-study-of-swift-and-ceph-performance

Virtualization used for OpenStack implementations

OpenStack services can be installed on virtual machines created using ESX, KVM, Hyper-V, and so on. The following table lists the virtualization technology used in a few implementations:

Virtualization	Implementation/organization	Links
KVM	eNovance, Workday, CERN, Canonical, Metacloud Inc. (Cisco), Nuage Network, Numergy, and eBay	<ul style="list-style-type: none"> • https://www.openstack.org/summit/openstack-summit-hong-kong-2013/session-videos/presentation/ceph-the-de-facto-storage-backend-for-openstack • https://www.openstack.org/summit/openstack-summit-hong-kong-2013/session-videos/presentation/workday-on-openstack • https://www.openstack.org/user-stories/cern/ • https://www.openstack.org/summit/openstack-summit-atlanta-2014/session-videos/presentation/canonical-keynote • https://www.openstack.org/summit/openstack-paris-summit-2014/session-videos/presentation/under-the-hood-with-nova-libvirt-and-kvm-part-two • https://www.openstack.org/summit/openstack-paris-summit-2014/session-videos/presentation/sponsor-demo-theater-nuage-networks-deployable-neutron-networking • https://www.openstack.org/summit/openstack-paris-summit-2014/session-videos/presentation/hybrid-your-cloud-with-numergy-and-nuage • https://www.openstack.org/summit/openstack-paris-summit-2014/session-videos/presentation/seamless-migration-from-nova-network-to-neutron-in-ebay-production

Virtualization	Implementation/ organization	Links
VMware, VMware NSX	VMWare, Canonical, SUSE, and Red Hat	<ul style="list-style-type: none">• https://www.openstack.org/summit/openstack-summit-hong-kong-2013/session-videos/presentation/hands-on-with-openstack-vmware• https://www.openstack.org/summit/openstack-summit-atlanta-2014/session-videos/presentation/vmware-openstack-accelerating-openstack-in-the-enterprise

Provisioning and distribution tools

The most common provisioning and deployment tools used to deploy OpenStack are Puppet, Chef, Ansible, Saltstack, and Juju. This table lists the tools and some OpenStack installations that they are used in:

Provisioning/ deployment	Implementation/ organization	Links
Puppet	CERN, NeCTAR, Kickstart, Cisco Webex, Liveperson, Expedia, Intel, and Comcast	<ul style="list-style-type: none">• https://www.openstack.org/user-stories/cern/• https://www.openstack.org/user-stories/nectar/• https://www.openstack.org/summit/openstack-summit-hong-kong-2013/session-videos/presentation/kickstack-rapid-openstack-deployment-with-puppet• https://www.openstack.org/user-stories/cisco-webex/• http://www.openstack.org/user-stories/liveperson/• https://www.openstack.org/summit/openstack-paris-summit-2014/session-videos/presentation/openstack-in-a-hybrid-world• https://www.openstack.org/summit/openstack-paris-summit-2014/session-videos/presentation/what-should-we-take-into-consideration-to-build-a-production-openstack-cloud• https://www.openstack.org/summit/openstack-paris-summit-2014/session-videos/presentation/ci-cd-in-practice-real-world-deployment-and-management

Provisioning/ deployment	Implementation/ organization	Links
Chef	Workday, Opscode, MercadoLibre, and Red Hat	<ul style="list-style-type: none"> • https://www.openstack.org/summit/openstack-summit-hong-kong-2013/session-videos/presentation/workday-on-openstack • https://www.openstack.org/summit/openstack-summit-hong-kong-2013/session-videos/presentation/openstack-deployment-with-chef-workshop • https://www.openstack.org/user-stories/mercadolibre/ • https://www.openstack.org/summit/openstack-summit-atlanta-2014/session-videos/presentation/demo-theater-inktank-ceph-update-erasure-coding-and-tiering
Juju	VMWare, AMD, Microsoft, Juniper, NEC, HPCC systems, HP, NTT, MongoDB, Joyent, Cisco, TeleStax, Ericsson, and IBM	<ul style="list-style-type: none"> • https://www.openstack.org/summit/openstack-summit-hong-kong-2013/session-videos/presentation/vmware-and-openstack-bridging-the-divide-using-ubuntu-and-juju • https://www.openstack.org/summit/openstack-summit-atlanta-2014/session-videos/presentation/canonical-keynote
Compass	Huawei	<ul style="list-style-type: none"> • https://www.openstack.org/summit/openstack-summit-hong-kong-2013/session-videos/presentation/compass-yet-another-openstack-deployment-system

Monitoring and graphing tools

The following table lists tools that can be used in addition to OpenStack Swift to enable monitoring (some of them are mentioned in earlier chapters too):

Monitoring tool	Download link	Organization implementations
Groundwork	http://sourceforge.net/projects/gwmos/	NeCTAR: http://www.openstack.org/user-stories/nectar/
Ganglia: graphing tool	http://sourceforge.net/apps/trac/ganglia/wiki/ganglia_quick_start	CERN: https://www.youtube.com/watch?v=jRkTVh27XBQ
Graphite	https://github.com/etsy/statsd/blob/master/docs/graphite.md	Rackspace: https://www.openstack.org/summit/openstack-summit-hong-kong-2013/session-videos/presentation/an-intimate-look-at-running-openstack-swift-at-scale
Zabbix	http://www.patlathem.com/zabbix-beginners-guide-installing-and-configuring-the-monitoring-server/	
Nagios	http://www.nagios.org/download	Red Hat, Mirantis, and Dell crowbar

Additional information

These links provide additional information on OpenStack Swift:

- <http://swift.openstack.org>
- <https://github.com/openstack/swift>

The following blog provides more up-to-date information on the topics discussed in this chapter. It also provides more updated user stories, OpenStack implementations by customers, deployment tools, monitoring and graphing tools, and more information related to OpenStack implementations:

- <http://www.vedams.com/blog/>

Additional support, including mailing lists, is available at the following links, and users have the ability to review previously answered questions or post new questions to the community via a launch pad:

- <http://www.openstack.org/community/>
- <http://www.openstack.org/blog/>
- <http://webchat.freenode.net/>
- <https://swiftstack.com/blog/>
- <https://launchpad.net/swift>
- <https://www.mail-archive.com/openstack@lists.openstack.org/>

Summary

As we can see from our discussion in this chapter, OpenStack Swift is relevant to every user segment, from the individual consumer to the large service provider, for a large variety of use cases.

At this point, we hope you have a good idea of what cloud storage is and how OpenStack can be used to create cloud storage. We also hope you are confident in terms of how to install, manage, and use OpenStack Swift, including some finer points such as hardware selection and performance tuning. It is now time to get involved with the OpenStack Swift community as a user, contributor, or evangelist.

Swift CLI Commands

This appendix provides details on the set of commands that can be run from a Swift CLI session. These commands can be used to perform CRUD operations.

Commands

The commands that can be run from the Swift CLI are `list`, `stat`, `post`, `upload`, `download`, and `delete`. Each command has a detailed help menu, which can be displayed by running the `swift <command> -h` command, for example, `swift list -h`.

list

The `list` command is used to list the containers for an account or the objects for a container. The following code shows the usage of the `list` command along with arguments:

```
# swift list <container>
-A Auth_URL
-U User -K Key
--os-username=<auth-user-name>
--os-password=<auth-password>
--os-tenant-id=<auth-tenant-id>
--os-tenant-name=<auth-tenant-name>
--os-auth-url=<auth-url>
--os-auth-token=<auth-token>
--os-storage-url=<storage-url>
```

```
--os-region-name=<region-name>
--os-service-type=<service-type>
--os-endpoint-type=<endpoint-type>
--prefix=PREFIX
--os-cacert=<ca-certificate>
--insecure --no-ssl-compression
--long --totals
--delimiter=DELIMITER
```

Examples

You can list containers with size information using these commands:

```
# swift -V 2.0 -A https://auth.vedams.com/v2.0 -U admin:user1 -K t1
list --lh
# swift -V 2.0 -A https://auth.vedams.com/v2.0 -U admin:user1 -K t1
list --long
```

You can list the objects within a container using the following command:

```
# swift -V 2.0 -A https://auth.vedams.com/v2.0 -U admin:user1 -K t1
list con1
```

You can list containers with size information and a prefix of `con1` using this command:

```
# swift -V 2.0 -A https://auth.vedams.com/v2.0 -U admin:user1 -K t1
list --lh --prefix con1
```

stat

The `stat` command is used to display information about an account, a container, or an object. The following code shows the usage of the `stat` command along with arguments:

```
# swift stat <container> <object>
-A Auth_URL -U User -K Key
--os-username=<auth-user-name>
--os-password=<auth-password>
--os-tenant-id=<auth-tenant-id>
```

```
--os-tenant-name=<auth-tenant-name>
--os-auth-url=<auth-url>
--os-auth-token=<auth-token>
--os-storage-url=<storage-url>
--os-region-name=<region-name>
--os-service-type=<service-type>
--os-endpoint-type=<endpoint-type>
--os-cacert=<ca-certificate> --insecure --no-ssl-compression --lh
```

Examples

You can display the metadata of the account using this command:

```
# swift -V 2.0 -A https://auth.vedams.com/v2.0 -U admin:user1 -K t1
stat
```

You can display the metadata of the `container2` container using the following command:

```
# swift -V 2.0 -A https://auth.vedams.com/v2.0 -U admin:user1 -K t1
stat container2
```

You can display the metadata of the `key.txt` object in the `container3` container using this command:

```
# swift -V 2.0 -A https://auth.vedams.com/v2.0 -U admin:user1 -K t1
stat container3 key.txt
```

Finally, you can display the metadata of the account in the `regionOne` region in long format with totals, using the following command:

```
# swift -V 2.0 -A https://auth.vedams.com/v2.0 -U admin:user1 -K t1
stat --lh --os-region-name=regionOne
```

post

The `post` command is used to update metadata information about the account, container, or object. The following code shows the usage of the `post` command along with arguments:

```
# swift post <container> <object>
--read-acl <acl>
--write-acl <acl>
```

```
--meta <name:value>
--header <header>
-A Auth_URL -U User -K Key
--os-username=<auth-user-name>
--os-password=<auth-password>
--os-tenant-id=<auth-tenant-id>
--os-tenant-name=<auth-tenant-name>
--os-auth-url=<auth-url>
--os-auth-token=<auth-token>
--os-storage-url=<storage-url>
--os-region-name=<region-name>
--os-service-type=<service-type>
--os-endpoint-type=<endpoint-type>
--os-cacert=<ca-certificate>
--insecure --no-ssl-compression
--read-acl=READ_ACL
--write-acl=WRITE_ACL
--sync-to=SYNC_TO
--sync-key=SYNC_KEY
--meta=META
```

Examples

You can update the `read-acl` option for the `container1` container using this command:

```
# swift -V 2.0 -A https://auth.vedams.com/v2.0 -U admin:user1 -K t1
post container1 --read-acl=account1
```

You can add the `Size:Large` and `Color:Blue` metadata to the `container2` container using the following command:

```
# swift -V 2.0 -A https://auth.vedams.com/v2.0 -U admin:user1 -K t1
post container2 -m Size:Large -m Color:Blue
```

You can update the `content-type` header as `text/plain` for the `container3` container using this command:

```
# swift -V 2.0 -A https://auth.vedams.com/v2.0 -U admin:user1 -K t1
post container3 -H "content-type:text/plain"
```

You can update the `read-acl` metadata of the `container1` container by accessing the Swift cluster through the `regionOne` region, using the following command:

```
# swift -V 2.0 -A https://auth.vedams.com/v2.0 -U admin:user1 -K t1
post container4 --read-acl=account1 --os-region-name=regionOne
```

Finally, you can update the storage policy to `regular` for `container5` using this command:

```
# swift -V 2.0 -A https://auth.vedams.com/v2.0 -U admin:user1 -K t1
post container5 -H "X-Storage-Policy: regular"
```

upload

The `upload` command is used to upload the specified files and directories to the given container. The following code shows the usage of the `upload` command along with arguments:

```
# swift upload <container> <file_or_directory>
--changed --segment-size <size>
--segment-container <container>
--leave-segments --header <header>
-A Auth_URL -U User -K Key
--os-username=<auth-user-name>
--os-password=<auth-password>
--os-tenant-id=<auth-tenant-id>
--os-tenant-name=<auth-tenant-name>
--os-auth-url=<auth-url>
--os-auth-token=<auth-token>
--os-storage-url=<storage-url>
--os-region-name=<region-name>
--os-service-type=<service-type>
--os-endpoint-type=<endpoint-type>
--os-cacert=<ca-certificate>
--insecure --no-ssl-compression --changed
--skip-identical --segment-size=SEGMENT_SIZE
--segment-container=SEGMENT_CONTAINER
--object-threads=OBJECT_THREADS
--segment-threads=SEGMENT_THREADS
--header=HEADER --use-slo --object-name=OBJECT_NAME
```

Examples

You can upload the `key.txt` object to the `container1` container using the following command:

```
# swift -V 2.0 -A https://auth.vedams.com/v2.0 -U admin:user1 -K t1
upload container1 key.txt
```

You can upload multiple objects (`key1.txt`, `key2.txt`, and `key3.txt`) to the `container1` container using this command:

```
# swift -V 2.0 -A https://auth.vedams.com/v2.0 -U admin:user1 -K t1
upload container1 key1.txt key2.txt key3.txt
```

Upload the `key.txt` object to the `container2` container, using a segment size (`segment-size`) of 100 bytes. Swift has an object size limit of 5 GB by default. Larger files can be uploaded using the `segment-size` option. The object will be stored as multiple segments in the Swift object store.

In this example, each segment created will be of 100 bytes, and there will be several such segments uploaded, based on the size of the object. The `--changed` option is used to upload the file only if that file has changed from the time it was last uploaded, like this:

```
# swift -V 2.0 -A https://auth.vedams.com/v2.0 -U admin:user1 -K t1
upload container2 key.txt --changed --segment-size=100
```

Upload the `key.txt` object to the `container3` container using a segment size (`segment-size`) of 100 bytes, by using the following command. The command also explicitly specifies the `seg_container3` segment folder, where the segments will be uploaded:

```
# swift -V 2.0 -A https://auth.vedams.com/v2.0 -U admin:user1 -K t1
upload container3 key.txt --segment-size=100 --segment-
container=seg_container3
```

Upload the `key.txt` object to the `container2` container using a segment size (`segment-size`) of 100 bytes. The `use-slo` option is specified to create a static large object instead of the default dynamic large object, as shown in the following command:

```
# swift -V 2.0 -A https://auth.vedams.com/v2.0 -U admin:user1 -K t1
upload container4 key.txt --segment-size=100 --use-slo --os-region-
name=regionOne
```

Upload the `myfile.txt` object to `container5`:

```
# swift -V 2.0 -A https://auth.vedams.com/v2.0 -U admin:user1 -K t1
upload container5 myfile.txt
```

download

The download command is used to download objects from containers. The following code shows the usage of the download command along with arguments:

```
# swift download <container> <object>
--all --prefix <prefix> --output <out_file>
-A Auth_URL -U User -K Key
--os-username=<auth-user-name>
--os-password=<auth-password>
--os-tenant-id=<auth-tenant-id>
--os-tenant-name=<auth-tenant-name>
--os-auth-url=<auth-url>
--os-auth-token=<auth-token>
--os-storage-url=<storage-url>
--os-region-name=<region-name>
--os-service-type=<service-type>
--os-endpoint-type=<endpoint-type>
--os-cacert=<ca-certificate>
--insecure --no-ssl-compression
--marker=MARKER
--object-threads=OBJECT_THREADS
--container-threads=CONTAINER_THREADS --no-download
--header=HEADER --skip-identical
```

Examples

Download all objects from all the containers using this command:

```
# swift -V 2.0 -A https://auth.vedams.com/v2.0 -U admin:user1 -K t1
download --all
```

Download all objects with the key prefix from the container1 container using the following command:

```
# swift -V 2.0 -A https://auth.vedams.com/v2.0 -U admin:user1 -K t1
download container1 --prefix key
```

Download the key.txt object from the container1 container using this command:

```
# swift -V 2.0 -A https://auth.vedams.com/v2.0 -U admin:user1 -K t1
download container1 key.txt
```


Download all objects from all the containers by utilizing two threads for object downloads, using the following command:

```
# swift -V 2.0 -A https://auth.vedams.com/v2.0 -U admin:user1 -K t1
download --all --object-threads 2
```

delete

The `delete` command is used to delete a container or objects within a container. The following code shows the usage of the `delete` command along with arguments:

```
# swift delete <container> <object> --all --leave_segments
-A Auth_URL -U User -K Key
--os-username=<auth-user-name>
--os-password=<auth-password>
--os-tenant-id=<auth-tenant-id>
--os-tenant-name=<auth-tenant-name>
--os-auth-url=<auth-url>
--os-auth-token=<auth-token>
--os-storage-url=<storage-url>
--os-region-name=<region-name>
--os-service-type=<service-type>
--os-endpoint-type=<endpoint-type>
--os-cacert=<ca-certificate> --insecure --no-ssl-compression
--object-threads=OBJECT_THREADS
--container-threads=CONTAINER_THREADS
```

Examples

Delete the `key.txt` object from the `container1` container using this command:

```
# swift -V 2.0 -A https://auth.vedams.com/v2.0 -U admin:user1 -K t1
delete container1 key.txt
```

Delete all objects from the `container2` container, including the container, and leave the segments as they are:

```
# swift -V 2.0 -A https://auth.vedams.com/v2.0 -U admin:user1 -K t1
delete container2 --leave-segments
```

Delete all objects and all containers using the following command:

```
# swift -V 2.0 -A https://auth.vedams.com/v2.0 -U admin:user1 -K t1  
delete --all
```

Delete all objects and all containers by utilizing two threads for deleting objects, using this command:

```
# swift -V 2.0 -A https://auth.vedams.com/v2.0 -U admin:user1 -K t1  
delete --all --object-threads=2
```


Index

A

access control lists (ACL) 18

Amazon S3 API

- compatibility 57, 58
- S3 commands, using 59

Apache jclouds library

- URL 60

architectural principles, OpenStack Swift

- available and eventually consistent 11
- heterogeneous 11
- load spreading 10
- loose coupling 10
- masterless design 10
- multi-tenancy layer 10
- self-healing 10

B

background daemons 117

basic commands, for Docker user

- about 92
- storage container, setting up with
 - Docker image 94, 95
- Swift proxy container, setting up with
 - Docker image 93, 94

C

capacity planning

- about 84
- drives, adding 84
- storage and proxy servers, adding 85

CAP theorem

- about 4

availability 4

consistency 4

tolerance to partial failures 4

client installation

- cURL command 45
- specialized REST API client 45
- Swift client CLI 45

client libraries

- Java 60
- Python 61
- Ruby 61
- used, for accessing OpenStack Swift 60

cloud storage

- about 1
- elements 2
- limitations 3

cloud storage, limitations

- about 3
- new APIs 4
- performance 3, 4

containers

- about 9
- cURL, using 50
- metadata, updating 52
- objects, listing 50
- Swift client CLI, using 49
- using 49

COSBench tool

- about 111
- controller 112
- driver 112
- URL 113, 114

cross-origin resource sharing (CORS) 21

cURL command 45

D

data path software servers

- account server 14
- container server 14
- create operation 14, 15
- delete operation 16
- object server 14
- proxy server 14
- read operation 15
- storage server 13
- update operation 16

data path software tuning

- max_clients parameter 118
- object_chunk_size parameter 119
- threads_per_disk parameter 119
- workers parameter 118

delete command

- about 140
- examples 140, 141

Docker

- about 89, 90
- basic user commands 91
- installation 91
- OpenStack, using with 90
- user, basic commands 91, 92

Dockerfile

- used, for creating proxy container 96, 97
- used, for creating storage container 97
- used, for setting up Swift cluster 96

Docker image

- used, for setting up storage container 94, 95
- used, for setting up Swift proxy container 93, 94

download command

- about 139
- examples 139, 140

E

elements, cloud storage

- availability 3
- data durability 3
- elastic 2
- multitenancy 3
- on-demand 2

- reduced total cost of ownership (TCO) 2
- universal access 3
- unlimited scalability 2

environment variables

- about 52
- container ACLs 54-56
- OS_AUTH_URL 52
- OS_PASSWORD 52
- OS_TENANT_NAME 52
- OS_USERNAME 52
- pseudo-hierarchical directories 53

Extent file system (XFS) 13

F

failure management

- about 81
- drive failure, handling 82
- drive failures, detecting 82
- node failure, handling 83
- region failure 84
- zone failure 84

features, OpenStack Swift

- cluster health 21
- cross-origin resource sharing (CORS) 21
- large object support 20
- metadata 21
- multirange support 21
- server-side copies 21

H

Hadoop 8545 66

Hadoop cluster

- setting up, with Sahara 66

hardware prerequisites

- account servers 100
- auth servers 100
- container servers 100
- firewall and security appliance 100
- JBODs 100
- load balancer / SSL acceleration 100
- network switch (or switches) 99
- on-premise cloud gateway 100
- proxy server (or servers) 99
- storage servers 99

hardware, OpenStack Swift installation
 planning 23, 24
HEAD method 21

I

inline middleware options
 about 18
 authentication 18
 modules 19

J

Java Script Object Notation (JSON) 76
just a bunch of disks (JBOD) 12, 100

K

Keystone authentication
 used, for creating token 46
Keystone service
 about 32
 installing 33-37
 MariaDB, installing 32

L

LAN-on-motherboard (LOM) 101
large objects
 transferring 56, 57
list command
 about 133
 examples 134
logical organization, objects
 about 9
 objects 9
 pseudo-directories 10
 tenant 9
 users 10

M

metadata, containers
 REST API, using 52
 Swift client CLI, using 52
 updating 52

metadata information, for account
 displaying 47
 displaying, with cURL 48
 displaying, with specialized REST API
 client 48, 49
 displaying, with Swift client CLI 47

metadata information, for container
 displaying 47
 displaying, with cURL 48
 displaying, with specialized REST API
 client 49
 displaying, with Swift client CLI 47

metadata information, for object
 displaying 47
 displaying, with cURL 48
 displaying, with specialized REST API
 client 48, 49
 displaying, with Swift client CLI 47

migrations 85, 86

monitoring tools, OpenStack Swift
 Ganglia 130
 Graphite 130
 Groundwork 130
 Nagios 130
 Zabbix 130

N

Nagios
 URL 70

network-attached storage (NAS) 125

**network configuration, OpenStack Swift
 installation**
 about 24
 Keystone service 32
 multiregion support 39
 proxy server node, setting up 31
 public network 24
 replication network 25
 ring setup 37
 storage network 25
 storage server nodes, setting up 27

network interface card (NIC) 101

Network Time Protocol (NTP) 25

node failure
 handling 83
 proxy server failure 83

O

objects, containers

- about 9
- cURL, using 51
- listing 50
- REST API, using 51
- Swift client CLI, using 50

object storage

- about 5
- tasks 5
- using 5, 6

Open Compute Platform (OCP)

- about 109
- URL 110

OpenStack Swift

- about 7-9
- accessing, with client libraries 60
- accessing, with S3 commands 59, 60
- architectural principles 10
- architecture 10
- capacity planning 84
- configuration guide specifications 102
- data path software servers 13, 14
- downloading 26
- failure management 81
- features 20
- functionality 7
- hardware list 99
- implementations, URL 130
- implementing, operating systems
 - used 125, 126
- inline middleware options 18
- in Sahara 65
- installing 23-27
- issues 10
- key architectural principles 10
- migrations 85
- physical data organization 11, 12
- post-processing software components 16
- provisioning and deployment
 - tools 128, 129
- reference links 130
- routine management 69
- Tulsi 75
- using, for virtual machine storage 63, 64

- using, with Docker 90
- using, with Sahara 66, 67

OpenStack Swift, functionalities

- about 8
- account container object structure 7
- global cluster capability 7
- large object support 8
- Middleware architecture 8
- open source 7
- open standards 7
- partial object retrieval 8
- storage policies 8

OpenStack Swift implementations

- operating systems, using 126, 127
- virtualizations, using 127, 128

OpenStack Swift installation

- about 23
- finalizing 40
- hardware planning 23
- network configuration 24
- pre-installation, steps 25
- server setup 24
- storage policies 40

operating systems, OpenStack Swift implementations

- CentOS 127
- Debian 127
- HP Cloud OS 126
- Red Hat 126
- Ubuntu 126

operations 114

P

partitions 12

performance benchmarking

- about 111-116
- audio file sharing and collaboration 113
- COSBench tool 111
- document archiving 113
- examples 113
- ssbench tool 111
- swift-bench tool 111

performance-related sizing techniques

- client-facing network 105
- internal storage cluster 105
- replication network 105

**physical data organization hierarchy,
OpenStack Swift**

- disk (devices) 12
- region 11
- storage servers 12
- zone 11

plugins, Sahara

- Cloudera 65
- Hortonworks Data Platform 65
- MapR 65
- Spark 65
- Vanilla 65

post command

- about 135
- examples 136

post-processing software components

- account reaper process 17
- auditors 17
- container reconciler 18
- container-to-container synchronization 17
- drive audit process 17
- object expirer process 17
- replication 16
- updaters 17

post-processing software tuning

- bytes_per_second parameter 120
- concurrency parameter 119
- disk_chunk_size parameter 120
- files_per_second parameter 120
- interval 119

provisioning and deployment tools,

OpenStack Swift

- Chef 129
- Huawei 130
- Juju 130
- Puppet 128

proxy container

- creating, with Dockerfile 96, 97

python-swiftclient library

- URL 61

R

read affinity 15

ring 12

ring builder 12

routine management

- about 69, 70
- Swift cluster monitoring 70

rsyslog

- URL 121
- used, for logging 81

ruby-openstack library

- URL 61

S

Sahara

- architecture 65
- Hadoop Cluster, setting up 66
- job, running 67
- Openstack Swift, using 66, 67

selection criteria, hardware

- about 101-108
- account and container server configuration,
selecting 104
- additional networking equipment,
selecting 107
- availability 108
- cloud gateway, selecting 107
- durability 108
- heterogeneous hardware 107
- manageability 109
- network hardware, selecting 105
- proxy server configuration,
selecting 104, 105
- region and zone configuration,
determining 103
- scale 101
- server type ratios, selecting 106
- serviceability 108
- storage server configuration,
selecting 101-103

server design element configuration

- CPU performance 100
- Disk/JBOD 101
- flash memory 100
- hardware management 101
- memory 100
- network I/O 101

software tuning

- about 117
- data path software tuning 118, 119

- filesystem parameter 120
- logging parameter 121
- Memcached parameter 120
- network stack parameter 121
- operating system parameter 121
- post-processing software tuning 119, 120
- ring considerations 117, 118
- system time parameter 120
- stat command**
 - about 134
 - examples 135
- StatsD**
 - about 73, 74
 - URL 73
- storage container**
 - creating, with Dockerfile 97
 - setting up, with Docker image 94, 95
- Storage Container Object (SCO) 64**
- storage policies**
 - about 40
 - applying 43, 44
 - implementing 41-43
- storage server nodes**
 - hard disks, formatting 29
 - hard disks, mounting 29
 - RSYNC 30, 31
 - RSYNCD 30, 31
 - services, installing 27
 - setting up 27
 - upgrading 85, 86
- Swauth 46**
- Swift CLI commands**
 - about 133
 - delete 140
 - download 139
 - list 133
 - post 135
 - stat 134
 - upload 137
- Swift client CLI tool 45**
- Swift clusters**
 - monitoring 70-74
 - setting up, with Dockerfile 96
 - StatsD 73, 74
 - Swift dispersion tool 73
 - Swift Informant 72
 - Swift metrics 74
 - Swift Recon 70, 71
- swift-dispersion-report tool 21**
- Swift dispersion tool 73**
- Swiftfs 67**
- Swift Informant**
 - about 72
 - URL 72
- Swift proxy**
 - container, setting up with Docker image 93
 - used, for authentication 67, 68
- Swift Recon 70, 71**
- syslog-ng**
 - URL 121

T

- tapes 124**
- tenant 9**
- third-party middleware modules**
 - bulk archive autoextraction 20
 - bulk delete 19
 - CNAME lookup 19
 - container and account quotas 19
 - domain remap 19
 - form post 20
 - health check 19
 - logging 19
 - profiler 20
 - rate limiting 19
 - Recon 20
 - static web 20
 - Swift origin server 20
 - TempURL 20
- token**
 - creating, with Keystone authentication 46
- Tulsi**
 - about 75, 76
 - anomaly detection 80
 - architecture 76
 - client module 76
 - deploying 76, 77
 - functions 75
 - package, downloading 76, 77
 - running 77-79
 - server module 76

U

updaters 17

upload command

about 137

examples 138

use cases

about 123

archival 124

backup 124

collaboration 125

content repository 124

data lakes 125

OpenStack Swift 123

V

vendor selection strategy

about 109

branded hardware 109

commodity hardware 109, 110

virtual machine storage

OpenStack Swift, using 63, 64

W

wimpy servers 104

work items 114

workload 114

workstages 114

write affinity 15

WSGI servers 117



Thank you for buying **OpenStack Object Storage (Swift) Essentials**

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at www.packtpub.com.

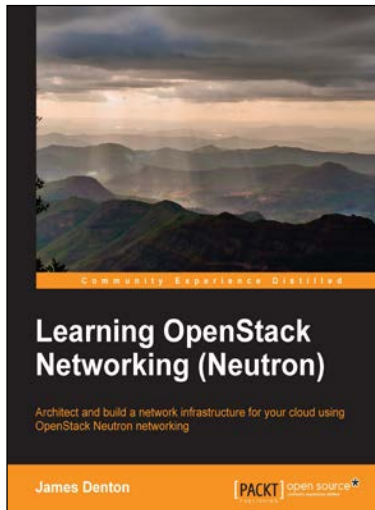
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



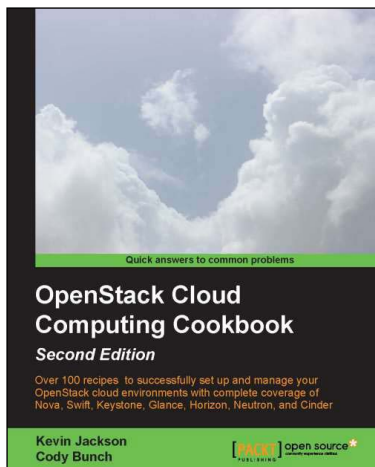
Learning OpenStack Networking (Neutron)

ISBN: 978-1-78398-330-8

Paperback: 300 pages

Architect and build a network infrastructure for your cloud using OpenStack Neutron networking

1. Build a virtual switching infrastructure for virtual machines using the Open vSwitch or Linux Bridge plugins.
2. Create networks and software routers that connect virtual machines to the Internet using built-in Linux networking features.
3. Scale your application using Neutron's load-balancing-as-a-service feature using the haproxy plugin.



OpenStack Cloud Computing Cookbook

Second Edition

ISBN: 978-1-78216-758-7

Paperback: 396 pages

Over 100 recipes to successfully set up and manage your OpenStack cloud environments with complete coverage of Nova, Swift, Keystone, Glance, Horizon, Neutron, and Cinder

1. Learn how to build your Private Cloud utilizing DevOps and Continuous Integration tools and techniques.
2. Updated for OpenStack Grizzly.
3. Learn how to install, configure, and manage all of the OpenStack core projects including new topics like block storage and software defined networking.

Please check www.PacktPub.com for information on our titles



Implementing Cloud Storage with OpenStack Swift

ISBN: 978-1-78216-805-8

Paperback: 140 pages

Design, implement, and successfully manage your own cloud storage cluster using the popular OpenStack Swift software

1. Learn about the fundamentals of cloud storage using OpenStack Swift.
2. Explore how to install and manage OpenStack Swift along with various hardware and tuning options.
3. Perform data transfer and management using REST APIs.



Citrix® XenDesktop® 7 Cookbook

ISBN: 978-1-78217-746-3

Paperback: 410 pages

Over 35 recipes to help you implement a fully featured XenDesktop® 7 architecture with a rich and powerful VDI experience

1. Implement the XenDesktop 7 architecture and its satellite components.
2. Learn how to publish desktops and applications to the end-user devices, optimizing their performance and increasing the general security.
3. Designed in a manner which will allow you to progress gradually from one chapter to another or to implement a single component only referring to the specific topic.

Please check www.PacktPub.com for information on our titles