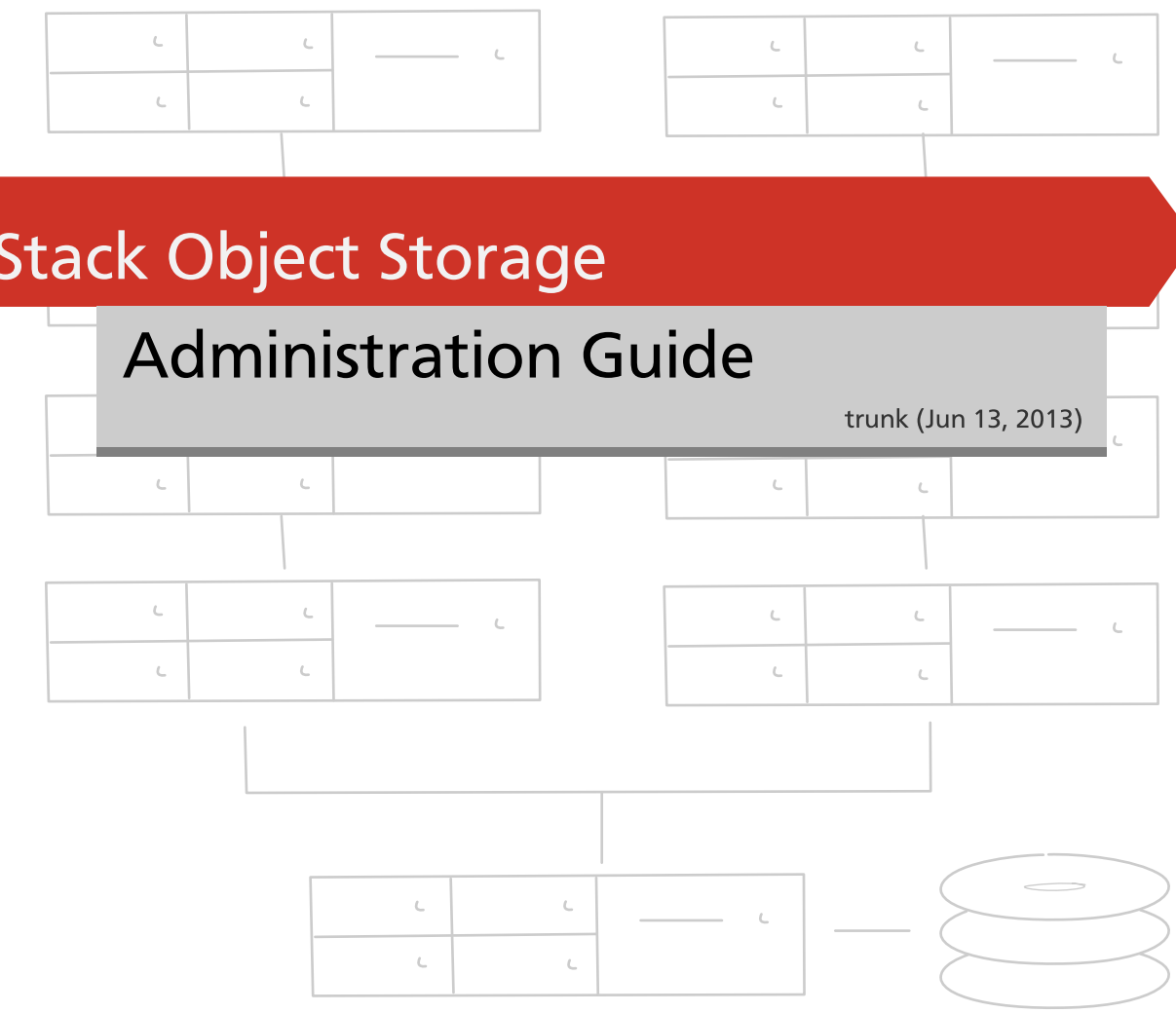
OpenStack Object Storage

Administration Guide

trunk (Jun 13, 2013)



openstack™

docs.openstack.org

OpenStack Object Storage Administration Guide

trunk (2013-06-13)

OpenStack™ Object Storage offers open source software for cloud-based object storage for any organization. This manual provides guidance for installing, managing, and understanding the software that runs OpenStack Object Storage.

Table of Contents

List of Figures

List of Tables

1. Getting Started with OpenStack

OpenStack is a collection of open source technologies that provides massively scalable cloud computing software. OpenStack can be used by corporations, service providers, VARS, SMBs, researchers, and global data centers looking to deploy large-scale cloud deployments for private or public clouds.

Why Cloud?

In data centers today, many computers suffer the same under-utilization in computing power and networking bandwidth. For example, projects may need a large amount of computing capacity to complete a computation, but no longer need the computing power after completing the computation. You want cloud computing when you want a service that's available on-demand with the flexibility to bring it up or down through automation or with little intervention. The phrase "cloud computing" is often represented with a diagram that contains a cloud-like shape indicating a layer where responsibility for service goes from user to provider. The cloud in these types of diagrams contains the services that afford computing power harnessed to get work done. Much like the electrical power we receive each day, cloud computing provides subscribers or users with access to a shared collection of computing resources: networks for transfer, servers for storage, and applications or services for completing tasks.

These are the compelling features of a cloud:

- On-demand self-service: Users can provision servers and networks with little human intervention.
- Network access: Any computing capabilities are available over the network. Many different devices are allowed access through standardized mechanisms.
- Resource pooling: Multiple users can access clouds that serve other consumers according to demand.
- Elasticity: Provisioning is rapid and scales out or in based on need.
- Metered or measured service: Just like utilities that are paid for by the hour, clouds should optimize resource use and control it for the level of service or type of servers such as storage or processing.

Cloud computing offers different service models depending on the capabilities a consumer may require.

- SaaS: Software as a Service. Provides the consumer the ability to use the software in a cloud environment, such as web-based email for example.

- PaaS: Platform as a Service. Provides the consumer the ability to deploy applications through a programming language or tools supported by the cloud platform provider. An

example of platform as a service is an Eclipse/Java programming platform provided with no downloads required.

- IaaS: Infrastructure as a Service. Provides infrastructure such as computer instances, network connections, and storage so that people can run any software or operating system.

When you hear terms such as public cloud or private cloud, these refer to the deployment model for the cloud. A private cloud operates for a single organization, but can be managed on-premise or off-premise. A public cloud has an infrastructure that is available to the general public or a large industry group and is likely owned by a cloud services company. The NIST also defines community cloud as shared by several organizations supporting a specific community with shared concerns.

Clouds can also be described as hybrid. A hybrid cloud can be a deployment model, as a composition of both public and private clouds, or a hybrid model for cloud computing may involve both virtual and physical servers.

What have people done with cloud computing? Cloud computing can help with large-scale computing needs or can lead consolidation efforts by virtualizing servers to make more use of existing hardware and potentially release old hardware from service. People also use cloud computing for collaboration because of its high availability through networked computers. Productivity suites for word processing, number crunching, and email communications, and more are also available through cloud computing. Cloud computing also avails additional storage to the cloud user, avoiding the need for additional hard drives on each user's desktop and enabling access to huge data storage capacity online in the cloud.

For a more detailed discussion of cloud computing's essential characteristics and its models of service and deployment, see http://www.nist.gov/itl/cloud/, published by the US National Institute of Standards and Technology.

What is OpenStack?

OpenStack is on a mission: to provide scalable, elastic cloud computing for both public and private clouds, large and small. At the heart of our mission is a pair of basic requirements: clouds must be simple to implement and massively scalable.

If you are new to OpenStack, you will undoubtedly have questions about installation, deployment, and usage. It can seem overwhelming at first. But don't fear, there are places to get information to guide you and to help resolve any issues you may run into during the on-ramp process. Because the project is so new and constantly changing, be aware of the revision time for all information. If you are reading a document that is a few months old and you feel that it isn't entirely accurate, then please let us know through the mailing list at https://launchpad.net/~openstack or by filing a bug at https://bugs.launchpad.net/openstack-manuals/+filebug so it can be updated or removed.

Components of OpenStack

There are currently seven core components of OpenStack: Compute, Object Storage, Identity, Dashboard, Block Storage, Network and Image Service. Let's look at each in turn.

- Object Store (codenamed "Swift") provides object storage. It allows you to store or retrieve files (but not mount directories like a fileserver). Several companies provide commercial storage services based on Swift. These include KT, Rackspace (from which Swift originated) and Internap. Swift is also used internally at many large companies to store their data.

- Image (codenamed "Glance") provides a catalog and repository for virtual disk images. These disk images are mostly commonly used in OpenStack Compute. While this service is technically optional, any cloud of size will require it.

- Compute (codenamed "Nova") provides virtual servers upon demand. Rackspace and HP provide commercial compute services built on Nova and it is used internally at companies like Mercado Libre and NASA (where it originated).

- Dashboard (codenamed "Horizon") provides a modular web-based user interface for all the OpenStack services. With this web GUI, you can perform most operations on your cloud like launching an instance, assigning IP addresses and setting access controls.

- Identity (codenamed "Keystone") provides authentication and authorization for all the OpenStack services. It also provides a service catalog of services within a particular OpenStack cloud.

- Network (codenamed "Quantum") provides "network connectivity as a service" between interface devices managed by other OpenStack services (most likely Nova). The service works by allowing users to create their own networks and then attach interfaces to them. OpenStack Network has a pluggable architecture to support many popular networking vendors and technologies.

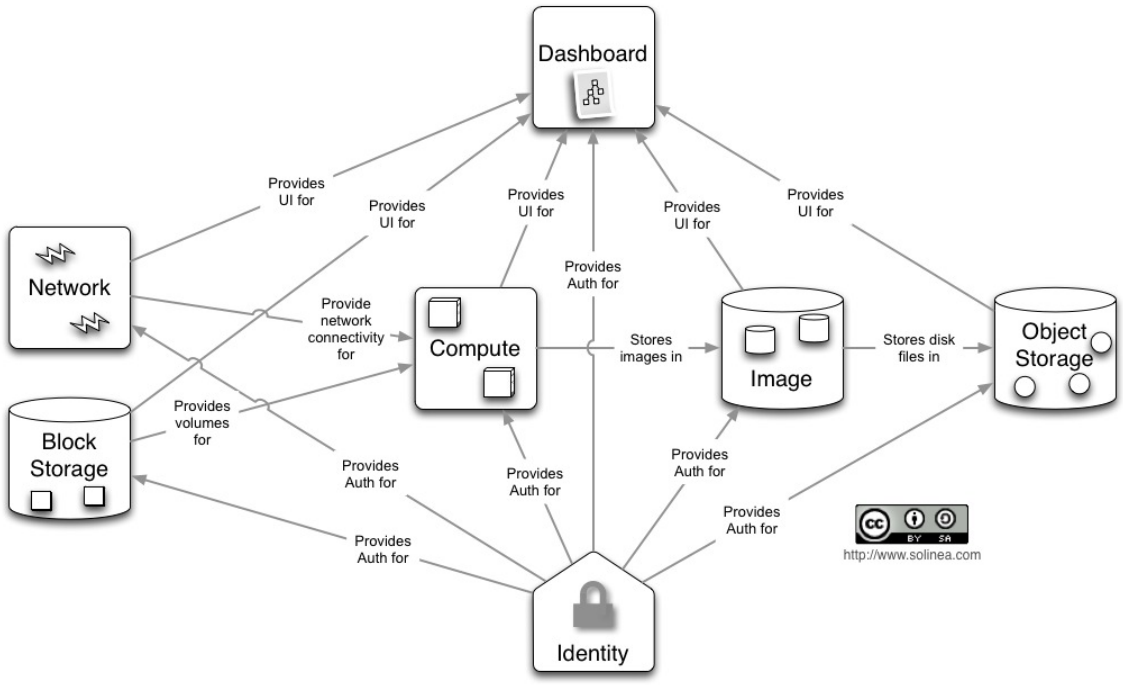
- Block Storage (codenamed "Cinder") provides persistent block storage to guest VMs.

In addition to these core projects, there are also a number of "incubation" projects that are being considered for future integration into the OpenStack release.

Conceptual Architecture

The OpenStack project as a whole is designed to deliver a massively scalable cloud operating system. To achieve this, each of the constituent services are designed to work together to provide a complete Infrastructure as a Service (IaaS). This integration is facilitated through public application programming interfaces (APIs) that each service offers (and in turn can consume). While these APIs allow each of the services to use another service, it also allows an implementer to switch out any service as long as they maintain the API. These are (mostly) the same APIs that are available to end users of the cloud.

Conceptually, you can picture the relationships between the services as so:



- Dashboard ("Horizon") provides a web front end to the other OpenStack services
- Compute ("Nova") stores and retrieves virtual disks ("images") and associated metadata in Image ("Glance")
- Network ("Quantum") provides virtual networking for Compute.
- Block Storage ("Cinder") provides storage volumes for Compute.
- Image ("Glance") can store the actual virtual disk files in the Object Store("Swift")
- All the services authenticate with Identity ("Keystone")

This is a stylized and simplified view of the architecture, assuming that the implementer is using all of the services together in the most common configuration. It also only shows the "operator" side of the cloud – it does not picture how consumers of the cloud may actually use it. For example, many users will access object storage heavily (and directly).

Logical Architecture

The following paragraphs give some details on the main modules in the OpenStack components.

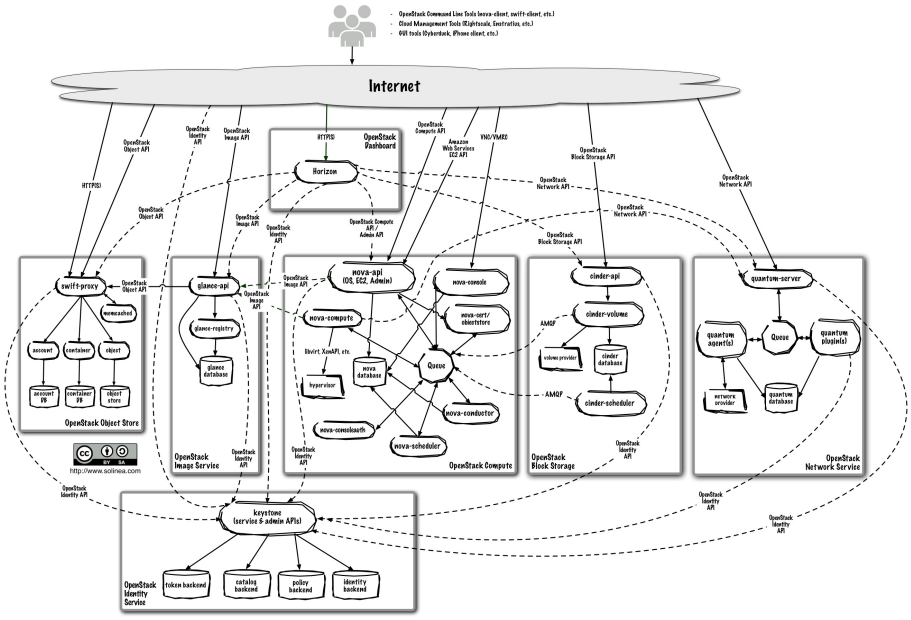
These details are not meant to be exhaustive; the objective is to describe the relevant aspects that administrators need to know to better understand how to design the deployment, and install and configure the whole platform.

Modules are organized according to the functional area they belong (i.e. the kind of functions they implement or deliver) and classified according to their type.

These are the types:

- daemon: runs as a daemon and, on Linux platforms, is usually installed as a service;
- script: a script run by an external module when some event happens (at the moment, it is used as a co-routine of dnsmasq for managing IP Addresses released to instances via DHCP protocol);
- client: a client for accessing the Python bindings for a service
- CLI: a Command Line Interpreter for submitting commands to OpenStack Compute for example

As you can imagine, the logical architecture is far more complicated than the conceptual architecture shown above. As with any service-oriented architecture, diagrams quickly become "messy" trying to illustrate all the possible combinations of service communications. The diagram below, illustrates the most common architecture of an OpenStack-based cloud. However, as OpenStack supports a wide variety of technologies, it does not represent the only architecture possible.



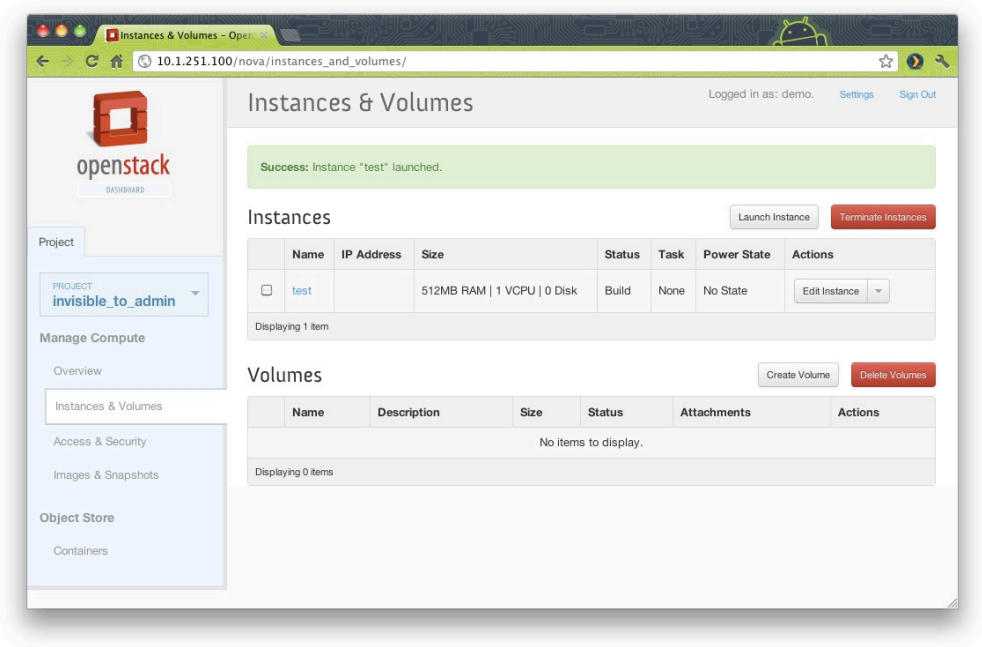
This picture is consistent with the conceptual architecture above in that:

- End users can interact through a common web interface (Horizon) or directly to each service through their API
- All services authenticate through a common source (facilitated through Keystone)
- Individual services interact with each other through their public APIs (except where privileged administrator commands are necessary)

In the sections below, we'll delve into the architecture for each of the services.

Dashboard

Horizon is a modular Django web application that provides an end user and administrator interface to OpenStack services.



As with most web applications, the architecture is fairly simple:

- Horizon is usually deployed via mod_wsgi in Apache. The code itself is separated into a reusable python module with most of the logic (interactions with various OpenStack APIs) and presentation (to make it easily customizable for different sites).
- A database (configurable as to which one). As it relies mostly on the other services for data, it stores very little data of its own.

From a network architecture point of view, this service will need to be customer accessible as well as be able to talk to each service's public APIs. If you wish to use the administrator functionality (i.e. for other services), it will also need connectivity to their Admin API endpoints (which should be non-customer accessible).

Compute

Nova is the most complicated and distributed component of OpenStack. A large number of processes cooperate to turn end user API requests into running virtual machines. Main modules are implemented in Python. The following lists are organized by functional areas:

API

- `nova-api` accepts and responds to end user compute API calls. It supports OpenStack Compute API, Amazon's EC2 API and a special Admin API (for privileged users to perform administrative actions). It also initiates most of the orchestration activities (such as running an instance) as well as enforces some policy.
- `nova-api-metadata` accepts metadata requests from instances (more details). The nova-api-metadata service is generally only used when running in multi-host mode with nova-network installations.

Computing core

- The `nova-compute` process is primarily a worker daemon that creates and terminates virtual machine instances via hypervisor's APIs (XenAPI for XenServer/XCP, libvirt for KVM or QEMU, VMwareAPI for VMware, etc.). The process by which it does so is fairly complex but the basics are simple: accept actions from the queue and then perform a series of system commands (like launching a KVM instance) to carry them out while updating state in the database.

- The `nova-schedule` process is conceptually the simplest piece of code in OpenStack Nova: take a virtual machine instance request from the queue and determines where it should run (specifically, which compute server host it should run on).

- The `nova-conductor` module, introduced in the Grizzly release, works as a "mediator" between nova-compute and the database. It is aimed at eliminating all the direct accesses to the cloud database made by nova-compute. The nova-conductor module scales horizontally but it shouldn't be deployed on the same node(s) where nova-compute runs. You can read more about the new service here.

Networking for VMs

- The `nova-network` worker daemon is very similar to `nova-compute`. It accepts networking tasks from the queue and then performs tasks to manipulate the network (such as setting up bridging interfaces or changing iptables rules). This functionality is being migrated to OpenStack Networking, a separate OpenStack service.

- `nova-dhcpbridge` (script) This script tracks IP address leases and records them in the database using dnsmasq's dhcp-script facility. This functionality is also migrated to OpenStack Networking; a different script is provided when using OpenStack Networking (code-named Quantum).

Console Interface

- The `nova-consoleauth` daemon authorizes user's tokens that console proxies provide (see nova-novncproxy and nova-xvpnvcproxy). This service must be running in order for console proxies to work. Many proxies of either type can be run against a single nova-consoleauth service in a cluster configuration. Read more details.

- The `nova-novncproxy` (daemon) provides a proxy for accessing running instances through a VNC connection. It supports browser-based novnc clients.

- The deprecated `nova-console` daemon is no longer used with Grizzly, and the nova-xvpnvncproxy is used instead.

- The `nova-xvpnvncproxy` daemon is a proxy for accessing running instances through a VNC connection. It supports a Java client specifically designed for OpenStack.

- The `nova-cert` daemon manages x509 certificates.

Image Management (EC2 scenario)

- The `nova-objectstore` daemon provides an S3 interface for registering images onto the image management service (see glance) It is mainly used for installations that need to support euca2ools. The euca2ools tools talk to nova-objectore in "S3 language" and nova-objectstore translates S3 requests into glance requests

- The `euca2ools` client is not an OpenStack module but it can be supported by OpenStack. It's a set of command line interpreter commands for managing cloud resources. Provided that nova-api is configured to support EC2 interface, euca2ools can be used to issue cloud management commands. For more information on euca2ools, see http://www.eucalyptus.com/eucalyptus-cloud/documentation/2.0.

Command Line Interpreter/Interfaces

- The `nova` client enables you to submit either tenant administrator's commands or cloud user's commands.
- The `nova-manage` client submits cloud administrator commands.
- The queue provides a central hub for passing messages between daemons. This is usually implemented with RabbitMQ today, but could be any AMPQ message queue (such as Apache Qpid), or Zero MQ.
- The SQL database stores most of the build-time and run-time state for a cloud infrastructure. This includes the instance types that are available for use, instances in use, networks available and projects. Theoretically, OpenStack Nova can support any database supported by SQL-Alchemy but the only databases currently being widely used are sqlite3 (only appropriate for test and development work), MySQL and PostgreSQL.

Nova interacts with many other OpenStack services: Keystone for authentication, Glance for images and Horizon for web interface. The Glance interactions are central. The API process can upload and query Glance while `nova-compute` will download images for use in launching images.

Object Store

The swift architecture is very distributed to prevent any single point of failure as well as to scale horizontally. It includes the following components:

- Proxy server (`swift-proxy-server`) accepts incoming requests via the OpenStack Object API or just raw HTTP. It accepts files to upload, modifications to metadata or container creation. In addition, it will also serve files or container listing to web browsers. The proxy server may utilize an optional cache (usually deployed with memcache) to improve performance.
- Account servers manage accounts defined with the object storage service.
- Container servers manage a mapping of containers (i.e folders) within the object store service.
- Object servers manage actual objects (i.e. files) on the storage nodes.
- There are also a number of periodic process which run to perform housekeeping tasks on the large data store. The most important of these is the replication services, which ensures consistency and availability through the cluster. Other periodic processes include auditors, updaters and reapers.

Authentication is handled through configurable WSGI middleware (which will usually be Keystone).

Image Store

Glance has four main parts to it:

- `glance-api` accepts Image API calls for image discovery, image retrieval and image storage.
- `glance-registry` stores, processes and retrieves metadata about images (size, type, etc.).
- A database to store the image metadata. Like Nova, you can choose your database depending on your preference (but most people use MySQL or SQlite).
- A storage repository for the actual image files. In the diagram above, Swift is shown as the image repository, but this is configurable. In addition to Swift, Glance supports normal filesystems, RADOS block devices, Amazon S3 and HTTP. Be aware that some of these choices are limited to read-only usage.

There are also a number of periodic process which run on Glance to support caching. The most important of these is the replication services, which ensures consistency and availability through the cluster. Other periodic processes include auditors, updaters and reapers.

As you can see from the diagram in the Conceptual Architecture section, Glance serves a central role to the overall IaaS picture. It accepts API requests for images (or image metadata) from end users or Nova components and can store its disk files in the object storage service, Swift.

Identity

Keystone provides a single point of integration for OpenStack policy, catalog, token and authentication.

- `keystone` handles API requests as well as providing configurable catalog, policy, token and identity services.
- Each Keystone function has a pluggable backend which allows different ways to use the particular service. Most support standard backends like LDAP or SQL, as well as Key Value Stores (KVS).

Most people will use this as a point of customization for their current authentication services.

Network

OpenStack Networking provides "network connectivity as a service" between interface devices managed by other OpenStack services (most likely Compute). The service works by allowing users to create their own networks and then attach interfaces to them. Like many of the OpenStack services, OpenStack Networking is highly configurable due to its plug-in architecture. These plug-ins accommodate different networking equipment and software. As such, the architecture and deployment can vary dramatically. In the above architecture, a simple Linux networking plug-in is shown.

- `quantum-server` accepts API requests and then routes them to the appropriate OpenStack Networking plugin for action.
- OpenStack Networking plugins and agents perform the actual actions such as plugging and unplugging ports, creating networks or subnets and IP addressing. These plugins and agents differ depending on the vendor and technologies used in the particular cloud. OpenStack Networking ships with plugins and agents for: Cisco virtual and physical switches, Nicira NVP product, NEC OpenFlow products, Open vSwitch, Linux bridging and the Ryu Network Operating System.

  The common agents are L3 (layer 3), DHCP (dynamic host IP addressing) and the specific plug-in agent.
- Most OpenStack Networking installations also make use of a messaging queue to route information between the quantum-server and various agents as well as a database to store networking state for particular plugins.

OpenStack Networking interacts mainly with OpenStack Compute, where it provides networks and connectivity for its instances.

Block Storage

The OpenStack Block Storage API allows for manipulation of volumes, volume types (similar to compute flavors) and volume snapshots.

- `cinder-api` accepts API requests and routes them to cinder-volume for action.
- `cinder-volume` acts upon the requests by reading or writing to the Cinder database to maintain state, interacting with other processes (like `cinder-scheduler`) through a message queue and directly upon block storage providing hardware or software. It can interact with a variety of storage providers through a driver architecture. Currently, there are drivers for IBM, SolidFire, NetApp, Nexenta, Zadara, GlusterFS, linux iSCSI and other storage providers.
- Much like `nova-scheduler`, the `cinder-scheduler` daemon picks the optimal block storage provider node to create the volume on.
- OpenStack Block Storage deployments will also make use of a messaging queue to route information between the cinder processes as well as a database to store volume state.

Like OpenStack Network, OpenStack Block Storage will mainly interact with OpenStack Compute, providing volumes for its instances.

Storage Concepts

Storage is found in many parts of the OpenStack stack, and the differing types can cause confusion to even experienced cloud engineers. Here's a simple chart to kick-start your understanding:

Table 1.1. Types of Storage

On-instance / ephemeral	Volumes block storage (Cinder)	Object Storage (Swift)
Used for running Operating System and scratch space	Used for adding additional persistent storage to a virtual machine (VM)	Used for storing virtual machine images and data

On-instance / ephemeral	Volumes block storage (Cinder)	Object Storage (Swift)
Persists until VM is terminated	Persists until deleted	Persists until deleted
Access associated with a VM	Access associated with a VM	Available from anywhere
Implemented as a filesystem underlying OpenStack Compute	Mounted via OpenStack Block-Storage controlled protocol (for example, iSCSI)	REST API
Administrator configures size setting, based on flavors	Sizings based on need	Easily scalable for future growth
Example: 10GB first disk, 30GB/core second disk	Example: 1TB "extra hard drive"	Example: 10s of TBs of dataset storage

Other points of note include:

- *OpenStack Object Storage is not used like a traditional hard drive.* Object storage is all about relaxing some of the constraints of a POSIX-style file system. The access to it is API-based (and the API uses http). This is a good idea as if you don't have to provide atomic operations (that is, you can rely on eventual consistency), you can much more easily scale a storage system and avoid a central point of failure.

- *The OpenStack Image Service is used to manage the virtual machine images in an OpenStack cluster, not store them.* Instead, it provides an abstraction to different methods for storage - a bridge to the storage, not the storage itself.

- *OpenStack Object Storage can function on its own.* The Object Storage (swift) product can be used independently of the Compute (nova) product.

2. Introduction to OpenStack Object Storage

OpenStack Object Storage is a scalable object storage system - it is not a file system in the traditional sense. You will not be able to mount this system like traditional SAN or NAS volumes. Since OpenStack Object Storage is a different way of thinking when it comes to storage, take a few moments to review the key concepts listed below.

Accounts and Account Servers

The OpenStack Object Storage system is designed to be used by many different storage consumers or customers. Each user must identify themselves using an authentication system.

Nodes that run the Account service are a separate concept from individual accounts. Account servers are part of the storage system and must be configured along with Container servers and Object servers.

Authentication and Access Permissions

You must authenticate against an Authentication service to receive OpenStack Object Storage connection parameters and an authentication token. The token must be passed in for all subsequent container/object operations. One authentication service that you can use as a middleware example is called swauth and you can download it from https://github.com/gholt/swauth. You can also integrate with the OpenStack Identity Service, code-named Keystone, which you can download from https://github.com/openstack/keystone.

Note

Typically the language-specific APIs handle authentication, token passing, and HTTPS request/response communication.

You can implement access control for objects either for users or accounts using X-Container-Read: accountname and X-Container-Write: accountname:username, which allows any user from the accountname account to read but only allows the username user from the accountname account to write. You can also grant public access to objects stored in OpenStack Object Storage but also limit public access using the Referer header to prevent site-based content theft such as hot-linking (for example, linking to an image file from off-site and therefore using other's bandwidth). The public container settings are used as the default authorization over access control lists. For example, using X-Container-Read: referer:any allows anyone to read from the container regardless of other authorization settings.

Generally speaking, each user has their own storage account and has full access to that account. Users must authenticate with their credentials as described above, but once authenticated they can create/delete containers and objects within that account. The only way a user can access the content from another account is if they share an API access key or a session token provided by your authentication system.

Containers and Objects

A container is a storage compartment for your data and provides a way for you to organize your data. You can think of a container as a folder in Windows® or a directory in UNIX®. The primary difference between a container and these other file system concepts is that containers cannot be nested. You can, however, create an unlimited number of containers within your account. Data must be stored in a container so you must have at least one container defined in your account prior to uploading data.

The only restrictions on container names is that they cannot contain a forward slash (`/`) and must be less than 256 bytes in length. Please note that the length restriction applies to the name after it has been URL encoded. For example, a container name of `Course Docs` would be URL encoded as `Course%20Docs` and therefore be 13 bytes in length rather than the expected 11.

An object is the basic storage entity and any optional metadata that represents the files you store in the OpenStack Object Storage system. When you upload data to OpenStack Object Storage, the data is stored as-is (no compression or encryption) and consists of a location (container), the object's name, and any metadata consisting of key/value pairs. For instance, you may chose to store a backup of your digital photos and organize them into albums. In this case, each object could be tagged with metadata such as `Album : Caribbean Cruise` or `Album : Aspen Ski Trip`.

The only restriction on object names is that they must be less than 1024 bytes in length after URL encoding. For example, an object name of `C++final(v2).txt` should be URL encoded as `C%2B%2Bfinal%28v2%29.txt` and therefore be 24 bytes in length rather than the expected 16.

The maximum allowable size for a storage object upon upload is 5 gigabytes (GB) and the minimum is zero bytes. You can use the built-in large object support and the swift utility to retrieve objects larger than 5 GB.

For metadata, you should not exceed 90 individual key/value pairs for any one object and the total byte length of all key/value pairs should not exceed 4KB (4096 bytes).

Warning

The Object Storage API uses HTTP which specifies that header fields, such as those used to send metadata are case insensitive. Please do not expect the case of metadata items being preserved.

Operations

Operations are the actions you perform within an OpenStack Object Storage system such as creating or deleting containers, uploading or downloading objects, and so on. The full list

of operations is documented in the Developer Guide. Operations may be performed via the REST web service API or a language-specific API; currently, we support Python, PHP, Java, Ruby, and C#/.NET.

Important

All operations must include a valid authorization token from your authorization system.

Language-Specific API Bindings

A set of supported API bindings in several popular languages are available from the Rackspace Cloud Files product, which uses OpenStack Object Storage code for its implementation. These bindings provide a layer of abstraction on top of the base REST API, allowing programmers to work with a container and object model instead of working directly with HTTP requests and responses. These bindings are free (as in beer and as in speech) to download, use, and modify. They are all licensed under the MIT License as described in the COPYING file packaged with each binding. If you do make any improvements to an API, you are encouraged (but not required) to submit those changes back to us.

The API bindings for Rackspace Cloud Files are hosted at http://github.com/rackspace. Feel free to coordinate your changes through github or, if you prefer, send your changes to cloudfiles@rackspacecloud.com. Just make sure to indicate which language and version you modified and send a unified diff.

Each binding includes its own documentation (either HTML, PDF, or CHM). They also include code snippets and examples to help you get started. The currently supported API binding for OpenStack Object Storage are:

- PHP (requires 5.x and the modules: cURL, FileInfo, mbstring)
- Python (requires 2.4 or newer)
- Java (requires JRE v1.5 or newer)
- C#/.NET (requires .NET Framework v3.5)
- Ruby (requires 1.8 or newer and mime-tools module)

There are no other supported language-specific bindings at this time. You are welcome to create your own language API bindings and we can help answer any questions during development, host your code if you like, and give you full credit for your work.

3. Installing OpenStack Object Storage

The OpenStack Object Storage services work together to provide object storage and retrieval through a REST API.

System Requirements

Hardware: OpenStack Object Storage specifically is designed to run on commodity hardware.

Table 3.1. Hardware Recommendations

Server	Recommended Hardware	Notes
Object Storage object servers	Processor: dual quad core Memory: 8 or 12 GB RAM Disk space: optimized for cost per GB Network: one 1 GB Network Interface Card (NIC)	The amount of disk space depends on how much you can fit into the rack efficiently. You want to optimize these for best cost per GB while still getting industry-standard failure rates. At Rackspace, our storage servers are currently running fairly generic 4U servers with 24 2T SATA drives and 8 cores of processing power. RAID on the storage drives is not required and not recommended. Swift's disk usage pattern is the worst case possible for RAID, and performance degrades very quickly using RAID 5 or 6. As an example, Rackspace runs Cloud Files storage servers with 24 2T SATA drives and 8 cores of processing power. Most services support either a worker or concurrency value in the settings. This allows the services to make effective use of the cores available.
Object Storage container/account servers	Processor: dual quad core Memory: 8 or 12 GB RAM Network: one 1 GB Network Interface Card (NIC)	Optimized for IOPS due to tracking with SQLite databases.
Object Storage proxy server	Processor: dual quad core Network: one 1 GB Network Interface Card (NIC)	Higher network throughput offers better performance for supporting many API requests. Optimize your proxy servers for best CPU performance. The Proxy Services are more CPU and network I/O intensive. If you are using 10g networking to the proxy, or are terminating SSL traffic at the proxy, greater CPU power will be required.

Operating System: OpenStack Object Storage currently runs on Ubuntu, RHEL, CentOS, or Fedora and the large scale deployment at Rackspace runs on Ubuntu 10.04 LTS.

Networking: 1000 Mbps are suggested. For OpenStack Object Storage, an external network should connect the outside world to the proxy servers, and the storage network is intended to be isolated on a private network or multiple private networks.

Database: For OpenStack Object Storage, a SQLite database is part of the OpenStack Object Storage container and account management process.

Permissions: You can install OpenStack Object Storage either as root or as a user with sudo permissions if you configure the sudoers file to enable all the permissions.

Object Storage Network Planning

For both conserving network resources and ensuring that network administrators understand the needs for networks and public IP addresses for providing access to the APIs and storage network as necessary, this section offers recommendations and required minimum sizes. Throughput of at least 1000 Mbps is suggested.

This document refers to two networks. One is a Public Network for connecting to the Proxy server, and the second is a Storage Network that is not accessible from outside the cluster, to which all of the nodes are connected. All of the OpenStack Object Storage services, as well as the rsync daemon on the Storage nodes are configured to listen on their STORAGE_LOCAL_NET IP addresses.

Public Network (Publicly routable IP range): This network is utilized for providing Public IP accessibility to the API endpoints within the cloud infrastructure.

Minimum size: 8 IPs (CIDR /29)

Storage Network (RFC1918 IP Range, not publicly routable): This network is utilized for all inter-server communications within the Object Storage infrastructure.

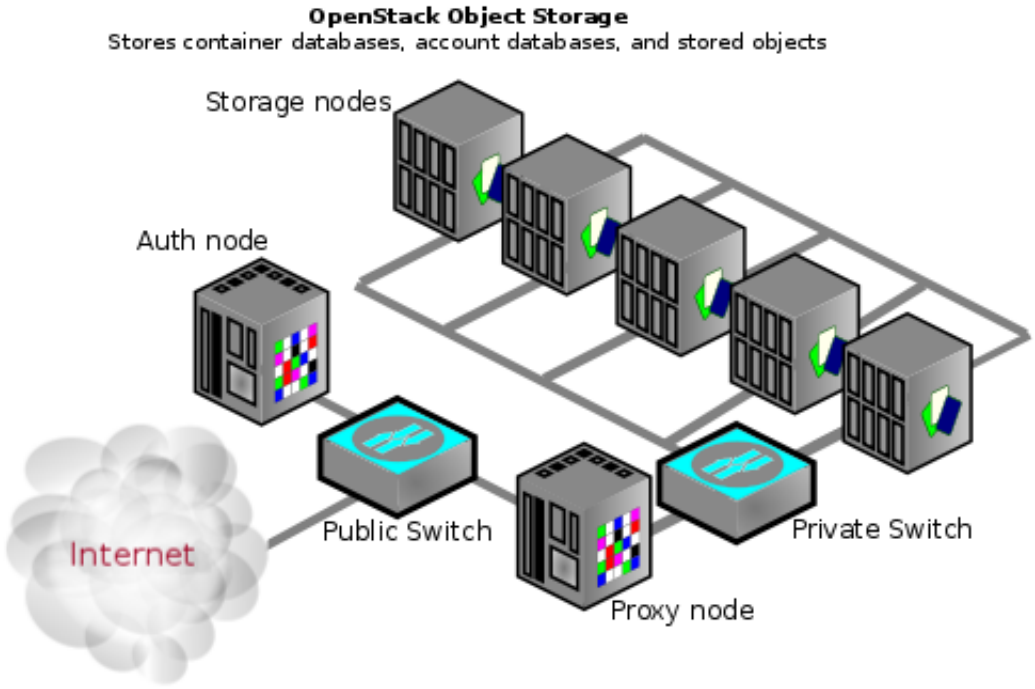
Recommended size: 255 IPs (CIDR /24)

Example Object Storage Installation Architecture

- node - a host machine running one or more OpenStack Object Storage services
- Proxy node - node that runs Proxy services
- Auth node - an optionally separate node that runs the Auth service separately from the Proxy services
- Storage node - node that runs Account, Container, and Object services
- Ring - a set of mappings of OpenStack Object Storage data to physical devices

To increase reliability, you may want to add additional Proxy servers for performance.

This document describes each Storage node as a separate zone in the ring. It is recommended to have a minimum of 5 zones. A zone is a group of nodes that is as isolated as possible from other nodes (separate servers, network, power, even geography). The ring guarantees that every replica is stored in a separate zone. This diagram shows one possible configuration for a minimal installation.



Installing OpenStack Object Storage on Ubuntu

Though you can install OpenStack Object Storage for development or testing purposes on a single server, a multiple-server installation enables the high availability and redundancy you want in a production distributed object storage system.

If you would like to perform a single node installation on Ubuntu for development purposes from source code, use the Swift All In One instructions or DevStack. See http://swift.openstack.org/development_saio.html for manual instructions or http://devstack.org for all-in-one including authentication and a dashboard.

Before You Begin

Have a copy of the Ubuntu Server installation media on hand if you are installing on a new server.

This document demonstrates installing a cluster using the following types of nodes:

- One Proxy node which runs the swift-proxy-server processes and may also run the optional swauth or tempauth services, this walkthrough uses the Identity service code-named Keystone. The proxy server serves proxy requests to the appropriate Storage nodes.

- Five Storage nodes that run the swift-account-server, swift-container-server, and swift-object-server processes which control storage of the account databases, the container databases, as well as the actual stored objects.

Note

Fewer Storage nodes can be used initially, but a minimum of 5 is recommended for a production cluster.

General Installation Steps

1. Install the baseline operating system, such as Ubuntu Server (12.04) or RHEL, CentOS, or Fedora, on all nodes.
2. Install core Swift files and openSSH.

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

```
# yum install openstack-swift openstack-swift-proxy openstack-swift-account
 openstack-swift-container openstack-swift-object memcached
```

3. Create and populate configuration directories on all nodes:

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

4. Create /etc/swift/swift.conf:

```
[swift-hash]
# random unique string that can never change (DO NOT LOSE)
swift_hash_path_suffix = fLIbertYgibbitZ
```

Note

The suffix value in `/etc/swift/swift.conf` should be set to some random string of text to be used as a salt when hashing to determine mappings in the ring. This file should be the same on every node in the cluster!

Next, set up your storage nodes, proxy node, and an auth node, in this walkthrough we'll use the OpenStack Identity Service, Keystone, for the common auth piece.

Installing and Configuring the Storage Nodes

Note

OpenStack Object Storage should work on any modern filesystem that supports Extended Attributes (XATTRS). We currently recommend XFS as it demonstrated the best overall performance for the swift use case after considerable testing and benchmarking at Rackspace. It is also the only filesystem that has been thoroughly tested.

1. Install Storage node packages:

```
# apt-get install swift-account swift-container swift-object xfsprogs
```

```
# yum install openstack-swift-account openstack-swift-container openstack-
swift-object xfsprogs
```

2. For every device on the node you wish to use for storage, setup the XFS volume (`/dev/sdb` is used as an example). Use a single partition per drive. For example, in a server with 12 disks you may use one or two disks for the operating system which should not be touched in this step. The other 10 or 11 disks should be partitioned with a single partition, then formatted in XFS.

```
# fdisk /dev/sdb
```

```
mkfs.xfs -i size=1024 /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
chown -R swift:swift /srv/node
```

3. Create `/etc/rsyncd.conf`:

```
uid = swift
gid = swift
log file = /var/log/rsyncd.log
pid file = /var/run/rsyncd.pid
address = <STORAGE_LOCAL_NET_IP>

[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
```

4. Edit the following line in `/etc/default/rsync`:

```
RSYNC_ENABLE = true
```

5. Start rsync daemon:

```
# service rsync start
```

Note

The rsync daemon requires no authentication, so it should be run on a local, private network.

6. Create the swift recon cache directory and set its permissions.

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

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

Installing and Configuring the Proxy Node

The proxy server takes each request and looks up locations for the account, container, or object and routes the requests correctly. The proxy server also handles API requests. You enable account management by configuring it in the `proxy-server.conf` file.

Note

It is assumed that all commands are run as the root user.

1. Install swift-proxy service:

```
# apt-get install swift-proxy memcached python-keystoneclient python-
swiftclient python-webob
```

```
# yum install openstack-swift-proxy memcached openstack-utils python-
swiftclient python-keystone-auth-token
```

2. Create self-signed cert for SSL:

```
# cd /etc/swift
# openssl req -new -x509 -nodes -out cert.crt -keyout cert.key
```

3. Modify memcached to listen on the default interfaces. Preferably this should be on a local, non-public network. Edit the following line in `/etc/memcached.conf`, changing:

```
-l 127.0.0.1
to
-l <PROXY_LOCAL_NET_IP>
```

4. Restart the memcached server:

```
# service memcached restart
```

5. RHEL/CentOS/Fedora Only: To set up Object Storage to authenticate tokens we need to set the keystone Admin token in the swift proxy file with the openstack-config command.

```
# openstack-config --set /etc/swift/proxy-server.conf filter:authtoken
 admin_token $ADMIN_TOKEN
# sudo openstack-config --set /etc/swift/proxy-server.conf filter:authtoken
 auth_token $ADMIN_TOKEN
```

6. Create `/etc/swift/proxy-server.conf`:

```
[DEFAULT]
bind_port = 8888
user = swift

[pipeline:main]
pipeline = healthcheck cache authtoken keystoneauth proxy-server

[app:proxy-server]
```

```
use = egg:swift#proxy
allow_account_management = true
account_autocreate = true

[filter:keystoneauth]
use = egg:swift#keystoneauth
operator_roles = Member,admin,swiftoperator

[filter:authtoken]
paste.filter_factory = keystoneclient.middleware.auth_token:filter_factory

# Delaying the auth decision is required to support token-less
# usage for anonymous referrers ('.r:*').
delay_auth_decision = true

# cache directory for signing certificate
signing_dir = /home/swift/keystone-signing

# auth_* settings refer to the Keystone server
auth_protocol = http
auth_host = 192.168.56.3
auth_port = 35357

# the same admin_token as provided in keystone.conf
admin_token = 012345SECRET99TOKEN012345

# the service tenant and swift userid and password created in Keystone
admin_tenant_name = service
admin_user = swift
admin_password = swift

[filter:cache]
use = egg:swift#memcache

[filter:catch_errors]
use = egg:swift#catch_errors

[filter:healthcheck]
use = egg:swift#healthcheck
```

Note

If you run multiple memcache servers, put the multiple IP:port listings in the [filter:cache] section of the proxy-server.conf file like:

```
10.1.2.3:11211,10.1.2.4:11211
```

Only the proxy server uses memcache.

7. Create the *signing_dir* and set its permissions accordingly.

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

8. Create the account, container and object rings. The builder command is basically creating a builder file with a few parameters. The parameter with the value of 18 represents 2 ^ 18th, the value that the partition will be sized to. Set this “partition power” value based on the total amount of storage you expect your entire ring to use.

The value of 3 represents the number of replicas of each object, with the last value being the number of hours to restrict moving a partition more than once.

```
# 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
```

9. For every storage device on each node add entries to each ring:

```
# swift-ring-builder account.builder add z<ZONE>-
<STORAGE_LOCAL_NET_IP>:6002/<DEVICE> 100
# swift-ring-builder container.builder add z<ZONE>-
<STORAGE_LOCAL_NET_IP_1>:6001/<DEVICE> 100
# swift-ring-builder object.builder add z<ZONE>-
<STORAGE_LOCAL_NET_IP_1>:6000/<DEVICE> 100
```

For example, if you were setting up a storage node with a partition in Zone 1 on IP 10.0.0.1. The mount point of this partition is /srv/node/sdb1, and the path in `rsyncd.conf` is /srv/node/, the DEVICE would be sdb1 and the commands would look like:

```
# swift-ring-builder account.builder add z1-10.0.0.1:6002/sdb1 100
# swift-ring-builder container.builder add z1-10.0.0.1:6001/sdb1 100
# swift-ring-builder object.builder add z1-10.0.0.1:6000/sdb1 100
```

Note

Assuming there are 5 zones with 1 node per zone, ZONE should start at 1 and increment by one for each additional node.

10. Verify the ring contents for each ring:

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

11. Rebalance the rings:

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

Note

Rebalancing rings can take some time.

12. Copy the `account.ring.gz`, `container.ring.gz`, and `object.ring.gz` files to each of the Proxy and Storage nodes in /etc/swift.

13. Make sure all the config files are owned by the swift user:

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

14. Start Proxy services:

```
# service proxy-server start
```

Start the Storage Nodes Services

Now that the ring files are on each storage node the services can be started. On each storage node run the following:

```
# service swift-object start
# service swift-object-replicator start
# service swift-object-updater start
# service swift-object-auditor start
# service swift-container start
# service swift-container-replicator start
# service swift-container-updater start
# service swift-container-auditor start
# service swift-account start
# service swift-account-replicator start
# service swift-account-updater start
# service swift-account-auditor start
```

OpenStack Object Storage Post Installation

Verify the Installation

You can run these commands from the proxy server or any server with access to the Identity Service.

1. First, export the swift admin password (setup previously) in a variable so it can be re-used.

```
$ export ADMINPASS=secrete
```

Note

If you do not wish to have the swift admin password stored in your shell's history, you may perform the following:

```
$ export SWIFT_PROXY_CONF="/etc/swift/proxy-server.conf export
 ADMINPASS=$( grep super_admin_key ${SWIFT_PROXY_CONF} | awk
 '{ print $NF }' )
```

2. Run the swift CLI, swift, with the correct Identity service URL. Export the information for ADMINPASS using `$ export ADMINPASS=secrete`.

```
$ swift -V 2.0 -A http://<AUTH_HOSTNAME>:5000/v2.0 -U demo:admin -K
 $ADMINPASS stat
```

3. Get an X-Storage-Url and X-Auth-Token:

```
$ curl -k -v -H 'X-Storage-User: demo:admin' -H 'X-Storage-Pass: $ADMINPASS'
 http://<AUTH_HOSTNAME>:5000/auth/v2.0
```

4. Check that you can HEAD the account:

```
$ curl -k -v -H 'X-Auth-Token: <token-from-x-auth-token-above>' <url-from-x-
storage-url-above>
```

5. Use swift to upload a few files named 'bigfile[1-2].tgz' to a container named 'myfiles':

```
$ swift -A http://<AUTH_HOSTNAME>:5000/v2.0 -U demo:admin -K $ADMINPASS
 upload myfiles bigfile1.tgz
$ swift -A http://<AUTH_HOSTNAME>:5000/v2.0 -U demo:admin -K $ADMINPASS
 upload myfiles bigfile2.tgz
```

6. Use swift to download all files from the 'myfiles' container:

```
$ swift -A http://<AUTH_HOSTNAME>:5000/v2.0 -U demo:admin -K $ADMINPASS
 download myfiles
```

Note

If you are using swauth in preference to the OpenStack Identity service, you should use the `default_swift_cluster` variable to connect to your swift cluster. Please follow the swauth documentation to verify your installation.

Adding an Additional Proxy Server

For reliability's sake you may want to have more than one proxy server. You can set up the additional proxy node in the same manner that you set up the first proxy node but with additional configuration steps.

Once you have more than two proxies, you also want to load balance between the two, which means your storage endpoint (what clients use to connect to your storage) also changes. You can select from different strategies for load balancing. For example, you could use round robin dns, or a software or hardware load balancer (like pound) in front of the two proxies, and point your storage url to the load balancer.

Configure an initial proxy node for the initial setup, and then follow these additional steps for more proxy servers.

1. Update the list of memcache servers in `/etc/swift/proxy-server.conf` for all the added proxy servers. If you run multiple memcache servers, use this pattern for the multiple IP:port listings:

```
10.1.2.3:11211,10.1.2.4:11211
```

   in each proxy server's conf file.:

```
[filter:cache]
use = egg:swift#memcache
memcache_servers = <PROXY_LOCAL_NET_IP>:11211
```

2. Next, copy all the ring information to all the nodes, including your new proxy nodes, and ensure the ring info gets to all the storage nodes as well.

3. After you sync all the nodes, make sure the admin has the keys in `/etc/swift` and the ownership for the ring file is correct.

Note

If you are using swauth in preference to the OpenStack Identity service, there are additional steps to follow for the addition of a second proxy server. Please follow the swauth documentation Installation section, paying close attention to the `default_swift_cluster` variable.

4. System Administration for OpenStack Object Storage

By understanding the concepts inherent to the Object Storage system you can better monitor and administer your storage solution.

Understanding How Object Storage Works

This section offers a brief overview of each concept in administering Object Storage.

The Ring

A ring represents a mapping between the names of entities stored on disk and their physical location. There are separate rings for accounts, containers, and objects. When other components need to perform any operation on an object, container, or account, they need to interact with the appropriate ring to determine its location in the cluster.

The Ring maintains this mapping using zones, devices, partitions, and replicas. Each partition in the ring is replicated, by default, 3 times across the cluster, and the locations for a partition are stored in the mapping maintained by the ring. The ring is also responsible for determining which devices are used for handoff in failure scenarios.

Data can be isolated with the concept of zones in the ring. Each replica of a partition is guaranteed to reside in a different zone. A zone could represent a drive, a server, a cabinet, a switch, or even a datacenter.

The partitions of the ring are equally divided among all the devices in the OpenStack Object Storage installation. When partitions need to be moved around (for example if a device is added to the cluster), the ring ensures that a minimum number of partitions are moved at a time, and only one replica of a partition is moved at a time.

Weights can be used to balance the distribution of partitions on drives across the cluster. This can be useful, for example, when different sized drives are used in a cluster.

The ring is used by the Proxy server and several background processes (like replication).

Proxy Server

The Proxy Server is responsible for tying together the rest of the OpenStack Object Storage architecture. For each request, it will look up the location of the account, container, or object in the ring (see below) and route the request accordingly. The public API is also exposed through the Proxy Server.

A large number of failures are also handled in the Proxy Server. For example, if a server is unavailable for an object PUT, it will ask the ring for a hand-off server and route there instead.

When objects are streamed to or from an object server, they are streamed directly through the proxy server to or from the user – the proxy server does not spool them.

You can use a proxy server with account management enabled by configuring it in the proxy server configuration file.

Object Server

The Object Server is a very simple blob storage server that can store, retrieve and delete objects stored on local devices. Objects are stored as binary files on the filesystem with metadata stored in the file's extended attributes (xattrs). This requires that the underlying filesystem choice for object servers support xattrs on files. Some filesystems, like ext3, have xattrs turned off by default.

Each object is stored using a path derived from the object name's hash and the operation's timestamp. Last write always wins, and ensures that the latest object version will be served. A deletion is also treated as a version of the file (a 0 byte file ending with ".ts", which stands for tombstone). This ensures that deleted files are replicated correctly and older versions don't magically reappear due to failure scenarios.

Container Server

The Container Server's primary job is to handle listings of objects. It doesn't know where those objects are, just what objects are in a specific container. The listings are stored as sqlite database files, and replicated across the cluster similar to how objects are. Statistics are also tracked that include the total number of objects, and total storage usage for that container.

Account Server

The Account Server is very similar to the Container Server, excepting that it is responsible for listings of containers rather than objects.

Replication

Replication is designed to keep the system in a consistent state in the face of temporary error conditions like network outages or drive failures.

The replication processes compare local data with each remote copy to ensure they all contain the latest version. Object replication uses a hash list to quickly compare subsections

of each partition, and container and account replication use a combination of hashes and shared high water marks.

Replication updates are push based. For object replication, updating is just a matter of rsyncing files to the peer. Account and container replication push missing records over HTTP or rsync whole database files.

The replicator also ensures that data is removed from the system. When an item (object, container, or account) is deleted, a tombstone is set as the latest version of the item. The replicator will see the tombstone and ensure that the item is removed from the entire system.

Updaters

There are times when container or account data can not be immediately updated. This usually occurs during failure scenarios or periods of high load. If an update fails, the update is queued locally on the file system, and the updater will process the failed updates. This is where an eventual consistency window will most likely come in to play. For example, suppose a container server is under load and a new object is put in to the system. The object will be immediately available for reads as soon as the proxy server responds to the client with success. However, the container server did not update the object listing, and so the update would be queued for a later update. Container listings, therefore, may not immediately contain the object.

In practice, the consistency window is only as large as the frequency at which the updater runs and may not even be noticed as the proxy server will route listing requests to the first container server which responds. The server under load may not be the one that serves subsequent listing requests – one of the other two replicas may handle the listing.

Auditors

Auditors crawl the local server checking the integrity of the objects, containers, and accounts. If corruption is found (in the case of bit rot, for example), the file is quarantined, and replication will replace the bad file from another replica. If other errors are found they are logged (for example, an object's listing can't be found on any container server it should be).

Server Configuration Reference

Swift uses paste.deploy to manage server configurations. Default configuration options are set in the `[DEFAULT]` section, and any options specified there can be overridden in any of the other sections.

Object Server Configuration

An example Object Server configuration can be found at etc/object-server.conf-sample in the source code repository.

The following configuration options are available:

Table 4.1. object-server.conf Default Options in the [DEFAULT] section

Option	Default	Description

swift_dir	/etc/swift	Swift configuration directory
devices	/srv/node	Parent directory of where devices are mounted
mount_check	true	Whether or not check if the devices are mounted to prevent accidentally writing to the root device
bind_ip	0.0.0.0	IP Address for server to bind to
bind_port	6000	Port for server to bind to
bind_timeout	30	Seconds to attempt bind before giving up
workers	1	Number of workers to fork
backlog	4096	Maximum number of allowed pending TCP connections.
expiring_objects_container_divisor	86400	
log_name	swift	Label used when logging
log_facility	LOG_LOCAL0	Syslog log facility
log_level	INFO	Logging level
user	swift	User to run as
db_preallocation	true	Preallocate disk space for new SQLite databases to decrease fragmentation
eventlet_debug	false	Turn on debug logging for eventlet
disable_fallocate	false	Disable "fast fail" fallocate checks if the underlying filesystem does not support it.
fallocate_reserve	0	The number of bytes for fallocate to reserve, whether there is space for the given file size or not. The default fallocate_reserve is 0, meaning "no reserve". Some systems behave badly when they completely run out of space. To alleviate this problem, you can set fallocate_reserve. When the disk free space falls at or below this amount, fallocate calls will fail, even if the underlying OS fallocate call would succeed. For example, a fallocate_reserve of 10737418240 (10G) would make all fallocate calls fail, even for zero-byte files, when the disk free space falls under 10G.

Table 4.2. object-server.conf Server Options in the [object-server] section

Option	Default	Description
use		The paste.deploy entry point for the object server. For most cases, this should be `egg:swift#object`.
log_name	object-server	Label used when logging
log_facility	LOG_LOCAL0	Syslog log facility
log_level	INFO	Logging level
log_requests	True	Whether or not to log each request
user	swift	User to run as
node_timeout	3	Request timeout to external services
conn_timeout	0.5	Connection timeout to external services

network_chunk_size	65536	Size of chunks to read/write over the network
disk_chunk_size	65536	Size of chunks to read/write to disk
max_upload_time	86400	Maximum time allowed to upload an object
slow	0	If > 0, Minimum time in seconds for a PUT or DELETE request to complete
auto_create_account_prefix	.	Prefix used when automatically creating accounts
allowed_headers	content-disposition, content-encoding, x-delete-at, x-object-manifest,	Comma-separated list of headers that can be set in metadata of an object
mb_per_sync	512	On PUTs, sync data every n MB

Table 4.3. object-server.conf Replicator Options in the [object-replicator] section

Option	Default	Description
log_name	object-replicator	Label used when logging
log_facility	LOG_LOCAL0	Syslog log facility
log_level	INFO	Logging level
daemonize	yes	Whether or not to run replication as a daemon
run_pause	30	Time in seconds to wait between replication passes
concurrency	1	Number of replication workers to spawn
timeout	5	Timeout value sent to rsync –timeout and –contimeout options
http_timeout	60	Maximum duration for an HTTP request
lockup_timeout	1800	Attempts to kill all workers if nothing replicates for lockup_timeout seconds.
stats_interval	300	Interval in seconds between logging replication statistics
reclaim_age	604800	Time elapsed in seconds before an object can be reclaimed
recon_cache_path	/var/cache/swift	Directory where stats for a few items will be stored
recon_enable	no	Enable logging of replication stats for recon
ring_check_interval	15	How often (in seconds) to check the ring
rsync_io_timeout	30	Passed to rsync for max duration (seconds) of an I/O op
rsync_timeout	900	Max duration (seconds) of a partition rsync
vm_test_mode	no	Indicates that you are using a VM environment

Table 4.4. object-server.conf Updater Options in the [object-updater] section

Option	Default	Description
log_name	object-updater	Label used when logging

log_facility	LOG_LOCAL0	Syslog log facility
log_level	INFO	Logging level
interval	300	Minimum time for a pass to take
concurrency	1	Number of updater workers to spawn
node_timeout	10	Request timeout to external services
conn_timeout	0.5	Connection timeout to external services
slowdown	0.01	Time in seconds to wait between objects

Table 4.5. object-server.conf Auditor Options in the [object-auditor] section

Option	Default	Description
log_name	object-auditor	Label used when logging
log_facility	LOG_LOCAL0	Syslog log facility
log_level	INFO	Logging level
log_time	3600	Frequency of status logs in seconds.
files_per_second	20	Maximum files audited per second. Should be tuned according to individual system specs. 0 is unlimited.
bytes_per_second	10000000	Maximum bytes audited per second. Should be tuned according to individual system specs. 0 is unlimited.
zero_byte_files_per_second	50	Maximum zero byte files audited per second.

When configured, the StaticWeb WSGI middleware serves container data as a static web site with index file and error file resolution and optional file listings. This mode is normally only active for anonymous requests.

Table 4.6. object-server.conf staticweb Options in the [filter:staticweb] section

Option	Default	Description
cache_timeout	300	Number of seconds to cache container x-container-meta-web-* header values.
log_facility	LOG_LOCAL0	Indicate which facility to use for logging staticweb requests.
log_level	INFO	Indicate which level of logging to use for staticweb requests.
log_name	staticweb	Indicate which level of logging to use for staticweb requests.
log_headers	f	Indicate which headers to use for logging staticweb requests.
access_log_level	'INFO'	Indicate which level of logging to use for staticweb container access.
access_log_facility	LOG_LOCAL0	Indicate which facility to use for logging access to staticweb containers.
access_log_name	staticweb	Indicate which level of logging to use for staticweb container access.

Container Server Configuration

An example Container Server configuration can be found at etc/container-server.conf-sample in the source code repository.

The following configuration options are available:

Table 4.7. container-server.conf Default Options in the [DEFAULT] section

Option	Default	Description
swift_dir	/etc/swift	Swift configuration directory
devices	/srv/node	Parent directory of where devices are mounted
mount_check	true	Whether or not check if the devices are mounted to prevent accidentally writing to the root device
bind_ip	0.0.0.0	IP Address for server to bind to
bind_port	6001	Port for server to bind to
bind_timeout	30	Seconds to attempt bind before giving up
workers	1	Number of workers to fork
user	swift	User to run as
db_preallocation	true	preallocate disk space for new SQLite databases to decrease fragmentation
eventlet_debug	false	turn on debug logging for eventlet
disable_fallocate	false	Disable "fast fail" fallocate checks if the underlying filesystem does not support it.
fallocate_reserve	0	The number of bytes for fallocate to reserve, whether there is space for the given file size or not. The default fallocate_reserve is 0, meaning "no reserve". Some systems behave badly when they completely run out of space. To alleviate this problem, you can set fallocate_reserve. When the disk free space falls at or below this amount, fallocate calls will fail, even if the underlying OS fallocate call would succeed. For example, a fallocate_reserve of 10737418240 (10G) would make all fallocate calls fail, even for zero-byte files, when the disk free space falls under 10G.
log_name	swift	Label used when logging
log_facility	LOG_LOCAL0	Syslog log facility
log_level	INFO	Logging level
auto_create_account_prefix	.	Prefix used when automatically creating accounts
backlog	4096	Maximum number of allowed pending TCP connections.

Table 4.8. container-server.conf Server Options in the [container-server] section

Option	Default	Description

use		The paste.deploy entry point for the container server. For most cases, this should be `egg:swift#container`.
log_name	container-server	Label used when logging
log_facility	LOG_LOCAL0	Syslog log facility
log_level	INFO	Logging level
log_requests	true	Whether or not to log each request
allow_versions	false	Whether to allow versions of containers
node_timeout	10	Request timeout to external services
conn_timeout	0.5	Connection timeout to external services

Table 4.9. container-server.conf Replicator Options in the [container-replicator] section

Option	Default	Description
log_name	container-replicator	Label used when logging
log_facility	LOG_LOCAL0	Syslog log facility
log_level	INFO	Logging level
per_diff	1000	Limit number of items to get per diff
concurrency	8	Number of replication workers to spawn
run_pause	30	Time in seconds to wait between replication passes
node_timeout	10	Request timeout to external services
conn_timeout	0.5	Connection timeout to external services
reclaim_age	604800	Time elapsed in seconds before a container can be reclaimed
interval	30	Minimum time for a pass to take (s)
max_diffs	100	Caps how long the replicator spends trying to sync a database per pass
vm_test_mode	no	Indicates that you are using a VM environment

Table 4.10. container-server.conf Updater Options in the [container-updater] section

Option	Default	Description
log_name	container-updater	Label used when logging
log_facility	LOG_LOCAL0	Syslog log facility
log_level	INFO	Logging level
interval	300	Minimum time for a pass to take
concurrency	4	Number of updater workers to spawn
node_timeout	3	Request timeout to external services
conn_timeout	0.5	Connection timeout to external services
slowdown	0.01	Time in seconds to wait between containers

account_supression_time	60	Seconds to suppress updating an account that has generated an error (timeout, not yet found, etc.)

Table 4.11. container-server.conf Auditor Options in the [container-auditor] section

Option	Default	Description
log_name	container-auditor	Label used when logging
log_facility	LOG_LOCAL0	Syslog log facility
log_level	INFO	Logging level
interval	1800	Minimum time for a pass to take
containers_per_second	200	Maximum containers audited per second. Should be tuned according to individual system specs. 0 is unlimited.

Table 4.12. container-server.conf Sync Options in the [container-sync] section

Option	Default	Description
log_name	container-sync	Label used when logging
log_facility	LOG_LOCAL0	Syslog log facility
log_level	INFO	Logging level
container_time	60	Maximum amount of time to spend syncing each container
sync_proxy		If you need to use an HTTP Proxy, set it here (eg http://127.0.0.1:8888); defaults to no proxy.

Account Server Configuration

An example Account Server configuration can be found at etc/account-server.conf-sample in the source code repository.

The following configuration options are available:

Table 4.13. account-server.conf Default Options in the [DEFAULT] section

Option	Default	Description
swift_dir	/etc/swift	Swift configuration directory
devices	/srv/node	Parent directory or where devices are mounted
mount_check	true	Whether or not check if the devices are mounted to prevent accidentally writing to the root device
bind_ip	0.0.0.0	IP Address for server to bind to
bind_port	6002	Port for server to bind to
bind_timeout	30	Seconds to attempt bind before giving up
workers	1	Number of workers to fork
user	swift	User to run as
backlog	4096	Maximum number of allowed pending TCP connections.

db_preallocation	true	preallocate disk space for new SQLite databases to decrease fragmentation
eventlet_debug	false	turn on debug logging for eventlet
disable_fallocate	false	Disable "fast fail" fallocate checks if the underlying filesystem does not support it.
fallocate_reserve	0	The number of bytes for fallocate to reserve, whether there is space for the given file size or not. The default fallocate_reserve is 0, meaning "no reserve". Some systems behave badly when they completely run out of space. To alleviate this problem, you can set fallocate_reserve. When the disk free space falls at or below this amount, fallocate calls will fail, even if the underlying OS fallocate call would succeed. For example, a fallocate_reserve of 10737418240 (10G) would make all fallocate calls fail, even for zero-byte files, when the disk free space falls under 10G.

Table 4.14. account-server.conf Server Options in the [account-server] section

Option	Default	Description
use		Entry point for paste.deploy for the account server. For most cases, this should be `egg:swift#account`.
log_name	account-server	Label used when logging
log_facility	LOG_LOCAL0	Syslog log facility
log_level	INFO	Logging level
log_requests	true	Whether or not to log each request
auto_create_account_prefix	.	prefix used when automatically creating accounts

Table 4.15. account-server.conf Replicator Options in the [account-replicator] section

Option	Default	Description
log_name	account-replicator	Label used when logging
log_facility	LOG_LOCAL0	Syslog log facility
log_level	INFO	Logging level
per_diff	1000	Limit number of items to get per diff
max_diffs	100	Caps how long the replicator spends trying to sync a database per pass
concurrency	8	Number of replication workers to spawn
run_pause	30	Time in seconds to wait between replication passes
node_timeout	10	Request timeout to external services
conn_timeout	0.5	Connection timeout to external services
reclaim_age	604800	Time elapsed in seconds before an account can be reclaimed

vm_test_mode	no	Indicates that you are using a VM environment
interval	30	Minimum time for a pass to take
error_suppression_interval	60	Time in seconds that must elapse since the last error for a node to be considered no longer error limited
error_suppression_limit	10	Error count to consider a node error limited

Table 4.16. account-server.conf Auditor Options in the [account-auditor] section

Option	Default	Description
log_name	account-auditor	Label used when logging
log_facility	LOG_LOCAL0	Syslog log facility
log_level	INFO	Logging level
interval	1800	Minimum time for a pass to take
accounts_per_second	200	Maximum accounts audited per second. Should be tuned according to individual system specs. 0 is unlimited.

Table 4.17. account-server.conf Reaper Options in the [account-reaper] section

Option	Default	Description
delay_reaping	0	Number of seconds to delay reaper from deleting account info for deleted accounts. 2592000 = 30 days, for example
log_name	account-auditor	Label used when logging
log_facility	LOG_LOCAL0	Syslog log facility
log_level	INFO	Logging level
concurrency	25	Number of replication workers to spawn
interval	3600	Minimum time for a pass to take
node_timeout	10	Request timeout to external services
conn_timeout	0.5	Connection timeout to external services

Proxy Server Configuration

An example Proxy Server configuration can be found at etc/proxy-server.conf-sample in the source code repository.

The following configuration options are available:

Table 4.18. proxy-server.conf Default Options in the [DEFAULT] section

Option	Default	Description
bind_ip	0.0.0.0	IP Address for server to bind to
bind_port	80	Port for server to bind to

bind_timeout	30	Seconds to attempt bind before giving up
swift_dir	/etc/swift	Swift configuration directory
workers	1	Number of workers to fork
user	swift	User to run as
cert_file		Path to the ssl .crt
key_file		Path to the ssl .key
eventlet_debug	false	turn on debug logging for eventlet

Table 4.19. proxy-server.conf Server Options in the [proxy-server] section

Option	Default	Description
use		Entry point for paste.deploy for the proxy server. For most cases, this should be `egg:swift#proxy`.
log_name	proxy-server	Label used when logging
log_facility	LOG_LOCAL0	Syslog log facility
log_level	INFO	Log level
log_headers	True	If True, log headers in each request
recheck_account_existence	60	Cache timeout in seconds to send memcached for account existence
recheck_container_existence	60	Cache timeout in seconds to send memcached for container existence
object_chunk_size	65536	Chunk size to read from object servers
client_chunk_size	65536	Chunk size to read from clients
memcache_servers	127.0.0.1:11211	Comma separated list of memcached servers ip:port.
node_timeout	10	Request timeout to external services
client_timeout	60	Timeout to read one chunk from a client
conn_timeout	0.5	Connection timeout to external services
error_suppression_interval	60	Time in seconds that must elapse since the last error for a node to be considered no longer error limited
error_suppression_limit	10	Error count to consider a node error limited
allow_account_management	false	Whether account PUTs and DELETEs are even callable

Table 4.20. proxy-server.conf Paste.deploy Options in the [filter:swauth] section

Option	Default	Description
use		Entry point for paste.deploy to use for auth, set to: `egg:swauth#swauth` to use the swauth downloaded from https://github.com/gholt/swauth
log_name	auth-server	Label used when logging
log_facility	LOG_LOCAL0	Syslog log facility
log_level	INFO	Log level
log_headers	True	If True, log headers in each request

reseller_prefix	AUTH	The naming scope for the auth service. Swift storage accounts and auth tokens will begin with this prefix.
auth_prefix	/auth/	The HTTP request path prefix for the auth service. Swift itself reserves anything beginning with the letter v.
default_swift_cluster	local#http://127.0.0.1:8080/v1	The default Swift cluster to place newly created accounts on; only needed if you are using Swauth for authentication.
token_life	86400	The number of seconds a token is valid.
node_timeout	10	Request timeout
super_admin_key	None	The key for the .super_admin account.

Table 4.21. proxy-server.conf Server Options in the [filter:cache] section

Option	Default	Description
log_name	cache	Label used when logging
log_facility	LOG_LOCAL0	Syslog log facility
log_level	INFO	Log level
log_headers	False	If True, log headers in each request
memcache_servers	127.0.0.1:11211	Comma separated list of memcached servers ip:port. Note: the value in here will be ignored if it is also specified in /etc/swift/proxy-server.conf

Object Layout on Storage

Swift uses the underlying filesystem to store the data on disk. An administrator can use normal filesystem tools to find and inspect this data. Swift uses the following convention for storing objects:

*/path_to_mount_points/device/*objects*/partition/hash_suffix/hash/*

Accounts and containers are stored similarly, with the "objects" part of the path replaced with "accounts" or "containers".

The directory is where all of the data for the object is stored. Inside the directory, there is normally just one file (named *<timestamp>*.data). The object's data is stored in the file, and the object's metadata is stored in the extended attributes (xattrs) of the file.

If a user deletes the object, the .data file is deleted and a *<timestamp>*.ts ("ts" for "tombstone") file is created as a zero-byte file. This is a delete marker that will be eventually reaped, but it exists to ensure that the delete properly propagates to all replicas in the cluster.

Swift uses this scheme of timestamps in the file name to implement conflict resolution. Each piece of data is assigned a timestamp by the proxy server when the request comes to the cluster, and that timestamp is what's used to name the file on disk on the object (or account or container) server. For the common case, there will be a single file in the directory. If Swift receives multiple, concurrent requests to write to the same object, it can

happen that multiple `.data` files are written to this directory. Swift uses "last-write-wins" to resolve such conflicts, choosing the most recent file by timestamp.

Configuring and Tuning OpenStack Object Storage

This section walks through deployment options and considerations.

You have multiple deployment options to choose from. The swift services run completely autonomously, which provides for a lot of flexibility when designing the hardware deployment for swift. The 4 main services are:

- Proxy Services
- Object Services
- Container Services
- Account Services

The Proxy Services are more CPU and network I/O intensive. If you are using 10g networking to the proxy, or are terminating SSL traffic at the proxy, greater CPU power will be required.

The Object, Container, and Account Services (Storage Services) are more disk and network I/O intensive.

The easiest deployment is to install all services on each server. There is nothing wrong with doing this, as it scales each service out horizontally.

At Rackspace, we put the Proxy Services on their own servers and all of the Storage Services on the same server. This allows us to send 10g networking to the proxy and 1g to the storage servers, and keep load balancing to the proxies more manageable. Storage Services scale out horizontally as storage servers are added, and we can scale overall API throughput by adding more Proxies.

If you need more throughput to either Account or Container Services, they may each be deployed to their own servers. For example you might use faster (but more expensive) SAS or even SSD drives to get faster disk I/O to the databases.

Load balancing and network design is left as an exercise to the reader, but this is a very important part of the cluster, so time should be spent designing the network for a Swift cluster.

Preparing the Ring

Note

"Partition" in this section refers to the logical partitions of the swift ring - not physical partitions on Storage node drives. You should setup your Storage

Node disk partitions with one physical partition per disk, as per the installation instructions .

The first step is to determine the number of partitions that will be in the ring. We recommend that there be a minimum of 100 partitions per drive to insure even distribution across the drives. A good starting point might be to figure out the maximum number of drives the cluster will contain, and then multiply by 100, and then round up to the nearest power of two.

For example, imagine we are building a cluster that will have no more than 5,000 drives. That would mean that we would have a total number of 500,000 partitions, which is pretty close to 2^19, rounded up.

It is also a good idea to keep the number of partitions small (relatively). The more partitions there are, the more work that has to be done by the replicators and other backend jobs and the more memory the rings consume in process. The goal is to find a good balance between small rings and maximum cluster size.

The next step is to determine the number of replicas to store of the data. Currently it is recommended to use 3 (as this is the only value that has been tested). The higher the number, the more storage that is used but the less likely you are to lose data.

It is also important to determine how many zones the cluster should have. It is recommended to start with a minimum of 5 zones. You can start with fewer, but our testing has shown that having at least five zones is optimal when failures occur. We also recommend trying to configure the zones at as high a level as possible to create as much isolation as possible. Some example things to take into consideration can include physical location, power availability, and network connectivity. For example, in a small cluster you might decide to split the zones up by cabinet, with each cabinet having its own power and network connectivity. The zone concept is very abstract, so feel free to use it in whatever way best isolates your data from failure. Zones are referenced by number, beginning with 1.

You can now start building the ring with:

```
swift-ring-builder <builder_file> create <part_power> <replicas> <min_part_hours>
```

This will start the ring build process creating the <builder_file> with 2^<part_power> partitions. <min_part_hours> is the time in hours before a specific partition can be moved in succession (24 is a good value for this).

Devices can be added to the ring with:

```
swift-ring-builder <builder_file> add z<zone>-<ip>:<port>/<device_name>_<meta> <weight>
```

This will add a device to the ring where <builder_file> is the name of the builder file that was created previously, <zone> is the number of the zone this device is in, <ip> is the ip address of the server the device is in, <port> is the port number that the server is running on, <device_name> is the name of the device on the server (for example: sdb1), <meta> is a string of metadata for the device (optional), and <weight> is a float weight that determines how many partitions are put on the device relative to the rest of the devices in the cluster (a good starting point is 100.0 x TB on the drive). Add each device that will be initially in the cluster.

Once all of the devices are added to the ring, run:

```
swift-ring-builder <builder_file> rebalance
```

This will distribute the partitions across the drives in the ring. It is important whenever making changes to the ring to make all the changes required before running rebalance. This will ensure that the ring stays as balanced as possible, and as few partitions are moved as possible.

The above process should be done to make a ring for each storage service (Account, Container and Object). The builder files will be needed in future changes to the ring, so it is very important that these be kept and backed up. The resulting .tar.gz ring file should be pushed to all of the servers in the cluster. For more information about building rings, running swift-ring-builder with no options will display help text with available commands and options.

Considerations and Tuning

Fine-tuning your deployment and installation may take some time and effort. Here are some considerations for improving performance of an OpenStack Object Storage installation.

Memcached Considerations

Several of the Services rely on Memcached for caching certain types of lookups, such as auth tokens, and container/account existence. Swift does not do any caching of actual object data. Memcached should be able to run on any servers that have available RAM and CPU. At Rackspace, we run Memcached on the proxy servers. The `memcache_servers` config option in the `proxy-server.conf` should contain all memcached servers.

System Time

Time may be relative but it is relatively important for Swift! Swift uses timestamps to determine which is the most recent version of an object. It is very important for the system time on each server in the cluster to by synced as closely as possible (more so for the proxy server, but in general it is a good idea for all the servers). At Rackspace, we use NTP with a local NTP server to ensure that the system times are as close as possible. This should also be monitored to ensure that the times do not vary too much.

General Service Tuning

Most services support either a worker or concurrency value in the settings. This allows the services to make effective use of the cores available. A good starting point to set the concurrency level for the proxy and storage services to 2 times the number of cores available. If more than one service is sharing a server, then some experimentation may be needed to find the best balance.

At Rackspace, our Proxy servers have dual quad core processors, giving us 8 cores. Our testing has shown 16 workers to be a pretty good balance when saturating a 10g network and gives good CPU utilization.

Our Storage servers all run together on the same servers. These servers have dual quad core processors, for 8 cores total. We run the Account, Container, and Object servers with 8 workers each. Most of the background jobs are run at a concurrency of 1, with the exception of the replicators which are run at a concurrency of 2.

The above configuration setting should be taken as suggestions and testing of configuration settings should be done to ensure best utilization of CPU, network connectivity, and disk I/O.

RAID Considerations

We recommend that you do not use RAID with Swift.

The workload for Swift is very write-heavy, with small random IO accesses. This type of workload performs very poorly for most parity RAID (e.g., RAID 2-6). Testing done by Rackspace suggests that under heavy workloads, the overall RAID performance can degrade to be as slow as a single drive.

Furthermore, a drive failure in a RAID array can result in very poor performance on the node until the RAID rebuilds, which can take a long time. Testing at Rackspace, using nodes with 24 2T drives, revealed that a RAID rebuild after a drive failure could take on the order of two weeks, during which time the node performance suffered dramatically as the RAID array functioned in a degraded state. This kind of significantly degraded performance can potentially have ripple effects across the rest of the cluster.

Filesystem Considerations

Swift is designed to be mostly filesystem agnostic–the only requirement being that the filesystem supports extended attributes (xattrs). After thorough testing with our use cases and hardware configurations, XFS was the best all-around choice. If you decide to use a filesystem other than XFS, we highly recommend thorough testing.

If you are using XFS, some settings that can dramatically impact performance. We recommend the following when creating the XFS partition:

```
mkfs.xfs -i size=1024 -f /dev/sda1
```

Setting the inode size is important, as XFS stores xattr data in the inode. If the metadata is too large to fit in the inode, a new extent is created, which can cause quite a performance problem. Upping the inode size to 1024 bytes provides enough room to write the default metadata, plus a little headroom. We do not recommend running Swift on RAID, but if you are using RAID it is also important to make sure that the proper sunit and swidth settings get set so that XFS can make most efficient use of the RAID array.

We also recommend the following example mount options when using XFS:

```
mount -t xfs -o noatime,nodiratime,nobarrier,logbufs=8 /dev/sda1 /
srv/node/sda
```

For a standard swift install, all data drives are mounted directly under /srv/node (as can be seen in the above example of mounting /dev/sda1 as /srv/node/sda). If you choose to

mount the drives in another directory, be sure to set the `devices` config option in all of the server configs to point to the correct directory.

General System Tuning

Rackspace currently runs Swift on Ubuntu Server 10.04, and the following changes have been found to be useful for our use cases.

The following settings should be in `/etc/sysctl.conf`:

```
# disable TIME_WAIT.. wait..
net.ipv4.tcp_tw_recycle=1
net.ipv4.tcp_tw_reuse=1

# disable syn cookies
net.ipv4.tcp_syncookies = 0

# double amount of allowed conntrack
net.ipv4.netfilter.ip_conntrack_max = 262144
```

To load the updated sysctl settings, run `sudo sysctl -p`

A note about changing the TIME_WAIT values. By default the OS will hold a port open for 60 seconds to ensure that any remaining packets can be received. During high usage, and with the number of connections that are created, it is easy to run out of ports. We can change this since we are in control of the network. If you are not in control of the network, or do not expect high loads, then you may not want to adjust those values.

Another helpful tuning parameter on slower systems that helps to ensure enough time is allowed for service restarts is the `-k N` (or `--kill-wait N`) parameter of **swift-init**. This allows you to change the default (15 second) time (N, in seconds) that swift waits for processes to die and notably you can pass **–run-dir** flag with **swift-init** to set where PID's will be stored.

Logging Considerations

Swift is set up to log directly to syslog. Every service can be configured with the `log_facility` option to set the syslog log facility destination. We recommend using syslog-ng to route the logs to specific log files locally on the server and also to remote log collecting servers.

Working with Rings

The rings determine where data should reside in the cluster. There is a separate ring for account databases, container databases, and individual objects but each ring works in the same way. These rings are externally managed, in that the server processes themselves do not modify the rings, they are instead given new rings modified by other tools.

The ring uses a configurable number of bits from a path's MD5 hash as a partition index that designates a device. The number of bits kept from the hash is known as the partition power, and 2 to the partition power indicates the partition count. Partitioning the full MD5 hash ring allows other parts of the cluster to work in batches of items at once which ends up either more efficient or at least less complex than working with each item separately or the entire cluster all at once.

Another configurable value is the replica count, which indicates how many of the partition->device assignments comprise a single ring. For a given partition number, each replica's device will not be in the same zone as any other replica's device. Zones can be used to group devices based on physical locations, power separations, network separations, or any other attribute that would lessen multiple replicas being unavailable at the same time.

Managing Rings with the Ring Builder

The rings are built and managed manually by a utility called the ring-builder. The ring-builder assigns partitions to devices and writes an optimized Python structure to a gzipped, pickled file on disk for shipping out to the servers. The server processes just check the modification time of the file occasionally and reload their in-memory copies of the ring structure as needed. Because of how the ring-builder manages changes to the ring, using a slightly older ring usually just means one of the three replicas for a subset of the partitions will be incorrect, which can be easily worked around.

The ring-builder also keeps its own builder file with the ring information and additional data required to build future rings. It is very important to keep multiple backup copies of these builder files. One option is to copy the builder files out to every server while copying the ring files themselves. Another is to upload the builder files into the cluster itself. Complete loss of a builder file will mean creating a new ring from scratch, nearly all partitions will end up assigned to different devices, and therefore nearly all data stored will have to be replicated to new locations. So, recovery from a builder file loss is possible, but data will definitely be unreachable for an extended time.

About the Ring Data Structure

The ring data structure consists of three top level fields: a list of devices in the cluster, a list of lists of device ids indicating partition to device assignments, and an integer indicating the number of bits to shift an MD5 hash to calculate the partition for the hash.

List of Devices in the Ring

The list of devices is known internally to the Ring class as devs. Each item in the list of devices is a dictionary with the following keys:

Table 4.22. List of Devices and Keys

Key	Type	Description
id	integer	The index into the list devices.
zone	integer	The zone the devices resides in.
weight	float	The relative weight of the device in comparison to other devices. This usually corresponds directly to the amount of disk space the device has compared to other devices. For

		instance a device with 1 terabyte of space might have a weight of 100.0 and another device with 2 terabytes of space might have a weight of 200.0. This weight can also be used to bring back into balance a device that has ended up with more or less data than desired over time. A good average weight of 100.0 allows flexibility in lowering the weight later if necessary.
ip	string	The IP address of the server containing the device.
port	int	The TCP port the listening server process uses that serves requests for the device.
device	string	The on disk name of the device on the server. For example: sdb1
meta	string	A general-use field for storing additional information for the device. This information isn't used directly by the server processes, but can be useful in debugging. For example, the date and time of installation and hardware manufacturer could be stored here.

Note: The list of devices may contain holes, or indexes set to None, for devices that have been removed from the cluster. Generally, device ids are not reused. Also, some devices may be temporarily disabled by setting their weight to 0.0.

Partition Assignment List

This is a list of array('I') of devices ids. The outermost list contains an array('I') for each replica. Each array('I') has a length equal to the partition count for the ring. Each integer in the array('I') is an index into the above list of devices. The partition list is known internally to the Ring class as _replica2part2dev_id.

So, to create a list of device dictionaries assigned to a partition, the Python code would look like: devices = [self.devs[part2dev_id[partition]] for part2dev_id in self._replica2part2dev_id]

array('I') is used for memory conservation as there may be millions of partitions.

Partition Shift Value

The partition shift value is known internally to the Ring class as _part_shift. This value used to shift an MD5 hash to calculate the partition on which the data for that hash should reside. Only the top four bytes of the hash is used in this process. For example, to compute the partition for the path /account/container/object the Python code might look like: partition = unpack_from('>I', md5('/account/container/object').digest())[0] >>self._part_shift

Building the Ring

The initial building of the ring first calculates the number of partitions that should ideally be assigned to each device based the device's weight. For example, if the partition power of 20 the ring will have 1,048,576 partitions. If there are 1,000 devices of equal weight they will

each desire 1,048.576 partitions. The devices are then sorted by the number of partitions they desire and kept in order throughout the initialization process.

Then, the ring builder assigns each partition's replica to the device that desires the most partitions at that point, with the restriction that the device is not in the same zone as any other replica for that partition. Once assigned, the device's desired partition count is decremented and moved to its new sorted location in the list of devices and the process continues.

When building a new ring based on an old ring, the desired number of partitions each device wants is recalculated. Next the partitions to be reassigned are gathered up. Any removed devices have all their assigned partitions unassigned and added to the gathered list. Any devices that have more partitions than they now desire have random partitions unassigned from them and added to the gathered list. Lastly, the gathered partitions are then reassigned to devices using a similar method as in the initial assignment described above.

Whenever a partition has a replica reassigned, the time of the reassignment is recorded. This is taken into account when gathering partitions to reassign so that no partition is moved twice in a configurable amount of time. This configurable amount of time is known internally to the RingBuilder class as min_part_hours. This restriction is ignored for replicas of partitions on devices that have been removed, as removing a device only happens on device failure and there's no choice but to make a reassignment.

The above processes don't always perfectly rebalance a ring due to the random nature of gathering partitions for reassignment. To help reach a more balanced ring, the rebalance process is repeated until near perfect (less 1% off) or when the balance doesn't improve by at least 1% (indicating we probably can't get perfect balance due to wildly imbalanced zones or too many partitions recently moved).

History of the Ring Design

The ring code went through many iterations before arriving at what it is now and while it has been stable for a while now, the algorithm may be tweaked or perhaps even fundamentally changed if new ideas emerge. This section will try to describe the previous ideas attempted and attempt to explain why they were discarded.

A “live ring” option was considered where each server could maintain its own copy of the ring and the servers would use a gossip protocol to communicate the changes they made. This was discarded as too complex and error prone to code correctly in the project time span available. One bug could easily gossip bad data out to the entire cluster and be difficult to recover from. Having an externally managed ring simplifies the process, allows full validation of data before it's shipped out to the servers, and guarantees each server is using a ring from the same timeline. It also means that the servers themselves aren't spending a lot of resources maintaining rings.

A couple of “ring server” options were considered. One was where all ring lookups would be done by calling a service on a separate server or set of servers, but this was discarded due to the latency involved. Another was much like the current process but where servers could submit change requests to the ring server to have a new ring built and shipped back out to the servers. This was discarded due to project time constraints and because ring changes are currently infrequent enough that manual control was sufficient. However, lack of quick automatic ring changes did mean that other parts of the system had to be coded

to handle devices being unavailable for a period of hours until someone could manually update the ring.

The current ring process has each replica of a partition independently assigned to a device. A version of the ring that used a third of the memory was tried, where the first replica of a partition was directly assigned and the other two were determined by “walking” the ring until finding additional devices in other zones. This was discarded as control was lost as to how many replicas for a given partition moved at once. Keeping each replica independent allows for moving only one partition replica within a given time window (except due to device failures). Using the additional memory was deemed a good tradeoff for moving data around the cluster much less often.

Another ring design was tried where the partition to device assignments weren't stored in a big list in memory but instead each device was assigned a set of hashes, or anchors. The partition would be determined from the data item's hash and the nearest device anchors would determine where the replicas should be stored. However, to get reasonable distribution of data each device had to have a lot of anchors and walking through those anchors to find replicas started to add up. In the end, the memory savings wasn't that great and more processing power was used, so the idea was discarded.

A completely non-partitioned ring was also tried but discarded as the partitioning helps many other parts of the system, especially replication. Replication can be attempted and retried in a partition batch with the other replicas rather than each data item independently attempted and retried. Hashes of directory structures can be calculated and compared with other replicas to reduce directory walking and network traffic.

Partitioning and independently assigning partition replicas also allowed for the best balanced cluster. The best of the other strategies tended to give +-10% variance on device balance with devices of equal weight and +-15% with devices of varying weights. The current strategy allows us to get +-3% and +-8% respectively.

Various hashing algorithms were tried. SHA offers better security, but the ring doesn't need to be cryptographically secure and SHA is slower. Murmur was much faster, but MD5 was built-in and hash computation is a small percentage of the overall request handling time. In all, once it was decided the servers wouldn't be maintaining the rings themselves anyway and only doing hash lookups, MD5 was chosen for its general availability, good distribution, and adequate speed.

The Account Reaper

The Account Reaper removes data from deleted accounts in the background.

An account is marked for deletion by a reseller through the services server's remove_storage_account XMLRPC call. This simply puts the value DELETED into the status column of the account_stat table in the account database (and replicas), indicating the data for the account should be deleted later. There is no set retention time and no undelete; it is assumed the reseller will implement such features and only call remove_storage_account once it is truly desired the account's data be removed.

The account reaper runs on each account server and scans the server occasionally for account databases marked for deletion. It will only trigger on accounts that server is the primary node for, so that multiple account servers aren't all trying to do the same work at

the same time. Using multiple servers to delete one account might improve deletion speed, but requires coordination so they aren't duplicating effort. Speed really isn't as much of a concern with data deletion and large accounts aren't deleted that often.

The deletion process for an account itself is pretty straightforward. For each container in the account, each object is deleted and then the container is deleted. Any deletion requests that fail won't stop the overall process, but will cause the overall process to fail eventually (for example, if an object delete times out, the container won't be able to be deleted later and therefore the account won't be deleted either). The overall process continues even on a failure so that it doesn't get hung up reclaiming cluster space because of one troublesome spot. The account reaper will keep trying to delete an account until it eventually becomes empty, at which point the database reclaim process within the db_replicator will eventually remove the database files.

Account Reaper Background and History

At first, a simple approach of deleting an account through completely external calls was considered as it required no changes to the system. All data would simply be deleted in the same way the actual user would, through the public REST API. However, the downside was that it would use proxy resources and log everything when it didn't really need to. Also, it would likely need a dedicated server or two, just for issuing the delete requests.

A completely bottom-up approach was also considered, where the object and container servers would occasionally scan the data they held and check if the account was deleted, removing the data if so. The upside was the speed of reclamation with no impact on the proxies or logging, but the downside was that nearly 100% of the scanning would result in no action creating a lot of I/O load for no reason.

A more container server centric approach was also considered, where the account server would mark all the containers for deletion and the container servers would delete the objects in each container and then themselves. This has the benefit of still speedy reclamation for accounts with a lot of containers, but has the downside of a pretty big load spike. The process could be slowed down to alleviate the load spike possibility, but then the benefit of speedy reclamation is lost and what's left is just a more complex process. Also, scanning all the containers for those marked for deletion when the majority wouldn't be seemed wasteful. The db_replicator could do this work while performing its replication scan, but it would have to spawn and track deletion processes which seemed needlessly complex.

In the end, an account server centric approach seemed best, as described above.

Replication

Since each replica in OpenStack Object Storage functions independently, and clients generally require only a simple majority of nodes responding to consider an operation successful, transient failures like network partitions can quickly cause replicas to diverge. These differences are eventually reconciled by asynchronous, peer-to-peer replicator processes. The replicator processes traverse their local filesystems, concurrently performing operations in a manner that balances load across physical disks.

Replication uses a push model, with records and files generally only being copied from local to remote replicas. This is important because data on the node may not belong there (as

in the case of handoffs and ring changes), and a replicator can't know what data exists elsewhere in the cluster that it should pull in. It's the duty of any node that contains data to ensure that data gets to where it belongs. Replica placement is handled by the ring.

Every deleted record or file in the system is marked by a tombstone, so that deletions can be replicated alongside creations. These tombstones are cleaned up by the replication process after a period of time referred to as the consistency window, which is related to replication duration and how long transient failures can remove a node from the cluster. Tombstone cleanup must be tied to replication to reach replica convergence.

If a replicator detects that a remote drive has failed, it will use the ring's “get_more_nodes” interface to choose an alternate node to synchronize with. The replicator can generally maintain desired levels of replication in the face of hardware failures, though some replicas may not be in an immediately usable location.

Replication is an area of active development, and likely rife with potential improvements to speed and correctness.

There are two major classes of replicator - the db replicator, which replicates accounts and containers, and the object replicator, which replicates object data.

Database Replication

The first step performed by db replication is a low-cost hash comparison to find out whether or not two replicas already match. Under normal operation, this check is able to verify that most databases in the system are already synchronized very quickly. If the hashes differ, the replicator brings the databases in sync by sharing records added since the last sync point.

This sync point is a high water mark noting the last record at which two databases were known to be in sync, and is stored in each database as a tuple of the remote database id and record id. Database ids are unique amongst all replicas of the database, and record ids are monotonically increasing integers. After all new records have been pushed to the remote database, the entire sync table of the local database is pushed, so the remote database knows it's now in sync with everyone the local database has previously synchronized with.

If a replica is found to be missing entirely, the whole local database file is transmitted to the peer using rsync(1) and vested with a new unique id.

In practice, DB replication can process hundreds of databases per concurrency setting per second (up to the number of available CPUs or disks) and is bound by the number of DB transactions that must be performed.

Object Replication

The initial implementation of object replication simply performed an rsync to push data from a local partition to all remote servers it was expected to exist on. While this performed adequately at small scale, replication times skyrocketed once directory structures could no longer be held in RAM. We now use a modification of this scheme in which a hash of the contents for each suffix directory is saved to a per-partition hashes file. The hash for a suffix directory is invalidated when the contents of that suffix directory are modified.

The object replication process reads in these hash files, calculating any invalidated hashes. It then transmits the hashes to each remote server that should hold the partition, and only suffix directories with differing hashes on the remote server are rsynced. After pushing files to the remote server, the replication process notifies it to recalculate hashes for the rsynced suffix directories.

Performance of object replication is generally bound by the number of uncached directories it has to traverse, usually as a result of invalidated suffix directory hashes. Using write volume and partition counts from our running systems, it was designed so that around 2% of the hash space on a normal node will be invalidated per day, which has experimentally given us acceptable replication speeds.

Managing Large Objects (Greater than 5 GB)

OpenStack Object Storage has a limit on the size of a single uploaded object; by default this is 5GB. However, the download size of a single object is virtually unlimited with the concept of segmentation. Segments of the larger object are uploaded and a special manifest file is created that, when downloaded, sends all the segments concatenated as a single object. This also offers much greater upload speed with the possibility of parallel uploads of the segments.

Using swift to Manage Segmented Objects

The quickest way to try out this feature is use the included swift OpenStack Object Storage client tool. You can use the -S option to specify the segment size to use when splitting a large file. For example:

```
# swift upload test_container -S 1073741824 large_file
```

This would split the large_file into 1G segments and begin uploading those segments in parallel. Once all the segments have been uploaded, swift will then create the manifest file so the segments can be downloaded as one.

So now, the following st command would download the entire large object:

```
swift download test_container large_file
```

The swift CLI uses a strict convention for its segmented object support.
In the above example it will upload all the segments into a second
container named test_container_segments. These segments will
have names like large_file/1290206778.25/21474836480/00000000,
large_file/1290206778.25/21474836480/00000001, etc.

The main benefit for using a separate container is that the main container listings will not be polluted with all the segment names. The reason for using the segment name format of <name>/<timestamp>/<size>/ is so that an upload of a new file with the same name won't overwrite the contents of the first until the last moment when the manifest file is updated.

The swift CLI will manage these segment files for you, deleting old segments on deletes and overwrites, etc. You can override this behavior with the –leave-segments option if desired; this is useful if you want to have multiple versions of the same large object available.

Direct API Management of Large Objects

You can also work with the segments and manifests directly with HTTP requests instead of having swift do that for you. You can just upload the segments like you would any other object and the manifest is just a zero-byte file with an extra X-Object-Manifest header.

All the object segments need to be in the same container, have a common object name prefix, and their names sort in the order they should be concatenated. They don't have to be in the same container as the manifest file will be, which is useful to keep container listings clean as explained above with st.

The manifest file is simply a zero-byte file with the extra X-Object-Manifest:`<container>/<prefix>` `header`, where <container> is the container the object segments are in and <prefix> is the common prefix for all the segments.

It is best to upload all the segments first and then create or update the manifest. In this way, the full object won't be available for downloading until the upload is complete. Also, you can upload a new set of segments to a second location and then update the manifest to point to this new location. During the upload of the new segments, the original manifest will still be available to download the first set of segments.

Here's an example using curl with tiny 1-byte segments:

First, upload the segments

```
$ curl -X PUT -H 'X-Auth-Token: <token>' http://<storage_url>/container/
myobject/1 --data-binary '1'
$ curl -X PUT -H 'X-Auth-Token: <token>' http://<storage_url>/container/
myobject/2 --data-binary '2'
$ curl -X PUT -H 'X-Auth-Token: <token>' http://<storage_url>/container/
myobject/3 --data-binary '3'
```

Next, create the manifest file curl -X PUT -H 'X-Auth-Token: <token>' \ -H 'X-Object-Manifest: container/myobject/' \ http://<storage_url>/container/myobject –data-binary '' # And now we can download the segments as a single object curl -H 'X-Auth-Token: <token>' \ http://<storage_url>/container/myobject

Additional Notes on Large Objects

- With a GET or HEAD of a manifest file, the X-Object-Manifest: `<container>/<prefix>` header will be returned with the concatenated object so you can tell where it's getting its segments from.
- The response's Content-Length for a GET or HEAD on the manifest file will be the sum of all the segments in the <container>/<prefix> listing, dynamically. So, uploading additional segments after the manifest is created will cause the concatenated object to be that much larger; there's no need to recreate the manifest file.
- The response's Content-Type for a GET or HEAD on the manifest will be the same as the Content-Type set during the PUT request that created the manifest. You can easily change the Content-Type by reissuing the PUT.

- The response's ETag for a GET or HEAD on the manifest file will be the MD5 sum of the concatenated string of ETags for each of the segments in the <container>/<prefix> listing, dynamically. Usually in OpenStack Object Storage the ETag is the MD5 sum of the contents of the object, and that holds true for each segment independently. But, it's not feasible to generate such an ETag for the manifest itself, so this method was chosen to at least offer change detection.

Large Object Storage History and Background

Large object support has gone through various iterations before settling on this implementation.

The primary factor driving the limitation of object size in OpenStack Object Storage is maintaining balance among the partitions of the ring. To maintain an even dispersion of disk usage throughout the cluster the obvious storage pattern was to simply split larger objects into smaller segments, which could then be glued together during a read.

Before the introduction of large object support some applications were already splitting their uploads into segments and re-assembling them on the client side after retrieving the individual pieces. This design allowed the client to support backup and archiving of large data sets, but was also frequently employed to improve performance or reduce errors due to network interruption. The major disadvantage of this method is that knowledge of the original partitioning scheme is required to properly reassemble the object, which is not practical for some use cases, such as CDN origination.

In order to eliminate any barrier to entry for clients wanting to store objects larger than 5GB, initially we also prototyped fully transparent support for large object uploads. A fully transparent implementation would support a larger max size by automatically splitting objects into segments during upload within the proxy without any changes to the client API. All segments were completely hidden from the client API.

This solution introduced a number of challenging failure conditions into the cluster, wouldn't provide the client with any option to do parallel uploads, and had no basis for a resume feature. The transparent implementation was deemed just too complex for the benefit.

The current “user manifest” design was chosen in order to provide a transparent download of large objects to the client and still provide the uploading client a clean API to support segmented uploads.

Alternative “explicit” user manifest options were discussed which would have required a pre-defined format for listing the segments to “finalize” the segmented upload. While this may offer some potential advantages, it was decided that pushing an added burden onto the client which could potentially limit adoption should be avoided in favor of a simpler “API” (essentially just the format of the ‘X-Object-Manifest' header).

During development it was noted that this “implicit” user manifest approach which is based on the path prefix can be potentially affected by the eventual consistency window of the container listings, which could theoretically cause a GET on the manifest object to return an invalid whole object for that short term. In reality you're unlikely to encounter this scenario unless you're running very high concurrency uploads against a small testing environment which isn't running the object-updaters or container-replicators.

Like all of OpenStack Object Storage, Large Object Support is living feature which will continue to improve and may change over time.

Throttling Resources by Setting Rate Limits

Rate limiting in OpenStack Object Storage is implemented as a pluggable middleware that you configure on the proxy server. Rate limiting is performed on requests that result in database writes to the account and container sqlite dbs. It uses memcached and is dependent on the proxy servers having highly synchronized time. The rate limits are limited by the accuracy of the proxy server clocks.

Configuration for Rate Limiting

All configuration is optional. If no account or container limits are provided there will be no rate limiting. Configuration available:

Table 4.23. Configuration options for rate limiting in proxy-server.conf file

Option	Default	Description
clock_accuracy	1000	Represents how accurate the proxy servers' system clocks are with each other. 1000 means that all the proxies' clock are accurate to each other within 1 millisecond. No ratelimit should be higher than the clock accuracy.
max_sleep_time_seconds	60	App will immediately return a 498 response if the necessary sleep time ever exceeds the given max_sleep_time_seconds.
log_sleep_time_seconds	0	To allow visibility into rate limiting set this value > 0 and all sleeps greater than the number will be logged.
log_name	swift	Label used when logging
log_facility	LOG_LOCAL0	Syslog log facility
log_level	INFO	Logging level
log_headers	False	If True, log headers in each request
account_ratelimit	0	If set, will limit all requests to /account_name and PUTs to /account_name/container_name. Number is in requests per second
account_whitelist	''	Comma separated lists of account names that will not be rate limited.
account_blacklist	''	Comma separated lists of account names that will not be allowed. Returns a 497 response.
container_ratelimit_size	''	When set with container_limit_x = r: for containers of size x, limit requests per second to r. Will limit GET and HEAD requests to /account_name/container_name and PUTs and DELETEs to /account_name/container_name/object_name
rate_buffer_seconds	5	Number of seconds the rate counter can drop and be allowed to catch up

		(faster than the listed rate). A larger number will result in larger average spikes but better average accuracy.

The container rate limits are linearly interpolated from the values given. A sample container rate limiting could be:

container_ratelimit_100 = 100

container_ratelimit_200 = 50

container_ratelimit_500 = 20

This would result in

Table 4.24. Values for Rate Limiting with Sample Configuration Settings

Container Size	Rate Limit
0-99	No limiting
100	100
150	75
500	20
1000	20

Additional Features

This section aims to detail a number of additional features in Swift and their configuration.

Health Check

Health Check provides a simple way to monitor if the swift proxy server is alive. If the proxy is access with the path /healthcheck, it will respond with "OK" in the body, which can be used by monitoring tools.

Table 4.25. Configuration options for filter:healthcheck in proxy-server.conf file

Option	Default	Description
log_name	swift	Label used when logging
log_facility	LOG_LOCAL0	Syslog log facility
log_level	INFO	Logging level

Domain Remap

Domain Remap is middleware that translates container and account parts of a domain to path parameters that the proxy server understands.

Table 4.26. Configuration options for filter:domain_remap in proxy-server.conf file

Option	Default	Description
log_name	swift	Label used when logging

log_facility	LOG_LOCAL0	Syslog log facility
log_level	INFO	Logging level
log_headers	False	If True, log headers in each request
path_root	v1	Root path
reseller_prefixes	AUTH	Reseller prefix
storage_domain	example.com	Domain to use for remap

CNAME Lookup

CNAME Lookup is middleware that translates an unknown domain in the host header to something that ends with the configured storage_domain by looking up the given domain's CNAME record in DNS.

Table 4.27. Configuration options for filter:cname_lookup in proxy-server.conf file

Option	Default	Description
log_name	swift	Label used when logging
log_facility	LOG_LOCAL0	Syslog log facility
log_level	INFO	Logging level
log_headers	False	If True, log headers in each request
lookup_depth	1	As CNAMEs can be recursive, how many levels to search through.
storage_domain	example.conf	

Temporary URL

Allows the creation of URLs to provide temporary access to objects. For example, a website may wish to provide a link to download a large object in Swift, but the Swift account has no public access. The website can generate a URL that will provide GET access for a limited time to the resource. When the web browser user clicks on the link, the browser will download the object directly from Swift, obviating the need for the website to act as a proxy for the request. If the user were to share the link with all his friends, or accidentally post it on a forum, etc. the direct access would be limited to the expiration time set when the website created the link. To create such temporary URLs, first an X-Account-Meta-Temp-URL-Key header must be set on the Swift account. Then, an HMAC-SHA1 (RFC 2104) signature is generated using the HTTP method to allow (GET or PUT), the Unix timestamp the access should be allowed until, the full path to the object, and the key set on the account. For example, here is code generating the signature for a GET for 60 seconds on /v1/AUTH_account/container/object:

```
import hmac
from hashlib import sha1
from time import time
method = 'GET'
expires = int(time() + 60)
path = '/v1/AUTH_account/container/object'
key = 'mykey'
hmac_body = '%s\n%s\n%s' % (method, expires, path)
sig = hmac.new(key, hmac_body, sha1).hexdigest()
```

Be certain to use the full path, from the /v1/ onward. Let's say the sig ends up equaling da39a3ee5e6b4b0d3255bfef95601890afd80709 and expires ends up 1323479485. Then, for example, the website could provide a link to:

```
https://swift-cluster.example.com/v1/AUTH_account/container/object?
temp_url_sig=da39a3ee5e6b4b0d3255bfef95601890afd80709&
temp_url_expires=1323479485
```

Any alteration of the resource path or query arguments would result in 401 Unauthorized. Similary, a PUT where GET was the allowed method would 401. HEAD is allowed if GET or PUT is allowed. Using this in combination with browser form post translation middleware could also allow direct-from-browser uploads to specific locations in Swift. Note that changing the X-Account-Meta-Temp-URL-Key will invalidate any previously generated temporary URLs within 60 seconds (the memcache time for the key).

Table 4.28. Configuration options for filter:tempurl in proxy-server.conf file

Option	Default	Description
incoming_allow_headers		
incoming_remove_headers	x-timestamp	
outgoing_allow_headers	x-object-meta-public-*	
outgoing_remove_headers	x-object-meta-*	

Name Check Filter

Name Check is a filter that disallows any paths that contain defined forbidden characters or that exceed a defined length.

Table 4.29. Configuration options for filter:name_check in proxy-server.conf file

Option	Default	Description
forbidden_chars		Characters that are not allowed in a name
max_length	255	Maximum length of a name
forbidden_regexp	"/\./|/\.\./|/\.$\|/\.\.$"	Substrings to forbid, using regular expression syntax

Constraints

The `swift-constraints` section in `swift.conf` allows modification of internal limits within swift. These are advanced features for tuning the performance of the cluster and should be altered with caution.

Table 4.30. Configuration options for swift-constraints in swift.conf

Option	Default	Description
max_file_size	5368709122	the largest "normal" object that can be saved in the cluster. This is also the limit on the size of each segment of a "large" object when using the large object manifest support. This value is set in bytes. Setting it to lower than

		1MiB will cause some tests to fail. It is STRONGLY recommended to leave this value at the default (5 * 2**30 + 2).
max_meta_name_length	128	the max number of bytes in the utf8 encoding of the name portion of a metadata header.
max_meta_value_lenth	256	the max number of bytes in the utf8 encoding of a metadata value
max_meta_count	90	the max number of metadata keys that can be stored on a single account, container, or object
max_meta_overall_size	4096	the max number of bytes in the utf8 encoding of the metadata (keys + values)
max_object_name_length	1024	the max number of bytes in the utf8 encoding of an object name
container_listing_limit	10000	the default (and max) number of items returned for a container listing request
max_account_name_length	256	the max number of bytes in the utf8 encoding of an account name
max_container_name_length	256	the max number of bytes in the utf8 encoding of a container name

Cluster Health

There is a `swift-dispersion-report` tool for measuring overall cluster health. This is accomplished by checking if a set of deliberately distributed containers and objects are currently in their proper places within the cluster. For instance, a common deployment has three replicas of each object. The health of that object can be measured by checking if each replica is in its proper place. If only 2 of the 3 is in place the object's heath can be said to be at 66.66%, where 100% would be perfect. A single object's health, especially an older object, usually reflects the health of that entire partition the object is in. If we make enough objects on a distinct percentage of the partitions in the cluster, we can get a pretty valid estimate of the overall cluster health. In practice, about 1% partition coverage seems to balance well between accuracy and the amount of time it takes to gather results. The first thing that needs to be done to provide this health value is create a new account solely for this usage. Next, we need to place the containers and objects throughout the system so that they are on distinct partitions. The swift-dispersion-populate tool does this by making up random container and object names until they fall on distinct partitions. Last, and repeatedly for the life of the cluster, we need to run the swift-dispersion-report tool to check the health of each of these containers and objects. These tools need direct access to the entire cluster and to the ring files (installing them on a proxy server will probably do). Both swift-dispersion-populate and swift-dispersion-report use the same configuration file, `/etc/swift/dispersion.conf`. Example dispersion.conf file:

```
[dispersion]
auth_url = http://localhost:8080/auth/v1.0
auth_user = test:tester
auth_key = testing
```

There are also options for the conf file for specifying the dispersion coverage (defaults to 1%), retries, concurrency, etc. though usually the defaults are fine. Once the configuration

is in place, run swift-dispersion-populate to populate the containers and objects throughout the cluster. Now that those containers and objects are in place, you can run swift-dispersion-report to get a dispersion report, or the overall health of the cluster. Here is an example of a cluster in perfect health:

```
$ swift-dispersion-report
Queried 2621 containers for dispersion reporting, 19s, 0 retries
100.00% of container copies found (7863 of 7863)
Sample represents 1.00% of the container partition space

Queried 2619 objects for dispersion reporting, 7s, 0 retries
100.00% of object copies found (7857 of 7857)
Sample represents 1.00% of the object partition space
```

Now, deliberately double the weight of a device in the object ring (with replication turned off) and rerun the dispersion report to show what impact that has:

```
$ swift-ring-builder object.builder set_weight d0 200
$ swift-ring-builder object.builder rebalance
...
$ swift-dispersion-report
Queried 2621 containers for dispersion reporting, 8s, 0 retries
100.00% of container copies found (7863 of 7863)
Sample represents 1.00% of the container partition space

Queried 2619 objects for dispersion reporting, 7s, 0 retries
There were 1763 partitions missing one copy.
77.56% of object copies found (6094 of 7857)
Sample represents 1.00% of the object partition space
```

You can see the health of the objects in the cluster has gone down significantly. Of course, this test environment has just four devices, in a production environment with many many devices the impact of one device change is much less. Next, run the replicators to get everything put back into place and then rerun the dispersion report:

```
... start object replicators and monitor logs until they're caught up ...
$ swift-dispersion-report
Queried 2621 containers for dispersion reporting, 17s, 0 retries
100.00% of container copies found (7863 of 7863)
Sample represents 1.00% of the container partition space

Queried 2619 objects for dispersion reporting, 7s, 0 retries
100.00% of object copies found (7857 of 7857)
Sample represents 1.00% of the object partition space
```

Alternatively, the dispersion report can also be output in json format. This allows it to be more easily consumed by third party utilities:

```
$ swift-dispersion-report -j
{"object": {"retries:": 0, "missing_two": 0, "copies_found": 7863,
 "missing_one": 0,
"copies_expected": 7863, "pct_found": 100.0, "overlapping": 0, "missing_all":
 0}, "container":
{"retries:": 0, "missing_two": 0, "copies_found": 12534, "missing_one": 0,
 "copies_expected":
12534, "pct_found": 100.0, "overlapping": 15, "missing_all": 0}}
```

Table 4.31. Configuration options for dispersion in proxy-server.conf file

Option	Default	Description
auth_version	1.0	Swift authentication API version
dispersion_coverage	1	
dump_json	'no'	
retries	3	
auth_url		
auth_user		
auth_key		
swift_dir	/etc/swift	

Static Large Object (SLO) support

This feature is very similar to Dynamic Large Object (DLO) support in that it allows the user to upload many objects concurrently and afterwards download them as a single object. It is different in that it does not rely on eventually consistent container listings to do so. Instead, a user defined manifest of the object segments is used.

Uploading the Manifest

After the user has uploaded the objects to be concatenated a manifest is uploaded. The request must be a PUT with the query parameter::

```
?multipart-manifest=put
```

The body of this request will be an ordered list of files in json data format. The data to be supplied for each segment is:

- path: the path to the segment (not including account) /container/object_name
- etag: the etag given back when the segment was PUT
- size_bytes: the size of the segment in bytes

The format of the list will be:

```
json:
[{"path": "/cont/object",
  "etag": "etagoftheobjectsegment",
  "size_bytes": 1048576}, ...]
```

The number of object segments is limited to a configurable amount, default 1000. Each segment, except for the final one, must be at least 1 megabyte (configurable). On upload, the middleware will head every segment passed in and verify the size and etag of each. If any of the objects do not match (not found, size/etag mismatch, below minimum size) then the user will receive a 4xx error response. If everything does match, the user will receive a 2xx response and the SLO object is ready for downloading.

Behind the scenes, on success, a json manifest generated from the user input is sent to object servers with an extra "X-Static-Large-Object: True" header and a modified Content-Type. The parameter: swift_bytes=$total_size will be appended to the existing Content-

Type, where total_size is the sum of all the included segments' size_bytes. This extra parameter will be hidden from the user.

Manifest files can reference objects in separate containers, which will improve concurrent upload speed. Objects can be referenced by multiple manifests.

Retrieving a Large Object

A GET request to the manifest object will return the concatenation of the objects from the manifest much like DLO. If any of the segments from the manifest are not found or their Etag/Content Length no longer match the connection will drop. In this case a 409 Conflict will be logged in the proxy logs and the user will receive incomplete results.

The headers from this GET or HEAD request will return the metadata attached to the manifest object itself with some exceptions:

- Content-Length: the total size of the SLO (the sum of the sizes of the segments in the manifest)
- X-Static-Large-Object: True
- Etag: the etag of the SLO (generated the same way as DLO)

A GET request with the query parameter::

```
?multipart-manifest=get
```

Will return the actual manifest file itself. This is generated json and does not match the data sent from the original multipart-manifest=put. This call's main purpose is for debugging.

When the manifest object is uploaded you are more or less guaranteed that every segment in the manifest exists and matched the specifications. However, there is nothing that prevents the user from breaking the SLO download by deleting/replacing a segment referenced in the manifest. It is left to the user use caution in handling the segments.

Deleting a Large Object

A DELETE request will just delete the manifest object itself. A DELETE with a query parameter:

```
?multipart-manifest=delete
```

will delete all the segments referenced in the manifest and then, if successful, the manifest itself. The failure response will be similar to the bulk delete middleware.

Modifying a Large Object

PUTs / POSTs will work as expected, PUTs will just overwrite the manifest object for example.

Container Listings

In a container listing the size listed for SLO manifest objects will be the total_size of the concatenated segments in the manifest. The overall X-Container-Bytes-Used for the

container (and subsequently for the account) will not reflect total_size of the manifest but the actual size of the json data stored. The reason for this somewhat confusing discrepancy is we want the container listing to reflect the size of the manifest object when it is downloaded. We do not, however, want to count the bytes-used twice (for both the manifest and the segments it's referring to) in the container and account metadata which can be used for stats purposes.

Configuring Object Storage with the S3 API

The Swift3 middleware emulates the S3 REST API on top of Object Storage.

The following operations are currently supported:

- GET Service
- DELETE Bucket
- GET Bucket (List Objects)
- PUT Bucket
- DELETE Object
- GET Object
- HEAD Object
- PUT Object
- PUT Object (Copy)

To use this middleware, first download the latest version from its repository to your proxy server(s).

```
$ git clone https://github.com/fujita/swift3.git
```

Optional: To use this middleware with Swift 1.7.0 and previous versions, you'll need to use the v1.7 tag of the fujita/swift3 repository. Clone the repo as above and then:

```
$ cd swift3; git checkout v1.7
```

Then, install it using standard python mechanisms, such as:

```
$ sudo python setup.py install
```

Alternatively, if you have configured the Ubuntu Cloud Archive, you may use:

```
$ sudo apt-get install swift-python-s3
```

To add this middleware to your configuration, add the swift3 middleware in front of the auth middleware, and before any other middleware that look at swift requests (like rate limiting).

Ensure that your proxy-server.conf file contains swift3 in the pipeline and the [filter:swift3] section, as shown below:

```
[pipeline:main]
pipeline = healthcheck cache swift3 swauth proxy-server

[filter:swift3]
use = egg:swift#swift3
```

Next, configure the tool that you use to connect to the S3 API. For S3curl, for example, you'll need to add your host IP information by adding y our host IP to the @endpoints array (line 33 in s3curl.pl):

```
my @endpoints = ( '1.2.3.4');
```

Now you can send commands to the endpoint, such as:

```
$ ./s3curl.pl - 'myacc:myuser' -key mypw -get - -s -v http://1.2.3.4:8080
```

To set up your client, the access key will be the concatenation of the account and user strings that should look like test:tester, and the secret access key is the account password. The host should also point to the Swift storage node's hostname. It also will have to use the old-style calling format, and not the hostname-based container format. Here is an example client setup using the Python boto library on a locally installed all-in-one Swift installation.

```
connection = boto.s3.Connection(
    aws_access_key_id='test:tester',
    aws_secret_access_key='testing',
    port=8080,
    host='127.0.0.1',
    is_secure=False,
    calling_format=boto.s3.connection.OrdinaryCallingFormat())
```

Managing OpenStack Object Storage with CLI Swift

In the Object Store (swift) project there is a tool that can perform a variety of tasks on your storage cluster named swift. This client utility can be used for adhoc processing, to gather statistics, list items, update metadata, upload, download and delete files. It is based on the native swift client library client.py. Incorporating client.py into swift provides many benefits such as seamlessly re-authorizing if the current token expires in the middle of processing, retrying operations up to five times and a processing concurrency of 10. All of these things help make the swift tool robust and great for operational use.

Swift ACLs

Swift ACLs work with users and accounts. Users have roles on accounts - such as '.admin', which allows full access to all containers and objects under the account. ACLs are set at the container level and support lists for read and write access, which are set with the X-Container-Read and X-Container-Write header respectively.

The swift client can be used to set the acls, using the post subcommand with the option '-r' for the read ACL, and '-w' for the write ACL. This example allows the user 'testuser' to read objects in the container:

```
$ swift post -r 'testuser'
```

This could instead be a list of users.

If you are using the StaticWeb middleware to allow OpenStack Object Storage to serve public web content, you should also be aware of the ACL syntax for managing allowed referrers. The syntax is '.r:' followed by a list of allowed referrers. For example, this command allows all referring domains access to the object:

```
$ swift post -r '.r:*'
```

Swift CLI Basics

The command line usage for swift, the CLI tool is:

```
swift (command) [options] [args]
```

Here are the available commands for swift.

stat [container] [object]

Displays information for the account, container, or object depending on the args given (if any).

list [options] [container]

Lists the containers for the account or the objects for a container. -p or -prefix is an option that will only list items beginning with that prefix. -d or -delimiter is option (for container listings only) that will roll up items with the given delimiter, or character that can act as a nested directory organizer.

upload [options] container file_or_directory [file_or_directory] [...]

Uploads to the given container the files and directories specified by the remaining args. -c or -changed is an option that will only upload files that have changed since the last upload.

post [options] [container] [object]

Updates meta information for the account, container, or object depending on the args given. If the container is not found, it will be created automatically; but this is not true for accounts and objects. Containers also allow the -r (or -read-acl) and -w (or -write-acl) options. The -m or -meta option is allowed on all and used to define the user meta data items to set in the form Name:Value. This option can be repeated.

Example: post -m Color:Blue -m Size:Large

download —all OR download container [object] [object] ...

Downloads everything in the account (with —all), or everything in a container, or a list of objects depending on the args given. For a single object download, you may use the -o [—output] (filename) option to redirect the output to a specific file or if "-" then just redirect to stdout.

delete —all OR delete container [object] [object] ...

Deletes everything in the account (with —all), or everything in a container, or a list of objects depending on the args given.

Example: swift -A https://auth.api.rackspacecloud.com/v1.0 -U user -K key stat

Options for swift

-version show program's version number and exit

-h, -help show this help message and exit

-s, -snet Use SERVICENET internal network

-v, -verbose Print more info

-q, -quiet Suppress status output

-A AUTH, -auth=AUTH URL for obtaining an auth token

-U USER, -user=USER User name for obtaining an auth token

-K KEY, -key=KEY Key for obtaining an auth token

Analyzing Log Files with Swift CLI

When you want quick, command-line answers to questions about logs, you can use swift with the -o or -output option. The -o —output option can only be used with a single object download to redirect the data stream to either a different file name or to STDOUT (-). The ability to redirect the output to STDOUT allows you to pipe "|" data without saving it to disk first. One common use case is being able to do some quick log file analysis. First let's use swift to setup some data for the examples. The "logtest" directory contains four log files with the following line format.

```
Files:
 2010-11-16-21_access.log
 2010-11-16-22_access.log
 2010-11-15-21_access.log
 2010-11-15-22_access.log

 Log lines:
 Nov 15 21:53:52 lucid64 proxy-server - 127.0.0.1 15/Nov/2010/22/53/52 DELETE /v1/AUTH_cd4f57824deb4
- test%3Atester%2CAUTH_tkcdab3c6296e249d7b7e2454ee57266ff - - - txaba5984c-aac7-460e-b04b-afc43f0
```

The swift tool can easily upload the four log files into a container named “logtest”:

```
$ cd logs
$ swift -A http://swift-auth.com:11000/v1.0 -U test:tester -K testing upload
 logtest *.log
```

```
2010-11-16-21_access.log
2010-11-16-22_access.log
2010-11-15-21_access.log
2010-11-15-22_access.log
```

Get statistics on the account:

```
$ swift -A http://swift-auth.com:11000/v1.0 -U test:tester -K testing -q stat
```

```
Account: AUTH_cd4f57824deb4248a533f2c28bf156d3
Containers: 1
Objects: 4
Bytes: 5888268
```

Get statistics on the container:

```
$ swift -A http://swift-auth.com:11000/v1.0 -U test:tester -K testing stat
 logtest
```

```
Account: AUTH_cd4f57824deb4248a533f2c28bf156d3
Container: logtest
Objects: 4
Bytes: 5864468
Read ACL:
Write ACL:
```

List all the objects in the container:

```
$ swift -A http:///swift-auth.com:11000/v1.0 -U test:tester -K testing list
 logtest
```

```
2010-11-15-21_access.log
2010-11-15-22_access.log
2010-11-16-21_access.log
2010-11-16-22_access.log
```

These next three examples use the -o —output option with (-) to help answer questions about the uploaded log files. The swift command will download an object, stream it to awk to determine the breakdown of requests by return code for everything during 2200 on November 16th, 2010. Based on the log line format column 9 is the type of request and column 12 is the return code. After awk processes the data stream it is piped to sort and then uniq -c to sum up the number of occurrences for each combination of request type and return code.

```
$ swift -A http://swift-auth.com:11000/v1.0 -U test:tester -K testing download
 -o - logtest 2010-11-16-22_access.log | awk ‘{ print $9”-“$12}’ | sort | uniq
 -c
```

```
805 DELETE-204
12 DELETE-404
2 DELETE-409
723 GET-200
142 GET-204
74 GET-206
80 GET-304
34 GET-401
5 GET-403
18 GET-404
166 GET-412
2 GET-416
50 HEAD-200
17 HEAD-204
20 HEAD-401
8 HEAD-404
30 POST-202
25 POST-204
22 POST-400
6 POST-404
842 PUT-201
2 PUT-202
32 PUT-400
4 PUT-403
4 PUT-404
2 PUT-411
6 PUT-412
6 PUT-413
2 PUT-422
8 PUT-499
```

This example uses a bash for loop with awk, swift with its -o —output option with a hyphen (-) to find out how many PUT requests are in each log file. First create a list of objects by running swift with the list command on the “logtest” container; then for each item in the list run swift with download -o - then pipe the output into grep to filter the put requests and finally into wc -l to count the lines.

```
$ for f in `swift -A http://swift-auth.com:11000/v1.0 -U test:tester -K
 testing list logtest` ; \
        do  echo -ne “PUTS - ” ; swift -A http://swift-auth.com:11000/v1.0 -U
 test:tester -K testing download -o -  logtest $f | grep PUT | wc -l ; \
    done
```

```
2010-11-15-21_access.log - PUTS - 402
2010-11-15-22_access.log - PUTS - 1091
2010-11-16-21_access.log - PUTS - 892
2010-11-16-22_access.log - PUTS - 910
```

By adding the -p —prefix option a prefix query is performed on the list to return only the object names that begin with a specific string. Let’s determine out how many PUT requests are in each object with a name beginning with “2010-11-15”. First create a list of objects by running swift with the list command on the “logtest” container with the prefix option -p 2010-11-15. Then on each of item(s) returned run swift with the download -o - then pipe

the output to grep and wc as in the previous example. The echo command is added to display the object name.

```
$ for f in `swift -A http://swift-auth.com:11000/v1.0 -U test:tester -K
 testing list -p 2010-11-15 logtest` ; \
        do  echo -ne “$f - PUTS - ” ; swift -A http://127.0.0.1:11000/v1.0 -U
 test:tester -K testing download -o - logtest $f | grep PUT | wc -l ; \
      done
```

```
2010-11-15-21_access.log - PUTS - 402
2010-11-15-22_access.log - PUTS - 910
```

The swift utility is simple, scalable, flexible and provides useful solutions all of which are core principles of cloud computing; with the -o output option being just one of its many features.

5. OpenStack Object Storage Tutorials

We want people to use OpenStack for practical problem solving, and the increasing size and density of web content makes for a great use-case for object storage. These tutorials show you how to use your OpenStack Object Storage installation for practical purposes, and it assumes Object Storage is already installed.

Storing Large Photos or Videos on the Cloud

In this OpenStack tutorial, we'll walk through using an Object Storage installation to back up all your photos or videos. As the sensors on consumer-grade and pro-sumer grade cameras generate more and more megapixels, we all need a place to back our files to and know they are safe.

We'll go through this tutorial in parts:

- Setting up secure access to Object Storage.
- Configuring Cyberduck for connecting to OpenStack Object Storage.
- Copying files to the cloud.

Part I: Setting Up Secure Access

In this part, we'll get the proxy server running with SSL on the Object Storage installation. It's a requirement for using Cyberduck as a client interface to Object Storage.

You will need a key and certificate to do this, which we can create as a self-signed for the tutorial since we can do the extra steps to have Cyberduck accept it. Creating a self-signed cert can usually be done with these commands on the proxy server:

```
$ cd /etc/swift
$ openssl req -new -x509 -nodes -out cert.crt -keyout cert.key
```

Ensure these generated files are in /etc/swift/cert.crt and /etc/swift/cert.key.

You also should configure your iptables to enable https traffic. Here's an example setup that works.

```
Chain INPUT (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source
 destination
76774 1543M ACCEPT     all  --  lo     any     localhost           anywhere

 416K  537M ACCEPT     all  --  any    any     anywhere            anywhere
          state RELATED,ESTABLISHED
  106  6682 ACCEPT     tcp  --  any    any     anywhere            anywhere
          tcp dpt:https
   13   760 ACCEPT     tcp  --  any    any     anywhere            anywhere
          tcp dpt:ssh
    3   124 ACCEPT     icmp --  any    any     anywhere            anywhere
          icmp echo-request
  782 38880 DROP       all  --  any    any     anywhere            anywhere
```

```
Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source
 destination
    0     0 DROP       all  --  any    any     anywhere             anywhere


Chain OUTPUT (policy ACCEPT 397K packets, 1561M bytes)
 pkts bytes target     prot opt in     out     source
 destination
```

If you don't have access to the Object Storage installation to configure these settings, ask your service provider to set up secure access for you.

Then, edit your `proxy-server.conf` file to include the following in the [DEFAULT] sections.

```
[DEFAULT]
bind_port = 443
cert_file = /etc/swift/cert.crt
key_file = /etc/swift/cert.key
```

Also, make sure you use https: for all references to the URL for the server in the .conf files as needed.

Verify that you can connect using the Public URL to Object Storage by using the "swift" tool:

```
$ swift -A https://yourswiftinstall.com:11000/v1.0 -U test:tester -K testing
 stat
```

Okay, you've created the access that Cyberduck expects for your Object Storage installation. Let's start configuring the Cyberduck side of things.

Part II: Configuring Cyberduck

Note

See the Cyberduck website for further details.

After installing Cyberduck you'll need to change the path/context used for the authentication URL. The default value shipped with Cyberduck is incorrect.

On OS X open a Terminal window and execute,

```
$ defaults write ch.sudo.cyberduck cf.authentication.context /auth/v1.0
```

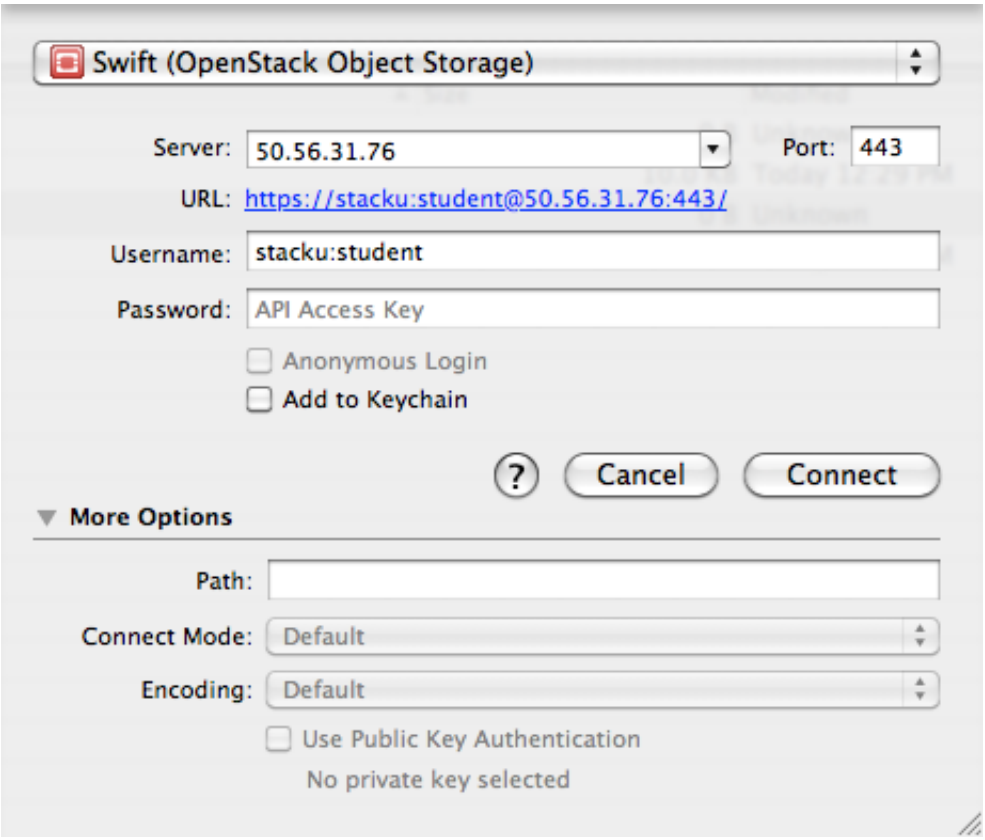
On Windows open the preferences file in a text editor. The exact location of this file is described here. If this path doesn't exist you may need to start and stop Cyberduck to have it generate the config file. Once you've opened the file add the setting,

```
<setting name="cf.authentication.context" value="/auth/v1.0" />
```

To connect to Swift start Cyberduck and click the *Open Connection* toolbar button or choose *File* > *Open Connection*. Select *Swift (OpenStack Object Storage)* in the dropdown and enter your cluster details.

Server	your proxy server's hostname or IP address
Port	443
Username	account name followed by a colon and then the user name, for example test:tester
Password	password for the account and user name entered above

Figure 5.1. Example Cyberduck Swift Connection



Connecting to a Unsecured Swift Cluster

An Unsecured Swift Cluster does not use https connections. Download the Unsecured Swift profile file and double click it to import it into Cyberduck.

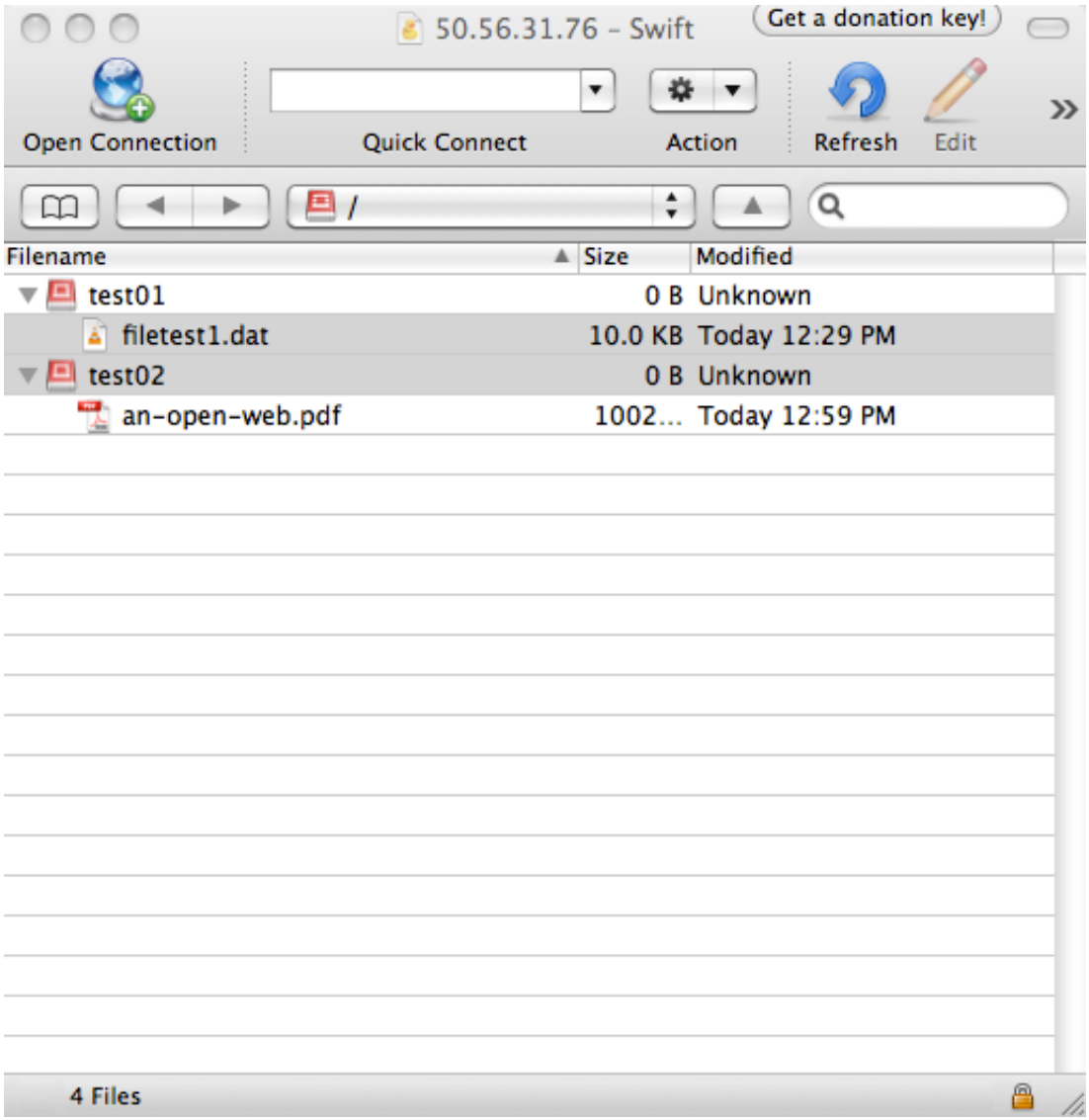
When creating a new connection select *Swift (HTTP)*. Enter your connection details as described above. You'll need to change the port since presumably you won't want to use 443.

Part III: Creating Containers (Folders) and Uploading Files

Now you want to create containers to hold your files. Without containers, Object Storage doesn't know where to put the files. In the Action menu, choose New Folder and name the folder.

Next you can drag and drop files into the created folder or select File > Upload to select files to upload to the OpenStack Object Storage service.

Figure 5.2. Example Cyberduck Swift Showing Uploads



Et voila! You can back up terabytes of data if you just have the space and the data. That's a lot of pictures or video, so get snapping and rolling!

6. OpenStack Object Storage Monitoring

Excerpted from a blog post by Darrell Bishop

An OpenStack Object Storage cluster is a complicated beast—a collection of many daemons across many nodes, all working together. With so many "moving parts" it's important to be able to tell what's going on inside the cluster. Tracking server-level metrics like CPU utilization, load, memory consumption, disk usage and utilization, etc. is necessary, but not sufficient. We need to know what the different daemons are doing on each server. What's the volume of object replication on node8? How long is it taking? Are there errors? If so, when did they happen?

In such a complex ecosystem, it's no surprise that there are multiple approaches to getting the answers to these kinds of questions. Let's examine some of the existing approaches to OpenStack Object Storage monitoring.

Swift Recon

The Swift Recon middleware can provide general machine stats (load average, socket stats, `/proc/meminfo` contents, etc.) as well as Swift-specific metrics:

- The MD5 sum of each ring file.
- The most recent object replication time.
- Count of each type of quarantined file: account, container, or object.
- Count of "async_pendings" (deferred container updates) on disk.

Swift Recon is middleware installed in the object server's pipeline and takes one required option: a local cache directory. Tracking of async_pendings requires an additional cron job per object server. Data is then accessed by sending HTTP requests to the object server directly, or by using the swift-recon command-line tool.

There are some good Object Storage cluster stats in there, but the general server metrics overlap with existing server monitoring systems and to get the Swift-specific metrics into a monitoring system, they must be polled. Swift Recon is essentially acting as a middle-man metrics collector. The process actually feeding metrics to your stats system, like collectd, gmond, etc., is probably already running on the storage node. So it could either talk to Swift Recon or just collect the metrics itself.

There's an upcoming update to Swift Recon which broadens support to the account and container servers. The auditors, replicators, and updaters can also report statistics, but only for the most recent run.

Swift-Informant

Florian Hines developed the Swift-Informant middleware to get real-time visibility into Object Storage client requests. It sits in the proxy server's pipeline and after each request to the proxy server, sends three metrics to a StatsD server:

- A counter increment for a metric like `obj.GET.200` or `cont.PUT.404`.
- Timing data for a metric like `acct.GET.200` or `obj.GET.200`. [The README says the metrics will look like `duration.acct.GET.200`, but I don't see the "duration" in the code. I'm not sure what Etsy's server does, but our StatsD server turns timing metrics into 5 derivative metrics with new segments appended, so it probably works as coded. The first metric above would turn into `acct.GET.200.lower`, `acct.GET.200.upper`, `acct.GET.200.mean`, `acct.GET.200.upper_90`, and `acct.GET.200.count`]
- A counter increase by the bytes transferred for a metric like `tfer.obj.PUT.201`.

This is good for getting a feel for the quality of service clients are experiencing with the timing metrics, as well as getting a feel for the volume of the various permutations of request server type, command, and response code. Swift-Informant also requires no change to core Object Storage code since it is implemented as middleware. However, because of this, it gives you no insight into the workings of the cluster past the proxy server. If one storage node's responsiveness degrades for some reason, you'll only see that some of your requests are bad—either as high latency or error status codes. You won't know exactly why or where that request tried to go. Maybe the container server in question was on a good node, but the object server was on a different, poorly-performing node.

Statsdlog

Florian's Statsdlog project increments StatsD counters based on logged events. Like Swift-Informant, it is also non-intrusive, but statsdlog can track events from all Object Storage daemons, not just proxy-server. The daemon listens to a UDP stream of syslog messages and StatsD counters are incremented when a log line matches a regular expression. Metric names are mapped to regex match patterns in a JSON file, allowing flexible configuration of what metrics are extracted from the log stream.

Currently, only the first matching regex triggers a StatsD counter increment, and the counter is always incremented by 1. There's no way to increment a counter by more than one or send timing data to StatsD based on the log line content. The tool could be extended to handle more metrics per line and data extraction, including timing data. But even then, there would still be a coupling between the log textual format and the log parsing regexes, which would themselves be more complex in order to support multiple matches per line and data extraction. Also, log processing introduces a delay between the triggering event and sending the data to StatsD. We would prefer to increment error counters where they occur, send timing data as soon as it is known, avoid coupling between a log string and a parsing regex, and not introduce a time delay between events and sending data to StatsD. And that brings us to the next method of gathering Object Storage operational metrics.

Swift StatsD Logging

StatsD was designed for application code to be deeply instrumented; metrics are sent in real-time by the code which just noticed something or did something. The overhead of sending a metric is extremely low: a `sendto` of one UDP packet. If that overhead is still too high, the StatsD client library can send only a random portion of samples and StatsD will approximate the actual number when flushing metrics upstream.

To avoid the problems inherent with middleware-based monitoring and after-the-fact log processing, the sending of StatsD metrics is integrated into Object Storage itself. The submitted change set currently reports 124 metrics across 15 Object Storage daemons and the tempauth middleware. Details of the metrics tracked are in the Swift Administration Guide.

The sending of metrics is integrated with the logging framework. To enable, configure `log_statsd_host` in the relevant config file. You can also specify the port and a default sample rate. The specified default sample rate is used unless a specific call to a statsd logging method (see the list below) overrides it. Currently, no logging calls override the sample rate, but it's conceivable that some metrics may require accuracy (sample_rate == 1) while others may not.

```
[DEFAULT]
     ...
log_statsd_host = 127.0.0.1
log_statsd_port = 8125
log_statsd_default_sample_rate = 1
```

Then the LogAdapter object returned by `get_logger()`, usually stored in `self.logger`, has the following new methods:

- `set_statsd_prefix(self, prefix)` Sets the client library's stat prefix value which gets prepended to every metric. The default prefix is the "name" of the logger (eg. "object-server", "container-auditor", etc.). This is currently used to turn "proxy-server" into one of "proxy-server.Account", "proxy-server.Container", or "proxy-server.Object" as soon as the Controller object is determined and instantiated for the request.

- `update_stats(self, metric, amount, sample_rate=1)` Increments the supplied metric by the given amount. This is used when you need to add or subtract more that one from a counter, like incrementing "suffix.hashes" by the number of computed hashes in the object replicator.

- `increment(self, metric, sample_rate=1)` Increments the given counter metric by one.

- `decrement(self, metric, sample_rate=1)` Lowers the given counter metric by one.

- `timing(self, metric, timing_ms, sample_rate=1)` Record that the given metric took the supplied number of milliseconds.

- `timing_since(self, metric, orig_time, sample_rate=1)` Convenience method to record a timing metric whose value is "now" minus an existing timestamp.

Note that these logging methods may safely be called anywhere you have a logger object. If StatsD logging has not been configured, the methods are no-ops. This avoids messy conditional logic each place a metric is recorded. Here's two example usages of the new logging methods:

```
# swift/obj/replicator.py
def update(self, job):
    # ...
    begin = time.time()
    try:
        hashed, local_hash = tpool.execute(tpooled_get_hashes, job['path'],
                do_listdir=(self.replication_count % 10) == 0,
                reclaim_age=self.reclaim_age)
        # See tpooled_get_hashes "Hack".
        if isinstance(hashed, BaseException):
            raise hashed
        self.suffix_hash += hashed
        self.logger.update_stats('suffix.hashes', hashed)
        # ...
    finally:
        self.partition_times.append(time.time() - begin)
        self.logger.timing_since('partition.update.timing', begin)
```

```
# swift/container/updater.py
def process_container(self, dbfile):
    # ...
    start_time = time.time()
    # ...
        for event in events:
            if 200 <= event.wait() < 300:
                successes += 1
            else:
                failures += 1
        if successes > failures:
            self.logger.increment('successes')
            # ...
        else:
            self.logger.increment('failures')
            # ...
        # Only track timing data for attempted updates:
        self.logger.timing_since('timing', start_time)
    else:
        self.logger.increment('no_changes')
        self.no_changes += 1
```

The development team of StatsD wanted to use the pystatsd client library (not to be confused with a similar-looking project also hosted on GitHub), but the released version on PyPi was missing two desired features the latest version in GitHub had: the ability to configure a metrics prefix in the client object and a convenience method for sending timing data between "now" and a "start" timestamp you already have. So they just implemented a simple StatsD client library from scratch with the same interface. This has the nice fringe benefit of not introducing another external library dependency into Object Storage.

7. Support

Online resources aid in supporting OpenStack and the community members are willing and able to answer questions and help with bug suspicions. We are constantly improving and adding to the main features of OpenStack, but if you have any problems, do not hesitate to ask. Here are some ideas for supporting OpenStack and troubleshooting your existing installations.

Community Support

Here are some places you can locate others who want to help.

ask.openstack.org

During setup or testing, you may have questions about how to do something, or end up in a situation where you can't seem to get a feature to work correctly. The ask.openstack.org site is available for questions and answers. When visiting the Ask site at http://ask.openstack.org, it is usually good to at least scan over recently asked questions to see if your question has already been answered. If that is not the case, then proceed to adding a new question. Be sure you give a clear, concise summary in the title and provide as much detail as possible in the description. Paste in your command output or stack traces, link to screenshots, and so on.

OpenStack mailing lists

Posting your question or scenario to the OpenStack mailing list is a great way to get answers and insights. You can learn from and help others who may have the same scenario as you. Go to https://launchpad.net/~openstack and click "Subscribe to mailing list" or view the archives at https://lists.launchpad.net/openstack/. You may be interested in the other mailing lists for specific projects or development - these can be found on the wiki. A description of all the additional mailing lists is available at http://wiki.openstack.org/MailingLists.

The OpenStack Wiki search

The OpenStack wiki contains content on a broad range of topics, but some of it sits a bit below the surface. Fortunately, the wiki search feature is very powerful in that it can do both searches by title and by content. If you are searching for specific information, say about "networking" or "api" for nova, you can find lots of content using the search feature. More is being added all the time, so be sure to check back often. You can find the search box in the upper right hand corner of any OpenStack wiki page.

The Launchpad Bugs area

So you think you've found a bug. That's great! Seriously, it is. The OpenStack community values your setup and testing efforts and wants your feedback. To log a bug you must have

a Launchpad account, so sign up at https://launchpad.net/+login if you do not already have a Launchpad ID. You can view existing bugs and report your bug in the Launchpad Bugs area. It is suggested that you first use the search facility to see if the bug you found has already been reported (or even better, already fixed). If it still seems like your bug is new or unreported then it is time to fill out a bug report.

Some tips:

- Give a clear, concise summary!
- Provide as much detail as possible in the description. Paste in your command output or stack traces, link to screenshots, etc.
- Be sure to include what version of the software you are using. This is especially critical if you are using a development branch eg. "Grizzly release" vs git commit bc79c3ecc55929bac585d04a03475b72e06a3208.
- Any deployment specific info is helpful as well, such as Ubuntu 12.04, multi-node install.

The Launchpad Bugs areas are available here - :

- OpenStack Compute: https://bugs.launchpad.net/nova
- OpenStack Object Storage: https://bugs.launchpad.net/swift
- OpenStack Image Delivery and Registration: https://bugs.launchpad.net/glance
- OpenStack Identity: https://bugs.launchpad.net/keystone
- OpenStack Dashboard: https://bugs.launchpad.net/horizon
- OpenStack Network Connectivity: https://bugs.launchpad.net/quantum

The OpenStack IRC channel

The OpenStack community lives and breathes in the #openstack IRC channel on the Freenode network. You can come by to hang out, ask questions, or get immediate feedback for urgent and pressing issues. To get into the IRC channel you need to install an IRC client or use a browser-based client by going to http://webchat.freenode.net/. You can also use Colloquy (Mac OS X, http://colloquy.info/) or mIRC (Windows, http://www.mirc.com/) or XChat (Linux). When you are in the IRC channel and want to share code or command output, the generally accepted method is to use a Paste Bin, the OpenStack project has one at http://paste.openstack.org. Just paste your longer amounts of text or logs in the web form and you get a URL you can then paste into the channel. The OpenStack IRC channel is: #openstack on irc.freenode.net. A list of all the OpenStack-related IRC channels is at https://wiki.openstack.org/wiki/IRC.

8. Troubleshooting OpenStack Object Storage

For OpenStack Object Storage, everything is logged in /var/log/syslog (or messages on some distros). Several settings enable further customization of logging, such as log_name, log_facility, and log_level, within the object server configuration files.

Handling Drive Failure

In the event that a drive has failed, the first step is to make sure the drive is unmounted. This will make it easier for OpenStack Object Storage to work around the failure until it has been resolved. If the drive is going to be replaced immediately, then it is just best to replace the drive, format it, remount it, and let replication fill it up.

If the drive can't be replaced immediately, then it is best to leave it unmounted, and remove the drive from the ring. This will allow all the replicas that were on that drive to be replicated elsewhere until the drive is replaced. Once the drive is replaced, it can be re-added to the ring.

Rackspace has seen hints at drive failures by looking at error messages in /var/log/kern.log - do consider checking this in your monitoring

Handling Server Failure

If a server is having hardware issues, it is a good idea to make sure the OpenStack Object Storage services are not running. This will allow OpenStack Object Storage to work around the failure while you troubleshoot.

If the server just needs a reboot, or a small amount of work that should only last a couple of hours, then it is probably best to let OpenStack Object Storage work around the failure and get the machine fixed and back online. When the machine comes back online, replication will make sure that anything that is missing during the downtime will get updated.

If the server has more serious issues, then it is probably best to remove all of the server's devices from the ring. Once the server has been repaired and is back online, the server's devices can be added back into the ring. It is important that the devices are reformatted before putting them back into the ring as it is likely to be responsible for a different set of partitions than before.

Detecting Failed Drives

It has been our experience that when a drive is about to fail, error messages will spew into /var/log/kern.log. There is a script called swift-drive-audit that can be run via cron

to watch for bad drives. If errors are detected, it will unmount the bad drive, so that OpenStack Object Storage can work around it. The script takes a configuration file with the following settings:

```
          [drive-audit]
          Option  Default  Description
          log_facility  LOG_LOCAL0  Syslog log facility
          log_level  INFO  Log level
          device_dir  /srv/node  Directory devices are mounted under
          minutes  60  Number of minutes to look back in /var/log/kern.log
          error_limit  1  Number of errors to find before a device is
unmounted
```

This script has only been tested on Ubuntu 10.04, so if you are using a different distro or OS, some care should be taken before using in production.

Emergency Recovery of Ring Builder Files

You should always keep a backup of Swift ring builder files. However, if an emergency occurs, this procedure may assist in returning your cluster to an operational state.

Using existing Swift tools, there is no way to recover a builder file from a ring.gz file. However, if you have a knowledge of Python, it is possible to construct a builder file that is pretty close to the one you have lost. The following is what you will need to do.

Warning

This procedure is a last-resort for emergency circumstances - it requires knowledge of the swift python code and may not succeed.

First, load the ring and a new ringbuilder object in a Python REPL:

```
>>> from swift.common.ring import RingData, RingBuilder
>>> ring = RingData.load('/path/to/account.ring.gz')
```

Now, start copying the data we have in the ring into the builder.

```
>>> import math
>>> partitions = len(ring._replica2part2dev_id[0])
>>> replicas = len(ring._replica2part2dev_id)

>>> builder = RingBuilder(int(Math.log(partitions, 2)), replicas, 1)
>>> builder.devs = ring.devs
>>> builder._replica2part2dev = ring.replica2part2dev_id
>>> builder._last_part_moves_epoch = 0
>>> builder._last_part_moves = array('B', (0 for _ in xrange(self.parts)))
>>> builder._set_parts_wanted()
>>> for d in builder._iter_devs():
            d['parts'] = 0
>>> for p2d in builder._replica2part2dev:
            for dev_id in p2d:
                builder.devs[dev_id]['parts'] += 1
```

This is the extent of the recoverable fields. For `min_part_hours` you'll either have to remember what the value you used was, or just make up a new one.

```
>>> builder.change_min_part_hours(24) # or whatever you want it to be
```

Try some validation: if this doesn't raise an exception, you may feel some hope. Not too much, though.

```
>>> builder.validate()
```

Save the builder.

```
>>> import pickle
>>> pickle.dump(builder.to_dict(), open('account.builder', 'wb'), protocol=2)
```

You should now have a file called 'account.builder' in the current working directory. Next, run `swift-ring-builder account.builder write_ring` and compare the new account.ring.gz to the account.ring.gz that you started from. They probably won't be byte-for-byte identical, but if you load them up in a REPL and their `_replica2part2dev_id` and `devs` attributes are the same (or nearly so), then you're in good shape.

Next, repeat the procedure for `container.ring.gz` and `object.ring.gz`, and you might get usable builder files.