SCALITY

openstack
CLOUD SOFTWARE

Integrating Scality RING into OpenStack

Unified Storage for Cloud Infrastructure

A Scality White Paper

Nicolas Trangez
Jordan Pittier
Björn Schuberg

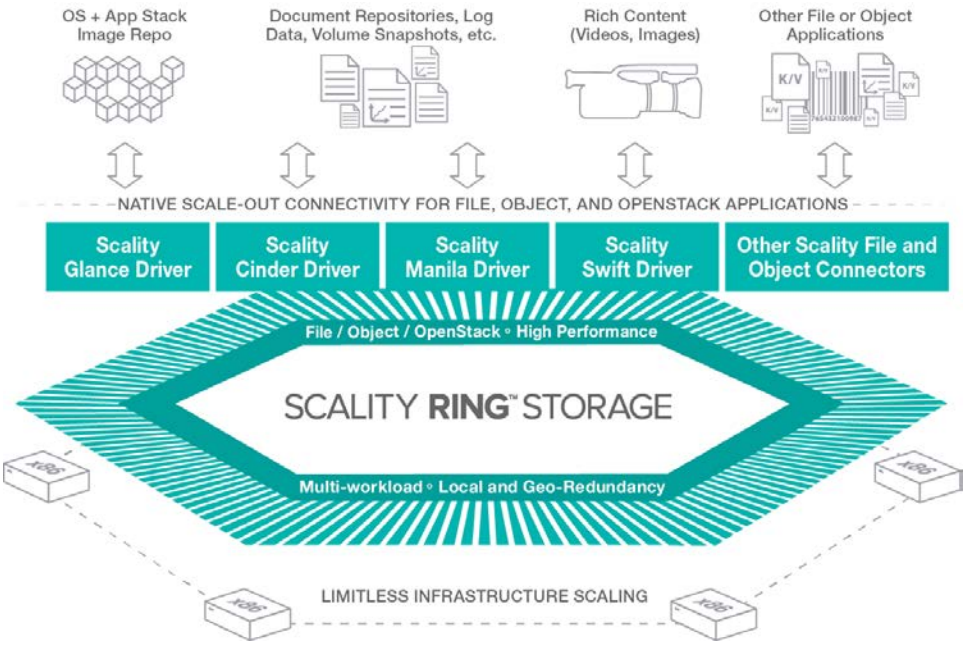
Contents

Introduction

As more organizations move their applications and IT management processes to cloud-style infrastructures, the benefits of consolidating a variety of application workloads to a single highly reliable storage platform are becoming ever more clear. The large public clouds have all taken advantage of this storage approach to gain economies of scale and to leverage industry standard hardware to the extreme, building availability, durability, and other data services into the software layer.

OpenStack is an increasingly viable choice for cloud-style infrastructure management. There are, however, persistence and storage needs throughout each application lifecycle that must be addressed, from the actual virtual machine instances to the data that is captured or served. In integrating its proven RING storage product into all of the common OpenStack storage workflows, Scality has engineered solutions that provide OpenStack users with the high availability, petabyte-scalability, and operational simplicity expected in Enterprise-class storage.

This Scality technical white paper describes how OpenStack users can fully leverage the RING and its integration to OpenStack storage services.



About OpenStack

Founded in 2010 as a joint project of NASA and RackSpace, OpenStack is a free and open-source software platform for cloud computing. The technology stack consists of various projects, most of them focusing on a specific infrastructure service, including storage, compute and networking. Other projects provide generic services like identity management or a shared web-based dashboard interface. Heavily service-oriented, all OpenStack components expose a well-defined ReSTful API which can be used by applications to provision and manage infrastructure.

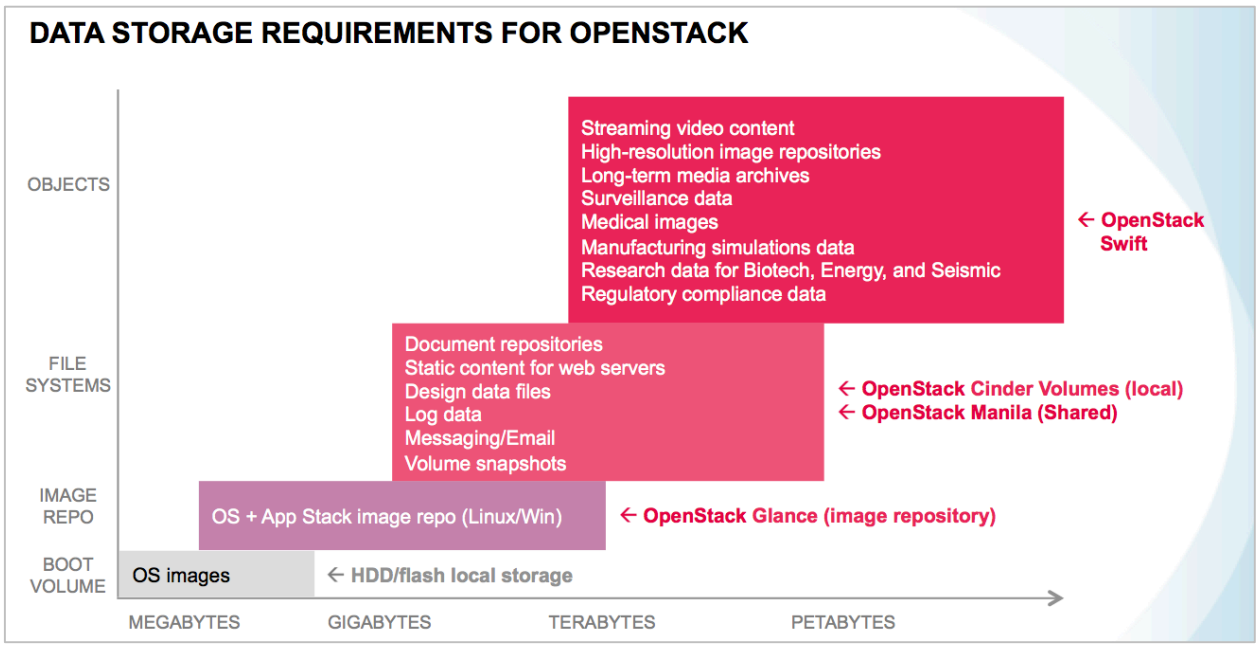
About Scality

Scality is the industry leader in software-defined storage for the information age, serving over 500 million users worldwide. Scality's RING provides 100% reliable storage with unmatched performance, and is perfect for capacity-driven workloads such as cloud services, high-definition video, and enterprise archiving. It runs on any standard x86 servers powered by Linux, such as the ones of HP, Dell, Cisco, SuperMicro, or Seagate, and creates an unlimited storage pool for file, object and OpenStack applications. Thanks to its underlying object storage architecture, the RING scales to exabytes of data and trillions of files. Seven of the top twenty telecommunication companies, petabyte-scale cloud services, and Global 2000 enterprises rely on the RING for mission-critical applications.

Services

Scality supports all four of OpenStack's persistent storage services, each of which is designed to handle a distinct type of data.

- OpenStack Swift (object storage)
- OpenStack Cinder (block storage)
- OpenStack Manila (file storage)
- OpenStack Glance (assets storage, including Virtual Machine – VM – images)



OpenStack Swift

The Swift service of an OpenStack deployment provides object-storage functionality that is similar to the S3 service in Amazon AWS. In Swift, objects are bucketed in containers that belong to a specific tenant. These objects are referenced via a key that is determined at object creation time, and form a flat namespace inside a single container.

To ensure object resiliency, OpenStack Swift relies on the replication of data across multiple machines and storage devices, through data partitioning in a so-called *ring (*a Swift term, not to be confused with Scality RING). This ring concept ensures the safe storage of three data types: account information, container metadata, and actual object data. In addition, OpenStack Swift provides preliminary support for parity-based Erasure Coding as a more efficient storage strategy for large object data (introduced as a beta feature in the 2015.1 'Kilo' release).

To support multiple classes of resilience, OpenStack Swift allows administrators to specify multiple "storage policies" in a Swift cluster. These policies define a certain replication level, Erasure Coding policy, or target rings/drives, and each can have a specific pricing scheme

attached to it. Thus, customers can choose the policy under which objects are stored on a per-container basis at the creation time of the container.

Architecture

OpenStack Swift comprises four loosely-coupled server components, each of which addresses a distinct area of concern: Account, Container, Object, and Proxy. Also playing a key role are the Swift "rings", which contain the topology (i.e., the map) of a Swift cluster.

Swift Servers

The Swift Servers are standalone daemons that can be run either on one server or on several servers. The Account server and Container server provide metadata-oriented services for organizational information (e.g., object listings, ACLs, quotas). The Object server has the sole purpose of storing object data, and the Swift Proxy server (which exposes a well-defined ReST API) ties the components together.

Account Server

An *Account* in Swift maps to a tenant in OpenStack. The Account server mainly provides a listing of the Containers controlled by an Account. Each Account has its own namespace, which allows Container names to be reused across different Accounts.

Container Server

Similar to the way the Account server tracks Container listings for Accounts, the Container server maintains an index of the Objects in each Container. Object access control is also enforced at this level, affecting all Objects in a given Container.

Object Server

The Object server is the only component that is concerned with storing object data. It exposes a rather simple interface, which among other operations provides storage, retrieval, or deletion of a particular object.

How an Object is stored depends on the so-called DiskFile implementation in use. DiskFile abstracts away the actual reading and writing of an Object, thus allowing third-party vendors to provide an alternative implementation. The default implementation simply stores an Object as a binary file on a regular file system, on one of the server's local disks.

Proxy Server

The entry point through which applications or users access Swift is the Proxy server, which ties together the different Swift services. When a client requests a listing of Objects in a Container, for instance, the proxy will check with the Account server and Container server to obtain this list. It also handles data operations by querying for the location of the Object to retrieve or store.

As stated in the introduction, Swift supports Erasure Coding. In such a configuration, the Proxy server handles the encoding/decoding and streams the fragments to/from the Object servers, depending on the nature of the operation.

Swift "Rings"

Several Swift *Rings* are in place to map entities to their physical locations; one for Accounts, another for Containers and one per storage policy for the actual data Objects stored.

Along with being aware of physical servers and devices, Swift Rings are also aware of zones. A zone is merely a concept for grouping servers of an installation into a physical location, such as a fire zone or a power separation of a data center. The object space itself is split into partitions, which are mapped onto all storage devices (e.g., disks).

To achieve data resiliency, objects are usually replicated across the Swift installation. Ideally, these replicas are positioned in different zones, as far away from each other as possible.

Swift Rings are static maps built by an external component called the ring-builder. When a component such as the Account server needs to access its backing data – accounts are stored as SQLite database files – it leverages the corresponding Swift Ring data structure to find the data's location.

Scality Integration

Scality RING can be integrated in an OpenStack Swift setup through the swift-scality-backend package. This package provides an alternative implementation for the storage layer of the Object server (refer to Architecture), replacing the disk-based storage with Scality RING through Sproxyd, the Scality RING's native ReST Connector.

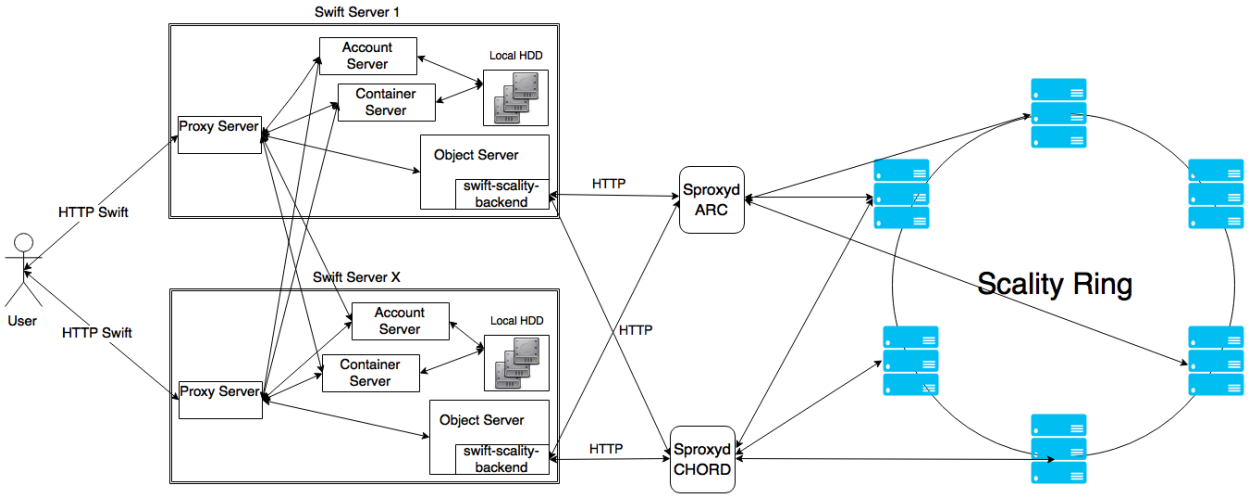
Scality RING integration ensures data safety through replication or ARC (Scality's EC mechanism), and thus OpenStack Swift should no longer manage this key aspect. As such, an OpenStack Swift installation that leverages a Scality RING back-end should be configured to store only a single replica of any object.

The integration fully supports OpenStack Swift storage policies by mapping such policies to sets of Scality ReST Connector endpoints that can be configured differently using multiple classes of service (e.g., replication levels, ARC configurations).

With a geo-distributed Scality RING, the integration can be set to lend preference to datacenter-local Sproxyd services in the storing or retrieving of data. In a mirrored RING setup, the Object server will correctly handle asynchronous replication by:

- Retrieving data from the source RING if it's not yet available in a local deployment
- Sending updates only to the source RING of the object (to avoid inconsistencies)

To ensure high-availability and performance, every Object server should be configured with a set of Sproxyd URLs. The Object server integration will periodically send a request to each Connector to determine connectivity, and feed a failure detection mechanism to take failing Connectors out of rotation. When a Connector is once again deemed stable it is automatically added back to the set of usable Connectors. Finally, a round-robin scheme is employed to spread requests over all reachable Connectors.



Benefits

First and foremost, when integrating OpenStack Swift with Scality RING, the Swift deployment leverages the RING's data safety benefits through its industry-grade replication and EC implementations. The Swift cluster leverages Scality RING's proven support for petabyte-scale object storage, including the product's day-to-day operational features.

As depicted in the Swift Integration diagram, the swift-scality-backend package does not replace OpenStack Swift (Accounts, Containers, security mechanisms are preserved), but instead plugs into the storage layer of an existing Swift installation. The benefits of which include:

- Scality Swift integration is guaranteed to expose the full OpenStack Swift ReSTful API
- Any existing Swift extensions (middleware) can also be used in a Scality RING-backed installation

OpenStack Cinder

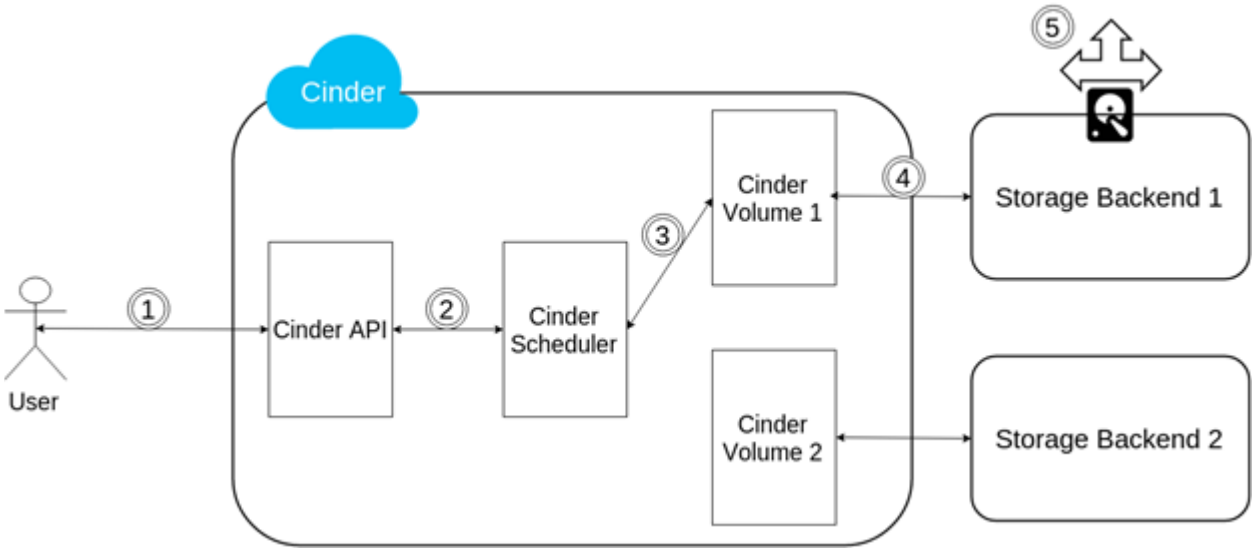
OpenStack Cinder is the block storage provisioning service for OpenStack. Similar to Amazon AWS's EBS service, its purpose is to create and manage storage in block devices called Cinder volumes, which are typically made available for use by virtual machines and which provide persistent storage.

Cinder is an interface between a storage consumer – a Virtual Machine, for example – and a storage provider where the data actually sits, and thus it is merely an API and does not actually store any data. In particular, once the consumer and the provider are connected, Cinder is not in the data path.

Cinder is pluggable, with more than sixty drivers available to manage countless storage provider solutions from almost every viable storage vendor. A single Cinder installation can manage one or more storage backends and provision a Cinder volume on the appropriate one, based on application requirements (utilization, requested performance characteristics, etc.).

Architecture

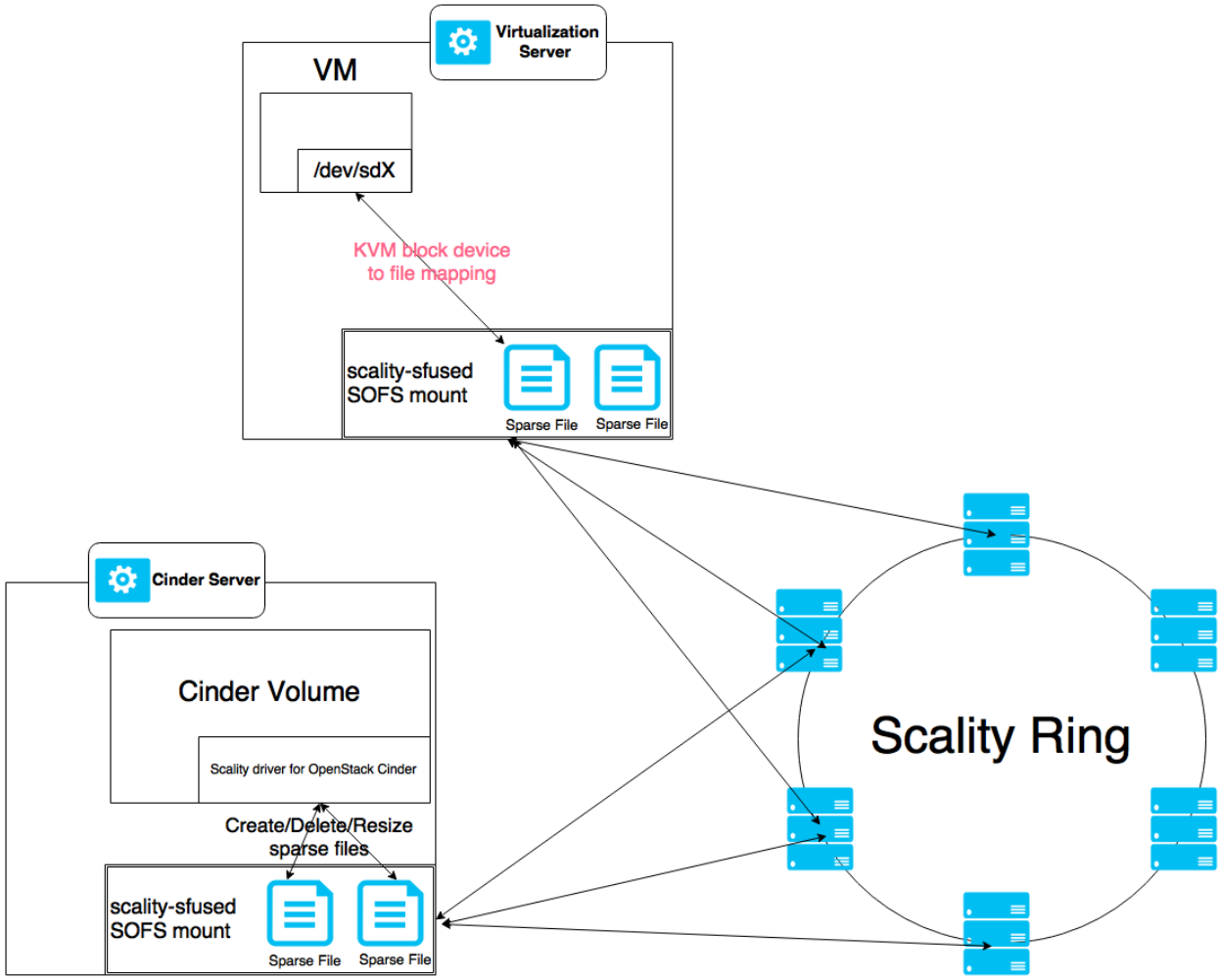
Cinder consists of three component types: Cinder API, Cinder Scheduler and Cinder Volume.



1. A User requests a 10 GB volume with a “Gold” Service Level Agreement (SLA).
2. *Cinder API* checks the user’s authorization and quotas, and if appropriate makes a request to Cinder Scheduler.
3. *Cinder Scheduler* selects the Cinder Volume service best suited to handle the user’s request, based on periodically reported Cinder Volume status information (e.g., free space, utilization, performance characteristics). In the example diagram, the Cinder Scheduler selects Cinder Volume 1 and asks it to create the volume.
4. *Cinder Volume 1* (which runs the driver for *Storage Backend 1*) creates the volume on the storage backend.
5. The volume is made available for external consumption via the network (iSCSI, FC, etc.).

Note: To provide a highly available service, several instances of Cinder API and Scheduler can be run simultaneously.

Scality Integration



With the Scality integration, each Cinder volume is backed by a sparse file stored on Scality's distributed file system, SOFS. The integration relies on a hypervisor to present a virtual block device transparently mapped to a file stored on the hypervisor's file system. Labeled "KVM block device to file mapping" in the diagram, this feature provides virtual block devices to VMs on top of the distributed file system.

The Scality driver for OpenStack Cinder – run as part of the Cinder Volume process – manages the creating and deleting of sparse files on SOFS, while the hypervisor only reads from and writes to existing sparse files. This requires that the *scality-sfused* package be installed and properly configured on all virtualization servers (hundreds, potentially, in a cloud environment) and Cinder Volume servers (usually 1-3). Also, because the sparse files created on SOFS are in the special QCOW2 format (Qemu Copy On Write) and not in RAW format, the Scality driver for Cinder supports efficient, copy on write, Cinder volume snapshots.

Benefits

The Scality driver for OpenStack Cinder is a significant aspect of the company's unified storage vision, providing OpenStack's block storage service with a highly scalable and highly available storage backend, and validating the RING's viability whenever OpenStack stores data.

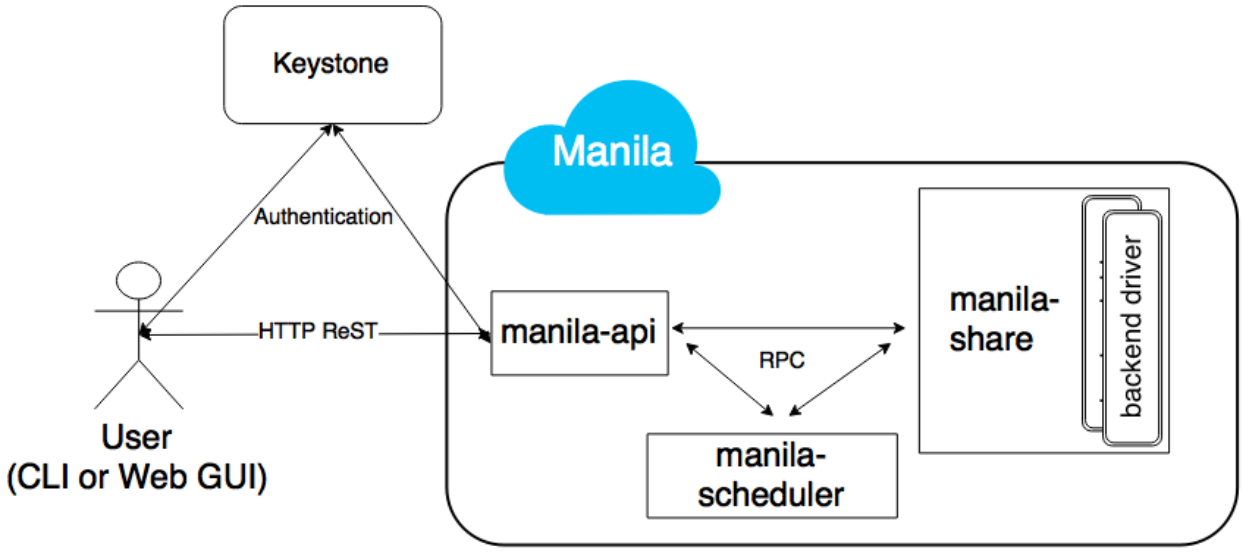
OpenStack Manila

Manila provides a shared file system service for consumption by a cloud tenants' compute instances. Shares provided by Manila can be mounted over the two standard protocols, NFS and CIFS.

At an earlier time, the management of fileshare provisioning was handled by compute instances acting as file servers. Deployment was handled by each tenant, which forced application developers and system administrators to manage maintenance and failure scenarios for the share server. Today, however, Manila alleviates these concerns by providing a first-class OpenStack fileshare service in the same manner in which Cinder provides block devices.

Architecture

Fileshares are provided to the consumer through a number of pluggable share backends, and not via Manila (which like Cinder is merely a provisioning API). Scality, along with a variety of other storage vendors, supports Manila through its own storage backends, several of which may be in simultaneous use.



Three main components comprise OpenStack Manila: *manila-api*, *manila-scheduler*, and *manila-share*. These components are decoupled and can be run on different hosts, with inter-component communication performed via RPC calls over a message queue (e.g., RabbitMQ), which is a common pattern across OpenStack projects. An internal registry is stored in a SQL database that is shared by the components. This registry is light-weight as it only contains the metadata required for Manila to operate.

manila-api

The *manila-api* component is the main entry point for any management task pertaining to Manila-provided shared file systems. It provides a ReST API that is used by both the Manila CLI and Horizon (OpenStack's dashboard), with authentication delegated to Keystone (OpenStack's identity management service).

ReST calls are handled either by retrieving a piece of information directly from the database, or by dispatching RPC calls to the other two Manila components, *manila-scheduler* and *manila-share*.

manila-scheduler

Each configured share backend periodically reports its usage statistics to the *manila-scheduler* component, which uses this information – taking into consideration the user-indicated share type, if any – to determine the backend on which a share will be created.

manila-share

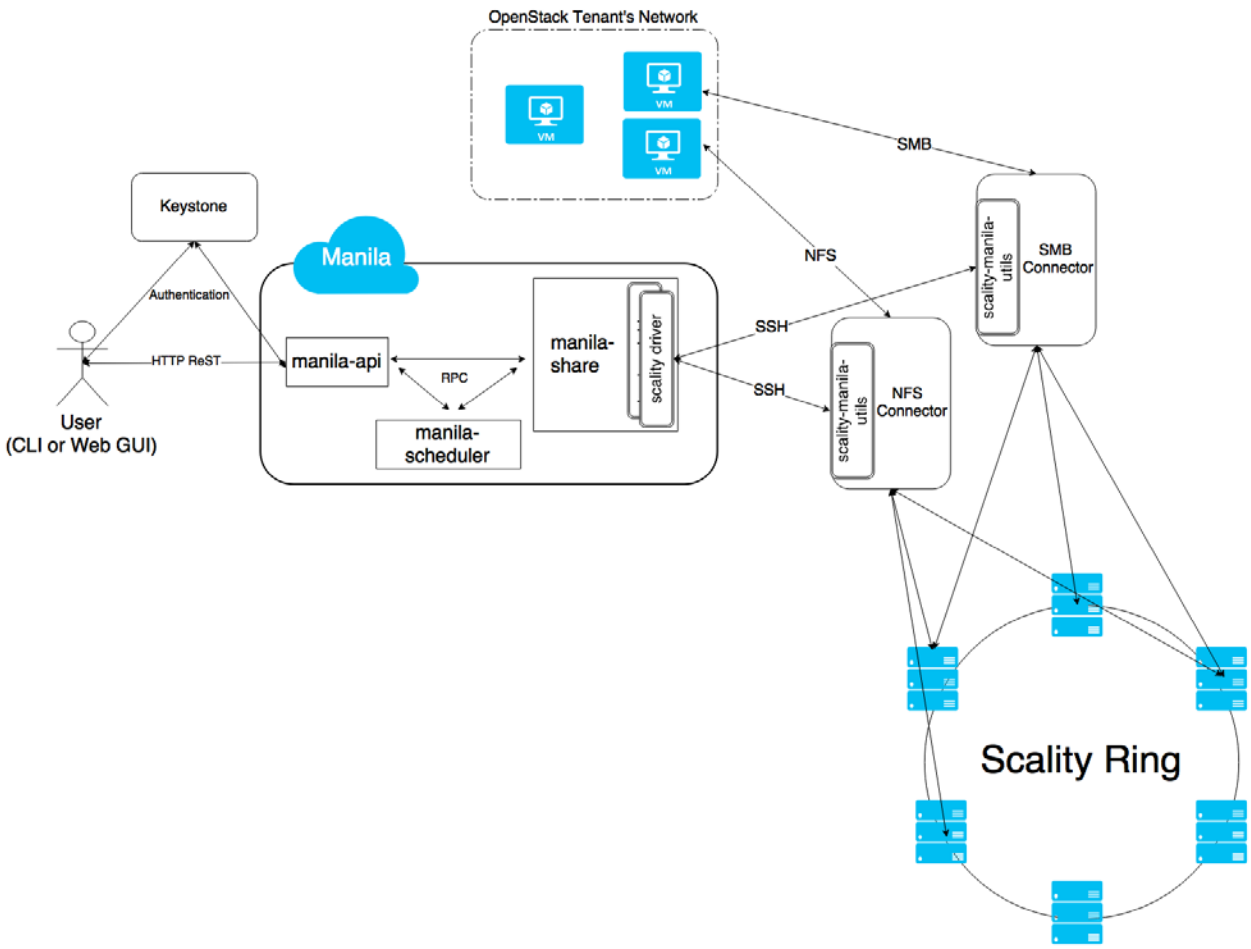
The *manila-share* component handles share-specific operations (e.g., the creation of a new share). It loads the configured share driver, the role of which is to perform essential interactions with the underlying storage solution and ensure that all indicated operations are completed.

In a multi-backend setup, the backing storage solution for a fileshare can be controlled by administrator-defined *share types*.

Scality Integration

Manila integration is two-fold. On one side there is the Scality driver that is loaded by the *manila-share* component, and on the other there are Scality Connectors that expose the fileshares backed by the RING. A Manila share corresponds to a directory residing on a SOFS volume, which is exported through either the Scality NFS or SMB Connector. One Scality volume per storage protocol must be set up, meaning that two Scality volumes are required when NFS and SMB shares are both provisionable.

Assuming that the Scality RING storage solution is installed on a dedicated network segment, the RING Connector must be set up on a host that is reachable from both the OpenStack control plane as well as the OpenStack tenant networks. In addition to the Connector (i.e., *scality-sfused* or *scality-cifs*), the host must have the *scality-manila-utils* package installed to handle the configuration of the Connector for purposes of share management. When *manila-share* delegates an operation (the creation of a new share, for instance) to the Scality driver, *scality-manila-utils* is invoked through SSH to carry out the requested operation.



Benefits

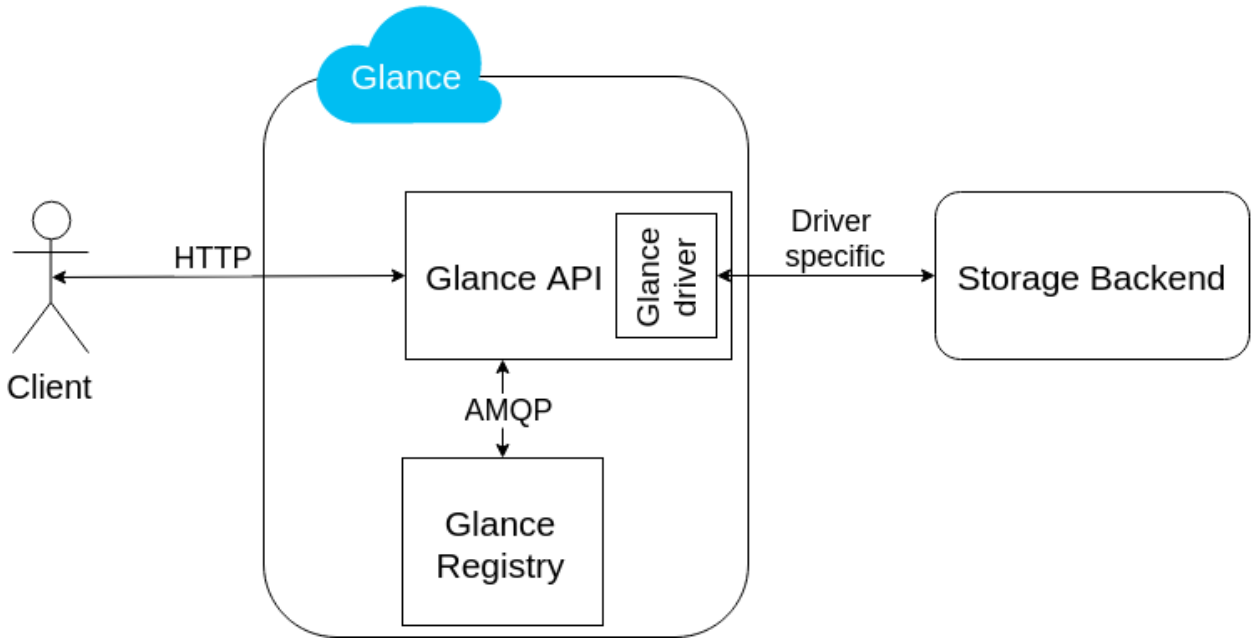
Manila greatly eases the way for applications that require fileshares to make it to the cloud, which is particularly interesting for enterprises that evaluate the cloud for IT. Legacy applications that rely on SMB shares can easily be migrated, as no extra file servers need to be set up in the tenant space to export fileshares. As such, by leveraging the Scality RING as the storage backend for Manila, the safe storage of all fileshares is assured and the system can scale as necessary.

OpenStack Glance

Glance is OpenStack's Virtual Machine images registry, providing a service through which users can discover, register, and retrieve VM images (aka VM templates or "golden images"). These images are full copies of pre-installed Operating Systems that can be duplicated for the purpose of instantiating new VM instances.

Architecture

Two components comprise the Glance service: Glance API and Glance Registry.



Glance API is the client-facing server, exposing a well-defined ReST interface and acting as a proxy between the storage backend and the storage consumer. The API is pluggable and allows each storage vendor to develop drivers, in this way enabling images to be stored on the actual storage solution.

Glance Registry holds the metadata associated with the images, which are small arbitrary key/value pairs that persist in an internal database.

Scality Integration

The Scality-Glance integration works by adding a Glance driver to Glance API which communicates with Scality RING via the Scality ReST Connector, Sproxyd (each Glance image is stored in the RING as a single large object).

Scality's Glance driver can be configured to use several Sproxyd Connectors. The driver has a failure detector (based on periodic health checking) that ensures that only healthy Sproxyd Connectors are accessed, thus increasing Glance's general availability.

Benefits

The Scality driver for OpenStack Glance provides a highly scalable and highly available storage backend to the OpenStack image service, further confirming the RING's viability whenever OpenStack performs data storage.

In Summary

In the realm of data storage, OpenStack provides APIs for the elastic and on-demand self-service provisioning of object, block and file storage.

Scality uniquely provides unified storage services for OpenStack through the implementation of all four of the OpenStack project's persistent storage services, which thus enables storage consolidation for OpenStack deployments. This can reduce the number of distinct storage silos under management, and thereby reduce storage management overhead and the costs associated with running a large-scale cloud environment.