Existing worlflows/methods used to store data in openstack

CINDER(block storage)

Block storage is a fundamental requirement for virtual infrastructures. It is the foundation for storing virtual machines and data used by those machines. Before block storage support was available, OpenStack virtual machines used so-called ephemeral storage, which meant the contents of the virtual machine were lost when that VM was shut down.

Cinder is the OpenStack component that provides access to, and manages, block storage. To OpenStack hosts, the storage appears as a block device that uses iSCSI, Fibre Channel, NFS or a range of other proprietary protocols for back-end connectivity.

The Cinder interface specifies a number of discrete functions, including basic functionality such as create volume, delete volume and attach. There are also more advanced functions that can extend volumes, take snapshots, clone and create volumes from a VM image.

Many storage array suppliers now provide Cinder block device support. These include EMC, Hitachi Data Systems, HP, IBM and NetApp. There is also considerable support for Cinder from startups, including SolidFire, Nexenta, Pure Storage and Zadara Storage.

Most suppliers provide support for iSCSI, with some including Fibre Channel and NFS connectivity.

OpenStack implementations are usually built around scale-out server configurations, so Fibre Channel is not the best choice of protocol. It is likely to be expensive and complex to implement due to hardware costs and the issues of scaling Fibre Channel over large numbers of storage nodes.

NFS support was introduced with the Grizzly release of OpenStack, although it had been brought in experimentally with Folsom. Virtual machine volumes under NFS storage are treated as individual files, in a similar way to the implementation of NFS storage on VMware or VHDs on Hyper-V.

By encapsulating the virtual disk as a file, systems that are able to perform snapshots or other functions at the file level can use this as a way of implementing features such as cloning.

Some startup suppliers support Cinder using their own protocols, for example Scality and Coraid. There are also open-source storage solutions from Ceph and GlusterFS that provide Cinder support using the Ceph RADOS Block Device (RBD) and the native GlusterFS protocol, respectively.

The Ceph implementation is interesting because it uses code that has already been integrated into the Linux kernel, making configuration and support easy to implement. Ceph can also be used as a target for Glance VM images.

Block storage (sometimes referred to as volume storage) provides users with access to block-storage devices. Users interact with block storage by attaching volumes to their running VM instances.

These volumes are persistent: they can be detached from one instance and re-attached to another, and the data remains intact. Block storage is implemented in OpenStack by the OpenStack Block Storage (cinder) project, which supports multiple back ends in the form of drivers. Your choice of a storage back end must be supported by a Block Storage driver.

Most block storage drivers allow the instance to have direct access to the underlying storage hardware's block device. This helps increase the overall read/write IO. However, support for utilizing files as volumes is also well established, with full support for NFS and other protocols.

These drivers work a little differently than a traditional "block" storage driver. On an NFS file system, a single file is created and then mapped as a "virtual" volume into the instance. This mapping/translation is similar to how OpenStack utilizes QEMU's file-based virtual machines stored

Object storage (swift)

Object storage support is implemented into OpenStack through the Swift component. Swift provides a distributed scale-out object store across nodes in an OpenStack cluObject stores reference data as binary objects (rather than as files or LUNs), typically storing or retrieving the entire object in one command. Objects are stored and referenced using HTTP (web-based) protocols with simple commands such as PUT and GET.

In the case of Swift, objects are physically stored on "object servers", one of a number of entities that form a "ring" that also includes proxy servers, container servers and account servers.

A ring represents components that deliver the Swift service. These server components manage access and track the location of objects in a Swift store. Metadata is used to store information about the object. Swift uses the extended attributes of a file to store metadata.

To provide resilience, rings can be divided into zones, within which data is replicated to cater for hardware failure. By default, three replicas of data are created, each stored in a separate zone. In the context of Swift, a zone could be represented by a single disk drive, a server or a device in another datacentre.

Swift uses the idea of eventual consistency when replicating data for resilience. This means data is not replicated synchronously across the OpenStack cluster, but rather

duplicated as a background task. Replication of objects may fail or be suspended if a server is down or the system is under high load.

The idea of eventual consistency may seem risky and it is possible that, in certain scenarios, data may be inconsistent if a server fails before replicating to other nodes in the OpenStack cluster. It is the job of the proxy server in Swift to ensure I/O requests are routed to the server with the most up-to-date copy of an object and, if a server is unavailable, to route the request elsewhere in the cluster.

As Swift has developed, new and enhanced features have been added to the platform.

The Grizzly release of OpenStack provided more granular replica controls, allowing rings to have adjustable replica counts. It also introduced the idea of timing-based sorting for object servers, allowing read requests to be served by the fastest-responding Swift server. This is especially useful in designs that distribute Swift servers over a WAN.

Of course, because Swift is implemented using the HTTP protocol, there is no requirement to store data locally and it would be perfectly possible to store object data in another platform, such as Cleversafe, Scality or even Amazon S3.

Swift provides the ability to deliver resilient scale-out storage on commodity hardware and this may be more preferable and cost-effective than using an external solution.

With object storage, users access binary objects through a REST API. You may be familiar with Amazon S3, which is a well-known example of an object storage system. Object storage is implemented in OpenStack by the OpenStack Object Storage (swift) project. If your intended users need to archive or manage large datasets, you want to provide them with object storage. In addition, OpenStack can store your virtual machine (VM) images inside of an object storage system, as an alternative to storing the images on a file system.

OpenStack Object Storage provides a highly scalable, highly available storage solution by relaxing some of the constraints of traditional file systems. In designing and procuring for such a cluster, it is important to understand some key concepts about its operation. Essentially, this type of storage is built on the idea that all storage hardware fails, at every level, at some point. Infrequently encountered failures that would hamstring other storage systems, such as issues taking down RAID cards or entire servers, are handled gracefully with OpenStack Object Storage.

A good document describing the Object Storage architecture is found within the developer documentation — read this first. Once you understand the architecture, you should know what a proxy server does and how zones work. However, some important points are often missed at first glance.

When designing your cluster, you must consider durability and availability. Understand that the predominant source of these is the spread and placement of your data, rather than the reliability of the hardware. Consider the default value of the number of replicas, which is three. This means that before an object is marked as having been written, at least two copies exist—in case a single server fails to write, the third copy may or may not yet exist when the write operation initially returns. Altering this number increases the robustness of your data, but reduces the amount of storage you have available. Next, look at the placement of your servers. Consider spreading them widely throughout your data center's network and power-failure zones. Is a zone a rack, a server, or a disk?

Object Storage's network patterns might seem unfamiliar at first. Consider these main traffic flows:

- Among object, container, and account servers
- Between those servers and the proxies
- Between the proxies and your users

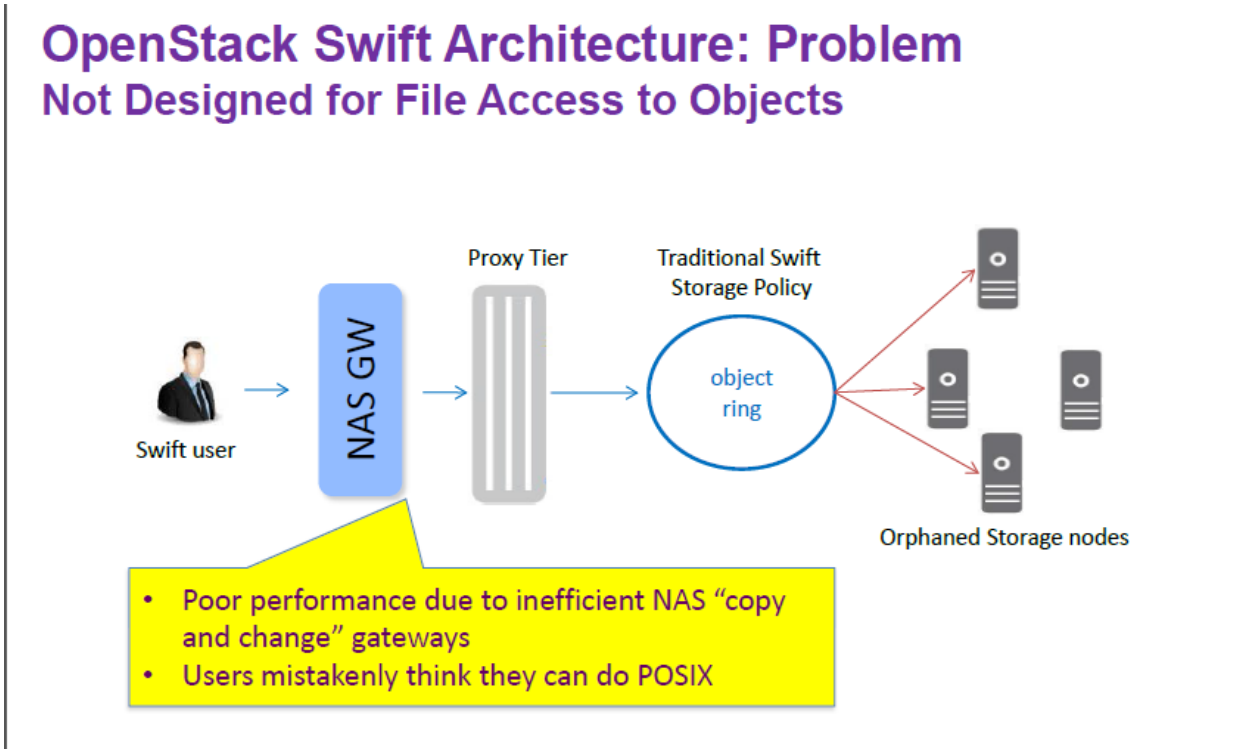
Object Storage is very "chatty" among servers hosting data—even a small cluster does megabytes/second of traffic, which is predominantly, "Do you have the object?"/"Yes I have the object!" Of course, if the answer to the aforementioned question is negative or the request times out, replication of the object begins.

Consider the scenario where an entire server fails and 24 TB of data needs to be transferred "immediately" to remain at three copies—this can put significant load on the network.

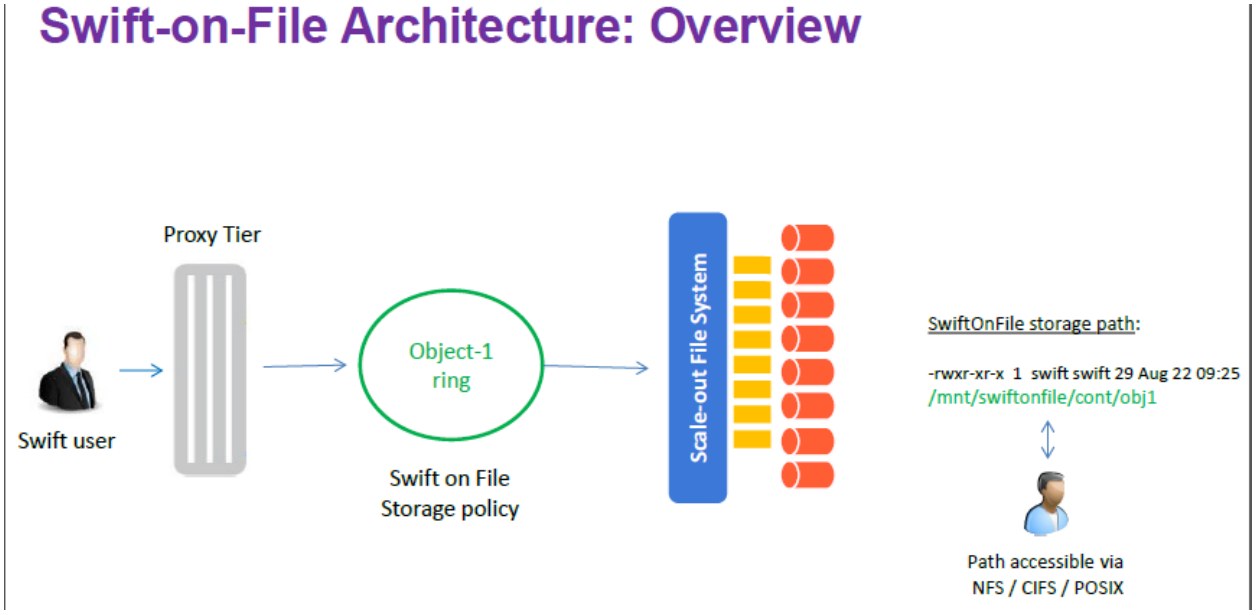
Another fact that's often forgotten is that when a new file is being uploaded, the proxy server must write out as many streams as there are replicas—giving a multiple of network traffic. For a three-replica cluster, 10 Gbps in means 30 Gbps out. Combining this with the previous high bandwidth demands of replication is what results in the recommendation that your private network be of significantly higher bandwidth than your public need be. Oh, and OpenStack Object Storage communicates internally with unencrypted, unauthenticated rsync for performance—you do want the private network to be private.

The remaining point on bandwidth is the public-facing portion. The `swift-proxy` service is stateless, which means that you can easily add more and use HTTP load-balancing methods to share bandwidth and availability between them.

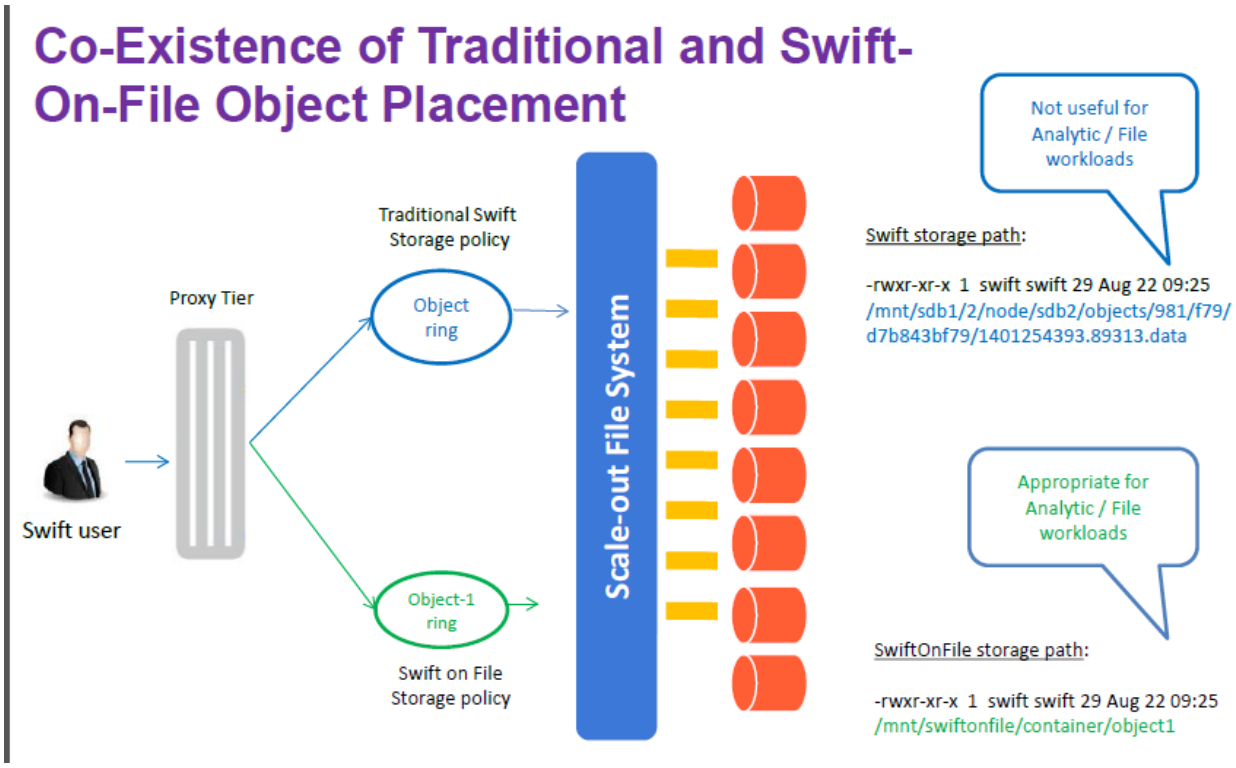
More proxies means more bandwidth, if your storage can keep up.


OpenStack Swift Architecture: Problem
Not Designed for File Access to Objects
Proxy Tier
Traditional Swift
Storage Policy
NAS GW
object
ring
Swift user
Orphaned Storage nodes
• Poor performance due to inefficient NAS "copy and change" gateways
• Users mistakenly think they can do POSIX

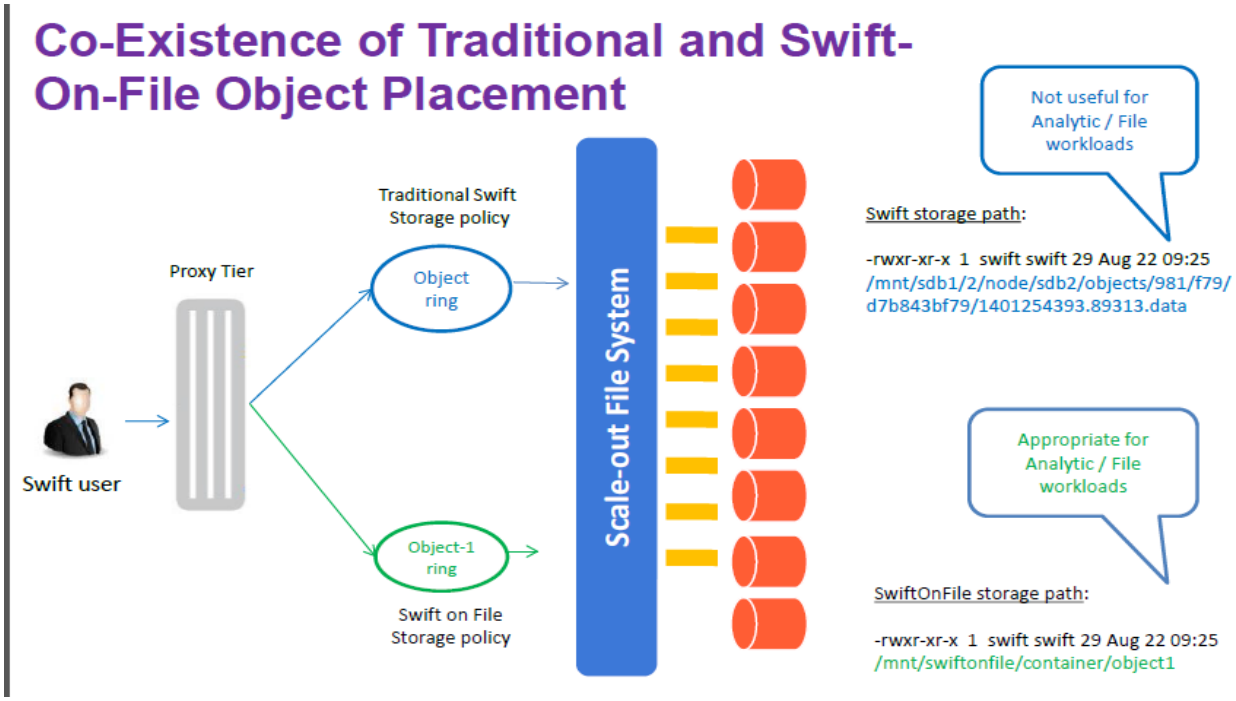
Fig(1)


Swift-on-File Architecture: Overview
Proxy Tier
Swift user
Object-1
ring
Swift on File
Storage policy
Scale-out File System
SwiftOnFile storage path:
-rwxr-xr-x 1 swift swift 29 Aug 22 09:25
/mnt/swiftonfile/cont/obj1
Path accessible via
NFS / CIFS / POSIX

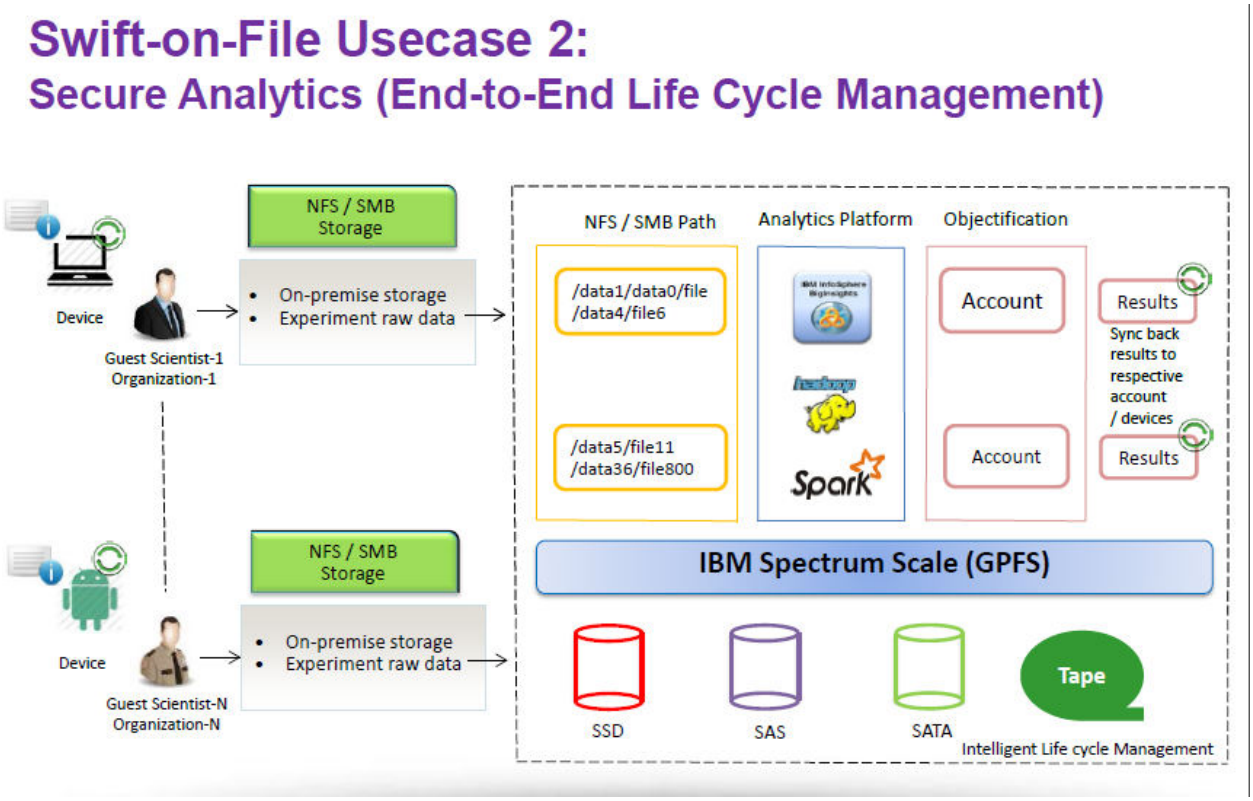
Fig(2)


Co-Existence of Traditional and Swift-On-File Object Placement
Not useful for Analytic / File workloads
Traditional Swift Storage policy
Proxy Tier
Object ring
Swift user
Scale-out File System
Swift storage path:
-rwxr-xr-x 1 swift swift 29 Aug 22 09:25
/mnt/sdb1/2/node/sdb2/objects/981/f79/d7b843bf79/1401254393.89313.data
Appropriate for Analytic / File workloads
Object-1 ring
Swift on File Storage policy
SwiftOnFile storage path:
-rwxr-xr-x 1 swift swift 29 Aug 22 09:25
/mnt/swiftonfile/container/object1

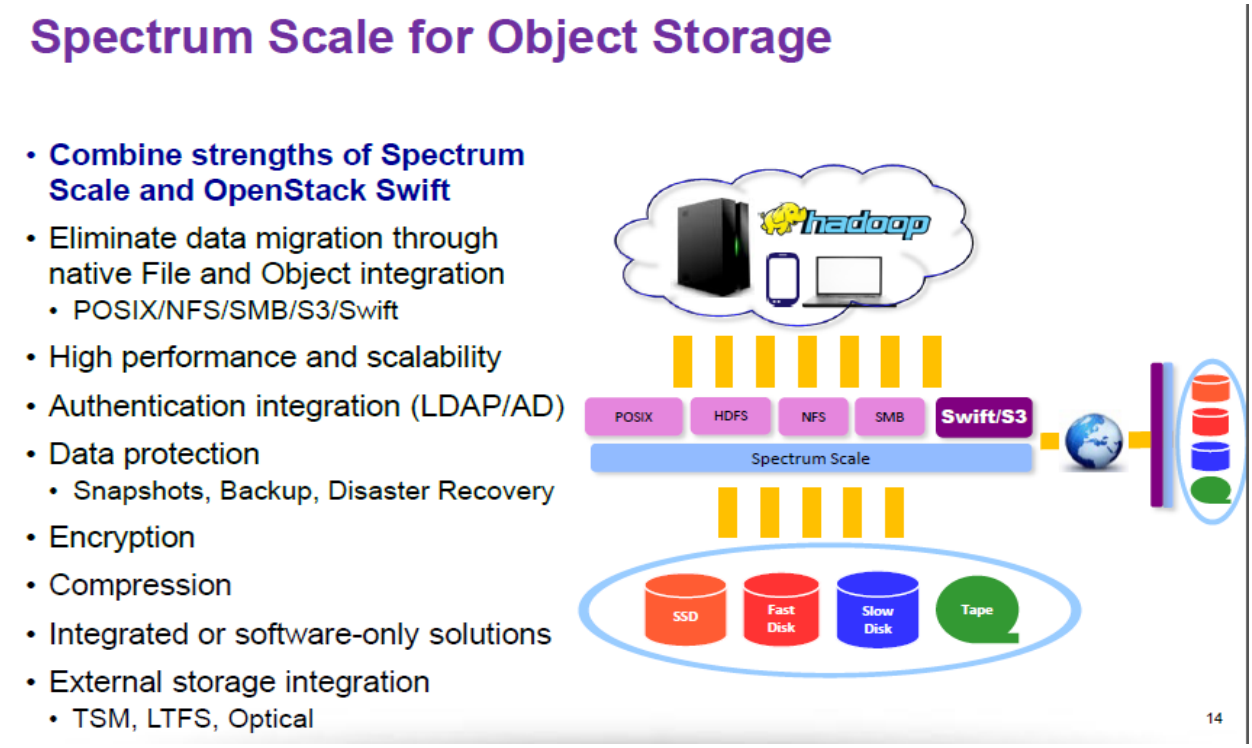
Fig(3)


Co-Existence of Traditional and Swift-On-File Object Placement
Not useful for Analytic / File workloads
Traditional Swift Storage policy
Proxy Tier
Object ring
Swift user
Scale-out File System
Swift storage path:
-rwxr-xr-x 1 swift swift 29 Aug 22 09:25
/mnt/sdb1/2/node/sdb2/objects/981/f79/d7b843bf79/1401254393.89313.data
Appropriate for Analytic / File workloads
Object-1 ring
Swift on File Storage policy
SwiftOnFile storage path:
-rwxr-xr-x 1 swift swift 29 Aug 22 09:25
/mnt/swiftonfile/container/object1

Fig(4)


Swift-on-File Usecase 2:
Secure Analytics (End-to-End Life Cycle Management)
NFS / SMB Storage
Device
Guest Scientist-1 Organization-1
On-premise storage
Experiment raw data
NFS / SMB Storage
Device
Guest Scientist-N Organization-N
On-premise storage
Experiment raw data
NFS / SMB Path
Analytics Platform
Objectification
/data1/data0/file
/data4/file6
/data5/file11
/data36/file800
Account
Account
Results
Sync back results to respective account / devices
Results
Spark
IBM Spectrum Scale (GPFS)
SSD
SAS
SATA
Tape
Intelligent Life cycle Management

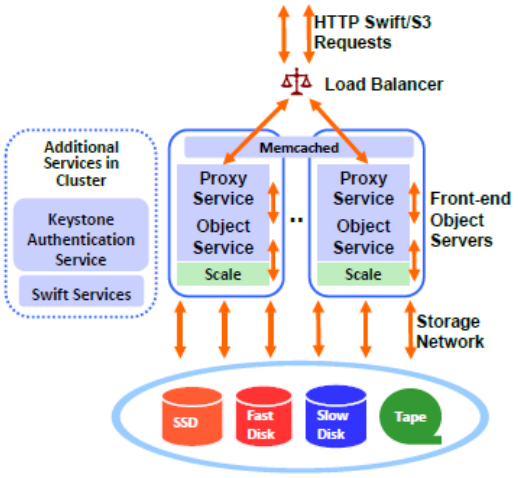
Fig(5)


Spectrum Scale for Object Storage
Combine strengths of Spectrum Scale and OpenStack Swift
Eliminate data migration through native File and Object integration
POSIX/NFS/SMB/S3/Swift
High performance and scalability
Authentication integration (LDAP/AD)
Data protection
Snapshots, Backup, Disaster Recovery
Encryption
Compression
Integrated or software-only solutions
External storage integration
TSM, LTFS, Optical
hadoop
POSIX
HDFS
NFS
SMB
Swift/S3
Spectrum Scale
SSD
Fast Disk
Slow Disk
Tape
14

Fig(6)

Spectrum Scale for Object Storage
Detailed Architecture

- Run all Swift and Scale processes on all front-end object servers
- Front-end servers access data directly from storage system
- Objects are erasure coded on storage cluster using Spectrum Scale Native RAID
- Use Swift policies to map containers to Spectrum Scale Filesets with specific features e.g. encryption, compression.



Fig(7)

Trove/volume-data-snapshot-design

Data volume snapshot

Description

This feature is being proposed as the addition backup/restore strategy

Volumes Snapshots

This introduction provides a high level overview of the two basic resources offered by the OpenStack Block Storage service. The first is Volumes and the second is Snapshots which are derived from Volumes.

slicknik (talk) The BP / spec is supposed to contain relevant information about requirements for the spec, and what changes are proposed to meet the requirements. If you feel that the intended audience might need more background about Volumes, or Snapshots you can link to the Cinder wiki / docs (eg. https://wiki.openstack.org/wiki/Cinder), rather than try and cover all background details in the spec, since it distracts from the purpose of the spec.

Volumes

Volumes are allocated block storage resources that can be attached to instances as secondary storage or they can be used as the root store to boot instances. Volumes are persistent R/W Block Storage devices most commonly attached to the Compute node via iSCSI.

Snapshots

A Snapshot in OpenStack Block Storage is a read-only point in time copy of a Volume. The Snapshot can be created from a Volume that is currently in use (via the use of '--force True') or in an available state. The Snapshot can then be used to create a new volume via create from snapshot.

Backup workflow

The actual flow will be:

1. ask if instance if it has the volume

slicknik (talk) Who is asking whom, and based on what? Can we get some details here?

1. prepare database for storage snapshot

slicknik (talk) Once again which components are "preparing" the database? What needs to be done to "prepare" the database? Does the FS need to be quiesced? What API calls need to be made?

1. snapshot

slicknik (talk) Can a consistent snapshot be guaranteed based on the prepare steps above? Does it depend on the API calls made previously? Do all cinder drivers support and API call to quiesce the FS?

1. return database in to the normal state

slicknik (talk) What does this entail? Is this even a task or a no-op?

Restore workflow

1. create a new volume from the given snapshot

slicknik (talk)Again, what API calls are involved here?

1. swap the volume

slicknik (talk) Swap with what? How does the swap occur? Does the instance come up fully (i.e. prepare call is finished), and then the swap occurs? Or does it happen as part of prepare? If it's

the former, what if the user writes data before the swap occurs? If it's the latter, how do you handle failures in prepare?

1. update backend record

slicknik (talk) Are we proposing to extend the backend record? What new updates need to be made, and to what fields? How do we deal with incremental backups?

1. delete initial volume

slicknik (talk) Seems extra work to create it just to delete it. Why can't we just boot an instance and attach the restored volume instead?

Recovery process

So, lets say, cinder failed to create the snapshot, for the Trove it's like - no problem, lets mark it as FAILED and thats all.

slicknik (talk) I'm unsure what this section is addressing. Are you trying to detail out areas where failures may occur which we have to handle? If so, I definitely see at least a few cases:

- Unable to connect to cinder - (this probably already exists and can be reused)
- Able to connect to cinder, but unable to snapshot and instance
- Unable to quiesce FS so consistent snapshots are not possible
- Able to snapshot, but restore volume from snapshot fails
- Restore succeed, but we're unable to "swap-out" volumes.

This is just a quick list and there are probably other error cases that I'm missing - so this needs to be given some more thought.

Justification/Benefits

Justification

Data could be backued in two ways:

1. Standart backup strategies (innobackupex, nodetoolsnapshot) + Swift container (already implemented).
2. Snapshot of the attached block storage (not implemented).

Basically, its the another way of backupin' data through standard OpenStack capabilities.

slicknik (talk) Well if it's _just_ another way of doing it, why do it at all? We already have a perfectly good way of doing it today, so what's the benefit in adding something else like this if it is just more code to write / maintain?

Benefits

Generic way to backup the data. This feature is not the datastore-type/version specific. Makes Swift storage optional.

slicknik (talk) These are all good points for the justification. You should expand on these.

Impacts

Changes the behavior to the backups made by Trove, it impacts at already implemented backuping process through native database tools (mysqldump, nodetool, etc.) and the Swift as storage container service. Changes are backward compatible.

slicknik (talk) How are changes backwards compatible? What if I was using innobackupex for backups so far, and now I want to switch to using Volumes? What happens to my existing backups? How can I restore from these? You haven't touched upon any of these scenarios.

Configuration

Name	Type	Default	Available variants
backup_agent	String	trove.guestagent.backup.backupagent.SwiftAgent	trove.guestagent.backup.backupagent.CinderAgent
storage_strategy	String	Swift	Cinder

configuration parameters are guest specific.

slicknik (talk) What do you mean by "guest specific"? Are they meant to be different on different guest agents? Please clarify

slicknik (talk) What if I have existing backups using Swift Storage Strategy, and then you switch to Cinder? How do we handle that?

Database

No changes

slicknik (talk) How will we be able to distinguish a swift backup from a cinder backup if there are no DB changes to the backup?

Public API

No changes

slicknik (talk) How does one specify which backup to take if there are no API changes?

Internal API

From trove-api to trove-taskamanger

No changes

From trove-taskamanger to trove-guestagent

No changes

slicknik (talk) How does the trove guest know which backup / storage strategy to pick / use if there are no internal API changes?

Guest Agent

Changes are backward compatible. Changes will be available for all datastores. This method of the backuping is generic for the all datastores types/versions.

slicknik (talk) How are changes backwards compatible? What if I was using innobackupex for backups so far, and now I want to switch to using Volumes? What happens to my existing backups? How can I restore from these? You haven't touched upon any of these scenarios

Elastic Data Processing (EDP)

Sahara's Elastic Data Processing facility or *EDP* allows the execution of jobs on clusters created from sahara. EDP supports:

- Hive, Pig, MapReduce, MapReduce.Streaming, Java, and Shell job types on Hadoop clusters
- Spark jobs on Spark standalone clusters, MapR (v5.0.0 - v5.2.0) clusters, Vanilla clusters (v2.7.1) and CDH clusters (v5.3.0 or higher).
- storage of job binaries in the OpenStack Object Storage service (swift), the OpenStack Shared file systems service (manila), or sahara's own database
- access to input and output data sources in
    - HDFS for all job types
    - swift for all types excluding Hive
    - manila (NFS shares only) for all types excluding Pig

- configuration of jobs at submission time
- execution of jobs on existing clusters or transient clusters

Interfaces

The EDP features can be used from the sahara web UI which is described in the *Sahara (Data Processing) UI User Guide*.

The EDP features also can be used directly by a client through the REST api

EDP Concepts

Sahara EDP uses a collection of simple objects to define and execute jobs. These objects are stored in the sahara database when they are created, allowing them to be reused. This modular approach with database persistence allows code and data to be reused across multiple jobs.

The essential components of a job are:

- executable code to run
- input and output data paths, as needed for the job
- any additional configuration values needed for the job run

These components are supplied through the objects described below.

Job Binaries

A *Job Binary* object stores a URL to a single script or Jar file and any credentials needed to retrieve the file. The file itself may be stored in the sahara internal database (but it is deprecated now), in swift, or in manila.

deprecated: Files in the sahara database are stored as raw bytes in a *Job Binary Internal* object. This object's sole purpose is to store a file for later retrieval. No extra credentials need to be supplied for files stored internally.

Sahara requires credentials (username and password) to access files stored in swift unless swift proxy users are configured as described in *Sahara Advanced Configuration Guide*. The swift service must be running in the same OpenStack installation referenced by sahara.

To reference a binary file stored in manila, create the job binary with the URL `manila://{share_id}/{path}`. This assumes that you have already stored that file in the appropriate path on the share. The share will be automatically mounted to any cluster nodes which require access to the file, if it is not mounted already.

There is a configurable limit on the size of a single job binary that may be retrieved by sahara. This limit is 5MB and may be set with the *job_binary_max_KB* setting in the `sahara.conf` configuration file.

Jobs

A *Job* object specifies the type of the job and lists all of the individual Job Binary objects that are required for execution. An individual Job Binary may be referenced by multiple Jobs. A Job object specifies a main binary and/or supporting libraries depending on its type:

Data Sources

A *Data Source* object stores a URL which designates the location of input or output data and any credentials needed to access the location.

Sahara supports data sources in swift. The swift service must be running in the same OpenStack installation referenced by sahara.

Sahara also supports data sources in HDFS. Any HDFS instance running on a sahara cluster in the same OpenStack installation is accessible without manual configuration. Other instances of HDFS may be used as well provided that the URL is resolvable from the node executing the job.

Sahara supports data sources in manila as well. To reference a path on an NFS share as a data source, create the data source with the URL `manila://{share_id}/{path}`. As in the case of job binaries, the specified share will be automatically mounted to your cluster's nodes as needed to access the data source.

Some job types require the use of data source objects to specify input and output when a job is launched. For example, when running a Pig job the UI will prompt the user for input and output data source objects.

Other job types like Java or Spark do not require the user to specify data sources. For these job types, data paths are passed as arguments. For convenience, sahara allows data source objects to be referenced by name or id. The section Using Data Source References as Arguments gives further details.

Job Execution

Job objects must be *launched* or *executed* in order for them to run on the cluster. During job launch, a user specifies execution details including data sources, configuration values, and program arguments. The relevant details will vary by job type. The launch will create a *Job Execution* object in sahara which is used to monitor and manage the job.

To execute Hadoop jobs, sahara generates an Oozie workflow and submits it to the Oozie server running on the cluster. Familiarity with Oozie is not necessary for using sahara but it may be beneficial to the user. A link to the Oozie web console can be found in the sahara web UI in the cluster details.

For Spark jobs, sahara uses the *spark-submit* shell script and executes the Spark job from the master node in case of Spark cluster and from the Spark Job History server in other cases. Logs of spark jobs run by sahara can be found on this node under the */tmp/spark-edp* directory.

General Workflow

The general workflow for defining and executing a job in sahara is essentially the same whether using the web UI or the REST API.

1. Launch a cluster from sahara if there is not one already available
2. Create all of the Job Binaries needed to run the job, stored in the sahara database, in swift, or in manila
    - When using the REST API and internal storage of job binaries, the Job Binary Internal objects must be created first
    - Once the Job Binary Internal objects are created, Job Binary objects may be created which refer to them by URL
3. Create a Job object which references the Job Binaries created in step 2
4. Create an input Data Source which points to the data you wish to process
5. Create an output Data Source which points to the location for output data
6. Create a Job Execution object specifying the cluster and Job object plus relevant data sources, configuration values, and program arguments
    - When using the web UI this is done with the Launch On Existing Cluster or Launch on New Cluster buttons on the Jobs tab
    - When using the REST API this is done via the */jobs/<job_id>/execute* method

The workflow is simpler when using existing objects. For example, to construct a new job which uses existing binaries and input data a user may only need to perform steps 3, 5, and 6 above. Of course, to repeat the same job multiple times a user would need only step 6.

Specifying Configuration Values, Parameters, and Arguments

Jobs can be configured at launch. The job type determines the kinds of values that may be set:

- *Configuration values* are key/value pairs.
    - The EDP configuration values have names beginning with *edp.* and are consumed by sahara
    - Other configuration values may be read at runtime by Hadoop jobs
    - Currently additional configuration values are not available to Spark jobs at runtime
- *Parameters* are key/value pairs. They supply values for the Hive and Pig parameter substitution mechanisms. In Shell jobs, they are passed as environment variables.
- *Arguments* are strings passed as command line arguments to a shell or main program

These values can be set on the Configure tab during job launch through the web UI or through the *job_configs* parameter when using the */jobs/<job_id>/execute* REST method.

In some cases sahara generates configuration values or parameters automatically. Values set explicitly by the user during launch will override those generated by sahara.

Using Data Source References as Arguments

Sometimes it's necessary or desirable to pass a data path as an argument to a job. In these cases, a user may simply type out the path as an argument when launching a job. If the path requires credentials, the user can manually add the credentials as configuration values. However, if a data source object has been created that contains the desired path and credentials there is no need to specify this information manually.

As a convenience, sahara allows data source objects to be referenced by name or id in arguments, configuration values, or parameters. When the job is executed, sahara will replace the reference with the path stored in the data source object and will add any necessary credentials to the job configuration. Referencing an existing data source object is much faster than adding this information by hand. This is particularly useful for job types like Java or Spark that do not use data source objects directly.

There are two job configuration parameters that enable data source references. They may be used with any job type and are set on the `Configuration` tab when the job is launched:

- `edp.substitute_data_source_for_name` (default **False**) If set to **True**, causes sahara to look for data source object name references in configuration values, arguments, and parameters when a job is launched. Name references have the form **datasource://name_of_the_object**.

  For example, assume a user has a WordCount application that takes an input path as an argument. If there is a data source object named **my_input**, a user may simply set the **edp.substitute_data_source_for_name** configuration parameter to **True** and add **datasource://my_input** as an argument when launching the job.

- `edp.substitute_data_source_for_uuid` (default **False**) If set to **True**, causes sahara to look for data source object ids in configuration values, arguments, and parameters when a job is launched. A data source object id is a uuid, so they are unique. The id of a data source object is available through the UI or the sahara command line client. A user may simply use the id as a value.

REFRENCES

http://docs.openstack.org/ops-guide/arch-storage.html

http://docs.openstack.org/kilo/config-reference/content/firewalls-default-ports.html

http://cloudcomputing.ieee.org/images/files/education/studygroup/Cloud_Management_Software_Platforms_OpenStack.pdf

https://community.netapp.com/fukiw75442/attachments/fukiw75442/virtualization-and-cloud-articles-and-resources/450/1/openstack-deployment-ops-guide.pdf

http://docs.openstack.org/developer/sahara/userdoc/edp.html

https://wiki.openstack.org/wiki/Trove/volume-data-snapshot-design

https://wiki.openstack.org/wiki/Trove/volume-data-snapshot-design

http://docs.openstack.org/security-guide/object-storage.html#securing-services-general