HORIZON 2020 FRAMEWORK PROGRAMME

IOStack

(H2020-ICT-2014-7-1)

Software-Defined Storage for Big Data on top of the OpenStack platform

System Object Model Design and Specification

Due date of deliverable: 31-12-2015
Actual submission date: 31-12-2015

Start date of project: 01-01-2015 Duration: 36 months

Summary of the document

Document Type	Deliverable
Dissemination level	Public
State	v1.0
Number of pages	35
WP/Task related to this document	WP3/T1.1
WP/Task responsible	MPS
Author(s)	William Oppermann, David Coffey, Michael Breen
Partner(s) Contributing	MPS, URV, BSC
Document ID	IOSTACK_D3_1_Public.pdf
Abstract	This document summarizes the activities and progress of tasks of the IOStack project regarding block storage.
Keywords	Block Storage, SDS REST API, Block storage Filters, SDS Architecture

Table of Contents

1 Executive Summary

Virtualized data analytics is a multitenant storage IO system. The storage IO is also a totally blended IO when received by the storage systems, that is, it is undifferentiated, so that the storage is unable to meaningfully assign priorities to the IOs that it receives. Order can be imposed on the IOs that are sent to the storage by adding consumer-side software that can order and prioritize the IO flows to the storage systems. This storage consumer-side software presents to the applications (which are data analytics virtual machines) local storage devices each of which connects to storage arrays through a virtualization layer that makes the IO traffic more controllable. This is achieved by a set of filters in the virtualization layer that can transform the data or enforce service level agreement (SLA) controls. A filter implementing a feature such as encryption can offload processing functions that might otherwise be done at the storage array, while a filter to enforce SLA controls can allow the block device traffic to be throttled up or down depending on its priority.

This document describes an initial design and implementation of block storage filter management, the platform for developing filters, the environment to test filters and the application interfaces to provision storage from storage arrays to the consumer nodes running the data analytic virtual machines.

2 Introduction

The IOStack use case is Big Data Analytics in which the compute component comprises clusters of virtual machines which require storage. This storage must be virtualized within the OpenStack environment. OpenStack provides storage capacity, but OpenStack does not provide storage quality of service (QoS), storage service level agreements (SLA) or in-line data flow services between the storage application and storage provider. When running clusters of virtual machines on an open multi-tenant cloud and where predictability of job run time is required many problems must be overcome. A key challenge in providing storage in a multi-tenant environment is controlling and sharing bandwidth and IOPS (Input/Output operations Per Second) between tenants. Another challenge is providing inline processing of the data flows, examples of which could be encryption or compression.

The model for block storage within IOStack is based on the key concepts of the storage provider, the storage consumer, and a managed connection between the provider and consumer. The storage provider is generally thought of as a traditional proprietary storage array with the storage consumer a node which runs storage centric applications, examples being a cluster of Big Data virtual machines or a database or an object store. The managed connection is the set of fabric paths and services between the storage centric applications on the consumer node and the storage providers. It is important to note that the managed connection is beween the storage centric application and storage provider: the storage application is frequently virtualized and is therefore mobile. The managed connection in this case can be though of as a rubber band of fabric paths following the virtual instance of the storage centric applications on whatever consumer node the application is executing on.

Storage technologies in this datacentre environment struggle because there is a requirements list that is very difficult to fulfil, and no single storage technology can deliver all requirements. We can identify six core storage requirements:

- Resiliency
- Security
- Scalability
- Diverse Workload Range and Multitenancy
- Diverse Consumer Types
- Low CapEx and OpEx (Capital and Operational expenditure)

By *resiliency* we mean not only the storage provider is resilient to failures but also the end to end connection between the storage provider and storage consumer. By *security* we mean systems that protect access, integrity and decoding of the data. By *scalability* we mean that the storage service can scale in capacity and performance dynamically without interruption of service. By *diverse workload range* we mean applications that span from low latency high IOPS bound to high MB/s data streaming applications all delivered to a diverse range of consumers such as virtual machines, storage applications or bare metal servers. OpEx and CapEx concerns are also important as the cloud business model is being driven by its consumers as a utility model, the utility model being a low cost model where every aspect of the service is metered and monetized. The candidate storage technologies are block, file and object storage. IOStack storage providers are either Block storage or Object Storage. This workflow focuses on block storage. Each of block, object and file technologies has advantages and disadvantages. In all use cases, block storage semantics for IOPS or MB/s lead to higher performance compared to object storage, while in some cases parallel file systems file systems can out perform both block and object storage.

The IOStack project addresses the diverse workload and security issues through the use of in-band filters and services through which block storage IOs flow. Scalability and resiliency are part of the design of software defined storage. Software defined storage with its focus on open hardware platforms and automation through control APIs reduces both CapEx and OpEx.

Block storage is more complex to manage than object storage and more complex to scale, however this block storage complexity is offset by the use of software defined storage technology. Block storage has a well defined resiliency and redundancy model which again is complex to manage compared to the more limited resiliency and redundancy model of Object storage.

The storage management framework for storage that can deliver these six core requirements must be highly automated and very flexible. The technology that can deliver this type of storage management is software defined storage.

2.1 Software Defined Storage technologies review

Software Defined Storage (SDS) technology is based on the same principals as software defined compute and software defined networking[1], i.e., a physical device is virtualized into a virtual device, the virtual device is controllable through an API, the virtual device is now software defined, and a software controller manages these virtual devices.[2]

The model for each device is consistent. For example, OpenVswitch transforms a physical ethernet device into a virtual ethernet device which is controllable through the OpenVswitch API. This model can be mapped to storage, the transformation function being the Virtual Block store which creates the virtual storage device (VSD). In IOStack the Virtual Block Store, called Konnector, is instantiated on the consumer node and provides to the consumer node applications a virtual storage device. Multiple Konnector VSD devices are managed by an SDS controller. The SDS controller itself is managed through a higher level programming interface.

The management of the virtual block storage controller is through the SDS controller, the role of which is to instantiate and control the virtual devices through a virtual device API. The SDS controller keeps all the metadata of the virtual device, allowing the virtual device to be instantiated anywhere in the datacentre without regard to physical storage devices. Virtual devices are inherently mobile as the SDS controller keeps all the device configuration information in the SDS CMDB (Common Management Data Base).

Software defined storage is a technology that is evolving: standards do not exist nor is there an agreed definition as to what Software Defined Storage (SDS) is. The IOStack project is developing a fully working model of block and object storage implementation for OpenStack.

The term "SDS" was quickly adopted in the storage industry after EMC in 2014 bought the Software Defined Networking (SDN) company Nicira for $1B[3]. Many traditional storage products were given a thin wrapper and relaunched as SDS solutions – usually by providing a plug-in for OpenStack. EMC ViPR was a solution developed around EMC storage arrays. The intent was to deliver a storage solution which would do for storage what Nicira did for networking. However, SDS solutions have not had the same impact in storage as SDN has had in networking. The SDN impact is important to analyse as it can inform the SDS debate. One aspect of SDN innovation is that it adds value to the "consumer" client side node, which is made possible by the client side node being a constantly-evolving, open-platform node, and dominated by open source Linux-type operating systems. In contrast, the storage provider side (storage arrays) are very stable, closed proprietary systems, developed over many years to provide a mission-critical function. Because they are inflexible and change so slowly, they are not very good candidates as an innovation platform. SDS is a technology that allows consumer-side nodes to add functionality on top of the inflexible Storage provider. SDS shifts management intelligence into the SDS controller provisioning between consumer and provider and ultimately all stages in between. Storage arrays will always be a core pillar of IT, however with SDS the provider storage is just one node in a connection, the termination of that storage in the consumer is of equal importance.

In this document we describe the block storage architecture used in the IOStack project. We cover the following topics:

- The storage provider and consumer model for IOStack
- SDS overview

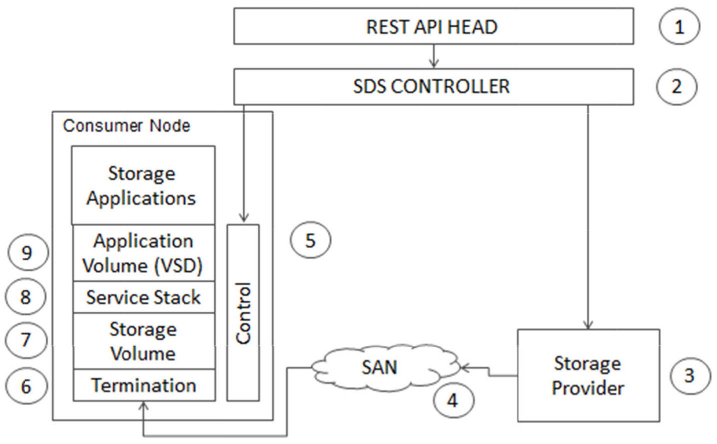


Figure 1: IOStack block storage provider-consumer model.

- Block storage for IOStack
- Filter framework
- Prototyping environment
- REST API

3 IOStack Block Storage Provider-Consumer Model

The provider-consumer model in the SDS framework can be explained with reference to Software Defined Networking. SDN has little to do with network switches: the key invention of SDN is to create a managed virtual in-band switch on the consumer node, the virtual switch being managed by an out-of-band SDN controller which automates the process of network configuration. Building on these SDN concepts the IOStack architecture extends them to an SDS framework.

Figure 1 shows this model, in which each device is transformed into a virtual device through a software layer. This creates a disaggregated pool of virtual devices which can then be combined together into virtual systems.

1. The REST API (Representational State Transfer Application Programming Interface) head allows storage to be provisioned at a storage provider node and attached to a storage consumer node and a service stack to be built between the attached storage and the storage application. The REST API integrates the user facing interface but also includes a generic internal storage object model that can map to a backend SDS controller or storage controller. The simplest SDS controller is a single storage array, but a more complex SDS controller could manage several different storage arrays each with a specific syntax. For the purpose of the IOStack project we are using the MPSTOR Orkestra SDS controller and Orkestra SAM storage arrays.

2. The SDS controller builds a model of the storage provider, understands the management semantics of one or more storage providers. In the general case, the SDS controller is a plug-in to the SDS REST API head. The SDS REST API head could manage several SDS controllers, each SDS controller managing several storage provider types. The SDS Controller is the provisioning engine, the entity that talks to the storage providers.

3. The storage provider is typically a proprietary storage array which has its own proprietary storage API or uses an industry standard such as SMI-S[1]. The SDS controller understands the storage provider syntax and storage model.

[1]Storage Management Initiative Specification

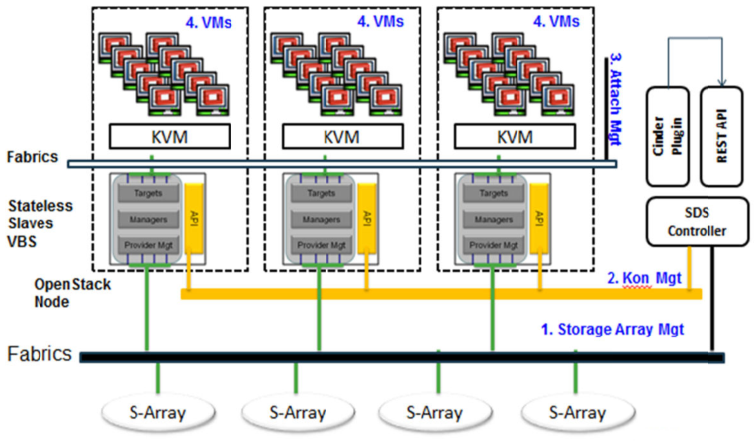


Figure 2: Software Defined Storage at Scale

4. The SAN is a storage area network over which storage volumes are exported from the storage provider and attached to the storage consumer node
5. The Control layer on the consumer node provides a control API to the SDS controller and/or the SDS REST API
6. The Termination layer allows the storage provider volumes to be terminated on the consumer node. This is a complex process as it requires setting up an end to end SAN connection between the consumer and provider.
7. The storage volume from the storage provider.
8. The service stack is a set of block storage volume functions, including filters which provide in-line data transformation functions.
9. The Virtual Storage Device (VSD) is the volume that sits on top of the service stack and is used by the application layer.

4 SDS overview

Software Defined Storage is a framework that sits on top of traditional storage. SDS automates storage provisioning from storage providers to storage consumers. Automation is achieved by automatic discovery of storage providers, storage consumers, the creation of meta-data and policies by the SDS tools, with these metadata and policies being interpreted by the SDS provisioning engine. Provisioning of storage from providers to consumers is simplified through a REST API. Policies are created within the SDS system which defines many of the properties of the storage, such as media tier, its resiliency, performance in BW and IOPS, fabric and levels of redundancy.

Metadata is used by the provisioning engine and is visible through REST API user policies and at the OpenStack user presentation layer. This allows the end-user to choose which storage service to use for his application. The end user can choose not only the capacity he needs but also the SLA/QoS that he requires for his application.

Provisioning storage can be simple if user choice is limited. When multiple SLA and QoS tiers with different storage policies are presented to the end-user than the provisioning of the back end storage in the datacentre is complex. Storage must be provisioned from different storage tiers, across different fabrics with different QoS (such as BW or IOPS rate limiting). This is simplified by using policies to encapsulate these properties and presenting to the user the different policies through the REST API. SDS provides a way of abstracting all the backend datacentre fabrics, storage tiers into simple storage services the user applications can consume. Figure 2 shows the scope of IOStack

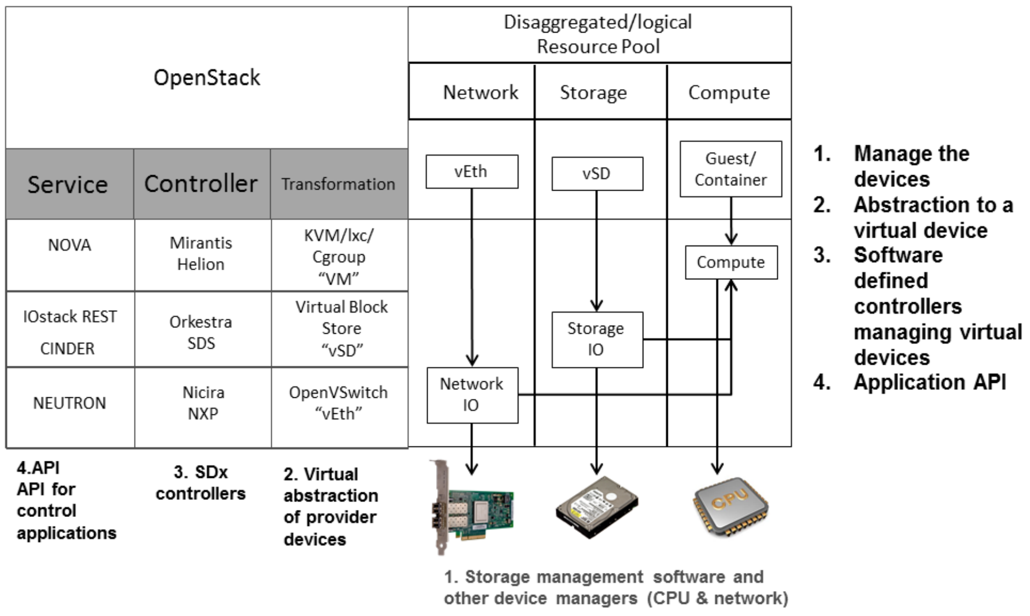

OpenStack
Disaggregated/logical Resource Pool
Network
Storage
Compute
Service
Controller
Transformation
NOVA
Mirantis Helion
KVM/lxc/ Cgroup "VM"
IOstack REST CINDER
Orkestra SDS
Virtual Block Store "vSD"
NEUTRON
Nicira NXP
OpenVSwitch "vEth"
vEth
vSD
Guest/ Container
Compute
Storage IO
Network IO
1. Manage the devices
2. Abstraction to a virtual device
3. Software defined controllers managing virtual devices
4. Application API
4.API API for control applications
3. SDx controllers
2. Virtual abstraction of provider devices
1. Storage management software and other device managers (CPU & network)

Figure 3: Software defined model

block storage. The SDS REST API provides a storage application API which translates the REST requests into downstream API calls to:

- The SDS controller (which in turn manages a number of storage arrays and fabrics)
- The Konnector termination point in the consumer node, this allows the REST API calls to attach storage from a storage array to a consumer node
- Attach of the Konnector storage device to the Virtual machine (this is OpenStack specific and part of the Konnector plugin)

The SDS controller provides an abstraction of the storage to a storage application (for example, OpenStack Cinder). Volumes are created on the storage array and terminated on the consumer nodes using the Konnector management API. A service stack and a set of filters is created on each storage volume. At the top of the filter stack a Virtual Storage Device (VSD) is presented to the application. In this example we have two stages, stage 1 between the storage array and consumer node, stage 2 between the service stack and the VM. In Figure 2 we note a typical use case where an OpenStack Cinder interfaces to the REST API. The Rest API manages an SDS controller which manages the proprietary storage arrays and the in-band Konnector API on each consumer node. Each consumer node may have attached storage array volumes from several storage arrays.

Figure 3 shows the disaggregated Software Defined model.

SDS and its managed storage have ten key characteristics:

1. Virtualization: Managing volumes accessed over a range of protocols and fabrics as mobile virtual containers that map to physical storage.
2. Service Creation: Admin tools that create storage meta-data which can be presented in end-user self-service dashboards and also be interpreted by provisioning engines that manage the physical storage.
3. Programmable: API's that allows control of the SDS controller, the storage arrays and consumer nodes.

4. Automation: The SDS process that creates storage array volumes and connects these volumes to the consumer, the SDS uses metadata to create and attach the volumes.
5. Scalability: The capability of the system to add capacity and performance without interruption of service.
6. Volume Mobility: The capability of the virtual storage volume to track the consumer as the consumer moves, virtual volumes have no mass, the physical storage does have mass.
7. Policy Management: The capability to create high-level user-visible policies used the SDS controller in the provisioning and management of storage.
8. Instrumentation: The creation of metadata from probes from the dynamic usage of the system.
9. Diagnosis and recovery: The capability to detect, present failures and initiate recovery procedure.
10. Resource Management and Transparency: The capability to track usage and provide insight to demand requirements. The capability to provide to the user his resource usage.

5 SDS Compared with Traditional Storage

The benefits of SDS compared to the traditional model may be seen by considering the storage workflow.

5.1 Deployment

The traditional approach and SDS both require planning, physically installing, and cabling new storage hardware. Traditionally, however, storage sits in independent silos and must be recorded and managed separately by an administrator.

SDS, in contrast, is increasingly part of a hyperconverged framework which includes tools to build datacentre rack systems, for example, using MPSTOR Cloud Configurator or Mirantis Fuel. In this approach, once physical storage has been installed, it can be automatically discovered and configured centrally through an interface that presents and maintains a unified picture of the installed storage nodes and their connectivity, greatly simplifying the installation effort.

5.2 Provisioning

Traditionally, the user makes a request for storage to an administrator. To make the provisioning decision, the administrator refers to a large database of information taking into account dependencies and constraints in order to identify a suitable storage device. The volume is then manually created on that storage node and exported using the required storage protocol. It is made available to the consumer - this may involve, e.g., choosing a CHAP[2] password, specifying the consumer node's IQN[3] or the consumer port ID, etc. Either the administrator or the user may then manually complete the attachment on the initiator (consumer) side to make it available as a local device to the application. This is a relatively complex, laborious task subject to error and requires knowledge of the SAN and the end points.

In the SDS model, the user request for a volume, specifying the required capacity and other characteristics of the storage, goes to the SDS system to be processed automatically without operator intervention. In MPSTOR's solution, for example, storage policies are defined in advance by the administrator with attributes defining the quality and performance of the storage (distinguishing, e.g., storage based on SSDs from that on spinning disks, or storage which an administrator has decided should be throttled from storage without bandwidth throttling, etc.), as well as the protocol and network fabric over which the storage is made available (e.g., iSCSI over ethernet). These policies allow the combinations of characteristics selectable by the user to be limited to those planned for and optimized in the chosen installation, simplifying the interface for the user. Depending on

[2]Challenge-Handshake Authentication Protocol
[3]iSCSI Qualified Name

the requirements of the consumer, a block storage volume may also be attached automatically across the network to the consumer node, appearing immediately as a usable device – all without operator intervention.

5.3 Operations

Traditionally, an administrator records and monitors consumption of resources. Changes to the system are difficult to manage as all the dependencies are managed manually (e.g., a change of FC HBA[4] may require configuration steps to be repeated). There is no single, overall management interface, and in the case of a fault condition the dependency tree is relatively opaque. Resource management and capacity planning is manual. Path failures to the consumer are not normally monitored and will be investigated and discovered only when a user reports a problem.

With SDS, a single pane of management allows the administrator to view the static and dynamic attributes and dependency chains of all physical and logical entities. The SDS controller can present different views of the compute and storage entity relationships. The SDS CMDB contains the current global used/unused capacity and its usage statistics. Alerts are generated when a provider device fails and when an attached path between provider and consumer fails.

5.4 Summary

Traditional storage is admin-centred: an administrator, when available, manually provisions storage for a user based on a user request. SDS creates a streamlined, self-service, user-centred model: once the infrastructure has been configured in an initial step, a user may at any time request storage of a given size and quality and a resource satisfying these criteria is automatically provisioned and appears immediately as a local device. For an administrator, the SDS model simplifies all aspects of management and operations, eliminating repetitive, laborious tasks, and making non-repetitive tasks much easier and faster to accomplish.

6 Block Storage and Filter Management for IOStack

6.1 Concepts and Definitions

In describing the model shown in Figure 4 between the storage array and consumer node a set of definitions is required.

Data flow: The IO operations between a client node application and the storage array. The IOs contain no information that is exploited by the Konnector, i.e there are no additional protocol headers in the data. Metadata can be created from the IOs such as BW, IO rate, and path information. This metadata can be used by the services and filters in Konnector, for example, to set an individual BW/IO limit per flow, to cap the tenant space overall BW/IO limit, to set priority levels between flows in a tenant space or within an entire node, to implement a caching tier per flow, to compress or encrypt a flow. This list is not exhaustive.

Endpoint: an endpoint provides an interface for a flow. Examples of endpoints are SAMBA clients and servers, NFS clients and servers, FC/SAS/FCoE/iSER/iSCSI targets and initiators.

Source/Destination Endpoint: Extremes of a data flow communication. This applies to both block and object flows.

Filter: Transformation to be applied to a data flow (compression, IO QoS differentiation).

Stage: Regions in between end-points (read/write path) where filters can be deployed and enforced. Flows ingress and egress across stages which are controllable by APIs. If Konnector presents its virtual block device (VBD) directly to a consumer in the node in which it resides then it is an end-point; if Konnector exports its VBD to another node it is a stage.

Stage Service: Stage services are value added services the stages can provide, such as compression. These may be bidirectional, e.g., encryption/decryption, or unidirectional, e.g., throttling writes.

Protocol: a property of an endpoint. The protocol defines how flows transfer between the endpoints of stages such as the client node and the storage array.

[4]Fibre Channel Host Bus Adapter

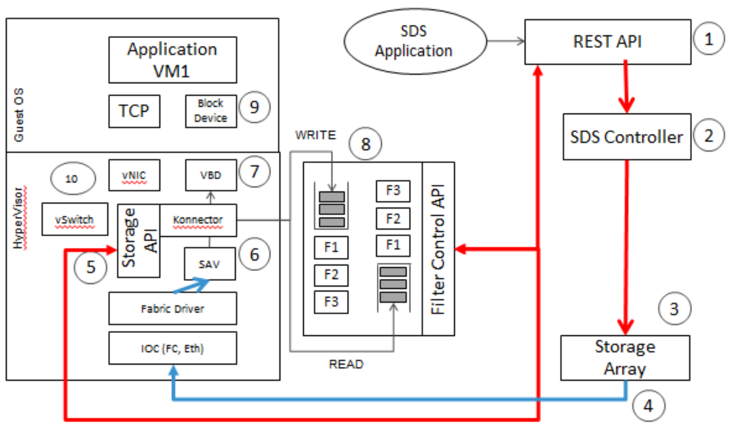


Figure 4: Konnector, the client side storage manager.

SDS Controller: Autonomous entity that has a holistic view of the storage system and orchestrates the application, configuration and deployment of filters across stages. The SDS controller manages both endpoints of a flow using the stage API: the stage in the Storage Array for the export (5) from the storage array, and the termination of the flow in the consumer client node using the Konnector API.

SDS Metadata (CMDB): Database that contains the metadata and configuration of the datacentre, including in our case the SDS configuration (filters, available hardware, ...).

Policy: used to create the default configuration of services in each stage that is applied to a flow. A policy could contain information such as the default fabric, caching, IO throttle, compression, encryption, and other user-defined filters. The client side management of storage is implemented in the Konnector module. The Konnector module provides an interface used by the SDS controller to terminate storage volumes from the storage arrays. The user application through the REST API provisions storage volumes and attaches those volumes across any media tier to the application node. Once the storage array volume (SAV) is attached to the Konnector module in the client node a set of services or filters can be built on that volume. An application volume is built on this stack of storage array volume plus services/filters and presented to the user application. This volume is the virtual block device (VBD). All application block device data will flow into the VBD through the services and filters provided by Konnector that have been configured for that volume. Each component of the Konnector module is decribed in detail below.

6.2 SDS Architecture for Block Storage

Figure 4 shows the IOStack architecture overlaid with MPSTOR-specific components (in blue) and IOStack (in red).

1. The SDS REST API manages the in-band controller Konnector. The SDS controller also contains a plug-in to manage any downstream SDS controller/storage array.

2. The SDS controller manages the storage endpoints of a flow. In this regard it is responsible for the export (4) from the storage array and termination of the export (6) on the consumer node.

3. This is the storage array API.

4. This is the storage array volume export created through the REST API-SDS Controller-Storage controller.

5. This is the API that Konnector presents to the SDS Controller. Konnector, which is the IN-BAND component on the node that homes the consumer application (e.g VMs), its role is to

terminate the Storage Array export (4), and create a service stack or set of filters and present the storage to the VM through the VSD (7). The Konnector layer (KON) presents an API to the SDS controller which the SDS controller uses to create VSD devices. Konnector is a stage, i.e it contains flows and is controllable by an API.

6. This is the export from the storage array (4) that is terminated on the Konnector consumer node. The export is built on the storage array, an export is a function is defined by (port, protocol, host, SAV). Examples of protocols are CIFS, NFS, iSCSI, iSER, FC, SAS, FCoE.

7. This is the virtual storage device. Data flows through this device to a set of services and filters(8). The services and filters in the data path are user controllable through the REST API.

8. These are the internals of Konnector which allows control of the data flows through filters presented to the VM through the VBD. These internals show two filter stacks on a write data queue and a read data queue. Data written or read to the block device are passed through the filters F1, F2, F3 as shown above.

9. This is the block device seen and used by the application, which could be a virtual machine in a big data cluster. Data read or written to this device will pass through the filters in Konnector that have been configured, through the filters to the back end storage array volume and finally across the configured protocol to the storage array volume.

7 Filter framework

7.1 Enabling technology

In Linux, there is a sharp distinction between kernel and userspace, with the former incorporating only reliable, secure, privileged code, and the latter offering the flexibility of being able to run any code without the danger of crashing critical functionality on which all programs depend. A prerequisite for implementing flexibly-deployable filters for block storage was to enable code to implement filters for block devices to run in userspace, even though those block devices are presented by kernel code. This is made possible by a recent project, TCMU, or Target Core Module in Userspace[4, 5]. MPSTOR's approach to implementing filters for block storage devices was to build on this foundation so that arbitrary code implementing filters could provide the backstore to the LIO[5] kernel target code.

Another required element was a method to select remotely the filter or filters to be installed on the data path on another node for a given connection. For this, it was decided to extend another open-source project, targetd[6], designed for a different but related purpose: a deamon accepting a set of commands for storage-related functionality. On this was built MPSTOR's Konnector module, which accepts a new set of commands (sent by the SDS controller) to allow filters to be interposed on the data path.

7.2 Filtered Block Storage Mechanics

Figure 5 shows the components of the Filter framework.

1. SDS REST API providing an interface to the provisioning application.

2. Konnector, the in-band control element on the consumer node. Konnector's API allows control of the termination of storage array volumes and the creation of the service stack and filters.

3. The SDS controller (MPSTOR's Provizo) which sends storage array specific commands to the storage array.

4. This is the terminated "back end" storage volume provisioned from the storage array.

[5]Linux I/O
[6]https://fedorahosted.org/targetd/

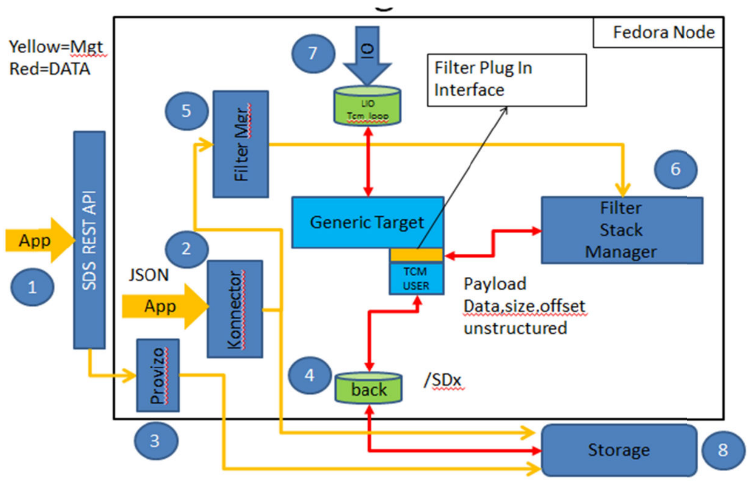


Figure 5: Filter framework mechanics.

5. This is the Filter manager which is responsible for building the service and filter stack on top of the "back end" storage
6. These are the control functions that call the filter entry points with the in-band data from the user to the storage devices.
7. User Application IO (Read/Write data).
8. The SAN storage device.
9. The storage array.

The sequence in this framework is as follows. On a request for block storage received through the REST API, a logical volume is provisioned by the SDS Controller on a suitable storage node with the required spare capacity. At any time thereafter, a request may be made to attach that volume to some consumer node. The SDS Controller sends the necessary commands to the storage array to make that volume available at the consumer. The Konnector daemon running on the consumer node is sent the necessary commands to terminate the connection and responds with a message identifying the local device (`/dev/sdX`) through which the remote unfiltered storage is now available. The SDS controller refers to its database to determine what, if any filters should be installed on the data path, and instructs Konnector to deploy these, which it does through the Filter Manager (see below); Konnector responds with the resulting device address to the SDS controller. Finally, the SDS controller informs the user of the filtered device address: in an OpenStack context, this is processed by an OpenStack driver and is made avabailable by OpenStack Nova in a VM on the consumer node, which now has a new block device. At a later time, the connection may be torn down and subsequently established with another VM on another compute node - or to the same VM, migrated to another node.

In this sequence, the filters work as follows. The filter stack shown in Figure 5, extends the Linux LIO kernel target by means of a user-space daemon (tcmu-runner). The daemon intercepts SCSI CDBs[7] en-route to bound target storage devices and allows them to be processed in user-space before they are committed to the target storage devices. The daemon has an API plug-in mechanism which allows handler modules to be inserted into the device I/O chain. The IOFilterStack module is just such a module and its purpose is to provide run-time linkage and to manag collections of externally defined filter objects. Using this mechanism, multiple filter chains of pre-determined execution sequence can be defined on top of kernel target storage devices which act on the SCSI payload data in the given sequence.

[7]Command Descriptor Blocks

7.3 Capabilities and limitations

The framework is designed from the outset to support a stack of filters, that is, filters can be chained together flexibly in an arbitrary sequence.

A limitation of the approach is that filters are set-up when the connection is established and cease to operate only when it is ended, a filter cannot be removed from or added to an existing connection; instead the block device must be unmounted (if mounted in a file system) at the consumer, and the connection broken and re-established - after which the block device may reappear at a different address (e.g., the volume which was at `/dev/sdy` may reappear at `/dev/sdz`). However, in some contexts, it may be desirable to alter the operation of a filter that is already deployed, for example, to reduce throttling on an existing attachment. In principle, there is no reason this could not be supported: it requires only that the filter code periodically refer to external, modifiable configuration data to modify its behaviour.

Were the limitation that filters are chosen only when the connection is made to be regarded as unduly restrictive, a similar approach might be taken to make it possible to add or remove filters dynamically in an existing data flow: a single metafilter on each connection could delegate to whatever filters happened to be configured for that connection at a particular time. However, such a requirement is not currently anticipated. Instead, it is assumed for now that the filters to be deployed on connecting a volume will be defined as part of the storage policy associated with the volume.

The current framework is also based on a pre-defined set of filters. Ultimately, there is no technical reason why it would not be possible for an administrator to define new filters to be deployed, or to allow languages other than C to be used. This, however, would require further work on defining the interfaces and mechanisms to support it, as well as ways to limit or restrict the possible operations to prevent buggy or poorly-written filters from having an adverse impact on performance or stability, perhaps by a domain-specific language with a restricted instruction set, similar to the way eBPF[8] filters can be written in a restricted version of C [9] [10].

The remainder of this section describes the definition and operation of filters in more detail.

7.4 Filter object definition and interface

The filter objects are built from C source files. There is a skeleton filter project called 'nop' which serves to act as a template for filter development. Within this filter, there are three functions which are run-time linked to the filter manager. The function:

```
void write_xform( void* buf, unsigned long cnt, unsigned long offset)
```

is passed a void pointer to the payload data together with a length argument 'cnt' and an argument 'offset' which is currently not utilised but reserved for future operations that might alter the size of the transformed data. Essentially, this function exists to perform a transformation at its defined filter level on payload write data bound for the device.

The associated read transformation:

```
void read_xform( void* buf, unsigned long cnt, unsigned long offset)
```

with the same prototype definition, is called at the same predefined filter execution level and is designed to act on the read payload data from the device en-route to the initiator. There is an additional helper function: `get_name()` which can be called from the manager. This just returns the name of the filter which is defined as a static string in the filter object. This is just added for debug purposes and is called when the tcmu daemon is executed in debug mode

7.5 Building FilterStack filter objects

Filter objects are linked to the filter stack at run-time. To facilitate run-time linkage they are built as Linux shared object (so) libraries. The filter stack handler looks for these objects in a specific folder (`/usr/lib64/tcmu-filters`). The sample filters have a simple convention, the filter source files are named in accordance with the action they perform. For instance the sample xor filter

[8] extended Berkeley Packet Filter
[9] https://suchakra.wordpress.com/2015/08/12/bpf-internals-ii/
[10] https://github.com/iovisor/bcc

is called xor_filter.c and it's source files are located in `/root/tcmu-runner/filters/xor`. In the `root/tcmu-runner/` directory, there is a simple bash script `build_filters.sh` which will build all the filters according to the existing convention in terms of naming and placement of the source files. The script builds the filter sources and generate the output `.so` files, copies the object to the defined filter object directory.

7.6 Creating filtered storage device objects

The creation and management of these devices is manged by Konnector, however for the purposes of the sandbox VM, vis. creating and experimenting with filter objects, the Linux LIO configuration tool 'targetcli' which is installed on the VM can also be used. For example, creating a filter chain, on top of a device (`/dev/vdb`) of 4GB, which acts on payload write data (and hence read data in the reverse order):

`Initiator>xor>nop>bitrev>/dev/vdb.`

We can use targetcli to create the filter stack and then to build a loopback target device on top of this stack and create a new block device `/dev/sdX` which appears to the system as a native block device but implicitly contains the filter stack manager and filter chain whose background operation is totally transparent to the end user. Running targetcli on a fresh installation with no previously defined devices and typing `ls` will display a configuration something like that in the figure below

To create the filter backed device according to the above definition we would type:

backstores/user:mp_filter_stack/ create name=test cfgstring=>xor>nop>bitrev>/dev/vdb size=4G

we can again type 'ls' to see the device and its relation to the LIO configuration. Now we want to create a loopback fabric based target which we can build on top of the device for export purposes. Assuming that we want to create a fresh target type:

`loopback/ create`

We should get a message – in this example: `Created target naa.50014050870d36d3`

Next we need to create a LUN using the object `test` that we created previously and export it on this target. To this end, we type:

`loopback/naa.50014050870d36d3/luns create /backstores:mp_filter_stack/test`

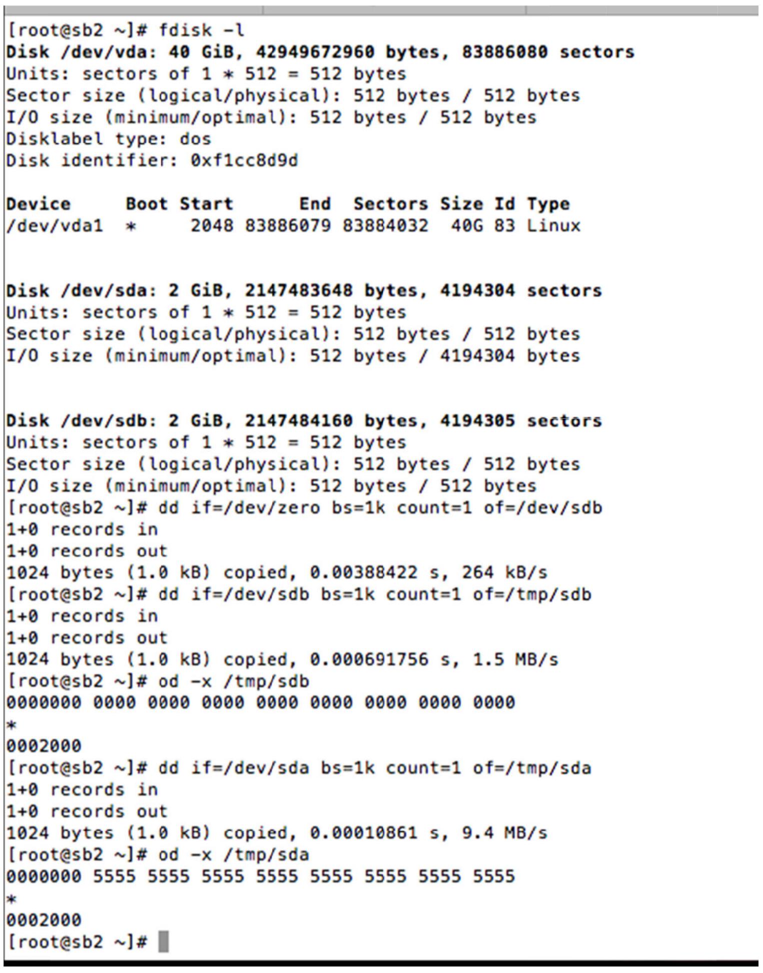
Now we can type 'saveconfig' to save our configuration and 'exit' to quit targetcli. We can now quickly do an `fdisk -l` to see that we have indeed added a new block device which in the case of the example is `/dev/sda`. As previously discussed, we can treat this device as a regular native block device and the filter action is totally implicit and transparent to the end-user.

7.7 Filter daemon operation notes

The tcmu-runner daemon used in the filter framework under normal operating conditions runs as a background process. It is normally located in /usr/bin and is launched by d-bus the first time any service requiring its operation is called. There is another mode of operation which has been built into it primarily for debugging purposes which may be useful for filter development. In order to utilise this operation mode, it is necessary to first kill the background process by finding the process ID via `ps aux| grep tcmu`. Using the process ID related to `/usr/bin/tcmu-runner` you can stop it using the `kill` command. If you then rename `/usr/bin/tcmu-runner` to something sensible for later reference, you can run the local daemon in the tcmu-runner directory with a debug switch by typing (inside the directory): `./tcmu-runner -d` With the daemon running in this mode, it is possible to see some useful debugging output in terms of filter linking and execution. An example of such output is given below.

Please note that daemon operation in this mode can significantly impact on device I/O performance and should strictly be used for debug purposes only. Under normal operation, the daemon should be executed natively from the /usr/bin directory.

Also note that all I/O bound for filtered devices will stall when the daemon is stopped so you should ensure that there are no pending I/Os bound for any managed devices prior to daemon stoppage.



```
[root@sb2 ~]# fdisk -l
Disk /dev/vda: 40 GiB, 42949672960 bytes, 83886080 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0xf1cc8d9d

Device     Boot Start      End  Sectors Size Id Type
/dev/vda1  *     2048 83886079 83884032  40G 83 Linux


Disk /dev/sda: 2 GiB, 2147483648 bytes, 4194304 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 4194304 bytes


Disk /dev/sdb: 2 GiB, 2147484160 bytes, 4194305 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
[root@sb2 ~]# dd if=/dev/zero bs=1k count=1 of=/dev/sdb
1+0 records in
1+0 records out
1024 bytes (1.0 kB) copied, 0.00388422 s, 264 kB/s
[root@sb2 ~]# dd if=/dev/sdb bs=1k count=1 of=/tmp/sdb
1+0 records in
1+0 records out
1024 bytes (1.0 kB) copied, 0.000691756 s, 1.5 MB/s
[root@sb2 ~]# od -x /tmp/sdb
0000000 0000 0000 0000 0000 0000 0000 0000 0000
*
0002000
[root@sb2 ~]# dd if=/dev/sda bs=1k count=1 of=/tmp/sda
1+0 records in
1+0 records out
1024 bytes (1.0 kB) copied, 0.00010861 s, 9.4 MB/s
[root@sb2 ~]# od -x /tmp/sda
0000000 5555 5555 5555 5555 5555 5555 5555 5555
*
0002000
[root@sb2 ~]#
```

Figure 6: Low-level demonstration of filtered block device.

8 Experiments

8.1 Test Cloud

A test cloud, shown in Figure 7, provided the infrastructure for experiments. It comprises the following nodes:

- a dual controller cloud controller
- x4 compute nodes with Fibre channel and Eth fabric support
- SDS controller
- Block storage with Eth and Fibre channel fabric support
- Object storage with Eth fabric support
- VSA (Virtual Storage Array, this allows many virtual storage arrays to be created from x1 physical device, this is useful in testing the REST API with many storage controllers.

The cloud was set up with the help of MPSTOR's Cloud Configurator tool. This technology allows rapid prototyping of the cloud topology so that multiple alternative configurations can be tested quickly.

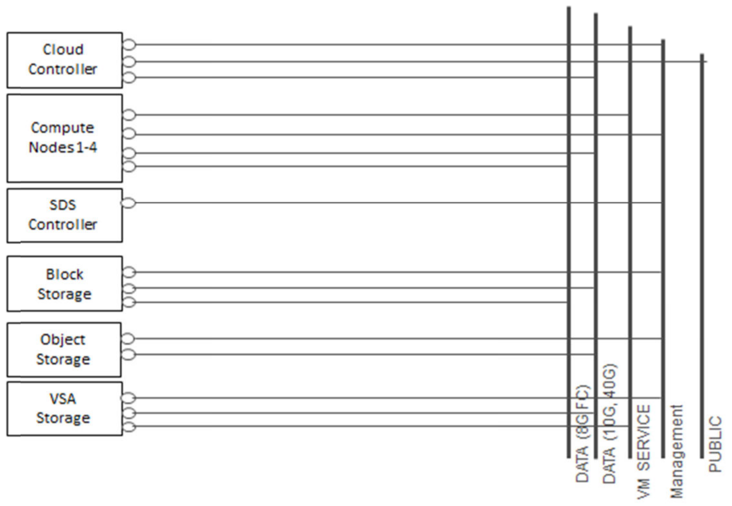


Figure 7: Test cloud configuration.

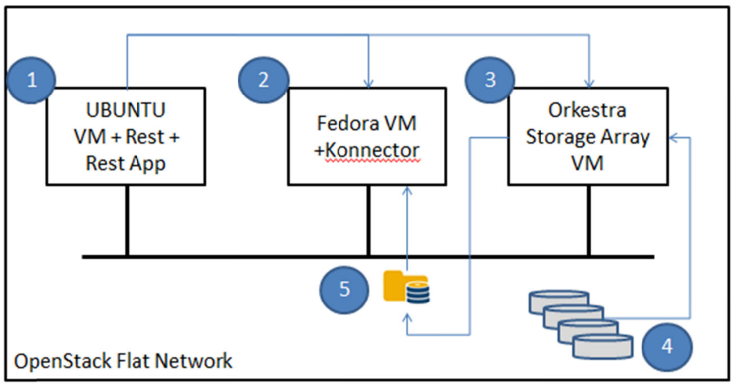


Figure 8: Filter framework development environment.

8.2 Filter Sandbox

A filter framework sandbox testbed has been created on a Fedora virtual machine (a Fedora 22 cloud-base image) which acts as a development platform and demonstration platform. This framework is composed of Konnector which provides a JSON interface for control, a set of user developed filters (dynamically linked shared-object files) and standard Linux components. The environment is shown in Figure 8 and consists of an MPSTOR Openstack distribution packaged with the following VMS:

- Ubuntu VM with the REST API Application
- Fedora VM with Konnector and Filter Management
- Orkestra Virtual Storage Array

The entire environment is virtualized, reducing the number of physical machines required for the testbed. This environment allows the development and demonstration of the REST API (1 in Figure 8) provisioning storage from the Virtual Storage Array (3) which has attached virtual disks on which RAIDs are built, attaching the storage array volume (5) to the Fedora VM (2).

We use a sandbox VM in which the Konnector service runs, and an Orkestra SDS node which provisions storage from an Orkestra SAM node (in a flat network configuration, i.e., where the SDS

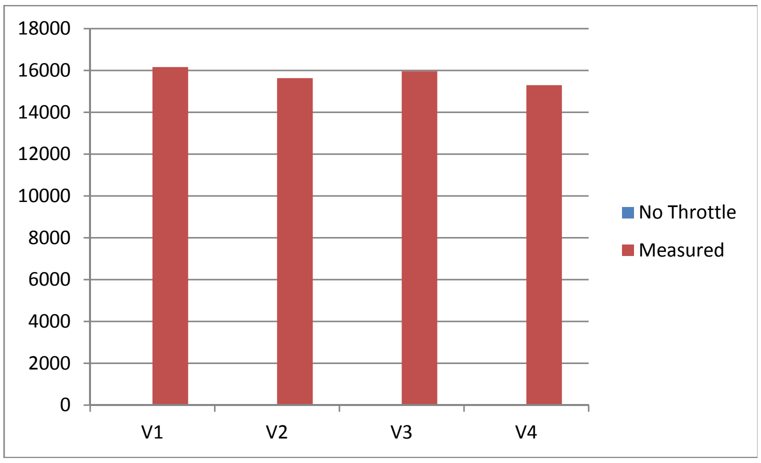


Figure 9: Unthrottled bandwidth, KB/s.

node is accessible from the VM, the SAM node may be the same as the SDS node). Note that a volume could equally be exported to the VM from another, non-Orkestra system.

8.3 Filter testing results

The filter framework was tested for the following

1. NOP filter, this filter simply intercepted the IOs between an application and the backend storage device, it simply logged the IOs to demonstrate the filtering could be achieved in an IOFLOW.
2. XOR filter, this filter implemented the following function data=data XOR key, the input flow was clear data, the data sent to the storage device was XORed with key. The data read from disk was obfuscated by the XOR function, however the data to the application was clear data.
3. Bitrev filter, this filter reversed the bit order of the data.
4. The filters were tested singly but also cascaded, the effect was to apply each function to the data flow during writes and the reverse order during reads. Data read through the application was correct, data stored on the backend device has the XOR and BITREV operations performed on it.

Figure 6 shows a dump of the data from the filter device and the backend storage, with the written user data on the disk being the result of a filter XORing each byte with 0x55.

8.4 Throttling test results

At time of writing, filters to implement bandwidth or IO throttling had not been written. As a base case for comparison (allowing the impact of any such filter itself on performance to be assessed), throttling on In order to test throttling, four volumes were created on the consumer node and FIO[11] was used as test tool to run a MB/s and an IO/s test.

Figure 9 shows the available bandwidth MB/s for the four volumes of approximately 64MB/s aggregate and approximately 16MB/s per volume. In a multi-tenant cloud configuration it is important that the data flow can be controlled at either the MB/s or IO/s by the top level SDS application. A non-controllable block storage means very low predictability of when tasks will complete. The right amount of IO/s and MB/s must be allocated to the storage application and in particular low priority tasks must be throttled back in preference to higher priority tasks.

[11]www.freecode.com/projects/fio

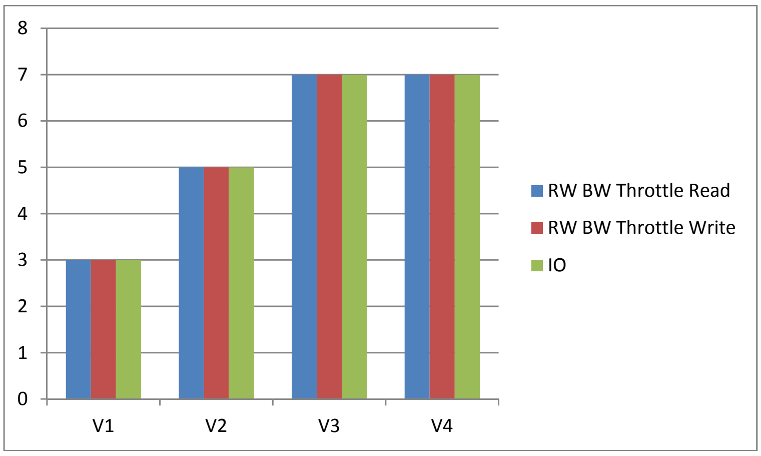


Figure 10: Result of throttling to 3, 5, 7, 7 MB/s (reads and writes).

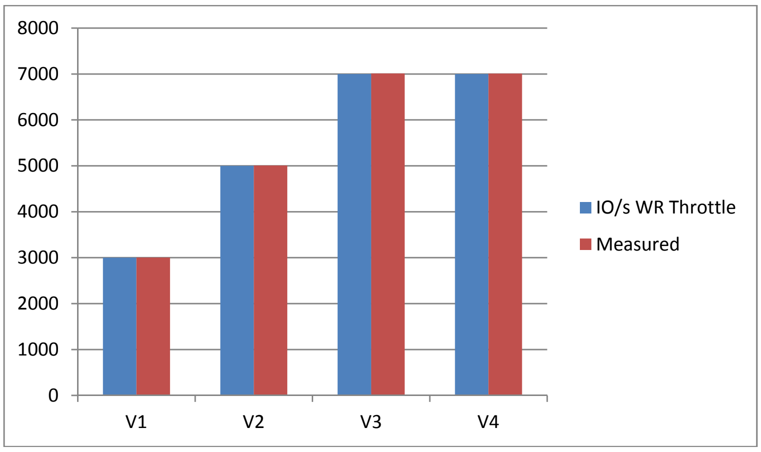


Figure 11: Result of throttling write operations to 3K, 5K, 7K, 7K IO/s.

In Figure 10 the four volumes have been throttled to 3,5,7,7 MB/s for both reads and writes. The measured results are practically 100% fit to the expected results. These throttle levels were chosen so that neither the network or storage were the bottleneck.

In Figure 11 the four volumes have been throttled to 3K, 5K, 7K and 7K IO/s for writes. The measured results are practically 100% fit to the expected results. These throttle levels were chosen so that neither the network or storage were the bottleneck.

In Figure 12 the four volumes have been throttled to 3K, 5K, 7K and 7K IO/s for reads. The measured results are lower than expected. Work is ongoing to try and understand what is happening as the throttle levels were chosen so that neither the network or storage were the bottleneck.

8.5 Block Storage versus Object storage

A test VM was used to run a standard analytics application.

The test was run using two backend storage configurations:

1. Block storage over 1G Ethernet
2. Object storage over 1G Ethernet

- The Block storage completed in 8 minutes
- The Object storage in 30 minutes.

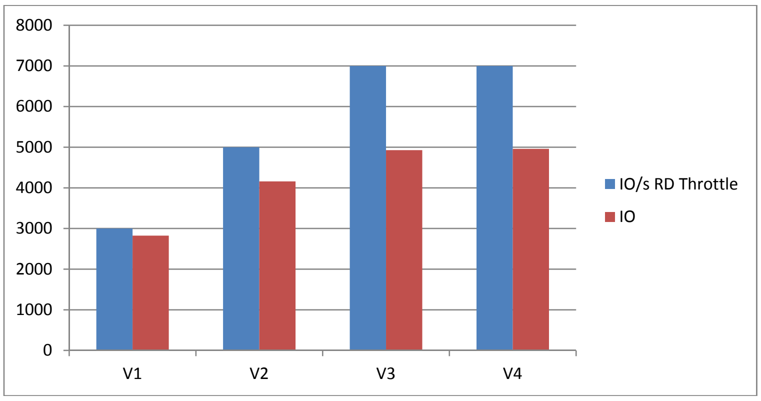


Figure 12: Result of throttling read operations to 3K, 5K, 7K, 7K IO/s.

Both tests were run on identical hardware over identical networks and identical host machines. The results indicate what is intuitively reasonable: though block storage is more complex to manage, it is more efficient for most workloads.

9 IOStack Block Storage SDS API

9.1 Design and Technical Background

This section describes the rationale for the implementated architecture and certain decisions with regard to the API.

9.1.1 Modular Design Based on an "Object API"

Key to the design is to present the existing SDS controller (Orkestra SDS) functionality using a general object model congruent with the REST API's model of linked resources subject to POST (create), GET, PATCH (modify), and DELETE: that is, as objects, each of which has both attributes and references to related objects, which can be created, read, written, and deleted. (The CRUD[12] acronym is applicable, but is usually understood to mean actions limited to updating a representation, i.e., a database, and so excluding the associated necessary actions such as communicating with the storage hardware to create a volume.) The module providing this interface is 'objectapi'. Put simply, the objectapi provides a kind of REST API as a set of function calls rather than HTTP methods; the REST API module's interaction with the Application is entirely through the objectapi interface. In this separation of concerns, the REST API module is responsible for the mechanics of serving client HTTP requests, the URLs for objects and collections of objects, the data format which is used (e.g., Collection+JSON, hm-json, hal+json), managing versioning of the REST API, and so on. A change in any of these should have no impact on objectapi or the other application code; conversely, the REST API code should be insulated from any change in, e.g., the database used by the application or the commands that need to be sent to the storage nodes to accomplish some action. Figure **??** shows the objectapi interface.

This functional division has advantages for testing: the functionality of the REST API can be substantially tested at the underlying objectapi interface, with the test code for the REST API code concentrating only on its responsibilities. Thus a change at the REST API, e.g., from using JSON to using XML, should be limited also in its impact on the test code.

The REST API module could conceivably present a resource structure that differs structurally from the object model it operates on, for example, by presenting two related objects as a single resource, but allowing for this is not a goal of the architecture, and it might not necessarily be easy or clean. Note also that while the REST API code is not constrained to use the same object identifiers as the objectapi, if an ID is not at least based on the objectapi ID (for example, the objectapi ID encrypted with a session key) then the REST API code may need to make additional calls to the

[12]CRUD: Create, Read, Update, Delete

```
+----------------------------------------+
| REST API code                          |
|                                        |
+----------------------------------------+

    +           /\
    |  calls    |  linked objects
    v           +

+----------------------------------------+
| objectapi                              |
| * create_object(type, **attributes)    |
| * get_objects(type, **filters)         |
| * modify_object(type, id, **attribute) |
| * delete_object(type, id)              |
+----------------------------------------+
```

Figure 13: The objectapi interface.

```
{"_type": "widget", "id": 48,
 "_refs": {"replaces": "widget", "replaced_by": "widget",
        "used_in": "doohickey", "components": "yokeybob"},
 "components": ["0xf12", "0xe11"],
 "replaces": null,
 "replaced_by": 67,
 "used_in": [],
 "name": "thingumajig",
 "created": "2015-07-10T12:14:33Z",
}
```

Figure 14: A Linked Typed Object.

objectapi where otherwise it would have made only one. Thus the objectapi module is best regarded as determining what the objects are and how they are identified: a widget with ID 41 might appear as a resource at /widgets/41 or /gadgets/41, or perhaps even at /users/17/things/41, but probably not at /user/17/things/5.

Because the data format for interaction with the object model is general and may be reusable in other contexts, it is given a name: Linked Typed Objects (LTO). It is JSON-compatible (so, e.g., times are strings), but objects may be passed as equivalent (lists of) Python dicts.

In LTO, an object is like any other object, with the addition of two special attributes: "_type" and (optionally) "_refs". An example is shown in Figure 14.

The "_type" attribute is mandatory, as is "id". The "_type" might also be called the object's class and is a string. For objects of the same "_type", the "id" is always an integer or always a string, which should be treated as opaque. The combination of "_type" and "id" uniquely identify an object. An object without a "_refs" object is equivalent to an object with an empty "_refs" object (). For each attribute of the parent object that is to be interpreted as the "id" of another object, or a list of such identifiers, the "_refs" object maps that attribute name to the "_type" to associate with the identifiers. In the above example, the object of type "widget" with ID 48 references two components of type "yokeybob" with IDs "0xf12" and "0xe47". A link attribute may be empty, in which case its value is null if it is an attribute of a kind that links to at most one other object, or an empty list ([]) if it is of a kind that could link to a list of objects.

9.1.2 Third-party library

The third-party library code to create the REST API was chosen after evaluating a number of alternatives, taking into account the following considerations. Heavyweight full-stack frameworks such as Django[13] appear to be overkill for a REST API. Therefore the technologies examined in more de-

[13] http://www.djangoproject.com

tail were those taking a more minimalist approach, including Flask[14], Morepath[15], CherryPy[16] and Bottle[17], together with various plugins or tools designed to be used with them. All of these were quite appealing. CherryPy has the advantage of including a favourably-reviewed flexible, multi-threaded server of its own which has been used even where the application itself has been developed using another framework. However, writing code for CherryPy to service REST API requests is less straightforward than with other frameworks, though this can be improved somewhat with the use of an additional tool (routes). While CherryPy is the oldest, Morepath is the most recent of the above mentioned and appears to benefit most from new ideas, though it may not be fully mature. Ultimately, Bottle was chosen: Although it includes some unneeded features, such as templates, it has the advantage of being a single source file with no dependencies. There is no multi-threaded server built-in, but it was found possible to fill this gap by adapting the wsgiref server which is part of Python.

9.1.3 Design

The design based on the Object API model and the use of the library Bottle is shown in Figure 15.

9.1.4 REST API: General principles

Despite the fact that the thesis of a single person, Roy Fielding, is universally referenced for the idea of REST, there is no universally accepted authority or consensus on best practices for REST APIs, with divergent opinions, for example, on whether to terminate URLs with a suffix indicating the protocol (typically XML or JSON), what HTTP error codes to use in particular cases, whether to use HTTP PATCH, and so on, down to more minor issues such as whether to use plural nouns in path components (/volumes instead of /volume).

One of original basic principles is "Hypertext As The Engine Of Application State" (HATEOS), which recommends the inclusion of descriptive links in each response so that, given an initial URL, someone could discover whatever it was possible to do through the API. Where clients base their actions only on the actions discoverable from the base URL (which itself involves a bit more effort, and involves additional requests to navigate to the required attributes), following this principle may make it easier to change the API without affecting clients.

However, there is no agreed way to do this: for JSON, alternatives include HAL[18], JSON Hyper-Schema[19], Siren[20], JSON-LD[21], Collection+JSON[22], Mason[23], etc. — and although such choices might be avoided by choosing XHTML instead, JSON is generally used as a simpler protocol. It is a principle that is widely disregarded in industry, with discoverability being replaced by full API specifications. Nevertheless, even disregarding the ideal of discoverability, it was decided to support, at least initially, one of the competing hypermedia formats, bearing in mind that it is a public interface, as well as the possible benefits of uniformity and the labor saved by following an existing thought-out and already-documented convention for organizing data.

Therefore, for the SDS REST API, HAL+JSON is supported. Although it is simpler than most alternatives, experience has shown that the specification is not as tight nor the guidance as clear as it might be. Ultimately, a JSON-based API was developed and the HAL interface may be deprecated.

9.1.5 Hierarchical Organization of Resources

It seems natural to envision entities arranged in a hierarchy with the resource URLs representing them reflecting this, e.g.,

[14]http://flask.pocoo.org/
[15]http://morepath.readthedocs.org/en/latest/
[16]http://cherrypy.org/
[17]http://bottlepy.org/
[18]http://stateless.co/hal_specification.html
[19]http://json-schema.org/latest/json-schema-hypermedia.html
[20]https://github.com/kevinswiber/siren
[21]https://www.w3.org/TR/json-ld/
[22]http://amundsen.com/media-types/collection/
[23]https://github.com/JornWildt/Mason

```
+--------------------------+
| Server                   |         /  one instance of each
|  .start() / .terminate() |         \  object on the left
+--------------------------+
                                        objects on the right are
     +                                  instantiated per request
     | has on .thread                             \/
     v

+---------------------------------+
| ThreadingWSGIServer             |
| wsgiref.simple_server.WSGIServer |
|  .set_app(app) / .get_app()     |
| BaseHTTPServer.HTTPServer       |      calls
| SocketServer.TCPServer          |   +---------+
| SocketServer.BaseServer         |             |
|  .RequestHandlerClass           |             |
+---------------------------------+             v

     +                    +-----------------------------------------+
     | has                | RequestHandler                          |
     | .application       | wsgiref.simple_server.WSGIRequestHandler |
     |                    | BaseHTTPServer.BaseHTTPRequestHandler   |
     |                    | SocketServer.StreamRequestHandler       |
     |                    | SocketServer.BaseRequestHandler         |
     |                    +-----------------------------------------+
     |
     v                                          +
                                                |    calls
+--------------------+                          v
| Middleware         |
|                    |         +------------------------------------+
| +----------------+ |   calls  | wsgiref.simple_server.ServerHandler |
| | Api            | |         | wsgiref.handlers.SimpleHandler     |
| | bottle.Bottle  | |  <-----+ | wsgiref.handlers.BaseHandler       |
| +----------------+ |         |  .run(app)                         |
+--------------------+         +------------------------------------+

          +
          |  calls
          v

    +------------+
    | objectapi  |
    +------------+
```

Figure 15: REST API code structure.

/nodes/X/raids/Y/volumes/Z

There are disadvantages, however, to adopting such an approach. (1) What seems natural now might break later. For example, the URL above already does not reflect the fact that volumes may be created across more than one RAID (although Orkestra SDS currently creates volumes on only one RAID). (2) More than one hierarchy is possible, e.g., a volume may be created on a disk rather than a RAID, so that another URL pattern would need to be created for such volumes (/nodes/X/disks/P/volumes/Q). (3) Some relationships resist organization into any single hierarchy, e.g., a volume is in a storage group, but might not be, and is also in a RAID; a disk is obviously on a node but may be an added storage disk corresponding to a volume on another node; an IDA volume is on a RAID on one node but is really composed of multiple volumes on other nodes, etc. (4) Some possible hierarchies may not be permanent during operation. If a volume moved between the controllers of a dual- controller system then would we really want to the URL at which it was located to change from /nodes/X/controllers/M/volumes/T to /nodes/X/controllers/N/volumes/T? (5) a client might wish to skip part of any given hierarchy, e.g., to get all volumes on a node (/nodes/X/volumes/) rather than query each individual collection of volumes (i.e., GET /nodes/X/raids, GET /nodes/X/disks, and then GET /nodes/X/raids/Y/volumes and GET /nodes/X/disks/P/volumes for each raid Y and disk P). (6) If in the pattern /widgets/A/yokeybobs/B/thingumajigs/C the identifier C is unique across all thingumajigs in any case (which should be easy to arrange) then why use the longer URL with redundant information - and should the server be checking that thingamajig C is actually in (or under) yokeybob B and widget A?

There is a counter-argument that it might sometimes be convenient to see the location of an object directly in its URL. However, it will still be possible to determine at least its immediate parent (or immediate parents, where it could have been arranged in more than one hierarchy) from its attributes.

Multiple collection resources which include the same object may be useful, e.g., /raids/Y/volumes, /nodes/X/volumes, as a limited way for a client to retrieve a particular subset of objects, but it is not necessary that the individual objects returned in such collections "live" at those addresses, i.e., the volumes returned by /nodes/X/volumes can be /volumes/A, /volumes/B, etc., or links to them.

Therefore the general approach adopted is that the canonical location of a resource is /<resources>/<id>, even if lists of the same objects are available elsewhere.

9.2 Version

For all requests, the API version number must be provided by the client in one of two ways:

(a) by including a HTTP header 'API-Version', e.g., 'API-Version: 0'; or

(b) by prefixing the URL, e.g., /v0/volumes.

Only a major version number X is required, in which case X.0 is implied. The server includes a minor version in its response, which may be equal to or greater than the minor version specified in the request: minor version increments retain existing attributes to avoid breaking older clients.

The version number header is omitted from the examples (as are most other headers).

9.3 HTTP status codes

The individual sections below do not list all possible values of the HTTP status code in the response, only the "normal" case. In other cases, there will be a "4xx" response for a user error, e.g., "403" if the specified resource does not exist, or a "5xx" response in case of a server error.

9.4 Media type

The default Content-Type is "application/json". HAL (Hypertext Application Language) [https://tools.ietf.org/h kelly-json-hal-07] is also supported, but this support may be removed: for a documented API, the benefits it confers over a non-hyperlinked format are not compelling.

9.5 Errors

When a request results in an error, an attempt is made to return useful information in the body according to the draft RFC http://tools.ietf.org/html/draft-nottingham-http-problem-07 "Problem Details for HTTP APIs". For example:

```
    # curl -X POST -H API-Version:0 $BASE_URL/policies/ -d'{"tier": "Business"}'
    {"status": 400, "detail": "Must specify a value for 'name'."}
```

9.6 Endpoints

9.6.1 /nodes

The storage nodes (storage arrays) known to the SDS server. A storage node may be a member of a storage group or several such groups.

```
Example:

    GET /nodes

Response:

    HTTP/1.0 200 OK

    [
        {
            "groups": [
                7,
                8
            ],
            "id": 1
        },
        {
            "groups": [
                7,
                10
            ],
            "id": 2
        }
    ]
```

```
Example:

    GET /nodes/2

Response:

    HTTP/1.0 200 OK

    {
        "groups": [
            7,
            10
        ],
        "id": 2
    }
```

9.6.2 /controllers

A storage node controller. In the case of a dual-controller storage node, two controllers will reference the same node.

```
Example:

    GET /controllers

Response:

    HTTP/1.0 200 OK

    [
        {
            "hostname": "10.0.15.20",
            "id": 1,
            "node": 1
        },
        {
            "hostname": "10.0.15.24",
            "id": 2,
            "node": 2
        }
    ]

Example:

    GET /controllers/2

Response:

    HTTP/1.0 200 OK

    {
        "hostname": "10.0.15.24",
        "id": 2,
        "node": 2
    }
```

9.6.3 /disks

The disks on the storage nodes known to the SDS server.

```
Example:

    GET /disks

Response:

    HTTP/1.0 200 OK

    [
        {
            "id": 1,
            "node": 1,
            "size_gib": 464
        },
        {
            "id": 1894,
            "node": 2,
            "size_gib": 4
        }
    ]

Example:

    GET /disks/1894

Response:

    HTTP/1.0 200 OK

    {
        "id": 1894,
        "node": 2,
        "size_gib": 4
    }
```

9.6.4 /volumes

The collection of all volumes created through this API. This includes any snapshot volumes. The parameters that can be specified when creating (POST) a volume are:

Name	Type	Description
size_gib	integer	Size in gigabytes of the volume to be created. Not required for snapshot volumes.
name	string	Optional: if no name is provided then a random unique name will be assigned.
group	integer	Optional. ID of storage group determining volume attributes and location.
snapshot_of	integer	Optional. If specified then the volume created will be a snapshot of the volume with the specified ID.

```
Example:

POST /volumes
    {"name": "test-vol", "size_gib": 5}

Response:

    HTTP/1.0 201 Created
    Location: /volumes/719

    {
        "created_at": "2015-09-29T13:03:12Z",
        "group": null,
        "id": 719,
        "iqn": "iqn.2004-04.com.mpstor:test-vol.i707f6c9c09baa3896706",
        "name": "test-vol",
        "node": 1,
        "portals": [
            "10.0.15.20:3260"
        ],
        "proxy_for": null,
        "serial": "0xb11c1b4d3a200172",
        "size_gib": 5,
        "snapshot_of": null,
        "snapshots": [],
        "status": null,
        "vendor": "MPStor",
        "wwns": []
    }
```

```
Example:

    GET /volumes

Response:

    HTTP/1.0 200 OK

    [
        {
            "created_at": "2015-09-25T08:05:53Z",
            "group": 8,
            "id": 718,
            "iqn": "iqn.2004-04.com.mpstor:rest-vol.i3e058770eb5b30dd8cac",
            "name": "rest-vol",
            "node": 1,
            "portals": [
                "10.0.15.20:3260"
            ],
            "proxy_for": null,
            "serial": "0xb11c1b4d3a200168",
            "size_gib": 5,
            "snapshot_of": null,
            "snapshots": [],
            "status": "OK",
            "vendor": "MPStor",
            "wwns": []
        },
        {
            "created_at": "2015-09-29T13:03:12Z",
            "group": null,
            "id": 719,
            "iqn": "iqn.2004-04.com.mpstor:test-vol.i707f6c9c09baa3896706",
            "name": "test-vol",
            "node": 1,
            "portals": [
                "10.0.15.20:3260"
            ],
            "proxy_for": null,
            "serial": "0xb11c1b4d3a200172",
            "size_gib": 5,
            "snapshot_of": null,
            "snapshots": [],
            "status": "OK",
            "vendor": "MPStor",
            "wwns": []
        }
    ]

Example:

    GET /volumes/719

Response:

    HTTP/1.0 200 OK

    {
        "created_at": "2015-09-29T13:03:12Z",
        "group": null,
        "id": 719,
        "iqn": "iqn.2004-04.com.mpstor:test-vol.i707f6c9c09baa3896706",
        "name": "test-vol",
        "node": 1,
        "portals": [
            "10.0.15.20:3260"
        ],
        "proxy_for": null,
        "serial": "0xb11c1b4d3a200172",
        "size_gib": 5,
        "snapshot_of": null,
        "snapshots": [],
        "status": "OK",
        "vendor": "MPStor",
        "wwns": []
    }

Example:

    DELETE /volumes/719

Response:

    HTTP/1.0 200 OK
```

9.6.5 /volumes/<ID>/snapshots

The collection of snapshots of a particular logical volume. All snapshots are volumes. A POST to /volumes/<ID>/snapshots is equivalent to POST /volumes with snapshot_of set to <ID>. Provided the snapshot is a snapshot of the specified parent volume, /volumes/<PARENT-ID>/snapshots/<ID> is equivalent to /volumes/<ID>, which should be preferred.

```
Example:

POST /volumes/719/snapshots
    {"name": "snap1-4"}

Response:

    HTTP/1.0 201 Created
    Location: /volumes/721

    {
        "created_at": "2015-09-29T13:40:35Z",
        "group": null,
        "id": 721,
        "iqn": "iqn.2004-04.com.mpstor:snap1-4.ia9d49363f927ae8ec6aa-1",
        "name": "snap1-4",
        "node": 1,
        "portals": [
            "10.0.15.20:3260"
        ],
        "proxy_for": null,
        "serial": "0xb11c1b4d3a200179",
        "size_gib": 5,
        "snapshot_of": 719,
        "snapshots": [],
        "status": null,
        "vendor": "MPStor",
        "wwns": []
    }

Example:

    GET /volumes/719/snapshots

Response:

    HTTP/1.0 200 OK
```

```
[
    {
        "created_at": "2015-09-29T13:40:35Z",
        "group": null,
        "id": 721,
        "iqn": "iqn.2004-04.com.mpstor:snap1-4.ia9d49363f927ae8ec6aa-1",
        "name": "snap1-4",
        "node": 1,
        "portals": [
            "10.0.15.20:3260"
        ],
        "proxy_for": null,
        "serial": "0xb11c1b4d3a200179",
        "size_gib": 5,
        "snapshot_of": 719,
        "snapshots": [],
        "status": "OK",
        "vendor": "MPStor",
        "wwns": []
    }
]
```

9.6.6 /volumes/<ID>/connections/

The connections that have been created for a particular volume. POST /volumes/<ID>/connections/ is equivalent to POST /connections with the value of the volume parameter set to <ID>. Provided the connection with ID <ID> is a connection of the specified volume, /volumes/<VOL-ID>/connections/<ID> is equivalent to /connections/<ID>, which should be preferred.

```
Example:

    POST /volumes/719/connections/

Response:

    HTTP/1.0 201 Created
    Location: /connections/30

    {
        "consumer": null,
        "created_at": "2015-09-29T15:27:52Z",
        "disk": null,
        "id": 30,
        "initiator": 12,
        "proxy": null,
        "volume": 719
    }

Example:

    GET /volumes/719/connections

Response:

    HTTP/1.0 200 OK

    [
        {
            "consumer": null,
            "created_at": "2015-09-29T15:27:52Z",
            "disk": null,
            "id": 30,
            "initiator": 12,
            "proxy": null,
            "volume": 719
        }
    ]
```

9.6.7 /initiators

An initiator represents a potential consumer of a volume. The parameters that can be specified when creating an initiator are:

Name	Type	Description
name	string	Optional: if no name is provided then a random unique name will be assigned.
iqn	string	Required for an iSCSI initiator. Example: "iqn.1993-08.org.debian:01:34dcc84aa8b".
username	string	For iSCSI initiators, if authentication by CHAP is required.
password	string	For iSCSI initiators, if authentication by CHAP is required.
wwn	string	For a Fibre Channel initiator, the World Wide Name of the consumer port as a hex string. Example: "abfedc0987654321".
sas_port	string	For SAS, the port ID of the SAS initiator port as a hex string. Example: "fedcba0987654321".

```
Example:

    POST /initiators
    {"wwn":"abfedc0987654321"}

Response:

    HTTP/1.0 201 Created
    Location: /initiators/21

    {
        "created_at": "2015-09-29T14:19:46Z",
        "id": 21,
        "iqn": null,
        "name": "uYFe6L4yUGl504at",
        "password": null,
        "sas_port": null,
        "username": null,
        "wwn": "0xabfedc0987654321"
    }

Example:

    GET /initiators

Response:

    HTTP/1.0 200 OK

    [
        {
            "created_at": "2015-09-16T15:06:12Z",
            "id": 12,
            "iqn": "iqn.1993-08.org.debian:01:34dcc84aa8b",
            "name": "l6MNf3CC3CS0qriZ",
            "password": null,
            "sas_port": null,
            "username": null,
            "wwn": null
        },
        {
            "created_at": "2015-09-29T14:19:46Z",
            "id": 21,
            "iqn": null,
            "name": "uYFe6L4yUGl504at",
            "password": null,
            "sas_port": null,
            "username": null,
            "wwn": "0xabfedc0987654321"
        }
    ]

Example:

    GET /initiators/12
```

```
Response:

    HTTP/1.0 200 OK

    {
        "created_at": "2015-09-16T15:06:12Z",
        "id": 12,
        "iqn": "iqn.1993-08.org.debian:01:34dcc84aa8b",
        "name": "l6MNf3CC3CS0qriZ",
        "password": null,
        "sas_port": null,
        "username": null,
        "wwn": null
    }

Example:

    DELETE /initiators/21

Response:

    HTTP/1.0 200 OK
```

9.6.8 /connections

A connection corresponds either to (1) permission granted to a particular initiator to read or write to a particular volume, or (2) the creation on a given consumer node of a proxy disk and, optionally, a proxy volume on that proxy disk. In the case of (2), reads or writes locally to the proxy are routed to the original volume on whatever node the volume was created.

The parameters that can be specified when creating a connection are:

Name	Type	Description
volume	integer	Required. The ID of the volume to which a connection is required.
initiator	integer	The ID of an initiator. Either an initiator or a (consumer) node must be specified.
consumer	integer	The ID of a node on which a proxy disk/volume is to be created.
proxy	null	If a consumer node is specified then the connection's proxy attribute may be specified as null, in which case no proxy volume will be created. Otherwise, the server will assign to this attribute the ID of the proxy volume it creates.

```
Example:

    POST /connections
    {"volume": 719, "consumer": 2}

Response:

    {
        "consumer": 2,
        "created_at": "2015-09-30T08:04:41Z",
        "disk": 1894,
        "id": 34,
        "initiator": null,
        "proxy": 722,
        "volume": 719
    }

Example:

    GET /connections

Response:

    HTTP/1.0 200 OK
    [
        {
            "consumer": null,
            "created_at": "2015-09-25T08:55:35Z",
            "disk": null,
            "id": 29,
            "initiator": 12,
            "proxy": null,
            "volume": 718
        },
        {
            "consumer": 2,
            "created_at": "2015-09-30T08:04:41Z",
            "disk": 1894,
            "id": 34,
            "initiator": null,
            "proxy": 722,
            "volume": 719
        }
    ]

Example:

    GET /connections/29

Response:

    HTTP/1.0 200 OK

    {
        "consumer": null,
        "created_at": "2015-09-25T08:55:35Z",
        "disk": null,
        "id": 29,
        "initiator": 12,
        "proxy": null,
        "volume": 718
    }

Example:

    DELETE /connections/34

Response:

    HTTP/1.0 200 OK
```

9.6.9 /policies

A storage policy may be associated with storage groups and determines certain characteristics of volumes created in those groups.

The parameters that can be specified when creating a policy are:

Name	Type	Description
name	string	Required. A unique name for the storage policy.
san_name	string	Storage Area Network Name: if specified, determines the ports on which new volumes are exported.
tier	string	Media tier. Required. New volumes will be created only on resources (normally RAIDs) of the specified tier. Possible tiers include "Mission Critical", "Business", and "Archive".

```
Example:

    POST /policies
    {"name": "gold", "san_name":"10Gdata", "tier": "Mission Critical"}

Response:
    HTTP/1.0 201 Created
    Location: /policies/8

    {
        "created_at": "2015-09-30T08:41:18Z",
        "id": 8,
        "name": "gold",
        "san_name": "10Gdata",
        "tier": "Mission Critical"
    }
```

Example:

```
    GET /policies
```

Response:

```
    HTTP/1.0 200 OK

    [
        {
            "created_at": "2015-09-21T11:03:30Z",
            "id": 6,
            "name": "bronze",
            "san_name": "1Gdata",
            "tier": "Archive"
        },
        {
            "created_at": "2015-09-30T08:41:18Z",
            "id": 8,
            "name": "gold",
            "san_name": "10Gdata",
            "tier": "Mission Critical"
        }
    ]
```

Example:

```
    GET /policies/8
```

Response:

```
    HTTP/1.0 200 OK

    {
        "created_at": "2015-09-30T08:41:18Z",
        "id": 8,
        "name": "gold",
        "san_name": "10Gdata",
        "tier": "Mission Critical"
    }
```

Example:

```
    PATCH /policies/6
    {"tier":"Business"}
```

Response:

```
    HTTP/1.0 200 OK

    {
        "created_at": "2015-09-21T11:03:30Z",
        "id": 6,
        "name": "jack",
        "san_name": null,
        "tier": "Business"
    }
```

Example:

```
    DELETE /policies/8
```

Response:

```
    HTTP/1.0 200 OK
```

9.6.10 /groups

A volume may be created in a storage group. A storage group determines the nodes on which that volume may be created and, by association with a storage policy, certain characteristics of the volume. Note: to change the nodes in a storage group, use the /groups/<ID>/nodes endpoint.

The parameters that can be specified when creating a group are:

Name	Type	Description
name	string	Required. A unique name for the storage group.
policy	integer	Required. The ID of a storage policy.

```
Example:

    POST /groups
    {"name": "local1..", "policy": 6}

Response:

    HTTP/1.0 201 Created
    Location: /groups/11

    {
        "created_at": "2015-09-30T09:02:28Z",
        "id": 11,
        "name": "local1..",
        "nodes": [],
        "policy": 6
    }

Example:

    GET /groups

Response:

    HTTP/1.0 200 OK

    [
        {
            "created_at": "2015-09-21T12:34:03Z",
            "id": 10,
            "name": "tip..",
            "nodes": [
                2
            ],
            "policy": 6
        },
        {
            "created_at": "2015-09-30T09:02:28Z",
            "id": 11,
            "name": "local1..",
            "nodes": [],
            "policy": 6
        }
    ]

Example:

    GET /groups/11

Response:

    HTTP/1.0 200 OK

    {
        "created_at": "2015-09-30T09:02:28Z",
        "id": 11,
        "name": "local1..",
        "nodes": [],
        "policy": 6
    }

Example:

    PATCH /groups/10
    {"policy": 5}

Response:

    HTTP/1.0 200 OK

    {
        "created_at": "2015-09-21T12:34:03Z",
        "id": 10,
        "name": "tip..",
        "nodes": [
            2
        ],
        "policy": 5
    }

Example:

    DELETE /groups/10

Response:

    HTTP/1.0 200 OK
```

9.6.11 /groups/<ID>/nodes

The storage nodes in a particular group. Provided that node <ID> is in the group, GET /groups/<GROUP-ID>/nodes/<ID> is equivalent to GET /nodes/<ID>, which should be preferred; if the node does not exist or is not in the group, a 404 response issues.

```
Example:

    GET /groups/10/nodes

Response:

    HTTP/1.0 200 OK

    [
        {
            "groups": [
                10
            ],
            "id": 2
        }
    ]

Example:
Add a storage node to a storage group.
(Note: a "201" response will issue even if the node was already in the group.)

    PUT /groups/10/nodes/1

Response:

    HTTP/1.0 201 Created

Example:
Remove a storage node from a storage group.
(This does not delete the storage node.)

    DELETE /groups/10/nodes/1

Response:

    HTTP/1.0 200 OK
```

10 Conclusions and further work

The work to date demonstrates the feasibility of a Software Defined Storage model incorporating filters in the context of block storage, and its value given the greater performance available with block storage compared to object storage. Further work will involve the implementation of more realistic filters, investigating the possibility of allowing user-defined filters as opposed to choosing from a fixed set of filters, and providing access to filters through the REST API, whether as attributes of storage policies or in another way. In the current model, filters act independently for each connection to a block device, but one can also envisage a possible need for co-ordination among filters, for example, where the objective is to limit the aggregate bandwidth provided to a collection of block devices rather than a single volume. Such functionality would add significant complexity and may involve a performance penalty.

References

[1] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker, "Onix: A distributed control platform for large-scale production networks," in Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, OSDI'10, (Berkeley, CA, USA), pp. 1–6, USENIX Association, 2010.

[2] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu, "Ioflow: a software-defined storage architecture," in ACM SOSP'13, pp. 182–196, 2013.

[3] J. Bort, "The inside story of a $1 billion acquisition that caused cisco to divorce its closest partner, emc." http://www.businessinsider.com/inside-the-1-billion-vmware-nicira-buy-2014-10, October 2014.

[4] "Tcm userspace design." https://www.kernel.org/doc/Documentation/target/tcmu-design.txt.

[5] A. Grover, "Lio and the tcmu userspace passthrough: The best of both worlds," in Vault 2015, 2015. http://events.linuxfoundation.org/sites/events/files/slides/tcmu-bobw_0.pdf.