



HORIZON 2020 FRAMEWORK PROGRAMME

IOStack

(H2020-644182)

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

**D3.3 Stable Release and specifications of the SDS framework
for Analytics**

Due date of deliverable: 31/12/2017
Actual submission date: 09/01/2018

Start date of project: 01-01-2015

Duration: 36 months

Summary of the document

Document Type	Deliverable
Dissemination level	Public
State	v1.1
Number of pages	45
WP/Task related to this document	WP3 /T3.2
WP/Task responsible	MPSTOR
Leader	William Oppermann
Technical Manager	Michael Breen (MPSTOR)
Quality Manager	Toni Cortés (BSC)
Author(s)	William Opperman (MPSTOR), David Coffey (MPSTOR), Michael Breen (MPSTOR)
Partner(s) Contributing	MPSTOR, BSC
Document ID	IOStack_D3.3_Public.pdf
Abstract	Final report of of Block storage IOStack prototypes. Complete specification and APIs of the Block storage SDS toolkit. Description and evaluation of KPIs
Keywords	IOStack, Block Storage, Software Defined Block Storage, Konnector.

History of changes

Version	Date	Author	Summary of changes
1.0	05-12-2017	William Oppermann	Initial commit
1.1	23-12-2017	William Oppermann	Update of KPIs and figure labelling

Table of Contents

1	Executive summary	1
2	IOStack: Achievements and Impact	1
2.1	Introduction and Motivation	1
2.2	Goals of IOStack	3
3	KPIs	5
3.1	KPI2 Service creation Dashboard	6
3.2	KPI2 SDS In-line data filters	7
3.3	KPI9 In-line filters	8
3.4	KPI3 Automation of provisioning storage using dashboard	8
3.5	KPI3 Automation of provisioning storage using CLI	9
3.6	KPI10 Filters for data reduction, protection and QoS control	9
3.7	KPI9 SDS Block storage and Data protection	10
3.8	KPI2 Data metrics	10
4	The IOStack Toolkit	11
4.1	Architecture	11
5	Introduction	15
6	SDS for block storage	16
6.1	Features and benefits of the SDS toolkit	18
7	SDS Gateway and Vendor plugin operation	19
7.1	OpenStack Horizon dashboard extensions	21
8	Konnector operation	21
8.1	Filter Functions	22
8.2	Konnector design	24
8.3	Filter Implementation	24
8.4	Filter toolkit	25
8.5	Standard Toolkit Filters	26
9	Advanced Filter description	28
9.1	Output Compress Filter	28
9.2	Compress Cache Filter	29
9.3	Deduplicated Cache Filter	29
9.3.1	Controlled environment description	29
9.4	SDS toolkit functional test results	29
10	Conclusions	31
11	Appendix	34
11.1	SDS controller and REST API overview	34
12	Appendix	40
12.1	Konnector API	40
12.1.1	iSCSI initiator commands	40
12.2	Terminology	43

1 Executive summary

This document describes the components of the IOStack block storage toolkit from several viewpoints, functional, technical, monitoring and test. The technical framework has been detailed in previous reports, the overall architecture including new components will be reviewed in this document. The IOStack block storage toolkit achievements will be detailed and how those achievements were measured using a set of KPIs and how these KPIs relate to data centre use cases, in particular the overall IOStack KPIs;

- KPI2 Simplified Analytics virtualisation
- KPI3 Analytics scheduling
- KPI9 SDS Block Storage
- KPI10 Data Reduction

The test framework used to measure KPIs is composed of the Horizon testbed used in previous phase of the project and the Konnector node test framework. The Konnector node framework is used to test command line tools. These tools built with the SDS API demonstrate how managing complex configurations can be achieved using the IOStack SDS REST API, the Horizon dashboard for service creation and command line tools for automating repetitive provisioning tasks. The final deliverable describes the overall IOStack SDS Block architecture, documentation and the various repositories where code is available.

2 IOStack: Achievements and Impact

2.1 Introduction and Motivation

How can data centers, offering a cloud computing service such as analytics, deal with the complexities of scaling their cloud services platform to many hundreds or thousands of end users or applications while retaining the ability to offer each end user a customised, operationally efficient service. The answer lies in providing the end user of the service with the ability to ‘self serve’ and ‘right size’ his IT resources, through a dashboard, which automates the provisioning of those resources. The challenge, however, is increased if the targeted end users for the service vary from non-expert, who need simple dashboard semantics, to the more technically competent IT administrators, who need to configure more complex IT system.

The answer lies in part with the IOStack paradigm, the Software Defined (SD) component for Openstack, which abstracts the user interface from the physical infrastructure and which automatically translates the end user requests into virtual machines, customised to the end user requirements. The three main components of SD are software defined compute (SDc), networking (SDn) and storage (SDs), each responsible for abstracting the control of its own specific resource from the physical device where it is stored. SDc, previously called ‘server virtualization’, has been used by data centers for many years to increase server densities and to reduce capex and power consumption costs, however SDc on its own cannot easily scale if the networking and storage cannot be virtualized. A number of SDn solutions have recently been introduced which promise to greatly simplify network scaling so that today ‘storage virtualization’, or SDs, is the only significant remaining challenge to solve to make SD in the datacenter a reality.

The Sys Admin and SDS The traditional role of the sys admin IT Engineer was to administer hardware devices through the use of configuration files and management tools with low level access to the managed devices. As the scale of datacenter infrastructure has increased to thousands of managed CPU cores, Petabytes of data storage across multiple disk tiers all linked together by complex networks the scale of the management task has exponentially increased. A new way of managing this scaled out architecture is required that is not IT Manager administration driven but driven by intelligent software. This new paradigm of Software Defined interprets the user “Requirement Semantics” configured in a high level dashboard and implements the device provisioning, creation and

configuration without the need of an IT administrator. The IT Admin User is still required when more complex IT configurations are required that the User Dashboard semantics cannot fulfill, the IT administrator has at his disposal the SDS command line (CLI) tools. These SDS CLI tools give the IT Admin User fine level control that the IOStack Dashboard user does not have.

The impact of Datacentre Scale Before the introduction of server virtualization, the limits to scaling IT infrastructure were dictated by physical constraints such as space and power availability. Server virtualization, along with the introduction of multi-core processors, has facilitated a 100-fold increase in server densities to a point where it's now possible to have more than 5,000 virtual servers in a single 19-inch rack. The ability to create and manage many thousands of virtual servers, has been a key enabler of the growth in cloud computing as it enables many thousands of virtual servers to be quickly created and deployed. Virtualizing only the compute resource and not the network or storage resources, however, becomes increasingly complex as the cloud infrastructure scales and it ultimately becomes a bottleneck to further scalability. Only through the virtualization of all three resources can a scalable 'software defined' cloud infrastructure be created. As the software defined infrastructure grows, its value to the user moves from the virtual resources into the infrastructure management software that manages the automated provisioning of resources for the end user applications. The 'automation' prevents the datacenter administrator becoming the bottleneck to growth and it ensures that the cloud service can be delivered quickly, flexibly and efficiently, increasing responsiveness, asset utilization and operational gross margins. IOStack Block SDS toolkit solves these problems by;

- Virtualizing the physical infrastructure so that the user is not required to know the physical specifics.
- Providing a plugin for Openstack Horizon dashboard allowing automated provisioning
- Providing an SDS rest API for the data center DevOps team to provision storage generally
- Providing a set of tools based on the SDS Rest API that perform standard repetitive provisioning tasks saving time and reducing the potential for configuration errors.

The provisioning dilemma The provisioning dilemma can be best explained by the two options

- "any colour as long as its black" or
- "choose any type, any option, any colour"

The first strategy provides cost benefit over choice and the second provides choice at higher cost. IOStack SDS provides choice and cost efficiency through modelling the user choices, providing tools to create these service options, presenting these service options to the user and then automating the backend processes in the datacenter.

To understand the problem of choice in provisioning, consider fig 1, in this example we provision storage from 1 or 10 storage arrays, create a volume of a defined size or storage bucket, export the volume over iSCSI and attach over iSCSI to 1 of 10 servers. Multiplying the options we see we have about 100 options. This case is a very simple environment and simple tools or an Administrator could manage this.

Consider fig 2, the relative complexity of provisioning storage with more options is much greater. In fig 2 we see the number of Media Tiers, Protection levels and SAN's has increased, this provides greater choice required for various workloads but greater complexity in the provisioning challenge from 100 to 16,000 choices. If we add the option of in-line filters we quickly arrive at 250,000 possible combinations. The ability to provide these combinations do map to real datacenter service offerings, this is the "right sizing" component of the data centre offering. Provisioning this level of customisation "cost effectively" is where the service configuration component of SDS and the automation of the configured service offering is important. The IOSTACK SDS toolkit therefore not only provides new functionality such as inline filters but also allows new service offerings to be created and new

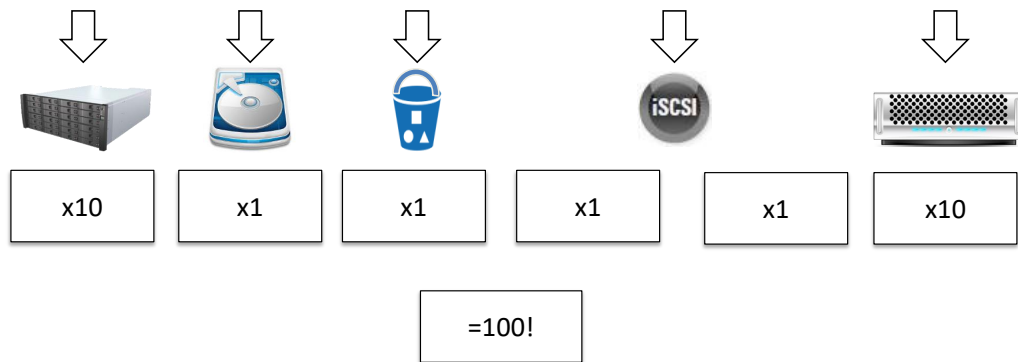


Figure 1: simple provisioning

operation efficiencies in data storage management. IOStack ensures that the cloud service can be delivered quickly, flexibly and efficiently, increasing responsiveness, asset utilization and operational gross margins.

Examples of datacenter needs In fig 3 below we show a number of examples of provisioning use cases and how IOStack SDS caters for each use case. These are a small sample of a very large number of use cases. The examples do serve to illustrate the type of software features that are required from the IOStack SDS controller framework such as:

- High level functional service creation
- Creation of a service and presenting this to a user (ex. Storage volumes with attached filters such as compression or encryption)
- SAN Block volume provisioning
- Provisioning of storage volumes on different SANs and attaching the volumes to the consumer physical machine/virtual machine
- Control of SAN (slow, medium, fast network)
- Control of SAN media Tiers (high IOPS media, fast streaming media, low cost media)

2.2 Goals of IOStack

The goals of block level provisioning within IOStack can therefore be summarised as follows

Create a Software-Defined Storage (SDS) models that can overcome the limitations of system administrator provisioning, those limits being time to provision and reducing the complexity of provisioning storage for analytics. Complexity can reach levels where manual provisioning is beyond

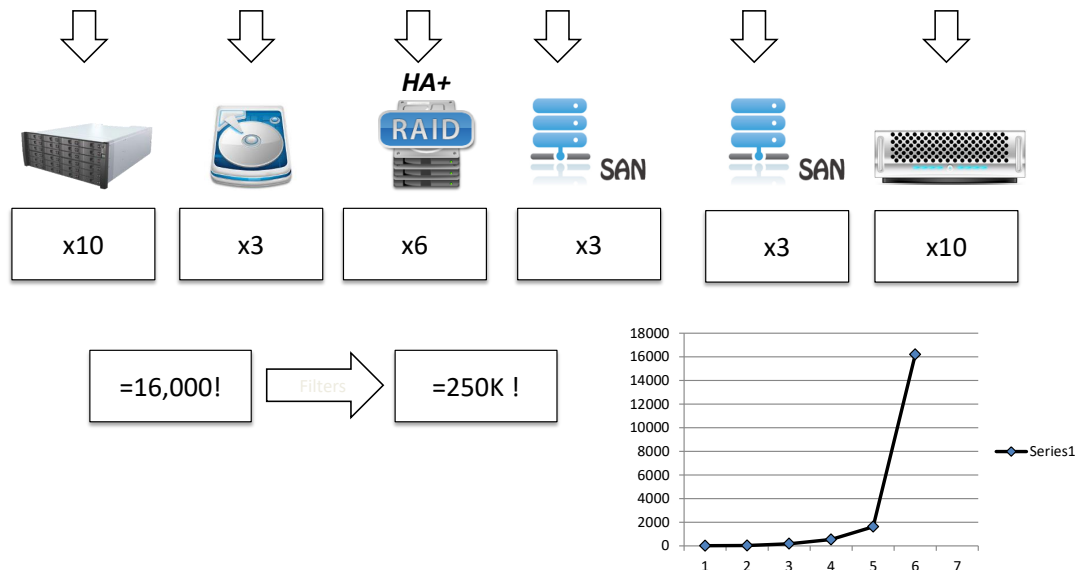


Figure 2: complex provisioning

the ability of a Sys Admin to effectively manage. SDS includes policy-based provisioning, service creation and automation features to greatly facilitate the task of system administrators to manage storage for analytics services. In addition real-time and monitoring tools of the Konnector agent are available for administrators to take decisions based on workload characteristics.

To meet the storage needs of modern analytics, the toolkit targets open the storage system used in analytics workflows with non-anticipated functionalities and optimizations for improving the performance and efficiency of the whole Big Data lifecycle. The Block storage filter mechanism allows a user defined function to be inserted in the data workflow.

The overall goals are achieved through the IOStack toolkit which includes the following capabilities

Service creation The tools for the system administrator to create a high level description of a service offering the IOStack user can use to use to provision storage volumes. This high level offering can specify what type of storage media to use, what SAN to use, what in-line filters to use when creating a storage volume.

OpenStack Plugin Provide a plugin that integrates the OpenStack compute and storage services into the IOStack block level SDS Rest API.

SDS plugin Provide a back-end implementation to a storage array as a base to develop additional storage plugins.

Compute node agent Provide a compute node agent (Konnector) for managing provisioned storage. The Konnector API provides an API used by the SDS layer that attaches storage, builds a filter stack and connects the top of the filter stack to the consumer (usually a virtual machine)

Need	Solution	Use Case	Benefit
1) Fast IOPS storage media for low latency applications	Create a storage pool from an SSD tier of storage	IOPS bound application	Manage a costly resource for those who need it.
2) Fast streaming of Data for Analytics or streaming data	Create a storage pool from a HDD tier of storage over a FAST 100G SAN	BW streaming bound application	Map low cost storage to a fast SAN for those who need it
3) Low cost bulk storage	Create a storage pool from a SATA HDD tier of storage over a 100G SAN	Archiving of data	Map cheap storage to slow SAN for a cost effective solution
4) Fast storage for a cluster of machines	Create a storage pool from an SSD tier but set an IOPS and BW throttle on the storage	IOPS bound cluster using shared storage	Share storage between multiple tenants but avoid any one tenant from hogging resources

Figure 3: use cases

3 KPIs

Figure fig 4 provides a quantitative measure of the achievements of Block SDS IOStack toolkit. Figure fig 4 summarizes Key Performance Indicators (KPIs) and their relation to the project use-cases. These KPIs can be classified as one or more of the following improvements

- Time gained through task automation or visualization
- Improved data reduction, protection or resiliency
- New functionality not possible without the SDS framework

KPI	KPI Description	SDS Feature	Metric	Improvement	Use Case
KPI2	Simplified analytics virtualization	Service creation Dashboard for storage group creation	Time to implement provisioning manually	100% in management cost	Provide storage SLA service to end user
KPI9	SDS Block storage	In-line filters to extend storage array capability	Ability to process in-line data flows	New capability not available without Konnector	Processing in-line data
KPI3	Analytics scheduling	Automation of provisioning storage using Dashboard	Time to provision a storage stack with filters from SDS dashboard	10x in management cost	Improved provisioning time
KPI3	Analytics scheduling	Automation of provisioning storage using CLI	Time to provision a storage stack with filters using CLI tools	100x in management cost	Improved provisioning time
KPI10	Data reduction	Filter Bandwidth and IOPS control of storage, compression etc	% efficiency of storage array performance	50% improvement in asset utilisation	Provisioning capacity and IOPW and BW
KPI9	SDS Block storage	Data protection	Redundancy of data.	100% improvement in data protection	Using filters to keep additional redundant copies of data
KPI2	Simplified analytics virtualization	Data metrics	Insights of data flows and resource usage per compute node	100% improvement in visibility of data usage	Monitoring and optimising asset usage

Figure 4: KPIs

3.1 KPI2 Service creation Dashboard

This KPI is relevant as it allows the IOStack administrator to create storage types (services) with rich features and storage properties. The provisioning of these complex storage services is then automated saving management time and providing new functionality not available from the storage array vendor. The difficulty of provisioning storage can be oversimplified, consider fig 5 below, a volume is created on a storage array and exported to host machine H1 machine 1, the reality is that many complex steps are required to implement this, the details of this task are shown in fig 6 below.

In this view fig 6 we can see many decisions need to be made such as which storage array to use, which media type to use, which SAN the volume is exported on, what is the H1 SAN ID (ex the iSCSI IQN or FC fabric WWN). On the Host side which host bus to connect the storage volume on, discover the storage array and attach it to the consumer. All of these steps require configuration decisions for each volume that is provisioned. The IOStack SDS dashboard allows the user to create a template where these parameters are fixed and labeled as a high level volume type (ex. Gold, Silver, Bronze storage). To provision a volume an IOStack user simply provisions a volume from that group, specifies the size and attaches to an IOStack consumer. The SDS controller receives the request from the IOStack dashboard, parses the request and with its knowledge of the internal architecture implements all of the steps in the provisioning process. These steps are shown in figure fig 7 and in fig 8.

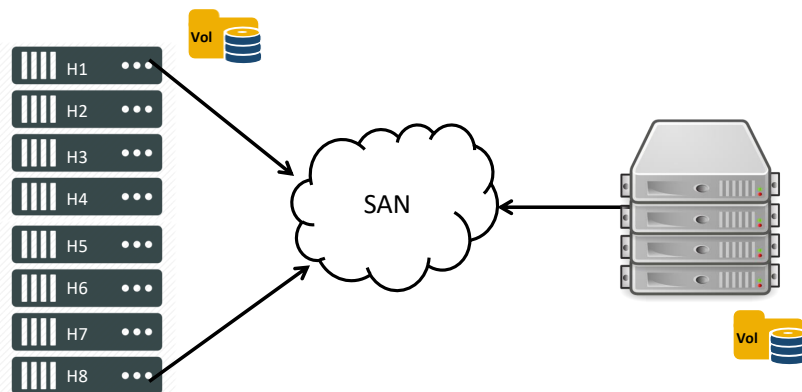


Figure 5: Simplified SAN

3.2 KPI2 SDS In-line data filters

Block storage arrays are normally proprietary closed systems and complex, they are perceived as expensive and are delivered with a fixed set of storage features. Being mission critical systems they are slow to evolve and are inflexible innovation platforms. In contrast to proprietary block storage arrays, standard high performance x86 server nodes used for compute or object storage have an open software architecture and are very flexible platforms for innovation. The IOStack SDS toolkit architecture leverage's the compute node open platform flexibility by moving storage functions (filters) into the compute node, see fig 9. These storage filters provide a means to create innovative block storage functions in-band on the compute node for the data analytics process. New services not available in the storage arrays can be created using the Filter development tool-set. The Filter development tool-set provides sample code to develop new filters. When a new filter is developed it can then be deployed to the compute node where Konnecter is installed. The filter is managed by creating a new volume type using the SDS dashboard service creation tools which references and links to this filter. Once a volume is provisioned and instantiated the filter stack is created on the Konnecter node and attached to the consumer (usually an Analytics virtual machine). The improvement is 100 percent as this feature allows a sys administrator user to create new services and automate the provisioning of the new storage volume type. This is shown in fig 9 below with a Konnecter enabled compute node, storage array volumes (A) can be attached to this node and attached to a virtual machine or compute node service however the properties of the volume or dependent of the feature set of the storage array. Advanced Storage Array feature sets are usually costly add-ons to vendor equipment. The Konnecter strategy allows storage array volumes to be attached (B) and attached to the consumer through a user defined filter stack. The Konnecter framework is a generic method to intercept the data flow and then apply a processing function on the data flow. A user can develop any new in-lone storage function and apply that function to any storage array provisioned volume. This is a new

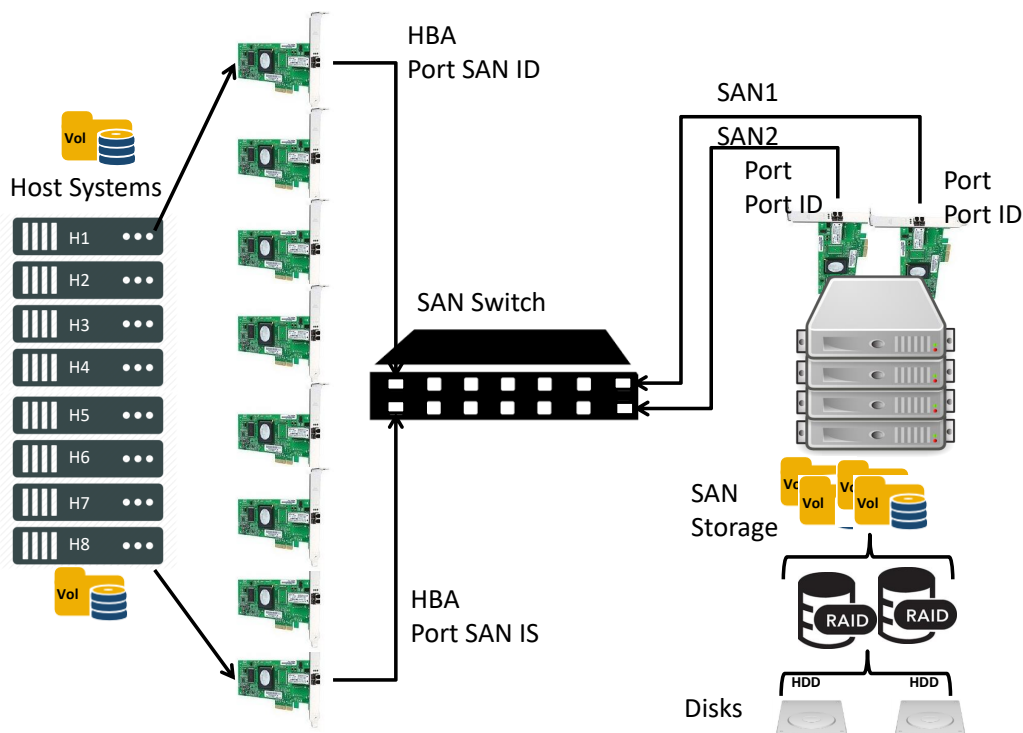


Figure 6: Real SAN

capability that IOStack delivers to its users.

3.3 KPI9 In-line filters

This KPI is delivered to the filter driver developer through the Konnector SDK. The SDK process shown in fig 9 can be described in the following steps

- Download and run the SDK Konnector sandbox VM
- Edit the sample filter for the specific function to be created
- Compile the filter as dynamic linked library (.so) file
- Copy the file to the Konnector nodes that will support this new filter
- Create a storage group that references this new filter
- Provision storage volumes and attach to compute nodes that have been Konnector enabled

3.4 KPI3 Automation of provisioning storage using dashboard

This KPI focuses on the management time gained by automating the provisioning of storage to an IOStack consumer using the Horizon and SDS Dashboard extensions. In this case we consider the provisioning a single volume within a storage group. A manual provisioning of storage compared to automated process is in the order of 10-20 times faster. This can be understood from looking at the steps shown in fig 11 and the equivalent IOStack SDS dashboard command. The workflow is shown below, once the storage group, policy and filter stack definition have been created a user just iterates between 1 and 2 to create and attach additional volumes. The steps in grey are the initial once off setup steps.

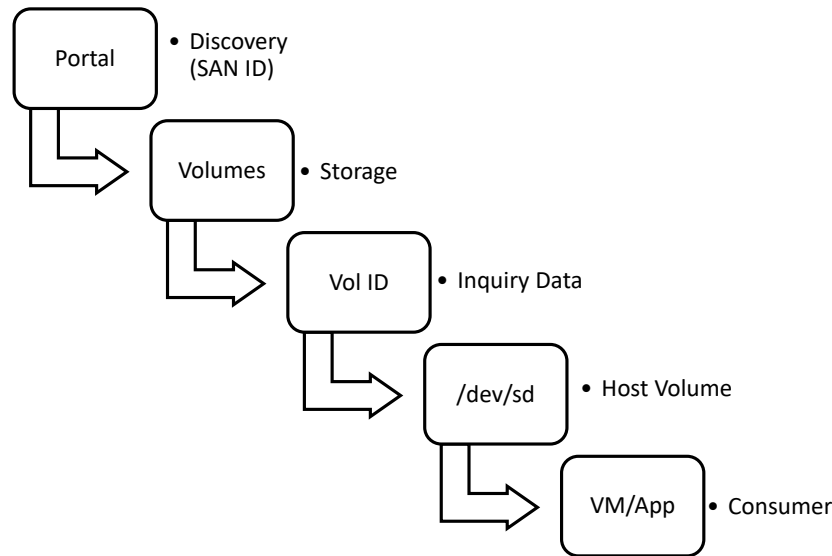


Figure 7: HostStorage

3.5 KPI3 Automation of provisioning storage using CLI

This KPI focuses on the management time gained by automating the provisioning of multiple volumes using template tools. In this test gains are in the order of 100 times faster. CLI tools with templates are very flexible and fast means to provision large numbers of volumes. The use case does require that there be some pattern to the desired configuration otherwise the templated task reduces back to a single volume provisioning request. Templated provisioning works best when there is a pattern to the provisioning requirement, such as described in fig 12 below.

For example, provision the following Konnector nodes in node-list with each with n volumes of name, index, size gigabytes of storage from storage-pool. The storage pool definition will define what type of storage is to be used, the SAN, the media tier and what Filter stack is to be constructed. When N is 1 the gain is not significant but when is 10, 100 etc the gains are considerable and tasks that can take hours are reduced to minutes.

This operation is shown in figure 6 below where the KN-cli tools automatically provisions;

i.e N volumes named V1 to Vn from storage Pool and attaches x1 volume to each Konnector nodes specified in the command line.

3.6 KPI10 Filters for data reduction, protection and QoS control

This KPI focuses on the ability of developed filters to provide a real gain in the used storage. For example compression can reduce the amount of storage required, IOPS/BW filters increase the asset utilization by managing badly behaved tenants and limiting the IOPS/BW (performance QoS) required for a specific workload ensuring fairness of use of the storage asset. Filters are executed on the compute node and therefore compete with the compute node cpu, network and memory resources. Not all filter functions we can imagine are therefore suitable candidates. The Table 8.1a shows filter types that are appropriate given the value provided to the user and the cost in system

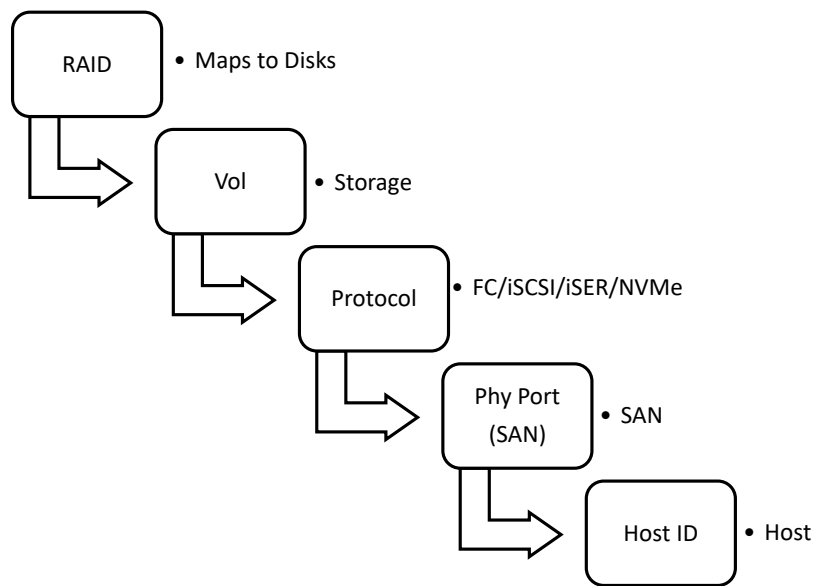


Figure 8: SAN Storage

resources.

3.7 KPI9 SDS Block storage and Data protection

This KPI measures the extra resiliency that specific filters can provide for the application. By improving resiliency the data flowing across the filter is better protected since multiple copies of the data are kept. Data protection is a key feature that concerns each user. Data protection is different from data resiliency. Data resiliency is a storage array property defined by what RAID level is implemented, what is the disk media type, what redundancy is built into the SAN, what controller fault tolerance is provided for at the array level. High resiliency storage is very expensive, low resiliency storage is cheaper. In order to use cost appropriate storage but still provide protected data a data protection filter was created which is capable of doing NxWAY real time copy of the data as it flows through the filter. This allows the user to keep multiple copies of his data on separate storage arrays. The filter provides great benefit since the data is kept in multiple copies in real time and allows for storage array failures.

3.8 KPI2 Data metrics

This KPI measures the benefit and improvement of metrics relating to the data being consumed by the application a wide range of properties. Understanding the balance or trade-off between using compute side filters and the cpu/memory/network cost is important. The Konnector layer is instrumented with probes which measure the resources it uses per volume and pushes these statistics to a Grafana database. Understanding this usage allows sys ops to identify bottlenecks and tailor storage group parameters for example turn on/off filters (compression/encryption) or set thresholds and limits such as IOPS or traffic BW. Examples of meta data visualisation can be seen in fig 13, fig 14, fig 15.

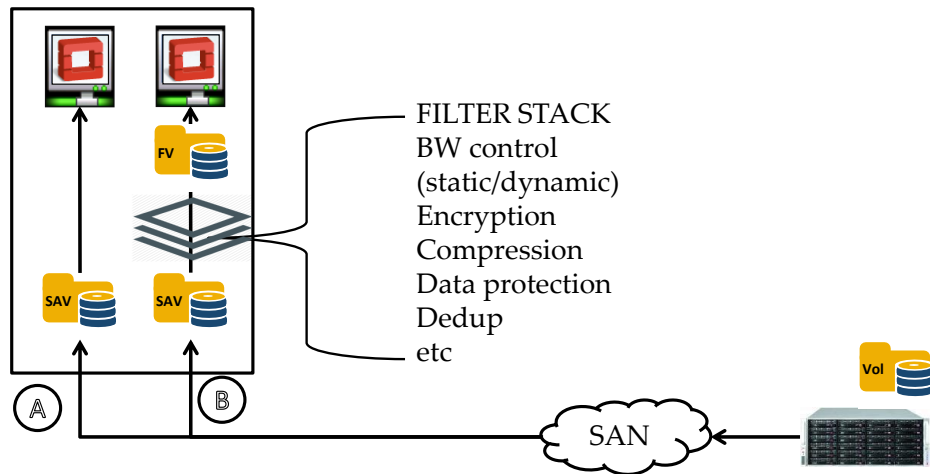


Figure 9: Filter

4 The IOStack Toolkit

4.1 Architecture

The core components of the IOStack block storage management can be grouped into two blocks, the first block being the SDS dashboard and SDS controller and the second block Konnector and its in-line storage filters and command line tools. The block 1 components provide a very high level interface to a cloud user, in our case an OpenStack user. The user sees only a very high level abstraction called a storage group and its associated policy. The storage group associates storage array resources and a policy to each group. The policy metadata of the storage group describes what properties the virtualised storage devices will have such as which Konnector filters will be applied, which physical SAN the volume be visible on and which storage tier will be used. The concept of the policy can be extended to an almost limitless number of metadata properties, each property mapping to a data storage array function or a Konnector filter function. This approach is very useful since the user can provision storage from the datacenter storage array network (SAN) using only a very high level group description of his requirement, the expertise needed to implement all the provisioning steps are executed by the SDS controller and Konnector thus removing all the complexity of provisioning data volumes.

Storage Filters A storage filter can be defined as a general-purpose data transformation applied inline to specific data flows (e.g. data compression, data protection, IO bandwidth differentiation). For any given application using attached storage the IOStack user can therefore enforce transparently on his data flow properties of the data flow and operations on the data flow. For example; controlling the volume bandwidth, compressing the data, protecting the data by writing to multiple locations. These operations can be very complex to implement in a traditional datacenter and require specialist knowledge, yet through simple configurations of SDS policies these operations can be applied simply

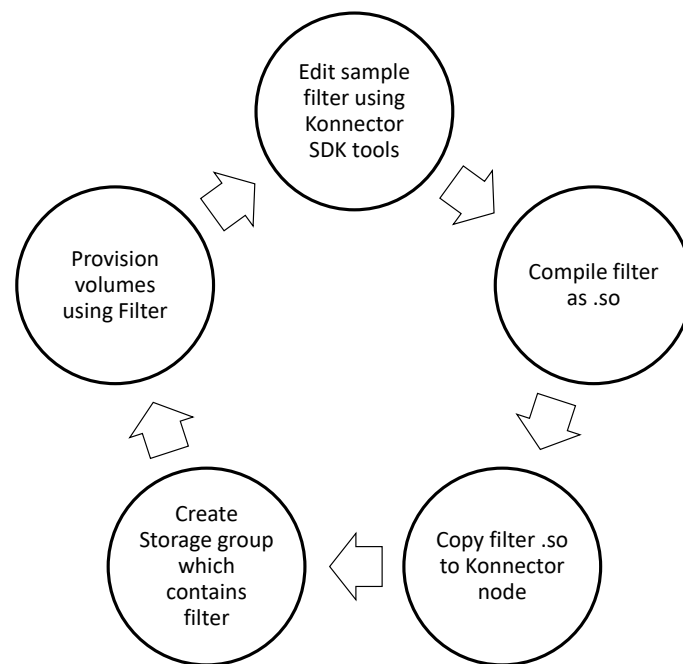


Figure 10: Filter Development Process

to any IOStack data flow.

Monitoring functions IOStack is designed for multi-tenant environments where many virtual machines compete for and use the resources of the cloud. It is therefore important to measure and monitor how each node is performing. Should an application not perform as expected being able to have adequate telemetry of the system under workload is important to diagnosing where the problem is located. The Konnector framework which manages the storage volumes and filters also collects and pushes to a real time DB and graphing tool (Influx DB and Grafana) various storage performance metadata.

Test framework The test framework consists of the Arctur tested and a virtual cloud node which can be used to develop new filters and test them.

Testing Filters Testing filters is twofold, firstly does the filter perform the correct operation and secondly does it operate and perform correct in a multi-tenant configuration. Performance is measured using the Grafana dashboards developed for IOStack allowing the user to view individual volumes, nodes and the collective SDS and provisioned real time behaviour.

The exploitation of data analytics has many challenges. One of these challenges is managing data storage back-ends for data analytics workloads. The traditional approach in building Big Data systems is to physically merge compute and storage on the same hardware node, this approach provides excellent coupling and performance between compute and storage but is very inflexible compared to a virtualised approach. The IOStack method is to use virtualisation, in particular the use of the open source OpenStack framework with extensions to allow easy menu driven deployment of data analytics jobs. This virtualisation of the analytics framework for various workloads shown in fig 16 creates a problem in how to provide storage for these workload clusters. Data analytics workloads can have many constraints, such as the amount of working storage required, the amount of compute required and the time to complete a given job. Meeting these constraints requires provisioning of the correct

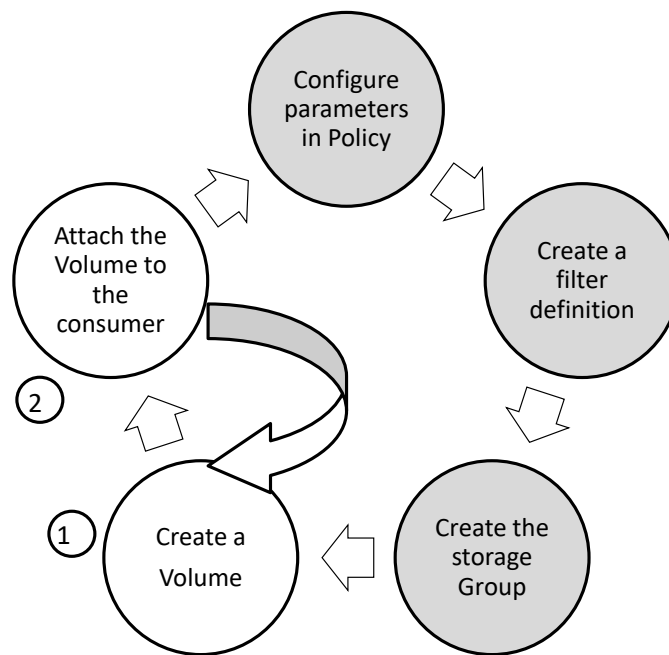


Figure 11: Dashboard Provisioning Process

storage capacity at a given performance level. The IOStack project seeks to solve this problem with an SDS toolkit. The SDS toolkit is a set of stand alone storage tools as well as an integration with OpenStack to allow easy automated provisioning of data storage capacity at defined performance levels. Data analytic workloads can use three underlying storage types;

- block storage
- file storage
- object storage

Block and File storage are characterised as high cost per terabyte solutions whilst object storage is lower cost per terabyte. Block and File solutions have higher performance compared to object storage when measured in IO per second (IOPS) and megabytes per second of bandwidth (BWPS). Normally the higher IOPS of block solutions means the IOPS cost is lower for block storage than IOPS cost for object storage. This may seem counter intuitive as there are several open source object storage solutions however their IOPS performance is low when amortized over the hardware cost.

There is no easy rule in choosing block storage over object over file storage, the choice depends on the workloads and the performance, redundancy and resiliency that is required by the user. What is important is that the data-centre administrator has a choice of storage back ends so that he can choose and use the most appropriate storage for his workload. By choice we mean block or object storage, hard disk or solid state disk devices, choice of fabric and protocol, such as Fibre Channel or 1G, 10G or 100G Ethernet and what processing should be applied to his storage volumes.

The SDS toolkit developed for WP3.2 provides an IOStack administrator the capability to create user defined storage services for specific analytics workloads. The SDS toolkit allows a user using a self service dashboard to provision block storage attach it to compute nodes, create an in-band

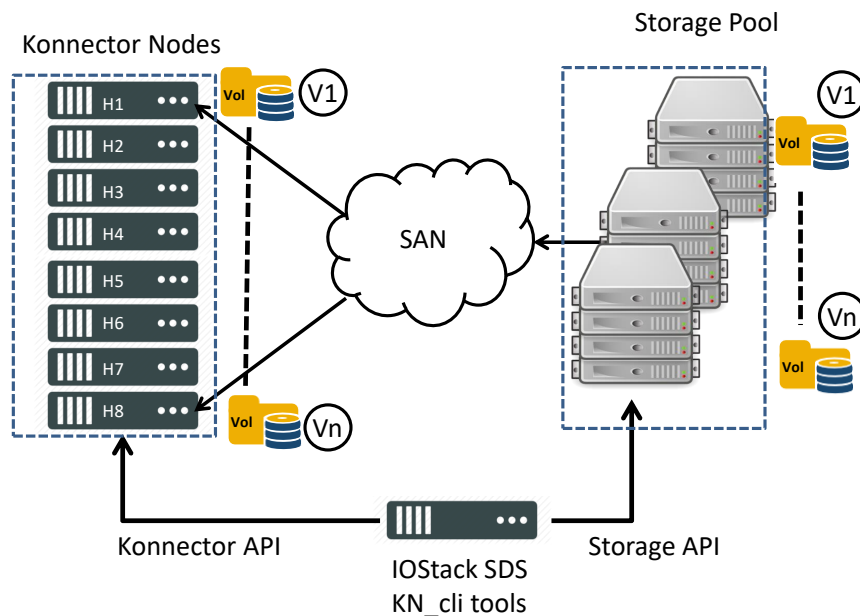


Figure 12: Multi Volume Provisioning

stack of storage processing operations (known as storage filter functions) on top of a storage array volume and insert these filter functions between the storage consumer, in this case an OpenStack virtual machine and the storage array volume.

Block storage arrays are normally proprietary closed systems and complex, they are perceived as expensive and are delivered with a fixed set of storage features. Being mission critical systems they are slow to evolve and are inflexible innovation platforms. In contrast to proprietary block storage arrays, standard high performance x86 server nodes used for compute or object storage have an open software architecture and are very flexible platforms for innovation.

The IOStack SDS toolkit architecture leverage's the compute node open platform flexibility by moving storage functions (filters) into the compute node. These storage filters provide a means to create innovative block storage functions in-band on the compute node for the data analytics process.

The SDS toolkit has been implemented in the Arctur data center and has been successfully used to demonstrate automated block storage service provisioning as well as the development, test and deployment of in-band filter processing functions on data flows. The SDS toolkit is a working prototype with a list of planned enhancements.

This report presents the D3.2 work package (SDS tool kit) which includes a description of the block storage environment, a functional description of the SDS tool kit, the results of testing the SDS toolkit and several implementations of block storage filter functions.

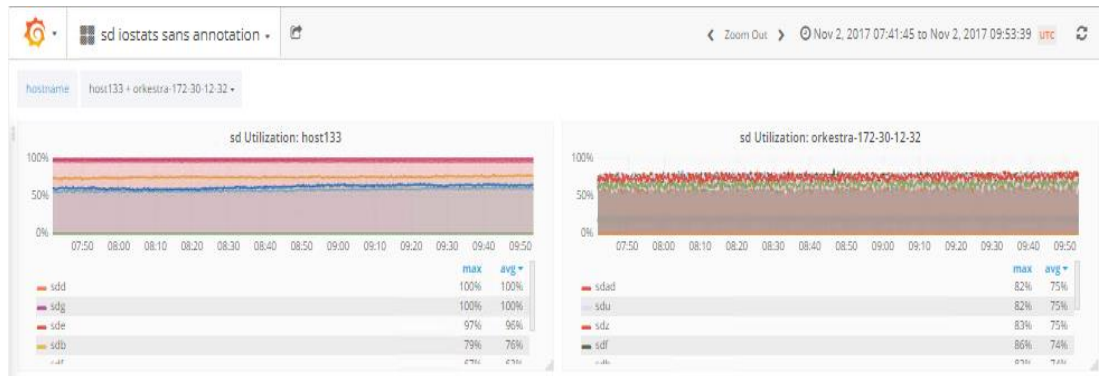


Figure 13: Volume Usage

5 Introduction

A trend in data centres is the constant growth of data and the processing of this data using data analytics to extract financial value. Storing and processing of data creates a provisioning challenge for the analytics user defining and running a data analytics job. The user when configuring his job is required to choose how many virtual machines he will use, how many cores and how much memory he will use as well as how much storage capacity and storage performance required. When the user is not the data centre administrator this is impossible without virtualised infrastructure using self service dashboards. Provisioning the resources required for the user should be automated without any intervention of the data centre administrator.

In IOStack provisioning of these resources is done using a high level self-service dashboard. The SDS toolkit provides the means for the user to choose at a high level storage capacity from a storage group and the means for the data centre administrator to create these storage groups with policies which define and enforce storage performance properties.

Provisioning requests from OpenStack are forwarded to the SDS Gateway which automates the provisioning of storage according to a configured policy so that the user gets the appropriate storage for his workload.

Policy based provisioning allows the SDS tools to automate provisioning with very fine grained storage properties, properties which are specifically required for a specific user workload. In Fig 24 we see how this fine grained provisioning is implemented. A storage group encapsulates which storage array devices should be used, the SAN type (ex. 40, 10, 1G eth or Fibre Channel 16G or 8G), media tier (ex. HDD, SSD) and which filter functions on the compute node should be inserted in-band in the data flow between the Virtual machine and the storage node. A chapter is provided for those who unfamiliar with storage terminology.

OpenStack is a general purpose cloud management software which can run data analytic work-

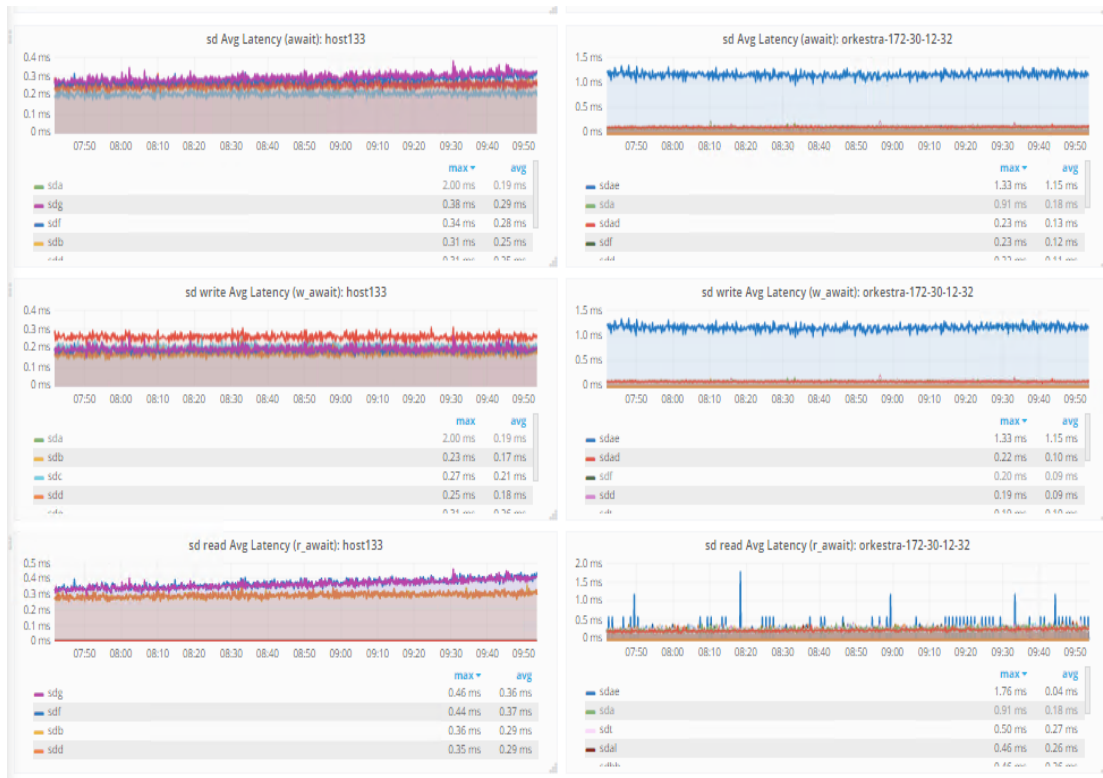


Figure 14: Volume Usage

loads. OpenStack has a pluggable and extensible architecture, including extensions for data analytics.

The SDS toolkit manages block storage arrays in OpenStack allowing a user choice of different block storage media tiers, fabrics and protocols and importantly a means to extend block storage functionality by providing an in-band data filter stack on an OpenStack compute node. A filter stack is a layer of software on a compute node that is in-band between the storage array volume and the consumer on the compute node of the storage volume. In the data analytic environment the consumer is a Virtual machine (VM) running some stage of the data analytic process. A filter stack can provide functionality to improve the data analytic function such as encryption, compression, de-duplication or any other real time in-band data processing function on the data flow between the consumer (VM) and the storage array. One use case for the consortium partner Idiada was analysed and a filter implemented for this use case.

6 SDS for block storage

The SDS toolkit has three main components;

- An extension to the OpenStack management dashboard to allow SDS specific configurations and operations
- An SDS gateway which receives storage provisioning requests from the OpenStack dashboard and processes these requests
- A compute based module called Konnector which implements the storage filter framework. The storage filter framework is independent of the filter functions. Filter functions are software functions inserted in the data flow between the consumer such as virtual machine and the storage array volume. The filter framework is agnostic to filter implementation as long as the filter implements a set of standard entry points. A set of filters can be instantiated on demand



Figure 15: Volume Usage

by the filter framework as filters are implemented as Linux dynamic linked libraries files (.so files).

The Konnector project is stored in a public git repository <https://github.com/MPSTOR/Konnector>. The project website can be found at <https://mpstor.github.io/Konnector/>.

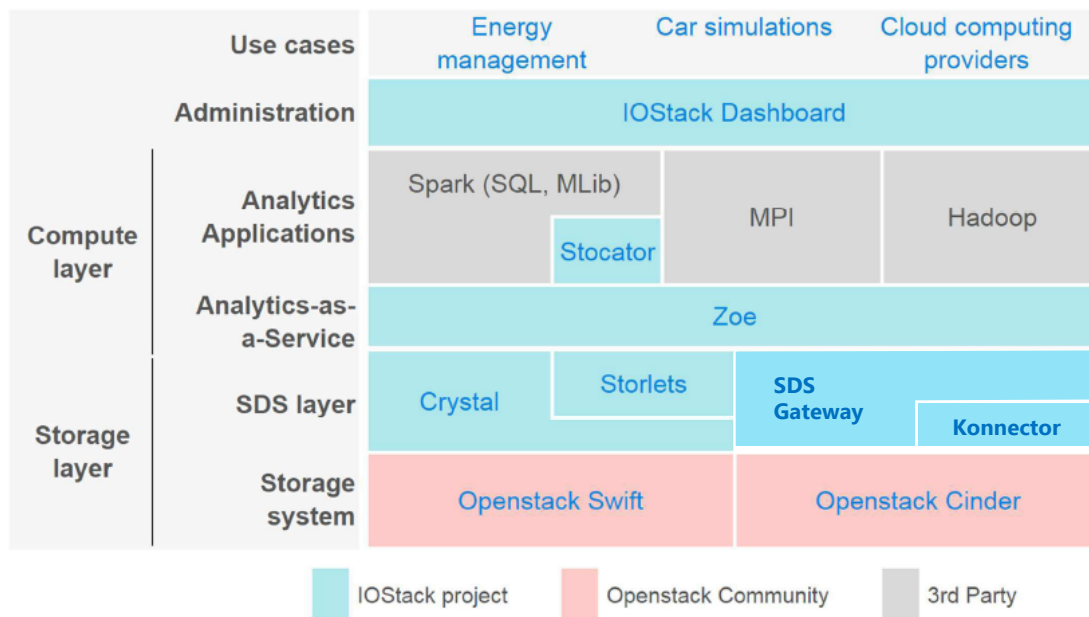
The SDS toolkit in an OpenStack environment, shown in fig 17 allows a user to provision block storage through the openstack Horizon user dashboard according to a user defined policy (1), attach it to compute node (3), create a set of filters based on the user policy on top of the storage array volume (2) and attach a filter volume (4) to the consumer, in this case an OpenStack virtual machine.

In this deliverable, we also spent efforts validating the SDS framework for block storage against our KPIs. In particular, our validation methodology comprised of two sets of tests:

- functional testing of the SDS extensions to the management tools of OpenStack, in particular the Horizon user dashboard
- performance testing of the filter framework

Functional testing of the SDS toolkit within an OpenStack environment allowed the creation of storage groups with user defined policies. Policies provided the user with a means to configure the filter stack configuration per storage group. Volumes were provisioned from storage nodes and were successfully attached to compute nodes across a wide range of filter stacks. A filter stack is a set of inline storage operators inserted in the data flow between the storage provider (the storage Array) and the storage consumer ex. a virtual machine on a compute node.

Performance testing allowed the measurement of the overhead of the filter framework and the performance of individual filters. The overhead of the filter framework was measured by comparing a number of performance metrics at the top and bottom of the filter stack. Measuring performance at



46

Figure 16: Iostack Big Data components

the bottom of the filter stack measures the raw storage array performance. Measuring performance at the top of the filter stack measures the loss or extra overhead of the inline filters.

6.1 Features and benefits of the SDS toolkit

The SDS toolkit provides a number of benefits in managing storage arrays and storage volumes with the filters;

SDS abstraction allows user choice of virtualised storage resources The SDS extension for the Horizon dashboard allows a user to choose not only capacity but also the storage type. A user can therefore choose whichever storage type that best suits his workload, an example could be a Gold Volume type which uses storage arrays with Solid State media tier, over a 10G SAN with a compression filter on the consumer node.

SDS abstraction allows data centre service creation through the creation of storage groups, data tiers and policies The SDS toolkit allows the datacentre Administrator to create specific storage services for particular workloads. These storage services map to storage groups, media tiers and policies over a range of fabrics. By combining filters within the storage group different services can be created, for example layering compression and encryption filters can be presented as a particular service within the datacentre.

Better management of shared storage arrays A key metric in purchasing storage is the cost per IOPS (IO Per Second) of the storage. This metric is increasingly important as data centres transition to All Flash storage arrays. All Flash storage arrays provide high performance in terms of IOPS however they are expensive. It is therefore important to provision not just the amount of storage in terms of Giga/Tera bytes of storage but also the IOPS and MBPS (Megabytes/sec) of each volume provisioned by the user. The filter framework allows the administrator to set a policy for the IOPS and MBPS for each volume created in a storage group. The user can then select a volume of X Gigabytes at an

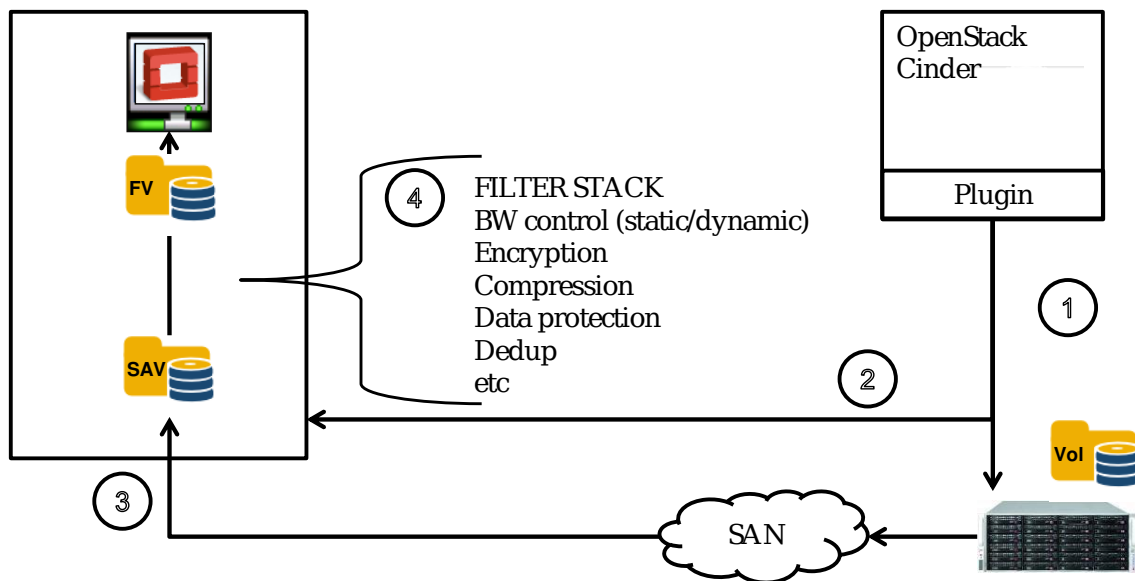


Figure 17: SDS Operation Overview

IOPS=100 or 1000 or 10000 depending on whether the storage array can provision this level of IOPS. A typical all flash array can provide up to 1Million 8K Random IOPS. In it therefore critical that the user has the tools to provision the IOPS and MBPS that his workload requires. In a multi-tenant environment a low priority workload may well use more IOPS and MBPS than the user intends for this workload, this "bad tenant" can therefore starve a workload that does require a specific amount of IOPS and MBPS for the workload to run within a specific amount of time. The SDS toolkit for OpenStack provisions storage from a storage group, each group has a policy that defines which set of storage arrays the storage volumes are provisioned and also the filters used on the consumer node. These filters can set the IOPS or MBPS of the provisioned volume allowing the user fine grain control of the MBPS and IOPS used by each workload.

Predictable behaviour In a multi-tenant environment by allocating not only the amount of storage but also the IOPS and MBPS of the storage for each workload leads to predictable behaviour. Testing has shown that the available IOPS of a storage can be arbitrarily carved up between different workloads. If the IOPS and MBPS of the storage are known this leads to predictable behaviour irrespective of what parallel workloads are being processed.

Lower management costs The SDS toolkit extension for the OpenStack dashboard allows the user to choose a volume type, this volume type is tagged to the compute group within which is configured a set of storage arrays, storage tiers, fabrics and consumer node filter functions. When a volume is created within this storage group the SDS Gateway virtualises and automates for the user all the operations of choosing which storage array, fabric and media tier to use. This greatly simplifies the process of provisioning storage. The second step of attaching a storage array volume and creating on the consumer node a filter stack is also a fully automated process. These complex provisioning tasks reduce the cost of provisioning managed storage by removing the need of the user to know anything about the datacentre internals and simply use the services created by the the datacentre administrator.

7 SDS Gateway and Vendor plugin operation

The SDS Toolkit has been implemented in the configuration shown in fig 18. The configuration is composed of the following components;

- (1) OpenStack Horizon plugins for Block Storage using the REST API plugin providing a Block storage SDS extension to OpenStack

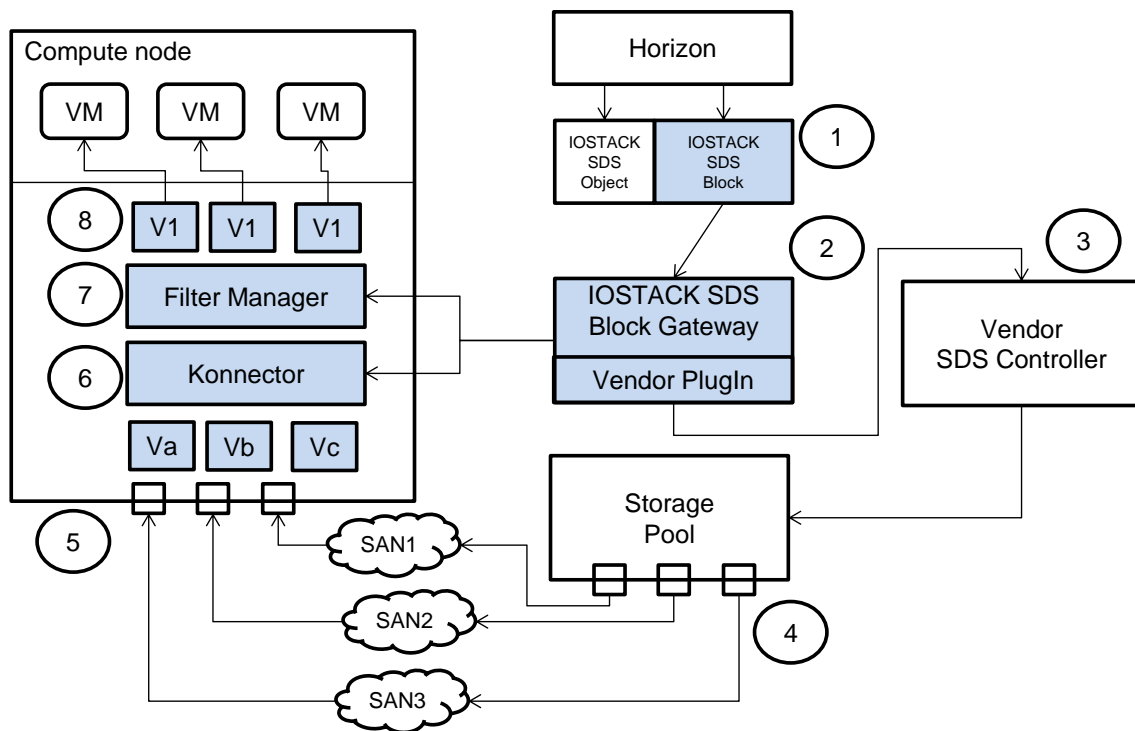


Figure 18: SDS Toolkit Overview

- (2) SDS Gateway
- (3) SDS controller
- (4) Storage pool equipment over multiple SANs
- (5) Consumer node SAN termination points
- (6) Konnector storage terminator of storage volumes
- (7) Filter Manager
- (8) Filter Volumes on top of storage volumes

The SDS Gateway provides a REST API for a client to manage storage. In fig 18 the client is the OpenStack Horizon dashboard issuing storage provisioning requests. These requests are CINDER API commands, CINDER is the OpenStack storage API used by the OpenStack Horizon dashboard. These CINDER API commands are issued to the SDS Gateway which translates them into requests understood by the backend storage arrays. This translation takes place in the Vendor plugin layer of the SDS gateway.

The SDS Gateway maintains an object model shown in fig 20 which is configured by the Horizon Dashboard SDS extension. A CINDER API request to the SDS Gateway will provision a storage volume from a specific storage group. The storage group can be created by the SDS extension of the Horizon dashboard using the SDS Gateway API. The SDS Gateway API provides a set of SDS functions such as creation of storage groups. A storage group has a name, a set of storage nodes and a policy. The group name is used by the user to select what type of storage is required for the workload (example Gold, Silver, Bronze). The storage provisioned will be from one of the nodes in the storage group. The policy configures default storage options for all volumes provisioned within that storage group. Some of the options are storage node specific, such as the fabric to export the volume on, other options are specific to the consumer node (Compute node) where the storage volume is

attached to. The compute node specific part of the storage group policy defines the filter stack that is created when the storage array volumes is attached to the compute node.

7.1 OpenStack Horizon dashboard extensions

The SDS extension to OpenStack provides the user with a set of configuration and monitoring options. The configuration options allows the user to create storage groups which contain storage nodes, each storage group has one policy. The policy defines which media tier to use, default fabric and filter stack to create when a volume is attached to the compute node. The filter stack is instantiated only when a volume is attached to a compute node. The Horizon extensions are used to create the configuration model shown below. This model is then used by the SDS Gateway and Konnector modules to instantiate and configure the filter stacks and volumes.

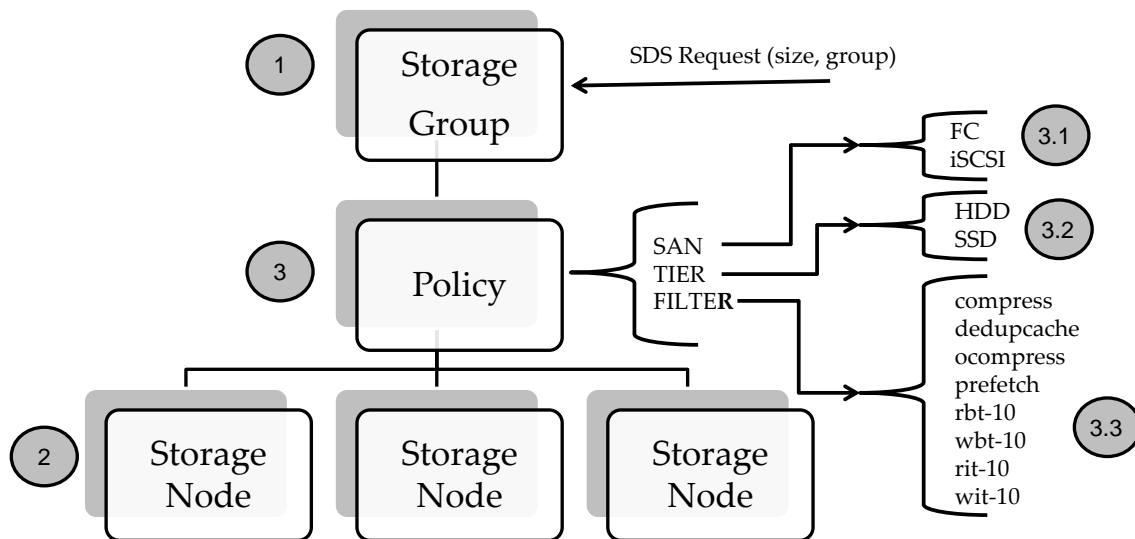


Figure 19: SDS groups and policy

8 Konnector operation

The consumer node (compute node) specific component defined in the storage group policy is a description of the filter stack that is dynamically built once the storage volume has been attached to the consumer node. Once a storage volume has been attached to a consumer node, the SDS Gateway shown in fig 20 uses the Konnector API to dynamically create a filter stack between the attached volume and the volume presented to the final consumer, ex. a Virtual Machine. This operation is shown in fig 18 in the steps 5, 6, 7 and 8. This schema allows storage volumes to be created on storage nodes and a stack of dynamic storage filter functions to be applied to the storage volume independent of the storage array volume and the storage node hosting that volume. For example a volume could be created on a storage node but the data to and from that volume could be compressed by a compute node filter.

The goal of the Konnector filter stack is to provide a flexible platform for innovation and value added functions on top of the storage array volumes.

Storage volumes are natively managed in OpenStack compute nodes by Kernel based drivers. Inserting filter functions on a data flow in Linux Kernel space would be extremely difficult. The Konnector framework moves the data flow from the kernel space into the userland space where filters can be dynamically created at run time. Run time creation of the filter stack is mandatory because the volume and its stack are only instantiated when the storage array volume is attached to the consumer node/consumer VM.

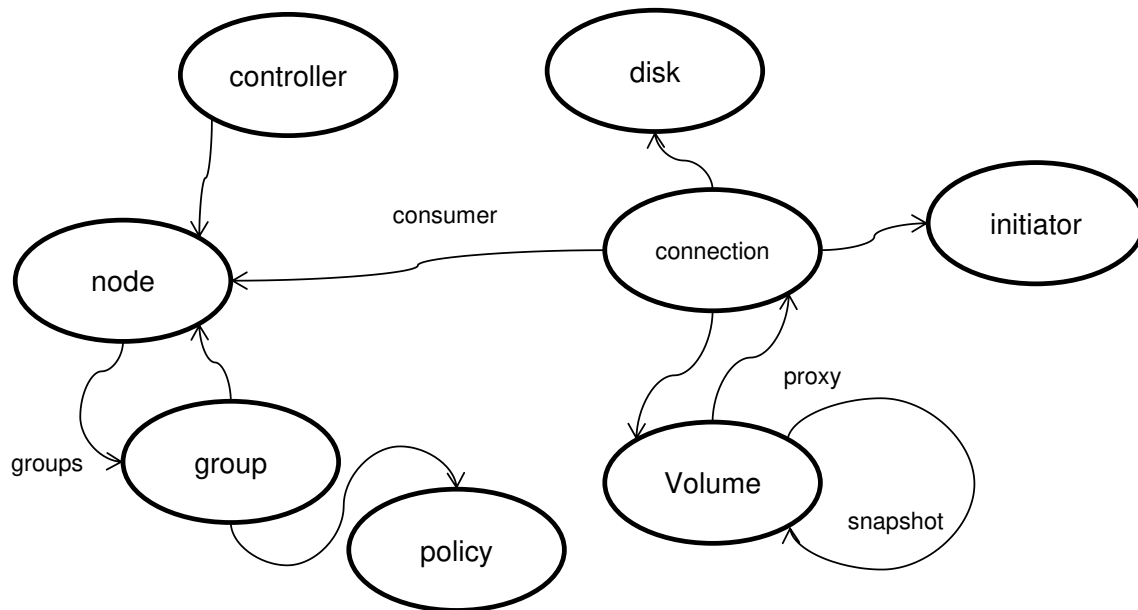


Figure 20: SDS gateway model

Table 8.1a: Candidate Storage Functions to be implemented as Filters

Filter Function	suitability	CPU load	IO Latency	BW impact
Compression	medium	high	medium	medium
Encryption	high	medium	medium	medium
Deduplication	medium	high	high	high
xN way Sync copy	high	medium	medium	medium
xN way Async copy	high	medium	low	low
Backup	high	medium	low	low
IO control	high	low	low	low
BW control	high	low	low	low
Block to Object	high	medium	low	low
Sample filters	High	low	low	low

8.1 Filter Functions

Filter though a stand-alone function are an integral part of Konnector package. Filter functions are managed by the Konnector API. In theory any number of filter functions can be created, this is key to unlocking innovation of top of data flows. In Table 8.1a we present a list of examples is shown and the motivation behind developing these filters.

Compression Reduce the memory used for cache, compressing blocks. Increases the effective cache space in compressible content, but non-compressible content may suffer penalties. Other filters may generate compressed output, so the benefits come from a better space usage, however the main challenge is to export this to the user as the device needs to be expanded with a fixed ratio as it dynamic volume expansion is not allowed at this level. Compression is a cpu intensive algorithm and therefore will add latency to any operations either compressing or uncompressing the data. The more BW the more the compression algorithm will have to process and this will negatively impact the bandwidth. In some cases data will compress and uncompress easily (ex all zeros) in these special cases performance will increase.

Encryption Encrypted data flows provide two key advantages;

- Deletion of data can be achieved by simply removing the key, this reduces the deletion time of data from hours to seconds.
- Data is protected in a multi-tenant environment from data centre personnel and data centre tenants.

A Key management system is required to manage Keys securely under the user's control. Encryption is a cpu intensive algorithm and therefore will add latency to any operations. The more BW the more the encryption algorithm will have to process and this will negatively impact the bandwidth.

De-duplication De-duplication removes multiple copies of data stored in a the storage array. De-duplication reduces consumed space as only one copy of every unique record is stored, this can improve performance on de-duplicated data writes. A record of say 8Kbytes of data is known by an SHA value, if records have the same SHA the data is not written twice. De-duplication is a complex function, requiring a large high speed working storage space. De-duplication is also very cpu intensive in calculating SHA values for every record on the compute node. However, deduplication can be used also to increase the memory available for cache in a non-persistent way. The filter stores the last used blocks into memory (with a red black tree) indexed by the memory content itself. Deduplication is a very CPU intensive application requiring sha calculations and read modify write operations of the basic records, this adds considerable latency delay and lowers bandwidth.

Sync copy Providing multiple secure copies of data improves data resiliency. Data resiliency is one of the key attractions of Object storage. By providing filter based data resiliency low cost single controller block storage devices with high performance can be used in place of costly high availability enterprise class storage systems. A considerable amount of high speed meta-data is required to keep the status of the written blocks, the implementation of recovery procedures when a failed write IO occurs are complex. Copy is not a cpu intensive application as IOs are simply forwarded to multiple destinations. Sync copy requires that all writes be completed before a write is completed, this constraint increases the latency and lowers bandwidth.

Async copy Providing multiple secure copies of data improves data resiliency. Asynchronous data resiliency is one of the key attractions of Object storage. By providing filter based asynchronous data resiliency, a better compromise of performance and data resiliency could be achieved than by using Object storage. This is especially true if low cost single controller block storage devices are used. A considerable amount of high speed metadata is required to keep the status of the written blocks is required. Since the data is copied asynchronously recovery procedures to failed IOs are simpler than Sync copy to implement. Copy is not a cpu intensive application as IOs are simply forwarded to multiple destinations. Async copy does not requires that all writes be completed before a write is completed, the lack of this constraint means latency and BW are not negatively impacted.

Backup A filter that marks block ranges as dirty when written to a storage volume could then read and copy these block ranges to a backup store. Backup is means of achieving additional data resiliency and copies of storage volumes at a point in time. A point in time backup requires a snapshot of the volume, this is a complex operation. Backup also requires that the volume is in a flushed state from the host OS, i.e no cached data is in the compute system, everything has been flushed from the filesystem to the storage volume. This is actually a simpler task to implement for a filtered volume than a non filtered volume as filtered volumes have local knowledge of the compute node file system. Backup is not a cpu intensive application as IOs are copied from a source volume to a remote when required with little data processing. Backup is essentially a scheduled activity so does not interfere with the normal IO operation and has little impact on latency or bandwidth.

IOPS control IOPS control is important as it allows a user to provision not only terabytes of data from a storage array but also a portion of its IO capability. IOPS control is completely compute node based and requires no API interface to the storage array controller. Controlling IOPS on storage arrays with high IOPS rates does not work in badly behaved multi-tenant systems. A badly behaved

tenant is one which does not obey any IOPS rate limiting. For IOPS provisioning to be effective all volumes must subscribe to an IOPS rate control mechanism.

BWPS control BWPS control is similar to IOPS control except the read and write megabytes per second (BWPS) is controlled. BWPS faces the same challenges as IOPS control.

Block to object For applications requiring block semantics but Object storage class speed and cost, a filter to make the block to object transformation would be very useful, examples of such tools are S3backer and CEPHs RDB driver. Porting a tool like S3Backer to the Filter Framework.

Sample filters Sample filters are useful templates for anyone developing a more complex filter. The sample filter should be a good example of how to code a more comprehensive filter.

8.2 Konnector design

The Filter stack is dynamically created as storage array volumes are attached to a compute node, this is made possible as each filter is implemented as a .SO (a dynamic linked library). The Filter manager when requested through the Konnector API will dynamically build the filter stack using a set of .SO dll library files. This .SO approach allows the filter stack to be built on demand, it also allows any third party to create filter functions and add a data inline processing function into the data flow between the consumer (ex. Virtual Machine VM) and the storage Array volume. The storage array volume is agnostic to the filter function since the storage array just sees data in/out of the attached storage array volume on the consumer node, it does not see nor is aware of any of the filter transformations.

The Konnector block diagram is shown in fig 21, the principal components of the system are the flow of SAN SCSI commands (1) into the LINUX LIO core, the backing store for this SCSI device is shown as /dev/sdx (2). This device /dev/sdx (2) is the storage array volume created by the SDS gateway, exported from the storage array and imported in the consumer node. In non IOStack Open-Stack this volume would simply be attached to a virtual machine. In the IOStack enabled compute node the data flow (SCSI Read and Write Commands) are written through two buffers (3-4) into userland space. In userland space the filter manager (5) recovers the data and routes the data to the stacked up filters configured for that volume (7). Data from one filter can be routed into one filter then to succeeding filters (6) in the stack.

8.3 Filter Implementation

The filter objects are built from C source files. The SDS toolkit provides a number of skeleton filter samples such as 'nop' which serves to act as a template for filter development. Within this filter, there are five functions which are run-time linked to the filter manager for version filter manager 1.1.

- write transformation
- read transformation
- get-name
- pass-args
- pre_read

The write transformation - void write-xform(void* buf, unsigned long cnt, unsigned long offset, bool *doWrite) is passed a void pointer to the payload data together with a length argument 'cnt' and an argument 'offset'. Essentially, this function exists to perform a transformation at its defined filter level on payload write data bound for the device. The last parameter is an output parameter intended to avoid writing the content to the disk, as it is already done in the filter. For example, filters that write to other zones of the device will bypass the final write.

The read transformation - void read-xform(void* buf, unsigned long cnt, unsigned long offset) uses 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.

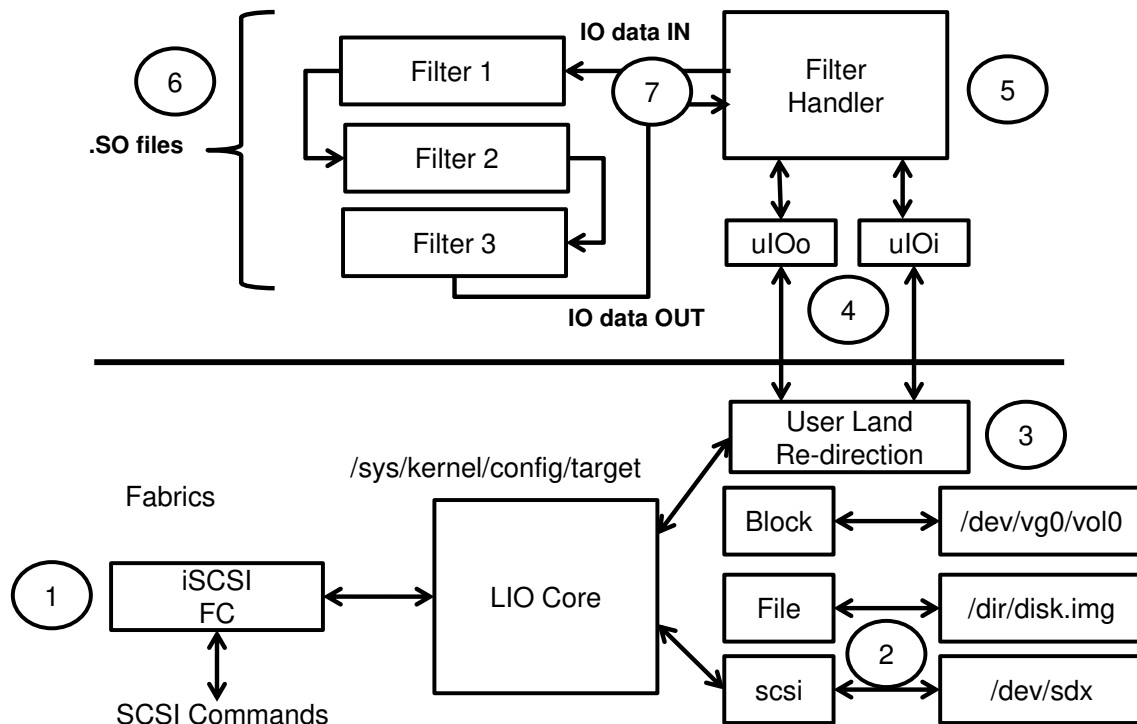


Figure 21: Konnector Block Diagram

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 filter daemon is executed in debug mode.

pass-args - This function allows the user to pass arguments to the filter function when the filter is instantiated. The arguments can be specified using the syntax of the filter specification in the Horizon dashboard, ex myfilter arg1=0x10, arg2=helloWorld. The Arguments are filter specific and are not parsed by the filter manager, they are passed to the filter transparently.

pre_read - `int pre_read(void* buf, unsigned long cnt, unsigned long int offset, unsigned int fdesc, bool * doRead)`. This function that is called before the read-xform by the filter framework offers a way to the filter to bypass the read function that goes to the non-filtered device. It is intended for cache filters, so if the content is already in memory there is no need to go to the non-filtered device and then the doRead parameter will be 'false' and the content requested will be already in the buf pointer.

8.4 Filter toolkit

The filter framework provides a development toolkit that allows easy creation of filters.

The current SDK provides a build directory which compiles all filters and copies to the filter directory where the filter framework attempts to match the API call to create a filter volume with the name of an existing filter. A filter is instantiated through the Konnector API, the API is passed the name of the filter which must be present in the filter directory. The filter name to be instantiated and its arguments are stored in the storage group policy created by the OpenStack Horizon dashboard. Every storage policy has a property which is the list of filters to be instantiated when a storage volume is attached to a compute node.

Each filter must code the four entry points described in the previous section.

- write transformation
- read transformation

- get-name
- pass-args
- pre_read

Once these entry points and the added value functions coded in the write and read transformations the filter is ready to deploy to the Konnector enabled system.

The filter framework and the filters are entirely decoupled. A filter could be compiled anywhere and simply copied to the Konnector filter directory.

The listing below shows how the filter files are compiled and stored in the Konnector-filters directory. If a Konnector API call to start a filter with name XYZ is called, a search for the filter with name "XYZ" is made in the directory "Konnector-filters" and if found the filter is deployed in the filter stack for the designated volume. Filters can be stacked one on top of another, the data flow will flow from the top of the stack to bottom for "data writes" and from the bottom to the top for "data reads". A sample filter BitRev is shown in section 10.4 BitRev Filter example. In Section 10.4 the source, header and make tools for the filter are shown. The example shows how the required entry points can be coded as described in section 5.3. To write a new filter the following steps are performed.

- step 1: code the filter entry points as shown in the example with the particular function you wish to implement, this is the coders decision.
- step 2: create a header file as shown in the example.
- step 3: build the filter using the script in the example or use the script below to build all the filters in filter directory.
- step 4: copy the filter .so file to the directory Konnector-filters

The filter manager will now be able to access and load the filter into a filter stack on top of a storage array source volume.

Script to build all filters in the default filter source directory.

```
for i in $( ls filters ); do
( cd filters/$i && gcc -Wall -fPIC -c $i"_filter.c" \
  && gcc -shared -Wl,-soname,"lib_"$i".so.1" -o $i".so" $i"_filter.o" \
  && if [ -f $i".so" ];
then
  cp -f $i".so" /usr/lib64/Konnector-filters
else
  echo " File$i"so"_"does_not_exist."
fi )
done
```

8.5 Standard Toolkit Filters

The toolkit filters serve the purpose of test filters to verify the filter framework is working. They also serve as sample filters for people to develop their own filters. The IOPS and WBPS filters are part of the filter framework that provides IO and BW rate limiting.

The SDS toolkit provides a number of standard filters including;

The NOP filter provides no processing on data flow between the attached storage and the storage consumer. In this regard it can be seen as the overhead of the filter framework itself.

The XOR filter provides a simple obfuscation of the data written to the storage volume by performing an XOR operation on the data written to the storage and an XOR operation on all data read from the storage array. The XOR filter is a simple filter but is useful in that it requires the filter function to perform a transformation on all read and written data. To use the xor filter add the string "xor" in the SDS dashboard policy field.

The BitReverse filter provides a simple obfuscation of the data written to the storage volume by performing a bit reverse operation on the data written to the storage and an bit reverse operation on all data read from the storage array. The BitReverse filter is a simple filter but is useful in that it requires the filter function to perform a transformation on all read and written data. To use the BitReverse filter add the string "bitrev" in the SDS dashboard policy field.

The IOPS filter allows the user to set a value of the IOPS allowed to traverse the filter stack. The IOPS can be set for read IOPS, write IOPS or both. To use the IOPS filter add the string "rit 10, wit 10" in the SDS dashboard policy field for say 10 read IOPS and 10 write IOPS.

The BW filter allows the user to set a value of the BWPS allowed to traverse the filter stack. The BWPS can be set for read IOs, write IOs or both. To use the MBPS filter add the string "rbt 10, wbt 10" in the SDS dashboard policy field for say 10 megabytes per second read and 10 10 megabytes per second writes.

9 Advanced Filter description

The filters developed here go a step further of the original behaviour of the filter framework. The original behaviour was a 1:1 block transformation (reading or writing), so for each read or write request to the non-filtered device the same request is done to the filtered device. For our scenarios we need to be able to do a $n:n$ transformation, so we need to do additional reads or writes to the non-filtered or filtered device. For example, prefetching or cache filters need to check before the real read if the content is available or not, then the framework knows if the real read should be done or not. On the other hand, output compress filter generates a new filesystem to the real device transparently, so before it writes the real data it should know if the data should be written or not (doWrite parameter of the write-xform call).

The filters developed at BSC are classified in four types:

1. Prefetch filters (prefetch)
2. Cache filters (dedupcache and compress)
3. Output modification filters (OCompress)
4. Evaluation filters (mockup)

Prefetch filters preload data in advance to the filter to avoid the network latency. prefetch filter is divided in two filters, the first one logs the offset and sizes of the data that is being used in the VM. The second filter preloads the data in advance. We are going to develop a new filter on the next period that will enable Just-in-time prefetching so the blocks are only preloaded just when they are going to be used. Prefetching will allow important latency reduction on storage as the devices are not directly attached to the client and are provided through a network.

Cache filters stores any block read into the filter. The filter objective is to increase the performance on workloads reusing data. The cache is reduced using deduplication (dedup) and compression (compress), so we can store more data with less memory usage and surpass the buffer cache space. The deduplication method was used on a FP7 project called IOLanes [1], implemented directly in the kernel.

Output modification filters generates a different output from the one expected. The main difference is that the filter is persistent, so the output can only be read if the same filter is used. ocompress generates a compressed file system using two compressors (for Idiada use case). The filter is transparent for the user. However, further modifications in the filter framework are needed to allow to export a, i.e., 6 GB real volume, and present it inside the VM as a 10 GB volume. The main issue is that the increment of space should be static and in advance (using some % gathered or introduced by the user) and can not be changed as it will confuse the VM operating system. Idiada needs this filter due to that their data is highly compressible (but heterogeneous) and they are using a lot of space. Idiada is using actually a in-home special compressor that provides better space reductions than other available compressors, but with the use of the filter they will be able to access the compressed data transparently.

Evaluation filters tries to introduce several parameters to evaluate the framework, it can also be used to introduce latency delays or CPU usage delays in the filter workflow. mockup filter has those parameters, so delays are introduced on each read or write request.

The advanced filters are available at github.com/bsc-ssrg/BlockStorageFilters-IOSTACK. These filters include

9.1 Output Compress Filter

The filter creates a compressed filesystem inside the unfiltered device and present a normal device to the user. The compressed filesystem was created for the Idiada use case due to that they have big output files that are highly compressible.

9.2 Compress Cache Filter

The compress cache filter (and the deduplication filter using the same concept but with deduplication instead of compression) is a filter that shows benefits when the data cached is reused. As this is not the case on the different use cases, we are going to evaluate it using the Idiada dataset and forcing a reuse.

9.3 Deduplicated Cache Filter

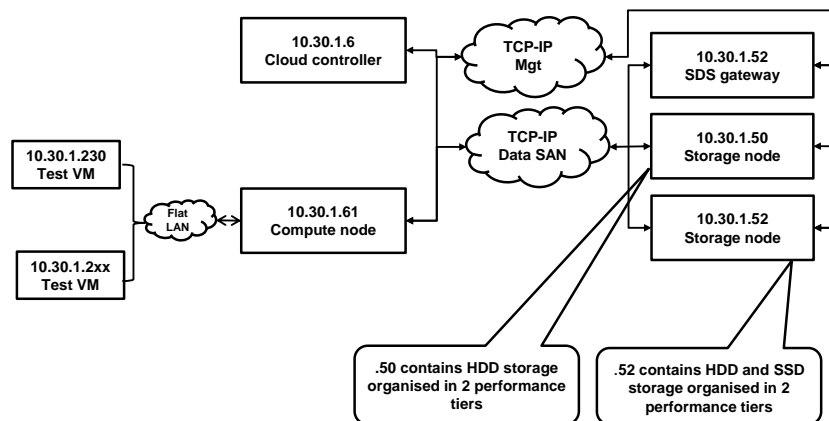
Similar to the previous filter, but using deduplication and with a 100% deduplicable dataset (10 GB) we will obtain better results as the cost of compression is removed. The duplicated filter stores the blocks in a red-black tree having as key the block content, and in another red-black tree, indexed by the offset, the pointer to the content on the first tree¹.

9.3.1 Controlled environment description

The controlled environment is using direct attached devices to the VM running the filter framework. This allows to test directly the overhead of the different filters, and allows us to control the memory offered to the filter so we can check cache policies. This also avoids possible network bottlenecks.

The VM is setup to run 4 GB of memory and 2 cores. The devices used are a 256 GB SSD (SAMSUNG SSD PM851 mSATA) and for some filters we also use a slower device (USB with higher latency) to show more clearly the benefits. The overhead of the filter framework can be extracted of the comparison of the use of the direct native device and the framework with a NOP filter. However, a more exhaustive evaluation has been done in Arctur testbed.

The host system is an Intel(R) Core(TM) i7-4700 CPU with 16 GB of memory. We always clean the different caches of the system on each repetition of the tests.



39

Figure 22: Arctur Overview

9.4 SDS toolkit functional test results

Table 9.4a below shows the functional test results. As demonstrated in the EU commission review all the functional components of the SDS toolkit are demonstrable and functional. The SDS toolkit has

¹The memory overhead is 24 bytes per new block and 12 bytes per deduplicated block as minimum, depending on the rb tree implementation

been deployed in Arctur and is available as a test and development platform for BSC and MPSTOR. The functional testing required configurations of storage groups with associated policies. Fig 19, Fig 24, Fig 25 below shows the setup steps.

1. A storage group is created (ex. Gold storage)
2. Physical storage nodes (by ip address) are associated with the storage group
3. A policy is created which specifies the SAN type, the media tier and a set of filters that will be deployed when the storage volume is attached to the compute node. The filter definition is a string of filter names and arguments for example in the filter field the following **xor, rit-10, wit-10, nop** would instantiate a stack of "xor" filter with a read IOPS of 10 IO per second, a write IOPS of 10 IO per second and finally the "nop" filter.

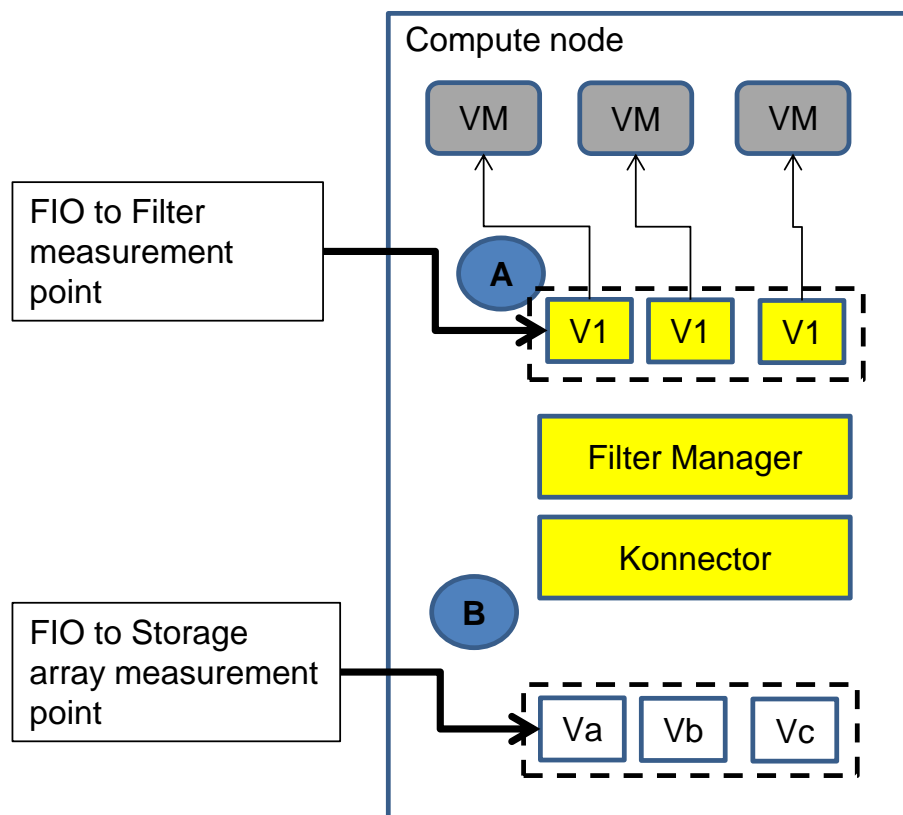


Figure 23: Konnector Performance Test measuring points



Figure 24: Storage policy

10 Conclusions

The conclusion of the testing of the filter framework is that the SDS toolkit is an effective set of tools that meet the goals of the project. The goals achieved were:

Working prototype A working prototype has been developed which shows all the component parts functioning together.

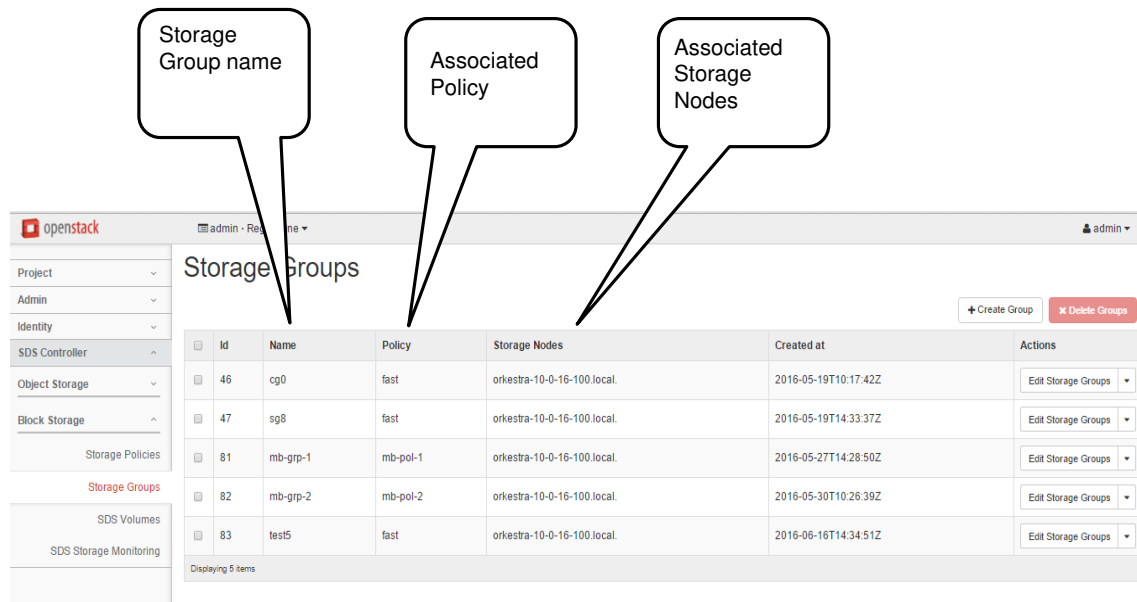
User control of datacenter infrastructure The extensions to the Openstack horizon dashboard allows an administrator to create volume services using policy parameters so that the user can choose the correct type and capacity of storage for his work-load.

Automated provisioning The policies, storage groups and volume filter services configured in the admin dashboards are automatically deployed when volumes are created and instantiated, this means a high level of automation of complex configurations has been achieved. This allows the user to deploy the correct storage configuration easily for his workload.

Inline filters The Filter framework allows the creation and real time deployment of inline pro-

Table 9.4a: Functional testing results

Test	result	comment
Create storage policy with defined filter	pass	Uses SDS dashboard
Create storage policy with defined media tier	pass	Uses SDS dashboard
Create storage groups with defined set of storage nodes	pass	Uses SDS dashboard
Provision attach storage volumes from storage groups to compute nodes	pass	Uses SDS dashboard
Instantiate filter volumes on compute node	pass	uses Konnector
Attach filter volume to VN	pass	uses Konnector
IOPS Filter	pass	uses FIO and VM
BWPS Filter	pass	uses FIO and VM
Volume metrics	pass	uses metrics dashboard 9



The screenshot shows the OpenStack dashboard interface. On the left is a sidebar with navigation links: Project, Admin, Identity, SDS Controller, Object Storage, Block Storage, Storage Policies, Storage Groups (highlighted), SDS Volumes, and SDS Storage Monitoring. The main content area is titled 'Storage Groups' and contains a table with 5 columns: Id, Name, Policy, Storage Nodes, Created at, and Actions. There are buttons for '+ Create Group' and 'x Delete Groups' in the top right. Three callout boxes are present: one pointing to the 'Name' column labeled 'Storage Group name', one pointing to the 'Policy' column labeled 'Associated Policy', and one pointing to the 'Storage Nodes' column labeled 'Associated Storage Nodes'.

Id	Name	Policy	Storage Nodes	Created at	Actions
46	cg0	fast	orkestra-10-0-16-100.local	2016-05-19T10:17:42Z	Edit Storage Groups
47	sg8	fast	orkestra-10-0-16-100.local	2016-05-19T14:33:37Z	Edit Storage Groups
81	mb-grp-1	mb-pol-1	orkestra-10-0-16-100.local	2016-05-27T14:28:50Z	Edit Storage Groups
82	mb-grp-2	mb-pol-2	orkestra-10-0-16-100.local	2016-05-30T10:26:39Z	Edit Storage Groups
83	test5	fast	orkestra-10-0-16-100.local	2016-06-16T14:34:51Z	Edit Storage Groups

Displaying 5 items

Figure 25: Storage groups

cessing in the compute node of the dataflow between a VM and the storage Volume, this inline data flow processing was a core goal of the project.

Added value Inline filters Added value filters were developed that go beyond proving the feasibility of filter use. These added value filters improve the workload requirements of the application.

The toolkit provides the following features: A Horizon dashboard interface to create storage groups of storage nodes, media tiers and consumer node filters.

A flexible means to create storage policies and attach storage policies to storage groups.

By using storage policies storage volumes with defined IOPS or BW characteristics the valuable resource of storage array IOPS and BW can be managed correctly.

A filter framework that does not adversely affect the performance of the attached storage but does require extra CPU resources depending on the filter type being executed.

Future work: The SDS toolkit in this work package was tested with generic workloads representative of Big Data workloads. The next steps will run Big Data workloads with added value filters and demonstrate improvements of the Big Data applications.

A number of lessons have been learned during the project implementation, testing and presentation at industry event which were used to direct the development.

MPSTOR has learned that storage provisioning in the data center is now a basic requirement. The lowest practical level of provisioning is through storage APIs and command lines tools, low level command line (python based) tools are seen as very powerful means to automate datacenter processes. It is not obvious which virtualisation frameworks will be winners so therefore which higher level frameworks to integrate with such as OpenStack, Kubernetes etc is difficult to judge. The best strategy is to have a set of provisioning tools that range from low level API, storage Array command lines tools, SDS command line tools to finally SDS integration with higher level cloud or virtualisation frameworks. A second key lesson is that end to end provisioning is the valuable part of storage provisioning, it is not simply the creation of storage volumes from storage arrays that is important but the attaching through datacenter fabrics of those volumes to the final consumer of the storage. The final lesson we learned was in-line filters is a useful framework as it allows the storage array functionality to be extended. The filter framework competes for compute resources so careful consideration needs to be given to what filters should be developed, we believe for general workloads data protection, data security are more appropriate filters than say compression or deduplication. For very specific applications filters provide a very good framework to do inline processing of the data.

Since the start of IOStack many changes have occurred in the datacenter, such as new fabric protocols iSER and new fabrics and protocols such as NVMe. NVMe has the potential to radically change how the disaggregation of compute and storage is implemented. NVMe by itself is almost designed to solve the disaggregation requirement for block storage in IOStack. MPSTOR will therefore try to find the best fit between new technology changes (ex NVMe), datacenter practice (devops CLI tools) and virtualisation framework changes (ex Docker, Kubernetes, OpenStack) and tailor the IOStack block layer development so that it becomes part of the standard automation tools from MPSTOR.

11 Appendix

11.1 SDS controller and REST API overview

The code providing the REST API is a standalone module. This module contains a plugin which can interface directly to a downstream storage array controller assuming the storage array has a API. In IOSTACK development we have used the MPSTOR API to manage the storage Array layer.

The REST API module reads the storage Array API to fulfil queries (GET requests). The REST API module contains a plugin, the APR layer (as a Python library) to communicate with the storage array controller using a socket-based XML communication with the storage array nodes. This library is specific for each storage array vendor and has to be customised if a different storage array is used. The rest of the SDS module does not change. In fig 26 the module objectapi is the where the storage specific functions are coded for any particular storage array.

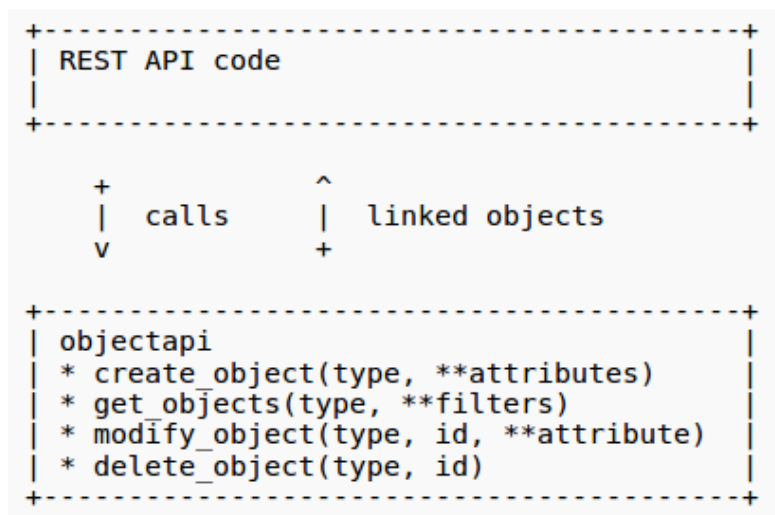


Figure 26: SDS Rest API to Storage module

In addition to managing the storage arrays, volumes are attached to the compute nodes. To complete the automation on the compute side a JSON interface is used to communicate with the Konnecter in-band layer. A third-party library code was used to create the REST API, it was chosen after evaluating a number of alternatives, taking into account the following considerations. Heavyweight "full stack" frameworks such as Django have a high learning curve as well as being overkill for a REST API. Therefore the technologies examined in more detail were those taking a more minimalist approach, including Flask, Morepath, CherryPy and Bottle, 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 was 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, which is easily included in the storage array library package. 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. The SDS Rest API model is shown below.

Internal Operation The Server class starts a ThreadingWSGIServer thread which in turn creates a RequestHandler on a new thread for each request. There is only one instance of the application (Middleware + API) for all threads, but it uses thread local variables where necessary, including an instance of apmserver.APMRequestHandler (the "coreserver"). Bottle is used as a web framework library: the application is a subclass of bottle.Bottle. Underlying the objectapi is the same code that currently responds to APR requests sent over an XML interface, but its functionality is invoked directly (method call) by the REST API rather than by sending an XML request to a server.

Objectapi Key to the architecture is to present the storage controller 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 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 Array to accomplish some action. The overview is shown in fig 27.

This functional division should also 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 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. The details of the objectapi get-objects method are not included here, but to allow the REST API code to work efficiently, the filters supported may need to be more sophisticated than attribute-X equals value-Y.

For example, get-objects<"widget",id in<23,27,56>>. 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 e.g., times are strings, but objects may be passed as equivalent lists of Python dictionaries i.e dicts. In LTO, an object is like any other object, with the addition of two special attributes: "-type" and optionally "-refs". For example, 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

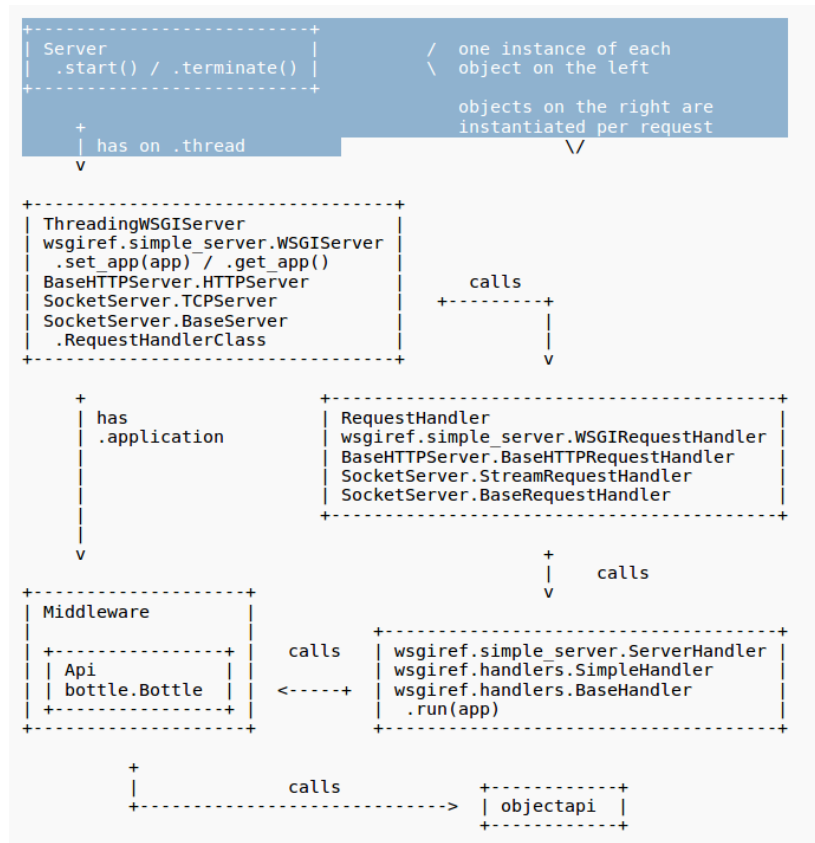


Figure 27: SDS Rest API

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.

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 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, JSON Hyper-Schema, Siren, JSON-LD, Collection+JSON, Mason, ... - though such choices might be avoided by choosing XHTML instead, JSON is generally used as a simpler protocol), and 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 may be better to adopt 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 labour saved by following an existing thought-out and already-documented convention for organizing data.

For the SDS REST API, HAL plus JSON is used. Although it is simpler than most alternatives, the specification is not as tight nor the guidance as clear as it might be. See, for example, the long discussions following questions like this one. Therefore the following guiding principles are defined:

- Put any attribute that does not correspond to an addressable object name, size, etc in the body, i.e., at the same level as -links and -embedded;
- Wherever an attribute corresponds to an object available at another URI or a list of such objects, include it instead in the "-links" section;
- Omit any link attribute from -links where its value would be null or an empty list see note below;
- For collection resources, include under -embedded complete representations of the items;
- CURIEs and profiles are not used see note below;
- Represent errors based on the approach described in Problem Details for HTTP APIs - as this is consistent with HAL, the same Content-Type, i.e., application hal plus json, will be used for errors.
- Empty links: The HAL spec states that "The reserved "-links" property ... is an object whose ... values are either a Link Object or an array of Link Objects", which implies that null is not allowed, and, though it is unstated, consistency would suggest that empty arrays are also excluded. However, disallowing empty links means that external query-string filters based on link attributes,
- e.g., /volumes?snapshot-of=null cannot be supported see Sorting and Filtering below.

Profiles and CURIEs A formal profile defining the semantics of the SDS REST API, e.g., using ALPS, is considered unnecessary. Both profiles and CURIEs are optional in the HAL specification, but it does recommend that non-standard link relations be URIs, for which CURIEs are an abbreviation. Thus, for link relations specific to the SDS API, a prefix such as "sds:" could be used. However, such relations effectively serve as attribute names, in which a colon ':' is an inappropriate character which could cause problems - in fact, if an "sds" CURIE were defined then, e.g., "sds:raid" would really be an abbreviation for something like "http://mpstor.com/sds/link-relations/raid" with which strictly they should be replaced, which is even worse. (Problems with the use of CURIEs and related ambiguities in the HAL specification have been discussed.) The pragmatic approach therefore is to use recognized link relations where suitable and to add others as necessary in the same namespace. (This might be considered a divergence from the HAL approach which requires these strings to be link relations rather than a violation of the Web Linking RFC 5988.)

Versioning This is an extremely contentious topic with many different approaches. HATEOS purists argue that versioning should not be necessary because clients should always follow links starting at the root URL to discover how to perform the required operation or retrieve the desired information. This is rather unrealistic and it is difficult to find any example of a major commercial API which is based on such an approach. After the principle of versioning is accepted, there are discussions about whether the whole API or individual resources should be versioned. Versioning at the level of individual resources seems unnecessarily complex and would make managing compatibility more difficult without any particular advantage. The more common approach is simply to have a single version for the entire API and that is the convention favoured here. What granularity should be used for versioning? Cogent arguments are made for using only a single integer, and this was the approach used for XML Protocol Versioning. However, the XML solution allowed for a client that could send requests conforming to more than one version simultaneously, so that a newer client could work with an older server, whereas the same approach is not possible using JSON, except in the less elegant way of having a client that repeats failed requests using a different API version. It is

important to note here that many discussions about versioning on the internet assume there is only one server - a website - and so disregard the possibility of clients being newer than the server. The convention we adopt here follows that described by Jean-Jacques Dubray in this discussion: A client may send a request using version X.Y to which the server is free to respond using API version X.Z where $Z \geq Y$ and minor version increments do not break existing clients, assuming that clients ignore new resources or attributes added to the API. (A variation on this approach is the use of something other than an integer as a minor version number: Stripe uses a date. However, no advantage accrues in our context.) The final area of contention is how the version should be specified. Note that the HTTP "Accept:" header is properly used for the media type, not the API version: see, e.g., Richardson et al., "RESTful Web APIs", Chapter 9; also, versioning the Accept header is appropriate for GET, but not CREATE. Perhaps the most pragmatic observation here is that of Troy Hunt, who, agreeing that all approaches are "wrong" in one way or another, argues that clients be given a choice of which wrong approach to use. Following this recipe, the REST API provides clients with a choice: either add a custom HTTP "API-Version" header (the recommended solution), or, where it is more convenient, add a /vX.Y/ prefix to the URL. Dates All dates and times transferred over the REST API will be UTC. It will be the client's responsibility to convert these to and from local time wherever that is required.

Sorting and filtering Sorting and filtering of collection resources e.g., /volumes could be supported using query parameters in a URL, e.g., /volumes?filter=size:ge:5,name:eq:'i*&sort=-size,name would return a list of volumes with a size of at least 5GiB and a name beginning with the letter 'i', sorted with the biggest volumes listed first (the '-' prefix inverts the default ascending order) and volumes of equal size sorted by name. Sorting is considered a lower priority feature, as the API will not be used directly in a browser.

Paging Where a collection of resources is accessed, the chosen convention see "HAL conventions" above is to include a full representation of each resource in order to avoid a client being required to make many separate queries where it needed to list not only the URIs of the individual resources but also some attributes for each of them (e.g., to show name, size, and status for each volume listed). In some cases, especially where a filter is not used, this could result in a very large response from the REST API server.

The standard approach to dealing with this is to divide the results into pages whenever the number of resources in the list exceeds, e.g., 100, so that in navigating to /volumes a list limited to 100 volumes is returned together with a "next" link to /volumes?page=2.

A potential problem arises however where an offset parameter like "page" is used: the REST API, being stateless, stores no context between requests, so if one of the volumes listed on page 1 is deleted before the request to get page 2 then the second set of one hundred volumes starts at a different offset and so omits the first volume that would otherwise have been shown on that page. Thus, the client following all the "next" links to the last page may not see all the volumes.

A workaround is to use a "cursor" parameter instead, e.g., a "next" link to a URL like /volumes?after=327, where 327 identifies the last volume displayed on the current page. However, where volumes are sorted by a parameter which may change, this also fails. For example, suppose a collection is sorted by name and the first page has been returned; now, before the client request for page 2, one of the items that would have appeared on that page is renamed "AAA" so that it should appear on page 1. If it is agreed that the solution of somehow introducing state (e.g., by caching the results on the first request) is impractical and unacceptable, then no good solution to this seems possible. Even if paging of results is supported, it may be limited to "next" as support for "prev"/"previous", "first", and "last" links presents similar issues and is unnecessary, especially given that clients are not expected to be human beings directly accessing the API with a browser.

Hierarchies of resources It seems natural to envision entities arranged in a hierarchy with the resource URLs representing them reflecting this, e.g., /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 storage (chosen storage array for the IOStack project) 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 thingumajig C is actually in (or under) yokeybob B and widget A?

12 Appendix

12.1 Konnector API

This section describes only the Konnector extensions to the targetd API. The functions below are accessed in the same way as described in the targetd API document using JSON RPC on TCP port 18700. For example, using curl, a create-filtered-volume command could be executed like this:

```
IP=192.168.2.61 or wherever the Konnector service is running curl -user admin:password -X POST http://IP:18700/targetrpc" -d '{"jsonrpc":"2.0", "id":0, "method":"create-filtered-volume", "params":{"name":"fvol1", "device":"/dev/sde", "filters":["xor"]}'
```

Since only the "method" and "params" field are significant, the descriptions of the methods which follow include only the parameters of the method, i.e., the possible contents of the "params" field. Similarly, the sample responses show only the "result" portion of the JSON object returned.

The following are the Konnector API primitives used by the SDS gateway to attach storage to a consumer node and build a filter stack terminated by a top of filter stack volume.

iSCSI initiator commands

1. get initiator name
2. discover portal
3. display discovery
4. display discovery summary
5. delete discovery
6. login target
7. logout all targets
8. display node summary
9. delete node
10. delete all nodes
11. display session
12. purge
13. create filtered volume
14. delete filtered volume

Filtered volume commands

1. create filtered volume
2. delete filtered volume

12.1.1 iSCSI initiator commands

get-initiator-name Get the iSCSI initiator IQN of the Konnector node. No parameters.

Example response: "iqn.1994-05.com.redhat:b0d684d83bb4"

discover-portal Discover all targets for a given iSCSI discovery portal.

hostname the iSCSI target node hostname or IP

discovery-method "sendtargets" (default) or "isns"

auth-method null (default), "chap" or "mutual-chap"

username used only with auth-method "chap" or "mutual-chap"

password used only with auth-method "chap" or "mutual-chap"

username-in used only with auth-method "mutual-chap"

password-in used only with auth-method "mutual-chap"

Example response:

```
{
  "iqn.2004-04.com.mpstor:mb-vol-1": {
    "192.168.2.162": {
      "interface": "default",
      "portal": [
        "192.168.2.162",
        3260,
        1
      ]
    }
  },
  "iqn.2004-04.com.mpstor:mb-vol-2": {
    "192.168.2.162": {
      "interface": "default",
      "portal": [
        "192.168.2.162",
        3260,
        1
      ]
    }
  }
}
```

display-discovery Display all data for a given discovery record.

hostname the iSCSI target node hostname or IP

discovery-method "sendtargets" (default) or "isns"

Example response:

```
{
  "sendtargets": {
    "address": "192.168.122.239",
    "auth": {
      "authmethod": "None",
      "password": "",
      "password-in": "",
      "username": "",
      "username-in": ""
    },
    "discoveryd-poll-inval": "30",
    "iscsi": {
      "MaxRecvDataSegmentLength": "32768"
    },
    "port": "3260",
    "reopen-max": "5",
  }
}
```

```
    "timeo": {
      "active-timeout": "30",
      "auth-timeout": "45"
    },
    "use-discoveryd": "No"
  },
  "startup": "manual",
  "type": "sendtargets"
}
```

display-discovery-summary No parameters.

Example response:

```
{
  "192.168.2.162": [
    3260,
    "sendtargets"
  ]
}
```

delete-discovery Delete discovery of targets at a given IP address.

input parameters and description

hostname the iSCSI target node hostname or IP

discovery-method "sendtargets" (default) or "isns"

login-target Login to a given target.

input parameters and description

targetname target name, e.g., "iqn.2004-04.com.mpstor:mb-vol-2"

hostname e.g., "192.168.2.162"

auth-method null (default), "chap" or "mutual-chap"

username used only with auth-method "chap" or "mutual-chap"

password used only with auth-method "chap" or "mutual-chap"

username-in used only with auth-method "mutual-chap"

password-in used only with auth-method "mutual-chap"

Example response:

```
"/dev/sdb"
```

logout-target Logout for a given target.

input parameters and description

targetname target name, e.g., "iqn.2004-04.com.mpstor:mb-vol-2"

hostname e.g., "192.168.2.162"

logout-all-targets Logout for all targets.

No parameters.

display-node Display all data for a given node record.

input parameters and description

targetname target name, e.g., "iqn.2004-04.com.mpstor:mb-vol-2"

hostname e.g., "192.168.2.162"

display-node-summary Display data for all node records.

No parameters.

delete-node Delete a given node record.

input parameters and description

targetname target name, e.g., "iqn.2004-04.com.mpstor:mb-vol-2"

hostname e.g., "192.168.2.162"

delete-all-nodes Delete all node records.

No parameters.

display-session Display all data for a given session.

targetname | null (default) to display all sessions hostname | null (default) to display all sessions

purge Delete all records.

No parameters.

Filtered volume commands

create-filtered-volume Create a new volume by adding a stack of block storage filters to an existing block device. The filtered device is returned.

input parameters and description

name a unique name to assign to the underlying target created

device the device to which to add the filters e.g., "/dev/sda"; default null

serial alternative identification of device, e.g., "0xfed70573ae21"

filters a list of filters, e.g., ["xor"]; default null

handler need not be specified - default "mp-filter-stack" Example response:

"/dev/sdb"

delete-filtered-volume Delete a volume created using create-filtered-volume().

input parameters and description

name the unique name assigned in create-filtered-volume

12.2 Terminology

SAN A storage area network which is used to transfer data from storage arrays to consumer nodes usually servers running a storage centric or compute centric application. SANs are characterised by a protocol such as Ethernet, Fibre Channel, Infiniband, a speed 40G, 10G, 1G for Ethernet or 4G, 8G or 16G for Fibre Channel.

Software Defined Storage (SDS) A technology which virtualises storage hardware and allows users of that storage to provision storage capacity with defined properties without knowing anything about the underlying hardware infrastructure. SDS uses two three core concepts A) A logical to physical mapping of the storage hardware and storage SAN B) An out of band automation process that receives requests for storage and provisions this storage from storage arrays C) An in-band software layer that provides to the consumer a virtual block device (VBD) that is mapped to a physical storage array volume (SAV).

SDS Gateway An SDS Gateway is a software controller that interfaces to an upper level API such as CINDER OpenStack and a lower level storage controller such as a storage array or an SDS controller.

Storage Array A physical device that manages disk media (Solid State or spinning disks) and exports that storage over a SAN connection.

Storage Array Volume A volume on a storage array exported on a SAN. SAVs are usually built on a RAID made up of several disks.

Virtual Block Device A Virtual block device is a device attached to a storage application which is mapped to a storage array volume. A VBD may be mapped to a single SAV or multiple SAVs. This mapping is part of the SDS in-band software layer.

Storage Filters Storage filters are software functions inserted by the SDS controller between the SAV and the VBD on a compute node. Filters can implement a wide range of functions specific to the application using the VBD.

References

- [1] FORTH, IBM, BSC, INTEL, NEUROCOM, "IOLanes FP7 EU Project 248615." <http://www.iolanes.eu/>.