HORIZON 2020 FRAMEWORK PROGRAMME

IOStack

(H2020-ICT-2014-7-1)

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

D2.2 IOStack Architecture Specifications and Benchmarking Framework

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

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

Summary of the document

Document Type	Deliverable
Dissemination level	Public
State	v1.0
Number of pages	48
WP/Task related to this document	WP2 / T2.2
WP/Task responsible	URV
Author(s)	Raúl Gracia-Tinedo, Edgar Zamora-Gómez, Pedro García-López, Marc Sánchez-Artigas, Ramon Nou, Marc Siquier, Yosef Moatti, Eran Rom, William Oppermann
Partner(s) Contributing	URV, BSC, IBM, MPS
Document ID	IOStack_D2.2_Public.pdf
Abstract	This document provides a comprehensive description of the architecture of IOStack. In particular, this document describes each of the building blocks that constitute the Software-Defined Storage (SDS) toolkit as well as their interactions within the architecture. Moreover, we show experimental results of IOStack that demonstrate the feasibility of the architectural design and the correctness and potential of the toolkit itself.
Keywords	Architecture, Design Principles, Benchmarking

Table of Contents

1 Executive summary

In this document, we describe in depth the technical aspects of the architecture of IOStack, as well as the framework employed to validate the correctness and performances of its components.

On the one hand, we describe the main elements of IOStack's architecture: *SDS Controller*, *Filter Framework*, *Dynamic Storage Policies Framework*, *Storage Monitoring* and *Compute Cluster Monitoring & Cross-layer Strategies*. First, the SDS Controller enables an administrator to manage the SDS system via a user-friendly web dashboard, natively integrated in the OpenStack dahsboard. Among other functionalities, the IOStack dashboard exposes the SDS Controller API to administrators, which is the gateway to orchestrate and manage the underlying SDS services in IOStack. In this document, we describe the SDS Controller API that enables managing SDS services for both object (Swift) and block storage (Cinder) systems.

In this sense, one of the main services accessible via the SDS Controller API is the IOStack filter framework. In IOStack, a storage filter can be defined as a *performance control or general-purpose data transformation* that applies to specific data flows (e.g., compression, IO bandwidth differentiation). Thus, the IOStack filter framework enables administrators to enforce, delete or modify the behavior of storage filters on a tenant's data flows, for instance. We also describe the filter framework for both object and block storage systems. Compared to existing SDS systems, IOStack is the first to enable the enforcement of general-purpose data transformations on data flows with a high degree of flexibility.

Furthermore, from an administrator's viewpoint, the feature that glues together the management of both block and object storage systems is the definition of *dynamic storage policies*. That is, the SDS Controller enables datacenter administrators to write storage policies, thanks to a simple yet powerful domain specific language (DSL). For instance, a datacenter administrator may define the use of a compression filter to tenant's `T1` data flows in case its write throughput goes below a certain threshold: `FOR T1 WHEN Throughput < 10MBps DO SET Compression`. The system will monitor the `Throughput` of tenants and it will enforce automatically a compression filter if the condition is satisfied, irrespective of the target storage system (block, object). As we argue in the document, this novel approach of managing general-purpose storage services has high potential.

Clearly, dynamic storage policies are impractical without a proper storage monitoring system. The last point of the SDS architecture of IOStack refers to a unified monitoring of storage metrics for both object and block storage systems based on CollectD and Grafana.

Similarly, we equipped the compute cluster with a monitoring system to understand the behavior and requirements of Big Data applications. Based on such monitoring information, we propose to enable direct cooperation between disaggregated compute and storage clusters via cross-layer scheduling and provisioning mechanisms. Currently, IOStack supports *i)* explicit[1] computation offloading to the storage system and, *ii)* a data locality service to enable co-location of compute instances (VMs, containers) on the storage servers where the required data resides. We argue that this type of cooperation opens the door for a variety of new optimizations regarding data analytics in the cloud.

On the other hand, we present the benchmarking framework that will validate the correct operation and quantify the performance of the SDS layer.

Our benchmarking framework consists of three main pillars: *Testing platform*, *synthetic benchmarks* and *trace replays*. The testing platform refers to the available hardware available in the IOStack project to deploy and test newly developed IOStack components. The first phase in the IOStack benchmarking lifecycle are synthetic benchmarks; that is, we have collected a battery of standard stress-test oriented tools to perform experiments with variable workloads in a controller manner.

Once the synthetic benchmark phase ends, we propose to benefit from our use-case partners to test IOStack under more realistic workloads. In particular, all use-case partners will provide academic partners with storage workload traces at different levels (block level, file system level). These traces, in turn, will provide academic partners with valuable opportunities of researching novel mechanisms for adapting IOStack to situations that cannot be observed in synthetic workloads. This

[1] "Explicit" means that the request to execute the filter is build by the client.

"virtuous circle" in the benchmarking lifecycle of the project will make the platform mature enough to be presented as a usable product prototype at the end of the project.

Finally, we present experiments of the IOStack prototype that certify the correct operation of some of its features. In particular, we show *ii)* the enforcement of storage filters in multi-tenant workloads, *ii)* the performance of the IO bandwidth differentiation filter, and *iii)* the advantages for GridPocket (IOStack use-case) of pushing-down simple computations to the storage cluster from the compute cluster in the data analytics process.

2 IOStack: Motivation, Goals and Building Blocks

Nowadays, the amount of Big Data stored in cloud storage services is growing at unprecedented rates, as well as the variety and heterogeneity of workloads supported by datacenter infrastructures. At the same time, datacenter administrators should respond with increasing agility to changing business demands in a cost-effective manner, which is cumbersome due to the complexity of large cloud environments.

Software-Defined Storage (SDS) has recently become a prime candidate to simplify storage management in the cloud. To ease the work of datacenter administrators, the incipient literature in the field defines that SDS should provide a storage infrastructure with *ii) storage automation*, *ii) optimization*, and *iii) policy-based provisioning* [1, 2]. Normally, this is achieved by explicitly decoupling control and data planes at the storage layer.

Storage automation enables easy provisioning of resources to tenants from a datacenter administrator's viewpoint. This includes the virtualization of storage services (volumes, file-systems) on top of performance-specific servers and network fabrics orchestrated by the SDS system. Storage optimization may be seen as an advanced and dynamic case of storage automation where resources are automatically allocated to the most suitable workloads, dealing with potential heterogeneity across both workloads and resources [2].

Moreover, SDS enables the enforcement of policies to data flows for providing performance control and value-added services to the storage system [1]. This includes, for instance, the application of data reduction services, computations or IO bandwidth limits on a tenant's storage requests [3].

At the storage layer, the objective of IOStack is to enable storage automation and optimization based on Software-Defined Storage (SDS) models, while simplifying the administration of the storage system via policy-based provisioning. To better understand our goals, let us draw an example of a multi-tenant Big Data scenario.

2.1 A Motivating Example

Imagine an object store and 3 different tenants that access it concurrently. On the one hand, tenant *T1* represents several servers that are uploading data gathered from a sensor network. On the other hand, tenants *T2* and *T3* represent sets of virtual machines in a compute cluster that perform computations on data objects containing logs. This is shown in Fig. 1.

In such scenario, a datacenter administrator may wish to enforce distinct *policies* to these tenants for optimizing the system's operation. Intuitively, he could apply a data compression policy to *T1* for reducing its storage space demands, given that log-like data is potentially redundant [4]. Tenants *T2* and *T3*, however, may apply data filters to import only the fraction of a dataset actually needed for a specific computation task, thus reducing download traffic [5].

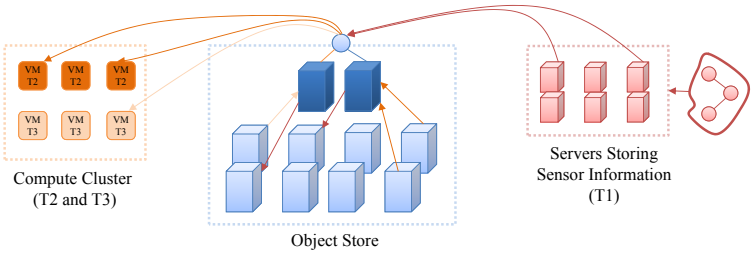

Figure 1: Example of an OpenStack Swift deployment (proxy nodes in dark blue, storage nodes in light blue) concurrently accessed by various tenants. Storage policies may be enforced on object requests to optimize the system and enrich the service.

Naturally, the enforcement of these policies may permit an object store to manage concurrent workloads more efficiently. However, today's object stores are lacking from a flexible and transparent way of enforcing storage policies on object requests. This is precisely the objective of IOStack.

2.2 Mission of IOStack

The previous example opens the door to apply *storage optimizations* under multi-tenant workloads [2], as well as to offer *different Quality-of-Service (QoS) policies* based on a tenant's requirements. Moreover, from a datacenter administrator's perspective, these goals need to be achieved transparently, involving minimal human intervention. To realize this vision, IOStack[2] features:

- **Policy-based provisioning**: In IOStack, datacenter administrators simply assign provisioning policies to tenants. For example, a *static policy* may be defined to enforce compression (e.g., `gzip`) on tenant *T1*'s requests as {Compression, [gzip]} ⇒ *T1*. As we will see, after establishing that policy IOStack transparently applies data compression on *T1*'s requests. Moreover, IOStack provides *dynamic policies* with monitoring information in order to change a tenant's QoS either on demand or by a workload-based decision.
- **Filters**: A *filter* represents the actual logic executed on an storage request to enforce a policy. IOStack has a suitable *architecture to favor the integration of new filters* by third-parties that increase the value and functionality of OpenStack storage systems (block, object). IOStack also includes a ready-to-use filter framework that enables the *execution of user code on storage requests* at different stages along a request's write/read path. Thus, a developer integrating a new filter only needs to contribute the filter's logic; the deployment and execution of the filter is managed by IOStack.

To achieve these high-level goals, there are several software building blocks involved that are responsible for different tasks. Besides, it should be noted that these building blocks may differ depending on the target storage system: *object and block storage*. In the following, we overview the main building blocks that will constitute the storage architecture of IOStack[3].

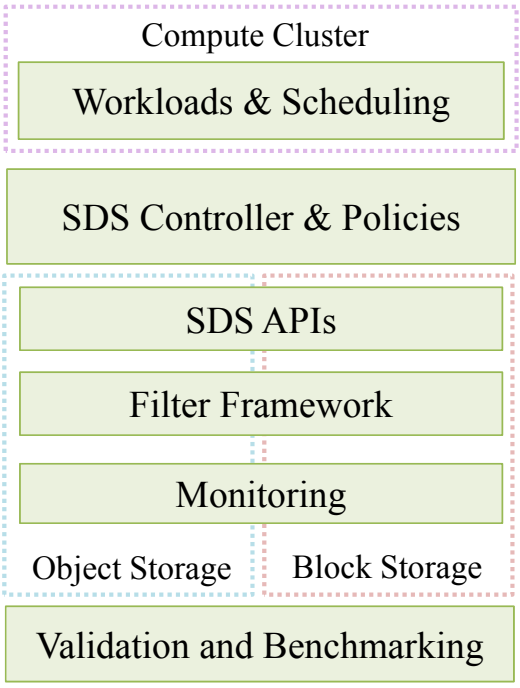

Figure 2: Overview of IOStack architectural building blocks described in this document.

[2] `http://iostack.eu`
[3] `https://github.com/iostackproject`

2.3 IOStack Building Blocks

As visible in Fig. 2, the IOStack storage architecture can be divided in five main components: *SDS Controller*, *Filter Framework*, *Dynamic Storage Policies*, *Storage Monitoring* and *Compute Cluster*. Moreover, we include a *Validation and Benchmarking Framework* that provides methodologies to exercise and analyze their correctness and performance of all these components. We briefly describe each component as follows:

SDS Controller: The SDS Controller represents the control plane of IOStack[4,5]. The SDS Controller is also the contact point of a datacenter administrator with IOStack, enabling simplified management of the storage layer[6]. The SDS controller gives direct access to the SDS Controller API, which abstracts the available services that the underlying SDS system offers (see Section 3). Among these services, we include the management of filters, the provisioning of performance-specific storage resources or the access to real-time monitoring metrics. Naturally, the SDS storage APIs vary depending on the targeted storage system (block/object), and may evolve during the development of the project. The current IOStack APIs are explained in Section 3.2.

Filter Framework: In IOStack, filters enable arbitrary transformation on data flows. These transformations may be enforced at one or various stages through the write/read path, based on a storage policy defined in the SDS Controller. In Sections 4 and 5, we describe the IOStack filter framework for both object[7] and block storage[8], respectively. Furthermore, we describe the currently available filters in IOStack, including an IO bandwidth differentiation filter[9] and a SQL push-down filter for Spark that enables to explicitly offload specific computations to the storage cluster (see Section 4.1.3).

Dynamic Storage Policies: A salient feature of IOStack is that it enables administrators to define dynamic storage policies. Essentially, these storage policies trigger a filter (or a set of filters) based on a live workload-based decision. To this end, the SDS Controller offers a Domain Specific Language (DSL) that makes it easy to manage and optimize —statically or dynamically— the storage system, depending on the existing workloads. Also, the SDS Controller unifies the management of IOStack for object and block storage. The dynamic policy framework is depicted in detail in Section 6.

Storage Monitoring: Naturally, dynamic storage policies need from monitoring information for triggering a filter (or a set of filters) based on a live workload-based decision. Section 7 describes the IOStack monitoring system used to track the activity of both block and object storage systems[10].

Compute Cluster: Monitoring, Analysis and Cross-layer Strategies: IOStack will enable disaggregated compute and storage clusters to cooperate for optimizing Big Data analytics in the cloud. To this end, IOStack currently provides explicit computation off-loading to the storage and a data locality service to co-locate compute instances (VMs, Containers) with the required data at the storage cluster. As we describe in Section 8, such a cross-layer scheduling and provisioning strategies may improve data analytics in the cloud, where storage and compute clusters are disaggregated.

Validation and Benchmarking Framework: To evaluate the progress and correctness of the development of IOStack, we provide a set of testing and benchmarking methodologies. These methodologies include benchmarking tools[11], testing platforms, trace replays and experiments based on use case workloads. Moreover, we also show a battery of experiments to validate some IOStack components. We describe our benchmarking framework in Section 9.

In the following, we describe the architecture and operation of IOStack's SDS Controller.

[4]Source code available at: `https://github.com/iostackproject/SDS-Controller-for-Object-Storage`
[5]Source code available at: `https://github.com/iostackproject/sds_block_api`
[6]Source code available at: `https://github.com/iostackproject/SDS-dashboard`
[7]Source code available at: `https://github.com/iostackproject/SDS-Storlet-Middleware`
[8]Source code available at: `https://github.com/iostackproject/Konnector`
[9]Source code available at: `https://github.com/iostackproject/IO-Bandwidth-Differentiation`
[10]Source code available at: `https://github.com/iostackproject/SDS-Storage-Monitoring-Swift`
[11]Source code available at: `https://github.com/iostackproject/SDGen`

3 SDS Controller

A central element of IOStack's architecture is the SDS Controller as it constitutes the control plane of the SDS system. The SDS Controller is not a monolithic entity, but a framework itself that is designed to accommodate an arbitrary number of filters. In what follows, we describe the design of the SDS Controller and its integration with the rest of building blocks in IOStack. Specifically, the SDS Controller is constituted by: *IOStack Dashboard*, *SDS Controller API* and the *Metadata Store*.

3.1 IOStack Administration Dashboard

The first contact point between a datacenter administrator and IOStack is the *dashboard*. The IOStack dashboard is a web front-end that enables administrators to easily manage the SDS framework via a user-friendly GUI (graphical user interface). In particular, the IOStack dashboard is integrated in the OpenStack dashboard (Horizon[12]); this decision enables administrators to benefit from a centralized administration point for the entire system with a familiar GUI. The main objective of the IOStack dashboard is to provide administrators with a simple way of accessing the underlying SDS Controller API (see Section 3.2) for managing the system, as well as to provide information of the current state of the storage and compute clusters.

To this end, the IOStack dashboard has 3 main administration areas: *object storage*, *block storage* and *system monitoring*. On the one hand, both object storage and block storage administration areas refer to the IOStack capabilities for managing OpenStack Swift and OpenStack Cinder, respectively. We separate the administration areas of both systems due to their inherent differences, which are reflected also in the SDS Controller API, as we show later on. All in all, in both areas there are common elements, such as the operations to define policies and enforce storage filters.

On the other hand, the system monitoring area of IOStack's dashboard provides real-time information of the state of the storage system and the compute cluster. This is important for an administrator in order to understand the state of both compute and storage clusters at any moment for reacting accordingly. In essence, this area provides a variety of real-time plots based on CollectD[13] and Grafana[14] that consume monitoring events from both the storage and compute cluster monitoring systems. While the monitoring of the storage cluster is further discussed in Section 7, the compute cluster monitoring is extensively described in deliverable 5.1.

Thus, one of the essential goals of the IOStack dashboard it to offer a user-friendly way of managing the internal functionality of the SDS system, which exposed as an API. Next, we describe the goals of the SDS Controller API in IOStack.

3.2 Understanding the SDS Controller API

The SDS Controller API is the gateway for managing the underlying SDS services in IOStack. From a datacenter's administrator viewpoint, the SDS Controller API provides the opportunity to orchestrate provisioning policies with a simple Representational State Transfer (REST) API calls[15]. This standard technology makes the SDS Controller API easy to manage from Web or other types of clients.

To better understand the role of the SDS Controller in the IOStack architecture, let us retake the example proposed in Fig. 1. In that example, the administrator wants to apply a data compression policy to `T1` in order to save storage space. Thus, the administrator sends a simple HTTP request to the SDS Controller such as:

```
http://sds-controller/filters/T1/deploy/compression
Body:  {"engine":"gzip"}
```

When the SDS Controller receives that request, it automatically forwards the request to the appropriate filter subscribed in the system (compression filter). Then, the filter's policy manager should handle the request and persist the associated changes in the IOStack metadata store. Thus, once this

[12] `http://docs.openstack.org/developer/horizon`
[13] `http://collectd.ord`
[14] `http://grafana.org`
[15] `https://en.wikipedia.org/wiki/Representational_state_transfer`

change is propagated to the filter's data plane, every data object stored/retrieved by tenant `T1` will be (de)compressed using `gzip` compression engine.

Overall, we point out that the datacenter administrator only needs to send a simple request to the SDS Controller in order to transparently change the behavior of the storage system. Furthermore, thanks to our dynamic policy framework (see Section 6), most calls to the SDS Controller API could be automatically made by the system itself, depending on the workload at hand.

Naturally, the IOStack API is an evolving entity as it is intended to federate as many SDS services as possible to enrich the functionality of the storage layer. Next, we describe the main services already integrated in the SDS Controller API that enable datacenter administrators to manage the SDS layer in IOStack (see Fig. 3).

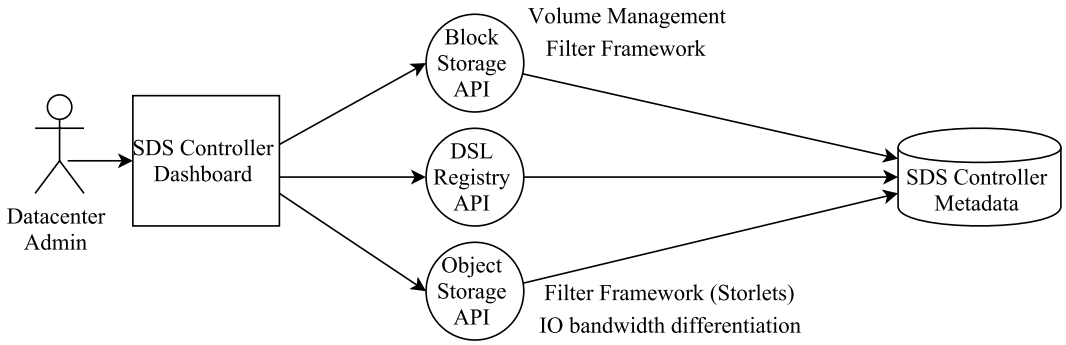

Figure 3: Interaction of a datacenter administration with the different modules of IOStack's SDS Controller API. Moreover, the actions performed against the SDS Controller API reflect changes in the SDS metadata sore.

SDS Controller API for Object Storage: In the first stages of the project, the consortium has extensively focused their efforts on providing a rich filter framework for OpenStack Swift. While technically challenging, the enforcement of filters on data flows will provide unique opportunities for optimizing multi-tenant Big Data workloads in the storage side. Specifically, the SDS Controller API for object storage provides a battery of calls for managing a *filter framework based on Storlets* and a *IO bandwidth differentiation* filter.

First, as we illustrate in Section 4, IOStack provides a general-purpose filter framework based on IBM Storlets [6]. Any developer may create new filters that can be executed on particular object request to provide data management or computation services, to name a few. In this sense, the SDS Controller API for object storage provides a complete set of calls to perform the deployment and management of Storlet filters (see Table 4.1a).

However, despite that our filter framework for object storage is very flexible, there are types of filters that are difficult to implement efficiently via Storlets. For this reason, IOStack also enables third party filters to be plugged-in as independent components of the architecture and managed via a specific API. This is the case when achieving IO differentiated bandwidth in a multi-tenant scenario; such a filter requires low-level control of the underlying storage devices (disks, SSDs) in order to provide fine-grained, high-performance IO bandwidth differentiation to various tenants. In Section 4.2 we provide the available calls for this filter, and experimental results are presented in Section 9.4.

Moreover, in the next stages of the project, this API will be expanded with calls to automate the provisioning of *performance-specific containers and/or accounts* to customers. To wit, we will exploit the already existing *differentiated rings* in OpenStack Swift to help administrators managing different types of data redundancy strategies (e.g., replication, erasure codes).

SDS Controller API for Block Storage. Regarding the API for block storage, we have focused on two main points: *ii)block volume provisioning* and *ii) filter framework*[16]. In IOStack, block volumes

[16]Extensive details for the IOStack block storage subsystem can be found in Deliverable D3.1.

are provisioned via a REST API in the SDS controller, so that administrators can easily provision block volumes of definable characteristics and fabrics. The provisioning of block volumes is done via advanced virtualization technologies and this mechanism is exposed and integrated in the IOStack dashboard. The API for volume provisioning is depicted in Section 5.2.

Moreover, the block volume subsystem of IOStack has a filter framework, analogous to the object storage one. The filter framework is based on LIO[17] and it essentially offers a client-side stage where external code, namely *filters*, can be executed on volume IOs. We describe the API to manage the filter framework for block storage in Section 5.3.

DSL Registry: As we describe in Section 6, we introduce in IOStack a new DSL language to enable administrators defining policies that will trigger the enforcement of storage filters under certain workload conditions. To keep the definition of policies as simple as possible, in our DSL syntax we enable administrators to refer with simple keywords, such as `Compression` to storage filters or other elements within the framework. Similarly, an administrator may want to enforce the same policy to a group of tenants, which yields that the management of tenant groups is important for defining policies. All this functionality is also managed via a REST API to a specific module called DSL Registry. We describe the DSL Registry API (Table 6.3a) and the technical details of it in Section 6.3.

The federation of APIs from the available SDS services' constitute our SDS Controller API. Clearly, a primary objective of most of these calls is to change metadata at the SDS layer (e.g., set the enforcement of a filter to a tenant). Therefore, the SDS metadata store is a fundamental part of the IOStack architecture. Next, we describe the objectives and implementation of the metadata store in IOStack.

3.3 SDS Controller Metadata Store

All the metadata related to IOStack is persistently stored in the metada store, which resembles the traditional concept of configuration management database (CMDB). Currently, the IOStack metadata store contains **metadata of policies** and **metadata of filters**. On the one hand, *metadata of policies* refers to the relationship of tenants and filters, and the conditions under a certain filter should be applied to a tenant. On the other hand, *metadata of filters* is the necessary information for the deployment and correct operation of a filter within the storage system.

To design the metadata store, we targeted three main requirements: *performance*, *scalability* and *flexibility*. We justify these requirements as follows.

First, as we will describe later on, the enforcement of filters is transparent. That is, once a policy has been defined, the system infers which filters (if any) would be applied upon the arrival of storage operation at a certain moment. Naturally, the metadata store will need to serve a potentially *high number of requests* in this regard, as it stores the information of policies. Moreover, it is worth noting that the metadata store will receive a *read-only workload* as most queries will require reading metadata items. Thus, creating and/or modifying metadata items, as introducing new policies in the system, is likely to be a marginal part of the metadata activity.

Second, the metadata store should be easy to *scale and work in a distributed fashion*. That is, depending on the implementation of a filter, a potentially high number of machines may need to retrieve metadata from the system. Thus, scaling-out metadata to an arbitrary number of servers can *balance metadata requests* naturally, and in turn, improving *query latencies and availability*.

Moreover, the general-purpose filter framework of IOStack yields that the metadata of filters can be *very heterogeneous*. For instance, as we describe further in this document, the number of parameters needed to configure a compression filter *may be quite different* from the configuration of a bandwidth differentiation filter. This was one of the main reasons to discard the use of traditional database technologies, as it is complex to achieve such a degree of flexibility making use of SQL tables.

Considering these requirements, we resorted to an **in-memory data structure store** as the metadata layer of IOStack. In particular, we make use of `Redis`[18]: a high-performance open-source in-memory store. `Redis` provides advanced and high performance caching techniques, which fits per-

[17] `https://en.wikipedia.org/wiki/LIO_(SCSI_target)`.

[18] `http://redis.io`

fectly the read-only workload expected in the IOStack SDS framework. Moreover, `Redis` enables *generic objects* to be stored in memory, as they are abstracted via serialization techniques into byte-level representations. `Redis` also provides developers with a large collection of complex data structures (e.g., maps, lists) in order to facilitate programming with metadata objects.

Due to its distributed nature, `Redis` enables a variety of deployment strategies depending on the target scenario. For instance, we might deploy the metadata store in an specialized cluster (i.e., separated from the storage servers), or we might benefit from the existing storage infrastructure by deploying `Redis` on a subset of storage servers (e.g., proxy nodes in OpenStack Swift).

One of the main goals of the SDS Controller is to enable the execution of a storage filter in IOStack. In what follows, we describe the IOStack filter framework for both object and block storage, as well as the currently available filters.

4 Filters in Object Storage

In this section, we describe the filters that are currently available in IOStack for OpenStack Swift (object storage). First, we present our filter framework for object storage based on IBM Storlets; this framework enables administrators to deploy and execute general-purpose code on specific object requests. Based on this framework, we describe a battery of filters already implemented and tested in Section 9.4 (data reduction, compute-close-to-data).

However, despite their flexibility, it is difficult to efficiently implement all types of filters making use of IBM Storlets. That is, there are some types of filters that may need specific, low-level requirements and control of the underlying storage infrastructure to be enforced in a fine-grained, high performance manner. This is the case when providing differentiated IO bandwidth QoS to tenants. Such a filter requires having a "physical" notion of the IO bandwidth being consumed at a certain moment at any storage node, as well as the available storage nodes that may contain copies of currently requested object to take real-time scheduling decisions. Therefore, since IO bandwidth differentiation is a primary feature of SDS system, in IOStack we provide this service as a separate filter component integrated in the IOStack architecture.

4.1 Filter Framework with Storlets

To easily design and deploy a wide variety of filters, IOStack provides its own *filter framework* that enables developers to run general-purpose code on object requests. In a sense, IOStack borrows ideas from active storage literature [7, 8] as a mean of building filters to enforce policies.

4.1.1 IBM Storlets

The core of IOStack's filter framework is based on IBM Storlets [6, 9]. To enable the use of Storlets, the storage system needs to be augmented with a *Storlet Engine* that provides the storage cloud with capability to run Storlets in a sandbox that insulates the Storlet from the rest of the system, other Storlets and for a given Storlet from other tenants. The Storlet Engine provides a powerful extension mechanism to the cloud storage without changing its code; thus making the storage flexible, customizable and extensible. It expands the storage system's capability from only storing data to directly producing value from the data.

In IOStack, we make use of the Storlet Engine for OpenStack Swift Object Storage. Storlets extend OpenStack Swift with the capability to run computations near the data in a secure and isolated manner making use of Docker[19] as application container. Anyway, the Storlet Engine architecture is generic with respect to the cloud object storage system. It includes two main services:

- *Extension Service*: connects to the object storage and evaluates whether a Storlet should be triggered and in which storage node.
- *Execution Service*: deploys and executes Storlets in a secure manner.

The extension service is tied to the storage system at interception points and identifies requests for Storlets. In particular, it examines the metadata fields in the request and by evaluating the defined

[19] `https://www.docker.com/`

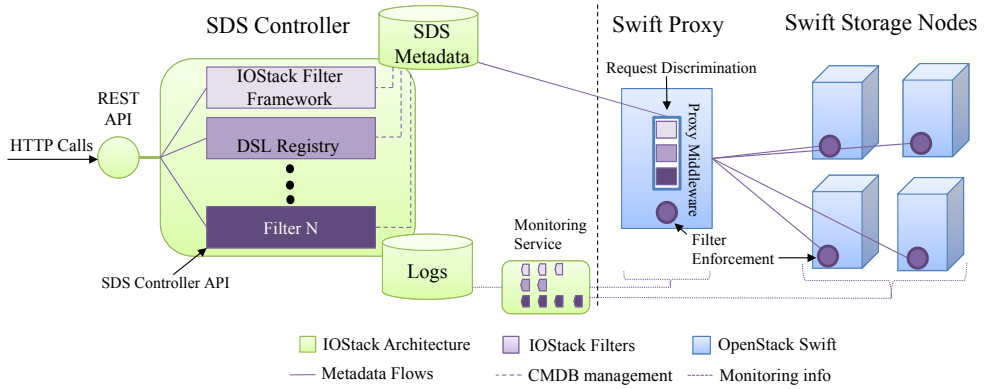

Figure 4: Architecture of IOStack filter framework for object storage. Via the API, the administrator defines the enforcement of a filter on a tenant's requests and this decision is persisted in the IOStack metadata store. Then, upon the arrival of a new object request, a IOStack module in the Swift proxy checks the metadata store and infers whether the filter should be enforce or not. Depending on its configuration, the filter enforcement/execution will take place either at the proxy or the storage node.

Storlets to be executed. The extension service then communicates with one of the execution services that executes the Storlet in a sandbox according to the client request and its credentials. The only component that will change when implementing the Storlet Engine in one cloud storage or another is the intercept handler of the extension service. The intercept handler needs to adapt the hooks to the object storage and may need for that either the source code or an open interface to internally connect to cloud storage.

The Storlet Engine can reside in the storage interface node (i.e., proxy node), and in the storage local node (i.e., storage node). Intercepting requests in both proxy and storage node stages is possible thanks to the WSGI middleware integrated in OpenStack Swift[20] that can be used to "wrap" the storage request and response of a Python WSGI application (i.e. a webapp, or REST/HTTP API), like Swift's WSGI servers (proxy-server, account-server, container-server, object-server). In fact, Swift uses middleware to add (sometimes optional) behaviors to the Swift WSGI servers.

Naturally, executing an Storlet either at the proxy or storage node stage has implications. That is, the Swift proxy service is responsible for the external interface. It is generally more CPU and network intensive; thus, the proxy nodes that host those proxy services have generally powerful CPUs and large network bandwidth. The Swift object service holds the actual data objects. The object services are generally more disk and I/O intensive; thus, the storage nodes that host those object services have generally large disks and good I/O throughput. In IOStack, Storlets can run either in the interface proxy servers or in the local object servers to take advantages of each node underutilized resources. As we show in Section 9.4, depending on whether the task to be done is more compute intensive or IO intensive, it is preferable to execute Storlets in the proxy or storage nodes, respectively.

The Storlet Engine supports a *synchronous operation mode*. That is, the Storlet runs within the HTTP request/response that initiated its execution, namely the HTTP request ends after the Storlet ends its execution. For this reason, Storlets are specifically tailored for data management and streaming operations on object requests.

Moreover, a Storlets Maketplace can be used as a repository of Storlets from different vendors. An application on top of the storage can mashup and use different Storlets from the marketplace for creating its functionality. In IOStack, this marketplace is managed by the datacenter administrator under stringent security requirements. In this scenario, the same Storlet code can be used to enforce filters in various tenants, drastically reducing the effort on filter development.

[20] `http://docs.openstack.org/developer/swift/development_middleware.html`

REST Call Description	HTTP Method	URL
Create a filter	POST	`/filters`
List filters	GET	`/filters`
Delete a filter	DELETE	`/filters/:filter_id`
Get filter metadata	GET	`/filters/:filter_id`
Update filter medatada	PUT	`/filters/:filter_id`
Upload a filter's logic	PUT	`/filters/:filter_id/data`
List deployed filters of an account	GET	`/filters/:account_id/deploy`
Deploy a filter	PUT	`/filters/:account_id/deploy/:filter_id`
Undeploy a filter	PUT	`/filters/:account_id/undeploy/:filter_id`
Create a dependency	POST	`/filters/dependencies`
List dependencies	GET	`/filters/dependencies`
Delete a dependency	DELETE	`/filters/dependencies/:dependency_id`
Get dependency metadata	GET	`/filters/dependencies/:dependency_id`
Update dependency metadata	PUT	`/filters/dependencies/:dependency_id`
Upload dependency data	PUT	`/filters/dependencies/:dependency_id/data`
List deployed dependencies of an account	GET	`/filters/dependencies/:account_id/deploy`
Deploy a dependency	PUT	`/filters/dependencies/:account_id/deploy/:dependency_id`
Undeploy a filter	PUT	`/filters/dependencies/:account_id/undeploy/:dependency_id`

Table 4.1a: SDS Controller API for the filter framework for object storage. Regarding our filter framework for object storage, filters are implemented as Storlets.

4.1.2 Storlet Orchestration and Management

As previously described, IBM Storlets provide the ground to support low-level data transformations on object requests (i.e., filters). In IOStack, we aim at intelligently managing and orchestrating Storlets to leverage high-level SDS services. The architecture of the IOStack filter framework based on Storlets is depicted on Fig. 4. In particular, it consists of 3 main components:

Storlet manager: The objective of the Storlet manager module is to provide a management interface for *Storlets code and their execution* on particular requests in the SDS Controller. Concretely, this task is done via a REST API integrated in the SDS Controller (see Table 4.1a).

Regarding the management of Storlet code, in Table 4.1a there two types of calls: *filter* and *dependency* calls. Filter calls refer to those operations that manage the execution code of Storlets, such as `.jar` files that offer compression or caching services. Moreover, Storlets may need external libraries to operate, which are called *dependencies* in our terminology. For this reason, the second type of calls are devoted to manage the dependencies that each Storlet need for their correct execution. As can be observed, the API provided in the strolet manager enables an administrator to easily upload new Storlet code to the system, specially via the IOStack dashboard.

Apart from managing the code and dependencies of Storlets, the most common task of the Storlet manager module is to enable and administrator to *define the enforcement of filters on a particular tenant or account*. For this reason, we provide in the API 3 calls to enable a datacenter administrator the management of filter/tenant relationships: `Deploy`, `Undeploy` and `List` filters for a particular tenant. As we describe in Section 6, IOStack will also make use of these calls to automatically trigger the enforcement of filters under certain workload conditions.

Request classification: In IOStack, Storlets can be triggered either *explicitly* or *implicitly*, depending on whether the HTTP request to execute the Storlet is built by the client or by the SDS layer, respectively. Since the explicit execution is interesting in some scenarios (see Section 4.1.3), it does not require any additional software component. That is, a tenant can execute a REST API with the appropriate headers to execute a Storlet. Conversely, to make the execution of Storlets implicit or transparent from the client's viewpoint requires additional efforts.

To this end, our filter framework discriminates the filters to be applied on a particular data flow at the Swift's proxy (see Fig. 4, Proxy Middleware). Technically, an IOStack module in the Swift proxy middleware contacts the IOStack metadata store *to infer the filters to be executed* on a tenant's request. Naturally, the module should query the metadata store for each request[21].

[21] It should be noted that the IOStack metadata store is a distributed in-memory store that can be deployed on the Swift

After performing the query to the metadata store, the Storlet manager may need to enforce the execution of a certain filter on that request. To this end, based on the applicable filters to that tenant, the Storlet manager sets the appropriate HTTP headers to the incoming request in order to trigger the subsequent Storlet execution.

Moreover, we store data objects with an *extended metadata* to keep track of the filters enforced on an object, and the execution order of filters. This is necessary to trigger inverse transformations of filters that change the content of objects (e.g., compression, encryption).

Sandboxed filter execution: Upon the arrival of a tenant's request with the appropriate HTTP headers, a filter can then be executed on a Docker instance for that tenant. Moreover, the execution can take place either at the proxy or storage node stages; such a decision that depends on the Storlet configuration. For instance, a compression filter can efficiently performed at the proxy, whereas compute tasks on data objects may be more suitable at the storage node. For Static Large Objects (SLO), Storlets can only be run the Storlet at the proxy node since the data of the object is disseminated into multiple data nodes.

In the first version of IOStack filter framework for object storage, we can enforce only one filter per tenant. That is, the Storlet framework is not yet capable of pipelining several Storlets on the same object request. This feature is expected to be integrated in IOStack in the second year of the project.

Overall, one of the novelties of our this framework is its high flexibility. As we show next, the IOStack filter framework can support many filter types of filters, such as data reduction, storage optimization and general computations on data objects.

4.1.3 Available Data Management Filters in IOStack

Developing storage filters as Storlets is specially suitable for data management purposes. Next, we describe the current filters that are already working in the object storage subsystem of IOStack.

Compression filter: Compression is, perhaps, one of the most intuitive data management filters that one can devise. Our compression filter is developed as an Storlet and integrates various compression engines. At the moment, our compression filter can work with `gzip` and `lz4` compressors, which represent distinct points in the time/compression ratio trade-off. Other compressors can be easily added as third party libraries. Furthermore, our filter also allows to parametrize the degree of compression level used by these compression engines (e.g., `gzip-6` or `gzip-9`).

Spark SQL push-down filter: As we describe later on (Section 8), the SQL Spark pushdown mechanism is an example of cooperation between disaggregated storage and compute clusters. In particular, it enables delegating to the storage cluster (Swift) part of Spark SQL queries. Despite the concept of offloading SQL to the storage is general to any data type, the current implementation is specifically tailored to work with comma separated files (CSV), which are very common in Big Data workloads.

In this sense, the "push down" functionality refers to the transparent communication between Spark and Swift to import CSV files from the storage cluster to the compute cluster after execution SQL statements, such as *column selection and row filtering*.

To make use of the SQL mechanism, there are several requirements: *ii)* The Spark compute cluster is connected to a Swift object store, *ii)* Spark jobs of interest issue SQL queries against one or more file in CSV format. The queries may either be issued directly by the user (in a spark-shell for instance) or indirectly through Spark libraries such as Machine Learning libraries; *iii)* The IBM Storlet framework augments the Swift object store functionality[22].

The design of the SQL pushdown mechanism is logically divided into two parts: first, the modification of the *Spark SQL query interpreter* to implicitly offload SQL task to Swift. Second, *a new Storlet for IOStack* specifically designed to efficiently process CSV files.

Regarding the modification of Spark (the compute side), we modified the Spark SQL query interpreter to treat a SQL statement and identify parts of it that can be pushed-down to the storage.

proxies themselves. This yields that the cost of accessing the metadata store from a proxy's viewpoint is low (i.e., access to local memory).

[22]See details at `https://github.com/openstack/storlets`.

This yields to change the source code of the Spark interpreter, which is itself challenging due to the complexity of the framework. Thus, in our modified Spark SQL interpreter, a sentence like:

```
SELECT user_id, age FROM user_table WHERE age>18
```

In this simple example, the role of our Spark SQL pushdown mechanism is very important. That is, the interpreter detects that both column selection (`SELECT user_id, age`) and row filtering (`WHERE age>18`) are susceptible to be delegated to the storage cluster. Thus, the modified interpreter extracts both selections and projections and propagates them as parameters to the various Spark tasks.

The SQL projections and selections propagated to the Spark tasks will be included as *metadata to the HTTP requests headers of CSV files*, to import the necessary data to execute the targeted computation. Such extended metadata is the *communication channel* to offload SQL tasks from Spark to Swift, and it basically dictates the functionality to be pushed down to the Storlet.

Technically, the extended metadata is formed by two parameters (possibly empty) in form of strings which will be concatenated to the URI of the targeted object. The first one describes the *indexes of the requested columns*. If empty then no column selection is to be done, that is all returned rows will contain all the fields. The second one describes the parts of the `WHERE` clause of the SQL query that can be pushed down.

Therefore, these 2 metadata parameters are propagated to all the Spark tasks so that each of them, assuming at least one of the two strings is not empty, will modify the Swift GET request to invoke the CSV Storlet to offload particular SQL tasks.

At the storage side, a new Storlet, namely *CSVStorlet*, has been implemented to intercept the Swift `GET` HTTP request and implements the push down functionality set at the Swift side parameters passed from Spark to the Storlet.

We tried to build on open source technologies to implement the pushed down functionality. Concretely, We use the `univocity`[23] open source CSV parser to process the CSV file and perform the column projection. The same package is also used by the spark-csv project so that we align the Storlet code with what is done when no pushdown is done. In a similar manner we used in the CSV storlet, parquet libraries used in Spark for evaluating the `WHERE` clause of the SQL query.

As we show in Section 9.4, this mechanism improve the performance of Big Data analytics by making the storage cluster to cooperate in compute tasks. Finally, this functionality is currently being tailored to improve the data analytics tasks of GridPocket, a IOStack use case company.

4.2 IO Bandwidth Differentiation Filter

Swift offers to the user a rich REST API to manage objects, and manages replicas (or erasure codes) to add resilience to the system. The basic interface with applications are the GET and PUT methods to retrieve and store objects, respectively. The user sends a GET requests to the Swift proxy and the request is directed to any of the Object Storage nodes (randomly by default). Once the object server has the request, an iterator is returned and offered to the user. The iterator offers to the user chunks of data, on demand, issuing read operations to the operating system via the *DiskFileReader* class.

Such requests are issued by different threads (round-robin), and even if the object is still the same (and the requests are sequential) the operating system sees them as different applications are asking for the same file producing fairness and disabling prefetching problems.

This aspect is important for the I/O scheduler. One of the most advanced ones, CFQ [10], includes heuristics and different queues to manage different applications and requests, and the original Swift behaviour breaks the optimal performance of CFQ. Although it does not have a major impact on SSD devices due to their non-sequential performance (and non-specialized schedulers), HDD devices suffer from a high penalty due to seek times and reduced burstiness.

4.2.1 Swift Analysis

To see what is happening on Swift, we used a SAIO [11] (All-in-One) installation that avoids a lot of complexity and prepares a controllable environment to experiment with Swift. We modified the configuration files to have a single object storage node, and remove all the replication and resilience

[23] http://www.univocity.com/

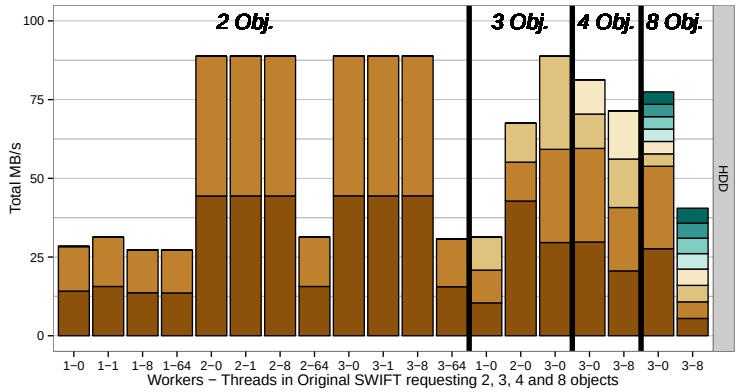

Figure 5: Performance obtained using the original *Swift* with different workers and threads numbers and with requests for 2, 3, 4 and 8 big objects. 0 Threads mean synchronous operations inside a worker.

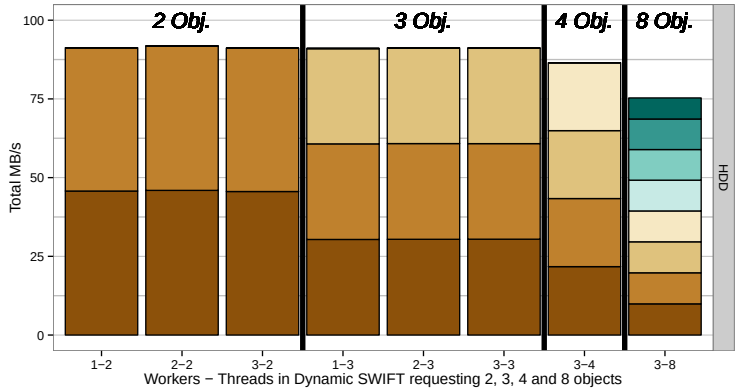

Figure 6: Performance obtained using the dynamic Swift with 1 thread per each object and requesting 2, 3, 4 and 8 objects with different workers values.

features to reduce I/O interferences.

We stored a set of 5 GB files inside the object storage and started asking from different clients simultaneously. The operating system behavior was analyzed using blktrace [12] and Paraver [13] using a translator called blktrace2paraver [14].

With such tools we see what happens inside the system (reads, writes, merges, idle times) in a two-dimensional view: Time in x-axis and threads in the y-axis. With Paraver we can easily track if a single thread is asking the same objects in a sequential way, or for instance, different threads are asking different regions without a logical rule on the same object.

Swift is configured at boot with two main parameters: The number of workers that serve each server (TCP request level) and the number of threads per worker. The workers are created in the WSGI [15] layer for each client request, then the request is served (I/O operations) inside the object server using n threads.

We explored multiple clients (or tenants) requesting different objects, with different Swift parameters (thread / worker combinations) to see their effect on the performance and the disk behavior. We used HDDs. We can see a summary of the results in Figure 5 with requests from 2, 3, 4 and 8 clients.

The Figure 5 has the y-axis showing the bandwidth obtained on each of the scenarios represented by the bars on the x-axis (# Workers - # Threads) and the bars are divided in different colors to identify the individual bandwidth obtained from each object. Bars' division helps to detect fairness

REST Call Description	HTTP Method	URL
IO bandwidth info	GET	`/bw`
IO Object Servers info (ip:port, devices, thread identifiers, oid, accounts, policies, MB/s served and MB/s asked)	GET	`/bw/osinfo`
Get IO bandwidth info of an account	GET	`/bw/:account_id`
Clear all the IO bandwidth assignments for all accounts and policies	PUT	`/bw/clear`
Clear all the IO bandwidth assignments entries for the selected account	PUT	`/bw/clear/:account_id`
Clear all the IO bandwidth assignments entries for the selected account and policy	PUT	`/bw/clear/:account_id/:policy_id`
Assign IO bandwidth to all the policies of the selected account	PUT	`/bw/:account_id/:bw_value`
Assign IO bandwidth to the selected account and policy	PUT	`/bw/:account_id/:policy_id/:bw_value`

Table 4.2a: SDS Controller API for IO bandwidth differentiation filter.

oAuth PARAMETER	Description
X-Auth-Token	Admin token obtained after a sucessful authentication in keystone.

Table 4.2b: Header to authenticate bandwidth differentiation filter API calls.

anomalies.

From the Figure 5 we can observe the relation *number of objects requested - number of workers*, if the number of workers is lower than the simultaneous objects requested, the performance is severely affected. It does not help to increase the number of threads, as we can see on the 1 worker - 64 threads scenario. The worker parameter is shaping the performance values more than the number of threads but both are important. An interesting value is observed using 1, 2 and 3 workers with three simultaneous objects: with 1 worker we get a fair allocation (but bad performance) for each object, but with 2 workers the scenario is totally unfair: we obtain the 50% of the performance for one single object and the 50% for the two other objects. This is due to that 1 worker is serving only one object and the other worker is serving with round-robin the remaining two objects. A similar behaviour is observed at the 8 clients scenario with 3 workers, and again, increasing the number of threads does not show any improvement (it decreases the bandwidth obtained).

This behavior produces an undesired performance impact and fairness loss in the CFQ I/O scheduler. This behavior is also observed on other environments as KVM virtual machines. On KVM, the I/O requests are sent to the kernel using a threadpool of 64 threads, in a round-robin fashion. This produces the activation of false cooperative threads in the I/O scheduler, and only a data stream is served, the other ones are waiting until there is an idle period or the chosen data stream finishes. This I/O scheduler behavior produces starvation as we are loosing some semantics and metadata included on the I/O requests like the originating PID inside the VM machine (guest). Having such disorder creates unnecessary overhead inside the kernel.

4.2.2 Changing the threading model

From the previous results, we observed that having 1 worker per object offers the better performance. The CFQ scheduler also works better as the requests follow the rule one stream - one PID and being more bursty increases the performance of HDDs.

Our next action is try to get this model dynamic, so to be able to fix a data stream into a thread, we decided to change the threading model (initially a pool of threads) to a dynamic-threading model fixing 1 thread to 1 stream (or object). Although it may seem that it will impact performance, as we are always creating and destroying threads. It is not as important as on other applications, as we are ending into a storage device and the overhead is removed once we pass through the different layers. However, other threading models can also be implemented if needed, for example a fixed threadpool

```
"127.0.0.1:6010": {
    "sdb1": {
        "0": {"account": "AUTH_test", "identifier": "object", "needed_BW": 10, "policy": "silver", "objects": [
                {"oid": "/AUTH_test/FilesSilver/file1G2.dat", "oid_calculated_BW": 5.835894944734798,
                    "range": "0-end"},
                {"oid": "/AUTH_test/FilesSilver/file1G3.dat", "oid_calculated_BW": 4.730541865223458,
                    "range": "0-end"}
            ]},
        "1": {"account": "AUTH_test2", "identifier": "object", "needed_BW": 25, "policy": "silver", "objects": [
                {"oid": "/AUTH_test2/FilesSilver/file1G3.dat", "oid_calculated_BW": 28.807570522146083,
                    "range": "0-512000"}
            ]}
    }
},
"127.0.0.1:6020": {
    "sdb2": {
        "0": {"account": "AUTH_test", "identifier": "object", "needed_BW": 20, "policy": "gold", "objects": [
                {"oid": "/AUTH_test/Files/file1G.dat", "oid_calculated_BW": 19.330390515058488,
                    "range": "0-end"},
            ]},
        "1": {"account": "AUTH_test2", "identifier": "object", "needed_BW": 25, "policy": "silver", "objects": [
                {"oid": "/AUTH_test2/FilesSilver/file1G.dat", "oid_calculated_BW": 13.02260571792931,
                    "range": "0-end"}
                {"oid": "/AUTH_test2/FilesSilver/file1G2.dat", "oid_calculated_BW": 10.476129811035005
                    "range": "102400-end"}
            ]}
    }
}
```


Figure 7: Example JSON /bw/osinfo response

model with stream id labels.

Our implementation uses a hash created by the object id as the identifier of the thread. As Swift is stateless we have several issues, for example, we do not know when the object is not going to be used again. To solve it, we setup a deadline of 10 seconds to clear any used thread and free it. No major changes are done to the code more than changing the ThreadPool class used and setting 0 as initial parameter for the number of threads.

Although we have some performance changes, the most important benefits are a removal of the two configuration parameters (workers and threads) and the result of a improved fairness on all the scenarios. The workers parameter can be removed as we ensure that each object is going to be server by a thread in a asynchronous way, so the worker will be ready to serve another request immediately. This may have an impact on different workloads, however using several benchmarks, for example, COSBench [16] we could not detect any issue on setting the number of Swift workers to 1 with the new threading model, so the modification also simplifies administrators' tasks.

As we can see on Figure 6 all the scenarios show a fair sharing of resources between clients.

4.2.3 Bandwidth differentiation implementation

Thanks to the new threading model we can track each stream of data at the object server. We can apply several policies that are unavailable using the original thread pool model. For example, we can apply directly I/O priorities [17] to create bandwidth differentiation policies. Although it is true that we can also have bandwidth differentiation on other middleware layers, but we can only share spare disk bandwidth when we use kernel I/O priorities. The kernel sets the priority of the I/O requests using the PID, so that in order to use it, we need to keep it fixed.

The Operating Systems offers several mechanisms to classify the I/O requests, one of such mechanism is the IO Priority. On the last Linux kernels we have 3 classes: *Idle*, *Best Effort*, *Real Time*. *Best Effort*, has 8 priority levels (0 maximum, 7 minimum). IO Priority is rarely used, but we tried to use such mechanism to differentiate, important requests (Requests that need to be served because we need the bandwidth) and non-important requests (Requests that can be delayed as the stream is well served in the bandwidth metric).

Using such priority component simplifies the mechanism as we can avoid using delays in the code. We are using only BestEffort 0 and IDLE priorities. Other sharing policies, for example gold-silver-bronze clients, can be implemented using more levels.

With all those modifications, we have speed improvements in HDDs devices (due I/O scheduler can do a better work). We have also checked the effect of those modifications on SSD devices, but as there is no specific I/O scheduler for SSDs there are no benefits on performance.

In order to ensure the needed bandwidth for a tenant we may need to select the *Object Store* nodes that are underused, that is to provide to the proxy server the instantaneous bandwidth of the object store disks.

To provide this information we use an external process generating such information each second, and a Swift middleware, similar to *healthcheck*, that provides an URL inside the object server providing the needed information of all the devices inside the *Object Server*. Such information is obtained when a sorted list of nodes is needed, being able to sort the node list per bandwidth used and to distribute the request to different *Object Servers*. This information is also used to provide time statistics about the usage of each disk in a *Object Server*.

4.2.4 API

All the calls of the SDS Controller API for IO bandwidth differentiation filter (Table 4.2a) must be authenticated as admin, so after successfully receiving the credentials from keystone, it is necessary to add a header (Table 4.2b) with the obtained token to all the bandwidth differentiation filter calls. This SDS Controller API directly connects to a similar API in the Swift's proxy that sends and collects the information from all the *Object Servers*

All the GET calls return the information gathered using a JSON format. As we can see in Figure 7 example of a `/bw/osinfo` response there is a lot of information about objects and about the object servers that are serving them at that moment.

4.2.5 Relation with other components

In order to provide the proxy the bandwidth assignments to each tenant, we will store this meta-data into a Redis[24] database in the SDS Controller. We left aside the traditional database to store meta-data as the meta-data workload will be read-only, Redis provides a lot of flexibility to store objects that represent the different meta-data and is much easier to scale.

So, when a bandwidth assignment call arrives to the SDS Controller, it directly connects to a similar API in the Swift's proxy. Then, the proxy notifies all the *Object Servers* about the new bandwidth assignment and it will be stored in memory. After that, the proxy will also update the bandwidth information about that tenant in the SDS Controller's Redis database, where it will be automatically available for future reference.

5 Block Storage in IOStack

In IOStack, the automated provisioning of SDS for block volumes is the second pillar of the architecture (see Deliverable 3.1 for more details). Similarly to the object storage architecture, IOStack offers a SDS controller managed via REST APIs to orchestrate traditional block storage back-ends (e.g., storage arrays). Currently, the IOStack block storage architecture provides administrators with two main services: *i) volume provisioning* and *ii) storage filters*. In what follows, we describe the architecture and high-level operation of the block storage subsystem in IOStack.

5.1 Architecture and Services

In Fig. 8, we illustrate the architecture of IOStack for block storage. Concretely, the figure shows IOStack components overlaid with MPStor specific components. The combination of these components constitute the SDS block storage subsystem.

At the high-level, we can differentiate 3 main layers in the IOStack block storage architecture. First, we find the *SDS Controller*, which accepts the requests from administrators via a REST API service to orchestrate the SDS layer (volume provisioning, filters). Second, we find the *client-side*

[24] `https://redis.io`

components of the block storage architecture (`Konnector`, `VSD`). This part of the architecture enables the interception of data flows between client VMs and block volumes mapped at the storage back-end. This client-side part of the IOStack block storage architecture is the basis for our filter framework, as we can intercept IOs between client VMs and storage arrays and apply arbitrary transformations or control to them. Finally, IOStack provides *server-side components* (MPStor Orkestra) to discover and orchestrate. This makes it possible for the SDS Controller to automate the provisioning of block volumes with performance-specific storage fabrics.

To better understand the block storage architecture of IOStack let us describe the components that can be found in a standard deployment:

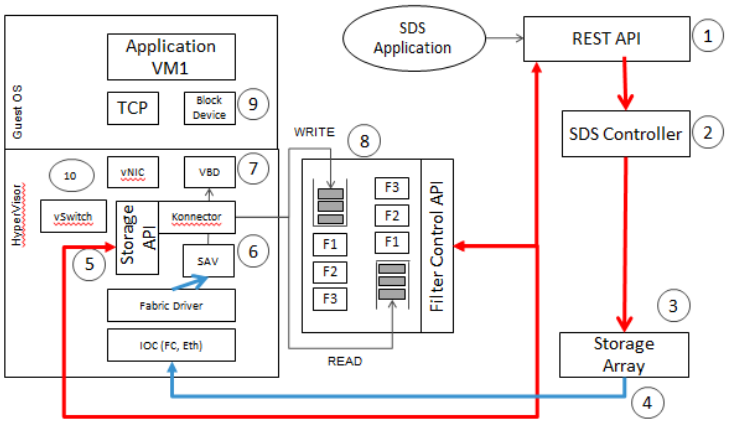

Figure 8: Architecture of the block storage framework. In particular, the Storage API clients at the VM (Hypervisor) enables the SDS Controller to coordinate them via simple HTTP requests. The SDS Controller also coordinates the block storage filter framework, which is located at the export termination of the block volume (Konnector). Moreover, the SDS Controller can coordinate several (e.g., storage arrays) via a server-side API module.

SDS REST API - #1. The REST API head allows a datacenter administrator to provision storage from a storage provider and attach that storage to a storage consumer node and build a filter stack between the attached storage and the storage application. The code for providing the REST API is a standalone application with direct access to the SDS metadata store to persist the result of changes in the SDS layer.

The SDS REST API for block storage is built as a library to communicate with the SDS controller, JSON commands with the `Konnector` layer on compute nodes and socket-based XML communication with Orkestra SAN storage nodes. The top level architecture of the REST API is shown in Fig. 9.

SDS Controller - #2. The SDS controller is an entity that builds a model of the storage provider, understands the management semantics of one or more storage providers. In practice, the IOStack SDS Controller integrates the block storage REST API, which is exposed to the administrator via the IOStack dashboard.

Storage back-end - #3. The storage provider is typically a proprietary storage array that has its own proprietary storage API or uses an industry standard such as SMI-S.

Storage Area Network (SAN) - #4. The SAN is a storage area network over which storage volumes are exported from the storage provider and attached to the storage consumer node.

Konnector - #5. The `Konnector` is an in-band component on the client node that homes the *consumer application* (e.g Virtual Machines). The role of `Konnector` is to terminate the *storage array export* (see (#5) in Fig. 8). Moreover, the `Konnector` creates a service stack or a set of storage filters by presenting the storage array export to the VM through the Virtual Storage Device (VSD). The `Konnector` layer

presents an API to the SDS controller, the SDS controller uses this API to create VSD devices. Drawing an analogy with IOFlow [1], the `Konnector` represents a stage, it is a point to enforce policies on data flows and it is orchestrated by the SDS Controller.

Volume Termination/Export - #6. The Termination layer allows the storage provider volumes to be terminated on the consumer node. This is a complex process as it requires setting up an end to end SAN connection between the consumer and provider.

Virtual Storage Device (VSD) - #7. The Virtual Storage Device (VSD) is the volume that sits on top of the service stack and is used by the application layer.

Filter Framework - #8. The service stack is a set of block storage volume functions, including filters which provide in-line data transformation functions.

Block volume - #9. The storage volume from the storage provider.

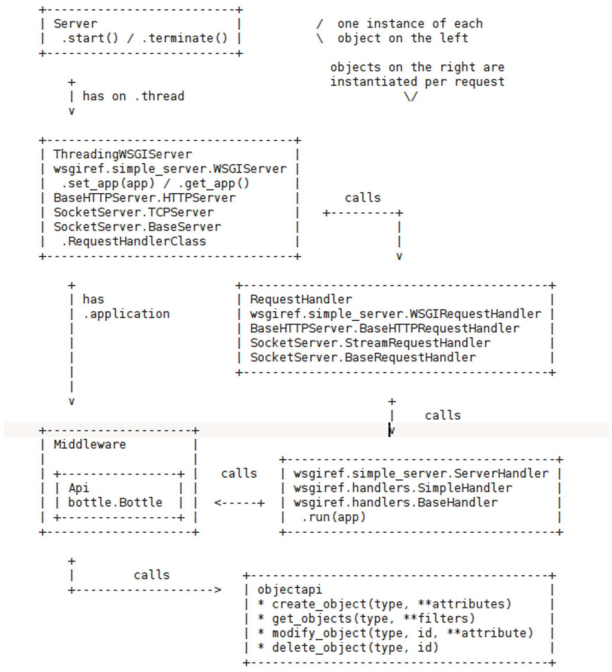

Figure 9: Architecture of the SDS REST API service for block storage.

5.2 Provisioning of Volumes

From the block storage perspective, IOStack is a framework that sits on top of traditional storage, and takes care of automating storage provisioning. Storage provisioning automation matches storage consumers with storage providers taking into consideration storage policies.

An example of a storage policy might be the definition of a media (storage) tier that has concrete resiliency, performance (in BW and IOPs) and fabric type requirements. Storage provisioning automation matches a consumer asking for a media storage with providers that fulfill the requirements of that tier. There are SDS tools in place to define both the policies as well as the metadata that describes the provided storage characteristics (e.g. Resiliency, BW).

To achieve this goal, the IOStack REST API service and the server-side modules on top of storage arrays enables the SDS Controller to discover the underlying storage fabrics and enforce performance defined volume provisioning (see Deliverable 3.1 for technical details). Moreover, the provisioning of volumes is greatly simplified thanks to the visualization of available resources at the IOStack dashboard.

5.3 Filter Framework for Block Storage

A Fundamental requirement from the block storage layer in the IOStack architecture is the ability to dynamically deploy filters within the data flow. Block storage code is typically high performance, well optimized code that runs in the system's kernel. Dynamically deploying filters into such code is a challenge. The technique adopted to address the challenge is to route the IOFlow from the kernel space into a user space where it can be directed easily to a run time deployed filter.

Filter Sandbox. A filter framework Sandbox testbed has been created on a Fedora virtual machine which acts as a development platform and demonstration platform. This framework is composed of Konnector which provides a JSON interface for control, a set of user developed filters (which are dynamically linked .so files) and standard Linux components. The sandbox environment was developed for demonstration and development purposes. The environment consists of an MPSTOR Openstack distribution packaged with the following VMS:

- UBUNTU VM with the REST API Application
- FEDORA VM with Konnector and Filter Management
- Orkestra Virtual Storage Array

The entire environment is virtualized; this reduces the number of physical machines required for the testbed. As shown in Fig. 10, this environment allows the development and demonstration of the REST API (1) provisioning storage from the Virtual Storage Array (3) which has attached virtual disks on which RAIDs are built, attaching the storage array volume (5) to the Fedora VM (2).

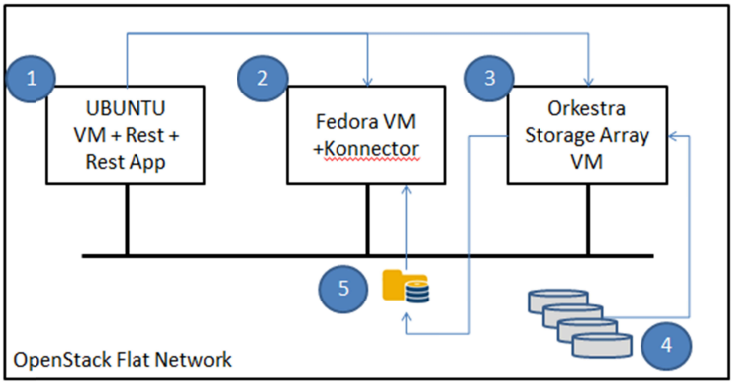

Figure 10: Sandbox of the SDS block storage framework provided for development purposes.

Filtered Block Storage Example. Fig. 11 (left) below shows the components of the Filter framework.

- SDS REST AP providing an interface to the provisioning application.
- Konnector, this is the in-band control element on the consumer node, Konnector's API allows control of the termination of storage array volumes and the creation of the service stack and filters.
- The SDS controller plug-in (3rd party) which sends storage array specific commands to storage array. This service can run anywhere, for convenience its running on the Fedora sandbox VM.
- This is the terminated "back end" storage volume provisioned from the storage array.
- This is the Filter manager which is responsible for building the service and filter stack on top of the "back end" storage.

- These are the control functions that call the filter entry points with the in-band data from the user to the storage devices.
- User Application IO (Read/Write DATA)
- The SAN storage device.

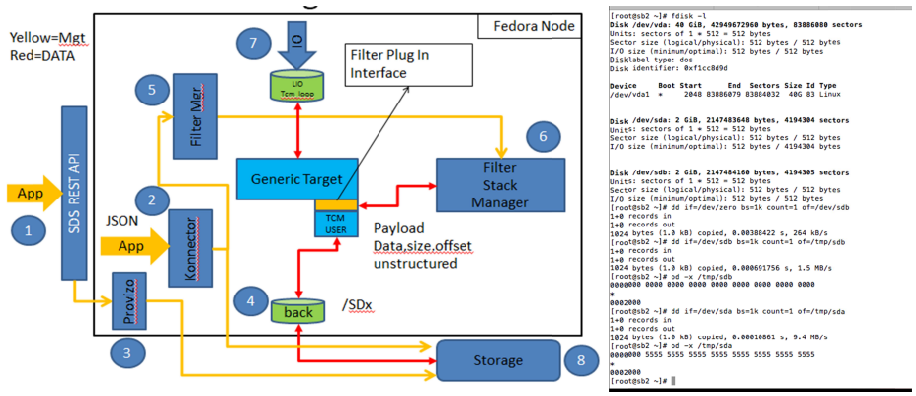

Figure 11: Architecture diagram of our VM sandbox containing the filter framework for block storage (left). On the right, we applied 3 filter to a data flow related to a block volume (`NOP`, `XOR` and `BITREV`).

Filter testing results. Although a more extensive validation of the filter framework for block storage is provided in Section 9.4, as a proof of concept the filter framework was tested for the following filters:

- *NOP filter*: This filter simply intercepted the IOs between an application and the back-end storage device, it simply logged the IOs to demonstrate the filtering could be achieved in data flow.
- *XOR filter*: this filter implemented the following function data=data XOR key, the input flow was clear data, the data sent to the storage device was xor'ed with key. The data read from disk was obfuscated by the XOR function, however the data to the application was clear data.
- *BITREV filter*: This filter reversed the bit order of the data.

The filters were tested singly but also cascaded, the effect was to apply each function to the data flow during writes and the reverse order during reads. Data read through the application was correct, data stored on the back-end device has the `NOP`, `XOR` and `BITREV` operations performed on it. As we can observe, Fig. 11 (right) shows a dump of the data from the filter device and the back-end storage, we can clearly see the written user data on the disk is the function `data=data XOR 0x55`. This proves that the filters have been enforced through the write path from the volume export to the back-end storage.

In what follows, we describe a common mechanism for both object and block storage subsystems to define dynamic storage policies; that is, storage policies that are enforced depending on specific workload conditions.

6 Dynamic Storage Policies

In previous sections, we described the elements that enable an administrator to enforce a filter on a tenant's requests, by making use of the Storage API via the SDS Controller front-end. However, a key feature of SDS is the ability of enforcing filters automatically based on the workload exhibited by tenants. We believe it necessary to devise new models to provide dynamic storage provisioning in IOStack.

6.1 Architecture and Lifecycle

One of the distinctive points of the IOStack architecture is the ability of setting policies that may dynamically enforce storage filters, depending on live workload-based decisions. Moreover, a datacenter administrator is abstracted from the complexity of dynamic filter enforcement thanks to an intuitive, high-level DSL. In this section, prior delving into technical details, we overview the process of defining and deploying a storage policies in IOStack (see Fig. 12).

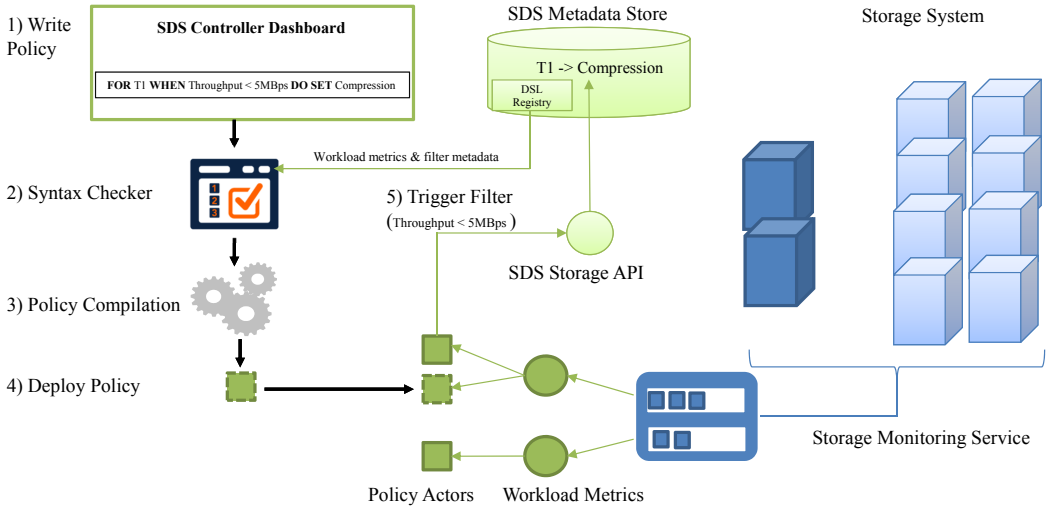

Figure 12: Policy definition, compilation and deployment lifecycle. When a policy implementation is deployed as an isolated process, it checks the appropriate workload metrics to evaluate if the condition clause is satisfied. In the affirmative case, the policy process enforces a new filter.

First, the SDS Controller offers to the administrator the possibility of writing *storage policies* in the front-end dashboard. These policies, enable the enforcement of storage filters to *targets* (e.g., tenants, containers/volumes) under certain workload conditions. To better understand this, let's see a simple example:

```
FOR T1 WHEN Throughput < 10MBps DO SET Compression
```

The previous policy defines a condition that, in case of being satisfied at any moment in time (`Throughput < 10MBps`) , will trigger the enforcement of a storage filter (`Compression`) on `T1`'s requests. We will describe in depth the syntax of policies in Section 6.2.

Given that policy definition, the IOStack DSL framework should verify that the expressions that it contains are syntactically correct according to the DSL grammar. Furthermore, to consider a storage policy syntactically correct, the compiler should also verify that the *workload metrics* (Section 6.5) and *filters* referenced in the policy do exist. In particular, workload metrics are processes (or *actors*[25]) that consume monitoring information from the storage system to provide a useful metric to define new policies (e.g., `Throughput`). As explained before, filters are the actual transformations executed on data flows (e.g., `Compression`). Both, workload metrics and filters should be registered in the DSL Registry (see Section 6.3), so that the DSL compiler could verify its existence.

If a policy definition is syntactically correct, it is compiled into a code file and instantiated as an *actor*. The *policy actor is then subscribed* to one or more workload metrics, depending on the conditions specified in the policy definition. Automatically, the policy actor will process incoming workload metric values to evaluate whether the policy condition is satisfied or not (e.g., `Throughput < 10MBps`). If a policy's condition is satisfied by the workload at hand, the policy actor will automatically call the SDS Controller API in order to enforce a filter to the appropriate target, and persist that decision in the IOStack metadata store. This part of the lifecycle is described in Section 6.4.

[25] `https://en.wikipedia.org/wiki/Actor_model`

As we describe in Section 4, from that point onwards the filter framework will acknowledge that a compression filter should be enforced on `T1`'s requests. Note that during this process, the only intervention of the datacenter administrator with IOStack is on the policy definition; the rest of the process is automatically carried out by the SDS system.

As one can infer, this novel approach of defining SDS storage services opens multiple possibilities of management and optimization. In the following, we describe more extensively all the mentioned steps in the definition and deployment of storage policies in IOStack.

6.2 IOStack DSL for Policies

IOStack provides a simple and extensible domain-specific language (DSL) to enable administrators defining storage policies. In this section, we depict the structure of storage policies, as well as the current capabilities of the syntax, which are expected to grow in the future.

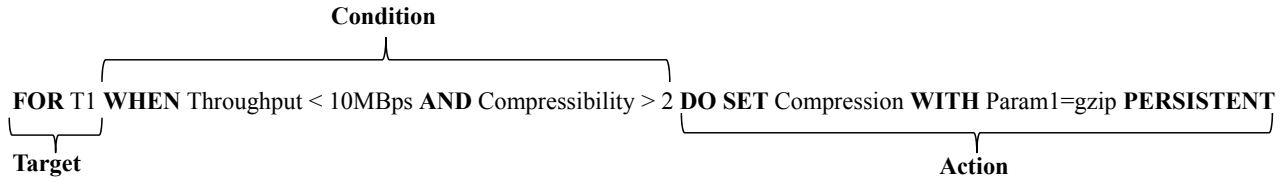

Figure 13: Description of the clauses constituting a storage policy.

Targets. The *target* of a policy represents the recipient a policy's action (e.g., filter enforcement) and it is mandatory to specify it on every policy definition. In our DSL design, targets can be *tenants*, *containers/volumes* or even individual *data objects*. In our view, the hierarchical granularity of targets enables high flexibility from an administration viewpoint.

Moreover, a policy's target can be a single entity or a group of them. To this end, the current version of the DSL syntax enables defining target groups. For instance, we can define a group such as `G1:{T1, T2, T3}` and use `G1` as a target in the policy definition (see Table 6.3a). Internally, during compilation process the system will generate individual policy actors for each tenant. Note that we include in the syntax the keyword `ALL` followed by the type of target (`TENANT`, `CONTAINER`, `OBJECT`) to refer all elements of a target type. We expect to provide advanced grouping capabilities in our DSL to ease the management of a potentially large number of targets.

An obvious consequence of this design is that *more than one filter may be enforced for the same request depending on the policy's target*. That is, an administrator may define a policy at the *tenant scope*, and another one for a certain data object of that tenant, which behaves in a particular way.

In this situation, we apply the filter to an object request with the *most specific target granularity*. Therefore, filters enforced at the data object have priority over filters defined at the container/volume target granularity, and analogously, filters defined at the container/volume will be executed instead of filters defined at the tenant granularity. This model has been traditionally applied in object oriented programming (OOP) under the name of dynamic dispatch[26].

The previous situation differs from that of *applying more than one filter to a certain target*. If an administrator wants to enforce more than one filter, he/she should specify *more than one filter in the action clause*, as we describe next.

Conditions. A central part of a policy definition is the *condition*. A policy may have one or more condition clauses that specify the situation that will trigger the enforcement of a certain filter on the target. In general, condition clauses consists of *workload metrics*, *operands* and *values*.

The first member of each condition clause refers to a workload metric. As we will see later on, workload metrics processes compute values that represent a specific aspect of the current storage workload. Also, to be able of using a workload metric in a policy condition, it should be registered in the DSL Registry (Section 6.3). By registering the available workload metrics, it is possible for

[26] `https://en.wikipedia.org/wiki/Dynamic_dispatch`

the DSL system to obtain the necessary information for the compilation and deployment phases of a policy.

Second, our DSL provides administrators with *internal and external operands* on condition clauses. Internal operands are the elements within a condition clauses that dictate the *type of comparison* to be done against the metric produces. For instance, common internal operands are "greater-than" (>), "lower-than" (<) and "equals" (=). Moreover, an administrator can define complex multi-clause conditions by making use of external operands, like `AND` and `OR`.

Finally, the condition clauses evaluate the workload metric values (dynamic) against the value defined by the administrator as a threshold to trigger the action. Clearly, the value defined by the administrator in a condition clause depends on the values produced by the workload metric at hand; this encompasses the use of absolute units (e.g., 10MBps) or fractions/percentages.

Actions. The action of a policy definition represents what should be done after the condition evaluates true. Normally, the action refers to in order to add the enforcement of a filter via the SDS Controller API. Therefore, the main goal of our storage policy framework is to enable automatic enforcement of filters once the workload-based condition defined in a policy is satisfied.

In the current version, the action clause of a policy makes it possible to enforce (`SET`) or to remove (`DELETE`) a filter from a target. That is, both types of actions are useful, considering targets that by default may have filters enforced or not. Similarly, at the end of the action clause we can define the behavior of actions. On the one hand, we consider *persistent actions* those ones that once are triggered the filter enforcement remains indefinitely (keyword `PERSISTENT`). On the other hand, our framework also enables actions to enforce a filter only during the period where a condition is satisfied, rolling back the action if the workload does not satisfies the condition in the future (keyword `TRANSIENT`). This is specially important if we take into consideration the behavior of filters: there are permanent transformations on data (e.g., compression, encryption) and non-permanent ones (e.g., IO bandwidth differentiation, caching).

Moreover, filters defined in the action clause of a policy may be parametrized. As we describe in our filter framework (Section 4), the internal operation of a filter may provide an external interface to tune its behavior. For instance, retaking the example of a compression filter, we may decide to enforce compression using a `gzip` or an `lz4` engine, and even to decide the compression level of these engines. For this reason, in the action clause we enable the addition of key/value pairs that will be passed as input parameters to the the filter to be triggered by the policy.

Our DSL already supports the definition of more than one filter in the action clause. That its, the action clause can be defined as follows: `DO SET Compression WITH Param1=gzip PERSISTENT, Caching Param1=LRU TRANSIENT`. However, the actual execution of this policy requires the ability of pipelining several filters in data stream. This feature is not yet available in the current version of our filter framework, but it is planned for the second year of the project.

Naturally, the IOStack DSL system requires information about workload metrics and filters, such as the network location of the former and the type of parameters accepted by the latter, among other aspects. Next, we describe a part of the IOStack metadata system (DSL Registry) intended to storage the necessary meta-information of both workload metrics and filters to be used by storage policies.

6.3 IOStack DSL Registry

The IOStack DSL Registry is a key element to abstract administrators from the technical complexity of low-level aspects involved in our DSL policy framework, such as inter-process communication and syntax checking. The objective of the DSL Registry is to make it possible for an administrator to use simple keywords, such as `Throughput` or `Compression`, in the definition of storage policies. To this end, the DSL Registry stores the metadata of *workload metrics* and *filters*.

For each *workload metric*, the DSL Registry needs to store its *name*, *network location* and *metric type*. The name of a workload metric is the keyword to be used in condition clauses of storage policy definitions. Workload metric names should be unique and self-descriptive to ease the design of storage policies. As workload metrics are independent processes that consume workload monitoring information, which yields that various metrics may be executed on one or more machines. This

REST Call Description	HTTP Method	URL
Add a workload metric	POST	`/registry/metrics`
Get all workload metrics	GET	`/registry/metrics`
Update a workload metric	PUT	`/registry/metrics/:metric_id`
Get workload metric metadata	GET	`/registry/metrics/:metric_id`
Delete a workload metric	DELETE	`/registry/metrics/:metric_id`
Add a filter	POST	`/registry/filters`
Get all filters	GET	`/registry/filters`
Update a filter	PUT	`/registry/filters/:filter_id`
Get filter metadata	GET	`/registry/filters/:filter_id`
Delete a filter	DELETE	`/registry/filters/:filter_id`
Add a tenant group	POST	`/registry/gtenants`
Get all tenant groups	GET	`/registry/gtenants`
Add a member to a tenant group	PUT	`/registry/gtenants/:gteanant_id`
Get all tenants of a group	GET	`/registry/gtenants/:gteanant_id`
Delete a tenant group	DELETE	`/registry/gtenants/:gteanant_id`
Delete a member of a tenant group	DELETE	`/registry/gtenants/:gteanant_id/tenants/:tenant_id`

Table 6.3a: SDS Controller API of the DSL Registry.

requires the metadata information of a workload metric to provide the *network location* to reach the process and obtain the computed metric. Moreover, a workload metric's metadata should define the type of metric produces, such a integer or a boolean, to enable the DSL syntax checker to infer if values in condition clauses belong to the appropriate type.

In the DSL Registry, all filters should provide the following metadata: *name*, *identifier*, *activation URL* and *valid parameters*. Analogously that for workload metrics, filters should be identified in the DSL Registry with a unique and self-explanatory name in order to be used in policy definitions. The identifier field is only required by out filter framework for object storage based on Storlets (Section 4). To wit, as our framework is capable of executing a variety of filters, we should provide the identifier of the filter to be executed.

Moreover, as different filter types may have distinct calls from the SDS Controller API viewpoint, we need to provide the base URL to be used to trigger the filter activation. The base activation URL stored in the DSL Registry will be completed on compilation time by adding in the parameters and values defined in the policy. By doing this, the resulting policy actor will only need to send an HTTP request to the SDS Controller API using that URL to trigger the appropriate action. In this sense, all filters in the DSL Registry should specify in their metadata the parameters are valid (if any) as well as their type (e.g., float, boolean). As in the case of workload metrics, this information will be used by our DSL framework during the syntax checking phase.

The IOStack DSL framework enables to deploy and register new workload metrics and filters in the system while running. However, adding a new workload metric or filter requires the administrator to introduce the associated metadata in the registry. This task is carried out making use of the SDS Controller API calls related to the DSL Registry (see Table 6.3a).

With the appropriate information in the DSL Registry, the DSL policy framework is able to parse and compile a policy definition into a process that communicates with workload metric processes and triggers the enforcement of filters. We describe the compilation and deployment process of policies in the next section.

6.4 Checking, Compiling and Deploying Policy Actors

When the administrator defines a storage policy in IOStack, the system should check whether the policy is *syntactically correct or not*, according to the IOStack DSL grammar. In the affirmative case, the definition of a policy is compiled into a code file and instantiated as a independent process or actor to run in the appropriate machine within the system. Once the policy actor starts, it subscribes to the necessary workload metrics in order to evaluate the workload related to its target, and trigger the action if necessary. We explain the phases of this process more technically as follows.

Syntax checking: The IOStack DSL grammar previously described is implemented using PyParsing,

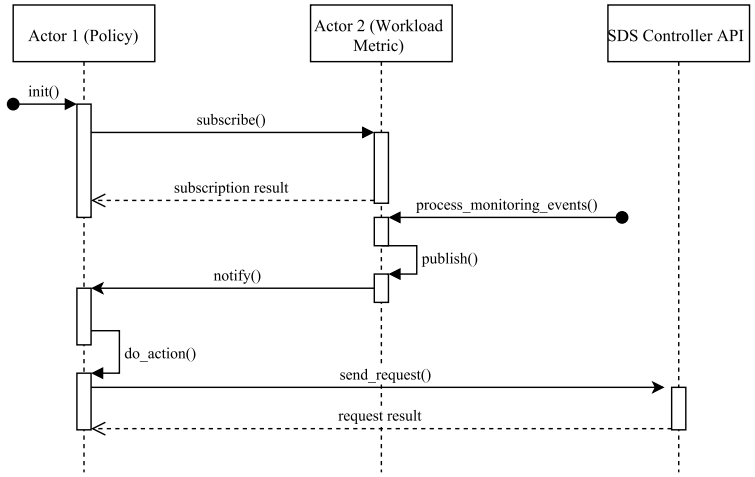

Figure 14: Sequence diagram that shows the interactions among workload metrics, policy actors and the SDS Controller API.

the most popular DSL for Python.[27] Once a policy definition is submitted for syntax checking, a DSL parser module validates the input policy definition. Note that during the syntax checking process, the DSL parser should also contact the DSL Registry in order to infer whether the references to workload metrics and filters are correct or not. This includes, for instance, to check if the workload metric or filter has been registered, to infer if the workload values to be compared in condition clauses are of the correct type, or to verify if the parameters passed to a filter are appropriate. In case of a syntactically incorrect policy definition, the system returns a human-friendly error message to the administrator, so that the policy definition can be fixed in subsequent trials.

Policy Compilation: When a submitted policy is found to be syntactically correct, the system starts the compilation phase. As a basis for the compilation process, our DSL framework provides a *policy implementation template or skeleton*. Such a template includes the basic logic that any policy implementation will execute. Specifically, the policy implementation template contains the logic *i)* for subscribing to one or more workload metrics, *ii)* to check the workload metrics values for evaluating if the policy condition is satisfied and *iii)* to send the request to the SDS Controller API for enforcing a filter. During the compilation phase, the internal attributes of the template are overwritten with the appropriate information from the policy definition.

Policy Deployment: Once the compilation phase ends, the system instantiates a new policy process or actor based on the resulting modified policy implementation template. In the initialization phase of the policy actor it is subscribed to the appropriate workload metrics. From that point onwards, the policy actor enters in the execution phase, which consists of evaluating if the received workload metric values satisfy the policy condition or not, in order to enforce a filter.

6.5 Workload Metrics and Policy Actors: Interactions

In IOStack, both workload metrics and compiled policies are independent processes or *actors* based on our PyActive [18] actor and communication middleware[28]. PyActive enables our DSL framework to execute policy processes on one or many physical machines, co-located or not with workload metrics. Moreover, inter-process communication is transparent to the system, thanks to the communication primitives of PyActive. The actor model perfectly suits our DSL framework for two main reasons: *i)* We need isolated execution of both metrics and policy actors, and *ii)* these processes should be capable of running in a distributed setting for scalability reasons.

The relationship between workload metrics and policy actors is modeled following the *observer*

[27] `http://pyparsing.wikispaces.com/`

[28] `https://github.com/cloudspaces/pyactive`

design pattern[29]. On the one hand, workload metrics are *observable* entities, as they produce information that is of interest for other entities. On the other hand, policy actors are *observers* since they are *subscribed* to one or more observables based on their interests (i.e., workload metrics to be evaluated on condition clauses).

Fig. 14 shows how actors interact in our DSL framework. First, when a policy is deployed and initialized, it attempts to subscribe to all the workload metrics that are references in its condition clauses. This is done by calling the remote method `subscribe(self, target_granularity, target_id)` of each workload metric actor. As we mentioned, the inter-process communication is greatly simplified thanks to the use of remote invocation primitives of PyActive. Upon a subscription, the workload metric actor adds the reference of the subscriber (policy actor) into its internal list of subscribers. The reference (`self`) is needed to notify subscribers via push of new workload metric values.

Concurrently, workload metrics process monitoring events from the storage monitoring system. When a new value of the metric value is calculated (e.g., every 5 seconds), the workload metric actor calls the `notify_all()` method, which yields sending the new calculated value to all the subscribers.

In this sense, it is worth mentioning that the `subscribe()` method has two additional parameters: `target_granularity` and `target_id)`. These parameters identify the target which pertains to this policy, and therefore, constitute the subscription for this policy actor. To put an example, let us imagine that the `Throughput` workload metric calculates the number of MBps that every tenant is writing/reading in the system, every 5 seconds. Then, a new policy actor sends a `subscribe(self, tenant, T1)` request to the `Throughput` workload metric. When the `Throughput` workload metric calculates the throughput values for the active tenants, it will only notify the new policy actor in the case that there is a throughput metric value related to `T1`. Otherwise, the policy actor for `T1` will not receive any message, as it is only interested on the throughput of `T1`.

Once the new metric value is received by the policy actor, it evaluates whether the policy condition is satisfied or not. In the affirmative case, the policy actor executes the `send_request()` method using the activation URL. The activation URL points to the appropriate call of the SDS Controller API with the necessary parameters in order to start the enforcement of a filter on a specific target.

At this point, we described in detail how policies can be defined, compiled and deployed in IOStack. Moreover, during their execution, dynamic storage policies trigger a filter (or a set of filters) based on a live workload-based decision. Therefore, leveraging dynamic storage policies requires a storage monitoring system that provides information about the existing workloads. The IOStack storage monitoring system is described next.

7 Storage Monitoring in IOStack

IOStack is designed to provide *dynamic storage policies* that access live information of any aspect of the workload supported by the storage system. Clearly, this requires a high-performance, scalable and flexible approach of collecting monitoring information.

To this end, IOStack advocates for a Message Oriented Middleware (MOM) to capture monitoring information of the storage system [3]. Concretely, we resort to CollectD[30] in order to send monitoring information to a RabbitMQ message broker[31]. On the one hand, RabbitMQ is known to provide high-performance event processing service, which may be scaled-out with multiple parallel instances. On the other hand, CollectD is a standard tool to collect monitoring information in cloud systems, including Cinder and Swift. This has an important advantage: our storage monitoring system unifies both block and object storage systems.

The architecture of our storage monitoring system can be observed in Fig. 15. As shown in Fig. 15, storage nodes (Swift, Cinder) send monitoring information in form of events to the MOM broker. To provide a clear organization of monitoring events, we instantiate a *queue per input metric type*. That is, all the events related to the IO transfers of storage nodes will be inserted into one queue, whereas events related to the storage capacity of storage nodes will belong to another queue. By doing this,

[29] `https://en.wikipedia.org/wiki/Observer_pattern`
[30] `https://www.collectd.org`
[31] `https://www.rabbitmq.com`

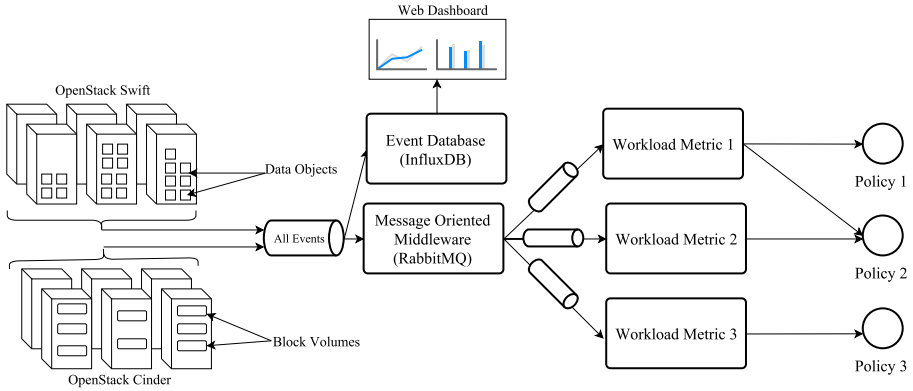

Figure 15: Centralized monitoring system for block and object storage via a Message-Oriented Middleware (RabbitMQ).

we ease the consumption of monitoring events from the viewpoint of workload metric processes. In the case that multiple workload metric processes are interested in a single monitoring metric, we can make use of RabbitMQ *fan outs* to replicate monitoring events to individual workload metric queues. In fact, multiple strategies of event organizations are possible with RabbitMQ[32].

By default, CollectD provides interesting information about the physical usage of the storage system, including the IO capacity or the current CPU state of a storage node, to name a few.

Apart from information about the physical resource utilization, by following the guidelines of the CollectD plug-in framework, we can generate monitoring information concerning the OpenStack service at hand (e.g., Swift, Cinder). That is, we integrated a plug-in that parses and converts log information of OpenStack services into events that are injected in RabbitMQ. Thus, we are capable of extracting workload metrics related to the logical viewpoint of the service, such as the read/write throughput (i.e., MBps) of a tenant within a time interval, or the number of read/write operations performed by a tenant. This approach represents a more accurate notion of the workload supported by the service and enables us to track and analyze the activity of tenants and enforce the appropriate filters on them.

Moreover, we can generate and inject arbitrary types of monitoring events to the system, for their further exploitation by different types of dynamic policies. To better understand this, let us draw an example. CollectD does not provide information about the *content of object request*, which may be of great interest for some types of filters. For instance, let us imagine that we could monitor the *compressibility of data objects* that are transferred to/from the system. This would enable us to write a policy that triggers a compression filter automatically, based on the contents of files:

```
FOR T1 WHEN Compressibility > 2 DO SET Compression
```

Such kind of policies can be achieved in IOStack by introducing processes than generate the desired metrics and inject them in CollectD. We already demonstrated this feature by providing one of such *custom metrics* regarding the IO bandwidth exhibited by tenants, containers and objects (see Section 4.2). However, state-of-the-art open source cloud storage systems are very far from this degree of flexibility and automation. Furthermore, understanding the potential of the different combinations of filters and monitoring metrics that IOStack can accommodate deserves important research efforts within the project, specially considering disparate multi-tenant workloads.

It is important to understand the *degree of dependence between the resolution of monitoring information and the definition of dynamic storage policies*. That is, if every monitoring information event is related to the tenant, container/volume and data object, workload metric processes can enable triggering policies also at the tenant, container/volume and data object granularity. Otherwise, if monitoring events are only defined at the tenant granularity, we can only define storage policies at the tenant granularity as well. Therefore, we should provide the maximum degree of precision related to the

[32] `https://www.rabbitmq.com/tutorials/tutorial-three-python.html`

storage monitoring, since it will be reflected on the target granularity of dynamic policy definitions (see Section 6.2).

Finally, it is worth mentioning that IOStack can persistently store monitoring information for further analysis in order to enhance a filter's operation. Next, we describe the monitoring framework that enables real time monitoring in the compute cluster, as well as cross-layer strategies to cooperate with the storage cluster.

8 Compute Cluster: Monitoring, Analysis and Cross-Layer Strategies

In IOStack, the compute cluster is not only a client of SDS services, but an active entity within the system. In this section, we describe *i)* the design and implementation of a system performance monitoring, and *ii)* we devise novel cross-layer strategies between the disaggregated compute and storage clusters for optimizing resources in Big Data analytics in the cloud.

8.1 Compute Cluster Monitoring

In first place, we decided to pursue the design and implementation of the system performance monitoring on two different path[33]. The first is to re-use Open-Source tools that are already well known by the community and the second is to write some new tool for resources monitoring. These two direction allow us to have two type of monitoring system with different granularity and control. The first monitor generic resources over the entire platform. The second is used to extract specific and raw information across a subset of VMs in order to have a better insight on the possible bottleneck that can afflict a specific deployment configuration. Both monitoring systems have a low intrusiveness on the platform. Similarly to the storage monitoring in Section 7, the open source software that we use are CollectD and Grafana. CollectD is a small daemon which collects system information periodically and provides mechanisms to store and monitor the values in different ways. Grafana is used for visualizing time series data for Internet infrastructure and application analytics, it features pluggable panels and data sources allowing easy extensibility and variety of panels, including fully featured graph panels with rich visualization options. The data collected by CollectD are parsed and visualized in time series by Grafana. Grafana does not come with prepared templates, so a part of our job was to create the necessary graphics needed to monitor the different aspect of the platform.

Our monitoring system in the compute cluster enables to better understand the requirements of Big Data analytic applications. To inform this argument, our compute cluster monitoring system enables us to pinpoint possible bottleneck of different compute instances deployment strategies. As we extensively describe in deliverable 5.1, to accomplish this task we ran extreme and exhaustive tests on different deployments with the help of both the monitoring system explained before. When running an analytic applications, two major layers are in place: *i)* Compute layer and *ii)* Data layer. We define the first as the union of all VMs that host the analytic framework in charge of the computation role (e.g.: Spark, Hadoop), the second is defined as the union of all VMs that host the input dataset and output results (e.g.: HDFS, Swift). A deployment strategy takes into account of the position of both Compute and Data layer. We analyzed different strategies that lead to different Compute-to-Data paths. We define Compute-to-Data path as the path that a Compute instance has to purse in order to retrieve the needed data from the physical storage. We found that indeed a correct placement of Compute and Data layer lead to lower analytics application run time.

Moreover, we integrated our monitoring system in the compute cluster with the IOStack dashboard (Section 3.1). Thus, the datacenter administrator can simultaneously inspect in real time the state of both storage and compute clusters in a single web dashboard. Such information is of great help in order to define storage policies or perform other types of administration actions.

8.2 Cooperative Compute & Storage Clusters: Cross-Layer Strategies

Often, it is common to find large-scale data processing scenarios where compute clusters and data stores are physically disaggregated, usually being inter-connected by a high speed networking layer. This approach advocates for specializing storage and computation clusters in terms of hardware, which seems reasonable given the disparate requirements of storage and computing tasks. Moreover,

[33] For full details on the design of the compute cluster monitoring system we refer to Deliverable 5.1.

this scenario enables storage and compute resources to be easily shared across multiple tenants via virtualization techniques.

Despite its benefits, the unintended consequence of disaggregating storage and compute is that it represents a performance barrier to scale-out Big Data analytics. That is, data should be moved or batch loaded from the data store to the compute cluster before the actual computation takes place (i.e., bulk processing). Thus, companies are forced to adopt a "store-first-query-later" approach for processing large volumes of data. Indeed, this has various drawbacks.

In IOStack, we propose to enable compute and storage clusters to cooperate for reducing the overhead of traditional bulk processing Big Data analytics. In particular, we currently propose two cross-layer strategies in this regard:

Explicit computation off-loading to the storage: The first type of cooperation between the compute cluster and the storage subsystem in IOStack is related with our filter framework. That is, in Section 4.1.3 we described that Spark instances may explicitly offload parts of SQL statements (projections, selections) to a SQL-like filter in the storage side. In Section 9.4, we show the promising benefits in performance of this type of cooperation between storage and compute clusters.

SDS Data locality service for co-located computations: Our SDS Controller provides a REST API call that retrieves information to locate the physical machine where a certain data object or volume resides. Concretely, we name this API call a *data locality service*, as it provides information about the physical location of data (e.g., `get(objectID)` $\rightarrow$ `machineAddress`).

One of the major advantages of our data locality services is that it enables compute instances to be deployed directly on the machines containing the required data. Naturally, this may save important amounts of network traffic within a datacenter, which may be translated into shorter compute times.

As one can infer, the cooperation of compute and storage clusters opens a wide spectrum of possible cross-layer strategies [19]. In Section 11, we describe future innovative mechanisms that we plan to integrate in IOStack to reduce the usage of resources and to enable predictive and automatic storage policy definitions.

9 Benchmarking Framework

In the following, we define the benchmarking framework and use case scenarios required to validate the overall results of IOStack. First of all, we will describe in detail several benchmarking tools (standard benchmarks, load generators) and platforms (IOStack partners' testing infrastructure) for developing experimental stress-tests. This is essential to perform an exploratory analysis of new SDS components and features in a controlled manner.

Besides, our benchmarking framework is aimed at validating the achievements of the project making use of use case workloads: Idiada, GridPocket and Arctur. In all cases, the target company will provide different workloads traces and data sets of their existing infrastructures. This will help academic partners to analyze the incoming workloads, infer the SDS services suitable in each situation and reproduce workloads in a controlled environment.

The last benchmarking phase in IOStack is the use of real-workloads to test the behavior of the system in a pre-production setting. Naturally, this phase is expected to be achieved when the platform is mature enough to support actual customer workloads. The mentioned phases in our benchmarking framework are shown in Fig. 16.

Figure 16: Phases in the IOStack benchmarking framework lifecycle.

Overall, the objective of our benchmarking framework is to validate the entire IOStack platform. In this sense, this section concludes by showing a battery of experiments corresponding to already developed IOStack components. In particular, we demonstrate that data services deployed in the SDS layer can obtain reasonable performance and optimize their previous massive bulk data transfers between storage and computing layers (storage optimization). Moreover, we show the advantages of storage automation and policy management by enforcing various filters. Our early results give a sense on the advantages that IOStack provides to the entire life cycle of Big Data flows involving storage and computation.

9.1 Benchmarking Platforms

The cornerstone of our benchmarking framework are the available platforms that the IOStack consortium contributes to perform tests and experiments. In this section, we provide a brief description per partner of the infrastructure committed for experimentation purposes.

Arctur platform: Arctur will configure and host a separate, private cloud installation of OpenStack to support IOStack experiments. This will enable us to install, configure and manage a separate instance of each component, allowing us to also install various experimental components or packages as well as alpha and beta IOStack components and patches. Front-end as well as management shell will be accessible via Internet, providing appropriate username/password and username/sshkey authentication for all interested project partners. We will adjust separate components in accordance with project requirements during project lifetime. Object storage will be backed by mid-end storage servers (12 SATA disks per server). Compute resources will be backed by high-end compute servers (12 cores, $32 - 128$ GB RAM memory per server).

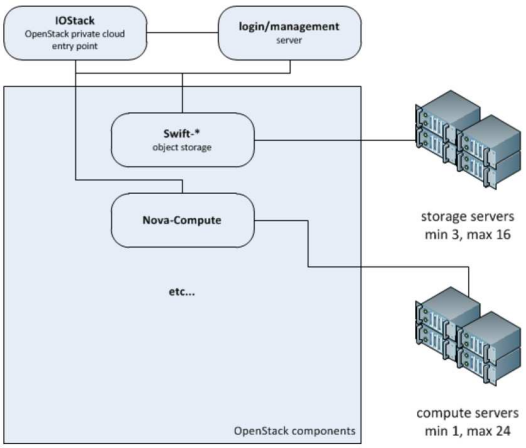

Figure 17: Experimental platform provided by Arctur.

Eurecom platform: The virtualization layer adopted by the Eurecom platform is based on OpenStack, an open-source cloud operating system used by many companies like Rackspace, Paypal and others. In what follows, we describe the baseline setup and a series of instances of the Eurecom platform.

The first hardware for the Eurecom platform began arriving at Eurecom in December 2012. While the hardware configuration and network topology has been stable since the first installation, the software side of the Eurecom platform has seen several iterations, passing through early evaluation to a production-like environment.

Configuring and tuning OpenStack is complex: the large number of parameters that govern system behavior, the variety of hypervisor technologies, the different flavors of storage systems (various file-system and logical volume management combinations), the number of alternatives to implement network switching (GRE tunnels, dynamic VLANs) all play a crucial role in determining the overall system performance.

Eurecom's cluster uses a heterogeneous set of physical machines: Eurecom has two master nodes running on a dual quad-core Xeon L5320 server clocked at 1.86GHz, with 16GB of RAM, two 1TB hardware RAID5 volumes, and two 1Gbps network interfaces. 6 Workers nodes execute on six dual exa-core Xeon E5-2650L (with hyperthreading enabled) servers clocked at 1.8GHz, with 128GB of RAM, ten 1TB disks (configured as JBOD) and four 1Gb/s network cards. 5 Workers nodes execute on six dual exa-core Xeon E5-2630 (with hyperthreading enabled) servers clocked at 2.4GHz, with 128GB of RAM, ten 1TB disks (configured as JBOD) and four 1Gb/s network cards.

Each machine in the cluster runs the same Linux distribution, a Ubuntu 14.04 LTS, updated with the most recent patches. All energy saving settings in the BIOS are disabled, since they cause severe performance penalties. We use the KVM hypervisor, with virtio and vhost net acceleration modules enabled. Virtualization support in the CPUs is enabled (VMX) and KVM uses it automatically. The hypervisor is configured by OpenStack Nova to use LVM for VM storage.

Eurecom uses the Juno release of OpenStack, which is installed via the Ubuntu cloud repository. One of the master nodes runs the OpenStack management services: the web-based dashboard console, cinder, glance, keystone, and neutron (including the server, layer 2/3 services and DHCP agents). Worker nodes are configured as compute-only nodes, and they host all the VMs created by our tenants and users. Currently, we configured neutron to use a flat network with the linuxbridge plugin.

MPStor platform: MPStor has set a test cloud deployment (see Fig. 18) that is comprised the by following components:

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

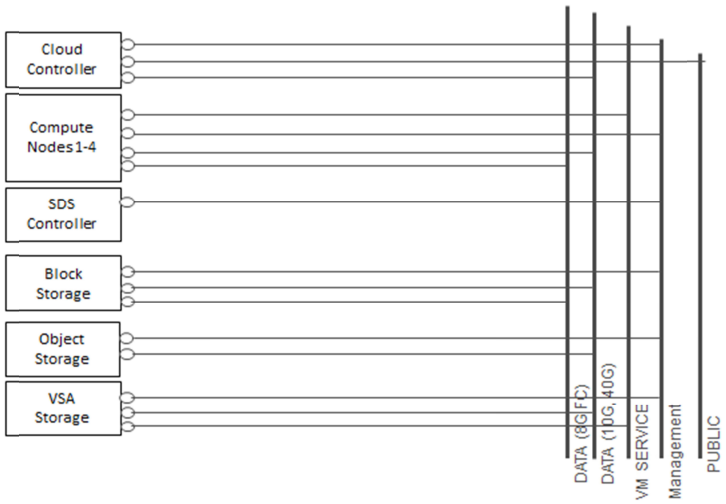

Figure 18: Diagram of MPStor testing platform.

URV platform: As a necessary task for developing our research, we set up a cluster of 12 nodes that will be one of our testing environments in this project.

To fit the needs of the projects, we disaggregated the (3) compute and (7) storage nodes. Compute nodes are intended to virtualize data processing tools, like Hadoop, whereas storage nodes will support both block and object storage services. Moreover, there are 2 proxy nodes that are committed to process external requests and run the resource-consuming services of OpenStack. Machines are connected via 1Gbit switched network links.

In terms of software, we did a complete installation of OpenStack Kilo. The most important deployed services related to the project are Nova for virtualization, Swift for object storage and Cinder for block storage. We also deployed other services which are required for the correct operation of the cluster, such as Glance for VM management and Neutron to configure virtual networks within the cluster. The design of our testing platform is presented in Fig. 19.

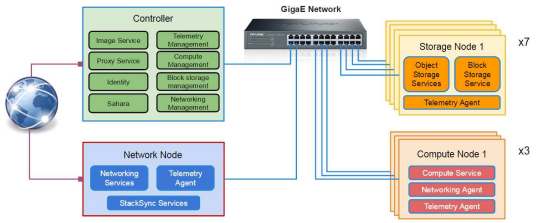

Figure 19: URV's experimental platform for IOStack.

9.2 Synthetic Benchmarks

When the design and development of a new IOStack component (e.g., filter, module) reaches a pre-alpha state, we should initiate a benchmarking phase of that component to evaluate its performance. Therefore, we need a set of benchmarking tools that allows us to execute synthetic experiments under controlled conditions. In the following, we describe a set of benchmarking tools that are present during the benchmarking phase of IOStack components, which are classified into four categories depending on their purpose: *(OS)* for object storage benchmarks, *(BS)* for block storage benchmarks, *(DA)* for data analytics benchmarks and *(OT)* for other tools.

ssbench[34] **(OS)**: SwiftStack Benchmark Suite (ssbench) is a flexible and scalable benchmarking tool for the OpenStack Swift object storage system. ssbench is specifically designed for *load*, *stress* and *soak* testing. Among its advantages, it enables the distributed execution of workloads (multi-node), it provides detailed performance measurement metrics, it exhibits a great workload flexibility and it supports authentication with both OAuth 1.0 and 2.0.

Regarding multi-node workload generation, the coordination between the ssbench *master process* and one or more ssbench *worker processes* is managed through a pair of PyZMQ sockets. This allows ssbench master process to distribute the benchmark run across many, many client servers while still coordinating the entire run (each worker can be given a job referencing an object created by a different worker).

ssbench is designed to run benchmark "scenarios" against an OpenStack Swift cluster, utilizing one or more distributed ssbench-worker processes and saving statistics about the run to a file. An scenario represents the configuration of the workload being executed (object size, put/get ratios, etc.). When the benchmark finishes, the ssbench can then generate a report from the saved statistics. By default, ssbench will generate a report to STDOUT immediately following a benchmark run in addition to saving the raw results to a file.

COSBench[35] **(OS)**: COSBench [16] is a benchmarking tool developed by Intel to measure the performance of Cloud Object Storage services. COSBench provides more flexibility than ssbench in the workload generation, but it is slightly more complex on its configuration and set up.

[34] `https://github.com/swiftstack/ssbench`
[35] `https://github.com/intel-cloud/cosbench`

In particular, to simulate diverse usage patterns, COSBench can generate different workloads from workload models defined upon the storage interface. The current workload model can be configured in terms of concurrency pattern, access pattern, usage limitation and others. This is even more flexible than the workload models of ssbench.

COSBench now supports OpenStack Swift and Amplidata v2.3, 2.5 and 3.1, as well as custom adaptors to any other object storage system.

SDGen (OT): Storage system benchmarks either use samples of proprietary data or synthesize artificial data in simple ways (such as using zeros or random data). However, many storage systems behave completely differently on such artificial data than they do on real-world data. This is the case with systems that include data reduction techniques, such as compression and/or deduplication.

To address this problem, in this project we contribute a benchmarking methodology called mimicking and apply it in the domain of data compression [4]. Our methodology is based on characterizing the properties of real data that influence the performance of compressors. Then, we use these characterizations to generate new synthetic data that mimics the real one in many aspects of compression. Unlike current solutions that only address the compression ratio of data, mimicking is flexible enough to also emulate compression times and data heterogeneity. We show that these properties matter to the system's performance.

In our implementation, called SDGen, characterizations take at most 2.5KB per data chunk (e.g., 64KB) and can be used to efficiently share benchmarking data in a highly anonymized fashion; sharing it carries few or no privacy concerns. We evaluated our data generator's accuracy on compressibility and compression times using real-world datasets and multiple compressors (`lz4`, `zlib`, `bzip2` and `lzma`). We are currently working on integrating SDGen in `ssbench` and/or `COSBench`, in order to provide these synthetic benchmarks with realistic contents.

fio (BS): fio[36] is an I/O tool meant to be used both for benchmark and stress/hardware verification. It has support for 19 different types of I/O engines (sync, mmap, libaio, posixaio, SG v3, splice, null, network, syslet, guasi, solarisaio, and more), I/O priorities (for newer Linux kernels), rate I/O, forked or threaded jobs, and much more. It can work on block devices as well as files. fio accepts job descriptions in a simple-to-understand text format. Several example job files are included. fio displays all sorts of I/O performance information, including complete IO latencies and percentiles. Fio is in wide use in many places, for both benchmarking, QA, and verification purposes. It supports Linux, FreeBSD, NetBSD, OpenBSD, OS X, OpenSolaris, AIX, HP-UX, Android, and Windows.

Bonnie++ (BS): Bonnie++ allows you to benchmark how your file systems perform with respect to data read and write speed, the number of seeks that can be performed per second, and the number of file metadata operations that can be performed per second[37]. Bonnie++ will provide IOStack partners devoted on block storage to analyze the performance of file systems mounted on top of IOStack block volumes.

Standard Spark Workloads (DA): Apart from validating the storage layer of IOStack with storage benchmarking tools, we also make use of Big Data standard workloads. In fact, optimizing these workloads is one of the major objectives of this project.

Reviewing the literature, we make use of target jobs used by "normal" users and other will stress the platform so that bottlenecks can be found easily. Some of those workloads are *i)* classic *WordCount application*, *ii)* our modified version of *TestDFSIO*[38], *iii)* our modified version of *TPC-DS*[39] as well as iv) real/industrial workloads from our use-cases or public trace repositories. Other workloads coming from recent research efforts may be also considered in our benchmarking framework [20].

We will focus on executing these data analytic workloads in disaggregated compute and storage clusters. This is the main scenario for which IOStack is designed, in order to provide real data analytics as a service in the cloud.

[36] `https://github.com/axboe/fio`

[37] `http://www.coker.com.au/bonnie++/`

[38] `https://github.com/michiard/TestDFSIO`

[39] `https://github.com/DistributedSystemsGroup/Spark-TPC-DS`

9.3 Use-case Workloads and Trace Collection

Apart from synthetic benchmarks, in IOStack we focus on gathering traces from real workloads to exercise the SDS system under realistic conditions. In this regard, the use-case companies of the consortium play a critical role. Next, we provide a brief description of the scenario and type of workload that these companies have to handle daily. Given that, we define the guidelines that use-case companies will follow to gather traces from their system to provide them to other partners.

9.3.1 Arctur

Scenario: Arctur is a PaaS, IaaS and SaaS provider. Besides pure infrastructure related services it also provides consulting and training as well as services customization. That said Arctur mainly targets at specific, end-user oriented market rather than mass-market. In particular, Arctur supports two main types of workloads: *i)* General computing virtual machines, and *ii)* High Performance Computing (HPC) applications.

Regarding the first scenario, the main production environment is based on VMware vSphere Enterprise Plus 5 with vCloud Director 5. Arctur can allocate a particular amount of resources (CPU, memory and storage) to interested IOStack partner where one can utilize all the features that vCloud Director offers (fast provisioning, etc.).

Regarding HPC, Arctur provides specialized offer for running HPC jobs on its HPC cluster. Services are sold as CPU-hours or by renting a number of physical nodes in the HPC cluster. Environment is pre-configured with various pre-installed software and tools (compilers, MPI, open-source end-user software solutions i.e. OpenFOAM, etc.)

The infrastructure where both types of workloads are running is presented in Fig. 20.

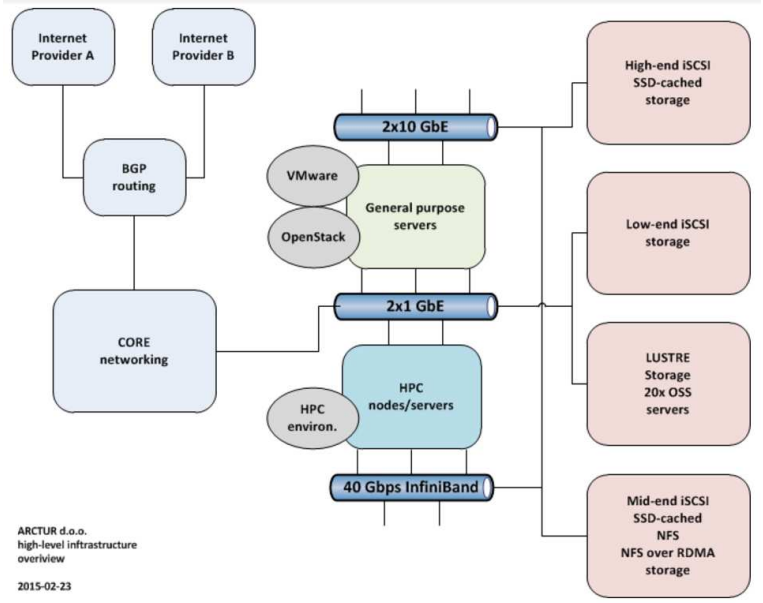

Figure 20: Arctur's infrastructure where different types of customer workloads (General Computing, HPC) will be captured.

Arctur is running virtualized environment on VMware vSphere platform with approximately 500 virtual machines and we could get some traces from VM storage usage. Moreover, Arctur has customers running HPC applications with heavy storage usage that are suitable for monitoring and trace collection.

Naturally, these workloads may be of great interest to exercise the IOStack platform under realistic conditions. In the following, we describe various aspects of Arctur workload that will be object of trace collection, as well as the tools needed to build such traces.

Traces: Arctur will provide several options for logging and capturing traces and activity as proposed, but not limited to the following:

- *Capturing system logs from servers on various services and data centre levels*. Arctur is using it's internal centralized `syslog` management therefore we are able to collect any kind of logs that are using the industry standardized `syslog` logging format. Logs can be later on anonymized and processed for the needs of the projects.
- *Capturing OpenStack Swift logs*. To this end, Swift already provides a fairly good logging system that enables one to analyze the storage interactions.
- *Capturing traces on file system level for various workloads*. Low level tracing of file system logging can be enabled on supported Linux filesystems. One example of these kind of tracing would be IO Block Traces (Drive). This type of traces describe the input/output activity of a hard/ssd drive. There are currently tools that do the hard work of monitoring and reporting logs about the drive activity with very low overhead (1-2%). In this sense we could use the `blktrace`[40] utility to evaluate the cost of making. This tool is integrated in the Linux Kernel and produces logs of the IO activity for particular drives. One potential use case for such tracing would be running IO intensive HPC applications while tracing the activity of the underlying storage infrastructure.
- *Capturing VMware virtual machines virtual disk traces for various workloads*. Arctur is running test and production VMware environment with hundreds of virtual machine. Therefore one of the option is also to trace the activity of the virtual disks activity for a particular subset of virtual machines.

9.3.2 Idiada Traces

Scenario: The workload of Idiada encompasses several stages, as we describe next. The first stage is called *model preparation*. In this phase, users create models in the workstation and they save the data into the shared storage. This access is made using a mount point through a NFS services if they are working under GNU/Linux or CIFS if they are working under Windows.

The second stage is called *simulation*. That is, once the model is prepared and saved in the shared storage they submit the jobs to the Grid Engine (GE). The GE sends the job to the simulation server which will run several processes. As first stage it will copy the data into a dedicated storage from scratch. Then, the simulation process runs the process over this dedicated storage. As last step in this stage, the results are copied to the shared storage, in the old path were the simulation stores the model. The last stage is the *data analysis*. Thus, once the simulation is finished, the end user reads the results with specific applications over the same mount point as the model preparation stage.

Traces: Given the previous workload, there are several aspects of our storage lifecycle than can be analyzed and inspected. In particular, we plan to instrument the following parts of our storage lifecycle for providing the consortium with traces.

First, data copied over the network from the workstation to the used server. To this end, we make use of `tcpdump` to trace NFS/CIFS access via the network. Moreover, we can provide dumps pf I/O operations in the server where model is stored using `blktrace`.

Second, while the simulation process runs. In this scenario, we also make use `blktrace` to capture I/O activity in the storage devices. Similarly, we can get the I/O activity data analysis stage, as it has the same behavior as the preparation stage. Traces of the data analysis stage would be very interesting in order to understand the I/O activity of data analytics applications at low level.

[40] `http://linux.die.net/man/8/blktrace`

9.3.3 GridPocket Workload

Scenario: The use case takes into account the technical deliverables of each partner, in particular IBM, URV and Eurecom, the state-of-the art of available technologies, and project objectives. The selected use case scenario will consist on deployment of a "privacy-by-design" architecture based on the IOStack platform.

Several energy utility companies and infrastructure manufacturers has been interviewed. These interviews have confirmed market interest for the privacy functionality in relation to the energy data. The technology benchmarking exercise has confirmed a possibility of proposing solution meeting market objectives and going beyond the industrial state-of-the-art. The regulatory aspects have been studied with respect to national and European initiatives. Two conference calls has been organized with experts of ENISA (European Union Agency for Network and Information Security).

GridPocket has identified functional requirements for the basic energy data anonimysation algorithm based on data down-sampling and up-sampling. This algorithm will leverage on the IOStack platform computing performance to offer better analytics experience for end-users and data scientists.

Traces: GridPocket has put in place a machine-to-machine (M2M) framework enabling collection of data (energy usage traces). This architecture has been setup independently from platforms of energy utilities in order to produce data sets free of privacy constraints and other legal limitations. The data collection architecture is composed of the private cloud systems and set of remote sensors. The cloud servers implement three layers architecture: machine-to-machine semantic communication, data storage and analytics layer, and data presentation layer. The system is deployed in a private cloud with usage of containers technology.

The collected data corresponds to electric energy consumption traces and meta data.

This system has been installed in real buildings with consent of occupants. The implemented energy sensors enable large flexibility in terms of data intervals and resolution. The data storage and analytics layer ensure generic smart grid data analytics capabilities.

Two user interfaces has been developed to enable exploitation of the platform. A utility administration interface enables the platform administrator to control the status of the system. An end-user interface enables building user to access energy consumption data.

9.4 Experimental Results of IOStack Platform

At this point, we have described the architecture of IOStack as well as the tools and platforms for validating its correct operation via real-world benchmarks and experiments. Next, we present a battery of experiments that already certify the correctness and performance of some IOStack components.

9.4.1 Enforcement of Storage Automation and Dynamic Policies

Storage Automation Policies. In this experiment, we want to evaluate the benefits of enforcing storage automation policies on Big Data workloads using the IOStack filter framework for object storage. Specifically, we executed in parallel workloads of tenants *T1* (write-only) and *T2* (read-dominated), drawing a similar scenario to the one proposed in Section 2.1.

The workload of *T1* is generated by an *object storage benchmark* (`ssbench`) that uploads 32K synthetic text objects of 10MB in size using 4 threads. *T2* is represented by a Spark deployment (3 worker VMs, 1 master VM) that downloads an existing *log file* of 164GB in size (64MB splits, `.csv` format). This log file represents the activity of users in a large storage system, and each line includes several fields like `user_id` to identify the user performing the action, `file_id` that specifies the file that is being managed, or `msg` to provide additional information (e.g., error messages), among other fields. Given that, after downloading the log, *T2* performs a simple *word count task* on the `user_id` field to calculate the number of occurrences of users.

We executed this experiment on the URV testing platform (see Section 9.1). Thus, compute nodes virtualize the Spark deployment (*T2*), whereas storage nodes and the proxy run Swift and

our IOStack prototype (SDS Controller and filter framework). We execute `ssbench` in other servers at URV, so *T1*'s `PUT` requests access our cluster from the Internet.

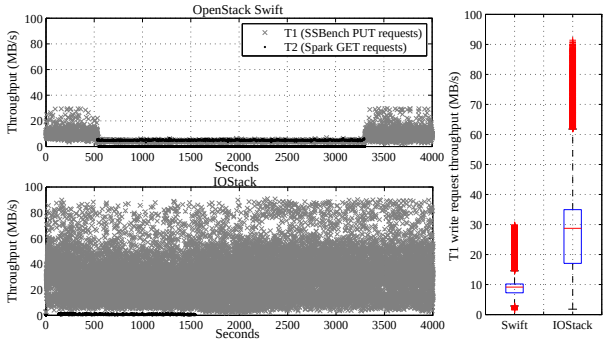

Figure 21: Comparison of Swift and IOStack in a multi-tenant scenario. Scatter plots show the throughput of tenants' requests and the boxplot depicts the throughput of `PUT` requests for *T1*.

Benefits for T1: *T1* is a write-only tenant that uploads log-like data to the system. Therefore, given that *log-like data tends to be highly redundant*, we enforced in IOStack a compression policy in the proxy —a filter that uses `gzip`— to tenant *T1* in order to *i)* improve transfer performance and *ii)* minimize storage usage. Hence, scatter plots in Fig. 21 show the throughput of *T1*'s `PUT` requests (`ssbench`) and *T2*'s `GET` requests (Spark), for both Swift and IOStack.

Observably, due to the parallelism of `PUT` requests, the Swift proxy cannot deliver to *T1* more than 30MBps per request. Furthermore, when Spark starts downloading data, the throughput of both tenants decreases drastically: most concurrent requests exhibited a throughput around 4-6MBps.

Conversely, IOStack performs significantly better that Swift for PUT requests of *T1* due to the enforcement of a compression policy on highly redundant data. That is, the boxplot in Fig. 21 demonstrates that IOStack may achieve a median write throughput of 3x higher than Swift. Furthermore, as visible in the lower scatter plot of Fig. 21, *T1*'s PUT operations are only slightly affected when *T2* starts its activity.

Apart from transfer gains, IOStack also involves important storage space savings. To wit, *T1* stored along the experiment 312GB of data in Swift —considering 3-way replication, the actual amount of consumed storage is 936GB. Due to the high redundancy of data produced by `ssbench` [4], IOStack compressed *T1*'s data to 0.1% of its original size.

Benefits for T2: *T2* uses Spark to download a dataset and to account the number of user id occurrences on it. Given that, we noted that *T2 only needs a fraction of the dataset to carry out such a task* (i.e., user id fields). Thus, we enforced in IOStack a compute-close-to-data policy that filters on the server side the data actually needed by *T2*. Intuitively, such an active storage filter may yield two advantages for *T2*: *i)* To reduce the total amount of data to be transferred from the object store to the compute cluster, and *ii)* to decrease data processing times.

Firstly, we noted that filtering the dataset at the source enables an important reduction of bandwidth for *T2*. Specifically, retrieving only user id fields instead of all fields per line of log reduces the amount of outgoing bandwidth in 95.6%. Although the throughput of *T2*'s transfers is lower for IOStack due to filtering overhead and the smaller object size, the traffic reduction greatly amortizes these penalties.

A consequence for *T2* of enabling IOStack to filter data objects at the source is that Spark processing times are much lower. That is, the Spark cluster exhibited a processing time of $9,625$s and $4,009$s for Swift and IOStack, respectively. This means that IOStack reduced the processing time of Spark in 58% compared to a regular Swift deployment.

Benefits for the administrator: These results are very interesting from a performance perspective. However, the major benefit of IOStack is to provide a datacenter's administrator with a simple way of

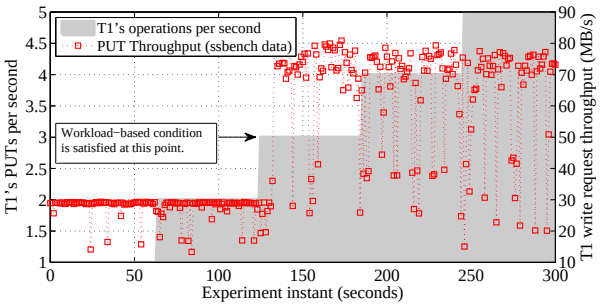

Figure 22: Example of a dynamic storage policy. When *T1*'s requests reach the workload condition, the system automatically triggers compression.

enforcing storage policies to object requests. To conclude, our experiments certify that IOStack enables easy and effective enforcement of a wide range of policies (data reduction, compute), which can greatly improve the operation of a multi-tenant object store.

Dynamic Policies: Next, we examine the operation of dynamic storage policies in IOStack. That is, Fig. 22 shows `T1` performing `PUT` requests with increasing intensity. Then, we defined a dynamic policy that will enforce data compression on `T1`'s requests if it exhibits ≥ 3 `PUT` per second:

```
FOR T1 WHEN PUTS_SEC > 3 DO SET COMPRESSION
```

Under such workload, our monitoring system updates the number of `PUT`/sec of `T1`. Then, the policy actor subscribed to this metric detects that the workload of `T1` satisfies the condition, and triggers the enforcement of a compression filter. From that point onwards, requests are compressed and, due to the redundancy of data objects, exhibit higher throughput. This demonstrates the ability of IOStack to manage dynamic storage policies, that may apply to a wide variety of filters.

9.4.2 Bandwidth differentiation results

A first set of experiments are done into a SAIO installation, using a 7200 rpm HDD for object storage. The workloads are sent to Swift using another client machine connected using a 1GB network. On some experiments, the client is executed also on the server machine to get the maximum disk performance not possible when using the network.

We have a second experiment to explore the Bandwidth assignation. It is done inside a Cluster from Arctur with 6 Object Servers and a Proxy. Results are presented using the time as x-axis and the Bandwidth obtained as the y-axis.

Bandwidth differentiation uses COSBench [16] as benchmark. As we do not have any cancel operation (needed to be able to relocate) in the server side, the bandwidth assignment tries to maintain a fair relation with the assigned BW. As a side effect, having different BW assignments creates some bursts in the data and HDD devices work better than without BW assignment.

We configured COSBench to use 300MB objects, with 300 seconds, using 2 drivers and 8 workers per tenant. Tests are done with 3 tenants.

Table 9.4a presents the results of different bandwidth assignations, including no assignations and the original Swift. Here we obtain better performance, even without BW assignation, due to a better scheduler behavior. But we achieve better performance when we assign different BW at each tenant due to more bursty requests. It is interesting to observe the Max Priority row, where we assign infinite bandwidth to one tenant, that COSBench did not achieved to get the Tenant 2 and Tenant 3 objects due to time outs on the client side. With HDDs is easily to observe that the concept of maximum BW is hard to define due to its dependency with the workload.

Using Arctur's Cluster we tested the Bandwidth differentiation mechanism using more object servers and bigger objects. We can observe that the relations between the different tenants are maintained, even if we exceed the cluster capacity. As we cannot cancel objects and the Bandwidth obtained is workload dependant, we should be proactive and try to do not send the objects to an overloaded server. On this experiment we have send request for 1GB objects with a single tenant with a

Experiment	Tenant 1	Tenant 2	Tenant 3	Total BW (MB/s)
No BW Diff	8±0,1	8±0,1	7,9±0,1	23,95
Max Priority: T1	101,8±0,6	0±0	0±0	101,80
BW Diff: 50/-/-	67,4±4,2	4,8±0,6	4,7±0,5	76,85
BW Diff: 70/-/-	78,2±2,5	4,7±1,5	4,7±1,5	87,51
BW Diff: 25/25/-	32±5,1	30,4±5,4	3,9±0,7	66,32
BW Diff: 15/20/15	16,6±1,7	24,6±4,3	17±2,1	58,13
Original	7,7±0,3	7,7±0,3	7,7±0,3	23,15

Table 9.4a: BBandwidth differentiation using HDDs. Numbers are MB/s. Includes 95% confidence interval.

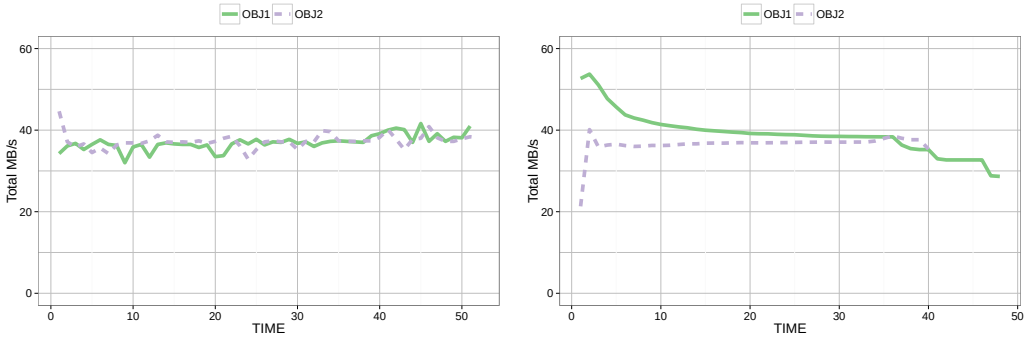

Figure 23: Performance obtained with 1 tenant requesting two 1 GB objects with the original Swift (left) and the modified one (right). The requests go to different object servers. Background I/O noise due to replication.

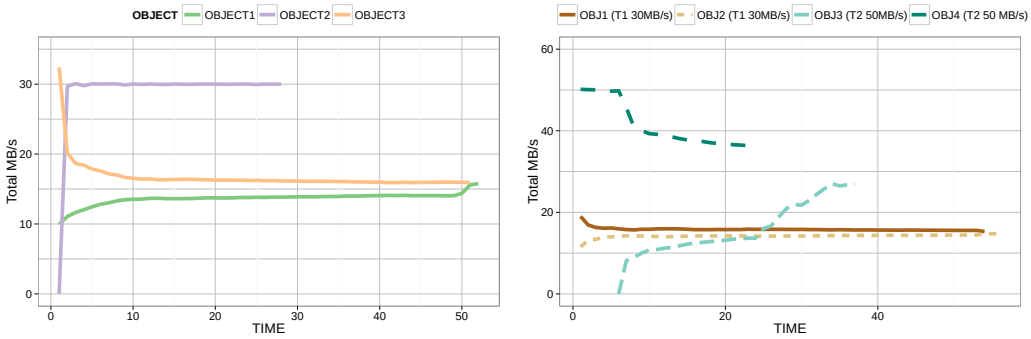

Figure 24: Performance obtained with 1 tenant requesting three 1 GB objects with 30 MB/s Bandwidth Differentiation (left). Performance obtained with 2 tenants requesting two 1 GB objects each one with 30 MB/s and 50 MB/s. Each tenant is directed to a different Object server (right).

bandwidth differentiation of 30MB/s per Object server and without bandwidth differentiation.

As we can see on Figure 23 (left), the requests does not get a lot of throughput, being 40MB/s the maximum per object server due to background noise created by Swift replication mechanisms. On Figure 23 (right). Using the modified Swift the results are similar but more stable, obtaining a slightly better performance.

However, if we setup bandwidth differentiation, requests start to being reordered, prioritized and burst opportunities start to arise. Figure 24 (left) shows how a request on a single object server obtains the 30MB/s and the other two requests, going to another object server share the 30MB/s

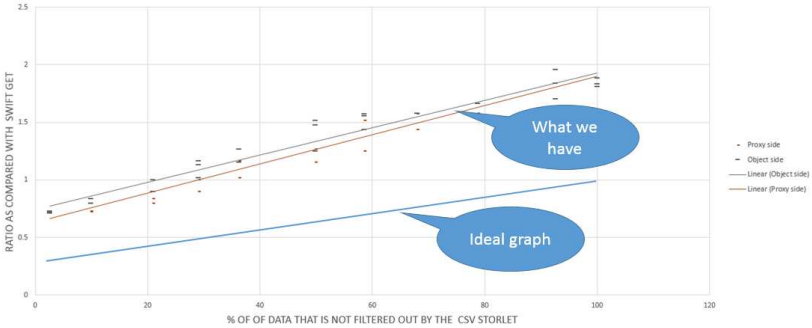

Figure 25: CSVStorlet performance relative to plain Swift GET.

assignment with 15MB/s each one. As observed, the Bandwidth obtained does not go higher than the required, due to the background I/O activity. However, the required level is guaranteed thanks to the low-level implementation that manages all the I/O in the node.

On Figure 24 (right) we setup two tenants with 50MB/s and 30MB/s bandwidth. Then each tenant asks for two objects. On this scenario each tenant goes to a different object server, we can see how OBJ1 and OBJ2 are sharing the BW as they enter at the same time, but OBJ4 is obtaining the maximum (50MB/s) until OBJ3 enters the object server. In that moment the two objects start sharing the Bandwidth.

9.4.3 Experiences with the Spark SQL push-down filter

Setting. The Swift cluster has 2 physical nodes: 1 proxy node and 1 data node where each node has 8GB memory and 4 cores and where the nodes communicate through a 1Gbit network. The Storlet middleware is typically configured so that the storlets run in the proxy node, however for comparison we also rerun some of the experiments when the Storlet is run at the data node. Spark compute cluster consists of a single node which runs Spark 1.5 augmented with the SQL pushdown mechanism (see Section 4.1.3). Typical experiments consist at querying a 1GB CSV file. The experiments where performed in an isolated environment: no interference from other load was applied to any of the nodes.

Standalone invocation of the CSVStorlet. Our first set of experiments consisted at invoking the CSVStorlet directly from the proxy node using a `curl` command (thus Spark is not involved). This permits us to analyze the Storlet behavior while removing interferences due to Spark. Then we present an initial analysis of an end to end experiment where a SQL query invoked within a spark shell is handled with through the SQL pushdown mechanism.

We analyzed the time it takes to process a 1GB CSV file through the CSVStorlet when we varied the percentage of the data that was filtered out by the Storlet. Fig. 25 gives the ratio of the time it took to invoke the Storlet as compared with the average time it took to perform a plain GET Swift of the full file without invoking the Storlet. These results are given both when the Storlet is run at the proxy node and at the data node.

First of all, the results show that the performance is quite independent of where the Storlet runs (proxy versus data node). This obviously assumes that the node is not by itself a bottleneck due to unrelated load. Not surprisingly this ratio increases with increasing percentage of data which is not filtered out. We observe that when we filter out 75% of the data it takes approximately the same time to retrieve the data as getting it with a plain Swift GET.

During these experiments, the network was not a bottleneck, however if it were, the advantage of running the Storlet would certainly have been more evident. In addition when the Storlet is invoked, the Spark side benefits from the following factors: *i)* Not having to filter again the data, less CPU is invested at running the SQL query, *ii)* Receiving less data, the memory pressure on the Spark cluster is diminished.

Predicate evaluation	Write the data to output	Process time
Yes	Yes	100%
Yes	No	92%
No	Yes	91%
No	No	90%

Figure 26: Identifying potential performance bottlenecks in the Spark SQL push-down mechanism.

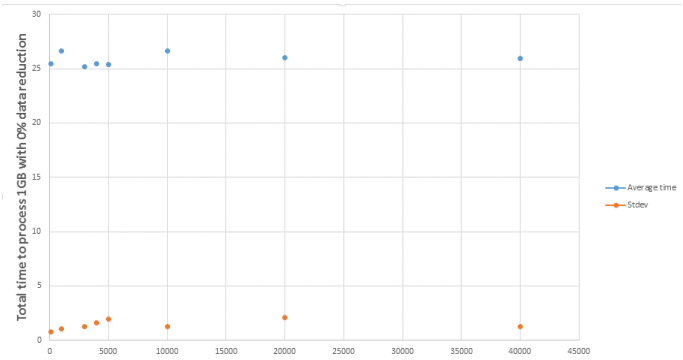

Figure 27: CSVStorlet performance related to the number of CSV rows processed per batch.

As we will see with the end to end experiments, the performance acceleration is much better than what this standalone graph can show.

Varying internal Storlet components to detect bottlenecks. We instrumented the Storlet code so that to skip the predicate evaluation (while taking care not to modify the output data) and/or the writing of the data to the output. Fig. 26 gives an idea of to what extend we could improve the Storlet performance by improving the predicate evaluation performance.

Analysis of the number of records processed per batch. The following graph gives the total time needed to process the 1GB targeted file as function of the number of rows processed by batch.

As can be seen in Fig. 27, the performance, at least in the sterile performance environment use, does not vary assuming when the number of records per batch varies between 100 and 40,000. Consequently we set the default value of this parameter to 1,000 so as to not to increase the memory pressure that could be felt when multiple Storlet invocations are run.

Concurrency analysis First we analyzed the effect of invoking concurrently the Storlet. Each invocation is still against a 1GB file.

We observe in Fig. 28 a linear degradation in the performance of the Storlet as concurrency increases. Resource analysis of the proxy node shown that it was CPU bound while the memory pressure was very light. This means that the total CPU resource of the proxy nodes should be adequately adjusted according to the expected Storlet invocation concurrency. This may be handled either by statically adjusting the CPU capacity of the Swift proxy nodes (that will run the Storlet tasks) or/and by dynamically adjusting the number of proxy nodes. The results obtained when concurrency is 6 will be clarified in the following paragraph.

Since Spark tasks typically address file ranges of between 32 and 128MB, we repeated the previous experiment but for smaller files (32 MB) where we measure the ratio of how long it took to invoke a

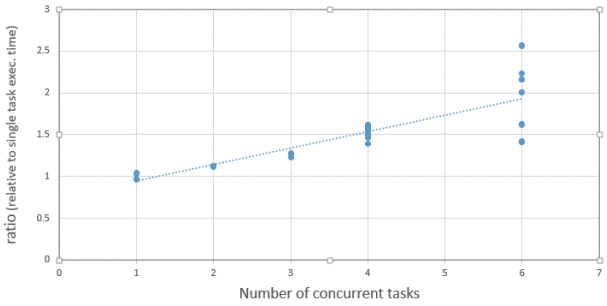

Figure 28: Concurrency influence for 1GB tasks.

single task with concurrency varying between 1 and 14 versus the average time that it takes to run a task with concurrency one.

The graph in Fig. 29 reproduces the results which can be explained as follows: First we have explain that we used the 5 default value for the *storlet_daemon_thread_pool_size* parameter[41], so that for a concurrency level of up to 5, we observe a (pseudo linear) degradation.

When we get to concurrency level to 6, one of the 6 tasks will be enqueued till the first task of the group of five tasks which got immediately scheduled completes. Thus between 6 and 11 we can observe that the classes are separated into two groups: The first one with approximately constant run time (for tasks who are scheduled immediately), while the second group consisting of the tasks who got enqueued, will run approximately as for a consistency level of 1 to 5 with the addition of the mean time to execute the tasks of the first group.

The same reasoning applies for higher number of tasks, so that between 12 and 15 the tasks are separated into three groups. We experience with varying values of the *storlet_daemon_thread_pool_size* parameter but could not observe improvements (we in fact observed a degradation). This is due to the fact that the specific proxy node that we use has 4 cores, so that 5 concurrent Storlet invocations is enough to saturate the CPU power use of the node. However, without any doubt this parameter should be tuned as function of the number of cores of the Swift nodes used so that proper concurrency can be reached.

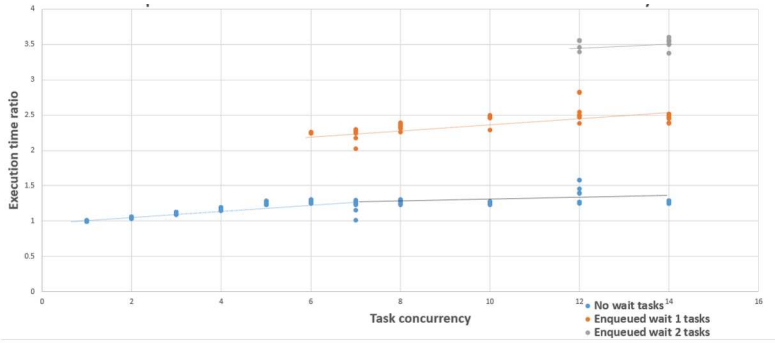

Figure 29: Concurrency influence for 32MB tasks.

End to end Spark to CSVStorlet invocations. First of all we compared the time to run a simple query (counting the number of records of a 1GB CSV file such that their 4th field was equal to a given value). We observed an acceleration factor of between 2 and 4. The specific experiment that we ran caused to filter out 98% of the data. We will further analyze the performance while varying that parameter in the near future. The SQL query spawned 33 spark tasks, each addressing a 32MB

[41] See `https://github.com/openstack/storlets/blob/master/Engine/swift/storlet_gateway/storlet_runtime.py`

range of the targeted file (which is slightly bigger than 1GB).
We analyzed for these experiments the timing of each of the 33 CSVStorlet invocations as detailed in Fig. 30 (left) and the invocation duration as detailed in Fig. 30 (right) where we can see, as expected that the time it takes to complete a Storlet invocation increases with the number of concurrent Storlet invocations.

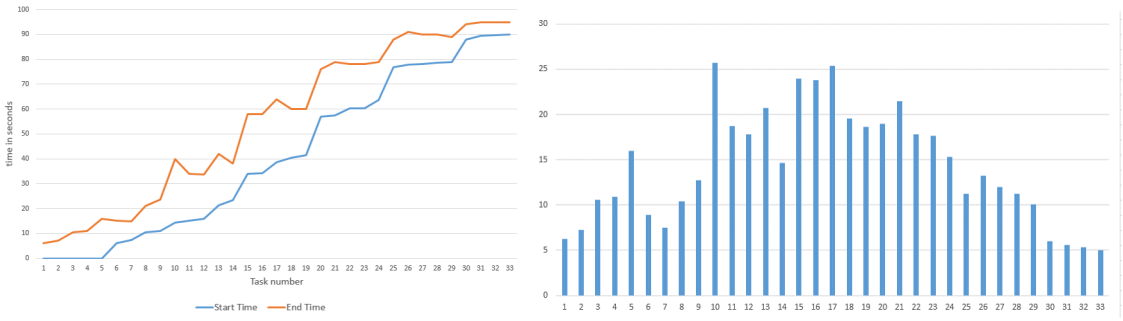

Figure 30: Storlet execution times.

9.4.4 Storage Filters in Block Storage

In order to test throttling, 4 block volumes were created on the consumer node. FIO was used as test tool to run a MB/sec and an IO/sec test. Fig 31 (left) shows the available BW/sec for the 4 volumes of approx. 64MB/sec aggregate and approx. 16MB/sec per volume. In a multi- tenant cloud configuration it is important that the data flow can be controller at either the BW/sec or IO/sec by the top level SDS application. A non-controllable block storage means very low predictability of when tasks will complete. The right amount of IO/sec and MB/sec must be allocated to the storage application and in particular low priority tasks must be throttled back in preference to higher priority tasks.

In Fig. 31 (right) the 4 volumes have been throttled to 3,5,7,7 MB/sec for both reads and writes. The measured results are practically 100% fit to the expected results. These throttle levels were chosen so that neither the network or storage were the bottleneck.

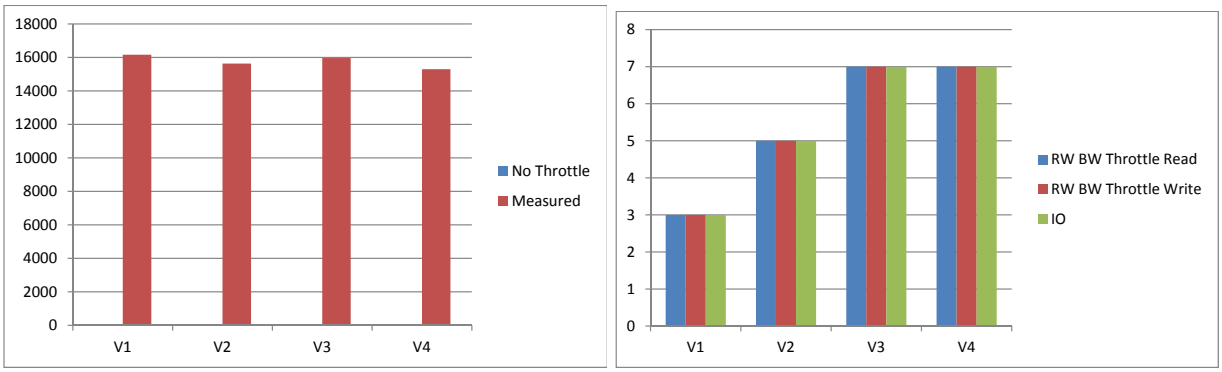

Figure 31: Default IO per second of tested volumes (left). Result of setting bandwidth throttling filter in MB per second (right).

In Fig. 32 (left) the 4 volumes have been throttled to 3K, 5K, 7K and 7K IO/sec for writes. The measured results are practically 100% fit to the expected results. These throttle levels were chosen so that neither the network or storage were the bottleneck.

In Fig. 32 (right) the 4 volumes have been throttled to 3K, 5K, 7K and 7K IO/sec for reads. The measured results are lower than expected. Work is ongoing to try and understand what is happening

as the throttle levels were chosen so that neither the network or storage were the bottleneck.

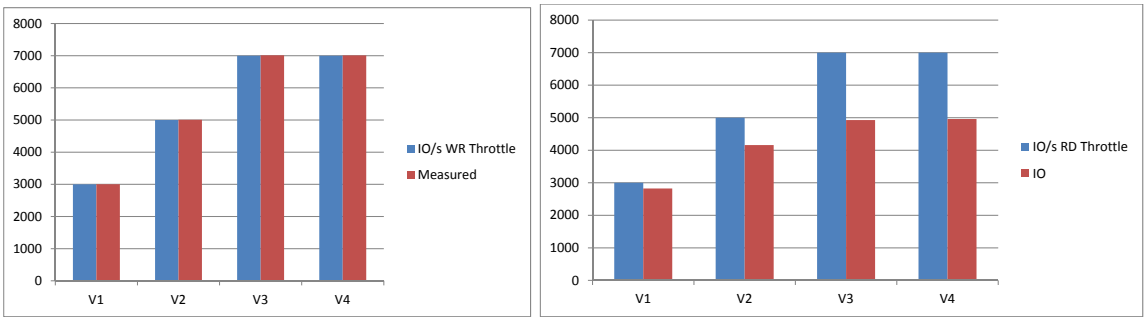

Figure 32: Results of applying the bandwidth throttling filter in IOs per second on volumes for writes (left) and reads (right).

10 Integration Among Building Blocks: Overview

IOStack has been planned to enable a natural convergence of the individual contributions of each partner into a single and unified SDS toolkit. Next, we describe and highlight the main interactions among partners that glue together the IOStack architecture:

- *Compute cluster hints to storage layer (EUR, MPS, URV)*: To integrate the disparate roles of compute and storage clusters, IOStack will enable both entities to cooperate for optimizing Big Data analytics in the cloud. Essentially, such a coordination is realized via cross-layer scheduling and provisioning strategies. Currently, we IOStack provides compute instances (e.g., VMs or containers) with means to communicate with the SDS Controller in order to retrieve *data locality* information. This will enable efficient co-located computations of ephemeral compute instances deployed directly on the physical machines where data is stored.

- *Object storage filter framework (BSC, IBM, URV)*: The filter framework for object storage is already integrating the individual contributions of various partners. In particular, the filter framework for the 1.5 year review will contain at least a data compression filter, a SQL push-down filter and a IO bandwidth differentiation filter. All these filters will be enforced via high-level storage policy definitions from the administration dashboard.

- *Dynamic storage policies (MPS, URV)*: The design of storage policies unifies the management of both object and block storage from the administration viewpoint. That is, the SDS Controller uses the same syntax and internal middleware to deploy and enforce filters via dynamic storage policies. Our dynamic storage policies framework is abstracted from the actual storage subsystem via a common monitoring framework and high-level REST APIs at the SDS Controller level.

- *Web Dashboard (MPS, EUR, ARC, URV)*: In terms of usability, IOStack aims at being a human-friendly framework easy to manage for datacenter administrators. The administration Web dashboard integrates the management of block and object storage subsystems, as well as real-time monitoring information of both the storage layer and the compute layer. It represents a point of integration among multiple building blocks, facilitating the management of the complete SDS system to an administrator.

- *Deployments and trace-based experiments (All)*: All partners, and specially the use-case ones, are doing joint efforts to define and extract workload traces to feed experiments with realistic data. Moreover, academic partners may benefit from these traces to find novel effects that could improve the operation of the SDS system.

11 Conclusions and Future Directions

In this document, we described the design and architecture of IOStack. From a bottom-up perspective, we first depicted our *framework to enforce storage filters*, for both object and block storage subsystems. Storage filters leverage a powerful SDS service for enabling general-purpose transformations on data flows. In IOStack, the management of filters (e.g., deployment, enforcement) is abstracted via *simple REST APIs*.

The filter framework API, as well as other federated SDS services in IOStack, are under the umbrella of a central management entity: *the SDS Controller*. In fact, the SDS Controller offers easy-to-use filter management and compute and storage cluster *monitoring information* facade, as well as a distributed and high-performance *metadata store* to preserve metadata related with the SDS layer. Such a set of functionalities lies behind an intuitive graphical *web dashboard*, that extends the original OpenStack Dashboard.

At the top of the SDS Controller operation, we find the *dynamic storage policy framework*. This mechanism enables datacenter administrators to write simple policy definitions that, in turn, describe a workload condition (e.g., `Throughput < 10MBps`) that should be satisfied to trigger the enforcement of a storage filter on a tenant's requests. From our viewpoint, dynamic storage policies greatly simplify the management of filters to administrators, providing automated provisioning to IOStack founded on live workload-based decisions.

Another innovative point in IOStack is the exploitation of cross-layer scheduling and provisioning strategies between disaggregated compute and storage clusters, typical of Big Data analytics in the cloud. As a first cross-layer strategy, we propose to provide compute instances with *data locality information* to perform ephemeral and efficient computations directly where data lives. As we describe next, these cross-layer strategies will bridge the gap between disaggregated storage and compute clusters; in IOStack, both *clusters cooperate* to achieve better coordination, accurate provisioning and higher resource optimization.

Apart from the architecture of IOStack, we describe the benchmarking framework that will assess the *correctness and performance of the resulting toolkit*. Our benchmarking framework combines both technologies and methodologies, including the active participation of use-case companies to leverage real-world traces for generating realistic workloads in our experiments. We also showed a battery of experiments that demonstrate both the usability of our benchmarking framework as well as the potential of IOStack.

In our view, the achievements of this deliverable represent a remarkable step towards the creation of an open-source SDS toolkit for Big Data. Furthermore, the progress of the project and the tight collaboration among partners draws promising milestones in the horizon. In particular, we describe the following innovative aspects that are related to the future architectural aspects of IOStack:

- **Filter pipelining for object storage**: Currently, one of the limitations of our filter framework for object storage is the lack of pipelining, that is, we can enforce only one filter per data flow. The next stages of development of our filter framework will clearly target the ability of *pipelining several Storlets for the same object request*. Moreover, this would be accompanied with advanced features in our storage policy framework to help the administrator to detect potential conflicts of pipelining disparate filters.

- **Decentralized controller strategies**: Today, the SDS Controller is devised as a single, centralized entity in charge coordinating the available SDS services. However, this architecture may represent a bottleneck for the system and it may exhibit communication inefficiencies in many situations. For this reason, we are exploring alternative designs in which the SDS Controller logic could be *decentralized and executed in form of micro-controllers* [21].

- **Storage Policy Stereotypes**: There are data analytics jobs that exhibit similar behavior and requirements. That is, the requirements of groups of jobs in a Big Data workload may be accurately described by a single *job stereotype* that encompasses a set of storage policies. Thus,

agents in the compute cluster may be used to *inform IOStack* of the requirements of incoming data analytics jobs (i.e., the stereotype they belong to), so that *the definition of storage policies to tenants is transparent to the administrator*.

References

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

[2] A. Alba et al., "Efficient and agile storage management in software defined environments," IBM Journal of Research and Development, vol. 58, no. 2/3, pp. 1–5, 2014.

[3] J. Mace, P. Bodik, R. Fonseca, and M. Musuvathi, "Retro: Targeted resource management in multi-tenant distributed systems," in USENIX NSDI'15, 2015.

[4] R. Gracia-Tinedo, D. Harnik, D. Naor, D. Sotnikov, S. Toledo, and A. Zuck, "SDGen: mimicking datasets for content generation in storage benchmarks," in USENIX FAST'15, pp. 317–330, 2015.

[5] D. Logothetis, C. Trezzo, K. C. Webb, and K. Yocum, "In-situ mapreduce for log processing," in USENIX ATC'11, p. 115, 2011.

[6] "IBM Storlets." https://github.com/openstack/storlets.

[7] E. Riedel, G. Gibson, and C. Faloutsos, "Active storage for large-scale data mining and multimedia applications," in VLDB'98, pp. 62–73, 1998.

[8] J. Piernas, J. Nieplocha, and E. J. Felix, "Evaluation of active storage strategies for the lustre parallel file system," in ACM/IEEE Supercomputing'07, p. 28, 2007.

[9] S. Rabinovici-Cohen, E. Henis, J. Marberg, and K. Nagin, "Storlet engine: performing computations in cloud storage," tech. rep., IBM Technical Report H-0320, 2014.

[10] J. Axboe, "Linux block io—present and future," in Ottawa Linux Symp, pp. 51–61, Citeseer, 2004.

[11] OpenStack, "Swift all-in-one." `http://docs.openstack.org/developer/swift/development_saio.html`.

[12] J. Axboe and A. D. Brunelle, "Blktrace User Guide," 2007.

[13] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "Paraver: A tool to visualize and analyze parallel code," WoTUG-18, pp. 17–31, 1995.

[14] R. Nou, "blktrace to Paraver trace converter." `https://github.com/mavy/blktrace-utils`.

[15] Python, "WSGI." `https://www.python.org/dev/peps/pep-3333/`.

[16] Q. Zheng, H. Chen, Y. Wang, J. Duan, and Z. Huang, "Cosbench: A benchmark tool for cloud object storage services," in IEEE CLOUD'12, pp. 998–999, 2012.

[17] L. Kernel, "I/O priorities." `http://man7.org/linux/man-pages/man2/ioprio_set.2.html`.

[18] E. Zamora-Gómez, P. García-López, and R. Mondéjar-Andreu, "Continuation Complexity: A Callback Hell for Distributed Systems," in LSDVE@EuroPar'15, 2015.

[19] L. M. L. Hilfi Alkaff, Indranil Gupta, "Cross-layer scheduling in cloud systems," in IEEE IC2E'15, 2015.

[20] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, et al., "Bigdatabench: A big data benchmark suite from internet services," in IEEE HPCA'14, pp. 488–499, 2014.

[21] R. Stutsman, C. Lee, and J. Ousterhout, "Experience with rules-based programming for distributed, concurrent, fault-tolerant code," in USENIX ATC'15, pp. 17–30, 2015.