HORIZON 2020 FRAMEWORK PROGRAMME

IOStack

(H2020-644182)

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

D2.3 Public release of the IOStack Toolkit

Due date of deliverable: 31-12-2016
Actual submission date: 31-12-2016

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

Summary of the document

Document Type	Deliverable
Dissemination level	Public
State	v1.2
Number of pages	46
WP/Task related to this document	WP2/T2.2
WP/Task responsible	URV
Leader	Raúl Gracia-Tinedo (URV)
Technical Manager	Gerard Paris (URV)
Quality Manager	Francesco Pace (EUR)
Author(s)	Raúl Gracia-Tinedo (URV), Yosef Moatti (IBM), Gerard París (URV), Josep Sampé (URV), Ramon Nou (BSC), Marc Siquier (BSC), Daniele Venzano (EUR), Pedro García-López(URV), Marc Sánchez-Artigas (URV).
Partner(s) Contributing	URV, IBM, BSC, EUR
Document ID	IOStack_D2.3_Public.pdf
Abstract	This deliverable describes the public release of stable prototypes of IOStack for analytics-as-a-service (Zoe), block storage (Konnector) and object storage (Crystal). We also describe how these prototypes are integrated within a single administration dashboard, resulting in a complete and unified toolkit. This document also reports the evaluation and results obtained from the validation of the object storage IOStack prototype using different use-case based workloads and benchmarks. We also delineate the future development and actions to be done in the last year of the project.
Keywords	Prototypes, specifications, evaluations.

History of changes

Version	Date	Author	Summary of changes
0.1	10-11-2016	Raúl Gracia (URV)	First draft version: Introduction, IOStack toolkit overview, Crystal description, future development.
0.2	11-11-2016	Ramon Nou (BSC)	Description of block storage filters.
0.3	14-11-2016	Gerard París (URV)	Dashboard/monitoring, Storlets section.
0.4	15-11-2016	Daniele Venzano (EURECOM)	Overview on Zoe.
0.5	22-11-2016	Marc Siquier (BSC)	Bandwidth Differentiation Methods Comparison.
0.6	22-11-2016	Yosef Moatti (IBM)	Revision of Stocator and Storlets sections.
0.7	24-11-2016	Gerard París (URV)	First version for internal review.
1.0	12-12-2016	Raúl Gracia (URV)	Revision after first round of reviews.
1.1	19-12-2016	Raúl Gracia (URV)	Revision after second round of reviews.
1.2	12-01-2017	Ramon Nou (BSC)	Removed Bandwidth Differentation Methods Comparison, added IO Priorities evaluation at Arctur.

Table of Contents

1 Executive summary

Exploiting Big Data analytics is a promising but still challenging step for many companies and organizations. In particular, the problems that may arise in this setting are related to i) *lack of advanced storage administration*, ii) *need of storage optimizations for analytics*, and iii) the *absence of flexible analytics virtualization tools*. This is specially true if we consider the large amounts of data that are usually managed by big companies in conjunction with the heterogeneity of today's analytics frameworks (e.g., Hadoop, Spark, etc.). Indeed, these problems may negatively impact on the storage life-cycle of Big Data, as well as the further data processing activity of analytics frameworks.

The IOStack project aims at solving these (and other) problems. In particular, IOStack advocates to develop policy-driven administration models and automation mechanisms for both storage and compute clusters. This greatly facilitates the management task of administrators in a Big Data cluster at scale. Moreover, a distinctive point of IOStack is to open storage and compute subsystems to be easily extended with new storage optimization algorithms and scheduling policies, respectively. This brings new opportunities to investigate novel optimization strategies to improve the performance and efficiency of complex Big Data workloads. To this end, the main outcome of the project is the *IOStack toolkit*: A Software-Defined Storage (SDS) Stack for Big Data analytics.

In the first part of the present deliverable, we describe the progress of the IOStack toolkit until month M24 and its software building blocks: *Zoe* (analytics virtualization), *Konnector* (block storage) and *Crystal* (object storage), as well as other components (administration dashboard, Stocator and Storlets). We illustrate that the toolkit is integrated and ready-to-use, with several deployments running. We also show that most of the software already implements code quality best-practices (testing, continuous integration) and has practical documentation for users and developers to use it.

In the second part of the deliverable we focus on Crystal: The first SDS architecture for object storage (OpenStack Swift). We describe how Crystal implements the *filter* abstraction to accommodate new storage optimizations that can be easily orchestrated via high-level policies. Moreover, we also demonstrate that Crystal exploits the control plane to dynamically react to changing workload conditions. We provide an extensive evaluation of Crystal in a 13-machine cluster under well-known benchmarks and trace replays of our use case companies (Idiada, Arctur). This will lead us to perform pilot deployment in the third year of the project. We also include a comparison of two bandwidth differentiation methods proposed on top of Openstack Swift: one that uses Linux kernel I/O priorities, and the other one that introduces variable delay time between chunks at middleware layer.

Finally, we overview the future development steps of the toolkit. Concretely, our objectives in the future will be to better exploit dynamic service provisioning and explore the convergence/cooperation of compute and storage building blocks to leverage cross-layer optimizations.

2 Introduction and Motivation

As the operation of companies and organizations increasingly involves more digital processes, their daily activity inherently generates larger amounts of *data* that should be stored along time. Such data has been recently acknowledged as *Big Data* given its volume, velocity and variety properties that, among other properties (5Vs), make their usage complex and resource consuming [1, 2]. There are a myriad of companies in diverse sectors — e.g., Internet of Things (IoT), Online Social Networks (OSNs), research institutions, automotive companies— that face the challenges of Big Data; these companies require scalable and practical means not only of storing such data, but also of extracting value from it.

To exploit these large amounts of data, Big Data *analytics* frameworks have rapidly become a key enabler technology increasingly involved within the business processes of many companies and organizations [3, 4, 5, 6, 7]. The reason for this phenomenon is simple: The parallel and scalable design of modern analytics frameworks represent a golden opportunity for diverse companies to effectively "extract value" from enormous amounts of data they produce. Nowadays, Big Data analytics frameworks provide rich suites of processing models, including MapReduce jobs, graph Data Bases (DBs) and parallel SQL engines, among others, which enable data scientists to explore and analyze large datasets [8, 9, 10].

The value extracted from large datasets may adopt different forms; for instance, in the case of analyzing logs of an e-commerce site, value may be a set of critical insights on how users behave within the site; or perhaps, if we consider a smart grid energy company like GridPocket, value may be instantiated as a deep understanding on the energy consumption of cities or even entire countries. Overall, this kind of information enables companies to take better and faster decisions, which make them more competitive than before.

2.1 Problems of Today's Analytics Platforms

Unfortunately, despite its potential, leveraging Big Data analytics at scale involves managing a complex ecosystem composed of storage systems and analytics frameworks, which has associated important administration and performance challenges, among others. In this project, we target the following ones:

Lack of advanced storage management for analytics. The first problem of Big Data analytics is at the storage layer, that is, where data lives. Most storage systems used in Big Data analytics (K/V stores, object storage, block storage) are scalable and provide high availability [11, 4, 12, 13], but lack from simplified and fine-grained management models. For instance, automation tools for storage provisioning and tiering (containers, block volumes) are still very early in many cases, which involves important efforts from an administrator viewpoint. Even worse, the lack of advanced management models prevents administrators from easily adapting the storage system to specific applications, in/out data flows, as well as providing specific Quality of Service (QoS) levels to tenants sharing a storage cluster.

Need for extensible storage optimizations: In a shared Big Data platform, the storage system may be subject to concurrent and heterogeneous workloads. For instance, we can imagine a storage cluster continuously storing data from IoT measurement devices or server logs. At the same time, one or many data analytics frameworks may be extracting data from the storage system for executing SQL queries. In this scenario, we believe that the storage system should be open to be extended with new optimization mechanisms to cope with such varied workloads. To illustrate this, if we retake the previous example and consider that IoT devices store compressible data, we could deploy a data compression or reduction technique on this particular data flow to save storage space. Similarly, in the case that an analytics application fetches a dataset to execute a SQL query, we could extend the storage system with a mechanism to discard useless data before serving it, which may significantly improve transfer performance.

Simple and definable deployment of analytics is a must. At the compute level, administrators deploying analytics frameworks waste important resources and time to perform the necessary configuration and setup tasks. If we consider a datacenter aiming at providing Big Data analytics in the cloud, it is clear that an advanced tool is needed for rapidly configuring and deploying a wide variety of analytics. Moreover, such a tool should be able of orchestrating and scheduling running analytics within a cluster in order to provide predictable completion deadlines.

In technical terms, these problems may negatively impact on the storage lifecycle of Big Data, as well as

the further data processing of analytics frameworks. Clearly, this may reduce the competitiveness of European companies aiming at benefiting from Big Data analytics, as they can face significant administration and optimization obstacles.

2.2 Goals of IOStack

Solving these (and other) problems is the main objective of the IOStack project. To this end, the main outcome of this project is the *IOStack toolkit*[1], which, among other assets, achieves the following key contributions:

- The IOStack toolkit materializes Software-Defined Storage (SDS) models that can overcome the lack of advanced administration and management capabilities of storage systems for analytics. This includes *policy-based provisioning and automation* features to greatly facilitate the task of system administrators to manage storage for analytics services. Moreover, real-time and off-line *monitoring tools* are available for administrators to take decisions based on workload characteristics.
- To meet the storage needs of modern analytics, our toolkit targets to open the storage system used in analytics workflows with non-anticipated functionalities and optimizations for improving the performance and efficiency of the whole Big Data lifecycle. We built a framework for object and block storage —OpenStack Swift and Cinder, respectively— that is capable of injecting new code, in form of an abstraction named *filter*, that enriches the functionalities of the system, even on-the-fly. System developers can also equip filters with *control algorithms* that dynamically change the behavior of the storage based on real-time monitoring metrics.
- At the compute layer, the IOStack toolkit leverages *automated and simplified deployment of analytics* frameworks. The concept of policy is also applied in this context: Administrators write simple deployment policies or descriptors to spawn entire clusters of analytics in seconds via lightweight container-based virtualization (Docker). Furthermore, running analytics instances can be then *intelligently scheduled* via policies that help administrators at predicting the execution of analytics in shared clusters.

As we describe in this document, the IOStack toolkit is a ready-to-use prototype, with several deployments currently running. We also show in this document (and in the other M24 deliverables) how the toolkit already solves problems in multiple aspects of the Big Data lifecycle of our use-case companies (Arctur, Idiada and GridPocket).

3 IOStack Toolkit: A Software-Defined Storage Stack for Big Data Analytics

In the following, we revisit the main design concepts already presented in Deliverable 2.2 to understand how they have been instantiated at each building block prototype. We also describe the current progress of the building blocks of the IOStack toolkit until M24.

3.1 Revisiting Design Concepts in IOStack

Following the principles of SDS, the IOStack toolkit is designed to decouple *control and data planes*. The control plane is intended to expose simple means for enabling administrators to orchestrate the underlying system, even resorting to intelligent, real-time algorithms. On the other hand, the data plane executes the actual logic on live workflows to enforce the services defined by the administrator at the control plane.

Concretely, to instantiate such a general design model, in IOStack we propose several abstractions. At the data plane we find the *metric and filter* abstractions; at the control plane, IOStack provides a *controller and policy* abstractions.

Filter[2]: In IOStack, a storage filter can be defined as a performance control or general-purpose data transformation that applies to specific data flows. This abstraction is quite general, as it can range from data compression or caching filters to IO bandwidth differentiation. Overall, the idea is that filters extend the functionalities of a storage system to meet requirements non-anticipated in their design.

Metric: This concept represents information of a particular aspect of the system operation at runtime. One can think in workload metrics that describe in real time some characteristics of the workload at hand, such as

[1] Available at `https://github.com/iostackproject`.

[2] This abstraction only applies to the storage building blocks, not for the compute building block of IOStack.

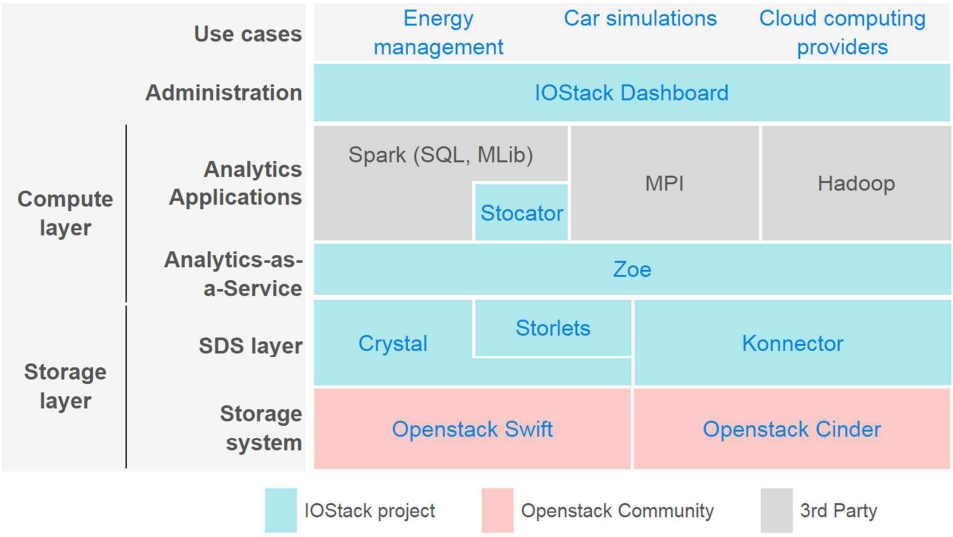


Figure 1: IOStack software stack.

the in/out bandwidth of a tenant, the number of IOPS of a volume, and so on. But metrics can also refer to the usage of the underlying resources, such as the CPU consumption or disk usage of storage servers.

Controller: A controller is an algorithm that receives as input workload metrics to manage the behavior of the system at runtime.

Policy: Contract with the SDS system to provision a service/resource to a tenant. This is the main mechanism for datacenter administrators to interact with the IOStack toolkit.

A key strength of IOStack is to simplify the management of data storage and analytics application deployments via *policies*. To enforce complex policies, the IOStack control plane builds a distributed layer to deploy arbitrary controllers; to wit, a developer can design a controller to manage the execution of storage filters, or controllers for dictating under which conditions an analytics deployment should scale. In this sense, IOStack controllers use runtime workload or resource metrics to take dynamic decisions; for instance, the IO bandwidth of containers/volumes or the CPU usage of compute instances. Moreover, in terms of storage, IOStack leverages the *storage filter* abstraction as a mean of executing computation of storage flows to optimize or provide added value services on specific workloads, such as data compression, caching or bandwidth differentiation.

In the following, we will overview the building blocks that constitute the IOStack toolkit, and we will describe how these building blocks mapped the previous design concepts in their respective domains.

3.2 Overview of the Toolkit

This section provides a high-level overview of the building blocks of the IOStack toolkit, as well as the relationships among them. As visible in Fig. 1, the architecture of IOStack toolkit can be divided into four main building blocks: *Administration*, *Analytics-as-a-Service*, *Block Storage* and *Object Storage*.

Administration. The IOStack web dashboard is the single point of access to all IOStack services. It integrates the other building blocks in a simple and functional web interface, allowing an administrator to manage the underlying services more easily via policies. The dashboard also integrates the monitoring services that present useful performance and activity information that can be used to take administrative decisions based on workload characteristics.

Analytics-as-a-Service (Zoe). Zoe is the compute component of IOStack: It provides a simple way to provision data analytics clusters and workflows using Docker containers. With Zoe, administrators can use the IOStack dashboard for launching complex data analytics frameworks in a few clicks, or use the APIs to call Zoe from your own scripts. Zoe is independent from applications. A generic application description language is used to build compositions of analytics services, define resource constraints and configuration options. For

example, a user can run Spark or MPI jobs on Zoe, by providing appropriate descriptions and Docker images. Moreover, Zoe exhibits high performance: it can create a fully configured Spark cluster, with 20 compute nodes and an iPython notebook in a few seconds. Zoe is built from the start to make full use of the available capacity in your Docker Swarm cluster. Not only Zoe is smart in placing containers, but when resources are exhausted, Zoe will queue new requests using state of the art scheduling algorithms.

Block Storage (Konnector). In IOStack, Konnector is the SDS framework for block storage (OpenStack Cinder). Konnector is instantiated on the "client side" and it provides to the consumer node applications a virtual storage device (VSD). The key feature of Konnector is to implement the filter abstraction at the block-level: A Konnector instance intercepts IOs from the client VM to the storage array and it can add arbitrary computations on the read/write path of these IOs, such as compression and encryption filters. From a design perspective, multiple Konnector VSD devices are managed by the SDS controller. The SDS controller itself is managed through a higher level programming interface and policies. The management of the virtual block storage controller is through the SDS controller, the role of which is to instantiate and control the virtual devices and their filters. The SDS controller keeps all the metadata of the virtual device, allowing the virtual device to be instantiated anywhere in the datacenter without regard to physical storage devices.

Object Storage (Crystal). Crystal is the first SDS architecture for object storage (OpenStack Swift) to efficiently support multi-tenancy and heterogeneous applications with evolving requirements. Crystal adds a filtering abstraction at the data plane (e.g., Storlet filters) and exposes it to the control plane to enable high-level, yet powerful, policies at the tenant, container and object granularities. Crystal translates these policies into a set of distributed controllers. As a result, Crystal offers a great deal of flexibility to dynamically adapt the system to the needs of specific applications, tenants and workloads.

Apart from the flagship building blocks, IOStack also contributed other components that are very important for the operation of the toolkit. On the one hand, in the context of IOStack we developed **Stocator**, a fast and efficient Spark driver to connect Spark to Swift. As we will see later on, Stocator enables us also to trigger custom computations on Swift from Spark, which can be exploited to accelerate analytics. On the other hand, in the IOStack project we open-sourced and contributed to the **OpenStack Storlets** project; this technology allows to execute sandboxed computations on Swift storage requests, opening the door to the implementation of a wide variety of filters in object storage.

It is worth mentioning that despite the fact that the IOStack building blocks are completely integrated and accessible within the administration dashboard, this does not prevent them to be exploited separately based on the necessities of a company. This model is flexible and maximizes the exploitation opportunities of the project's outcomes.

IOStack toolkit in action: Let us illustrate how these pieces work together in relation with the use cases of IOStack. From the administrator perspective, Arctur operators can rapidly deploy analytics applications in containers via the IOStack dashboard with few clicks, as well as monitor their execution, thanks to Zoe.

For example, Arctur administrators can deploy a Spark instance for GridPocket in order to execute SQL queries on IoT data stored in OpenStack Swift, which is being generated by their smart energy meters. With IOStack, connecting Spark to an object store like Swift is much more efficient than before thanks to the Stocator driver. Moreover, Crystal provides a rich layer of SDS services on top of the object store. That is, Arctur administrators can easily add a data compression filter to reduce the storage space consumed by smart meters storing redundant data. Even more, Arctur administrators can deploy active storage filters —implemented as Storlets for security and isolation— that optimize GridPocket SQL queries making the object store able to perform calculations close to the data.

As another usage example, IOStack enables Arctur administrators to deploy other types of applications, such as MPI parallel programs used by Idiada car crash simulations that manage data from block volumes. That is, apart from connecting to object stores, analytics applications virtualized with Zoe can also use block storage as a storage layer, either as physical volumes or by exploiting nested virtualization. In this scenario, the block storage part of IOStack helps Arctur administrators to deploy block volumes with different characteristics, according to a set of policies (e.g., network, storage tier, etc.). Moreover, IOStack also instantiates the concept of filter in the block storage world (OpenStack Cinder) thanks to Konnector: An administrator can mount a

Building Block	Filter	Metric	Controller	Policy
Zoe	N/A	● Docker/application metrics	◓ Distributed controllers	● Application deployment descriptors
Konnector	● Block filters	● Cinder volumes storage metrics	◓ Distributed controllers	● Block volume policies
Crystal	● Object filters	● Swift metrics framework	● Distributed controllers	● DSL-like policies on tenants/containers

●=Implemented and used, ◓=Available and future usage planned, N/A=Not Applies

Table 3.2a: Development progress of IOStack abstractions in the toolkit's building blocks.

volume for Idiada analytics with a pipeline of storage filters, such as caching or compression. This innovation allows cost reduction and performance improvements for analytics running in block volumes.

3.3 Development Progress of the Toolkit in M24

In this section, we discuss the development status of IOStack toolkit building blocks and the maturity of their respective abstractions. Table 3.2a may help the reader to understand the progress of the toolkit until M24.

Implemented abstractions in building blocks. As visible in Table 3.2a, most abstractions are already implemented in all building blocks of IOStack, according to our design.

First, all the building blocks already provide simplified *policy-based provisioning*; that is, the service layer that administrators can orchestrate is exposed via user-friendly policies and rules, avoiding the complexities of low level cluster management. As we will see later on in this document, Konnector, Crystal and Zoe help administrators to manage block volumes, object storage services and analytics deployments, respectively. For instance, Zoe offers policies in form of descriptors to deploy analytics applications, whereas Konnector enables to specify via its API or a dashboard form the creation and customization of block volumes. In addition, Crystal proposes a user-friendly DSL to provision SDS services in OpenStack Swift.

Second, all the building blocks also provide monitoring information to the control plane via the monitoring service, which is also presented to administrators in the dashboard monitoring panel. In the case of Zoe, the system can provide monitoring information of the Docker instances as well as the analytics running inside. Currently, Konnector gathers monitoring information of the filter framework installed at VMs mounting block volumes (IOPS, bandwidth); this opens the door to control the behavior of filters at the client side. Moreover, Crystal builds a metrics framework that allows administrators to deploy new metrics on-the-fly to orchestrate the behavior of controllers.

Moreover, in terms of storage, both Crystal and Konnector provide rich filter abstraction for both Swift and Cinder, respectively. In fact, both systems provide system developers with a filter framework that enables to develop code that intercepts storage flows.

Future development on dynamic provisioning. The last step in the IOStack toolkit road map is to fully exploit the dynamic provisioning of services, which is in turn the most complex aspect of the architecture.

The control plane of IOStack already provides means of deploying controllers that dynamically react to workload changes based on monitoring metrics. However, we have not yet exploited these controllers in i) Konnector to dynamically manage block filters, such as bandwidth control, and ii) Zoe to dynamically control the deployment of analytics frameworks, such as the scale in/out of instances based on resource metrics. In our view, the development efforts in these scenarios should be preceded by a research phase to discern when dynamism makes sense and can really benefit use cases. On the other hand, in the case of Crystal we already make use of controllers to dynamically manage the execution of filters.

4 Integrated Administration Dashboard and Monitoring

4.1 Administration Dashboard

The various building blocks that make up the IOStack toolkit are accessible through their respective APIs and can be independently integrated in third-party projects. In fact, this is a fundamental requirement in order to encourage the use of project results.

However, presenting all the building blocks together in an integrated toolkit offers the opportunity to visualize the relationships and interactions of the different components more clearly. To do so, we have extended Openstack Horizon, the canonical implementation of Openstack's Dashboard. Horizon provides a web based

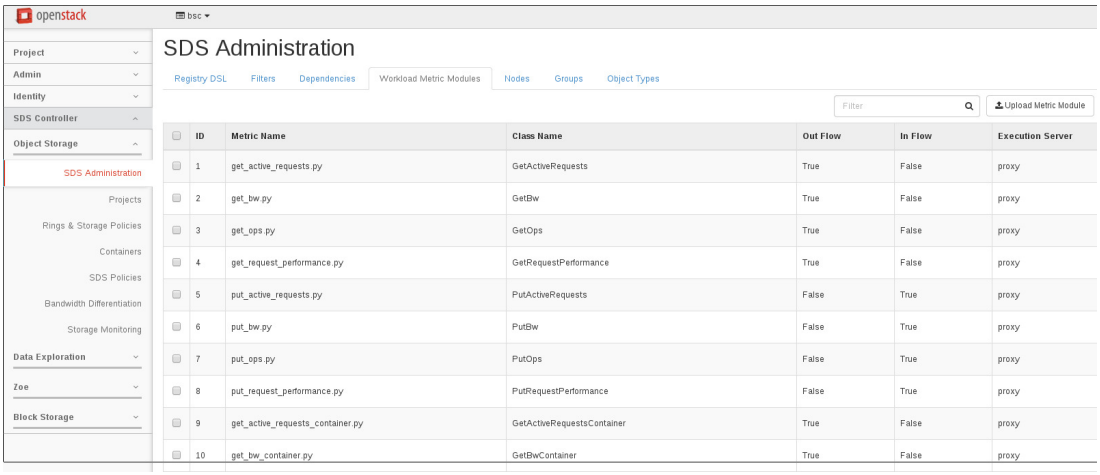


Figure 2: SDS Administration in the IOStack Dashboard.

user interface to various OpenStack services including Nova, Swift, etc. Our dashboard aims at providing a simple and functional interface to the toolkit, in order to ease its adoption by the enterprise community.

During the last year, all building blocks have been integrated in the web dashboard and all of them are now deployed and working in the Arctur testbed.

Our Horizon extension adds a new *SDS Controller* menu with four sections: Object Storage, Block Storage, Zoe and Data Exploration.

Object Storage section allows to invoke Crystal API actions from a web interface. It greatly simplifies the management of filters, metrics and policies. The administrator can upload new filters and metrics and define policies (using the Crystal DSL or a web form). The administration panel also offers more advanced features like creating groups of tenants or object types that maximize the specificity of the defined policies.

One of the main benefits of the object storage administration panel is that it helps figure at a glance which metrics are enabled or which policies are currently active. Information about the current status of the object storage cluster nodes is also provided at runtime, offering an option to restart them if they stop working.

This section also offers a pre-defined set of graphics that show real-time monitoring information (Fig. 3, left). This *Storage Monitoring* panel shows two different kinds of monitoring data: system resources usage and object storage activity. System resources data is obtained with collectd[3], a small daemon that runs on each host to be monitored, collects statistics about the system and provide mechanisms to forward the samples via the network to be centrally aggregated. Object storage monitoring information is obtained with our own metrics middleware that intercepts Swift requests and performs real-time measurements like the number of `GET` operations per second of a tenant (see Section 9.4. As we will see later on, all these monitoring information is sent to Logstash, stored in Elasticsearch and visualized with Kibana.

Block Storage section allows an administrator to define storage policies and storage groups, and to create SDS volumes based on these storage groups. Storage policies support filter pipelining, i.e. define a set of filters that will be executed one after the other for the same data blocks. Once a particular policy is defined, the administrator can create a storage group selecting the policy and the storage nodes that will form the group. Then, in the SDS volumes tab, the administrator can create an SDS Volume for this group. Once this volume is attached to an instance, read/writes to this volume will have the previously defined policy applied to them.

The block storage section also offers a monitoring panel, that with the help of additional code based on Elasticsearch and Kibana, helps to monitor performance of individual volumes and of underlying RAIDs and disks in the storage nodes.

Zoe integration allows an administrator to create application executions by configuring the number of workers and memory limits. The Zoe executions panel offers details of each execution like the scheduled time, the status, and information about each service endpoint URL.

[3] https://collectd.org

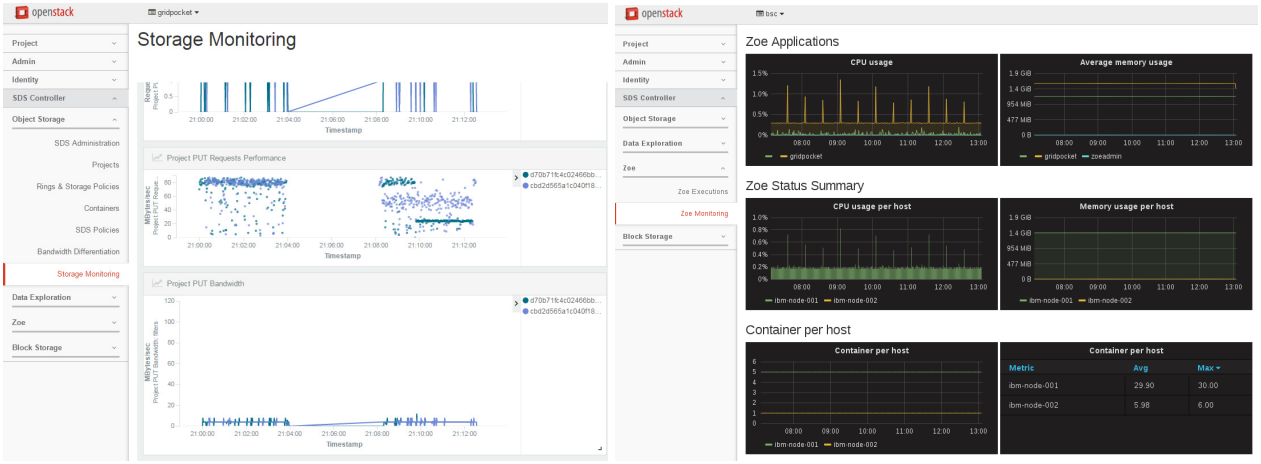


Figure 3: Object storage Monitoring in the IOStack dashboard (left). Zoe Monitoring in the IOStack Dashboard (right).

Monitoring is also integrated in this section (Fig. 3, right), showing graphical information of CPU and memory usage at the application and host level, as well as the number of containers per host. This monitoring solution was built combining 3 open source projects and custom tools developed to cater to IOStack specific needs; as it was explained in Deliverable D5.1, collectd was chosen for metrics gathering, Carbon/Graphite for storage, and Grafana for metrics visualization.

Finally, in **Data exploration** section, we use Kibana as an off-line monitoring tool (*data exploration*) that allows an administrator to explore data and create new plots and dashboards that may be useful to take decisions based on workload characteristics. Kibana is an open source data visualization plugin for Elasticsearch. It provides visualization capabilities on top of the content indexed on an Elasticsearch cluster. Users can create bar, line and scatter plots, or pie charts on top of large volumes of data. These plots can be combined to create custom dashboards that help administrators get an overview of the system.

4.2 Monitoring

Summarizing what we have outlined in the previous section, we see that the monitoring is a key part of IOStack toolkit. Monitoring data is used to:

- Define dynamic policies that react on workload changes and to have a control feedback to implement these policies.
- Have real-time monitoring in a web dashboard.
- Have off-line monitoring tool that allows to explore the data and helps administrators decide which policies can be applied to improve system performance.

In figure 4 we can see the monitoring data flow from the different components that collect or generate data, to the components that store and consume them. CollectD is used by the different building blocks to collect monitoring information from storage or compute systems. IOStack control plane captures monitoring information with RabbitMQ, a Message Oriented Middleware (MOM) broker that provides high-performance event processing service.

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, we ease the consumption of monitoring events from the viewpoint of workload metric processes.

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

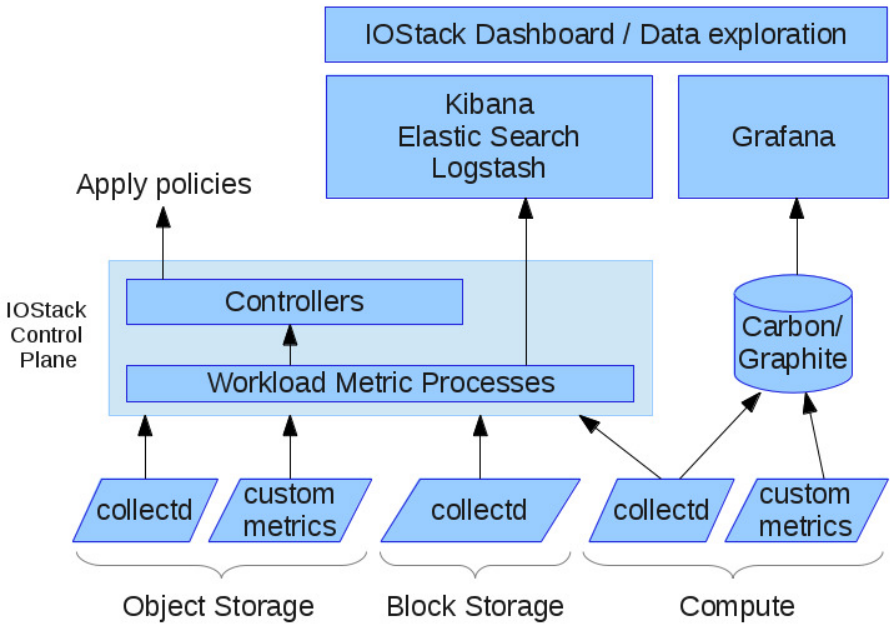


Figure 4: High-level overview of IOStack monitoring layer. As can be observed, the IOStack control plane serves as a centralized recipient for most monitoring information sources. Such monitoring information is then consumed by control algorithms that may dynamically change the behavior of building blocks.

about the physical resource utilization, we generate monitoring information concerning the OpenStack service at hand (e.g., Swift). That is, 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 GET/PUT requests 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. For instance, we could monitor the *compressibility* of data objects that are transferred to/from the system. Such kind of policies can be achieved in IOStack by introducing processes than generate the desired metrics and send the values to RabbitMQ. We already demonstrated this feature by providing one of such custom metrics regarding the IO bandwidth exhibited by tenants and containers.

IOStack monitoring provides a high degree of precision related to the storage monitoring, defining monitoring events at the tenant and container/volume granularity. This high degree of precision allows workload metric processes to enable triggering policies also at the tenant and container/volume granularity, because there is a degree of dependency between the resolution of monitoring information and the definition of dynamic storage policies.

As depicted in figure 4, workload metric processes also send monitoring information to an Elastic stack (Logstash, Elasticsearch and Kibana)[4]. Logstash is a dynamic data collection pipeline that is able to ingest data from a multitude of sources simultaneously, transforms it, and then routes it to a variety of outputs (in Elastic stack, data is sent to Elasticsearch). Elasticsearch stores all the monitoring data and acts as the search and analytics engine. On top of the content indexed by Elasticsearch, Kibana provides dynamic visualization capabilities. As said in the previous section, Kibana is used both for real-time and off-line monitoring.

Finally, it is worth mentioning that control plane gets data from all building blocks. This make it possible to consider the application of "cross-layer" optimizations, for example between compute and object storage building blocks.

[4]Zoe also provides standalone monitoring for custom metrics. The reason for this is that the IOStack control plane and Zoe had parallel development paths to avoid dependencies at the beginning of the project. We plan to integrated in the IOStack control plane all the monitoring components of the project to offer a unified version of the toolkit.

5 The Zoe system

Web page	http://zoe-analytics.eu/
Source Code	https://github.com/DistributedSystemsGroup/zoe
Documentation	http://docs.zoe-analytics.eu/
Continuous Integration	https://travis-ci.org/DistributedSystemsGroup/zoe
Mailing List	http://www.freelists.org/list/zoe

Zoe was created in August 2015 as an open source project [14] to satisfy the need of having an easy and integrated way to deploy distributed analytic applications on a cluster of physical or virtual machines. Users can define analytic applications starting from a number of ready-made building blocks, Zoe will schedule and deploy them matching resources requests and availability.

Zoe is developed in Python and is conceived as a thin layer that builds on top of an existing low-level cluster management system, which is used as a back-end to provision resources to applications. Raising the level of abstraction to manipulate analytic applications is beneficial for users and ultimately to the system design itself: application scheduling decisions can be taken with a small amount of state information, and do not happen at the same (extremely fast) pace at which low-level task scheduling does.

Next we overview Zoe's software design and implementation. In the final part of this section we will describe the role of Zoe in IOStack. Deliverable 5.2 contains more in-depth information about Zoe, including the road-map and community engagement information.

5.1 Zoe applications

Zoe schedules applications. Each application is made of one or more components, that run each in its own Linux container. For example, the Spark Notebook application that a user submits to Zoe is made of one Jupyter Notebook[15] component, one Apache Spark[16] master and one or more Apache Spark workers.

In order to produce useful work, in this case for the application to be useful to the user, there is a core set of components that can be identified: the notebook, the master and just one worker. The application can mark additional workers as "optional" (elastic in Zoe's terminology). Zoe will start them only if there are free and unused resources.

To simplify application descriptions and build a library of building blocks, an intermediate concept of frameworks (groups of components that work together) has been introduced.

Zoe applications are described by users via a simple JSON description that follows a high-level configuration language (CL) to specify applications, frameworks and components with their classes (core or elastic), resource reservations and constraints. The CL is simple and extensible: it aims at conciseness and, with framework templates, can be also used by "casual", in addition to "power" users [17]. An example of the simplicity and effectiveness of the Zoe CL, building a batch application for the distributed version of Tensorflow[18], only required less than 25 lines of CL.

A typical Zoe application description contains a number of application-global metadata items, such as size information for the size-based scheduler. Then it contains a list of the components, each with its own metadata, that includes:

- image: the Docker image location for the component
- environment: the environment variables that are used to configure the component
- volumes: external storage to mount inside the container
- resources: the resource reservations required to run one instance of this component
- total count: the total number of instances of this component that can be started
- essential count: the minimum number of instances (out of total count) required for the application to produce useful work

Currently a number of scripts help users create these application descriptions, but we are planning and developing a web-based tool in the style of an app-shop where users are free to compose and configure their own applications in a more intuitive way.

5.2 Internal architecture

Zoe is divided into two multi-threaded processes. The Zoe Master and the Zoe API. Both store state information into an external Postgres database, that has well-known reliability and fault tolerance characteristics.

Users interact with the Zoe API process. It offers the web interface and the REST API and does an initial validation of user input and application descriptions.

The Master process contains the scheduler and other threads to process events generated by the back-end, manage asynchronous application termination and respond to requests coming from Zoe API. The two processes use a Zero-MQ based protocol, that has been developed taking in all the recommendations to build a robust protocol in the face of crashes or disconnections.

The Master and the API processes do not store any state internally and can be restarted at will in case of upgrades or crashes without any consequence. The API process can be scaled horizontally behind an http-based load balancer.

5.2.1 State

Three main tables are maintained in the state store, the platform table, the executions table and the service table. In this context, executions are instances of Zoe applications, while services are instances of components.

The platform table stores information about the number of containers and the amount of free resources for each node available in the cluster. This information is taken as-is from the back-end API and stored into the database for caching reasons. Each entry describes the resources (memory, cores and containers) free and total for one single node. Data is updated whenever the scheduler is triggered.

Each entry in the execution table refers to a single application execution submitted by the user. It records information such as:

- identifier of the user who submitted the application
- timestamps of submission, start and termination events
- current status (submitted, running, error, etc.)
- error message in case the execution failed due to an error

Each entry in the services table refers to a single instance of a component. As was explained in the previous sections, application descriptions contain a list of component with total and essential counts for each of them. These descriptions are exploded into the state table in as many services as is the total number of instances requested. The information recorded for each instance is:

- status: the service status from Zoe point of view
- backend status: the container status from the back-end point of view
- backend identifier: the unique ID the back-end uses to identify this container
- service group: the component name as given into the application description
- error message: filled in when there is an error during the container lifetime (for example image not found or out-of-memory termination)

5.2.2 Scheduling and placement

The scheduler thread is triggered by several events:

- an execution being added to the scheduling queue
- an execution that terminates, either by itself (batch) or by the user

- a timer, to account for resources that are used outside of Zoe's control.

When the scheduler is triggered and selects an application execution to start, according to the configured policy, it performs a placement simulation, trying to find the best fit of core and elastic services in order to maximize the number of running executions. Executions for which all core services cannot be started at the same time are left in the queue, in order to start only executions that can produce useful work.

The placement simulation uses a simple filter-and-sort algorithm. For each container to be placed it filters out nodes that do not match the required resource constraints (amount of memory, but also special hardware needs can be taken into account). The container will be placed in the node with the least amount of running containers and the biggest amount of free memory.

This procedure is repeated for each service until either:

- all core services are placed: then the execution is started, following the simulated plan
- a core service cannot be placed: the execution is re-added to the queue and the simulation results are thrown away

The same filter-and-sort algorithm is used to take placement decisions in most modern cluster manager systems, like OpenStack Nova and Docker Swarm.

5.3 Back-ends

The main design idea of Zoe is to hide the complexities of low-level resource provisioning from application scheduling and use an existing cluster management system, for which many alternatives exists, instead. Currently, Zoe builds on top of Docker Swarm and uses it for orchestration, dependency management, resource isolation, naming and networking.

5.4 User interaction

Users can interact with Zoe through a web or a command-line tool. The web interface provides an high level overview of the application executions for each user and their status. The command-line tool is more advanced and, for example, can be used to script the execution of multiple batch applications, creating simple automated workflows. Zoe has also a REST client API that can be used to develop more tools and services.

When an application is submitted, via REST, command-line or web interface, Zoe creates an entry in the application state store, and adds it to a pending queue. Our system allows plugging several scheduling policies to manage the pending queue, ranging from simple to sophisticated size-based strategies. The scheduler strives at making sure the application selected for execution can make progress as soon as resources are allocated to it: to this end, it relies on the back-end to place all core components according to the simulated plan. Elastic components are scheduled when possible and contribute to decreased application run-time. As a consequence, batch applications (either rigid like Tensorflow or MPI, or elastic such as Spark) can make progress as soon as core components start; similarly, interactive applications – which can be given precedence to reduce queuing times and improve user experience – can also start being used even if not all elastic components (if any) are scheduled.

5.5 Zoe and IOStack

IOStack promotes the separation of compute and storage layers allowing increased flexibility in meeting the variable demand of computation resources, while keeping the data storage system in a stable state. Zoe targets the compute layer of IOStack, by providing a simple way to define and deploy arbitrary analytic applications.

Zoe has been integrated into the IOStack dashboard to give users a single point of access to all IOStack services. Zoe applications can use the storage services offered by the other IOStack components directly (Swift) or through the back-end (volumes). In the last year of the project we envision an even tighter integration with information flowing from the Zoe scheduler to the SDS controller and vice-versa, opening the possibility of automatically taking informed decisions about application scheduling and storage features at run-time.

6 Stocator: A Fast Spark Connector for Object Stores

Web page	https://spark-packages.org/package/SparkTC/stocator
Source Code	https://github.com/SparkTC/stocator
Documentation	https://github.com/SparkTC/stocator/blob/master/README.md
Continuous Integration	https://travis-ci.org/SparkTC/stocator/

Apache Spark can access multiple data sources that include object stores like Amazon S3, OpenStack Swift, IBM SoftLayer, and more. To access an object store, Apache Spark uses Hadoop modules that contain drivers to the various object stores.

Apache Spark needs only a small set of the object store functionalities. Specifically, Apache Spark requires the following operations: listing the containers, listing the objects of a given container creation, object read, and getting data partitions. Hadoop drivers, however, must be compliant with the Hadoop eco system. This means they support many more operations, such as shell operations on directories, including move, copy, rename, etc. which are not native object store operations. Moreover, Hadoop Map Reduce Client is designed to work with file systems and not with object stores. The temporary files and folders it uses for every write operation are renamed, copied, and deleted. All this leads to dozens of useless requests targeted at the object store. It's clear that Hadoop is designed to work with file systems and not object stores.

Stocator, although implementing the Hadoop is implicitly designed for the object stores, it has a very different architecture from the existing Hadoop driver. It does not depend on the Hadoop modules and interacts directly with object stores.

Stocator is a generic connector, that may contain various implementations for object stores. It was initially provided with complete Swift driver, based on the JOSS package[5], however it can be very easily extended to more object store implementations.

7 Konnector: SDS for Block Storage

Source Code	https://github.com/MPSTOR/Konnector
API specifications	https://github.com/MPSTOR/Konnector/blob/master/Konnector-API.adoc

In many scenarios, the execution of data analytics jobs involve alternative storage substrates, such as *block storage*[C]. In general, compute nodes executing analytics and *storage arrays* are physically disaggregated, which simplifies the management of a datacenter infrastructure. Thus, similarly to the usage of a physical disk or SSD, compute nodes interact with a storage array via standard block-level IO protocols (e.g., iSCSI) that operate through a high speed network link (e.g., 10GBps, fiber channel).

Once provided a block storage communication protocol, compute nodes can mount *logical volumes* that map physical storage space of the storage array for managing data. For instance, in the Idiada use-case, data scientists execute car crash simulations on VMs making use of file systems that mount volumes on large storage arrays. This makes life easier for Idiada's data scientists: they write analytics code that manages data from a large storage cluster, just as they would do in a file system of a single machine on top of a traditional HDD.

Although being a commercial standard, block storage arrays are normally proprietary closed systems, often complex to manage, and they are perceived as expensive. But more importantly, storage arrays being mission critical systems are slow to evolve, which results in an *inflexible innovation platform*. In contrast to block storage arrays, compute nodes have an open software architecture and they can execute an increasingly varied types of analytics. This mismatch is true even in the case that some storage arrays can be delivered with a fixed set of built-in storage features —e.g., de-duplication or replication—, given that implementing a new feature or adapting the existing ones to the particular necessities of Big Data analytics is not practical.

In IOStack, the SDS toolkit provides a framework to *manage block storage arrays* (SDS Gateway) and *extend their functionality* by providing a filter stack on the open compute platform (Konnector). In the following, we briefly overview these components; a full description of the SDS toolkit for block storage can be found in deliverable d3.2.

[5] `http://joss.javaswift.org`

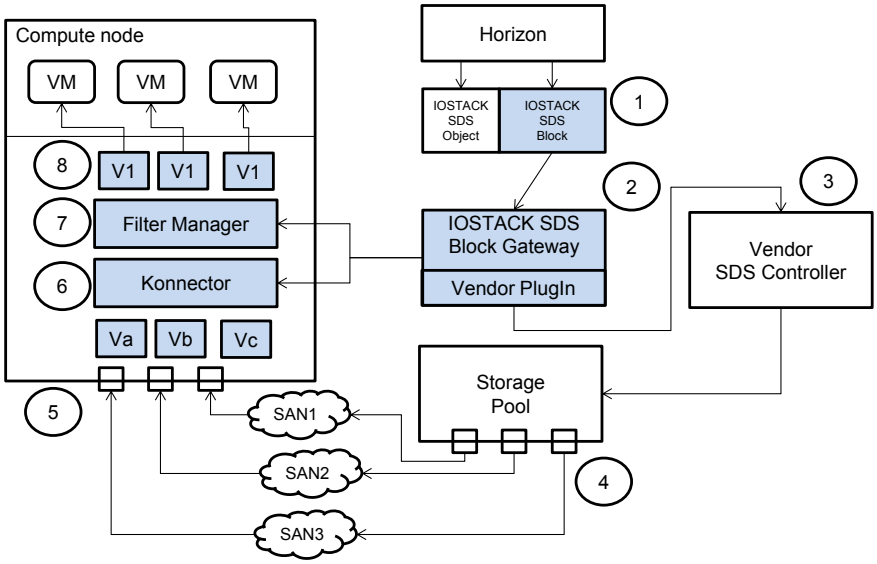


Figure 5: Overview of the SDS Gateway and Konnector block storage components.

7.1 SDS Gateway: Advanced Storage Provisioning and Automation

The IOStack SDS Gateway is the tool that enables administrators to manage storage volumes in an automated way. In particular, in an OpenStack environment, it allows an administrator to provision block storage according to a defined policy. Following the principles of SDS, the SDS Gateway allows the user to choose a volume type, this volume type is tagged to the compute group within which is configured with a set of storage arrays, storage tiers, fabrics and consumer node filter functions. When a volume is created within this storage group the SDS Gateway virtualizes for the user all the operations of choosing which storage array, fabric and media tier to use. This greatly simplifies the process of provisioning storage. The second step of attaching a storage array volume and creating on the consumer node a filter stack is also fully automated process. These complex provisioning tasks reduce the cost of provisioning managed storage by removing the need of the user to know anything about the datacenter internals and simply use the services created by the the datacenter administrator. This is specially true considering that all these options are exposed within the IOStack administration dashboard.

Technically, the SDS Gateway provides a REST API for a client to manage storage. In Fig. 5 the client is the OpenStack Nova Cinder controller. The Cinder API commands are translated and forwarded to the SDS Gateway. The SDS Gateway maintains an object model which is configured by the Horizon Dashboard SDS plugin.

The SDS plugin for the Horizon dashboard creates storage groups, a set of storage nodes are added to each storage group. Associated with each storage group is a policy. The policy configures configures default storage options for all volumes provisioned within that storage group. Some of the options are storage node specific, such as the fabric to export the volume on, other options are specific to the compute node —namely, consumer node— where the storage volume is attached to.

The consumer node specific component is a description of the filter that that is dynamically built once the storage volume has been attached to the consumer node. Once a storage volume has been attached to a consumer node, the SDS Gateway uses the Konnector API to dynamically create a filter stack between the attached volume and the volume presented to the final consumer, for example a VM. This operation is shown in Fig. 5 call-outs 5,6,7,8. This schema allows storage volumes to be created on storage nodes and a stack of dynamic storage functions to be applied to the storage volume independent of the storage node.

As we show next, the Konnector component leverage compute node flexibility by moving storage functions (filters) into the compute node.

7.2 Konnector: Extending the Functionalities of Block Storage

The goal of the filter stack is to provide a flexible platform for innovation and value added functions on top of the storage array volumes. That is, by moving the data flow from the kernel space into the "user space" filters can be dynamically created at run time. Run time creation of the filter stack in mandatory because the volume and its stack are only instantiated when the storage array volume is attached to the consumer node or VM.

As visible in Fig. 5, the filter stack is a layer between the terminated storage volume on the compute node and the consumer of the storage volume, such as a virtual machine. A filter stack can provide functionality such as encryption, compression de-duplication on the data flow between the consumer and the storage array.

In our implementation, the filter stack is dynamically created as each filter is implemented as a `.so` (a dynamic linked library). The Filter manager when requested through the Konnector API will dynamically build the filter stack using a set of `.so` dll library functions. This approach allows the filter stack to be built on demand, it also allows any third party to create filter functions and add an IO processing function into the data flow between the consumer (e.g., Virtual Machine) and the storage array volume. The storage array volume is agnostic to the filter function since the storage array just sees data in/out of the attached storage array volume on the consumer node, it does not see nor is aware of any of the filter transformations.

From a developer perspective, filter objects are built from C source files. The SDS toolkit provides a number of skeleton filter samples which serves to act as a template for filter development. Developers have several API functions to transform IO operations intercepted by connector:

- `write_xform`: This function exists to perform a transformation at its defined filter level on payload write data bound for the device.
- `read_xform`: It is called at the same predefined filter execution level and is designed to act on the read payload data from the device en-route to the initiator.
- `pre_read`: We can redirect reads to another zone of the disk.
- `pre_write`: We can redirect writes to another zone of the disk.
- `get_name`: This function can be called from the controller. This just returns the name of the filter which is defined as a static string in the filter object. This is just added for debug purposes and is called when the filter daemon is executed in debug mode.
- `pass_args`: This function allows the user to pass arguments to the filter function when the filter is instantiated. The arguments can be specified using the syntax of the filter specification in the Horizon dashboard.

Next, we describe some of the advanced filters that we developed making use of these API calls to improve the Big Data processing of our use cases.

7.3 Block Filters Designed with Konnector

Source Code	`https://github.com/bsc-ssrg/BlockStorageFilters-IOSTACK`

In the basic package of block filters, we include simple proof-of-concept ones such as a `xor` filter that transforms each byte of a stream with an "exclusive OR" transformation, or a `noop` filter that simply intercepts the IO flow without actually manipulating it. While useful for basic testing purposes, we still need to exploit the filter framework to develop real-world filter for our use cases.

For this reason, we developed filters that go a step further of the original behavior of the filter framework. The original behavior was a 1:1 block transformation (reading or writing), however, in our scenarios we need to be able also to perform *n*:*n* transformations. For example, data prefetching or caching filters need to check before the actual read occurs whether the content is available or not, then the framework knows if the read should be issued to the storage array. On the other hand, in the case filters that require to modify the IO flow, such as in the case of data compression and deduplication, require to manage their own metadata space transparently to the real block device; this is needed it writes the real data it should know if the data should be

written or not. As one can infer, the design of these filters may provide performance and cost reduction benefits to real use cases, but are challenging to materialize efficiently.

In particular, we developed the following filter for our use cases and for advanced evaluation purposes:

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

Prefetch filter. This filter preloads data in advance to reduce network latency. `prefetch` filter is divided into two filters, the first one logs the offset and sizes of the data that is being used in the VM. The second filter preloads the data in advance. We are also going to develop a new filter on the next period that will enable just-in-time prefetching so the blocks are only preloaded just when they are going to be used.

Cache filter. This filter stores any block read into its memory space. Its objective is to increase the performance on workloads that are being read twice or more times. The cache is reduced using deduplication (`dedup`) and compression (`compress`), so we can store more data with less memory usage and surpass the buffer cache space.

Output modification filters. These filters generate a different output from the original one, such as compression or encryption. The main difference is that the filter is persistent, so the output can only be read if the same filter is used. `ocompress` generates a compressed file system using two compressors (for Idiada use case). The filter is transparent for the user. However, further modifications in the filter framework are needed to allow to export a, i.e., 6GB real volume, and present it inside the VM as a 10 GB volume. The main issue is that the increment of space should be static and in advance (using some % gathered or introduced by the user) and can not be changed as it will confuse the VM operating system.

Evaluation filter. This filter tries to expose several "turning knobs" or parameters to evaluate the framework; for instance, we can emulate latency delays or CPU usage delays in the filter workflow. `mockup` filter has those parameters, so delays are introduced on each read or write request.

8 The Storlets Framework

Web page	https://wiki.openstack.org/wiki/Storlets
Source Code	https://github.com/openstack/storlets
Documentation	http://storlets.readthedocs.io/
Tests	Unit tests and functional tests
Continuous Integration	https://github.com/openstack-infra/project-config/blob/master/jenkins/jobs/projects.yaml

In the IOStack project we open-sourced and contributed to the **OpenStack Storlets** project, that was used to implement a wide variety of filters for object storage.

Storlets is a framework to intercept an execute sandboxed code on object requests in OpenStack Swift. Storlets provide a powerful extension mechanism to OpenStack Swift —without changing its code— to run computations close to the data in a secure and isolated manner making use of Docker. With Storlets a developer can write code, package and deploy it as a regular object, and then explicitly invoke it on data objects as if the code was part of the Swift's WSGI pipeline. Request interception can occur not only at the proxy but also at the object servers thanks to the Storlet's WSGI middleware integrated in Swift, which "wraps" storage requests and responses.

At the high level the storlet engine is made of the components described below:

8.1 The storlet middleware

The storlet middleware is a Swift WSGI middleware that intercepts any given storlet invocation requests and reroutes the stream of data through the Docker container in which the specified storlet is executed. The storlet middleware is deployed both in the proxy-server and the object-server pipelines.

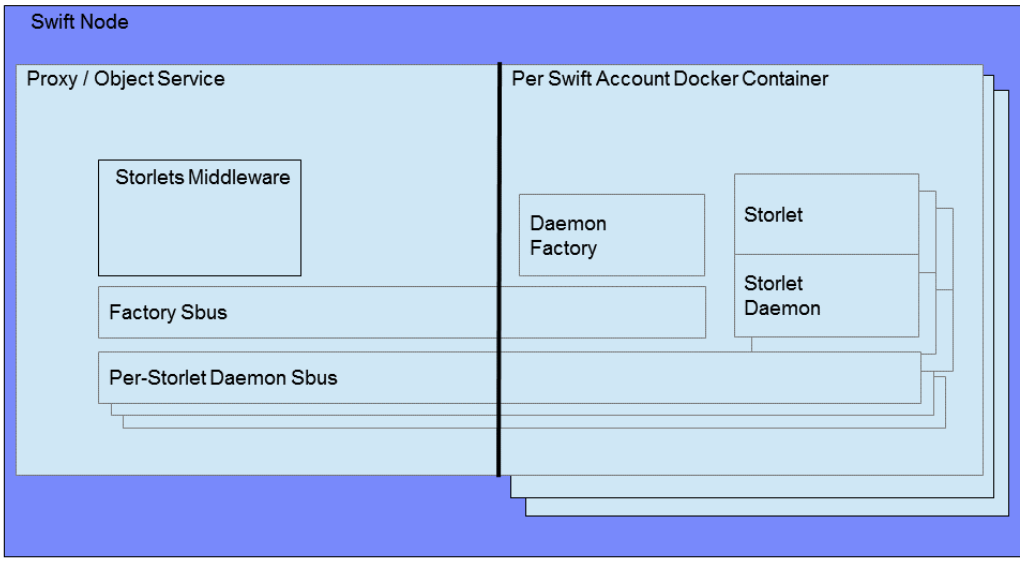


Figure 6: Storlets architecture

The storlet middleware is written in a way that allows to extend the engine to support sandboxing technologies other than Docker. All what is required from given sandbox is to implement the "storlet gateway" API which defines the functionality to run storlets.

8.2 Swift accounts

The storlet engine is tightly coupled with accounts in Swift in the following manners:

1. In order to invoke a storlet on a data object residing in some Swift account, that account must be enabled for storlets. That is, the designated, user defined, metadata flag on the account must be set to true.
2. Each Swift account must have certain containers required by the engine. One of these containers is the "storlet" container, in which storlets are being uploaded. After a given storlet has been uploaded from a given account to to the "storlet" container, it can be invoked on any data object pertaining to that account, given that the invoking user has read permissions to the "storlet" container.
3. Each account has a separate Docker image (and container) where storlets are being executed. All the storlets that are executed on data objects belonging to some account, will be executed in the same Docker container. This permits differentiating images as function of the Swift accounts. The Docker image name must be the account id to which it belongs.

8.3 The Docker image

As mentioned above there is a Docker image per account that is enabled for storlets. At a high level this image contains:

1. A Java run time environment. This is needed when you run storlets written in Java.
2. A daemon factory. A Python process that starts as part of the Docker container bring up. This process spawns the "per storlet daemons" upon a request from the "storlet docker gateway" that runs in the context of the storlet_middleware.

	FOR	*[TARGET]*	WHEN	*[TRIGGER CLAUSE]*	DO	*[ACTION CLAUSE]*
P1		**TENANT** T1		OBJECT_TYPE=DOCS		**SET** COMPRESSION **WITH** TYPE=LZ4, **SET** ENCRYPTION
P2		**CONTAINER** C1		GETS_SEC > 5 **AND** OBJECT_SIZE<10M		**SET** CACHING **ON PROXY TRANSIENT**
P3		**TENANT** T2				**SET** BANDWIDTH **WITH** MIN_BW=30MBps

Storage automation policies Resource management policies

Figure 7: Structure of the Crystal DSL.

3. A storlet daemon. The storlet daemon is a generic daemon that once spawned, loads a certain storlet code and awaits invocations. Different storlets, e.g. a filtering storlet and a compression storlet are loaded into different daemons. A daemon is invoked the first time a certain storlet needs to be executed.

4. The storlet common jar. This is the jar used for developing storlets in Java. Amongst other things it contains the code of the interface which must be implemented by java storlets.

8.4 The storlet bus

The storlet bus is a communication channel between the storlet middleware at the Swift side and the factory daemon and the storlet daemon in the Docker container. For each Docker container (or Swift account) there is a communication channel with the storlet factory of that container. For each storlet daemon in the container there is a communication channel on which it listens for invocations. These channels are based on unix domain sockets.

8.5 IOStack integration

As we will see later in the section 9.4.2, the filter framework for object storage integrates Storlets as one of the filter execution environments of IOStack. Storlets act as an isolated filter execution environment to run computations on object requests with higher security guarantees. Combined with Crystal, Storlets are enhanced with pipelining and stage execution control (i.e., proxy/storage node) functionalities.

Furthermore, IOStack web dashboard integrates the deployment of Storlets as well as native filters, allowing an administrator to transparently define a filter pipeline that combines both kinds of filters.

In the context of the Gridpocket use case (SQL pushdown mechanism), the Storlet WSGI middleware in Swift was extended to support running Storlets at storage nodes for byte ranges. Also, we contribute a new storlet that can perform projection and selection filters over CSV data. This work is explained in depth in Deliverable D4.2.

9 Crystal: SDS for Multi-tenant Object Stores

The objective of Crystal is to constitute the first SDS platform for object storage that efficiently handles workload heterogeneity and applications with evolving requirements. To achieve this, Crystal separates high-level policies from the mechanisms that implement them at the data plane, to avoid hard-coding the policies in the system itself. As mentioned in Section 3.1, it uses three main abstractions: *filter*, *metric (or trigger)*, and *controller*, in addition to *policies*.

9.1 Abstractions in Crystal

Filter. A filter is a piece of programming logic that a system administrator can inject into the data plane to perform custom computations. In Crystal, this concept is broad enough to include from arbitrary computations on object requests, such as compression or encryption, to resource management such as bandwidth differentiation. A key feature of filters is that the instrumented system is oblivious to their execution and needs no modification to its implementation code to support them.

Inspection trigger. It is an important abstraction in Crystal whose role is to automate the execution of filters based on the information accrued from the system. There are two types of information sources. A first type that corresponds to the real-time measurements got from the running workloads, like the number of `GET` operations per second of a tenant or the IO bandwidth allocated to a data container. As with filters, a fundamental feature

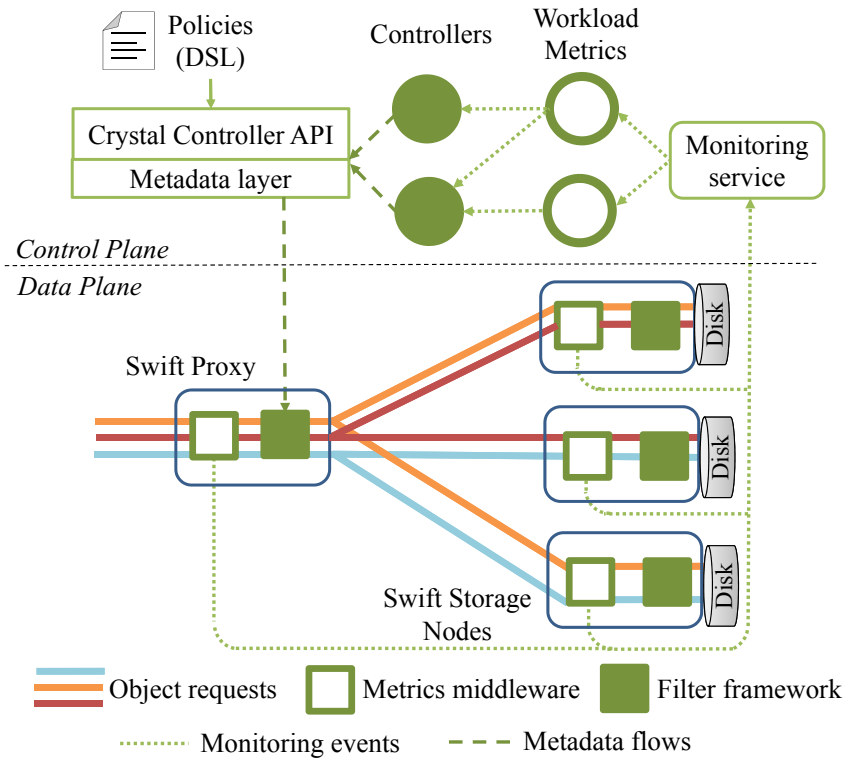


Figure 8: Overview of Crystal's architecture materialized on top of OpenStack Swift.

of workload metrics is that they can be deployed at runtime. A second type of source is the metadata from the objects themselves. Such metadata is typically associated with read and write requests and includes properties like the size or type of data objects.

Controller. In Crystal, a controller represents an algorithm that manages the behavior of the data plane based on monitoring metrics. A controller may contain a very simple rule to enforce compression filter on a tenant, or it may execute a complex bandwidth differentiation algorithm requiring global visibility of the cluster. Crystal builds a logically centralized control plane formed by supervised and distributed controllers. This allows an administrator to easily deploy new controllers on-the-fly that cope with the requirements of new applications.

Policy. Beyond making life easy for storage administrators, our SDS model targets great programmability for really opening the system to evolving requirements. This affects the structure of policies that should enable the incorporation of new control algorithms, workload metrics, custom filter logic, etc., in an easy manner.

To succinctly express policies, Crystal abides by a simple structure similar to those of the popular IFTTT (If This Then That) service [19]. This service allows users to express small rule-based programs, called "recipes", using *triggers* and *actions*. For example:

`TRIGGER`: compressibility of an object is $>$ 50%
`ACTION`: compress
`RECIPE: IF` compressibility is $>$ 50% `THEN` compress

An IFTTT-like language can reflect the extensibility capabilities of the SDS system; at the data plane, it is not hard to infer that triggers and actions are translated into resource metrics and filters, respectively. At the control plane, a policy is a "recipe" that guides the behavior of control algorithms. To illustrate this, Fig. 7 shows example policies: `P1`) enforce compression and encryption on document data objects of tenant `T1`; `P2`) apply data caching on small objects of container `C1` when the number of `GET`s per second is $>$ 5; or `P3`) try to provide at least 30MBps of aggregated bandwidth to tenant `T2`.

9.2 System Architecture

Next, we present the design of Crystal to add advanced SDS functionalities to OpenStack Swift. Fig. 8 shows that Crystal's architecture consists of:

Control Plane. In Crystal, administrators provision SDS services to tenants via high-level policies. To bring flexibility to the control plane, Crystal utilizes a DSL to express the policies in a concise manner. The IFTTT-like structure [19] of the policies makes it very easy to define them based on the abstractions of filters, controllers, and metrics, making the DSL system-agnostic. Further, it is extensible at runtime without side effects on the system operation. The control plane includes an API to digest policies, but also to manage the life-cycle and metadata of controllers, filters and metrics, as shown in Table 9.3a.

Moreover, the control plane is built upon a distributed model. Although logically centralized, the controller is, in practice, split into a set of autonomous micro-services, each running a separate control algorithm. The control loop is also extensible: developers can easily expose new metrics to the control plane and in this way, capture missing workload aspects. Such metrics are then made available to new controllers automatically by Crystal.

Data Plane. The data plane runs arbitrary computations and transformations on objects flows to satisfy storage policies. It provides two core extension points: Inspection triggers and filters. With the inspection triggers, Crystal provides distributed controllers with monitoring information on the state of the system at runtime. This allows the control plane to respond to changes in input workloads in real time. Rich data plane programmability is delivered through the filter framework, which intercepts object flows in a transparent manner and runs computations on them. A third party integrating a new filter only needs to contribute the logic; the deployment and execution of the filter is managed by Crystal.

9.3 Control Plane

The control plane offers advanced programmability over the data plane to manage multitenant workloads. It is formed by the DSL, the API and distributed controllers.

9.3.1 Crystal DSL

Crystal's DSL hides the complexity of low-level policy enforcement, thus achieving simplified storage administration (Fig. 7). The structure of our DSL is as follows:

Target: The target of a policy represents the recipient of a policy's action (e.g., filter enforcement) and it is mandatory to specify it on every policy definition. To meet the specific needs of object storage, targets can be *tenants*, *containers* or even individual *data objects*. This enables high management and administration flexibility.

Trigger clause (optional): Dynamic storage automation policies are characterized by the trigger clause. A policy may have one or more trigger clauses —separated by `AND`/`OR` operands— that specify the workload-based situation that will trigger the enforcement of a filter on the target. Trigger clauses consist of inspection triggers, operands (e.g, $>$, $<$, $=$) and values. The DSL exposes both types of inspection triggers: workload metrics (e.g., `GETS_SEC`) and request metadata (e.g., `OBJECT_SIZE<512`).

Action clause: The action clause of a policy defines how a filter should be executed on an object request once the policy takes place. The action clause may accept parameters after the `WITH` keyword in form of key/-value pairs that will be passed as input to customize the filter execution. Retaking the example of a compression filter, we may decide to enforce compression using a `gzip` or an `lz4` engine, and even their compression level.

To cope with object stores formed by proxy/storage nodes (e.g., Swift), our DSL enables to explicitly control the execution stage of a filter with the `ON` keyword. Also, dynamic storage automation policies can be *persistent or transient*; a persistent action means that once the policy is triggered the filter enforcement remains indefinitely (keyword `PERSISTENT`), whereas actions to be executed only during the period where the condition is satisfied are transient (keyword `TRANSIENT`).

Our DSL can be extended on-the-fly to accommodate new filters, controllers and inspection triggers. That is, in Fig. 7 we can use keywords `COMPRESSION` and `DOCS` in `P1` once we associate "`COMPRESSION`" with a given filter implementation and "`DOCS`" with some file extensions, respectively (see Table 9.3a). This makes Crystal's DSL extensible, in contrast to other systems [20].

Crystal Controller Calls	*Description*
`add_policy delete_policy list_policies`	Management API calls for creating, deleting and listing policies. The `add_policy` call relies on the DSL compiler either to directly enforce the policy or to instantiate a distributed controller.
`register_keyword delete_keyword`	Calls that interact with Crystal registry to associate DSL keywords with filters, inspection hooks or coin new terms to use in the DSL (e.g., `DOCS`).
`deploy_controller kill_controller`	These calls are used to manage the life-cycle of new dynamic policies and workload metric processes in the system.
Filter Framework Calls	*Description*
`deploy_filter undeploy_filter list_filters`	Calls for deploying, undeploying and listing filters associated to a target. `deploy`/`undeploy_filter` calls interact with the filter framework at the data plane for enabling/disabling filter binaries to be executed on a specific target.
Workload Metric Calls	*Description*
`deploy_metric delete_metric`	Calls for deploying and removing workload metrics from the data plane.
BW Differentiation Calls	*Description*
`update_bw list_bw_slo clear_bw_slo`	Calls to assign, list and delete a specified bandwidth SLO to the given tenant. Distributed bandwidth enforcement algorithms take as input this information in order to guarantee an aggregated IO bandwidth share at the data plane.

*For the sake of simplicity, we do not include call parameters in this table.

Table 9.3a: Main calls of Crystal controller, filter framework and bandwidth differentiation management APIs.

The Crystal DSL implements advanced administration features: i) *specialization* of policies based on the target scope, so that if several policies apply to the same request, only the most specific one is executed (e.g., container-level policy is more specific than a tenant-level one), ii) *pipelining* several actions on a single request (e.g., compression + encryption), similar to stream processing frameworks [21], and iii) *grouping*, that enables to enforce a single policy to a group of targets; that is, we can create a group like `WEB_CONTAINERS` to represent all the containers that serve Web pages.

As visible in Table 9.3a, Crystal offers a DSL compilation service via API calls. Crystal compiles static automation policies as *target→filter* relationships at the metadata layer. Next, we show how dynamic policies are materialized as controllers that extend the control plane.

9.3.2 Distributed Controllers

Crystal enables the creation of distributed controllers, in form of supervised processes, which can be deployed in the system at runtime to add new behaviors to the control plane [22, 23, 24][6].

We offer two types of controllers: *dsl-generated* and *custom* controllers. On the one hand, the Crystal DSL transparently compiles dynamic automation storage policies into dsl-generated controllers (e.g., `P2` in Fig. 7) that interact with our filter framework. On the other hand, custom controllers are not generated by the DSL; instead, by following a small set of conventions, Crystal enables developers to deploy controllers that contain complex algorithms rather than simple activation rules (e.g., `P3` in Fig. 7). For instance, this allowed us to deploy distributed IO bandwidth control algorithms (Section 9.5).

Dynamic control loop: Distributed controllers can be programmed to react and adapt to changes in the underlying storage system. Therefore, controllers must receive up-to-date inspection information from the data plane. As we explain in the next section, Crystal exposes monitoring metrics of the current state of workloads.

Technically, distributed controllers —dsl-generated and custom— and workload metrics interact in a publish/subscribe fashion. As visible in Fig. 9, once initialized, a distributed controller subscribes to the appropriate workload metric, taking into account the target granularity. The subscription request of a controller specifies the target to which it is interested in, such as tenant `T1` or container `C1`. Once the workload metric receives the subscription request, it adds the controller to its observer list. Periodically, the workload metric notifies the activity of the different targets to the interested controllers that may trigger the execution of filters.

9.4 Data Plane

To deal with heterogeneity, we offer two main extension hooks: Inspection triggers and a filter framework.

[6]Note that these controllers are also applicable to control dynamic functionalities in Zoe and Konnector.

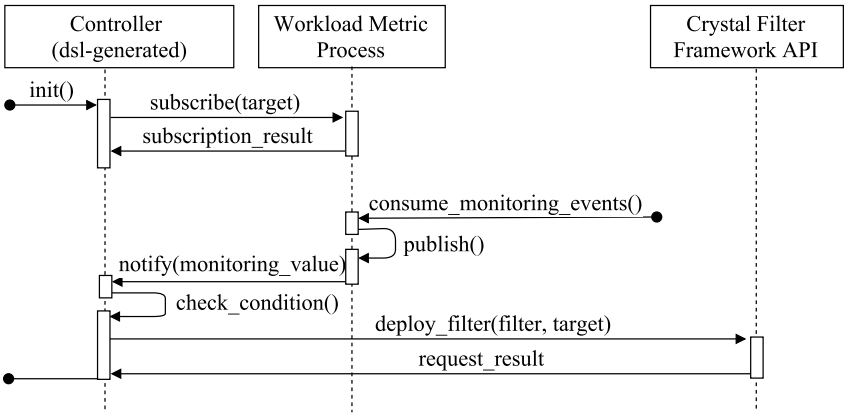


Figure 9: Interactions among dsl-generated controllers, workload metric processes and the filter framework.

9.4.1 Inspection Triggers

Inspection triggers enable controllers to dynamically respond to workload changes in real time. Specifically, we consider two types of introspective information sources: *object metadata* and *monitoring metrics*.

First, some object requests embed semantic information related to the object at hand in form of metadata. Crystal enables administrators to enforce storage filters based on such metadata. Concretely, our filter framework middleware (see Section 9.4.2) is capable of analyzing at runtime HTTP metadata of object requests to execute filters based on the object size or file type, among others.

Second, Crystal builds a metrics middleware to add new workload metrics on-the-fly. In our design, a new workload metric can inject events to the monitoring service without interfering with existing ones (Table 9.3a). A salient feature of our metrics framework is that it enables developers to plug-in metrics to inspect both the type of requests and their contents (e.g., compressibility).

At the top level of our monitoring system we find *workload metric processes*. These processes are devised to consume and aggregate monitoring information to be published to controllers (see Fig. 9). Each workload metric process consumes from a different monitoring metric at the data plane. For the sake of simplicity and isolation [24], we advocate to separate workload metrics not only per metric type, but also by target granularity.

9.4.2 Filter Framework

The Crystal filter framework enables developers to deploy and run general-purpose code on object requests. Crystal borrows ideas from active storage literature [25, 26] as a mean of building filters to enforce policies.

Our framework achieves flexible execution of filters. First, it enables to easily *pipeline several filters* on a single storage request. Currently, the execution order of filters is set explicitly by the administrator, although filter metadata can be exploited to avoid conflicting filter ordering errors [27]. Second, to deal with object stores composed by proxy/storage nodes, Crystal allows administrators to define the *execution point of a filter*.

To this end, the Crystal filter framework consists of i) filter middleware, and ii) filter execution environments.

Filter middleware: Our filter middleware is a hook to intercept data streams and classify incoming requests. Upon a new object request, the Crystal middleware may contact the metadata layer to infer the filters to be executed on that request depending on the target. If the target has associated filters to be enforced, the Crystal middleware sets the appropriate HTTP headers in the request (e.g., `GET`, `PUT`) for triggering the filter execution.

Filters that change the content of data objects may receive a special treatment, such as in case of compression or encryption filters. To wit, if we create a filter with the *reverse* flag enabled, it means that the execution of the filter when the object was stored should be always undone upon a `GET` request. For instance, this yields that we may activate data compression on certain periods, but tenants will always download decompressed data objects. To this end, we store data objects with an *extended metadata* to keep track of the enforced reverse filters. Upon a `GET` request, such metadata is fetched by the Crystal middleware to trigger reverse transformations on

the data object prior to the execution of regular filters.

Filter execution environments: Thanks to the interception capabilities of our middleware, it can support multiple execution platforms. Crystal features:

Isolated filter execution: Crystal provides an isolated filter execution environment to compute on object requests with higher security guarantees. To this end, we extended the Storlets framework [28] with pipelining and stage execution control functionalities. Storlets provide Swift with the capability to run computations near the data in a secure and isolated manner making use of `Docker` containers [29]. Invoking a Storlet on a data object is done in an isolated manner so that the data accessible by the computation is only the object's data and its user metadata. Moreover, a `Docker` container only executes filters of a single tenant.

Native filter execution: The isolated filter execution environment trades-off higher security for lower communication capabilities and interception flexibility. For this reason, we also contribute an alternative way to intercept and execute code natively. As with Storlets, a developer can install on runtime in Crystal code modules as filters following simple design guidelines. However, native filters can i) execute code at all the possible points of a request's life-cycle, and ii) communicate with external components (e.g, metadata layer), as well as to access storage devices (e.g., SSD). As Crystal is devised to execute trusted code from administrators, this environment represents a more flexible alternative.

9.5 Hands On: Extending Crystal

Next, we show the benefits of Crystal's design by extending the system with data management filters and distributed control of IO bandwidth for OpenStack Swift.

9.5.1 New Storage Management Policies

Goal: To define policies that enforce filters, like *compression*, *encryption* or *caching*, even dynamically:

```
P1:  FOR TENANT T1 WHEN OBJECT_TYPE=DOCS DO SET COMPRESSION ON PROXY, SET ENCRYPTION
P2:  FOR CONTAINER C1 WHEN GETS_SEC > 5 SET CACHING
```

Data plane (Filters): To enable such policies, we first need to *develop the filters* at the data plane. In Crystal this can be done either using the native or isolated execution environments.

The next code snippet shows how to develop a filter for our isolated execution environment. A system developer only needs to create a class that implements an interface (`IStorlet`), providing the actual data transformations on the object request streams (`iStream`, `oStream`) inside the `invoke` method. To wit, we implemented the compression (`gzip` engine) and encryption (AES-256) filters using storlets, whereas the caching filter exploits SSD drives at proxies via our native execution environment. Then, once these filters were developed, we installed them via the Crystal filter framework API.

```
public class StorletName implements IStorlet {

 @Override
 public void invoke(ArrayList<StorletInputStream> iStream,
   ArrayList<StorletOutputStream> oStream,
   Map<String, String> parameters, StorletLogger logger)
   throws StorletException {

    //Develop filter logic here
 }
}
```

Data plane (Monitoring): Achieving dynamic policy enforcement requires from monitoring information. Thus, we instructed our metrics middleware to inject monitoring information of object requests (e.g., `PUT`s/`GET`s per second of a tenant) in the monitoring service. Via the Crystal API (see Table 9.3a), we also deployed several workload metrics processes (one per metric and target granularity) that aggregate such monitoring information to be published to controllers. Also, our filter framework middleware is already capable of triggering filters based on object metadata, such as object size (`OBJECT_SIZE`) and type (`OBJECT_TYPE`).

Control Plane: Finally, we registered intuitive keywords for both filters and workload metrics at the metadata layer (e.g., `CACHING`, `GET_SEC_TENANT`) using the Crystal registry API. To achieve `P1`, we also regis-

Algorithm 1 `computeAssignments` pseudo-code embedded into a bandwidth differentiation controller

```
1:  function COMPUTEASSIGNMENTS(info):
2:                                          ▷ Retrieve the defined tenant SLOs from the metadata layer
3:      SLOs ← getMetadataStoreSLOs();
4:      disksUsage ← {};
5:                                          ▷ Compute disk assignments on current tenant transfers to meet SLOs
6:      SLOTenantAssignments ← minSLO(disksUsage, SLOs);
7:                                          ▷ Estimate spare bw at proxy/storage nodes based on current usage
8:      spareBW ← min(maxBWproxys(info), maxBWdisks(disksUsage));
9:      spareBWEnforcements ← {};
10:                                         ▷ Distribute spare bandwidth fairly across all tenants
11:     for tenant in info do
12:         spareBWEnforcements[tenant] ← spareBW / numTenants(info);
13:     end for
14:                                         ▷ Calculate disk assignments to achieve spare bw shares for tenants
15:     spareBWAssignments ← minSLO(disksUsage, spareBWEnforcements);
16:                                         ▷ Sum up SLO and spare bw tenant disk assignments
17:     return SLOTenantAssignments ∪ spareBWAssignments;
18: end function
```

tered the keyword `DOCS`, which contains the file extensions of common documents (e.g, `.pdf`, `.doc`). At this point, we can use such keywords in our DSL to design new storage policies.

9.5.2 Distributed IO Bandwidth Control

Goal: To provide Crystal with means of defining policies that enforce a global IO bandwidth SLO:

```
P3:  FOR TENANT T1 SET BANDWIDTH WITH PARAM1=30MBps
```

Data plane (Filter). To achieve global bandwidth SLOs on targets, we first need to locally control the bandwidth of object requests. Intuitively, bandwidth control in Swift may be performed at the proxy or storage node stages. At the proxy level this task may be simpler, as fewer nodes should be coordinated. However, this approach is agnostic of the background tasks (e.g., replication) executed by storage nodes, which impact on performance [20]. We implemented a native bandwidth control filter that enables the enforcement at both stages.

Our filter dynamically creates threads that serve and control the bandwidth allocation for individual tenants, either at proxies or storage nodes. Our filter garbage-collects control threads that are inactive for a certain timeout. Moreover, it has a consumer process that receives bandwidth assignments from a controller to be enforced on a tenant's object streams. Once the consumer receives a new event, it propagates the assignments to the filter that immediately take effect on current transfers.

Data plane (Monitoring): For building the control loop, our bandwidth control service integrates individual monitoring metrics per type of traffic (i.e., `GET`, `PUT`, `REPLICATION`); this makes it possible to define policies to each type of traffic, if needed. In essence, monitoring events contain a data structure that represents the bandwidth share that tenants exhibited at proxies or per storage node disk. We also deployed workload metric processes to expose these events to controllers.

Control plane. We equipped Crystal with a *base bandwidth controller* that encapsulates the logic to ingest bandwidth monitoring events and to disseminate the computed assignments across nodes for the different request types. The base controller also gets the SLOs to be enforced from the metadata layer (see Table 9.3a). Hence, to develop a bandwidth enforcement algorithm, practitioners only need to write a new controller that extends the base one and overrides the function `compute_assignments` (see Algorithm 1).

To show the feasibility of our bandwidth differentiation filter, we design an enforcement controller in Algorithm 1. Concretely, we aim at satisfying three main requirements: i) achieve a *minimum bandwidth per tenant*, ii) *work-conservation* (do not leave idle resources), and iii) provide *global fairness of spare bandwidth* across tenants. The challenge is to meet these requirements considering that we do not control neither the data access of tenants nor the data layout of Swift [30, 31].

To this end, Algorithm 1 works in three stages. First, the algorithm tries to ensure the SLO for tenants specified in the metadata layer by resorting to function `minSLO` (requirement 1, line 6). Essentially, `minSLO` first assigns a proportional bandwidth share to transfer of tenants with guaranteed bandwidth. Note that such assignment is done in descending order based on the number of parallel transfers, provided that tenants with

fewer transfers have less opportunities of meeting their SLOs. Moreover, `minSLO` checks whether there exist overloaded storage nodes in the system. In the affirmative case, the algorithm tries to reallocate bandwidth of tenants with multiple transfers from overloaded nodes to idle ones. In case that no reallocation is possible, the algorithm reduces the bandwidth share of tenants with SLOs on overloaded nodes.

In second place, once Algorithm 1 has calculated the assignments for tenants with SLOs, it estimates the spare bandwidth available to achieve full utilization of the cluster (requirement 2, line 8). Note that the notion of spare bandwidth depends on the cluster at hand, as the bottleneck may be either at the proxies or storage nodes.

Algorithm 1 builds a new assignment data structure in which the spare bandwidth is equally assigned to all tenants. The algorithm proceeds by calling again function `minSLO` to calculate the spare bandwidth assignments (requirement 3, line 15). Note that the second call to `minSLO` receives the `disksUsage` data structure that keeps the already reserved node bandwidth according to the SLO tenant assignments. The algorithm outputs the combination of SLO and spare bandwidth assignments per tenant. While more complex algorithms can be integrated in Crystal, our goal in Algorithm 1 is to offer an attractive simplicity/effectiveness trade-off, validating our bandwidth differentiation framework.

9.6 Crystal Prototype

Web page	http://crystal-sds.org/
Source Code	https://github.com/Crystal-SDS
Documentation	https://github.com/Crystal-SDS/controller
Continuous Integration	https://travis-ci.org/Crystal-SDS/controller
Test Coverage	https://coveralls.io/github/Crystal-SDS/controller
Code Quality	https://landscape.io/github/Crystal-SDS/controller

We tested the Crystal prototype in OpenStack Kilo version. The APIs at the control plane are implemented with Django framework to ease the integration with other architecture components. As metadata layer, in this work Crystal uses Redis 3.0. We resort to PyActive [32] for building supervised controllers and workload metric processes that can communicate either via TCP or a Message Oriented Middleware (MOM). We use RabbitMQ 3.6 as a communication service for monitoring information. Crystal also provides a dashboard that extends the OpenStack Horizon to ease the management of the SDS framework by administrators. The code of Crystal for object storage is publicly available[7] and our contributions to the Storlets framework are in process of acceptance by the official OpenStack repository.

9.7 Related Systems

SDS Systems. IOFlow [33], now extended as sRoute [34], was the first complete SDS architecture. IOFlow enables end-to-end (e2e) policies to specify the treatment of IO flows from VMs to shared storage. This was achieved by introducing a queuing abstraction at the data plane and translating high-level policies into queuing rules. The original focus of IOFlow was to enforce e2e bandwidth targets, which was later augmented with caching and tail latency control in [34, 35].

Despite Crystal shares with IOFlow design concepts (e.g., policies, control/data planes), our target is different; Crystal pursues to configure and optimize object stores to the evolving needs of applications, for it needs a richer data plane and a different suite of management abstractions and enforcement mechanisms. For instance, tenants require mechanisms to inject custom logic and abstractions to specify not only system activities but also application-specific transformations on objects.

Retro [20] is a framework for implementing resource management policies in multi-tenant distributed systems. It can be viewed as an incarnation of SDS, because as IOFlow and Crystal, it separates the controller from the mechanisms that implement it. A major contribution of Retro is the development of abstractions to enable policies that are system- and resource-agnostic. Crystal shares the same spirit of requiring low develop effort. However, its abstractions are different. Crystal must abstract not only resource management; it must enable the concise definition of policies that enable high levels of programmability to suit application needs. Retro is only extensible to handle custom resources.

[7] `https://github.com/Crystal-SDS`

Vertigo [36] is a framework where the control logic is directly embedded into data objects in the form of *micro-controllers*. This enables object-level policies avoiding a centralized control point. We believe that Crystal may also support policies that orchestrate micro-controllers.

IO bandwidth differentiation. Enforcing bandwidth SLOs in shared storage has been a subject of intensive research over the past 10 years, specially in block storage [37, 38, 39, 40, 41, 33, 20]. However, object stores have received much less attention in this regard; vanilla Swift only provides a non-automated mechanisms for limiting the "number of requests" [42] per tenant, instead of IO bandwidth. In fact, this problem resembles the one stated by Wang et al. [31] where multiple clients access a distributed storage system with different data layout and access patterns, yet the performance guarantees required are global. To our knowledge, Wu et al. [40] is the only work addressing this issue in object storage. It provides SLOs in Ceph by orchestrating local rate limiters offered by a modified version of the underlying file system (EBOFS). However, this approach is intrusive and restricted to work with EBOFS. In contrast, Crystal transparently intercepts and limits requests streams, enabling developers to design new algorithms that provide distributed bandwidth enforcement [43, 30].

Active storage. The early concept of *active disk* [25], i.e., a HDD with computational capacity, was borrowed by distributed file system designers in HPC environments in the last decade to give birth to active storage. The goal was to diminish the amount of data movement between storage and compute nodes [44, 45]. Piernas et al. [26] presented an active storage implementation integrated in the Lustre file system that provides flexible execution of code near to data in the user space. Crystal goes beyond active storage. It exposes through the filter abstraction a mechanism to inject custom logic into the data plane and expose it to management policies. This requires filters to be deployable at runtime, support sandbox execution [28], and be part of complex workflows.

10 Evaluation of Crystal

Next, we evaluate a prototype of Crystal for OpenStack Swift in terms of flexibility, performance and overhead.

Objectives: Our evaluation aims to show: i) Crystal can define policies at multiple granularities, achieving administration flexibility; ii) The enforcement of storage automation filters can be dynamically triggered based on workload conditions; iii) Crystal achieves accurate distributed enforcement of IO bandwidth SLOs on different tenants; iv) Finally, Crystal has low execution/monitoring overhead.

Workloads: We resort to well-known benchmarks and replays of real workload traces. First, we use `ssbench` [46] to execute stress-like workloads on Swift. `ssbench` provides flexibility regarding the type (CRUD) and number of operations to be executed, as well as the size of files generated. All these parameters can be specified in form of configuration "scenarios".

Moreover, to evaluate Crystal under real-world object storage workloads, we collected and replayed the following traces[8]: i) The first trace (65GB) captures a read-dominated Web workload consisting of requests related to 228K data objects from several Web pages hosted at Arctur datacenter for 1 month. ii) The second trace (779GB) was collected from a document database (write-dominated) storing 817K car testing/standardization documents (e.g., pictures, PDFs, docs) for 9 months at Idiada; a large company in the automotive sector. We resort to SDGen [47] to generate realistic contents for data objects based on the file types described in workload traces.

Platform: We ran our experiments in our own 13-machine cluster located at URV facilities. The cluster is formed by 9 Dell PowerEdge 320 nodes (Intel Xeon E5-2403 processors); 2 of them act as Swift proxy nodes (28GB RAM, 1TB HDD, 500GB SSD) and the rest are Swift storage nodes (16GB RAM, 2x1TB HDD). There are 3 Dell PowerEdge 420 (32GB RAM, 1TB HDD) nodes that are used as compute nodes to execute workloads. Also, there is 1 large node that runs the OpenStack services and the Crystal control plane (i.e., APIs, controllers, MOM, Redis). Nodes in the cluster are connected via GbE switched links.

10.1 Evaluating Storage Automation

Next, we present a battery of experiments that demonstrate the feasibility and capabilities of storage automation with Crystal. To this end, we make use of synthetic workloads and real trace replays (Ididada, Arctur). These experiments have been executed at the compute nodes against 1 swift proxy and 6 storage nodes.

Storage management capabilities of Crystal. Fig. 10 shows the execution of several storage automation policies on a workload related to containers `C1` and `C2` belonging to tenant `T1`. Specifically, we executed a

[8] Traces are available at `http://iostack.eu/datasets-menu`

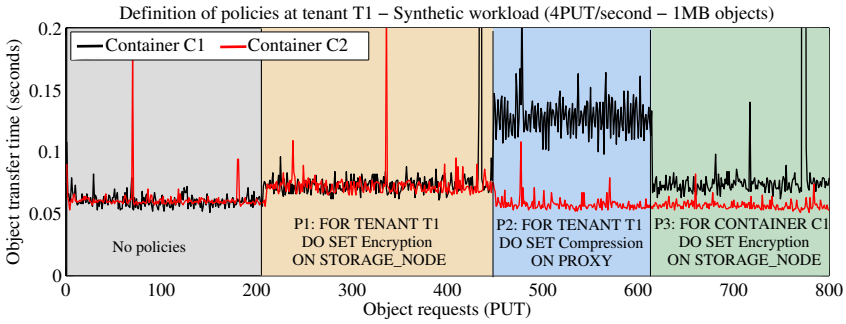


Figure 10: Enforcement of compression/encryption filters.

write-only synthetic workload (4PUT/second of 1MB objects) in which data objects stored at `C1` consist of random data, whereas `C2` stores highly redundant objects.

Due to the security requirements of `T1`, the first policy defined by the administrator is to encrypt his data objects (`P1`). Fig. 10 shows that the `PUT` operations of *both containers* exhibit a slight extra overhead due to encryption, given that the policy has been defined at the tenant scope. There are two important aspects to note from `P1`: First, the execution of encryption on `T1`'s requests is isolated from filter executions of other tenants, providing higher security guarantees [28] (Storlet filter). Second, the administrator had the ability to enforce the filter at the storage node in order to do not overload the proxy with the overhead of encrypting data objects (`ON` keyword).

After policy `P1` was enforced, the administrator decided to optimize the storage space of `T1`'s objects by enforcing compression (`P2`). `P2` also enforces compression at the proxy node to minimize communication between the proxy and storage node (`ON PROXY`). Note that the enforcement of `P1` and `P2` demonstrates the filter pipelining capabilities of our filter framework; once `P2` is defined, Crystal enforces compression at the proxy node and encryption at storage nodes for each object request. Also, as shown in Section 9.4, the filter framework tags objects with extended metadata to trigger the reverse execution of these filters on `GET` requests (i.e., decryption and decompression, in that order).

However, the administrator realized that the compression filter on `C1`'s requests exhibited higher latency and provided no storage space savings (incompressible data). Thus, the administrator defined policy `P3` that essentially enforces only encryption on `C1`'s requests. After the defining `P3`, the performance of `C1`'s requests exhibits the same behavior as before the enforcement of `P2`. Thus, the administrator is able to manage storage at different granularities, such as tenant or container. Furthermore, the last policy also proves the usefulness of our policy specialization mechanism; policy `P2` at the tenant scope applies to `C1`, whereas the system only executes `P3` on `C1`'s requests, as it is the most specialized policy.

Dynamic storage automation. Fig. 11 shows the enforcement of dynamic caching policy (`P1`). The filter exploits SSD drives at the proxy to provide fast object retrievals under high activity. We executed a synthetic oscillatory workload to verify the correctness of the dynamic enforcement of filters via controllers.

In Fig. 11, we show the average latency of `PUT`/`GET` requests and the intensity of the workload. As can be observed, the caching filter takes place when the workload exceeds 5 `GET`s per second. At this point, the filter starts caching objects at the proxy SSD on `PUT`s, as well as to lookup the SSD to retrieve potentially cached objects on `GET`s. First, the filter provides performance benefits for object retrievals; when the caching filter is activated, object retrievals are in median 29.7% faster compared to no-caching periods. Second, we noted that the costs of executing asynchronous writes on the SSD upon `PUT` requests may be amortized by offloading storage nodes; that is, the average `PUT` latency is in median 2% lower when caching is activated. This may be due to the fact that storage nodes are mostly free to execute writes, as a large fraction of `GET`s are being served at the proxy's cache.

In conclusion, Crystal's control loop enables dynamic enforcement of storage filters under variable workloads. Moreover, the native filters in Crystal enable complex workload optimizations.

Managing real workloads. Next, we show how Crystal policies can handle real workloads (12 hours).

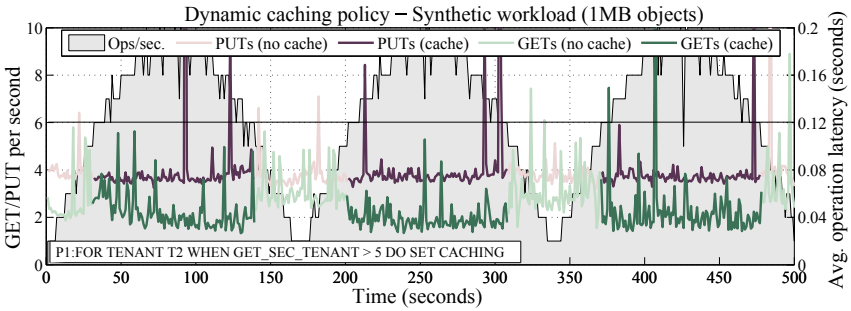


Figure 11: Dynamic enforcement of caching filter.

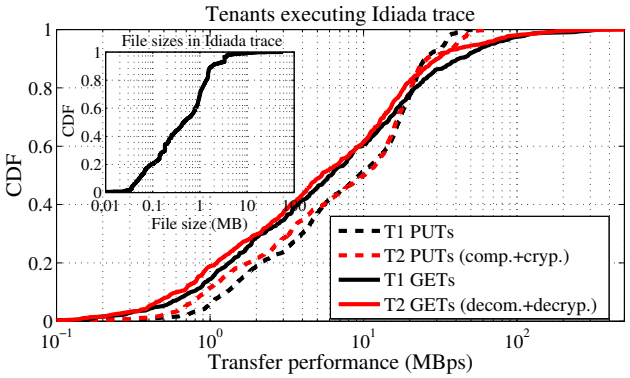


Figure 12: Policy enforcement on Idiada workload replay.

That is, we compress and encrypt documents (`P1` in Fig. 7) on a replay of the Idiada trace (write-dominated), whereas we enforce caching of small files (`P2` in Fig. 7) on a replay of Arctur workload (read-dominated).

Fig. 12 shows the request bandwidth exhibited during the execution of the Idiada trace. Concretely, we executed two concurrent workloads, each associated to a different tenant. We enforced compression and encryption only on tenant `T2`. Observably, tenant `T2`'s transfers are over 13% and 7% slower compared to `T1` for `GET`s and `PUT`s, respectively. This is due to the computation overhead of enforcing filters on `T2`'s document objects. As a result, `T2`'s documents consumed 65% less space compared to `T1` with compression and they benefit from higher data confidentially thanks to encryption.

Fig. 13 shows tenants `T2` and `T1`, both concurrently running a trace replay of Arctur. By executing a dynamic caching policy, `T2`'s `GET` requests are in median 99% faster compared to `T1`. That is, as the workload of Arctur is intense and almost read-only, caching was enabled for tenant `T2` for most of the experiment. This, in addition to the small size of files that fit in cache, makes caching at the proxy SSD drives a practical optimization filter. The median write overhead of `T2` compared to `T1` is 4.2%, which indicates that our filter efficiently intercepts the data stream for doing writes at the SSD.

The enforcement of these policies demonstrates the benefits of managing an object store with Crystal.

10.2 Achieving Bandwidth Differentiation

Next, we evaluate the effectiveness of our bandwidth differentiation filter. To this end, we executed a `ssbench` workload (10 concurrent threads) in each of the 3 compute nodes in our cluster, one of each representing an individual tenant. As we study the effects of replication separately (Fig. 17), the rest of experiments were performed using one replica rings.

Request types. Fig. 14 plots two different SLO enforcement experiments on three different tenants for `PUT` and `GET` requests, respectively (enforcement at proxy node). Appreciably, the execution of Algorithm 1 exhibits a near exact behavior for both `PUT` and `GET` requests. Moreover, we observe that tenants obtain their SLO plus an equal share of spare bandwidth, according to the expected policy behavior defined by colored

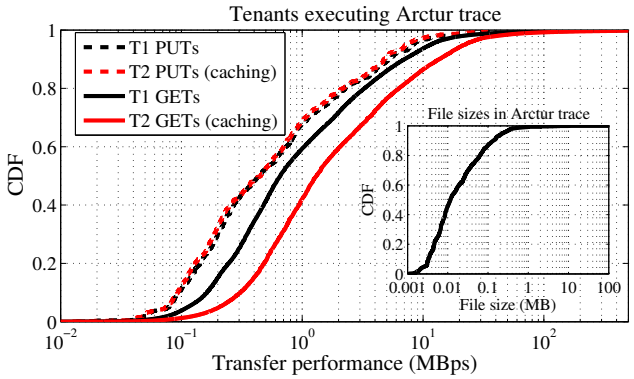


Figure 13: Policy enforcement on Arctur trace replay.

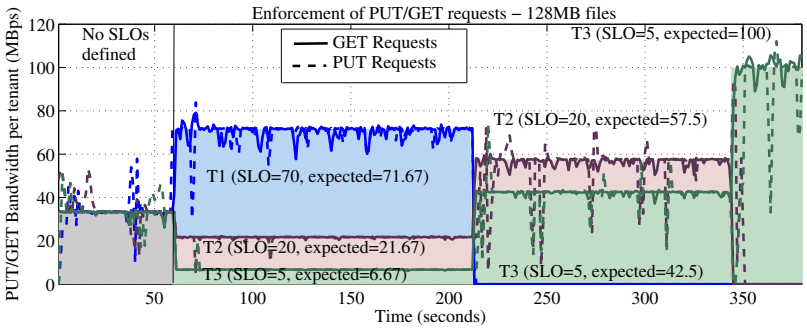


Figure 14: 1 proxy/3 storage nodes, bandwidth control at proxy.

areas. This demonstrates the effectiveness of our bandwidth control middleware for intercepting and limiting both requests types. Interestingly, we also observe in Fig. 14 that `PUT` bandwidth exhibits higher variability than `GET` bandwidth. A reason for this behavior may be that Swift buffers write requests. Concretely, after writing 512MB of data, we observed that the transfers of tenants stopped for a short interval in which the system might check if the data has been correctly stored. The side effect of this mechanism is that the stream of incoming requests is interrupted, inducing higher variability.

Impact of enforcement stage. An interesting aspect to study in our framework are the implications of enforcing bandwidth control at either the proxies or storage nodes. In this sense, Fig. 15 shows the enforcement SLOs on `GET` requests at both stages. At first glance, we observe in Fig. 15 that our framework makes it possible to enforce bandwidth limits at both stages. However, Fig. 15 also illustrates that the enforcement on storage nodes presents higher variability compared to proxy enforcement. This behavior arises from the relationship between the number of nodes to coordinate and the intensity of workload at hand. That is, given the same workload intensity, fewer nodes (e.g., proxies) offers higher bandwidth stability, as a tenant's requests are virtually a continuous data stream, being easier to control. Conversely, each storage node receives a smaller fraction of a tenant's requests, as normally storage nodes are more numerous than proxies. This yields that storage nodes should deal with shorter and discontinuous streams that are harder to control.

But enforcing bandwidth SLOs at storage nodes enables to control background tasks, like replication. Thus, we face trade-off between accuracy and control that may be solved with hybrid enforcement schemes.

Mixed tenant activity, variate file sizes. Next, we execute a mixed read/write workload using files of different sizes; small (8MB to 16MB), medium (32MB to 64MB) and large (128MB to 256MB) files. Besides, to explore the scalability, in this set of experiments we resort to a cluster configuration that doubles the size of the previous one (2 proxies and 6 storage nodes).

Appreciably, Fig. 16 shows that our enforcement controller achieves bandwidth SLOs under mixed workloads. Moreover, we observe that the bandwidth differentiation framework scales to larger cluster sizes, as the

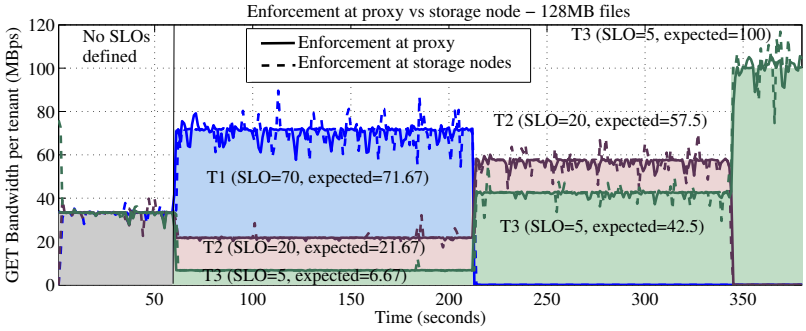


Figure 15: 1 proxy/3 storage nodes.

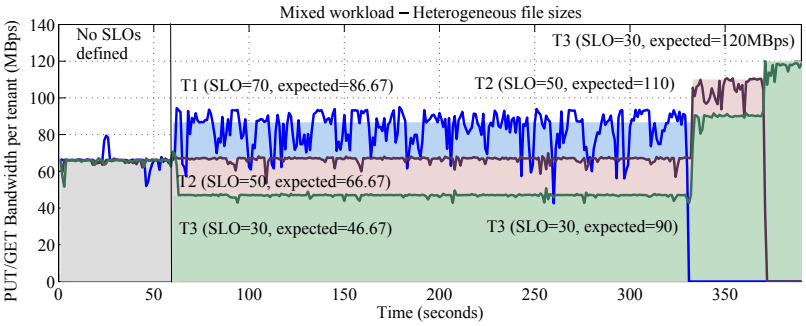


Figure 16: 2 proxy/6 storage nodes, bandwidth control at proxies.

policy provides tenants with the desired SLO plus a fair share of spare bandwidth, specially for `T1` and `T2`. However, Fig. 16 also illustrates that the `PUT` bandwidth provided to `T1` is significantly more variable than for other tenants; this is due to various reasons. First, we already mentioned the increased variability of `PUT` requests, apparently due to write buffering. Second, the bandwidth filter seems to be less precise when limiting streams that require an SLO close to the node/link capacity. Moreover, small files make the workload harder to handle by the controller as more node assignments updates are potentially needed, specially as the cluster grows. In the future, we plan to continue the exploration and mitigation of these sources of variability.

Controlling background tasks. An advantage of enforcing bandwidth SLOs at storage nodes is that we can also control the bandwidth of background processes via policies. To wit, Fig. 17 illustrates the impact of replication tasks on multi-tenant workloads. In Fig. 17, we observe that during the first 60 seconds of experiment (i.e., no SLOs defined) tenants are far from having a sustained `GET` bandwidth of $\approx$ 33MBps, meaning that they are importantly affected by the replication process. The reason is that, internally, storage nodes trigger hundreds of point-to-point transfers to write copies of already stored objects to other nodes belonging to the ring. Note that the aggregated replication bandwidth within the cluster reached 221MBps. Furthermore, even though we enforce SLOs from second 60 onwards, the objectives are not achieved —specially for tenants `T2` and `T3`— until replication bandwidth is under control. As soon as we deploy a controller that enforces a hard limit of 5MBps to the aggregated replication bandwidth, the SLOs of tenants are rapidly achieved. We conclude that Crystal has potential as a framework to define fine-grained policies for managing bandwidth allocation in object stores.

10.3 Crystal Overhead

Filter framework latency overheads. A relevant question to answer is the performance costs that our filter framework introduces to the regular operation of the system. Essentially, the filter framework may introduce overhead at i) *contacting the metadata layer*, ii) *intercepting the data stream through a filter*[9] and iii) *managing extended object metadata*. We show this in Fig. 18.

[9]We focus on isolated filter execution, as native execution has no additional interception overhead.

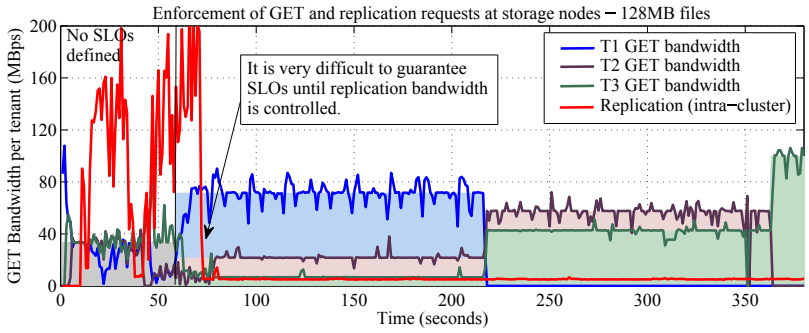


Figure 17: 1 proxy/3 storage nodes, bandwidth control at storage nodes.

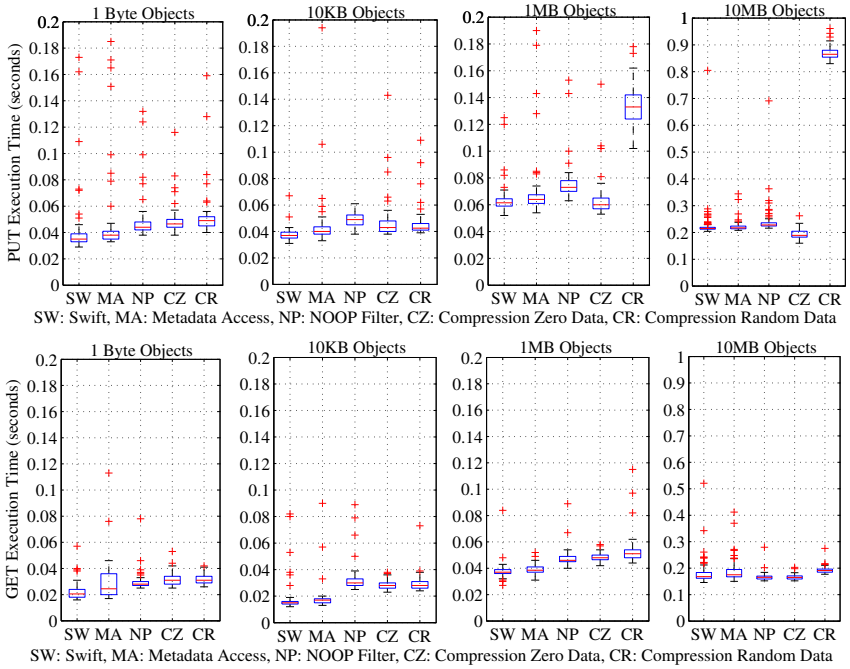


Figure 18: Performance overhead of filter framework metadata interactions and isolated filter enforcement.

Fig. 18 shows that the median latency of accessing the metadata store normally falls between 1.5ms and 3ms (`MA` boxplots) compared to vanilla Swift (`SW`). For 1MB objects, this represents a relative median latency overhead of 3.9% for both `GET`s and `PUT`s. Naturally, this overhead becomes slightly higher as the object size decreases, but is still practical (8% to 13% for 10KB objects). This confirms that our filter framework minimizes communication with the metadata layer (i.e., 1 query per request). Moreover, Fig. 18 shows that an in-memory store like Redis fits the metadata workload of Crystal, specially if it is co-located with proxy nodes.

Next, we focus on the isolated interception of object requests via Storlets, which trades-off performance for higher security guarantees (see Section 9.4). Fig. 18 illustrates that the median isolated interception overhead of a void filter (`NOOP`) oscillates between 3ms and 11ms. This cost comes from injecting the data stream into a `Docker` to achieve isolation. In addition, we have to consider the performance of the filter itself, which greatly depends on its implementation, or even on the data at hand. For instance, columns `CZ` and `CR` depict the performance of the compression filter for *highly redundant (zeros) and random* data objects. Visibly, the performance of `PUT` requests changes significantly (e.g., objects $\geq$ 1MB) as compression algorithms exhibit different performance depending on the data contents [47]. Conversely, `GET` requests are not significantly affected by decompressing different data contents. Hence, to improve performance, filters should be not only well implemented, but also enforced in the right conditions.

Finally, our filter framework enables managing extended metadata of objects to store a sequence of data transformations to be undone on retrievals (see Section 9.4). We measured that reading/writing extended object

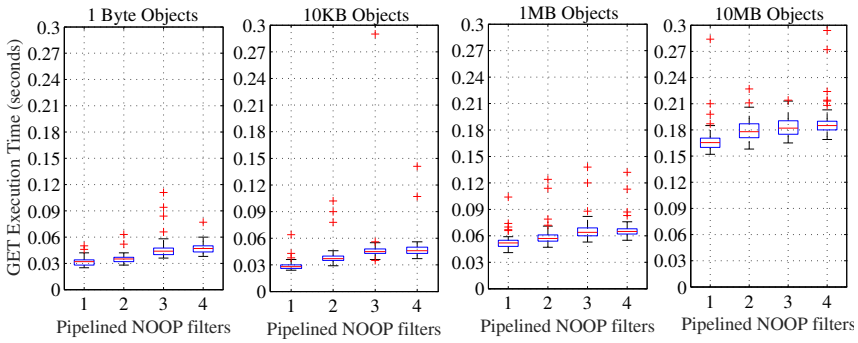


Figure 19: Pipelining performance for isolated filters.

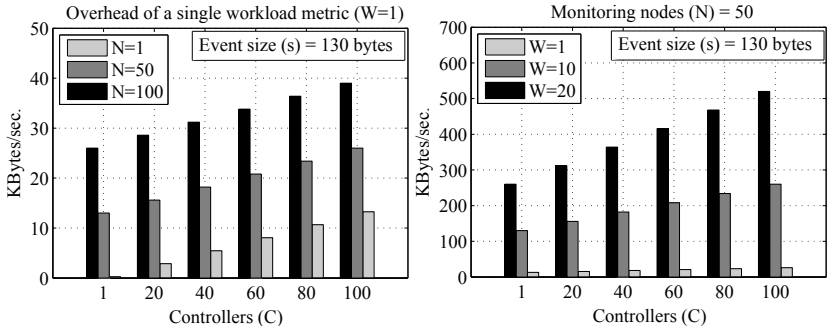


Figure 20: Traffic overhead of Crystal depending on the number of nodes, controllers and workload metrics.

metadata takes 0.3ms/2ms, respectively, which constitutes modest overhead.

Filter pipelining throughput. Next, we want to further explore the overhead of isolated filter execution. Specifically, Fig. 19 depicts the latency overhead of several `NOOP` Storlet filters to evaluate the costs of pipelining; a new feature we developed in this work.

Fig. 19 shows that the latency costs of intercepting a data stream through a pipeline of isolated filters is acceptable. To inform this argument, each additional filter in the pipeline incurs 3ms to 9ms of extra latency in median. This is slightly a lower latency than passing the stream through the `Docker` container for the first time. The reason is that pipelining tenant filters is done within the same `Docker` container, so the costs of injecting the stream into the container are present only once. Therefore, our filter framework is a feasible platform to dynamically compose and pipeline several isolated filters.

Monitoring overheads. We want to give a sense on the monitoring overhead of Crystal. To this end, we provide a measurement based estimation of various configurations of monitoring nodes, workload metrics and controllers in Crystal. To wit, the monitoring traffic overhead O related to $\mathcal{W}$ workload metrics is produced by a set of $\mathcal{N}$ nodes, either proxies or storage nodes. The set of $\mathcal{N}$ nodes periodically sends monitoring events of size s to the MOM broker, which are consumed by the $\mathcal{W}$ workload metrics. Then, a workload metric aggregates the messages of all producer nodes into a single monitoring message. The aggregated message is then published to a set of subscribed controllers $\mathcal{C}$. Therefore, we can do a worst case estimation of the total generated traffic per monitoring epoch (e.g., 1 second) as: $O = |\mathcal{W}| \cdot [s \cdot (2 \cdot |\mathcal{N}| + |\mathcal{C}|)]$. We also measured simple events (e.g., `PUT_SEC`) to be $s = 130$ bytes in size.

Given that, Fig. 20 shows that the estimated monitoring overhead of a single metric is modest; in the worst case, a single workload metric generates less than 40KBps in a 100-machine cluster with $|\mathcal{C}| = 100$ subscribed controllers. Clearly, the dominant factor of traffic generation is the number of workload metrics working in the system. However, even for a large number of workload metrics ($|\mathcal{W}| = 20$), the monitoring requirements in a 50-machine cluster do not exceed 520KBps. These overheads seem lower than existing SDS systems with advanced monitoring [20].

11 Future Development of the IOStack Toolkit

In this section, we outline some of the main development objectives that will be targeted for the last year of the project. These objectives mainly focus on better exploiting dynamic service provisioning and explore the convergence/cooperation of the different building blocks.

Dynamic block filters: In deliverable D3.2, we presented a complete evaluation of the filter framework for block storage provided by Konnector, as well as several filters that can benefit our use case companies. For instance, Idiada can now benefit from important storage savings in block volumes that store the results of car crash simulations thanks to novel data reduction filters.

In the third year, our objective is to continue this path to create block filters that may exploit runtime monitoring information of the system dynamically. This may solve performance control and resource management problems in shared storage clusters, such as providing a certain minimum bandwidth per block volume. Concretely, we aim at integrating a bandwidth differentiation filter in Konnector that can provide QoS differentiation on a tenant's VMs. To this end, we will need to deploy algorithms as distributed controllers at the control plane that control the bandwidth of running VMs based on monitoring information; similarly to our bandwidth control service in object storage, at the data plane we need a client-side Konnector filter that can control the bandwidth of a given volume/VM based on the instructions from the control plane. This will enable companies like MPStor or Arctur to enforce bandwidth control on multiple data processing applications via simple SDS policies.

Unifying object filters and micro-controllers: In this deliverable, we presented Crystal as the main building block to deploy SDS services on top of OpenStack Swift. We demonstrated that Crystal enables powerful and flexible ways of deploying storage filters to solve a wide variety of problems from our use-cases (Idiada, Arctur), including data management or performance control services.

But as one can infer, this is not the only way of doing so. Very recently, we designed an alternative for orchestrating SDS services in Swift, called the *micro-controllers* approach [36]. Micro-controllers are like regular objects, but they can contain *state, triggers and code*: i) The state of a micro-controller can be, for instance, the number of `PUT` requests that an object or group of objects received since their creation; ii) Triggers in a micro-controller are invoked when an object receives a particular operation, or even after some time (e.g., auto-call trigger every 24 hours); iii) A micro-controller can also contain code to be executed on one or many objects after some conditions or triggers are met, such as compression or migration to another Swift ring. The main advantage of micro-controllers is the ability of managing the life-cycle of objects in a decentralized manner, so one can avoid the monitoring and computation overhead of Crystal's dynamic filters. We believe that micro-controllers are specially suitable for building storage tiering and cold/hot data filters, which can be efficiently implemented in a decentralized manner for the Idiada use case. Our objective is to integrate micro-controllers filter into the existing Crystal building block.

Generic pushdown for broader types of analytics: In deliverable D4.2, we presented the design and extensive evaluation of the pushdown mechanism: A way of delegating part of the analytics computations from the compute cluster to the storage cluster. We demonstrated that we could greatly speedup Spark SQL queries from GridPocket up to x30, which is very important for data analysts in that company.

However, we believe that this research can be widened along two axes:

- Pushdown of SQL filter from Spark SQL to a storlet is not always doable, for instance when the data set comprises lot of small to medium objects. In this case we will work at developing techniques that permit to address only the subset of the objects that are relevant to the SQL query, thus just skipping the data ingestion of the non relevant objects.

- SQL pushdown is only a specific case of a potentially generic form of intelligently parallelizing computations among storage and compute clusters. In the third year of the project, we will work on making the pushdown mechanism generic enough to speedup different types of popular analytics algorithms, putting special emphasis on facilitating its use from a data analyst perspective.

Hyper-converged analytics: Deliverable D5.2 presented Zoe; the IOStack building block for providing analytics-as-a-service in a cloud. Zoe provides three main advantages: i) It facilitates the deployment of new

applications in a cluster via simple policies or descriptors; ii) Zoe exhibits higher performance than traditional VM virtualization thanks to the orchestration of Docker containers; iii) Zoe is a substrate to develop application scheduling policies in a shared compute cluster. This makes it possible for Arctur —and other companies that use Zoe, such as AirFrance or KPMG— to rapidly deploy, customize and orchestrate analytics applications in shared clusters.

Unfortunately, Zoe currently does not care about the storage system that its applications are making use of. This may lead, for example, that scheduling policies fail to meet a deadline due to the lack of control of storage resources. In addition to this, one can easily infer that different analytics applications may exhibit very different storage usage patterns, which makes the problem even harder. In other words, in IOStack compute and storage subsystems are now totally isolated and uncoordinated, which may prevent us from exploiting a wide variety of "hyper-convergent" or "cross-layer" optimizations across compute and storage building blocks. In the last year of the project we will investigate the feasibility of introducing coordination strategies between compute and storage for improving the performance and efficiency of virtualized analytics.

12 Conclusions

In this deliverable, we presented the release of the IOStack toolkit for month 24. At this point of the project's development, we have an integrated SDS solution for Big Data analytics with several deployments running. The toolkit consists of three main building blocks for analytics virtualization (Zoe), block storage (Konnector) and object storage (Crystal), as well as an administration/monitoring dashboard and other components that complete the toolkit (Storlets, Stocator). We also presented that the different software components of IOStack are adhered to design principles described in D2.2 and integrated into a single toolkit. Moreover, this does not prevent that each software component of the IOStack toolkit can be exploited in a standalone manner.

In the second part of the deliverable we described in depth the architecture of Crystal: The IOStack building block that provides SDS for object storage. Specifically, Crystal pursues an efficient use of multi-tenant object stores that need to support non-anticipated requirements. Crystal addresses unique challenges for providing the necessary abstractions to add new functionalities at the data plane that can be immediately managed at the control plane. For instance, it adds a filtering abstraction to separate control policies from the execution of computations and resource management mechanisms at the data plane. Also, extending Crystal requires low development effort. We demonstrate the feasibility of Crystal on top of OpenStack Swift through two use cases that target automation and bandwidth differentiation making use of benchmarks and real workloads from our use case companies (Arctur, Idiada). Our results show that Crystal is practical enough to be run in a shared cloud object store.

13 Appendix 1: Analysis of the kernel oriented Bandwidth Differentiation Filter

13.1 Objectives of the Study

Bandwidth differentiation can be explained as the creation of a controlled unfair sharing of the resource. In our case we define bandwidth differentiation as the ability of the object store to apply unfair but controlled bandwidth per object.

In the following sections, we will analyse the behaviour of the IO Priorities method [48] that uses Linux kernel IO priorities. This method does not need any configuration setting to adapt to a new hardware. However,it needs to deploy a modified Swift to track transfer streams and due to how Linux I/O stack works on writes, does not allow to control PUT requests with priorities without changing the Linux kernel implementation of the I/O stack.

In section 13.2 the method is described and also an example of control plane that ensures distributed BW enforcement among all object servers. In section 13.3 we propose different experiments in order to evaluate the behavior of the priorities method on overloaded and normal situations. Finally, in section 13.5 a few conclusions are given.

13.2 Bandwidth control at the operating system level

Thanks to a new threading model developed by BSC [48], and presented on the WP2 previous deliverable (D2.2), we can individually track each stream of data at the object server. That was not possible on the original model due to the use of a thread pool and it is an essential service for Software Defined Storage. Thanks to this new feature, now we can apply different policies to each stream of data, for example, we can apply directly I/O priorities [49] to create bandwidth differentiation policies, controlling the bandwidth or throughput offered to one object (or group, tenant, etc.) by each object server. Having a request level control allows the operating system to automatically share spare disk bandwidth. This type of control is not possible at higher layers, for example at the middleware layer as a plug-in, due to we cannot prioritize our requests over other I/O threads (interferences).

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

Using such priority component simplifies the implementation as we can avoid using delays in the code. However, as requests can not be cancelled or moved to another object server and the HDD performance is non-linear, it is a best effort: If the requested bandwidth is not obtained, the client will need to cancel the ongoing transfer, and repeat it. The proxy server will intercept it, and sent it to another object storage (selected from some defined policies) that has enough capacity. Other policies can be implemented using more levels and controlling them inside the proxy or external controller. However, the actions that the controller can take are limited to modify the bandwidth allocated up or down to specialize storage servers and to reduce seek time, as we can not move requests from one server to another. So one more time we could only achieve a **best effort** bandwidth differentiation. Trying to guarantee the SLO, with such restrictions, per object is not an easy task inside the control plane as it depends on how requests and objects are distributed (we will show that behaviour on the three 50/20 experiments). However, all but one of the results presented show that at least the median value of the bandwidth obtained per experiment fulfills the requested bandwidth with a control window of the inner filter of 1500 chunks even in the overloaded scenarios.

Internally, our implementation increases the priority of a request if the sum of BW of all objects of a tenant is below the *needed BW* of that tenant, and marks as a low priority request if the BW exceeds the *needed BW* value. We are using only *Best Effort* 0 and *IDLE* priorities, so the spare bandwidth is distributed to the other I/O processes. Using a higher priority (higher than *IDLE*, *Best Effort* 4 for example) will share the non-needed bandwidth among all the I/O processes and the objects. On any scenario, requests needed to maintain the bandwidth should have a higher priority than *Best Effort* 4 (the default one). There is one parameter that we can tune to have more stable timelines, the window frame duration that the filter uses to compute the obtained bandwidth in order to decide to increase or decrease the priority. On these experiments, we used a small window

Parameter	Value
Medium File size	160000000 (bytes)
Number of medium files	100
Tiny file size	16000000 (bytes)
Number of tiny files	100
User count	5
Container count	10
Workers	10

Table 13.3a: SSBench parameters used for experiments.

(1500 chunks, near 1 second at 100MB/s) but we recommend to increase it as reactivity of the control plane is not compromised and the results are more stable.

Control plane: Bandwidth enforcement algorithm: IOStack [50] has a SDS controller[51] with agents that provide a control plane to adjust the bandwidth and do actions like distribute and specialize object servers to reduce seek time trying to increase the performance. This control plane receives the bandwidth usage per tenant and makes decisions based in different possible rules in order to set the bandwidth limit per object-server and obtain a desired global bandwidth.

13.3 Evaluation

To test bandwidth differentiation we use ssbench [46] as a benchmark. This benchmark imitates several distributed clients requesting objects from *Swift*. We can set different parameters as, for example, the number of objects, size of them, number of simultaneous requests, type of these requests and the number of clients, among others. In the experiments described in section 13.3.1 two tenants are executing two different ssbench loads, but with the same scenario parameters shown in table 13.3a. All the tests presented in the section 13.3.1 are executed in the IOStack testbed [52] provided by ARCTUR. Swift installation consists of three single HDD storage nodes plus one proxy node connected with 10 Gigabit Ethernet (so maximum sustained transfer speed to the proxy is near 120 MB/s). ssbench loads and SDS-Controller are executed on three different machines.

Bandwidth data is directly retrieved from the SDS-Controller bandwidth metric that is plotting the bandwidth usage to the IOStack dashboard. This bandwidth is obtained from a middleware in each Object Server that calculates the bandwidth usage per tenant.

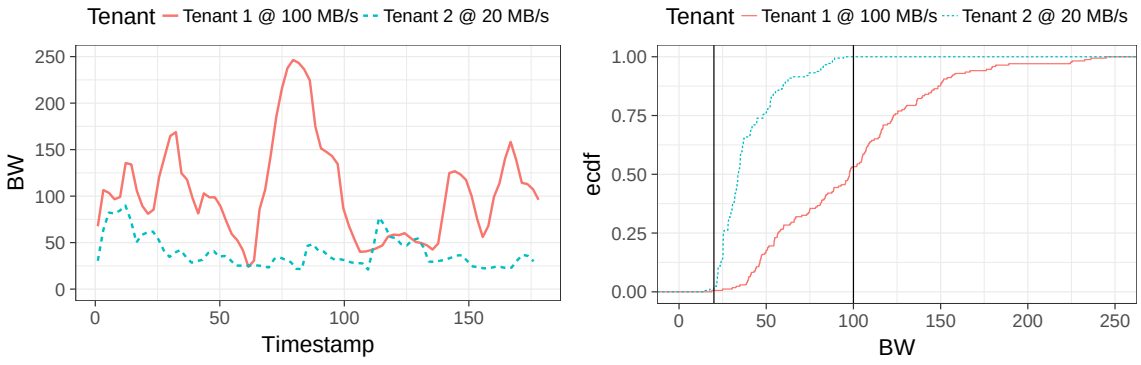
We also present results in the section 13.4 which show how the bandwidth differentiation can offer more performance than the original behaviour. Those test had been executed within a single object server and with big objects only.

13.3.1 Experimental Results

In this section we are going to evaluate the method proposed in different ways: having high bandwidth request (overload), middle bandwidth request (disk overload) and also considering possible interferences in the object servers (such as replication or other processes). We are interested on evaluating hard scenarios as they show how the component and Swift works clearly more than low demanding or stable workloads where the behaviour is predictable. All timeline plots have been smoothed (*loess* smoothing with 5% of the points) to have a better visualization of the results. Empirical Cumulative Distribution Frequency (ECDF) is calculated using the original data (not smoothed) in order to have real statistics about the bandwidth values. As a general rule, we want that, for a requested BW value, the ECDF curve shows a value under 0.5. That means that more than 50% of the points are above the requested BW value, so the mean of the whole execution is also above the requested BW value.

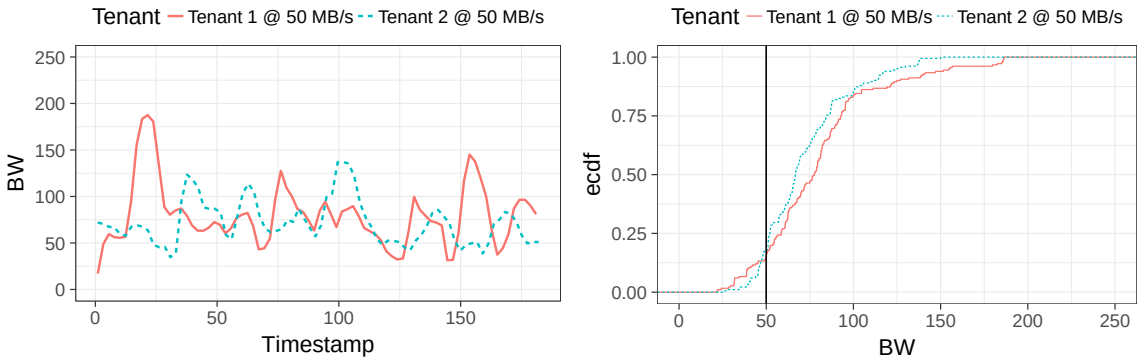
All logs and code for this evaluation can be found on the following github [53].

High Bandwidth requested (Overloaded) On this subsection we explore the results of the filter when we are on overload or near overload capabilities. Although we are on a distributed environment, and request go to different object servers, there is a net bottleneck on the proxy, so this is a high demanding scenario.



(a) Timeline (b) ECDF

Figure 21: 100 MB/s / 20 MB/s bandwidth assignments (overloaded).



(a) Timeline (b) ECDF

Figure 22: 50 MB/s / 50 MB/s bandwidth assignments.

In figure 21a we are running ssbench for two tenants and requesting 100Mb/s and 20Mb/s respectively. The figure shows the timeline of the instantaneous obtained BW per tenant. We obtain a sharp behaviour due to the overloaded setting. Looking into detail the ECDF plot in figure 21b, data can be read the following way: tenant 1 obtains more than 100 MB/s for the 50% of the points, and tenant 2 obtains more than 20 MB/s for the 99% of the points. We should note that we obtain more bandwidth for tenant 2 because the kernel sees that the disk is idle (for example, one server may serve only tenant 2 for a period of time due to request distribution). If Swift could move request from one server to another, tenant 1 could use that spare disk, but this is not possible. Finally, the data has high chances to be delayed by the network in this scenario.

In the second experiment with a near overloaded bandwidth assignation requests we try to see how the system behaves when we assign the same BW to two tenants. In this case we will assign 50 MB/s to each tenant and, as we can see in figures 22a and 22b, we obtain a similar behaviour for each tenant (as expected). We obtain bandwidth below 50 MB/s for 15% of the points and if we check for the 0.5 value we obtain around 75 MB/s for both tenants, so spare bandwidth is distributed evenly in this case (good distribution of requests, plus good distribution by the kernel of the spare bandwidth).

The last and most extreme experiment is pushing the needed BW to 100 MB/s for each tenant. As we can see in figure 23, the two curves are similar (it has more differences than the 50 / 50 due to the proxy overload). We did not get the requested bandwidth in mean as the network limits it to 120 MB/s, but we obtain 60 MB/s per tenant (1/2 of the theoretical maximum of the network) for more than the 63% of the points.

Medium bandwidth differentiation requests On this subsection we explore the results when we ask normal values that can be fulfilled by the different object servers. In this case we set lower BW needs in order to

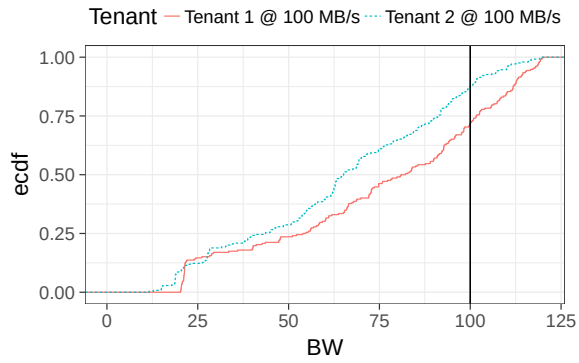
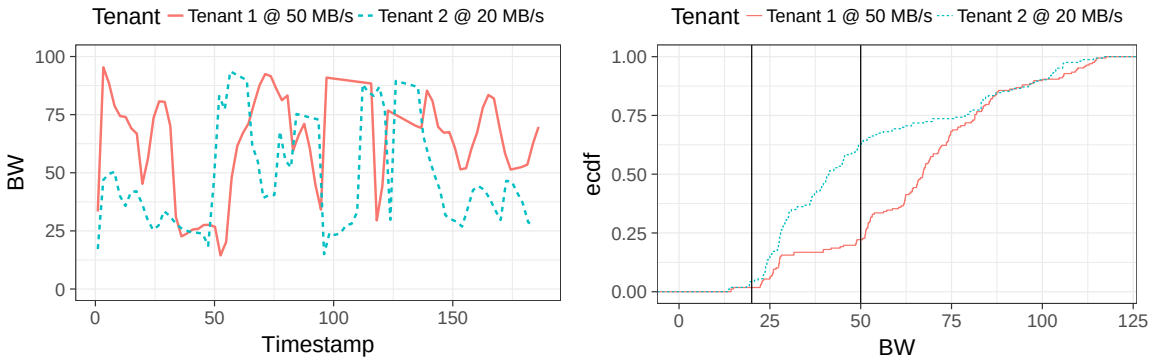


Figure 23: 100 / 100 MB/s bandwidth differentiation ECDF comparison.



(a) Timeline

(b) ECDF

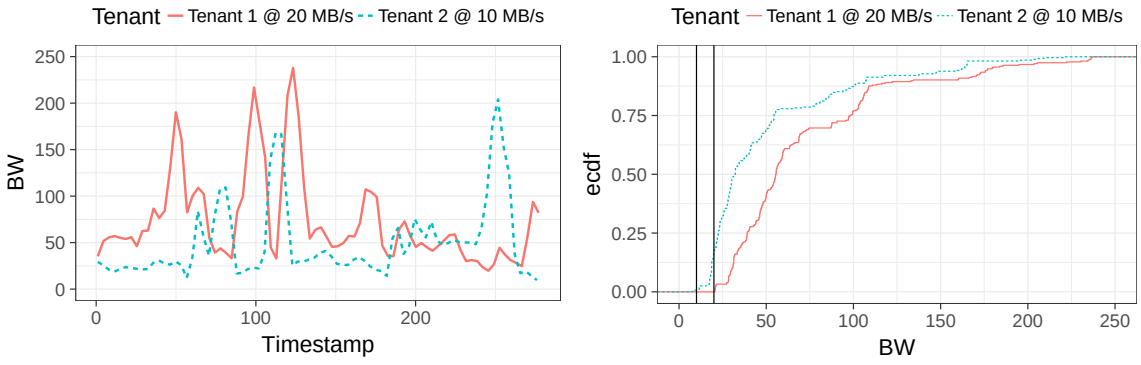
Figure 24: 50 MB/s / 20 MB/s bandwidth assignement (normal load, may hit HDD maximum performance).

avoid bottleneck problems in the proxy and get a smoother and better performance. To do that, the sum of the requested BW is not going to be over 10 GbE (120 MB/s).

In this experiment, we will assign 50 MB/s for one tenant and 20 MB/s for the other tenant. We can observe in figure 24a that normally the tenant 1 is always obtaining a bit more of bandwidth than tenant 2. As we can analyse in figure 24b we obtain more than 50 MB/s for the 75% of the points and more than 20 MB/s for the second tenant in more than 99% of the points. Data and request distribution has a high effect here, as we can see in figure 24a at the 50 seconds point. Tenant 2 obtains the requested bandwidth but tenant 1 obtains less breaking the SLO, at this point tenant 2 does not share the object server while tenant 1 has all its load at the same object server, dropping the performance by 10 MB/s per stream from the maximum of 70 MB/s. Repeating the experiment (as we will see on the interferences section) we may avoid this situation, but we decided to put it as it is a typical scenario that may happen as requests can not be moved or cancelled in the server side.

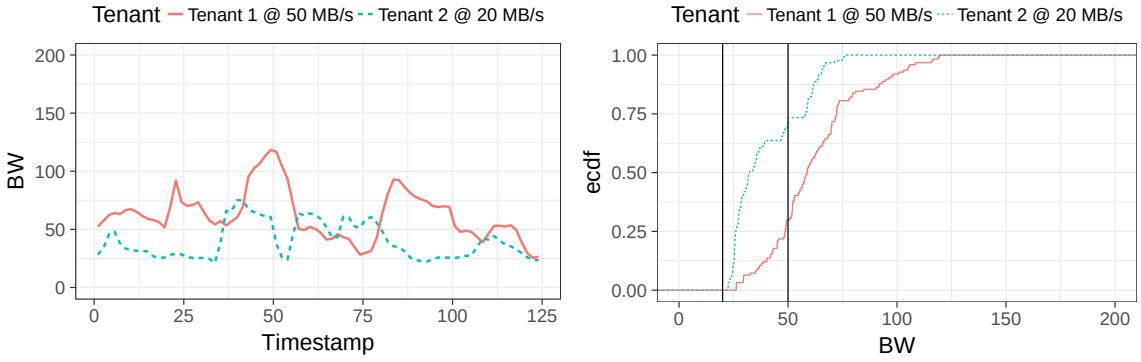
For the last experiment we will set very low bandwidth to each tenant (20 MB/s and 10 MB/s respectively) to see how the system behaves. In figure 25a we can see that if we do not overload the system the performance depends on how the requests are distributed among the different object-servers, as we can obtain better spare bandwidth utilization depending on how many streams are in each object server. In figure 25b, we can observe how we obtain more than the requested bandwidth for almost 99% of the points, having an average value of 60 MB/s for tenant 1 and 30 MB/s for tenant 2.

Interferences study On this subsection we explore the results of the two Bandwidth differentiation filters when we have a process doing I/O interferences (sustained 10 or 20 MB/s). This interferences are artificially generated by a simple python program that loops a file and reads it at a given bandwidth rate. As the interferences are run by python, they are automatically tagged by the kernel as *Best Effort 4*, so as IO Priorities move the scheduling point to the kernel and switch priorities of requests between *Best Effort 0* and *Idle*, we expect



(a) Timeline (b) ECDF

Figure 25: 20 MB/s / 10 MB/s bandwidth assignment (low load).



(a) Timeline (b) ECDF

Figure 26: 50 MB/s / 20 MB/s bandwidth assignment with a 10 MB/s sustained I/O interference.

that the effect of such interferences is reduced (in the cases we need the bandwidth to fulfill a requirement).

In this experiments we will do a medium bandwidth differentiation as before, we will request 50 MB/s for one tenant and 20 MB/s for the other tenant, and add on top of it sustained I/O interferences of 10 MB/s and 20 MB/s.

For the first experiment 10 MB/s interferences are generated. As we can see in figure 26a those interferences do not affect too much (compared to figure 24a). The ECDF plots (figure 26b) shows that we fulfill for the 75% of the points the 50 MB/s request and for all the points the 20 MB/s request. As we said before, this execution (more complex than the simple 50/20 without interferences) had distributed better the requests so the bandwidth assignment is fulfilled better than in the experiment without interferences (Figure 24b) as it can be observed in the timeline.

In the second experiment we will set the bandwidth of the interferences to 20 MB/s. In this scenario, we can see in the ECDF plot 27) how we obtain more than 50 MB/s for the 75 % of the points and 20 MB/s for more than 75 % of the points. Another time, despite the fact that we have interferences, the results due to the distribution of data and requests produces that the method works better than the experiment presented without interferences (Figure 24b).

13.4 Bandwidth differentiation effect inside an object server

In this small experiment, we will show the effect of the Swift modification and the bandwidth differentiation with respect the original Swift. We use only an object server and we configure the load with 300 MB objects, and 3 tenants.

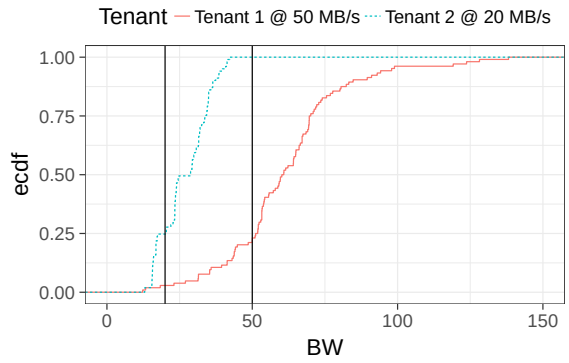


Figure 27: 50 MB/s / 20 MB/s bandwidth assignment ECDF with 20 MB/s sustained I/O interference.

Experiment	Tenant 1	Tenant 2	Tenant 3	Total BW
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 13.4a: Bandwidth differentiation using HDDs. Numbers are MB/s. Includes 95% confidence interval.

Table 13.4a presents the results of different bandwidth allocations, including no allocation and the original *Swift*. Here we obtain better performance, even without bandwidth allocation, due to a better scheduler behaviour as we have explored in the previous subsection. However, we achieve better disk performance when we offer different bandwidth at each tenant due to a more bursty and a behaviour prone to merge I/O. Observing the Max Priority line (we give infinite bandwidth to one tenant), it is interesting to note that we did not manage to get the Tenant 2 and Tenant 3 objects due to timeouts on the client side. On this situation, the client should request the object again and the proxy server will move it to another server. With the table results is easy to observe that the concept of "maximum bandwidth" or "maximum throughput" is hard to define on HDDs due to its dependency on the workload, thus having controls that use that concept as a metric or to distribute objects will produce wrong results as the "maximum" can not be predicted or calculated accurately.

13.5 Discussion of results

The bandwidth differentiation kernel method does not need specific configurations once it is installed. However, Swift needs to be modified in order to redirect each stream (object request) to its own thread (which removes two configuration parameters). Additionally, it does not support directly (without a kernel modification to tag write request with the original process id) to track PUT requests using the same methodology. On the other hand has several benefits, allows to share automatically the spare bandwidth from the disk, control the priority per chunk (64 KB) and it is able to avoid interferences. Other benefits come from a better I/O scheduler interaction, like an HDD increased performance due to the reduction of seeks, and the reduction of the number of I/O scheduling points to one in the kernel.

About the different results obtained, it has been clear in the 50 / 20 with and without interferences that the results depend on how the requests and data is distributed. So if a set of requests of one tenant go to a single object server, the HDD will be overloaded (each parallel request reduces performance by 10-15 MB/s) and the bandwidth will not be able to be fulfilled on that moment. Despite this, only the most demanding scenario evaluated (100/100, which is over the network bandwidth of 10GbE) is not achieving the requested bandwidth for more than the 50% of the points.

We include, for reference, results with a single node object server that demonstrates how the modification of Swift, along the bandwidth differentiation filter, show better performance on disks and why is important to do not relay on methods that define a maximum throughput of a HDD because it highly depends on the workload and how the I/O scheduler sends requests.

14 Appendix 2: Crystal Controller API

The next table summarizes the REST methods of Crystal Controller API. For further details and examples, refer to the documentation[10].

REST Call Description	HTTP Method	URL
Filters		
List filters	GET	`/filters`
Create a filter	POST	`/filters`
Upload filter data	PUT	`/filters/:filter_id/data`
Delete a filter	DELETE	`/filters/:filter_id`
Get filter metadata	GET	`/filters/:filter_id`
Update filter metadata	PUT	`/filters/:filter_id`
Deploy a filter to a project	PUT	`/filters/:project_id/deploy/filter_id`
Deploy a filter to a project and a container	PUT	`/filters/:project_id/:container/deploy/filter_id`
Undeploy a filter from a project	PUT	`/filters/:project_id/undeploy/filter_id`
Undeploy a filter from a project and a container	PUT	`/filters/:project_id/:container/undeploy/filter_id`
Dependencies		
Create a dependency	POST	`/filters/dependencies`
Upload dependency data	PUT	`/filters/dependencies/:dep_id/data`
Delete a Dependency	DELETE	`/filters/dependencies/:dep_id`
Get Dependency metadata	GET	`/filters/dependencies/:dep_id`
List Dependencies	GET	`/filters/dependencies`
Update Dependency metadata	PUT	`/filters/dependencies/:dep_id`
Deploy Dependency	PUT	`/filters/dependencies/:project_id/deploy/:dep_id`
Undeploy Dependency	PUT	`/filters/dependencies/:project_id/undeploy/:dep_id`
List deployed Dependencies of a project	GET	`/filters/dependencies/:project_id/deploy`
Workload metrics		
Add a workload metric	POST	`/registry/metrics`
Get all workload metrics	GET	`/registry/metrics`
Update a workload metric	PUT	`/registry/metrics/:metric_name`
Get metric metadata	GET	`/registry/metrics/:metric_name`
Delete a workload metric	DELETE	`/registry/metrics/:metric_name`

[10] https://github.com/Crystal-SDS/controller

Filters registry		
Register a filter	POST	`/registry/filters`
Get all registered filters	GET	`/registry/filters`
Update a registered filter	PUT	`/registry/filters/:filter_name`
Get registered filter metadata	GET	`/registry/filters/:filter_name`
Delete a registered filter	DELETE	`/registry/filters/:filter_name`
Projects group		
Add a projects group	POST	`/registry/gtenants`
Get all projects groups	GET	`/registry/gtenants`
Get projects of a group	GET	`/registry/gtenants/:group_id`
Update members of a projects group	PUT	`/registry/gtenants/:group_id`
Delete a projects group	DELETE	`/registry/gtenants/:group_id`
Delete a member of a projects group	DELETE	`/registry/gtenants/:group_id/tenants/:project_id`
Object type		
Create an object type	POST	`/registry/object_type`
Get all object types	GET	`/registry/object_type`
Get extensions of an object type	GET	`/registry/object_type/:object_type_name`
Update extensions of an object type	PUT	`/registry/object_type/:object_type_name`
Delete an object type	DELETE	`/registry/object_type/:object_type_name`
Metric modules		
Upload a metric module	POST	`/registry/metric_module/data`
Get all metrics modules	GET	`/registry/metric_module`
Get a metric module	GET	`/registry/metric_module/:metric_module_id`
Update a metric module	PUT	`/registry/metric_module/:metric_module_id`
Delete a metric module	DELETE	`/registry/metric_module/:metric_module_id`
DSL Policies		
List all static policies	GET	`/registry/static_policy`
Add a static policy	POST	`/registry/static_policy`
Get a static policy	GET	`/registry/static_policy/:project_policy_id`
Update a static policy	PUT	`/registry/static_policy/:project_policy_id`
Delete a static policy	DELETE	`/registry/static_policy/:project_policy_id`
List all dynamic policies	GET	`/registry/dynamic_policy`
Add a dynamic policy	POST	`/registry/dynamic_policy`
Delete a dynamic policy	DELETE	`/registry/dynamic_policy/:policy_id`
SLA Info for BW differentiation		
Get SLA info about all projects	GET	`/bw/slas`
Create a SLA for the selected project and policy	POST	`/bw/slas`
Get SLA info about a project and a policy	GET	`/bw/sla/:project_and_policy_id`
Edit a SLA for the selected project and policy	PUT	`/bw/sla/:project_and_policy_id`
Delete a SLA for the selected project and policy	DELETE	`/bw/sla/:project_and_policy_id`
Swift calls		
Enable SDS for a project	POST	`/swift/tenants`
Get a storage policies list	GET	`/swift/storage_policies`
Create a new storage policy	POST	`/swift/spolicies`
Obtain the locality of a project/container/object	GET	`/swift/locality/:project/:container/:swift_object`

15 Appendix 3: Crystal Development VM

There is a development virtual machine image available for Crystal. This VM emulates running a four node Swift cluster together with Keystone, Storlets and Crystal controller and middlewares.

The VM is accessible at `ftp://ast2-deim.urv.cat/s2caio_vm/`.

Glossary

Container (object storage): A container organizes and stores objects in Object Storage. Similar to the concept of a Linux directory but cannot be nested.

Domain-specific language (DSL): A computer language specialized to a particular application domain.

Logical Volume Manager (LVM): A device mapper target that provides logical volume management for the Linux kernel.

MD RAID: Also called Linux software RAID, makes the use of RAID possible without a hardware RAID controller.

Message Oriented Middleware (MOM): A software or hardware infrastructure supporting sending and receiving messages between distributed systems.

MOM broker: An intermediary program module that translates a message from the formal messaging protocol of the sender to the formal messaging protocol of the receiver. MOM brokers typically provide content and topic-based message routing using the publish/subscribe pattern.

Storage provisioning: The process of assigning storage, usually in the form of server disk drive space, in order to optimize the performance of a storage area network.

Storage tiering: the process of assigning different categories of data to various types of storage media to reduce total storage cost. Tiers are determined by performance and cost of the media, and data is ranked by how often it is accessed. Tiered storage policies place the most frequently accessed data on the highest performing storage. Rarely accessed data goes on low-performance, cheaper storage.

Tenant: represent the base unit of “ownership” in OpenStack, in that all resources in OpenStack should be owned by a specific tenant.

References

[1] S. LaValle, E. Lesser, R. Shockley, M. S. Hopkins, and N. Kruschwitz, "Big data, analytics and the path from insights to value," MIT sloan management review, vol. 52, no. 2, p. 21, 2011.

[2] D. Agrawal, S. Das, and A. El Abbadi, "Big data and cloud computing: current state and future opportunities," in Proceedings of the 14th International Conference on Extending Database Technology, pp. 530–533, 2011.

[3] T. White, Hadoop: The definitive guide. O'Reilly Media, Inc., 2012.

[4] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in IEEE MSST'10, pp. 1–10, 2010.

[5] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey, "Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language.," in USENIX OSDI'08, vol. 8, pp. 1–14, 2008.

[6] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in USENIX NSDI'12, pp. 2–2, 2012.

[7] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," Communications of the ACM, vol. 51, no. 1, pp. 107–113, 2008.

[8] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," Proceedings of the VLDB Endowment, vol. 2, no. 2, pp. 1626–1629, 2009.

[9] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al., "Spark sql: Relational data processing in spark," in ACM SIGMOD'15, pp. 1383–1394, 2015.

[10] X. Meng, J. Bradley, B. Yuvaz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, et al., "Mllib: Machine learning in apache spark," Journal of Machine Learning Research, vol. 17, no. 34, pp. 1–7, 2016.

[11] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," ACM SIGOPS operating systems review, vol. 37, no. 5, pp. 29–43, 2003.

[12] "Openstack swift." http://docs.openstack.org/developer/swift.

[13] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," ACM SIGOPS Operating Systems Review, vol. 44, no. 2, pp. 35–40, 2010.

[14] Eurecom, "Zoe." `https://github.com/DistributedSystemsGroup/zoe`.

[15] Jupyter, "Jupyter." `http://jupyter.org/`.

[16] Apache, "Spark." `http://spark.apache.org/`.

[17] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in ACM EuroSys'15, p. 18, 2015.

[18] Google, "Tensorflow." `https://www.tensorflow.org/`.

[19] "Ifttt." https://ifttt.com.

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

[21] S. R. Jeffery, G. Alonso, M. J. Franklin, W. Hong, and J. Widom, "A pipelined framework for online cleaning of sensor data streams," in IEEE ICDE'06, pp. 140–140, 2006.

[22] G. A. Agha, "Actors: A model of concurrent computation in distributed systems," tech. rep., The MIT Press, 1985.

[23] J. Armstrong, Programming Erlang: software for a concurrent world. Pragmatic Bookshelf, 2007.

[24] 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.

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

[26] 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.

[27] G. Cugola and A. Margara, "Processing flows of information: From data stream to complex event processing," ACM Computing Surveys (CSUR), vol. 44, no. 3, p. 15, 2012.

[28] "OpenStack Storlets." https://github.com/openstack/storlets.

[29] "Docker." https://www.docker.com.

[30] A. Gulati and P. Varman, "Lexicographic qos scheduling for parallel i/o," in ACM SPAA'05, pp. 29–38, 2005.

[31] Y. Wang and A. Merchant, "Proportional-share scheduling for distributed storage systems.," in USENIX FAST'07, 2007.

[32] E. Zamora-Gómez, P. García-López, and R. Mondéjar, "Continuation complexity: A callback hell for distributed systems," in LSDVE@Euro-Par'15, pp. 286–298, 2015.

[33] 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.

[34] I. Stefanovici, B. Schroeder, G. O'Shea, and E. Thereska, "sRoute: treating the storage stack like a network," in USENIX FAST'16, pp. 197–212, 2016.

[35] I. Stefanovici, E. Thereska, G. O'Shea, B. Schroeder, H. Ballani, T. Karagiannis, A. Rowstron, and T. Talpey, "Software-defined caching: Managing caches in multi-tenant data centers," in ACM SoCC'15, pp. 174–181, 2015.

[36] J. Sampé, M. Sánchez-Artigas, and P. García-López, "Vertigo: Programmable micro-controllers for software-defined object storage," in IEEE CLOUD'16, 2016.

[37] A. Gulati, I. Ahmad, C. A. Waldspurger, et al., "Parda: Proportional allocation of resources for distributed storage access.," in USENIX FAST'09, pp. 85–98, 2009.

[38] A. Gulati, A. Merchant, and P. J. Varman, "mclock: handling throughput variability for hypervisor io scheduling," in USENIX OSDI'10, pp. 1–7, 2010.

[39] A. Wang, S. Venkataraman, S. Alspaugh, R. Katz, and I. Stoica, "Cake: enabling high-level slos on shared storage systems," in ACM SoCC'12, p. 14, 2012.

[40] J. C. Wu and S. A. Brandt, "Providing quality of service support in object-based file system.," in IEEE MSST'07, vol. 7, pp. 157–170, 2007.

[41] N. Li, H. Jiang, D. Feng, and Z. Shi, "Pslo: enforcing the x th percentile latency and throughput slos for consolidated vm storage," in ACM Eurosys'16, p. 28, 2016.

[42] "Swift performance tunning." https://swiftstack.com/docs/admin/middleware/ ratelimit.html.

[43] D. Shue, M. J. Freedman, and A. Shaikh, "Fairness and isolation in multi-tenant storage as optimization decomposition," ACM SIGOPS Operating Systems Review, vol. 47, no. 1, pp. 16–21, 2013.

[44] "The Panasas activescale file system (PanFS)." http://www.panasas.com/products/panfs.

[45] "PVFS Project." http://www.pvfs.org/.

[46] Intel, "SSBench benchmark." `https://github.com/swiftstack/ssbench`.

[47] 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.

[48] IOSTACK, "Swift IO Bandwidth Differentiation Code." `https://github.com/iostackproject/IO-Bandwidth-Differentiation.git`.

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

[50] IOSTACK, "IOSTACK project code." `https://github.com/iostackproject/`.

[51] IOSTACK, "IOSTACK Openstack SDS-Controller enhancements code." `https://github.com/Crystal-SDS/controller`.

[52] IOSTACK, "IOSTACK testbed." `http://testbed.iostack.eu/`.

[53] M. Siquier, "Bandwidth tests logs." `https://github.com/marsqui/bw-tests`.