

IOWStack: Software-Defined Object Storage

As the complexity and scale of cloud storage systems grow, software-defined storage (SDS) has become a prime candidate to simplify cloud storage management. In this work, the authors present IOWStack: the first SDS architecture for object stores (such as OpenStack Swift). At the control plane, administrators provision SDS services to tenants according to policies expressed via a high-level DSL. At the data plane, IOWStack helps build a variety of filters, ranging from arbitrary computations on objects to data management mechanisms. Experiments illustrate that IOWStack enables easy and effective policy-based provisioning, which can significantly improve the operation of a multitenant object store.

Raúl Gracia-Tinedo, Pedro García-López, Marc Sánchez-Artigas, and Josep Sampé
Universitat Rovira i Virgili, Spain

Yosef Moatti, Eran Rom, and Dalit Naor
IBM Research—Haifa, Israel

Ramon Nou and Toni Cortés
Barcelona Supercomputing Center, Spain

William Oppermann
MPStor, Ireland

Pietro Michiardi
Institute Eurecom, France

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

Software-defined storage (SDS) has recently become a prime candidate to simplify storage management in the cloud. The incipient literature in the field states that SDS should provide a storage infrastructure with three items: automation, optimization, and policy-based provisioning.^{1,2} Typically, this is achieved by explicitly decoupling the control plane from the data plane at the storage layer.

Automation enables a datacenter administrator to easily provision resources and services to tenants. This includes the virtualization of storage services (volumes and filesystems) on top of performance-specific servers and network fabrics orchestrated by the SDS system. Optimization refers to the seamless ability to automatically allocate resources to meet the performance goals of the different workloads.² Finally, policy-based provisioning lets the datacenter administrator control I/O performance and other value-added services through the deployment of well-defined policies.¹ This includes, for instance, applying data-reduction techniques, computation, and I/O bandwidth differentiation on shared storage.³

Keeping this information in mind, we present IOWStack (<http://iowstack.eu>), the first SDS architecture for object storage (such as OpenStack Swift). In the following, we discuss what SDS

Related Work in Cloud Storage Management

In trying to simplify storage management in the cloud, IOStack benefits from the synergy between software-defined storage (SDS) and active storage.

SDS

IOFlow¹ describes the first SDS architecture — decoupled control and data planes — that provides policy-based provisioning. Although this work is inspiring, there are profound differences between IOFlow and our IOStack architecture. The most evident difference is that IOFlow is designed for a particular file system, whereas IOStack focuses on object storage. Moreover, IOFlow has a specific scope; it provides low-level I/O services (routing and classification) to control flows and guarantees I/O bandwidth limits. In contrast, IOStack's filter framework is more flexible and supports arbitrary computations on object requests. This enables heterogeneous filters to be easily added to the system.

Similarly, others² have recently proposed a system called Retro, which controls and monitors resource usage in a distributed system (control plane). Retro also enforces bandwidth and latency policies to guarantee a certain service-level agreement (or data plane). Although Retro isn't a complete SDS system, we believe that it's particularly interesting as a reference to build dynamic I/O bandwidth differentiation in IOStack.

Active Storage

The early concept of active disks³ — that is, hard drives with computational capacity — has been borrowed by distributed file system designers in high-performance computing environments (for example, active storage) for reducing the amount of data movement between storage and compute nodes. Concretely, Juan Piernas and his colleagues⁴ presented an active storage implementation integrated in the Lustre file system that provides flexible execution of code near data in the user

space. The industry also has taken remarkable steps in this direction by implementing commercial distributed file systems with compute power, such as the Panasas activescale file system (PanFS; www.panasas.com/products/panfs) and Parallel Virtual File System (PVFS; www.pvfs.org). Similarly, the filter framework of IOStack enables computations on data objects for policy enforcement. However, there are major differences between IOStack and previous efforts: first, previous work didn't focus on object storage; and second, IOStack provides isolated or sandboxed code execution.

Perhaps the closest technology to IOStack for leveraging active storage (IBM Storlets; see <https://github.com/openstack/storlets>) is ZeroCloud (www.zerovm.org/zerocloud.html). Both IBM Storlets and ZeroCloud rely on application containers — Docker and ZeroVM, respectively — for executing general-purpose code on Swift objects. However, the use of ZeroVM is more restrictive, as all code needs to be written in C and compiled via a proprietary tool chain. Further, Docker has advanced tools, such as Docker Swarm (<https://docs.docker.com/swarm>) or Zoe (<http://zoe-analytics.eu>), that will be exploited in IOStack to orchestrate the execution of filters in Swift nodes.

References

1. E. Thereska et al., "Ioflow: A Software-Defined Storage Architecture," *Proc. ACM Symp. Operating Systems Principles*, 2013, pp. 182–196.
2. J. Mace et al., "Retro: Targeted Resource Management in Multi-Tenant Distributed Systems," *Proc. Usenix Symp. Networked Systems Design and Implementation*, 2015; www.usenix.org/conference/nsdi15/technical-sessions/presentation/mace.
3. E. Riedel, G. Gibson, and C. Faloutsos, "Active Storage for Large-Scale Data Mining and Multimedia Applications," *Proc. Very Large Databases*, 1998, pp. 62–73.
4. J. Piernas, J. Nieplocha, and E.J. Felix, "Evaluation of Active Storage Strategies for the Lustre Parallel File System," *Proc. ACM/IEEE Supercomputing*, 2007, article no. 28.

technologies have contributed thus far, and how IOStack meets and exceeds the capabilities of current offerings.

Today's SDS Technologies

Today, SDS has become a buzzword to describe popular storage products such as EMC ViPR and IBM Spectrum Storage. Such products promise to make it easier for IT departments to handle large amounts of storage, by uncoupling the management from its underlying hardware. Other products, such as MPStor Orkestra, tap into storage virtualization and a centralized controller to provision a variety of virtualized storage and network resources.

Although these offerings have embraced this new way of managing storage, SDS goes beyond

automated resource provisioning.^{1,2} A distinctive feature of SDS is the ability to transparently enforce transformations on data flows based on simple policy definitions, as elaborated in IOFlow,¹ the first seminal work in the field. (We detail more on IOFlow and other cloud storage management efforts in the related sidebar.) The outstanding feature of IOFlow is that it decouples the data plane that enforces the policies from the control plane where the policy logic lies, allowing I/O control close to the source (typically a virtual machine, or VM) and destination (shared storage) endpoints.

However, full abstraction from the underlying hardware and storage stack isn't easy to achieve. For instance, the enforcement of policies can be done at the file, block, and object levels, making it difficult to apply the "one-size-fits-all" philosophy

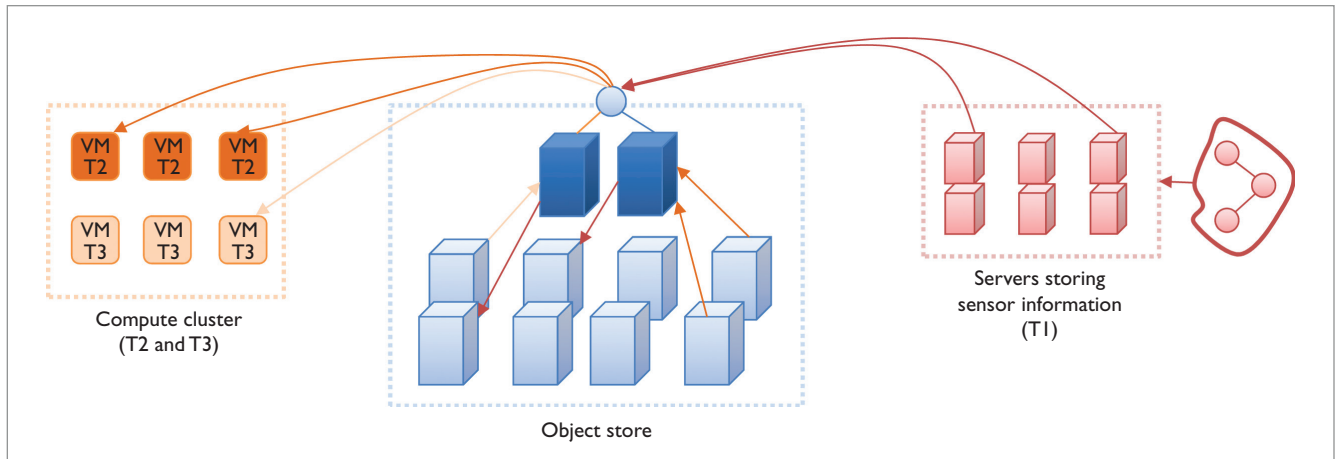


Figure 1. Example of an OpenStack Swift deployment (proxy nodes in dark blue, storage nodes in light blue) concurrently accessed by various tenants. Storage policies can be enforced on object requests to optimize the system and enrich the service.

to SDS. Whereas IOFlow can be classified as a file-level SDS architecture, there are no SDS systems for block and object storage yet.

Toward SDS for Object Storage

Object storage is becoming increasingly important for many customers and applications, because it's ideal for solving the increasing problems of data growth. As more data is generated, storage systems must grow at the same pace, which is difficult to achieve with block-based storage systems, for instance.

Object stores are suitable to store immutable data that might be subject to future analysis, such as server logs from Internet services,⁴ the upcoming data deluge of the Internet of Things (IoT),⁵ or even data coming from Web crawlers and sensor networks. There are also important synergies between object storage and Big Data scenarios⁶: DataBricks (<https://databricks.com>) – the company that develops Apache Spark – resorts to Amazon Web Services to deliver data processing services, including Simple Storage Service (S3). These disparate use cases can coexist in a multitenant object store, which reinforces our motivation for building an SDS architecture for object storage.

As a reference object store, we focus on OpenStack Swift (or simply Swift; see <http://docs.openstack.org/developer/swift>). Swift is accessed via a REST API similar to Amazon S3 (such as PUT or GET). Swift can be run on commodity servers and has been architected to automatically replicate data across available disks for providing scalability, availability, and

data integrity. Internally, Swift consists of proxy servers and storage nodes (see Figure 1). Proxy servers route user requests to the storage nodes that are the actual data containers and responsible for data maintenance and availability.

To better understand our goals, let's draw an example of a multitenant scenario. Imagine an object store and three different tenants that access the system concurrently. On the one hand, tenant T1 represents several servers that are uploading data gathered from a sensor network. On the other hand, tenants T2 and T3 represent sets of VMs in a computing cluster that perform computations on data objects containing logs (see Figure 1).

In such a scenario, a datacenter administrator might wish to define distinct policies for these tenants to optimize the system's operation or to enforce certain service-level agreements (SLAs). Intuitively, she could apply a data compression policy to T1 for reducing its storage space demands, given that log-like data is potentially redundant.⁷ Tenants T2 and T3, however, might apply data filters to import only the fraction of a dataset actually needed for a specific computation task, thus reducing download traffic.⁴ Further, the administrator might wish to assign different I/O bandwidth limits to the requests of T2 and T3.

As you can infer, the enforcement of these policies could permit an object store to manage concurrent workloads more efficiently. However, today's object stores lack a flexible and transparent way of enforcing storage policies on object requests. This is precisely the objective of IOStack.

IOStack's Design

The previous example opens the door to apply storage optimizations under multitenant workloads,² as well as to offer quality-of-service (QoS) differentiated policies based on a tenant's requirements. Moreover, from a datacenter administrator's perspective, these goals must be achieved transparently, involving minimal human intervention.

To realize this vision, we equipped IOStack with the following features:

- *Policy-based provisioning.* At the control plane, administrators provision SDS services to tenants via policies. Policies might target storage automation, such as enforcing compression or caching to a tenant's requests. Administrators might also define policies that target a certain objective or SLA. In this case, IOStack provides policies with a monitoring-based control loop to achieve their objective under dynamic workloads.
- *Filters.* At the data plane, filters perform actual data transformations on object requests to enforce storage policies. IOStack has a suitable architecture to favor the integration of new filters by third parties. IOStack also includes a ready-to-use filter framework that enables the execution of user code on object requests at different stages along an object's write/read path. A developer integrating a new filter only needs to contribute the filter's logic; the deployment and execution of the filter is managed by IOStack.

Next, we describe the design of policies in IOStack.

Administration: Storage Policies

Storage policies can be seen as a means of providing storage automation and/or SLA targets. In IOStack, datacenter administrators simply define provisioning policies to tenants via a simple domain-specific language (DSL). Each policy definition contains a target (for example, `TENANT` or `CONTAINER`), an action (`DO clause`), and optionally, a workload-based condition (`WHEN clause`). Hence, an administrator might define:

```
P1:FOR CONTAINER C1 DO SET CACHING
P2:FOR TENANT T1 WHEN PUTS_SEC > 3 DO
  SET COMPRESSION
P3:FOR TENANT T2 DO SET BANDWIDTH WITH
  PARAM1=30MBps
```

In this example, the first policy represents a storage-automation policy, as the system automatically performs data caching on container C1 after the definition of this policy. The second policy goes further, enabling data compression on tenant T1's requests if its throughput exceeds three PUTs per second. Similarly, the last policy aims at providing a certain amount of I/O bandwidth to T2, considering that multiple tenants might be transferring data concurrently. As we show later, objective-oriented policies require monitoring information to achieve their objectives.

Our DSL also supports grouping policies into QoS levels; that is, `GOLD` tenants might benefit from data compression, active storage tasks, and high bandwidth limits, whereas `BRONZE` tenants might receive only a small fraction of the available I/O bandwidth under multitenant workloads. Moreover, administrators can add workload metrics and actions dynamically to the language while the system is running.

As Figure 2 shows, policies feed the SDS controller. Next, we depict the role that the SDS controller plays in changing the system's behavior based on these policies.

Control Plane: SDS Controller

The SDS controller represents the IOStack's control plane. When an administrator defines a policy, the SDS controller checks its syntax and compiles it via the DSL compiler.

For storage automation policies, the compilation process ends by issuing an HTTP REST call to the appropriate filter-management API. Retaking the caching policy example, the REST call persists at the IOStack metadata store that caching should be enforced in container C1 (see P1 in Figure 2). From that point onward, data objects stored or retrieved from container C1 will be cached at the data plane. Policies applied to targets are persistently stored and replicated in the IOStack metadata store (based on Redis; see <http://redis.io>).

The compilation process for policies with a workload-based condition (for example, objective-oriented) is more complex. To wit, the DSL compiles policies as policy actors⁸ (similar to Ryan Stutsman and his colleagues' work⁹). Policy actors are processes that consume monitoring information to check if the workload satisfies the "WHEN clause" defined in the original policy. In the affirmative case, the policy actor

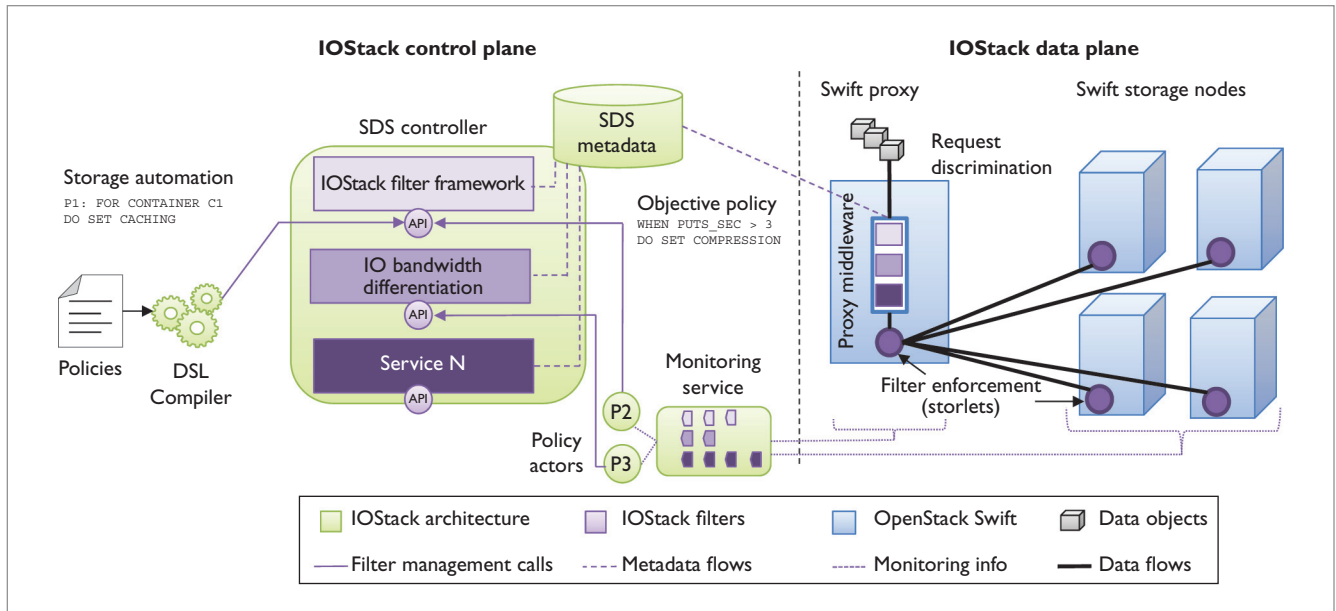


Figure 2. The IOSTack architecture and filter framework integrated in OpenStack Swift. Administrators have the ability to designate storage policies that feed the software-defined storage (SDS) controller in order to automatically enforce filters.

triggers a REST call to the appropriate filter-management API for automatically enforcing the policy.

Objective-oriented policies are possible thanks to the IOSTack monitoring system. IOSTack provides these policies with monitoring information to build a control loop. Thus, policies can trigger actions dynamically or execute distributed enforcement algorithms under workload changes.

IOStack integrates a message-oriented middleware (MOM) to disseminate monitoring information from the data plane (system resources metrics and tailored service metrics) to the control plane.³ Each workload metric is connected to a different queue at the MOM message broker. At the control plane, policy actors are subscribed to the workload metrics defined in the “WHEN clause,” enforcing a policy if the workload condition is satisfied. Figure 2 depicts this control loop.

Once a policy is stored as metadata in the metadata store, it’s accessible from storage filters at the data plane.

Data Plane: Filter Framework

At the data plane, filters are isolated software components that perform actual transformations on data objects. These transformations can be related to data content (such as compression or computation) or to data management (such as caching or I/O bandwidth differentiation).

Although you can integrate independent filter implementations in IOSTack, we provide

a filter framework that enables developers to run general-purpose code on object requests. IOSTack borrows ideas from active storage literature^{10,11} as a means of building filters to enforce policies.

The core of IOSTack’s filter framework is based on IBM Storlets (<https://github.com/openstack/storlets>). Storlets extend Swift with the capability to run computations near the data in a secure and isolated manner, making use of Docker (www.docker.com) as the application container. With Storlets a developer can write code, package and deploy it as a Swift object, and then explicitly invoke it on data objects as if the code was part of the Swift pipeline. 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. The Storlet engine executes a particular binary when the HTTP request for a data object contains the correct metadata headers in which it’s specified.

The filter framework in IOSTack has three main components: metadata and code management; request classification; and sandboxed filter execution.

Metadata and code management. This module resides at the SDS controller and exposes a high-level API to enable the management of filter and tenant relationships, and to manage filter binaries.

Request classification. Our framework discriminates the filters to be applied on a particular data flow at the Swift's proxy. Technically, an IOStack module in the Swift proxy middleware contacts the metadata store to infer the filters to be executed on a tenant's request. Given that, it sets the appropriate HTTP headers to the incoming request (such as GET or PUT) for triggering the subsequent filter execution. Moreover, in IOStack data objects are stored and replicated with an extended metadata to keep track of the executed filters that changed their content (such as compression). We use such metadata to trigger inverse transformations on GET requests.

Sandboxed filter execution. Upon the arrival of a tenant's request with the appropriate HTTP headers, a filter can then be executed either at proxy or storage node stages; a decision that depends on the filter developer. For instance, a compression filter can efficiently be performed at the proxy, whereas computing tasks on data objects might be more suitable at the storage node.

```
public class StorletName implements
IStorlet{

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

        //Filter code here
    }
}
```

The code snippet shows that developing a new filter in our framework is simple. A 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. The ambition of IOStack is to ease the development of new filters by the community to become a rich open source SDS system.

As we show next, the IOStack filter framework can support many filter types, such as

data reduction, storage optimization, and general computations on data objects.

Early Experiences

In our experiments, we execute in parallel workloads of tenants T1 (write-only) and T2 (read-dominated). For T1, we resort to an object-storage benchmark (`ssbench`; see <https://github.com/swiftstack/ssbench>) that uploads 32,000 synthetic text objects of 10 Mbytes in size using four threads. T2 is represented by a Spark instance (three worker VMs and one master VM) that downloads an existing log file of 164 Gbytes in size (64-Mbyte splits, in .csv format). After downloading the log, T2 performs a simple word count task on the `user_id` field to calculate the number of user occurrences.

Our hardware consists of a 12-machine cluster formed by three high-end computing nodes and eight storage nodes, plus one node that acts as a proxy. Machines are connected via 1-gigabit switched-network links. Compute nodes virtualize the Spark instance (T2), whereas storage nodes and the proxy run Swift and our IOStack prototype (the SDS controller and filter framework). We execute `ssbench` in other servers at Universitat Rovira i Virgili, so T1's PUT requests access our cluster from the Internet. Our cluster runs a complete OpenStack Kilo installation.

Storage Automation under Multitenancy

Next, we reproduce a multitenant scenario somewhat based on Figure 1 to assess IOStack's benefits compared to Swift.

Benefits for T1. T1 is a write-oriented tenant that uploads log-like data to the system. Therefore, we enforced in IOStack a compression policy – a filter that uses `gzip` – to tenant T1 to improve transfer performance and minimize storage usage. Hence, scatter plots in Figures 3a and 3b show the throughput of T1's PUT requests (`ssbench`) and T2's GET requests (Spark), for both Swift and IOStack.

Observably, due to the parallelism of PUT requests, the Swift proxy can't deliver to T1 more than 30 Mbytes per second (Mbps) per request. Furthermore, when Spark starts downloading data, the throughput of both tenants decreases drastically: most concurrent requests exhibited a throughput around 4–6 Mbps.

Conversely, IOStack performs significantly better for PUT requests of T1 due to the enforcement

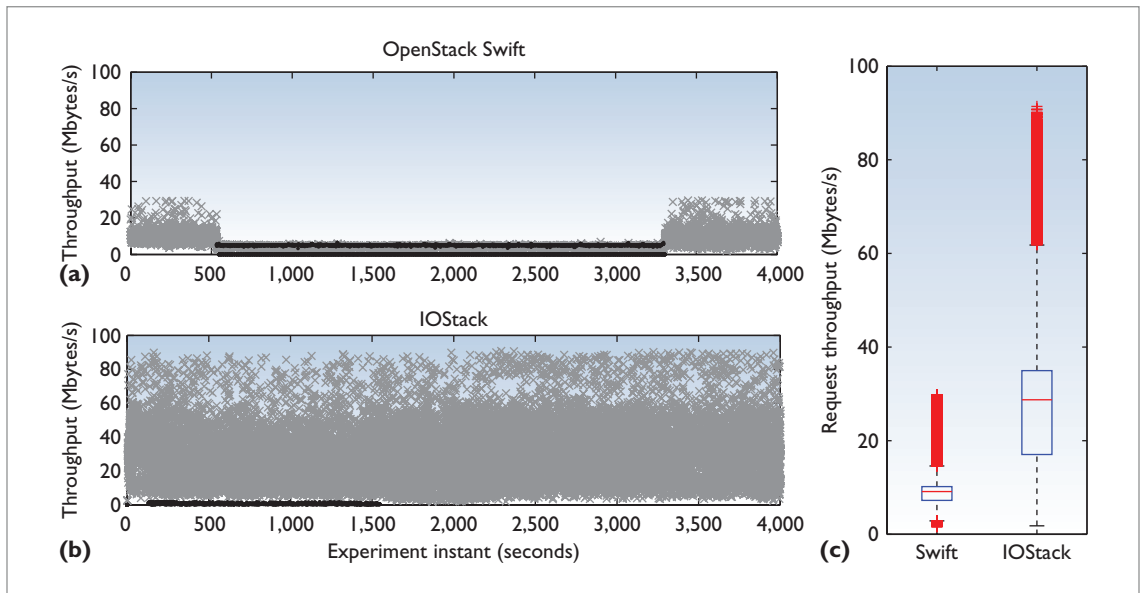


Figure 3. Comparison of Swift and IOStack in a multitenant scenario. (a and b) Scatter plots show the throughput of tenants' requests and (c) the boxplot depicts the throughput of PUT requests for T1.

of a compression policy on highly redundant data. That is, the boxplot in Figure 3c demonstrates that IOStack can achieve a median write throughput of $3\times$ higher than Swift. Furthermore, as visible in Figure 3b's scatter plot, T1's PUT operations are only slightly affected when T2 starts its activity.

Apart from transfer gains, IOStack also involves important storage space savings. To wit, along the experiment T1 stored 312 Gbytes of data in Swift – considering three-way replication, the actual amount of consumed storage is 936 Gbytes. Due to the high redundancy of data produced by *ssbench*,⁹ IOStack compressed T1's data to 0.1 percent of its original size.

Benefits for T2. T2 uses Spark to download a dataset and count the total number of user ID occurrences on it.

We noted that T2 only needs a fraction of the dataset to carry out such a task (for example, the user ID field). Thus, we enforced in IOStack a computing-close-to-data policy that filters on the server side the data actually needed by T2. Intuitively, such an active storage filter can yield two advantages for T2: first, to reduce the total amount of data to be transferred from the object store to the computing cluster; and second, to decrease data processing times.

First, we noted that filtering the dataset at the source enables an important reduction of

bandwidth for T2. Specifically, retrieving only the user ID field instead of all fields per line of log reduces the amount of outgoing bandwidth by 95.6 percent. Although the throughput of T2's transfers is lower for IOStack due to filtering overhead and the smaller object size, the traffic reduction greatly amortizes these penalties.

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

Benefits for the administrator. These results are interesting from a performance perspective. However, the major benefit of IOStack is to provide a datacenter's administrator with a simple way of enforcing storage policies to object requests. Overall, our experiments certify that IOStack enables easy and effective enforcement of a wide range of policies (data reduction and computation), which can greatly improve the operation of a multitenant object store.

Dynamic Provisioning

Next, we examine the operation of dynamic storage policies in IOStack. That is, Figure 4

shows T1 performing PUT requests with increasing intensity. Then we defined a dynamic policy that enforces data compression on T1's requests if T1 exhibits 3 PUTs per second (see P2 in Figure 2).

Figure 4 shows that our monitoring system updates the number of PUTs per second executed by T1 (the gray area). Then, the policy actor subscribed to the PUT/second metric detects that the workload of T1 satisfies the condition, and triggers the enforcement of a compression filter. From that point onward, requests are compressed and, due to the redundancy of data objects, they exhibit higher throughput. This demonstrates the ability of IOStack to manage dynamic storage policies, which could apply to a wide variety of filters.

We presented IOStack, the first SDS architecture for object storage (OpenStack Swift). IOStack enables policy-based provisioning: From an administrator viewpoint, policies define the enforcement of data services, namely filters, on a tenant's requests. Moreover, in IOStack filters can be built as independent components or integrated in our filter framework, which enables developers to write code – such as data reduction or optimization techniques – to be transparently executed on object requests. Our experiments certify that IOStack represents a step toward improving the administration and operation of object stores.

Despite its potential, IOStack is only the first step of an ambitious project (<https://github.com/iostackproject>). For object storage, we're currently working on the dynamic orchestration of filters in the IOStack filter framework, based on the resources that filters consume during their execution. We're also exploring ways of automatically detecting conflicting filters – or wrong filter ordering – enforced on the same tenant to simplify filter management. Furthermore, we're developing a block-storage version of IOStack for providing unified SDS management of both block and object storage.

Acknowledgments

This work has been funded by the European Union through project H2020 “IOStack: Software-Defined Storage for Big Data” (644182) and by the Spanish Ministry of Science and Innovation through project “Servicios Cloud y Redes Comunitarias” (TIN-2013-47245-C2-2-R).

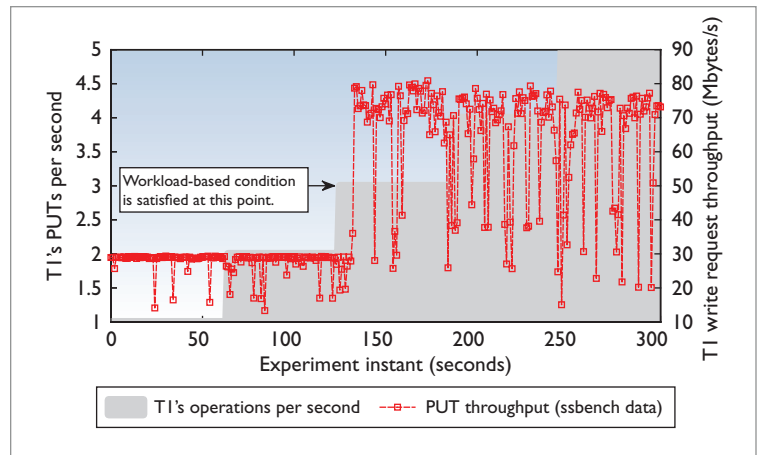


Figure 4. Example of a dynamic storage policy. When T1's requests reach the workload condition, the system automatically triggers compression.

References

1. E. Thereska et al., “Ioflow: A Software-Defined Storage Architecture,” *Proc. ACM Symp. Operating Systems Principles*, 2013, pp. 182–196.
2. A. Alba et al., “Efficient and Agile Storage Management in Software Defined Environments,” *IBM J. Research and Development*, vol. 58, nos. 2/3, 2014, pp. 1–5.
3. J. Mace et al., “Retro: Targeted Resource Management in Multi-Tenant Distributed Systems,” *Proc. Usenix Symp. Networked Systems Design and Implementation*, 2015; www.usenix.org/conference/nsdi15/technical-sessions/presentation/mace.
4. D. Logothetis et al., “In-Situ MapReduce for Log Processing,” *Proc. Usenix Ann. Technical Conf.*, 2011, p. 9.
5. L. Atzori, A. Iera, and G. Morabito, “The Internet of Things: A Survey,” *Computer Networks*, vol. 54, no. 15, 2010, pp. 2787–2805.
6. R. Riffe, “Big Data Needs Software-Defined Storage,” *InfoWorld*, 9 Apr. 2014; www.infoworld.com/article/2610828/infrastructure-storage/big-data-needs-software-defined-storage.html.
7. R. Gracia-Tinedo et al., “SDGen: Mimicking Datasets for Content Generation in Storage Benchmarks,” *Proc. Usenix Conf. File and Storage Technologies*, 2015, pp. 317–330.
8. E. Zamora-Gómez, P. García-López, and R. Mondéjar, “Continuation Complexity: A Callback Hell for Distributed Systems,” LNCS 9253, Springer, 2015, pp. 286–298.
9. R. Stutsman, C. Lee, and J. Ousterhout, “Experience with Rules-Based Programming for Distributed, Concurrent, Fault-Tolerant Code,” *Proc. Usenix Ann. Technical Conf.*, 2015, pp. 17–30.
10. E. Riedel, G. Gibson, and C. Faloutsos, “Active Storage for Large-Scale Data Mining and Multimedia Applications,” *Proc. Very Large Databases*, 1998, pp. 62–73.

11. J. Piernas, J. Nieplocha, and E.J. Felix, "Evaluation of Active Storage Strategies for the Lustre Parallel File System," *Proc. ACM/IEEE Supercomputing*, 2007, article no. 28.

Raúl Gracia-Tinedo is a postdoc in the Architectures and Telematic Services Research Group at Universitat Rovira i Virgili (URV), Spain. His research interests include distributed storage management, cloud computing, and performance evaluation of systems. Gracia-Tinedo received his PhD in computer engineering from URV in 2015. He received the Best Dataset Award at the 2015 ACM Sigcomm Internet Measurement Conference. Contact him at raul.gracia@urv.cat.

Pedro García-López is a professor in the Computer Engineering and Mathematics Department at URV, where he also leads the Architectures and Telematic Services Research Group. His research interests include distributed systems, peer-to-peer systems, cloud storage, software architectures, and middleware and collaborative environments. García-López has a PhD in computer science from University of Murcia. Contact him at pedro.garcia@urv.cat.

Marc Sánchez-Artigas is an assistant professor at URV. His research interests include building massive distributed systems and clouds, including peer-to-peer and novel storage systems. Sánchez-Artigas has a PhD in computer science from the Universitat Pompeu Fabra, Barcelona. He received the Best Paper Award at the 2007 IEEE Local Computer Networks Conference. Contact him at marc.sanchez@urv.cat.

Josep Sampé is a PhD student in the Department of Computer Engineering and Mathematics at URV. His research interests include software-defined storage management and cloud computing. Sampé has a BSc in computer engineering and mathematics from URV. Contact him at josep.sampe@urv.cat.

Yosef Moatti is a member of the research staff at IBM Research-Haifa, Israel. His research interests include Big Data analytics and storage frameworks. Moatti has a PhD in computer science from Télécom ParisTech. Contact him at moatti@il.ibm.com.


Eran Rom is a member of the research staff at IBM Research-Haifa. His research interests include distributed, scalable, and available storage systems. Rom has an MS in computer science from Tel-Aviv University. Contact him at eranr@il.ibm.com.


Dalit Naor is a senior manager of the Cloud Platforms Department at IBM Research-Haifa and a senior technical staff member. Her research interests include cloud object stores and its integration with analytics, and specializes in storage aspects such as cloud storage, data reduction techniques, long-term digital preservation, and power-efficient storage systems. Naor has a PhD in computer science from the University of California, Davis. Contact her at dalit@il.ibm.com.

Ramon Nou is a member of the Storage Systems Team at the Barcelona Supercomputing Center, Spain. His research interests include energy efficiency, optimization, simulation, and storage systems. Nou has a PhD in computer science from Universitat Politècnica de Catalunya (UPC). Contact him at ramon.nou@bsc.es.

Toni Cortés is an associate professor at the Computer Architecture Department at UPC and the manager of the Storage System Research Group at the Barcelona Supercomputing Center. His research interests include operating systems, middleware, and runtimes for parallel machines and clusters of workstations. Cortés has a PhD in computer science from UPC. Contact him at toni.cortes@bsc.es.

William Oppermann is a co-founder of MPStor. His research interests include the information and communications technologies industry, and in particular the enterprise data storage industry. Oppermann has an MscEng in engineering from University College Dublin. Contact him at wo@mpstor.com.

Pietro Michiardi is a professor of computer science at Institute Eurecom, where he leads the Distributed Systems Group. His research interests include large-scale distributed systems (including data processing and data storage), and scalable algorithm design to mine massive amounts of data. Michiardi has a PhD in computer science from Telecom ParisTech, as well as an HDR (Habilitation) from Universidad Nacional de San Agustín. Contact him at pietro.michiardi@eurecom.fr. 

 Selected CS articles and columns are also available for free at <http://ComputingNow.computer.org>.