IOStack: Software-Defined Object Storage

Raúl Gracia-Tinedo, Pedro García-López, Marc Sánchez-Artigas, Josep Sampé
Universitat Rovira i Virgili
Tarragona, Spain
pedro.garcia,marc.sanchez, raul.gracia,josep.sampe@urv.cat

Yosef Moatti, Eran Rom, Dalit Naor
IBM Research
Haifa, Israel
moatti,eranr,dalit@il.ibm.com

Ramon Nou, Toni Cortés
Barcelona Supercomputing Center
Barcelona, Spain
ramon.nou,toni.cortes@bsc.es

William Oppermann
MPStor
Cork, Ireland
wo@mpstor.com

Pietro Michiardi
Eurecom
Sophia-Antipolis, France
pietro.michiardi@eurecom.fr

ABSTRACT

The complexity and scale of today's cloud storage systems is growing fast. In response to these challenges, Software-Defined Storage (SDS) has recently become a prime candidate to simplify storage management in the cloud.

This article presents IOStack: The first SDS architecture for object stores (OpenStack Swift). At the control plane, the provisioning of SDS services to tenants is made according to a set of *policies* managed via a high-level DSL. Policies may target storage automation and/or specific SLA objectives. At the data plane, policies define the enforcement of SDS services, namely *filters*, on a tenant's requests. Moreover, IOStack is a framework to build a variety of filters, ranging from general-purpose computations close to the data to specialized data management mechanisms.

Our experiments illustrate that IOStack enables easy and effective policy-based provisioning, which can significantly improve the operation of a multi-tenant object store.

1. INTRODUCTION

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

Software-Defined Storage (SDS) has recently become a prime candidate to simplify storage management in the cloud. The incipient literature in the field states that SDS should provide a storage infrastructure with i) *automation*, ii) *optimization*, and iii) *policy-based provisioning* [15, 7]. 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, 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 [7]. Finally, *policy-based provisioning* allows to control IO performance and other value-added services through the deployment of well-defined policies [15]. This includes, for instance, the application of data reduction techniques, computation and IO bandwidth differentiation on shared storage [11].

1.1 Today's SDS Technologies

Today, SDS have become a commercial buzzword to describe popular storage products such as EMC ViPR and IBM Spectrum Storage. These offerings promise IT departments to better 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 [15, 7]. A distinctive feature of SDS is the ability of transparently enforcing transformations on data flows based on simple policy definitions as elaborated in IOFlow [15], the first seminal work in the field. 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 IO control close to the source (typically, a VM) and destination (shared storage) endpoints.

However, full abstraction from the underlying hardware and storage stack is not easy to achieve. For instance, the enforcement of policies can be done at the *file*, *block* and *object* levels, making it hard to apply the "one-size-fits-all" philosophy to SDS. Whereas IOFlow can be classified as a *file-level* SDS architecture, there are no SDS systems for block and object storage yet.

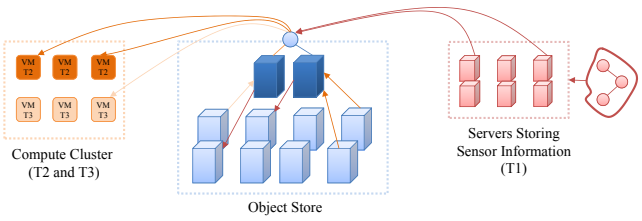

Figure 1: Example of an OpenStack Swift deployment (proxy nodes in dark blue, storage nodes in light blue) concurrently accessed by various tenants. Storage policies may be enforced on object requests to optimize the system and enrich the service.

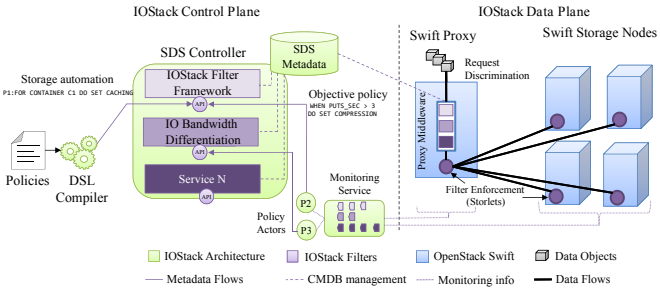

Figure 2: IOStack architecture and filter framework integrated in OpenStack Swift.

2. TOWARDS SDS FOR OBJECT STORAGE

Object storage is becoming increasingly important for many customers and applications, as it is ideal for solving the increasing problems of data growth. As more and more data is generated, storage systems have to 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 may be subject to future analysis, such as server logs from Internet services [10], the upcoming data deluge of the Internet-of-Things [8], or even data coming from web crawlers and sensor networks. There are also important synergies between object storage and Big Data scenarios [6]: DataBricks[1] —the company that develops Apache Spark— resorts to Amazon Web Services to deliver data processing services, including S3. These disparate use cases can coexist in a multi-tenant object store, which reinforces our motivation for building SDS object storage architecture.

As a reference object store, we focus on the OpenStack Swift (or simply *Swift*) [2]. Swift is accessed via a REST API similar to Amazon S3 (e.g., `PUT`, `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 Fig. 1). Proxy servers route user requests to the storage nodes that are the actual data containers and responsible for data maintenance and availability.

2.1 A Motivating Example

To better understand our goals, let us draw an example of a multi-tenant scenario. Imagine an object store and 3 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 virtual machines in a compute cluster that perform computations on data objects containing logs. This is shown in Fig. 1.

In such scenario, a datacenter administrator may wish to define distinct *policies* for these tenants to optimizing the system's operation or to enforce certain SLAs. Intuitively, he could apply a data compression policy to *T1* for reducing its storage space demands, given that log-like data is potentially redundant [9]. Tenants *T2* and *T3*, however, may apply data filters to import only the fraction of a dataset actually needed for a specific computation task, thus reducing download traffic [10]. Further, he may wish to assign different IO bandwidth limits to the requests of *T2* and *T3*.

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

3. IOSTACK DESIGN

The previous example opens the door to apply *storage optimizations* under multi-tenant workloads [7], 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 need to be achieved transparently, involving minimal human intervention.

To realize this vision, we present IOStack[2]. It features:

- **Policy-based provisioning**: At the control plane, administrators provision SDS services to tenants via *policies*. Policies may target *storage automation*, such as enforcing compression or caching to a tenant's requests. Administrators may 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.

4. 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* (e.g., `TENANT`, `CONTAINER`), an *action* (`DO` clause), and optionally, a *workload-based condition* (`WHEN` clause). Hence, an administrator may define:

[1]https://databricks.com

[2]http://iostack.eu

```
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 as it enables data compression on tenant `T1`'s requests if its throughput exceeds 3 `PUT`s per second. Similarly, the last policy aims at providing a certain amount of IO bandwidth to `T2`, considering that multiple tenants may be concurrently transferring data. As we show later on, objective-oriented policies require monitoring information to achieve their objectives.

Our DSL also supports grouping policies into QoS levels; that is, `GOLD` tenants may benefit from data compression, active storage tasks and high bandwidth limits, whereas `BRONZE` tenants may receive only a small fraction of the available IO bandwidth under multi-tenant workloads. Moreover, workload metrics and actions can be dynamically added to the language while the system is running.

As shown in Fig. 2, policies feed the SDS Controller. Next, we depict the role that the SDS Controller plays on changing the system's behavior based on these policies.

5. CONTROL PLANE: SDS CONTROLLER

The *SDS Controller* represents the IOStack's *control plane*. When an administrator defines a policy, the SDS Controller check 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 Fig. 2). From that point onwards, data objects stored/retrieved from container *C1* will be cached at the data plane.

The compilation process for policies with workload-based condition (i.e., objective-oriented), is more complex. To wit, the DSL compiles policies as *policy actors*[3] (similar to [14]). 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 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 that we describe next.

5.1 Monitoring for Dynamic Provisioning

IOStack provides objective-oriented policies with monitoring information to build a *control loop*. Thus, policies may dynamically trigger actions 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, tailored service metrics) to the control plane [11]. 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. Fig. 2 depicts this control loop.

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

[3]https://en.wikipedia.org/wiki/Actor_model

6. DATA PLANE: FILTER FRAMEWORK

At the data plane, filters are isolated software components that perform actual transformations on data objects. These transformations may be related to the *contents of data* (e.g., compression, computation) or to the *management of data* (e.g., caching, IO bandwidth differentiation).

Although independent filter implementations can be integrated 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 [13, 12] as a means of building filters to enforce policies.

The core of IOStack's filter framework is based on IBM Storlets [1]. Storlets extend Swift with the capability to run computations near the data in a secure and isolated manner making use of `Docker`[4] as 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 specifying to do so.

The filter framework in IOStack has 3 main components:

Metadata and code management: This module resides at the SDS Controller and exposes a high-level API i) to enable the management of filter/tenant relationships, and ii) 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 has the notion of which filters should be executed on a tenant's request. Given that, it sets the appropriate HTTP headers in the incoming request (e.g., GET, PUT) in order to trigger the subsequent filter execution.

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 compute tasks on data objects may be more suitable at the storage node.

```
public class StorletName implements IStorlet {

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

    //User 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.

[4]https://www.docker.com

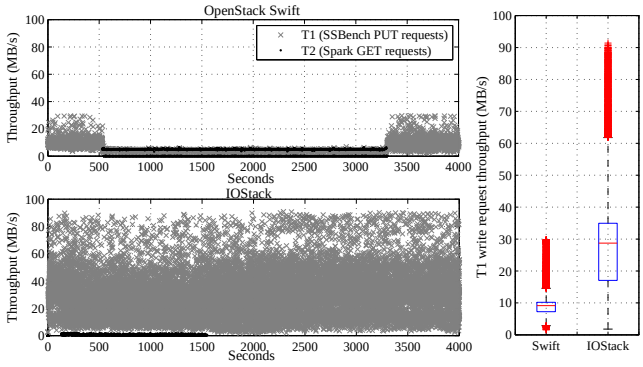

Figure 3: Comparison of Swift and IOStack in a multi-tenant scenario. Scatter plots show the throughput of tenants' requests and the boxplot depicts the throughput of `PUT` requests for *T1*.

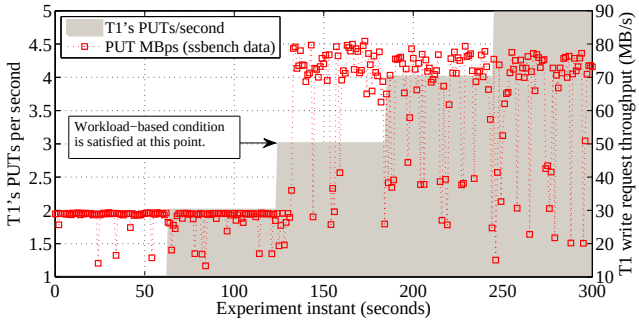

Figure 4: Example of a dynamic storage policy. When *T1*'s requests reach the workload condition, the system automatically triggers compression.

7. 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`[5]) that uploads 32K synthetic text objects of 10MB in size using 4 threads. *T2* is represented by a Spark instance (3 worker VMs, 1 master VM) that downloads an existing log file of 164GB in size (64MB splits, `.csv` format). After downloading the log, *T2* performs a simple word count task on the `user_id` field to calculate the number of occurrences of users.

Our hardware consists of a 12-machine *cluster* formed by 3 high-end compute nodes and 8 storage nodes, plus 1 node that acts as a proxy. Machines are connected via 1Gbit switched network links. Compute nodes virtualize the Spark instance (*T2*), whereas storage nodes and the proxy run Swift and our IOStack prototype (SDS Controller and filter framework). We execute `ssbench` in other servers at URV, so *T1*'s `PUT` requests access our cluster from the Internet. Our cluster runs a complete OpenStack Kilo installation.

7.1 Storage Automation Under Multi-tenancy

Next, we reproduce a multi-tenant scenario inspired in Fig. 1 to assess the benefits of IOStack 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* in order to i) improve transfer performance and ii) minimize storage usage. Hence, scatter plots in Fig. 3 show the throughput of *T1*'s `PUT` requests (`ssbench`) and *T2*'s `GET` requests (Spark), for both Swift and IOStack.

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

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

Apart from transfer gains, IOStack also involves important storage space savings. To wit, *T1* stored along the experiment 312GB of data in Swift —considering 3-way replication, the actual amount of consumed storage is 936GB. Due to the high redundancy of data produced by `ssbench` [9], IOStack compressed *T1*'s data to 0.1% of its original size.

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

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

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

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

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

7.2 Dynamic Provisioning

Next, we examine the operation of dynamic storage policies in IOStack. That is, Fig. 4 shows `T1` performing `PUT` requests with increasing intensity. Then, we defined a dynamic policy that will enforce data compression on `T1`'s requests if it exhibits ≥ 3 `PUT` per second (see `P2` in Fig. 2).

[5]https://github.com/swiftstack/ssbench

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

8. RELATED WORKS

Software-Defined Storage. IOFlow[15] describes the first SDS architecture —decoupled control and data planes— that provides policy-based provisioning. Although an inspiring work, there are profound differences between IOFlow and IOStack. The most evident one is that IOFlow is designed for a particular file-system, whereas IOStack focuses on object storage. Moreover, IOFlow has a very specific scope; it provides low-level IO services (routing, classification) to control flows and guarantee IO bandwidth limits. In contrast, the 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, authors in [11] have recently proposed a system, called Retro, that controls and monitors resource usage in a distributed system (control plane). Retro also enforces bandwidth/latency policies to guarantee a certain SLA (data plane). Although Retro is not itself a complete SDS system, we believe that it is particularly interesting as a reference to build dynamic IO bandwidth differentiation in IOStack.

Active Storage. The early concept of *active disk* [13] —i.e., hard drives with computational capacity— has been borrowed by distributed file system designers in HPC environments (i.e., *active storage*) for reducing the amount of data movement between storage and compute nodes. Concretely, Piernas et al. [12] presented an active storage implementation integrated in the Lustre file system that provides flexible execution of code near to data in the user space. The industry has also done remarkable steps in this direction by implementing commercial distributed file systems with compute power such as PanFS [4] and PVFS [3]. Similarly, the filter framework of IOStack enables computations on data objects for policy enforcement. However, there are major differences between IOStack and previous works: i) These works do not focus on object storage, and ii) IOStack provides isolated/sandboxed code execution.

Perhaps, the closest technology to IOStack for leveraging active storage (IBM Storlets [1]) is ZeroCloud [5]. Both IBM Storlets and ZeroCloud rely on application containers —Docker and ZeroVM, respectively— for executing general purpose code on Swift objects. However, the usage of ZeroVM is more restrictive, as all code needs to be written in "C" and compiled via a proprietary tool-chain. Also, ZeroVMs can hold maximum of few tens of MBs of RAM.

9. CONCLUSIONS AND FUTURE WORK

We presented IOStack: the first Software-Defined Storage 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 towards improving the administration and operation of object stores.

Despite its potential, IOStack is only the first step of an ambitious project[6]. For object storage, we are currently working on the dynamic orchestration of filters in the IOStack filter framework, based on the resources that filters consume during their execution. We are also exploring ways of automatically detecting conflicting filters —or wrong filter ordering— enforced on the same tenant to simplify filter management. Furthermore, we are also developing a block storage version of IOStack for providing unified SDS management of both block and object storage.

10. 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).

11. REFERENCES

[1] IBM Storlets. https://github.com/openstack/storlets.
[2] Openstack swift. http://docs.openstack.org/developer/swift/.
[3] PVFS Project. http://www.pvfs.org/.
[4] The Panasas activescale file system (PanFS). http://www.panasas.com/products/panfs.
[5] ZeroCloud. http://www.zerovm.org/zerocloud.html.
[6] Big data needs software-defined storage. http://www.infoworld.com/article/2610828/infrastructure-storage/big-data-needs-software-defined-storage.html, 2014.
[7] A. Alba et al. Efficient and agile storage management in software defined environments. *IBM Journal of Research and Development*, 58(2/3):1–5, 2014.
[8] L. Atzori, A. Iera, and G. Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.
[9] 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*, pages 317–330, 2015.
[10] D. Logothetis, C. Trezzo, K. C. Webb, and K. Yocum. In-situ mapreduce for log processing. In *USENIX ATC'11*, page 115, 2011.
[11] J. Mace, P. Bodik, R. Fonseca, and M. Musuvathi. Retro: Targeted resource management in multi-tenant distributed systems. In *USENIX NSDI'15*, 2015.
[12] J. Piernas, J. Nieplocha, and E. J. Felix. Evaluation of active storage strategies for the lustre parallel file system. In *ACM/IEEE Supercomputing'07*, page 28, 2007.
[13] E. Riedel, G. Gibson, and C. Faloutsos. Active storage for large-scale data mining and multimedia applications. In *VLDB'98*, pages 62–73, 1998.
[14] R. Stutsman, C. Lee, and J. Ousterhout. Experience with rules-based programming for distributed,

[6]https://github.com/iostackproject

concurrent, fault-tolerant code. In *USENIX ATC'15*, pages 17–30, 2015.

[15] 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*, pages 182–196, 2013.