Crystal: Software-Defined Storage for Multi-tenant Object Stores

Raúl Gracia-Tinedo, Josep Sampé, Edgar Zamora, Marc Sánchez-Artigas, Pedro García-López
Universitat Rovira i Virgili (Tarragona, Spain)
{*raul.gracia,josep.sampe,edgar.zamora,marc.sanchez,pedro.garcia*}@*urv.cat*

Yosef Moatti, Eran Rom
IBM Research (Haifa, Israel)
{*moatti,eranr*}@*il.ibm.com*

Abstract

Object stores are becoming pervasive due to their scalability and simplicity. Their broad adoption, however, contrasts with their rigidity for handling heterogeneous workloads and applications with evolving requirements, which prevents the adaptation of the system to such varied needs. In this work, we present *Crystal*, the first Software-Defined Storage (SDS) architecture whose core objective is to efficiently support multi-tenancy in object stores. Crystal adds a filtering abstraction at the data plane and exposes it to the control plane to enable high-level policies at the tenant, container and object granularities. Crystal translates these policies into a set of distributed controllers that can orchestrate filters at the data plane based on real-time workload information. We demonstrate Crystal through two use cases on top of OpenStack Swift: One that proves its storage automation capabilities, and another that differentiates IO bandwidth in a multi-tenant scenario. We show that Crystal is an extensible platform to deploy new SDS services for object stores with small overhead.

1 Introduction

Object stores are becoming an increasingly pervasive storage building block due to their scalability, availability and usage simplicity via HTTP RESTful APIs [1, 8]. These desirable features have spurred the adoption of object stores by many heterogeneous applications and systems, ranging from Personal Clouds [4, 25], Big Data companies such as DataBricks [2] and Mirantis [6] and analytics frameworks [22, 17], and Web applications [19], to name a few.

Despite their growing popularity, object stores are not well prepared for heterogeneity. Typically, a deployment of an object store in the cloud uses a monolithic configuration setup, even when the same object store acts as a substrate for different types of applications with time-varying requirements [17, 16]. This results in all applications experiencing the same service level, though the workloads from different applications can vary dramatically. For example, while a social network application such as Facebook would have to store a large number of small-medium sized photos (KB- to MB-sized objects), a Big Data analytics framework would probably generate read and write requests for large files. It is clear that using a static configuration *inhibits optimization of the system* to such varying needs.

But not only this; beyond the particular needs of a type of workload, the requirements of applications can also vary greatly. For example, an archival application may require of transparent compression, annotation, and encryption of the archived data. In contrast, a Big Data analytics application may benefit from the computational resources of the object store to eliminate data movement and enable in-place analytics capabilities [22, 17]. Supporting such a variety of requirements in an object store is challenging, because in current systems, custom functionality is hard-coded into the system implementation due to the *absence of a true programmable layer*, making it difficult to maintain as the system evolves.

1.1 Scope and Challenges

In this paper, we argue that Software-Defined Storage (SDS) is a compelling solution to these problems. As in SDN, the separation of the “data plane” from the “control plane” is the best-known principle in SDS [41, 34, 39, 23, 38]. Such separation of concerns is the cornerstone of supporting heterogeneous applications in data centers. However, the application of SDS fundamentals on cloud object stores is not trivial. Among other things, it needs to address two main challenges:

A flexible control plane. The control plane should be the key enabler that makes it possible to support multiple applications separately using dynamically configurable functionalities. Since the *de facto* way of expressing management requirements and objectives in SDS is via policies, they should also dictate the management rules for the different tenants in a shared object store. This is not easy since policies can be very distinct. They can be as simple as a calculation on an object such as compression, and as complex as the distributed enforcement of per-tenant IO bandwidth limits. Further, as a singular attribution of object storage, such policies have to express objectives and management rules at the tenant, container and object granularities, which requires of a largely distinct form of policy translation into the data plane compared with prior work [41, 39, 38]. Identifying the necessary abstractions to concisely define the management policies is not enough. If the system evolves over time, the control plane should be flexible enough to properly describe the new application needs in the policies.

An extensible data plane. Although the controller in all SDS systems is assumed to be easy to extend [41, 39, 38], data plane extensibility must be significantly richer for object storage; for instance, it must enable to perform "on the fly" computations as the objects arrive and depart from the system to support application-specific functions like sanitization, Extract-Transform-Load (ETL) operations, caching, etc. This entails the implementation of a lightweight, yet versatile computing layer, which do not exist today in SDS systems. Building up an extensible data plane is challenging. On the one hand, it requires of new abstractions that enable policies to be succinctly expressed. On the other hand, these abstractions need to be flexible enough to handle heterogeneous requirements, that is, from resource management to simple automation, which is not trivial to realize.

1.2 Contributions

To overcome the rigidity of object stores we present *Crystal*: The first SDS architecture for object storage to efficiently support multi-tenancy and heterogeneous applications with evolving requirements. Crystal achieves this by separating policies from implementation and unifying an extensible data plane with a logically centralized controller. As a result, Crystal allows to dynamically adapt the system to the needs of specific applications, tenants and workloads.

Of Crystal, we highlight two aspects, though it has other assets. First, Crystal presents an extensible architecture that unifies individual models for each type of resource and transformation on data. For instance, global control on a resource such as IO bandwidth can be easily incorporated as a small piece of code. A dynamic management policy like this is materialized in form of a distributed, supervised *controller*, which is the Crystal abstraction that enables the addition of new control algorithms (Section 5.2). In particular, these controllers, which are deployable at runtime, can be fed with pluggable per-workflow or resource *metrics*. Examples of metrics are the number of IO operations per second and the bandwidth usage. An interesting property of Crystal is that it can even use object metadata to better drive the system towards the specified objectives.

Second, Crystal's data plane abstracts the complexity of individual models for resources and computations through the *filter* abstraction. A filter is a piece of programming logic that can be injected into the data plane to perform custom calculations on object requests. Crystal offers a filter framework that enables the deployment and execution of general computations on objects and groups of objects. For instance, it permits the pipelining of several actions on the same object(s) similar to stream processing frameworks [30]. Consequently, practitioners and systems developers only need to focus on the development of storage filters, as their deployment and execution is done transparently by the system (Section 5.1). To our knowledge, no previous SDS system offers such a computational layer to act on resources and data.

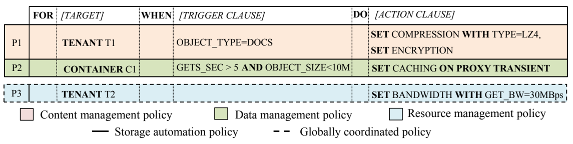
We evaluate the design principles of Crystal by implementing two use cases on top of OpenStack Swift: One that demonstrates the automation capabilities of Crystal, and another that enforces IO bandwidth limits in a multi-tenant scenario. These uses cases demonstrate the feasibility and extensibility of Crystal's design. The experiments with real workloads and benchmarks are run on a 13-machine cluster. Our experiments reveal that policies help to overcome the rigidity of object stores incurring small overhead. Also, defining the right policies may report performance and cost benefits to the system.

In summary, our key contributions are:

- Design of Crystal, the first SDS architecture for object storage that efficiently supports multi-tenancy and applications with evolving requirements;
- A control plane for multi-tenant object storage, with flexible policies and their transparent translation into the enforcement mechanisms at the data plane;
- An extensible data plane that offers a *filter* abstraction, which can encapsulate from arbitrary computations to resource management functionality, enabling concise policies for complex tasks;
- The implementation and deployment of policies for storage automation and IO bandwidth control that demonstrate the design principles of Crystal.

2 Crystal Design

Crystal seeks to efficiently handle workload heterogeneity and applications with evolving requirements in shared object storage. 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



	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** GET_BW=30MBps

🟥 Content management policy 🟩 Data management policy 🟦 Resource management policy
— Storage automation policy - - Globally coordinated policy

Figure 1: Structure of the Crystal DSL.

system itself. To do so, it uses three abstractions: *filter*, *metric*, and *controller*, in addition to *policies*.

2.1 Abstractions in Crystal

Filter. A filter is a piece of code that a system administrator can inject into the data plane to perform custom computations on incoming object requests[1]. In Crystal, this concept is broad enough to include *computations on object contents* (e.g., compression, encryption), *data management* like caching or pre-fetching, and even *resource management* such as bandwidth differentiation (Fig. 1). 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. This abstraction represents information accrued from the system to automate the execution of filters. There are two types of information sources. A first type that corresponds to the *real-time metrics* got from the running workloads, like the number of `GET` operations per second of a data container or the IO bandwidth allocated to a tenant. As with filters, a fundamental feature 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 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 *simple rule to automate* the execution of a filter, or a complex algorithm requiring *global visibility* of the cluster to control a filter's execution under multi-tenancy. 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. Our policies should be extensible for really allowing the system to satisfy evolving requirements. This means that the structure of policies must facilitate the incorporation of new filters, triggers and controllers.

To succinctly express policies, Crystal abides by a structure similar to that of the popular IFTTT (If-This-Then-That) service [5]. This service allows users to express small rule-based programs, called "recipes", using *triggers* and *actions*. For example:

[1]Filters work in an *online* fashion. To explore the feasibility of batch filters on already stored objects is matter of future work.

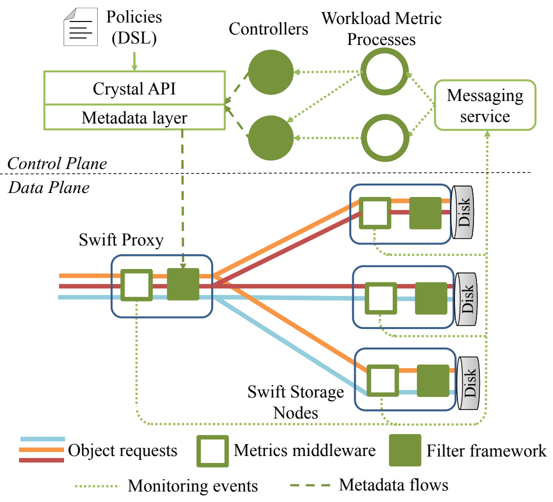


Figure 2: High-level overview of Crystal's architecture materialized on top of OpenStack Swift.

`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, we can infer that triggers and actions are translated into our inspection triggers and filters, respectively. At the control plane, a policy is a "recipe" that guides the behavior of control algorithms. Such apparently simple policy structure can express different policy types. On the one hand, Fig. 1 shows *storage automation policies* that enforce a filter either statically or dynamically based on simple rules; for instance, `P1` enforces compression and encryption on document objects of tenant `T1`, whereas `P2` applies data caching on small objects of container `C1` when the number of `GET`s/second is > 5. On the other hand, such policies can also express objectives to be achieved by controllers requiring *global visibility and coordination* capabilities of the data plane. That is, `P3` tells a controller to provide at least 30MBps of aggregated `GET` bandwidth to tenant `T2` under a multi-tenant workload.

2.2 System Architecture

Fig. 2 presents Crystal's architecture, which consists of:

Control Plane. Crystal provides administrators with a system-agnostic DSL (Domain-Specific Language) to define SDS services via high-level policies. The DSL "vocabulary" can be extended at runtime with new filters and inspection triggers. The control plane includes an API to compile policies and to manage the life-cycle and metadata of controllers, filters and metrics (see Table 1).

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. Other micro-services, called workload metric processes, close the control loop by exposing monitoring information from the data plane to controllers. The control loop is also extensible, given that both controllers and workload metric processes can be deployed at runtime.

Data Plane. Crystal's data plane has two core extension points: Inspection triggers and filters. First, a developer can deploy new workload metrics at the data plane to feed distributed controllers with new runtime information on the system. The metrics framework runs the code of metrics and publishes monitoring events to the messaging service. Second, data plane programmability and extensibility is delivered through the filter framework, which intercepts object flows in a transparent manner and runs computations on them. A developer integrating a new filter only needs to contribute the logic; the deployment and execution of the filter is managed by Crystal.

3 Control Plane

The control plane allows writing policies that adapt the data plane to manage multi-tenant workloads. It is formed by the DSL, the API and distributed controllers.

3.1 Crystal DSL

Crystal's DSL hides the complexity of low-level policy enforcement, thus achieving simplified storage administration (Fig. 1). 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.

Crystal Controller Calls	*Description*
`add_policy` `delete_policy` `list_policies`	Policy management API calls. For storage automation policies, the `add_policy` call can either to directly enforce the filter or to deploy a controller to do so. For globally coordinated policies, the call sets an objective at the metadata layer.
`register_keyword` `delete_keyword`	Calls that interact with Crystal registry to associate DSL keywords with filters, inspection triggers or coin new terms to be used as trigger conditions (e.g., `DOCS`).
`deploy_controller` `kill_controller`	These calls are used to manage the life-cycle of distributed controllers 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.
`update_slo` `list_slo` `delete_slo`	Calls to manage "tenant objectives" for coordinated resource management filters. For instance, bandwidth differentiation controllers take as input this information in order to provide an aggregated IO bandwidth share at the data plane.
Workload Metric Calls	*Description*
`deploy_metric` `delete_metric`	Calls for managing workload metrics at the data plane. These calls also manage workload metric processes to expose data plane metrics to the control plane.

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

Table 1: Main calls of Crystal controller, filter framework and workload metrics management APIs.

To cope with object stores formed by proxies/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 (by default), whereas actions to be executed only during the period where the condition is satisfied are transient (keyword `TRANSIENT`, `P2` in Fig. 1).

The vocabulary of our DSL can be extended on-the-fly to accommodate new filters and inspection triggers. That is, in Fig. 1 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 1).

The Crystal DSL has other 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 filters on a single request (e.g., compression + encryption) ordered as they are defined in the policy, similar to stream processing frameworks [30], and iii) *grouping*, which 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 1, Crystal offers a DSL compilation service via API calls. Crystal compiles simple automation policies as *target*$\rightarrow$*filter* relationships at the metadata layer. Next, we show how dynamic policies (i.e., with `WHEN` clause) use controllers to enforce filters.

3.2 Distributed Controllers

Crystal resorts to distributed controllers, in form of supervised micro-services, which can be deployed in the system at runtime to extend the control plane [15, 18, 40].

We offer two types of controllers: *automation* and *global* controllers. On the one hand, the Crystal DSL compiles dynamic storage automation policies into au-

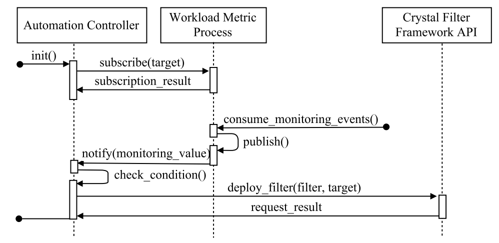


Figure 3: Interactions among automation controllers, workload metric processes and the filter framework.

tomation controllers (e.g., `P2` in Fig. 1). Their life-cycle consists of consuming the appropriate monitoring metrics and interact with the filter framework API to enforce a filter when the trigger clause is satisfied.

On the other hand, global controllers are not generated by the DSL; instead, by simply extending a base class and overriding its `computeAssignments` method, developers can deploy controllers that contain complex algorithms with global visibility and continuous control of a filter at the data plane (e.g., `P3` in Fig. 1). To this end, the base global controller class encapsulates the logic i) to ingest monitoring events, ii) to disseminate the computed assignments across nodes[2], and iii) to get Service-Level Objectives (SLO) to be enforced from the metadata layer (see Table 1). This allowed us to deploy distributed IO bandwidth control algorithms (Section 5).

Extensible control loop: To close the control loop, *workload metric processes* are micro-services that provide controllers with monitoring information from the data plane. While running, a workload metric process consumes and aggregates events from one workload metric at the data plane. For the sake of simplicity [40], we advocate to separate workload metrics not only per metric type, but also by target granularity.

Controllers and workload metrics processes interact in a publish/subscribe fashion [21]. For instance, Fig. 3 shows that, once initialized, an automation controller subscribes to the appropriate workload metric process, 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`; this ensures that controllers do not receive unnecessary monitoring information from other targets. Once the workload metric process receives the subscription request, it adds the controller to its observer list. Periodically, it notifies the activity of the different targets to the interested controllers that may trigger the execution of filters.

4 Data Plane

At the data plane, we offer two main extension hooks: Inspection triggers and a filter framework.

[2]For efficiency reasons, global controllers disseminate assignments to data plane filters also via the messaging service.

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 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. At the data plane, a workload metric is a piece of code that accounts for a particular aspect of the system operation and publishes that information. In our design, a new workload metric can inject events to the monitoring service without interfering with existing ones (Table 1). Our metrics framework allows developers to plug-in metrics that inspect both the type of requests and their contents (e.g., compressibility [29]). We provide the logic (i.e., `AbstractMetric` class) to abstract developers from the complexity of request interception and event publishing.

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 [36, 35] 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 [20]. Second, to deal with object stores composed by proxies/storage nodes, Crystal allows administrators to define the *execution point of a filter*.

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

Filter middleware: Our filter middleware intercepts data streams and classifies incoming requests. Upon a new object request, the middleware at the proxy performs a single metadata request to infer the filters to be executed on that request depending on the target. If the target has associated filters, the filter middleware sets the appropriate metadata headers in the request for triggering the execution of filters through the read/write path.

Filters that change the content of data objects may receive a special treatment (e.g., compression, encryption). 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.

That is, this yields that we may activate data compression on certain periods, but tenants will always download decompressed objects. To this end, prior to storing an object, we tag it with *extended metadata* that keeps track of the enforced filters with reverse flag set. Upon a `GET` request, the filter middleware fetches such metadata from the object itself to trigger the reverse transformations on it prior to the execution of regular filters.

Filter execution environments: Currently, our middleware can support two filter execution environments:

Isolated filter execution: Crystal provides an isolated filter execution environment to perform general-purpose computations on object streams with high security guarantees. To this end, we extended the Storlets framework [7] with pipelining and stage execution control functionalities. Storlets provide Swift with the capability to run computations close to the data in a secure and isolated manner making use of `Docker` containers [3]. 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. Also, a `Docker` container only runs 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 deploy code modules as native filters at runtime by following simple implementation 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.

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.

5.1 New Storage Automation 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 ON STORAGE_NODE
```

```
P2:FOR CONTAINER C1 WHEN GETS_SEC > 5 DO SET CACHING
```

Data plane (Filters): To enable such storage automation policies, we first need to *develop the filters* at the data plane. In Crystal this can be done using either 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): Via the Crystal API (see Table 1), we deployed metrics that capture various workload aspects (e.g., `PUTs`/`GETs` per second of a tenant) to satisfy policies like `P2`. Similarly, we deployed the corresponding 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 enforcing 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 registered 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.

5.2 Global Management of IO Bandwidth

Goal: To provide Crystal with means of defining policies that enforce a global IO bandwidth SLO on `GETs`/`PUTs`:

```
P3:FOR TENANT T1 DO SET BANDWIDTH WITH GET_BW=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 to the background tasks (e.g., replication) executed by storage nodes, which impact on performance [33]. 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,

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:      /* Compute assignments on current tenant transfers to meet SLOs */
5:      SLOAssignments ← minSLO(info, SLOs);
6:      /* Estimate spare bw at proxies/storage nodes based on current usage */
7:      spareBw ← min(spareBwProxies(SLOAssignments), spareBwStorageNodes(SLOAssignments));
8:      spareBwSLOs ← {};
9:      /* Distribute spare bandwidth equally across all tenants */
10:     for tenant in info do
11:         spareBwSLOs[tenant] ← $\frac{spareBW}{numTenants(info)}$;
12:     end for
13:     /* Calculate assignments to achieve spare bw shares for tenants */
14:     spareAssignments ← spareSLO(SLOAssignments, spareBwSLOs);
15:     /* Combine SLO and spare bw assignments on tenants */
16:     return SLOAssignments ∪ spareAssignments;
17: end function
```

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 for each type of traffic if needed. In essence, monitoring events contain a data structure that represents the bandwidth share that tenants exhibit at proxies or per storage node disk. We also deployed workload metric processes to expose these events to controllers.

Control plane. We deployed Algorithm 1 as a global controller to orchestrate our bandwidth differentiation filter. Concretely, we aim at satisfying three main requirements: i) A *minimum bandwidth share per tenant*, ii) *Work-conservation* (do not leave idle resources), and iii) *Equal shares 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 [28, 44].

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 tenants with guaranteed bandwidth. Note that such assignment is done in descending order based on the number of parallel transfers per tenant, provided that tenants with fewer transfers have fewer opportunities of meeting their SLOs. Moreover, `minSLO` checks whether there exist overloaded 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 function `spareSLO` to calculate the spare bandwidth assignments (requirement 3, line 15). Note that `spareSLO` receives the `SLOAssignments` 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 deployed in Crystal [27], our goal in Algorithm 1 is to offer an attractive simplicity/effectiveness trade-off, validating our bandwidth differentiation framework.

6 Prototype Implementation

We tested our prototype in OpenStack Kilo version. The Crystal API is implemented with the Django framework. The API manages the system's metadata from Redis 3.0 in-memory store [10]. We found that co-locating both Redis and the Swift proxies in the same servers is a suitable deployment strategy. As we show next, this is specially true as only the filter middleware in proxies accesses the metadata layer (once per request).

We resort to PyActive [46] for building distributed controllers and workload metric processes that can communicate among them (e.g., TCP, message brokers). For fault tolerance, the PyActive supervisor is aware of all the instantiated remote micro-services (either at one or many servers) and can spawn a new process if one dies.

We built our metrics and filter frameworks as standard WSGI middlewares in Swift. The code of workload metrics is dynamically deployed on Swift nodes, intercepts the requests and periodically publishes monitoring information (e.g., 1 second) via RabbitMQ 3.6 message broker. Similarly, the filter framework middleware intercepts a storage request and redirects it via a pipe either to the Storlets engine or to a native filter, depending on the filter pipeline definition. As both filters and metrics can run on all Swift nodes, in the case of server failures they can be executed in other servers holding object replicas.

The code of Crystal is publicly available[3] and our contributions to the Storlets project are submitted for acceptance to the official OpenStack repository.

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

7 Evaluation

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

Objectives: Our evaluation addresses the challenges of Section 1.1 by showing: 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` [11] 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".

To evaluate Crystal under real-world object storage workloads, we collected the following traces[4]: ii) The first trace captures 1.28TB of a write-dominated (79.99% write bytes) document database workload storing 817K car testing/standardization files (mean object size is 0.91MB) for 2.6 years at Idiada; an automotive company. i) The second trace captures 2.97TB of a read-dominated (99.97% read bytes) Web workload consisting of requests related to 228K small data objects (mean object size is 0.28MB) from several Web pages hosted at Arctur datacenter for 1 month. We developed our own workload generator to replay a part of these traces (12 hours), as well as to perform experiments with controllable rates of requests. Our workload generator resorts to SDGen [24] to create realistic contents for data objects based on the file types described in the workload traces.

Platform: We ran our experiments in a 13-machine cluster 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 were 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., API, controllers, messaging, metadata store). Nodes in the cluster are connected via 1 GbE switched links.

7.1 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 (Idiada, Arctur). These

[4]Available at `http://iostack.eu/datasets-menu`.

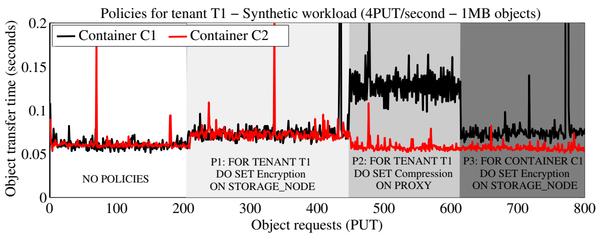


Figure 4: Enforcement of compression/encryption filters.

experiments have been executed at the compute nodes against 1 swift proxy and 6 storage nodes.

Storage management capabilities of Crystal. Fig. 4 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 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. 4 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 [7] (Storlet filter). Second, the administrator has 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 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). To overcome this issue, the administrator defined a new policy `P3` that essentially enforces only encryption on `C1`'s requests. After 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 policy specialization; policies `P1` and `P2`

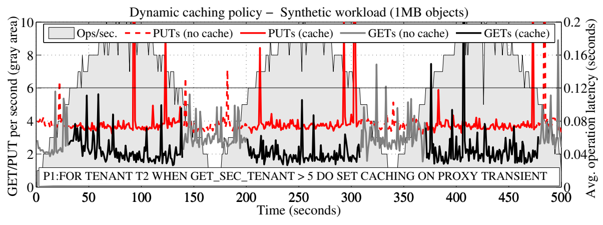


Figure 5: Dynamic enforcement of caching filter.

apply to `C2` at the tenant scope, whereas the system only executes `P3` on `C1`'s requests, as it is the most specialized policy.

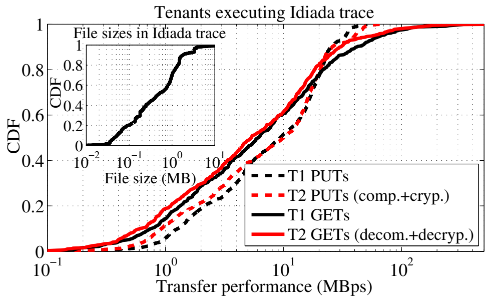
Dynamic storage automation. Fig. 5 shows a dynamic caching policy (`P1`) on one tenant. The filter implements LRU eviction and exploits SSD drives at the proxy to improve object retrievals. We executed a synthetic oscillatory workload of 1MB objects (gray area) to verify the correct operation of automation controllers.

In Fig. 5, 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 non-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. A reason for this may be that storage nodes are mostly free to execute writes, as a large fraction of `GET`s are being served at the proxy's cache.

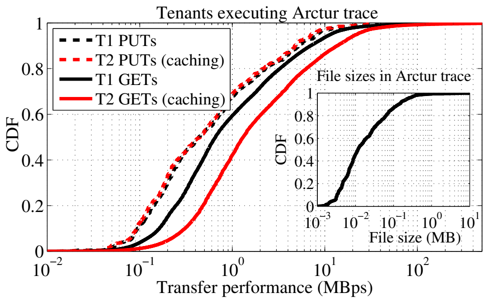
In conclusion, Crystal's control loop enables dynamic enforcement of storage filters under variable workloads. Moreover, native filters in Crystal allow developers to build complex data management filters.

Managing real workloads. Next, we show how Crystal policies can handle real workloads (12 hours). That is, we compress and encrypt documents (`P1` in Fig. 1) on a replay of the Idiada trace (write-dominated), whereas we enforce caching of small files (`P2` in Fig. 1) on a replay of Arctur workload (read-dominated).

Fig. 6(a) 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 benefited from



(a) Idiada workload (file sizes in inner plot).



(b) Arctur workload (file sizes in inner plot).

Figure 6: Policy enforcement on real trace replays.

higher data confidentially thanks to encryption.

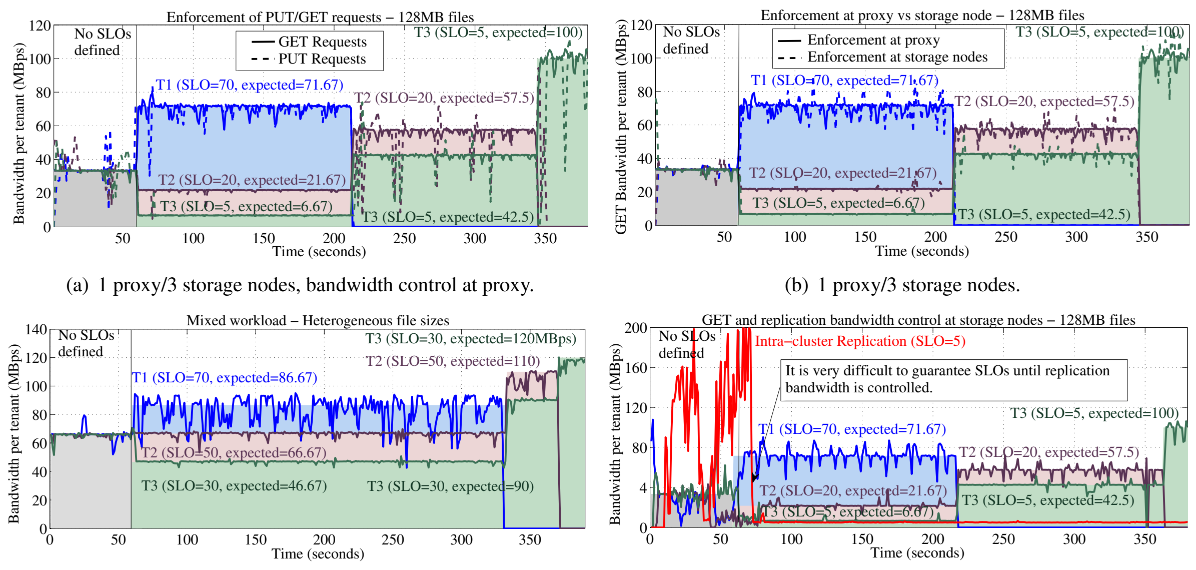
Fig. 6(b) shows tenants `T1` and `T2`, both concurrently running a trace replay of Arctur. By executing a dynamic caching policy, `T2`'s `GET` requests are in median 1.9x faster compared to `T1`. That is, as the workload of Arctur is intense and almost read-only, caching was enabled for tenant `T2` for most of the experiment. Moreover, because the requested files fitted in the cache, the SSD-based caching filter was very beneficial to tenant `T2`. The median write overhead of `T2` compared to `T1` was 4.2%, which suggests that our filter efficiently intercepts object streams for doing parallel writes at the SSD.

Our results with real workloads suggest that Crystal is practical for managing multi-tenant object stores.

7.2 Achieving Bandwidth SLOs

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 (in Fig. 7(d) we use 3 replicas), the rest of experiments were performed using one replica rings.

Request types. Fig. 7(a) 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



(a) 1 proxy/3 storage nodes, bandwidth control at proxy.

(b) 1 proxy/3 storage nodes.

(c) 2 proxy/6 storage nodes, bandwidth control at proxies.

(d) 1 proxy/3 storage nodes, bandwidth control at storage nodes.

Figure 7: Performance of the Crystal bandwidth differentiation service (SLOs per tenant are in MBps).

SLO plus an equal share of spare bandwidth, according to the expected policy behavior defined by colored areas. This demonstrates the effectiveness of our bandwidth control middleware for intercepting and limiting both requests types. We also observe in Fig. 7(a) that PUT bandwidth exhibits higher variability than GET bandwidth. Concretely, after writing 512MB of data, Swift stopped the transfers of tenants for a short interval; we will look for the causes of this in our next development steps.

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. 7(b) shows the enforcement SLOs on GET requests at both stages. At first glance, we observe in Fig. 7(b) that our framework makes it possible to enforce bandwidth limits at both stages. However, Fig. 7(b) 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 the workload at hand. That is, given the same workload intensity, a fewer number of 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 have to 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 a trade-off between accuracy and control that may be solved with hybrid enforcement schemes.

Mixed tenant activity, variable 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. 7(c) shows that our enforcement controller achieves bandwidth SLOs under mixed workloads. Moreover, the bandwidth differentiation framework works properly when doubling the storage cluster size, as the policy provides tenants with the desired SLO plus a fair share of spare bandwidth, specially for T1 and T2. However, Fig. 7(c) 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. 7(d) illustrates the impact of replication tasks on multi-tenant workloads. In Fig. 7(d), we observe that during the first 60 seconds of this 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.

7.3 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*[5] and iii) *managing extended object metadata*. We show this in Fig. 8.

Compared to vanilla Swift (`SW`), Fig. 8 shows that the metadata access of Crystal incurs a median latency penalty between 1.5ms and 3ms (`MA` boxplots). 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. 8 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 4). Fig. 8 illustrates that the median isolated interception overhead of a void filter (`NOOP`) oscillates between 3ms and 11ms (e.g., 5.7% and 15.7% median latency penalty for 10MB and 1MB `PUT`s, respectively). This cost mainly comes from injecting the data stream into a `Docker` container to achieve isolation. We also may consider filter implementation effects, or even the data at hand. To wit, 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 [24]. Conversely, decompression in

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

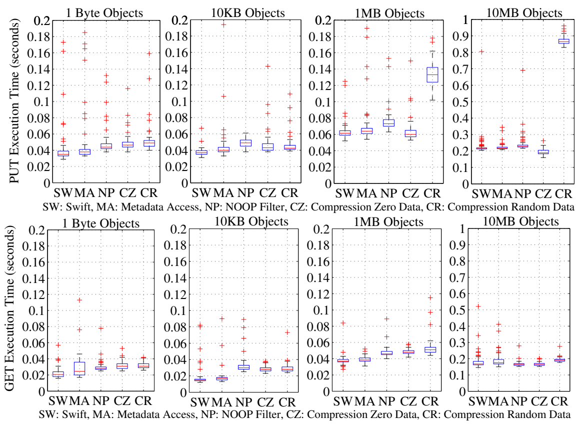


Figure 8: Performance overhead of filter framework metadata interactions and isolated filter enforcement.

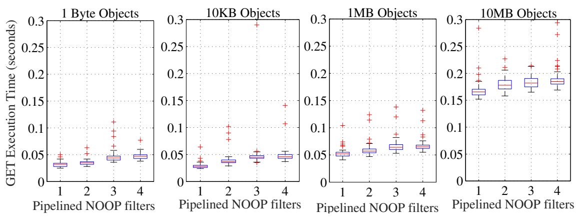


Figure 9: Pipelining performance for isolated filters.

`GET` requests is not significantly affected by data contents. Hence, to improve performance, filters should be 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 4). We measured that reading/writing extended object 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. 9 depicts the latency overhead of pipeling multiple `NOOP` Storlet filters. As pipelining is a new feature of Crystal, it required a separate evaluation.

Fig. 9 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 lower 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. To understand the monitoring costs of Crystal, we provide a measurement-based estimation of various configurations of monitoring nodes, workload metrics and controllers. To wit, the monitoring traffic overhead O related to $|\mathscr{W}|$ workload metrics is

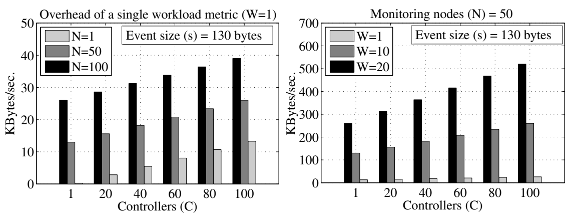


Figure 10: Traffic overhead of Crystal depending on the number of nodes, controllers and workload metrics.

produced by a set of nodes $\mathscr{N}$. Each node in $\mathscr{N}$ periodically sends monitoring events of size s to the MOM broker, which are consumed by $|\mathscr{W}|$ workload metric processes. Then, each workload metric process aggregates the messages of all nodes in $\mathscr{N}$ into a single monitoring message. The aggregated message is then published to a set of subscribed controllers $\mathscr{C}$. Therefore, we can do a worst case estimation of the total generated traffic per monitoring epoch (e.g., 1 second) as: $O = |\mathscr{W}| \cdot [s \cdot (2 \cdot |\mathscr{N}| + |\mathscr{C}|)]$. We also measured simple events (e.g., `PUT_SEC`) to be $s = 130$ bytes in size.

Fig. 10 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 $|\mathscr{C}| = 100$ subscribed controllers. Clearly, the dominant factor of traffic generation is the number of workload metrics. However, even for a large number of workload metrics ($|\mathscr{W}| = 20$), the monitoring requirements in a 50-machine cluster do not exceed 520KBps. These overheads seem lower than existing SDS systems with advanced monitoring [33].

8 Related Work

SDS Systems. IOFlow [41], now extended as sRoute [38], 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 [38, 39].

Crystal, however, targets a different scenario. Simply put, it pursues the configuration and optimization of object stores to the evolving needs of tenants/applications, for it needs a richer data plane and a different suite of management abstractions and enforcement mechanisms. For example, tenants require mechanisms to inject custom logic to specify not only system activities but also application-specific transformations on objects.

Retro [33] 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 needed to 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.

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 [26, 27, 43, 45, 32, 41, 33]. For instance, mClock [27] achieves IO resource allocation for multiple VMs at the hypervisor level, even in distributed storage environments (dmClock). However, object stores have received much less attention in this regard; vanilla Swift only provides a non-automated mechanism for limiting the "number of requests" [12] per tenant, instead of IO bandwidth. In fact, this problem resembles the one stated by Wang et al. [44] 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. [45] 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 [37, 28].

Active storage. The early concept of *active disk* [36, 14, 31, 42], i.e., a HDD with computational capacity, was borrowed by distributed file system designers in HPC environments two decades ago to give birth to active storage. The goal was to diminish the amount of data movement between storage and compute nodes [13, 9]. Piernas et al. [35] 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 way to inject custom logic into the data plane and manage it via policies. This requires filters to be deployable at runtime, support sandbox execution [7], and be part of complex workflows.

9 Conclusions

Crystal is a SDS architecture that pursues an efficient use of multi-tenant object stores. 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. Our results show that Crystal is practical enough to be run in a shared cloud object store.

Acknowledgments

We thank our shepherd Ajay Gulati and the anonymous reviewers. This work has been partly funded by the EU project H2020 "IOStack: Software-Defined Storage for Big Data" (644182) and Spanish research project "Cloud Services and Community Clouds" (TIN2013-47245-C2-2-R) funded by the Ministry of Science and Innovation.

References

[1] Amazon s3. https://aws.amazon.com/en/s3.

[2] Databricks. https://databricks.com.

[3] Docker. https://www.docker.com.

[4] Dropbox. https://www.dropbox.com.

[5] Ifttt. https://ifttt.com.

[6] Mirantis. https://www.mirantis.com.

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

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

[9] PVFS Project. http://www.pvfs.org/.

[10] Redis. https://www.redis.io.

[11] Ssbench. https://github.com/swiftstack/ssbench.

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

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

[14] A. Acharya, M. Uysal, and J. Saltz. Active disks: Programming model, algorithms and evaluation. *ACM SIGPLAN Notices*, 33(11):81–91, 1998.

[15] G. A. Agha. Actors: A model of concurrent computation in distributed systems. Technical report, The MIT Press, 1985.

[16] A. Anwar, Y. Cheng, A. Gupta, and A. R. Butt. Taming the cloud object storage with mos. In *Proceedings of the 10th Parallel Data Storage Workshop*, pages 7–12, 2015.

[17] A. Anwar, Y. Cheng, A. Gupta, and A. R. Butt. Mos: Workload-aware elasticity for cloud object stores. In *ACM HPDC'16*, pages 177–188, 2016.

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

[19] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a needle in haystack: Facebook's photo storage. In *USENIX OSDI'10*, pages 1–8, 2010.

[20] G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, 44(3):15, 2012.

[21] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131, 2003.

[22] M. Factor, G. Vernik, and R. Xin. The perfect match: Apache spark meets swift. https://www.openstack.org/summit/openstack-paris-summit-2014/session-videos/presentation/the-perfect-match-apache-spark-meets-swift, November 2014.

[23] R. Gracia-Tinedo, P. García-López, M. Sanchez-Artigas, J. Sampé, Y. Moatti, E. Rom, D. Naor, R. Nou, T. Cortes, and W. Oppermann. IOStack: Software-defined object storage. *IEEE Internet Computing*, 2016.

[24] 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.

[25] R. Gracia-Tinedo, Y. Tian, J. Sampé, H. Harkous, J. Lenton, P. García-López, M. Sánchez-Artigas, and M. Vukolic. Dissecting ubuntuone: Autopsy of a global-scale personal cloud back-end. In *ACM IMC'15*, pages 155–168, 2015.

[26] A. Gulati, I. Ahmad, C. A. Waldspurger, et al. Parda: Proportional allocation of resources for distributed storage access. In *USENIX FAST'09*, pages 85–98, 2009.

[27] A. Gulati, A. Merchant, and P. J. Varman. mclock: handling throughput variability for hypervisor io scheduling. In *USENIX OSDI'10*, pages 1–7, 2010.

[28] A. Gulati and P. Varman. Lexicographic qos scheduling for parallel i/o. In *ACM SPAA'05*, pages 29–38, 2005.

[29] D. Harnik, R. Kat, D. Sotnikov, A. Traeger, and O. Margalit. To zip or not to zip: Effective resource usage for real-time compression. In *USENIX FAST'13*, pages 229–241, 2013.

[30] 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*, pages 140–140, 2006.

[31] K. Keeton, D. A. Patterson, and J. M. Hellerstein. A case for intelligent disks (idisks). *ACM SIGMOD Record*, 27(3):42–52, 1998.

[32] 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*, page 28, 2016.

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

[34] M. Murugan, K. Kant, A. Raghavan, and D. H. Du. Flexstore: A software defined, energy adaptive distributed storage framework. In *IEEE MASCOTS'14*, pages 81–90, 2014.

[35] 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.

[36] E. Riedel, G. Gibson, and C. Faloutsos. Active storage for large-scale data mining and multimedia applications. In *VLDB'98*, pages 62–73, 1998.

[37] D. Shue, M. J. Freedman, and A. Shaikh. Fairness and isolation in multi-tenant storage as optimization decomposition. *ACM SIGOPS Operating Systems Review*, 47(1):16–21, 2013.

[38] I. Stefanovici, B. Schroeder, G. O'Shea, and E. Thereska. sRoute: treating the storage stack like a network. In *USENIX FAST'16*, pages 197–212, 2016.

[39] 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*, pages 174–181, 2015.

[40] R. Stutsman, C. Lee, and J. Ousterhout. Experience with rules-based programming for distributed, concurrent, fault-tolerant code. In *USENIX ATC'15*, pages 17–30, 2015.

[41] 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.

[42] M. Uysal, A. Acharya, and J. Saltz. Evaluation of active disks for decision support databases. In *IEEE HPCA'00*, pages 337–348, 2000.

[43] A. Wang, S. Venkataraman, S. Alspaugh, R. Katz, and I. Stoica. Cake: enabling high-level slos on shared storage systems. In *ACM SoCC'12*, page 14, 2012.

[44] Y. Wang and A. Merchant. Proportional-share scheduling for distributed storage systems. In *USENIX FAST'07*, 2007.

[45] J. C. Wu and S. A. Brandt. Providing quality of service support in object-based file system. In *IEEE MSST'07*, volume 7, pages 157–170, 2007.

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