Vertigo: Programmable Micro-controllers for Software-Defined Object Storage

Josep Sampé, Pedro García-López, Marc Sánchez-Artigas
Department of Computer Engineering and Mathematics
Universitat Rovira i Virgili
Tarragona, Spain
E-mail: {josep.sampe,pedro.garcia,marc.sanchez}@urv.cat

***Abstract*—Software-defined storage (SDS) aims to minimize the complexity of data management in the Cloud. SDS decouples the control plane from the data plane and simplifies the management of the storage system via automated storage policy enforcement.**

In this paper, we propose a novel SDS framework for Object Storage that allows to decentralize policy enforcement through the deployment of per-object management policies in the storage nodes. As in active storage systems, we leverage the underutilized CPU time in the storage nodes. But our framework goes one step further. It provides a new management abstraction called *micro-controllers* which operate on objects depending on their *state* and content, thereby permitting the implementation of sophisticated management policies, such as the automated deletion of an object based on its access history, and even allowing the orchestration of active storage tasks.

Our SDS system avoids the massive interception of data flows by moving that logic to the appropriate objects. Furthermore, our extensible model simplifies the customization of Object Storage services. We present in the validation several interesting use cases such as automated deletion, content level access control, and Web prefetching.

***Index Terms*—Object Storage; Software-defined Storage; OpenStack Swift; Active Storage**

I. Introduction

Object Storage systems like Amazon S3 or OpenStack Swift have gained enormous popularity in Cloud settings thanks to their scalable architecture and their flexibility to deal with unstructured data. An object usually includes the data itself, metadata, and a globally unique identifier. Object Storage provides two major advantages over other storage architectures like file or block storage systems. On the one hand, scalability and elasticity are ensured thanks to a flat namespace that can span multiple nodes. On the other hand, Object Storage offers more flexibility by providing interfaces (APIs) and custom metadata that can be directly accessed by third party applications.

Previous approaches like *active storage* [1]–[3] have tried to increase the flexibility and programmability of Object Storage. In this line, active storage allows computation close to the data, leveraging storage nodes resources. For instance, the IBM Storlet Engine [4] for OpenStack Swift enables the execution of active tasks like compression when an object is accessed in the Object Store. But active storage tasks are usually stateless tasks that must be instrumented in a per-object basis. Thus, they cannot make management decisions based on the *state* of an object such as its access history or on the content itself.

To increase the automation, flexibility, and programmability of Object Storage we propose here Vertigo, a novel *Software-defined Storage* (SDS) architecture that leverages previous work on active storage. As in *Software-defined Networking* (SDN), our framework also separates the control plane from the data plane to simplify policy enforcement and management of data storage services.

Our major difference is that the control plane and the data planes are co-located in the same storage nodes. In our case, the control plane is a meta layer that intercepts and modifies the behavior of the data plane.

The novelty of our approach is the decentralization of policy enforcement thanks to the deployment of management policies (micro-controllers) to the storage nodes. As in active storage systems, we leverage the computing resources of storage nodes to deploy micro-controllers that intercept the object life cycle. The distinguishing feature of micro-controllers is that they can react to the changes made on the *state* of an object, permitting the implementation of sophisticated management policies, like the automated deletion of an object based on its access history. In addition, micro-controllers can be used to orchestrate active storage tasks, which is not possible to do in frameworks like the IBM Storlet Engine [4], and even to manipulate objects in the data plane based on their content.

Object stores operate at the object level, mainly acting as simple repositories of data. One of the outstanding features of Vertigo is that we can work at the *content level*, which is essential to allow an object to adapt its behavior depending on who is accessing it, its state and the nature of its content. The result is an unprecedented amount of flexibility for automation and storage programmability.

Furthermore, our framework avoids massive interception of data flows in SDS by moving that logic to the appropriate objects. In the validation, we present interesting applications such as *automated deletion* and *control level access control*.

The remainder of this paper is organized as follows. In section II, we summarize the related work of SDS solutions and discuss their advantages and shortcomings. In section III, we describe the architecture of Vertigo. In section IV, we show the implementation of Vertigo on a real system based on OpenStack Swift. In section V, we present some applications that Vertigo supports. Finally, in section VI we evaluate these applications.

II. Related work

The flexibility and programmability of Object Storage fits nicely with software-defined storage solutions that automate provisioning and management of the storage service. In this work, however, we focus on those SDS solutions related to the life cicle, and the management of the objects. This is the case of the EMC Atmos [5], in which we can apply user-defined policies to groups of objects to determine, for instance, object layout, replication levels, and replica placement.

In Amazon S3[1], on the other hand, we can define life cycle configuration rules that intercept calls to objects in the system. Such configuration rules may simplify the life cycle management of objects. One example is automated transition of less-frequently accessed objects to low-cost storage alternatives, which changes replication level and replica placement of the objects.

Furthermore, Amazon has another service called Amazon lambda[2]. In Lambda, Amazon users can upload their own programs, called lambda functions, which will be triggered when certain events occur in some of their services. For instance, in Amazon Simple Storage Service, a user can trigger lambda functions when an object is uploaded or deleted from a container. With this service, the users can develop a wide variety of different lambda functions, for example, functions for live data processing, which generate metrics or filter registers, and functions for extraction, transformation and load (ETL) capabilities, as well as for other purposes like video transcoding, file indexing and content validation. This approach extends the functionality of the storage service, and the actual computation is done in a separate cluster. Amazon manages and scales out the resources as needed for each lambda function. Unlike Amazon Lambda, we advocate to leverage storage node resources to run computation tasks close to the data thanks to Active Storage techniques.

Another related system is Comet [6]. Comet is a distributed key-value storage system which stores active storage objects (ASOs). An ASO consists of a key, a value, and a set of handlers which are triggered as a result of timers or storage operations, such as put or get. The main difference with Comet is that in our approach we differentiate between control and active storage tasks. In our approach, a micro-controller may call active storage tasks over the data or not. For example, compressing an object to some users. In this case, the micro-controller will call the active storage program to compress the file when a specific user accesses the object, and will do nothing when other users access the object.

IOFlow [7] is a SDS architecture designed for a distributed file system. It uses a logical centralized control plane to enable high-level control policies like bandwidth differentiation, caching and data sanitization. Even if we share control plane and data plane abstractions, the programmability, granularity, and flexibility of our approach is completely different. Whereas IOFlow intercepts data flows, our framework works at the object level.

In an earlier work [1], method objects and policy objects were introduced to extend the functionality of the object storage devices. A method object is a special object that can be executed in object storage devices to perform operations on certain user objects. Policy objects, moreover, are the set of conditions that can be evaluated to decide whether or not execute a method object. In this work, they introduce an hybrid approach using both request-driven model (explicit method invocation in request) and policy-driven model (method execution when the conditions defined by policies are satisfied) to put method objects into execution. This model is limited to only one method object associated to a policy object, and the policy is only one condition that may be satisfied or not. In contrast, one functionality of Vertigo is that micro-controller objects may be policy objects, but with more than one condition which, therefore, can put into execution different method objects depending on which condition is satisfied. In our case, we can even put into execution more than one method object with a pipeline of methods.

In the context of file systems, the storage policy decisions are made for an entire file system. This granularity is too large and can sacrifice storage efficiency and performance, particularly since different files have different requirements. That is why in [8]–[10], they have created a framework that allows policy decisions to be made at the file granularity in different types of file systems. They have created a storage stack through which they pass all files before storing them in the disk. This stack is extensible by adding plugins, and a system administrator can set a flow of plugins along the storage stack. This plugins are enabled and disabled through the use of object attributes, which can be set by the user or by means of policies.

III. Architecture

Like other SDS systems [7], [11]–[13] our architecture includes a control plane and a data plane. Our major difference is that the control plane and the data planes are co-located in the same storage nodes. In our specific case, the control plane is a meta layer that intercepts and modifies the behavior of the data plane.

The entire architectural model revolves around the concept of *micro-controller*, and therefore, it is critical to define what a micro-controller is. As a working definition, a micro-controller is a data behavior or management policy that is associated in the deployment phase to one or more object(s) and to one or more storage operation(s) (`get`, `put`, `delete` and `update`). Compared with conventional active storage, micro-controllers go beyond simple active storage tasks for two main reasons:

1) Unlike active storage tasks, micro-controllers can persist their (object) *state* and use it afterward; and
2) Active storage tasks usually modify the current input or output data flow of one object, but micro-controllers can modify one or more objects and even instrument one or more active storage tasks.

The complete architecture is depicted in Fig. 1. As expected, the data plane includes the traditional Object Storage system which manages objects and their associated metadata. The control plane includes two major components: the *Controller Runtime* and the *Internal Client*. The *Controller Runtime* is the entity that intercepts all requests of the object life cycle and runs interceptors (micro-controllers and active storage tasks) in a safe and sandboxed environment. The *Internal Client* component enables the communication of micro-controllers with the objects in the data plane. The *Internal Client* also allows execution of interceptors at specific times without the need for external client requests.

To better understand the role of these components, consider a micro-controller that counts the number of read requests to one object in the past x days. To deploy it, the code of this micro-controller must be first installed in the *Controller Runtime*, and then tell the *Controller Runtime* that an `onGet` trigger is associated with this registered micro-controller for the targeted object (`KEY=value`) or objects. From that moment, every `get` request to this object will be intercepted and routed to this micro-controller.

[1]http://docs.aws.amazon.com/AmazonS3/latest/dev/
[2]http://docs.aws.amazon.com/lambda/latest/dg/

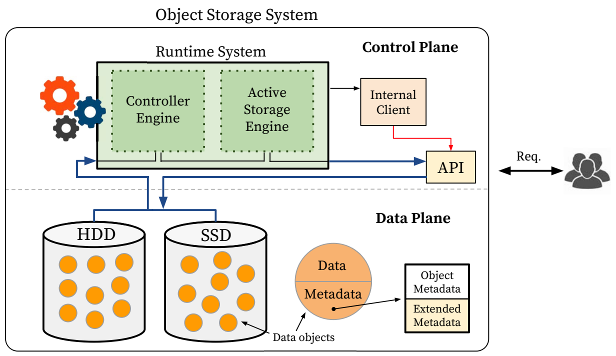


Fig. 1. High level architecture overview

A more advanced example could be a micro-controller that performs actions on the data plane when some condition is met. For example, the previous micro-controller could delete the object if there are no requests in the past 10 days. In this case, the micro-controller would then instrument the *Internal Client* component to perform such delete action on the data plane.

A. Controller Runtime

The *Controller Runtime* contains two major components for managing the different types of interceptors: the *Controller Engine* and the *Active Storage Engine*. Both components work on an isolated and sandboxed environment (Linux container) that intercepts calls to the object storage.

Active Storage Engine. It takes care of the management of active storage tasks. These tasks are stateless components that process the input and output data flows from and to an object. Typical active storage tasks like compression are used as data reduction techniques in Object stores. It is also responsible for the installation, deployment and configuration of active storage tasks in the control plane:

- *Installation* means uploading and installing the task code in the framework.
- *Deployment* refers to linking an installed task to one or more objects with one or more triggers. Triggers include the operations that may be intercepted in the selected object or objects (`get`, `put`, `delete`, `update`).
- *Configuration* refers to setting up the parameters and the necessary metadata of an already installed task, e.g., the compression ratio.

In Vertigo, active storage tasks are normally instrumented and executed from micro-controllers located in the *Controller Engine*.

Controller Engine. It takes care of the management of micro-controllers. It is responsible of the installation, deployment and configuration of micro-controllers in the control plane. These three tasks are equivalent to the ones explained above.

As explained before, once deployed, micro-controllers are triggered in reaction to life cycle events of objects in the data plane. Micro-controllers can store information in a *persistent* way (e.g., an access counter, the timestamp of the last access, the role of a user, etc.) by transparently benefiting from object metadata.

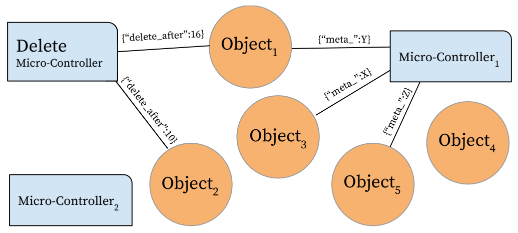


Fig. 2. Micro-controller-Object metadata

A micro-controller can take one or all of the following actions after intercepting a life cycle request on an object:

- It can update metadata (e.g., an access counter);
- It can execute one or orchestrate multiple *active storage* tasks (e.g., compression, encryption, etc.); and
- It can generate new requests to the object storage (e.g., to delete some object).

Fig. 2 illustrates the deployment of a micro-controller that deletes an object after x days. The idea is to develop a generic delete micro-controller and associate it with different objects. What changes is the metadata associated with the target object and the delete micro-controller: {"`delete_after`":16} for Object$_1$, and {"`delete_after`":10} for Object$_2$. Thanks to this declarative approach, it is then possible to reuse the same micro-controller for different objects.

A more sophisticated example may be a micro-controller for instrumenting an active storage task. For example, we could enforce the compression of an object if it is not very popular (few accesses in the last days) and the content type is text.

B. Object Storage Internal Client

The *Internal Client* offers two main functionalities to the *Controller Runtime*:

- *Scheduling periodic execution of micro-controllers*: Using a daemon service like `cron`, it is possible to create a new `onTimer` trigger and schedule the execution of specific micro-controllers. In this case, the micro-controller is not tied to the life cycle of the object, so that batch or periodic tasks can be orchestrated very easily.
- *Execution of object requests*: A simple API enables the micro-controllers to communicate with the object storage and execute `get`, `put`, `post` and `delete` requests on specific objects.

IV. Implementation

We have implemented a prototype of our SDS framework[3] on top of the OpenStack Swift[4] system and modified the open source IBM Storlet framework[5] to suit our requirements.

OpenStack Swift is a highly scalable Object Storage system that can store a large amount of data through a RESTful HTTP API similar to that of Amazon S3. The access path to an object consists of exactly three elements: */account/container/object*. The object is the exact data input by the user, while accounts and containers provide a way of grouping objects. Nesting of accounts and containers is not supported. Swift architecture is split into several components, which include account servers,

[3]https://github.com/iostackproject/swift-vertigo
[4]http://docs.openstack.org/developer/swift/
[5]https://github.com/openstack/storlets

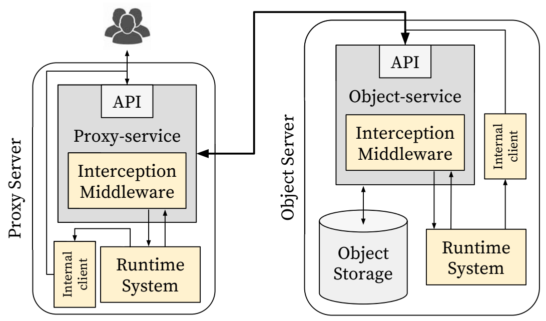


Fig. 3. High level architecture overview on OpenStack Swift

container servers, and object servers, the latter responsible for handling storage, replication, and management of objects. In addition to that, proxy servers expose the Swift API to users and stream objects to and from the client upon request.

As shown in Fig. 3, we have implemented our interception framework modifying both OpenStack Swift proxy and object servers.

On the other hand, Storlets extend Swift with the capability to run computations close to the data in a secure and isolated manner making use of `Docker`[6] 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.

Although Storlets have been the basis to implement active storage in Vertigo, we were forced to extend their functionality importantly, as they do not support *pipelining*, *implicit calls*, *metadata management*, and *orchestration*, among other issues. For instance, a simple composite function like this:

```
(compose (f1 f2)
   (lambda (x) (f1 (f2 x))))
(define grep-unzip (compose grep unzip))
```

cannot be implemented with the current Storelets framework. However, it can be easily achieved with Vertigo and its original micro-controller abstraction. This allows *active storage to be abstracted into smaller units that can be treated as disposable pieces*, which could be priced as a utility, among other benefits.

A. Micro-Controller Runtime Sytem

The *Runtime System* is the principal component of Vertigo. Taking Swift as basis, it is composed by three elements: the *Controller Engine*, the *Interception Middleware*, and the *Active Storage Engine*.

Swift Controller Engine. It is a daemon process providing isolation and safety thanks to an isolated environment using Linux containers (`Docker`). As a result, the micro-controllers are sandboxed: They have no direct network access, no system execution capabilities, no thread creation capabilities, and no access to the filesystem. The Controller Engine is shared by all the users of the same tenant, but for each container there exists one different instance. When this daemon receives the micro-controller list and all metadata, it runs all micro-controllers in the appropriate order.

[6]https://docs.docker.com/

Interception Middleware. We built a new Swift interception middleware with two major tasks: To deploy micro-controllers to a particular object(s) and to load micro-controllers to the Controller Engine.

Upon the deployment of a micro-controller, a trigger header of the type: `onGet`, `onPut`, `onUpdate`, `onDelete` and `onTimer`, must be appointed to tell the framework which life cycle events must be intercepted. For storing the micro-controller execution list, the framework uses Linux extended object attributes. Like Swift, which uses *user.swift.metadata* key for storing object metadata, Vertigo uses a personalized key (*user.swift.microcontroller*) for storing the trigger list.

As described in the previous section, a micro-controller may have specific metadata associated with the managed object. This metadata is stored along with the object in a separate file named *micro-controller-name.md*. For making this association, a user must issue a `put` request to the object that they want to assign the micro-controller, with the correspondent header, and then upload the metadata file associated to the micro-controller.

To better understand this, consider a user wants to associate a simple access counter micro-controller to a given object. The steps will be: First, a Swift `put` request with the header "X-MicroController-onGet:counter-1.0.jar" to associate the micro-controller *counter-1.0.jar* with an `onGet` trigger, followed by an upload of the *counter.md* metadata file to the storage server. The middleware is responsible for storing *counter.md* without modifying the contents of the target object.

Micro-controllers can be undeployed with the special header "X-MicroController-Delete" added in a `post` request. This delete operation removes the micro-controller from the trigger list and its related metadata file. Moreover, it is possible to retrieve the trigger list by adding the header "X-MicroController-list" into a `get` request over the object.

When a request arrives to an object, the middleware checks if the object has micro-controllers in the trigger related to the request. If there are micro-controllers assigned, the middleware automatically starts the Controller Engine and sends it the list of micro-controllers to execute, as well as the object metadata, the request metadata and the micro-controllers metadata.

One of the main characteristics of our SDS system is that micro-controllers can only be blocked during the execution a Storlet. If there is no Storlet (no active storage task to run), the micro-controller returns the control to the middleware, and the rest of micro-controller code runs asynchronously. If a micro-controller needs to put into execution one or more Storlets, the middleware remains waiting until it receives the processed data from the Storlet(s), and then sends it to the client.

Swift Active Storage Engine. This the third component of the Runtime System and is the responsible of processing the data. We leveraged the IBM Storlets framework to implement this component. Since the current Storlets framework only supports one Storlet per request, we modified it to enable the pipelining of Storlets in the same request. The Storlets can also be run in parallel; it is not necessary to finish one to run another, which allows for orchestration of active storage tasks, which was not possible with the original IBM Storlet framework.

B. Swift Object Storage Internal Client

The last architectural component of our framework is the Object Storage Internal Client. It converts requests from micro-controllers or from `cron` events to object storage requests. It can perform any Swift request such as `put`, `get`, `post` or `delete`. There are no restrictions at this level to make any request to Swift objects. But of course, all users are restricted to perform requests only to their owned objects and containers in Swift.

To configure an `onTimer` trigger, we adopted the following convention. When specifying the header "X-MicroController-onTimer" with the micro-controller name, the user must provide the date to execute the micro-controller, e.g $02/06$, or the date and the repetition frequency to execute it during a period of time, e.g., since $10/12$, once a week.

The Swift middleware stores all these data under the key *user.swift.microcontroller*, and then, it creates an entry for the `cron` daemon to automatically launch the task. To achieve this, `cron` issues a request to the Internal Client, which creates the Swift request and puts it into a queue until the nodes have sufficient resources to execute it.

V. Applications

In this section, we show some of the applications that our framework can support and the ease with which these applications can be built on top of Swift. As many other object stores, Swift, at the finest level, works at the object level, acting as simple repository of data. One of the outstanding features of Vertigo is that with simple programming abstractions, we can operate at the "content level", and for instance, automate the management of the objects according to the their *state* or the content itself. Here we provide some of these examples. All of them have been implemented over Swift and extensively evaluated in the next section.

Automated Deletion. A representative application of automated management is *self-deletion*, where certain objects self-destruct after a certain period of time or number of `get` operations. Such objects are meaningful for security applications. For instance, data protection and privacy laws in Europe[7] demand the deletion of personal data after a given retention time.

While components of the storage service may be put in place to periodically discover which objects are eligible for deletion, a more natural approach is to associate an automated deletion policy to each sensitive object and let it destroy itself[8]. This model offers several advantages over a centralized approach. For instance, it is more robust, since it does not depend on the effectiveness of the scheduled discovery jobs or any logically centralized management of deletions. On the contrary, each object decides itself when to be self-destructed without interfering with the rest, making the system more robust against failures and attacks.

Because an object may have dependencies on other objects, another characteristic of Vertigo is that allows to describe such dependencies in a metadata file, so when an object meets the conditions to be self-destructed, the micro-controller will also delete the related objects.

As a basic example, we have associated a micro-controller with an `onGet` trigger to implement objects that choose to delete themselves after being read a limited number of times. This limited-read objects could be used to keep personal data only for the number of runs that are absolutely necessary for their processing.

Content Level Access Control. With Vertigo, it is very easy to implement advanced forms of access control. Typically, access control in Object Storage operates at the granularity of objects, and hence, once an object is accessible to some party, he gets the full content of the object. Swift also follows this "all or nothing" approach where the access to objects inside a container is enforced through access control lists (ACLs). Such an access control mechanism may be insufficient in many cases, in particular, when objects contain sensitive content.

In the exercise to show another capability of our framework, we show how content level access control can be realized very easily in Swift thanks to our micro-controller abstraction. By "content level", we mean that Swift users will be able to access to certain parts of an object based on their credentials. To give a concrete example, consider the publicly available Adult dataset, from the UCI Machine Learning Repository[9], which contains about $48,000$ rows of census information. Each row contains attributes like `race`, `sex` and `marital-status`, which combined with explicit identifiers such as the `SSN`[10] that identify the record holders may leak sensitive information about a given individual. As a result, the records of this object should be accessed differently depending upon the user role. For instance, while an"FBI agent" should be able to access to all fields and issue an SQL query:

$Q1$: **SELECT** SSN, age, education, marital-status, race, sex, relationship, capital-gain, native-country **FROM** adult.data ,

a "census analyst" could be restricted to run SQL queries on a smaller view:

$Q2$: **SELECT** SSN, age, education, capital-gain, native-country **FROM** adult.data

To implement this example, we have linked a micro-controller to an `onGet` trigger to enforce content level access control on the object `adult.data`. We have defined a simple access policy that depending on the use role, "FBI agent" or "census analyst", allows to run queries on all the fields ($Q1$) or just onto smaller projection view ($Q2$).

This simple access policy has been stored as the metadata of the micro-controller, i.e., in the JSON formatted file *clac.md*. When a SQL query comes for the object `adult.data`, the target object server first checks the Swift ACL. If the object is accessible by that user, the micro-controller then reads the content level policy, executing the SQL query only if the user has the appropriate role. The main point of this example is that it shows *how our framework enables an object to change its behavior to suit the requirements of a given application*, thanks to the set of micro-controllers that specify how the object behaves.

Automated Prefetching. Among other features, Vertigo also provides a platform for managing the storage hierarchy. One clear example of this is prefetching. Prefetching adds efficiency because it actively preloads objects into a cache. And as a result, it can minimize disk IO operations. The distinguishing

[7]EU law applies to the processing of personal data as defined in article 2 of Directive 95/46/EC, namely to any information relating to an identified or identifiable natural person.

[8]We note that an object does not delete itself immediately, but rather stays available until all replicas are deleted due to eventual consistency.

[9]http://archive.ics.uci.edu/ml/datasets/Adult

[10]As the Adult dataset does not contain explicit identifiers, we added a random SSN to each row using the Fake Name Generator.

characteristic of our framework is that prefetching can be done per object to suit the application-specific requirements instead of system-wide. This flexibility is useful for applications that put different degrees of emphasis on performance and latency. For instance, based on the past surfing activities, an access to an HTML file may preload objects of other pages.

As a basic example, we have implemented a simple Web prefetching mechanism. When a user stores an HTML file for the first time, a micro-controller associated with an `onPut` trigger parses it to identify the embedded objects. The result of this process is a list of the Swift objects that compose each specific HTLM document. Such a list is stored in JSON format in the metadata file *prefetching.md* to enable the `onGet` micro-controller to preload all the embedded objects into the `memcached` [14] when the HTML file is fetched as a result of cache miss. Caching is done at the proxy servers for fast retrieval and to lighten the read load on the object servers.

Active Storage Orchestration. Finally, another feature of our framework is that it can bring computation close to data as another *active storage* framework. Although that idea is not a new concept, we wanted to show it here as a property of our SDS management framework. This capability has been inherited by the IBM Storlet Engine. However, our framework permits the *pipelining* of consecutive computations that is not possible to perform with the IBM Storlet Engine. This capability is also important for data management, as it allows to perform a sequence of transformations on an object to enforce a storage policy. For instance, a tenant may associate an `onPut` micro-controller with a large document of type "application/xml" to detect the differences with a previous version of this document and afterward use `gzip` to compress the resulting deltas. Also, this type of micro-controllers may be very useful to orchestrate *active storage* tasks and implement ELT (extract, load, and transform) functions. With our micro-controllers, we can provide an intermediary transformation layer between raw object storage and (big) data analytics.

As a simple example, we have built a pipeline of two *active storage* tasks orchestrated by an `onGet` micro-contoller. The input object is a PDF file that first goes through a transcoding task to convert it to a text file which is then input to a `grep` task to output only the lines that match a query. `grep` has been utilized to micro-benchmark systems like Hadoop and Spark [15], which shows the potential of our framework for data transformations.

VI. Evaluation

We ran micro-benchmarks to measure the overheads associated with our micro-controllers. We did so by running the four applications introduced in V, because they cover all the features of our novel SDS framework.

A. System Setup

Our experimental testbed consisted of a host (or client) with 2VCPUs and 4GB RAM. On the server side, we deployed Vertigo in an 8-machines rack OpenStack Swift (kilo version) installation formed by 1 proxy node Dell PowerEdge R320 with 12GB RAM and 7 storage nodes Dell PowerEdge R320 with 8GB RAM. All of these machines, including the client, were connected via GbE. All the rack machines ran Ubuntu Server 14.04. The client host ran Ubutnu 14.04.2 CloudImage.

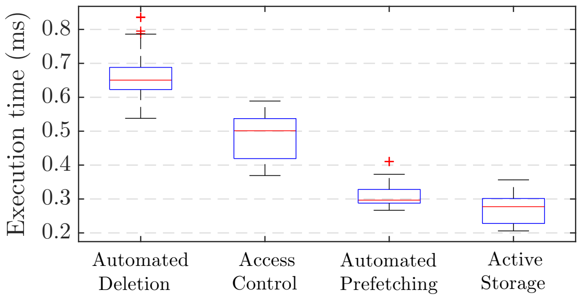


Fig. 4. Micro-controller execution time

B. Application Setup

Here we describe the specific setup for the four applications:

Automated Deletion. We utilized a random 100MB file. This use case represents the real overhead of Vertigo, because the micro-controller does not need to execute any Storlet, which will demonstrate the lightweight overhead of our approach.

Content Level Access Control. We used the dataset described in Section V but we extended to 100MB. As explained before, the dataset content is restricted to the type of user that launches the SQL query to the file. To this aim, we used Swift user roles to return only specific fields. In this case, the micro-controller reads from his metadata file the fields allowed for the user role (`age`, `education`, `capital-gain`, `native-country`), and then call the SQL Storlet to filter it.

Automated Prefetching. We took as an HTML file the Google HTML5 slide template[11] for creating Web presentations, which consists of a single HTML document (`index.html`) and several embedded objects, including Javascript and 10 image files of around 3.5MB. Since it is a presentation file, it is thus susceptible of simultaneous reading by multiple users (think of as a university lecture), for we believed it to be a good example of caching with Web prefetching. Although more sophisticated prediction techniques for Web prefetching should be applied in practice, the present example is by far sufficient to show the inventive aspects of Vertigo.

To simulate the HTTP requests and generate the workload, we made use of Apache JMeter[12].

Active Storage Orchestration. For this experiment, we used a single PDF document of 100MB. More precisely, the `onGet` micro-controller instrumented the `grep` Storlet to return all lines of this document that starts with 'a' (regex:"^a"), after being converted into text with the transcoder Storlet. As discussed before, this is a very nice example of active storage orchestration in which a pipeline of two active storage tasks is built at runtime. Notice that the `grep` Storlet needs to have all the text before filtering it with the regular expression, so our interception middleware remains dormant until this Storlet starts to return the answer.

C. Application characteristics

Table I shows information of our four Vertigo applications. The *Instructions* column gives the number of Java instructions

[11]https://code.google.com/archive/p/html5slides/
[12]http://jmeter.apache.org/

required to execute a micro-controller, while *Execution Time* gives the execution time for that micro-controller (the average of 200 executions as shown in Fig. 4). *Code size* shows the size of each micro-controller. From this table, it can be seen that our micro-controllers are very lightweight and the speed at which they run is always inferior to 1 ms.

TABLE I
MICRO-CONTROLLER INFORMATION

Application	**Instructions**	**Execution Time(mean)**	**Code Size**
Automated Deletion	≈ 15	0.66ms	2.2KB
Content Level Access Control	≈ 10	0.48ms	1.8KB
Automated Prefetching	≈ 5	0.30ms	1.6KB
Active Storage Orchestration	≈ 5	0.27ms	1.5KB

D. Results

We evaluated the the overhead of Vertigo by analyzing the improvement on the overall execution time. In this experiment, `get` operations were performed on the same Swift objects that make up each application. More specifically, we measured the response time, the transfer time and the total amount of data pushed out of Swift to the clients. We also recorded the individual execution times of the interception middleware, the micro-controllers and the involved Storlets. All the tests were run 200 times and the results were averaged to create the plots in Fig. 5, Fig. 6 and Fig. 7.

We evaluated the following two configurations: Traditional storage or baseline configuration (TS), namely, Swift without Vertigo, and the micro-controlled version of Swift (MC).

Execution Time. Fig. 5 shows the execution time breakdown for the different applications. We split the execution time into response time, transfer time and process time. The response time includes the time of processing the corresponding `get` request until the first byte of the object is received by the client host. The transfer time stands for the time elapsed between the reception of the first and the last byte of the object at the client host. The process time includes all the time that the client host spends running some computation over the data.

Overall, the system overhead is very low in all applications. For automated deletion, the overhead is only of 41ms. For the content level access control use case, the TS configuration needs to download all the file before filtering it. However, the MC configuration only requires to transfer 30% of the data. In this case. the main source of overhead comes from the fact the interception middleware remains waiting for the SQL Storlet to send the query results back to the client host as depicted in Fig. 7(b). Despite this, the MC configuration saves 1.6 seconds in comparison with the TS one.

In the automated prefetching application, the fact that all images are preloaded into the proxy server `memcached` saves around 54ms in the whole execution time as shown in Fig. 5(c).

Finally, Fig. 5(d) reports the execution time of the active storage orchestration use case. With no micro-controllers (TS), the client needs to download the complete PDF file, transcode it to text, and then apply the `grep` filter. Making this operation on the server side with Vertigo, the client hosts saves around 1.86 seconds, as Swift only needs to transfer the result of the `grep` Storlet.

Bandwidth Usage. Fig. 6 illustrates the total volume of data received by the client host in both configurations. As clearly shown in this figure, Vertigo can save significant bandwidth in some scenarios. For instance, for content level access control, it can avoid transferring 71, 7 MB of content, depending on the user role as a result of filtering out sensitive information. Such a property is very important, since it increases the scalability of a Swift deployment by transforming underutilized CPU cycles into bandwidth savings. For example, we could associate a micro-controller to certain objects and monitor their bandwidth usage. Later on, a periodical `onTimer` micro-controller could preprocess that objects to save bandwidth or to help perform sophisticated management policies.

Timeline. Fig. 7 shows the timeline of the `get` requests for the different uses cases, both on the client side (response and transfer) and the server side (middleware, micro-controller and Storlets).

For automated deletion, the `onGet` micro-controller does not need to call any Storlet, as shown in Fig. 7(a). Hence, when the micro-controller informs the interception middleware about this fact, Swift can start sending data to the user. In parallel, the micro-controller updates the number of accesses and checks if it is necessary to delete the object.

For the access control use case, however, the interception middleware must wait for the SQL Storlet to start, introducing some overhead into the system as depicted in Fig. 7(b). Despite this, the micro-controlled version of this use case is still more efficient as discussed before.

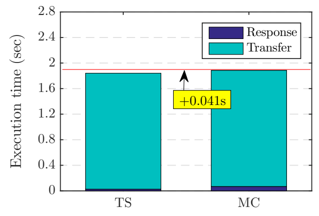
The timeline for the prefetching use case, and in all cases that is not necessary to process the data with a Storlet, the timeline will be the same as in the automated deletion application, with the difference in the response time. We assume that when a user uploads the `index.html` file, it has been executed a Storlet that extracts all static resources, and stores their Swift keys to the metadata. Upon the first `get` request, the micro-controller will load these static resources to the proxy server cache through the Internal Client. In this way, when the browser interprets the HTML file, and also for the next 199 `get` requests, all resources will be already preloaded into the cache, saving IO bandwidth and time.

Finally, the timeline for the active storage orchestration use case is shown in Fig. 7(c). The `grep` Storlet needs to have all the text before filtering it with the regular expression, and hence, the middleware remains idle until that Storlet starts to return the result of the processing.

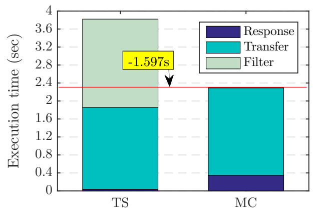
VII. CONCLUSION

This paper presents a novel software-defined architecture for Object Storage. Our architecture introduces the novel concept of object-based micro-controllers as a decentralized mechanism to transparently extend and intercept Object Stores. Our micro-controllers (control plane) are executed in a sandboxed environment in the storage nodes and they can intercept any life cycle request in the desired objects (data plane).

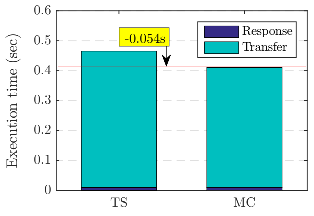
We have demonstrated how micro-controllers increase the programmability and data management of Object Stores with concrete examples: pre-fetching, content-level access control, and automated deletion. Furthermore, we demonstrated that our interception framework is very lightweight achieving low overheads. We believe that object micro-controllers can become a useful programming abstraction for extending Object Storage systems.



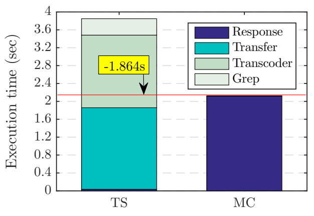
(a) Automated deletion



(b) Content level access control

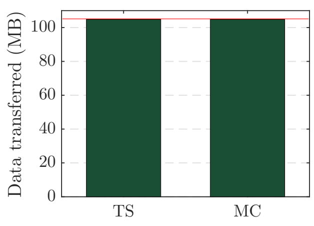


(c) Automated prefetching

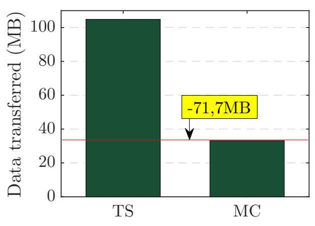


(d) Active storage orchestration

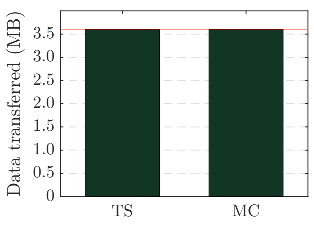
Fig. 5. Execution time breakdown for different applications



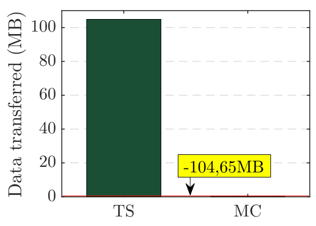
(a) Automated deletion



(b) Content level access control

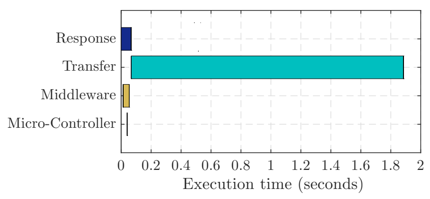


(c) Automated prefetching

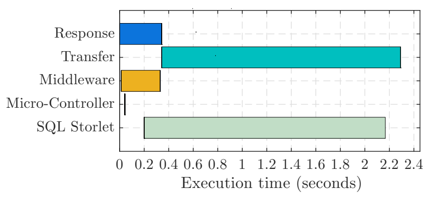


(d) Active storage orchestration

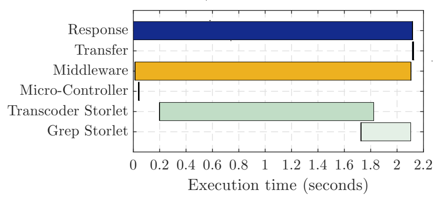
Fig. 6. Total amount of data transferred for different applications



(a) Automated deletion



(b) Content level access control



(c) Active storage orchestration

Fig. 7. Server-side execution time-line breakdown for different applications

Acknowledgment

This work has been partially supported by the EU in the context of the project IOStack: Software-defined Storage for Big Data (H2020-ICT-2014-7-1).

References

[1] L. Qin and D. Feng, "Active storage framework for object-based storage device," in *AINA'06*, 2006, pp. 97–101.
[2] T. M. John, A. T. Ramani, and J. A. Chandy, "Active storage using object-based devices," in *IEEE Cluster'08*, 2008, pp. 472–478.
[3] Y. Xie, D. Feng, Y. Li, and D. D. Long, "Oasis: an active storage framework for object storage platform," *Future Generation Computer Systems*, vol. 56, pp. 746–758, 2016.
[4] S. Rabinovici-Cohen, E. Henis, J. Marberg, and K. Nagin, "Storlet engine: performing computations in cloud storage," IBM Technical Report H-0320 (August 2014), Tech. Rep., 2014.
[5] EMC, "Emc atmos cloud storage architecture," EMC Technical Report H-9505.1 (September 2014), Tech. Rep., 2014.
[6] R. Geambasu, A. A. Levy, T. Kohno, A. Krishnamurthy, and H. M. Levy, "Comet: An active distributed key-value store." in *USENIX OSDI'10*, 2010, pp. 323–336.
[7] 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*, 2013, pp. 182–196.
[8] S. Narayan and J. A. Chandy, "Attest: Attributes-based extendable storage," *J. Syst. Softw.*, vol. 83, no. 4, pp. 548–556, Apr. 2010.
[9] O. Momin, C. Karakoyunlu, M. T. Runde, and J. A. Chandy, "Creating a programmable object storage stack," in *ACM PFSW'14*, 2014, pp. 3–10.
[10] S. Narayan and J. A. Chandy, "Extendable storage framework for reliable clustered storage systems," in *IEEE IPDPSW'10*, 2010, pp. 1–4.
[11] R. Gracia-Tinedo, P. García-López, M. Sánchez-Artigas, J. Sampé, Y. Moatti, E. Rom, D. Naor, R. Nou, T. Cortés, P. Michiardi, and W. Oppermann, "Iostack: Software-defined object storage," *IEEE Internet Computing*, to appear.
[12] A. Alba, G. Alatorre, C. Bolik, A. Corrao, T. Clark, S. Gopisetty, R. Haas, R. I. Kat, B. Langston, N. Mandagere *et al.*, "Efficient and agile storage management in software defined environments," *IBM Journal of Research and Development*, vol. 58, no. 2/3, pp. 5–1, 2014.
[13] A. Darabseh, M. Al-Ayyoub, Y. Jararweh, E. Benkhelifa, M. Vouk, and A. Rindos, "Sdstorage: a software defined storage experimental framework," in *IEEE IC2E'15*, 2015, pp. 341–346.
[14] B. Fitzpatrick, "Distributed caching with memcached," *Linux J.*, vol. 2004, no. 124, pp. 5–, 2004.
[15] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu, "Bigdatabench: A big data benchmark suite from internet services," in *IEEE HPCA'14*, 2014, pp. 488–499.