ENHANCING THE PROGRAMMABILITY OF CLOUD OBJECT STORAGE

Josep Sampé Domenech

ADVERTIMENT. L'accés als continguts d'aquesta tesi doctoral i la seva utilització ha de respectar els drets de la persona autora. Pot ser utilitzada per a consulta o estudi personal, així com en activitats o materials d'investigació i docència en els termes establerts a l'art. 32 del Text Refós de la Llei de Propietat Intel·lectual (RDL 1/1996). Per altres utilitzacions es requereix l'autorització prèvia i expressa de la persona autora. En qualsevol cas, en la utilització dels seus continguts caldrà indicar de forma clara el nom i cognoms de la persona autora i el títol de la tesi doctoral. No s'autoritza la seva reproducció o altres formes d'explotació efectuades amb finalitats de lucre ni la seva comunicació pública des d'un lloc aliè al servei TDX. Tampoc s'autoritza la presentació del seu contingut en una finestra o marc aliè a TDX (framing). Aquesta reserva de drets afecta tant als continguts de la tesi com als seus resums i índexs.

ADVERTENCIA. El acceso a los contenidos de esta tesis doctoral y su utilización debe respetar los derechos de la persona autora. Puede ser utilizada para consulta o estudio personal, así como en actividades o materiales de investigación y docencia en los términos establecidos en el art. 32 del Texto Refundido de la Ley de Propiedad Intelectual (RDL 1/1996). Para otros usos se requiere la autorización previa y expresa de la persona autora. En cualquier caso, en la utilización de sus contenidos se deberá indicar de forma clara el nombre y apellidos de la persona autora y el título de la tesis doctoral. No se autoriza su reproducción u otras formas de explotación efectuadas con fines lucrativos ni su comunicación pública desde un sitio ajeno al servicio TDR. Tampoco se autoriza la presentación de su contenido en una ventana o marco ajeno a TDR (framing). Esta reserva de derechos afecta tanto al contenido de la tesis como a sus resúmenes e índices.

WARNING. Access to the contents of this doctoral thesis and its use must respect the rights of the author. It can be used for reference or private study, as well as research and learning activities or materials in the terms established by the 32nd article of the Spanish Consolidated Copyright Act (RDL 1/1996). Express and previous authorization of the author is required for any other uses. In any case, when using its content, full name of the author and title of the thesis must be clearly indicated. Reproduction or other forms of for profit use or public communication from outside TDX service is not allowed. Presentation of its content in a window or frame external to TDX (framing) is not authorized either. These rights affect both the content of the thesis and its abstracts and indexes.

UNIVERSITAT
ROVIRA i VIRGILI

Enhancing the Programmability of Cloud Object Storage

JOSEP SAMPÉ DOMENECH

DOCTORAL THESIS
2018

UNIVERSITAT ROVIRA I VIRGILI
ENHANCING THE PROGRAMMABILITY OF CLOUD OBJECT STORAGE
Josep Sampé Domenech

UNIVERSITAT ROVIRA I VIRGILI
ENHANCING THE PROGRAMMABILITY OF CLOUD OBJECT STORAGE
Josep Sampé Domenech

UNIVERSITAT ROVIRA I VIRGILI
ENHANCING THE PROGRAMMABILITY OF CLOUD OBJECT STORAGE
Josep Sampé Domenech

Josep Sampé Domenech

Enhancing the Programmability of Cloud Object Storage

Doctoral Thesis

Advised by
Dr. Pedro García López
Dr. Marc Sánchez Artigas

Department of Computer Engineering and Mathematics

UNIVERSITAT ROVIRA i VIRGILI

Tarragona
2018

UNIVERSITAT ROVIRA I VIRGILI
ENHANCING THE PROGRAMMABILITY OF CLOUD OBJECT STORAGE
Josep Sampé Domenech

UNIVERSITAT
ROVIRA i VIRGILI

Departament d'Enginyeria
[DEIM]
Informàtica i
Matemàtiques

FAIG CONSTAR que aquest treball, titulat "Enhancing the Programmability of Cloud Object Storage", que presenta Josep Sampé Domenech per a l'obtenció del títol de Doctor, ha estat realitzat sota la meva direcció al Departament d'Enginyeria Informàtica i Matemàtiques d'aquesta universitat i que compleix els requeriments per poder optar a la Menció Internacional.

HAGO CONSTAR que el presente trabajo, titulado "Enhancing the Programmability of Cloud Object Storage", que presenta Josep Sampé Domenech para la obtención del título de Doctor, ha sido realizado bajo mi dirección en el Departamento de Ingeniería Informática y Matemáticas de esta universidad y que cumple los requisitos para poder optar a la Mención Internacional.

I STATE that the present study, entitled "Enhancing the Programmability of Cloud Object Storage", presented by Josep Sampé Domenech for the award of the degree of Doctor, has been carried out under my supervision at the Department of Computer Engineering and Mathematics of this university, and that it fulfills all the requirements to be eligible for the International Doctorate Award.

Tarragona, 20 de Setembre/20 de Septiembre/September 20, 2018

Els directors de la tesi doctoral
Los directores de la tesis doctoral
Doctoral thesis supervisors

Dr. Pedro García López

Dr. Marc Sánchez Artigas

UNIVERSITAT ROVIRA I VIRGILI
ENHANCING THE PROGRAMMABILITY OF CLOUD OBJECT STORAGE
Josep Sampé Domenech

Acknowledgements

This dissertation has been written during my time at the "Arquitectures i Serveis Telemàtics (AST)" research group at Universitat Rovira i Virgili. As such, in first place I would like to thank my advisors Dr. Pedro García López and Dr. Marc Sánchez Artigas their endless patience, help and advise through my formation as a researcher. Without their teamwork as advisors and their open-minded perspective of what is doing research, I would not have been able of completing this thesis. Also, I thank to all the people in the AST research group who shared these years with me. Specially, Raúl Gracia Tinedo for his help and guidance, and Gerard París, for the moments that we enjoyed together in the lab.

Secondly, I would like to thank Ofer Biran, Gil Vernik and Dalit Naor for accepting me as an intern in their research group at IBM Haifa Research Labs. The months I spent in Israel were not only fruitful from a professional perspective, but also very enriching from a personal viewpoint.

Last but not least, my deepest appreciation goes to my family and friends. I would like to specially thank my parents Amado and Teresa, my brother Xavi, his wife Montse, and the new member of my family, my nephew Aran, for their love and support.

Josep Sampé Domenech
Vilalba dels Arcs, September 2018

This work has been partially funded by the European Union Horizon 2020 Framework Programme, in the context of the project IOStack: Software-defined Storage for Big Data (H2020-644182), and by the Spanish Ministry of Science and Innovation (Grant TIN2016-77836-C2-1-R).

Abstract

In a world that is increasingly dependent on technology, digital data is generated in an unprecedented way. This makes companies that require large storage space, such as Netflix or Dropbox, use cloud storage solutions where data is remotely maintained, managed, and backed up, in an easy and cheap way. Particularly, cloud object stores are widely adopted and increasingly used for storing these huge amounts of data. This is mainly thanks to their built-in characteristics, such as simplicity, scalability and high-availability. Moreover, the evolution of cloud computing, in what refers, for example, to data analysis, make cloud object stores an important actor in today's cloud ecosystem.

However, cloud object stores face three main challenges: 1) Flexible management of multi-tenant workloads. Commonly, cloud object stores are multi-tenant systems, meaning that all tenants share the same system resources, which could lead to interference problems. Furthermore, it is now complex to manage heterogeneous storage policies in a massive scale. 2) Data self-management. Cloud object stores themselves do not offer much flexibility regarding data self-management by tenants. Typically, they are rigid, non-programmable systems, which prevent tenants to handle the specific requirements of their objects. 3) Elastic computation close to the data. Placing computations close to the data in the storage system can be useful to reduce data transfers. But, the challenge here is how to achieve elasticity in those computations without provoking resource contention and interferences in the storage layer.

In this thesis, we present three novel research contributions that solve the aforementioned challenges. Firstly, we introduce the first Software-defined Storage (SDS) architecture for cloud object stores that separates the control plane from the data plane, allowing to manage multi-tenant workloads in a flexible and dynamic way. For example, by applying different service levels of bandwidth to different tenants. Secondly, we designed a novel policy abstraction called microcontroller that transforms common objects into smart objects, enabling tenants to programmatically manage their behavior. For example, a content-level access control microcontroller attached to an specific object to filter its content depending on who is accessing it. Finally, we present the first elastic data-driven serverless computing platform that mitigates the resource contention problem of placing computation close to the data.

UNIVERSITAT ROVIRA I VIRGILI
ENHANCING THE PROGRAMMABILITY OF CLOUD OBJECT STORAGE
Josep Sampé Domenech

Thesis Publications

- Josep Sampé. *Towards Cooperative Analytics in Disaggregated Big Data Clusters* in 2nd URV Doctoral Workshop in Computer Science and Mathematics (eds Marc Sánchez Artigas and Aida Valls Mateu) (Publicacions URV, Tarragona, November 2015), pp.9-13. ISBN: 978-84-8424-399-1.

- Raúl Gracia-Tinedo, Pedro García-López, Marc Sánchez-Artigas, Josep Sampé, Yosef Moatti, Eran Rom, Dalit Naor, Ramon Nou, Toni Cortés, William Oppermann, Pietro Michiardi. *IOStack: Software-defined Object Storage* in IEEE Internet Computing Journal (Volume: 20, Issue: 3, May-June 2016) Impact Factor: 1.929, Q2, pp.10-18, ISSN: 1089-7801.

- Josep Sampé, Pedro García-López, Marc Sánchez-Artigas. *Vertigo: Programmable Micro-controllers for Software-defined Object Storage* in 2016 IEEE 9th International Conference on Cloud Computing (CLOUD '16) CORE B (June 2016), pp.180-187. ISBN: 978-1-5090-2620-3.

- Raúl Gracia-Tinedo, Josep Sampé, Edgar Zamora, Marc Sánchez-Artigas, Pedro García-López, Yosef Moatti, Eran Rom. *Crystal: Software-defined Storage for Multi-tenant Object Stores* in 15th Usenix Conference on File and Storage Technologies (FAST '17) CORE A (February 2017), pp.243-256, ISBN: 978-1-931971-36-2.

- Josep Sampé, Marc Sánchez-Artigas, Pedro García-López, Gerard París. *Data-driven Serverless Functions for Object Storage* in 18th ACM/IFIP/USENIX Middleware Conference (Middleware '17) CORE A (December 2017), pp.121-133. ISBN: 978-1-4503-4720-4.

Other Publications

- Raúl Gracia-Tinedo, Yongchao Tian, Josep Sampé, Hamza Harkous, John Lenton, Pedro García-López, Marc Sánchez-Artigas, Marko Vukolic *Dissecting UbuntuOne: Autopsy of a Global-scale Personal Cloud Back-end* in 2015 Internet Measurement Conference (IMC '15) CORE A (October 2015), pp.155-168. ISBN: 978-1-4503-3848-6.

- Yosef Moatti, Eran Rom, Raúl Gracia-Tinedo, Dalit Naor, Doron Chen, Josep Sampé, Marc Sánchez-Artigas, Pedro García-López, Filip Gluszak, Eric Deschdt, Francesco Pace, Daniele Venzano, Pietro Michiardi. *Too Big to Eat: Boosting Analytics Data Ingestion from Object Stores with Scoop* in 2017 IEEE 33rd International Conference on Data Engineering (ICDE '17) CORE A* (April 2017), pp.309-320, ISBN: 978-1-5090-6543-1.

- Josep Sampé, Gil Vernik, Marc Sánchez-Artigas, Pedro García-López. *Serverless Data Analytics in the IBM Cloud* **accepted in** 19th ACM/IFIP/USENIX Middleware Conference (Middleware '18) CORE A (December 2018), ISBN: 978-1-4503-6016-6

UNIVERSITAT ROVIRA I VIRGILI
ENHANCING THE PROGRAMMABILITY OF CLOUD OBJECT STORAGE
Josep Sampé Domenech

UNIVERSITAT ROVIRA I VIRGILI
ENHANCING THE PROGRAMMABILITY OF CLOUD OBJECT STORAGE
Josep Sampé Domenech

Table of contents

UNIVERSITAT ROVIRA I VIRGILI
ENHANCING THE PROGRAMMABILITY OF CLOUD OBJECT STORAGE
Josep Sampé Domenech

List of figures

UNIVERSITAT ROVIRA I VIRGILI
ENHANCING THE PROGRAMMABILITY OF CLOUD OBJECT STORAGE
Josep Sampé Domenech

List of tables

UNIVERSITAT ROVIRA I VIRGILI
ENHANCING THE PROGRAMMABILITY OF CLOUD OBJECT STORAGE
Josep Sampé Domenech

Acronyms

AOP Aspect Oriented Programming.

API Application Programming Interface.

CRUD Create, Read, Update and Delete.

DASD Direct Access Storage Device.

DSL Domain-specific Language.

ETL Extract-Transform-Load.

FC Fiber Channel.

GB Gigabyte.

GbE Gigabit Ethernet.

HDD Hard Disk Drive.

HDFS Hadoop Distributed File System.

HPC High-performance Computing.

HTML HyperText Markup Language.

HTTP Hypertext Transfer Protocol.

I/O Input/Output.

ID Identifier.

IFTTT If-This-Then-That.

IP Internet Protocol.

iSCSI Internet SCSI.

JVM Java Virtual Machine.

KB Kilobyte.

KBps Kilobytes per second.

LAN Local Area Network.

LRU Least Recently Used.

MB Megabyte.

Mbps Megabits per second.

MPI Message Passing Interface.

ms Millisecond.

NAS Network Attached Storage.

NFS Network File System.

OS Operating System.

OSD Object Storage Device.

PB Petabyte.

RAID Redundant Array of Independent Disks.

RDMS Relational Database Management Systems.

REST Representational State Transfer.

SAN Storage Area Network.

SATA Serial Advanced Technology Attachment.

SCSI Small Computer System Interface.

SDDC Software-defined Data Center.

SDN Software-defined Networking.

SDS Software-defined Storage.

SLA Service Level Agreement.

SLO Service-Level Objectives.

SMB Server Message Block.

SSD Solid-state Drive.

TB Terabyte.

TCP Transmission Control Protocol.

VM Virtual Machine.

WAN Wide Area Network.

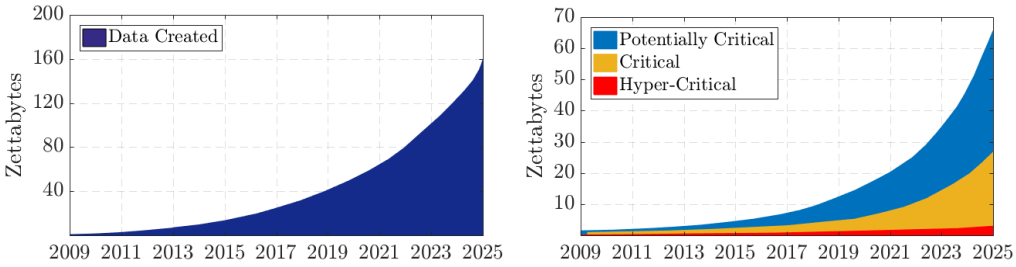
WSGI Web Server Gateway Interface.

UNIVERSITAT ROVIRA I VIRGILI
ENHANCING THE PROGRAMMABILITY OF CLOUD OBJECT STORAGE
Josep Sampé Domenech

Chapter 1

Motivation and Challenges

Nowadays, data is experiencing an incredible growth in the digital universe. The adoption of the new technologies by both common users and enterprises, increasingly generates data in a never-before-seen proportion. IDC, in a collaborative report with Seagate [1], estimates that by 2025 the global datasphere will grow to 163ZB. That is more than five times the 30.1ZB of data generated in 2018 (Fig. 1.1a). Most of this data is generated for further processing. Data analysis extracts relevant information thus increasing its value. In this sense, IDC also estimates that by 2025, nearly 20% of the data will be critical[1] to our daily lives and nearly 10% of that will be hypercritical[2] (Fig. 1.1b).



(a) Unstructured data growth prevision (b) Data criticality over time

Fig. 1.1 Evolution of the global datasphere

The data that makes up the global datasphere is commonly classified in two main types: structured and unstructured. While structured data represents that

[1]Critical. Data known to be necessary for the expected continuity of users' daily lives.
[2]Hypercritical. Data with direct and immediate impact on the health and wellbeing of users.

information which has a predefined structure, such as that data that is stored in relational databases, unstructured data is information that either does not have a predefined data model or is not organized in a predefined manner, such as audio and video files or text logs. In this sense, some analysis [2] show how unstructured data is growing much faster in relation with others. Without going any further, by 2022, 93% of all data in the digital universe is predicted to be unstructured [3]. At this moment, unstructured data is growing at the rate of 62% per year [4]. Consequently, organizations are moving away from traditional infrastructures. Nowadays, and each time more, the storage systems have to deal with this huge and growing mix of unstructured data, which comes from, for example, social media, health care, or science, and includes audio, video, logs, health records, sensor data, and emails to name few.

While traditionally this data was stored in structured block storage devices, the limitations of these systems in terms of simplicity and scalability made object stores pervasive in today's world. It is estimated that more than 80% of enterprise data will be stored in scale-out storage systems [3] in the next years. In this sense, object stores are specially designed to handle this huge data growth [5]. They can scale to hundreds of petabytes in a single namespace without suffering any sort of performance degradation, while providing high-availability of the data by storing copies of the same object on multiple nodes. An object storage system is capable of storing files and metadata about files, which consists of the attributes for the actual data being stored. Many attributes are identifiers for information to be easy sortable, analyzable, put into context, and create value. Moreover, object stores are software-based systems, which allow them to be deployed on standard servers or cloud-based resources.

1.1 Problem Statement

The world is becoming software driven, but that does not necessarily mean programmable [6]. This is what happens exactly with cloud object storage systems. By analyzing the offer catalog of the different cloud providers [7–11] and open-source solutions [12, 13], we noted that the available object storage systems do not offer too much ways of extensibility, which inevitably limits the programmability of the system. Programmability unlocks the full potential of computing and increases the innovation, and we think that cloud object stores lack of it. The continuously growing of the unstructured data makes necessary

cloud-based object stores to be more programmable, thus providing a new layer of storage automation and data management, both for storage administrators and tenants.

To get started, commonly, cloud object stores are services with multi-tenant support. That is, the object store allows to create different instances of the service for each tenant[3]. Multi-tenancy ensures data isolation and sharing among tenants and users of the same tenant, all of these over the same system software and infrastructure. Sharing the same infrastructure across all tenants is very beneficial in terms of both CAPEX[4] and OPEX[5]. However, it is also one of its main drawbacks. Operating over the same software and infrastructure means that all the tenants of the storage service share the same system resources, which could lead to performance issues. In this sense, the first question we pose about programmability in cloud object storage systems is the next:

Question 1: *Is there a lack of flexibility for handling multi-tenant workloads?*

Despite their growing popularity, object stores are not well prepared for heterogeneity. Typically, a deployment of an object store uses a monolithic configuration setup, even when the same object store acts as a substrate for different types of applications with time-varying requirements. This results in all applications experiencing the same service level, though the workloads from different applications can vary dramatically. Because tenants share system resources with other public tenants, performance can be wildly inconsistent. This performance unpredictability is also aggravated by the fact that tenants have little control over where their data resides, so it is not rare that hot data from two separate tenants go to the same physical node. 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), 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. In this sense, we think that a multi-tenant management technology is a required layer in cloud object stores. This will enhance the programmability of cloud object storage systems by adding a new software

[3]A tenant is a group of users who share a common access to the storage instance.

[4]Capital expenditure or capital expense (CAPEX) is the money a company spends to buy, maintain, or improve its fixed assets, such as hardware and systems.

[5]Operating expense (OPEX) is an ongoing cost for running a product, business, or system.

management layer. It will allow to dynamically orchestrate the storage cluster by means of high-level policies. However, this approach is purely oriented to be admin-based, and as a consequence, the next question we pose is:

Question 2: *Are tenants able to manage their data in cloud object stores?*

Unfortunately, in cloud object stores the tools given to users to manage their data is surprisingly limited. Typically, at the object level, users can only decide the expiration time for their data. When an object expires, that piece of data is automatically erased by the system and becomes no longer accessible. Beyond this, an object cluster is simply a data silo where to put data. Cloud vendors offer the possibility to set policies at a higher level, such as at the bucket level. However, these policies are limited to: access control polices, to decide which users of the tenant can access to the data contained in a specific bucket, and sometimes to: object replication level policies, which finally guarantee the availability of the objects of a bucket. Moreover, some object storage systems also offer the possibility to activate an encryption mechanism to secure the data in a specific bucket.

In spite of this, nowadays, a new computing abstraction called *serverless computing* enables cloud services to process data with user-defined functions. Functions can be event-based or proactively invoked. Referring to object storage services, functions can be invoked just when the data is uploaded to the cluster. That is, it is possible to create a policy that launches a function on the upload event. However, the computation occurs in a decoupled way, having to transfer all the data from the storage to the compute cluster, eventually storing the result back to the storage cluster if needed.

These limitations prevent cloud object stores to easily adapt to users' requirements. Serverless computing may solve some of the use cases of data management when the data is uploaded to the cloud. But the model does not allow to control the full lifecycle of the objects, leaving a spectrum of possible use cases without the possibility of management. In this sense, we think that a self-management layer for tenants is necessary in cloud object stores, in order to procure tenants the ability to manage their data in a flexible way.

Adding a new storage management layer, derived from *Question 1*, and a data management layer, derived from *Question 2*, are the two main research challenges to enhance the programmability of cloud object stores of this thesis. However, usually these enhancements mean overloading the storage cluster with new

functionalities and computation tasks that consume storage resources. Moving computation close to the data may inevitably produce resource contention. That is, there may be not enough compute resources to manage the object storage system. Thus, it is inevitable to pose the following question:

Question 3: *Are there enough compute resources in the storage layer for enhancing the programmability?*

Storage management and data management usually refer to running computation tasks within the storage cluster. However, data transformations tasks such as compression or encryption can uncontrollably increase the CPU and memory usage, interfering in the normal operation of the storage service. In a storage cluster, compute resources are limited, and the idea behind its scalability is to scale-up or scale-out the resources when more storage space is required, and not when CPU and memory is overloaded. Thus, to solve the more than likely elasticity and resource contention issues, one can think that the correct approach would be to decouple the compute capabilities from the storage. Without going any further, this is how many systems are currently built. Typically in the cloud, vendors offer separated compute and storage clusters. Disaggregation, among other things, ensures the correct management and scalability, and it greatly simplifies the deployment and maintenance of these services.

Nevertheless, the proposed systems derived from *Question 1* and *Question 2* have some benefits that we want to preserve. They are *data locality* and *inline processing*[6]. In this context, data locality is a term used to define that the computation is done where the data is, instead of moving the data where the computation is. Placing computation close to the data significantly decreases the data movement between storage and compute clusters, saving an enormous and limited quantity of bandwidth. Moreover, data locality together with inline processing may provide better performance in terms of execution time. The fundamental reason is that transferring input data to the compute cluster for processing, and then storing the results back to the storage cluster, may lengthen the total storage request time. Furthermore, it significantly increases disk read/write operations in the storage nodes.

It can be said that moving data between clusters is only an acceptable approach when real-time response is not required. However, in today's world, Internet is

[6]Inline processing refers to process an object data stream synchronously, as it comes in/out form the storage cluster.

becoming faster, and each time more applications demand real-time processing capabilities and low latency interconnection [1, 2]. In this instantaneous world, those applications which use cloud object stores also require the shortest possible request time in order to guarantee user experience. Therefore, data locality and inline processing are two of the well-known key enablers to deliver it. In this sense, we think that a technology that guarantees data locality and inline processing, at the same time that it prevents the elasticity and resource contention problems derived from the insitu computing tasks, it is necessary to keep the normal operation of an enhanced cloud object store.

1.2 Contributions of this Thesis

In what follows, we aim to relate specific contributions of this thesis. These contributions are derived from the previous posed questions.

Contribution 1: *Add a multi-tenant management layer.*

For the first contribution, we took as basis a paradigm called Software-defined Storage (SDS). SDS is specially designed to dynamically manage the storage infrastructure by means of high-level policies, at the same time that the flexibility and programmability of the storage system is enhanced. In this sense, the first contribution of this thesis is the design and implementation of Crystal, the first SDS architecture for cloud object storage systems that efficiently supports multi-tenancy and applications with evolving requirements. On the one hand, it provides a control plane for multi-tenant object storage, with flexible policies and their transparent translation into the enforcement mechanisms at the data plane. On the other hand, it provides 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. Moreover, we provide examples of policies for storage automation and I/O bandwidth control that demonstrate the design principles of our management layer.

Contribution 2: *Add self-management support for tenants.*

For the second contribution, we propose a new object management abstraction called *microcontroller*. Microcontrollers allow tenants to control objects behavior in a flexible, dynamic, and programmatic way. Moreover, microcontrollers extend

the previous work in *Contribution 1* with respect to fine-grained object management policies. While SDS enables the centralized management of storage service and infrastructure, it is rather ill-suited to manage the singular requirements of objects, which are bothersome to understand by storage administrators, but natural for tenants. Treating such particularities as regular policies could lead an SDS centralized control plane to evaluate thousands of policies at runtime, slowing down the system for critical management tasks at the bucket level. In this sense, it turns out to be more natural to let tenants self-manage their data at the object level, while leaving decision making at higher level to storage administrators. This vision can only be carried out with a powerful abstraction such as our microcontrollers. To wrap up, the second contribution of this thesis is the design and implementation of a novel distributed computing model based on microcontrollers for the flexible self-management of objects by tenants. We also provide examples of microcontrollers, such as content-level access control, object prefetching, and policy-based automated object deletion, which demonstrate the feasibility of our novel fine-grained management layer.

Contribution 3: *Scale-out policy enforcement in the data plane.*

For the third contribution, we propose a novel serverless computing platform whose abstractions are well suited for enabling the correct elasticity of the policy enforcement in the data plane. Through the usage of *serverless functions*, this new framework allows those systems that perform computations close to the data, like the previous ones built in this thesis, to better scale computing resources and to prevent resource contention problems. To do so, it offloads computing resources of the storage nodes by moving computation tasks into an intermediate computing layer, located in the storage path. This new distributed computing model allows administrator and user functions for serverless inline storage and data management. All of this, within the storage infrastructure, thus reducing data movement between storage and compute clusters needed in some of the current computing models. In this sense, the third contribution of this thesis is the design and implementation of a novel data-driven serverless computing framework that decouples the compute capabilities from the storage nodes, while ensuring elasticity, data locality and inline processing. This model enables previous contributions of this thesis to be less intrusive in the storage clusters. Finally, we provide examples of application that demonstrate the feasibility and versatility of this framework, both for storage and data management.

1.3 Outline of this Dissertation

In the following, we provide a summary of the thesis chapters:

Chapter 2: Background. This chapter provides definitions and concepts that are required throughout the thesis.

Chapter 3: State-of-the-Art. This chapter discusses the current literature regarding the main research areas related to the previous stated contributions. They include object management, software-defined storage, and active storage.

Chapter 4: Extending Multi-tenant Management. This chapter presents the first Software-Defined Storage architecture whose core objective is to efficiently support multi-tenancy in object stores. It adds a filtering abstraction at the data plane and exposes it to the control plane to enable high-level policies at the tenant and bucket granularities.

Chapter 5: Adding Self-management Support for Tenants. This chapter presents a novel framework for object stores that allows tenants to self-manage their data through the deployment of per-object management policies. With these management polices, named *microcontrollers*, it is possible to operate on the objects depending upon their state and content.

Chapter 6: Scaling out Policy Enforcement in the Data Plane. This chapter presents an innovative *data-driven serverless* computing framework for cloud object stores. It is a lightweight compute solution that allows users to create small, stateless functions that intercept and operate on data flows in a scalable manner without the need to manage a server or a runtime environment.

Chapter 7: Conclusions and Future Directions. This chapter presents the conclusions that ensue from this work and a variety of possible future research lines.

Chapter 2

Storage and Programmability: Background and Definitions

In this chapter, we aim at providing the necessary concepts and definitions to properly understand the rest of this thesis. We first give some background on the operation and architecture of storage systems, paying particular attention in the cloud object storage variant. In this sense, we take OpenStack Swift as a paradigmatic example, since together with Ceph, it may be the most adopted open source object storage system. Second, we describe the computing capabilities of object stores, describing the 'active storage' technique. Third, we illustrate the principles and concepts behind the 'software-defined' paradigm. In this case, paying special attention in the 'software-defined storage' variant, which is fundamental to understand one of the contributions of this thesis. In all cases, we overview a variety of existing systems to provide the reader with a big picture of the programmability of the storage systems.

2.1 Overview of Storage Systems

The storage systems could be divided in three different types depending on how data is stored, and the purpose of their usage, since as we describe in the following sections, each type of storage is better in different circumstances. They include: 1) block storage, where data is stored and managed as blocks within sectors and tracks of the hard disk, but no metadata providing further context. 2) File storage, where data is stored as files organized into a hierarchical file system,

alongside some few attributes, like *size* and *name* providing context. 3) Object storage, where data is stored as objects in massively scalable containers with a globally unique identifier, and with the possibility of large amounts of metadata.

Table 2.1 Summary of the main storage systems characteristics

Factors	**Block Storage**	**File Storage**	**Object Storage**
Storage Fit	Performance-based primary or secondary storage	Capacity-based secondary storage	highly reliable, cloud-scale, secondary storage
Amount of data	<500 TBs	<100 TBs or 100-500 TBs for scale-out NAS	> 500 TBs and in many cases petabytes of data
Latency	<10 ms or microsecond (flash)	Trade-off lower latency for storage simplicity	Latency-tolerant data access
Data type	Structured and unstructured data	Unstructured data	Unstructured data
Metadata	Fixed system attributes	Fixed file-system attributes	Custom metadata
Protocols	iSCSI, FC, SATA	NFS/SMB	REST over HTTP
Storage location	On premise or private cloud	On premise or private cloud	On premise, private, hybrid or public cloud
Client location	Centralized	Centralized	Centralized or geographically dispersed

In table 2.1 [14], it is summarized the main characteristics of each storage type, thus showing a comparison between them. The fields of the table show what follows: The *storage fit*, which indicates where or in what circumstances

each type of storage better fits. The *amount of data* or capacity that each one is designed for. The *latency* that clients have against the storage service. The *data type* that each one better accepts. The ability to accept *metadata*. The *protocols* needed to access the storage service. The *storage location* where the storage system is deployed. And the *client location* from where clients access the storage service, that is, *centralized*: clients are located within the same network, or *geographically dispersed*: clients access through a Wide Area Network (WAN).

2.2 Block-based Storage

Block storage is the native storage interface at the drive level, it is known as the fastest storage technology. The most common example is a Hard Disk Drive (HDD), which once formatted, it writes out and reads in blocks by their block address. In block storage data can be organized at high level into a file system or an application-specific structure. However, at low level, files are split into evenly sized blocks of data, each with its own address or block Identifiers (IDs) (e.g., sector number).

HDDs can be split into volumes of fixed block sizes. Each volume, called device, can be treated as an independent disk drive. This device offers a fixed storage capacity and can be mounted by the host Operating System (OS) as if it was a physical disk. Once mounted, it is possible to format it with a file system and start storing files on it, combine multiple devices into a RAID array, or configure a database to write directly to the block device. Some of the advantages of block storage devices are that they can be resized to accommodate growing needs, and they can be easily moved between machines. The main downside of block storage is that it has limited capability to handle metadata.

Moreover, cloud vendors have products that can provision block storage devices of any size and attach them to virtual machines. Normally, this block-based storage devices include data encryption, replication and deduplication as data management techniques.

2.2.1 Use Cases

Block-based storage is useful in a wide variety of use cases due to the high performance it offers, or its ability to dynamically create and manage volumes on HDDs. These use cases include databases and Virtual Machines (VMs).

Databases: Block storage is common in mission-critical applications that demand consistently high performance. Thus, it is ideal for databases, since a database requires consistent Input/Output (I/O) performance and low-latency connectivity.

Virtual Machines: Virtualization software use block-based storage to host the file systems for the guest operating systems packaged inside virtual machine disk images.

2.2.2 Examples and Existing Systems

The most common examples of block storage are Storage Area Network (SAN) and local disks (DASD). SANs are computer networks which provide access to consolidated, block level data storage, and always expose a block storage interface to the operating systems and client applications. SANs include 3 different layers: the *host layer*, integrated by the servers; the *fabric layer*, integrated by the network devices; and the *storage layer*, integrated by the storage devices. In SANs, the host layer contains the file system which uses the fabric layer to communicate with the storage layer. This occurs, for example, through Fiber Channel (FC) or Internet SCSI (iSCSI) protocols.

iSCSI is a TCP/IP based protocol, providing data transfer and management to remote block storage devices over IP networks. As a SAN protocol, iSCSI extends SANs across local and wide area networks (Local Area Networks (LANs), WANs and the Internet) providing location-independent data storage retrieval with distributed servers and arrays. In SANs, data is transferred, stored, and accessed on a block level. As such, a SAN does not provide data file abstractions, only block-level storage and operations. However, file systems have been developed to work with SAN software, thus providing file-level access. These file systems are known as *SAN file systems* (or *shared-disk file systems*) and they include: *Global File System 2* [15], and *IBM General Parallel File System* [16].

From the cloud storage perspective, cloud providers also offer block-based storage services. They focus mainly on providing durable and high performance block storage capacity (storage volumes) to VM. Typically, volumes are highly available and reliable, which can be attached to any running VM instance. Some of the commercial block storage options in the cloud include: *IBM Cloud Block Storage* [17], *AWS Elastic Block Storage (EBS)* [18], *Azure Premium Storage* [19], and *Google Persistent Disks* [20].

2.3 File-based Storage

File-based is the most straightforward way of storage. Data is organized through files with a name and some metadata attributes.Then, it is stored in folders under directories and sub-directories. In file storage, data is accessed as file IDs over a shared network, and the storage server manages the data on disk; On the one hand, an ID is the path of the file, which always includes the `server name` or `IP`, the `directory path`, and the `file name`, for example: `//local_data_server/documents/images/photo.jpg`. On the other hand, data can be accessed through the network by using the Network File System (NFS) protocol for Unix or Linux, or the Server Message Block (SMB) protocol for Microsoft Windows.

In file-based storage, servers use block storage with a local file system to organize the files. Thus, users only deal with the protocol and the file path, which makes this type of storage easy to use. Moreover, in file storage systems, metadata is composed by few fixed file attributes, normally stored in the file system. These system attributes include the size of file, and the dates of creation, last access and last modification. Usually, as in block-based storage, file storage systems integrate advanced data management techniques, for example, data snapshotting, encryption, replication, compression and deduplication.

2.3.1 Use Cases

File-based storage makes sense for a wide variety of scenarios, including:

Local archiving: The ability to seamlessly accommodate scalability with a scale-out solution makes file-level storage a cost effective option for archiving files in a small data center environment.

File sharing: The simplicity of file-level storage makes file sharing fit better in centralized storage servers. A single network storage appliance allows to consolidate multiple file servers for simplicity, ease of management, and space, which is ideal for this use case.

2.3.2 Examples and Existing Systems

The most common example of file-based storage is a Network Attached Storage (NAS) unit connected to a computer network. In this sense, NAS systems are networked appliances which contain one or more storage drives, often arranged

into logical redundant storage containers or RAID. For its implementation, it is possible to acquire proprietary devices (NAS-oriented software or appliances, such as NexentaStor and NetApp) or install open source software (NAS-oriented distributions of Linux, such as FreeNAS, NAS4Free, CryptoNAS and NASLite) in commodity hardware appliances.

A more advanced example of NAS is the *Scale-out NAS*. It is a type of storage that incorporates a *distributed file system* [21] that can scale a single volume with a single namespace across many nodes. Scale-out NAS file-level storage solutions can scale up to several petabytes, while handling thousands of clients. As capacity is scaled out, performance is scaled up. An example of a distributed file system is *GlusterFS* [22]. It is a software-based open source file system that aggregates disk storage resources from multiple servers, facilitating the centralized management of data through a single global namespace.

File-based storage is also offered as a service by cloud vendors. It provides simple, scalable, elastic file storage for VMs, and it is mainly used for sharing data between them in the cloud. Examples of file-based cloud storage systems include: *Amazon Elastic File System (EFS)* [23], *IBM Cloud File Storage* [24], and *Microsoft Azure Files* [25].

2.4 Object-based Storage

As the name suggests, object-based storage stores data in entities known as objects. Each object typically includes the data itself, a variable amount of metadata, and a globally unique ID, all of this stored in a flat namespace. In object stores, data is organized within *buckets*, that, at the same time, offer a way of grouping objects. Moreover, it provides larger namespaces in contrast to block- or file-based storage, which nullifies name collisions. With this guarantee, it is possible to retrieve an object from the storage service by simply presenting its unique ID, thus making information much easier to find in a large pool of data.

Object-based storage also offers much greater flexibility in terms of metadata. Unlike block and file storage, object storage allows to store large amounts of metadata alongside the data. The metadata provides information about the structure, definition, and administration qualities of the stored data. For instance, it is possible to customize metadata of the data that proceeds from certain applications, or tag the data that have similar content. In terms of infrastructure, one of the key advantages of object storage services is the scalability. The storage

system can scale with no limits to accommodate the storage capacity to the demand. Scaling out an object architecture is as simple as adding additional nodes to the storage cluster, being able to reach petabytes of storage space.

2.4.1 Variants

Object-based storage has uses in cloud storage and High-performance Computing (HPC). This different use of the storage system caused that each application domain has evolved their own object storage ecosystem. As a result, the differentiating features include access protocols, performance, security, replication, reliability, and metadata servers [26].

For example, HPC distributed file systems commonly use Object Storage Devices (OSDs) as data back-end. OSDs [27] can come in many forms, ranging from a single disk drive to a storage controller with an array of drives (disk, subsystem or appliance) that includes one or more HDDs, memory, CPU and networking capabilities. For example, the Seagate Kinetic unit or the Panasas storage blade [28]. Normally, OSDs are interconnected in a private scale-out SAN cluster through, for example, the Small Computer System Interface (SCSI) protocol. The difference between an OSD and a block-based device is the interface, not the physical media. Commonly, OSDs integrate an object storage interface [29], an extension of SCSI that implements the command set to operate over the objects, through, for example, FC, InfiniBand or iSCSI protocols.

In contrast, cloud object stores commonly use storage nodes as data back-ends, which are physical x86 servers with commodity hardware. They usually include one or more HDDs or Solid-state Drives (SSDs) for storing the actual data. Object-based cloud storage system are accessed through a RESTful Application Programming Interface (API), and through a Internet network, either LAN for local connections, or WAN for remote clouds. The RESTful API uses Hypertext Transfer Protocol (HTTP) requests to operate over the objects.

2.4.2 Easy Access through RESTful API

Simple access makes cloud object storage easy to use, and thus, widely adopted. While the communication with block- and file-based storage operates at the operating system level, object-based storage operates at application level thanks to the RESTful APIs that these services provide. APIs make objects accessible via HTTP and facilitate management functions related to authentication, per-

missions, and data properties. As a consequence, the format of an object ID includes the *server ID*, the *name of the bucket*, and the *name of the object*, for example: `http://server.url/img_bucket/photo.jpg`. Moreover, cloud object stores usually provide client applications, based on the API definitions, for easing end users the interaction with the storage servers. From uploads, going through downloads and deletions, these tools help users to manage their data.

2.4.3 Data Management

Data management is another central operational point of any distributed storage system. Like in block- and file-based storage, object stores always include data management techniques to orchestrate the storage system. It is important to note that these techniques may considerably differ depending on the system where they are built upon, and sometimes, these implementations make one service better than another. Next, we describe the main data management techniques in object stores:

Data reduction:

Deduplication: Deduplication technology shrinks the data footprint by eliminating redundant copies of data and thus reducing storage overhead. Data deduplication techniques ensure that only one unique instance of data is retained for storage. For example, a typical email system might contain 100 copies of the same 1 Megabyte (MB) file attachment. If the email platform is backed up or archived, all 100 copies are saved, requiring 100MBs of storage space. With data deduplication, only one copy of the attachment is stored; each subsequent copies is referenced back to the one saved. In this example, a 100MBs storage demand drops to 1MB, thus saving 99% of the required storage in the case of not using this data reduction technique.

Compression: Compression reduces the size of every piece of data based on some selectable algorithms. Compressing data can save storage capacity, speed up file transfer, and decrease costs for storage hardware and network bandwidth. Compression can be done standalone or in conjunction with data deduplication. The compression ratio depends on the compression algorithm and the type of data. For example, a simple text-log data object of 100MBs could be compressed with the `.gzip` format, which uses the DEFLATE algorithm [30], to around 10MBs, which represents a 90% of space saving in the storage system.

Data protection:

To maintain the desired level of data availability, object storage systems provide a degree of data redundancy. Object *replication* is, perhaps, the simplest way of producing data redundancy, being suitable for storage of small objects that are accessed frequently. Sometimes, the replication degree is configurable, however, the most common configuration is 3-way replication level for frequent access objects, and 1-way replication level for archival.

Moreover, other redundancy schemes based on *erasure codes* [31] can reduce the storage and communication costs compared to replication, thus increasing the data availability in front of possible storage failures. Erasure coding is a method of data protection in which data is broken into fragments, expanded and encoded with redundant data pieces. Erasure coding divides an object into pieces, and calculates multiple parities. In the event that the original file, or some of the pieces of it are lost, the system can use the parities and the remaining pieces to recalculate the original data.

Data placement:

Data placement is usually based on what is called *storage polices*. Therefore, a data placement policy is a pivotal element to the correct operation of the storage system. Normally, as described above, in object stores a 3-way replication level is applied when data object comes into the storage cluster. In other cases, for better data protection, it is applied an *erasure coding* algorithm, which splits the object into a certain number of blocks, eventually stored in different nodes.

In both cases, it is necessary to decide where to place the object replicas or blocks across a subset of storage nodes in order to be persisted. This inherently implies that the system should take a decision about which replicas or blocks are assigned to which storage nodes. In this sense, object storage systems use different data placement techniques, such as *hash rings*, in such a way that the data availability is ensured, and providing load balancing across all nodes.

2.4.4 Use Cases

Object-based storage is specially designed to face the huge growing of unstructured data occurring nowadays. Examples of unstructured data includes text files, music, videos, logs and images. In this sense, some ideal use cases include: big data, backup, file synchronization and sharing, and web-scale applications.

Big data: Thanks to the near infinite scaling capabilities, object storage has the ability to accommodate huge quantities of unstructured data.

Backup storage: Scalability makes object stores prepared for the massive amounts of data that typically accompany archived backups.

File sync and share: Tools like Dropbox, Microsoft OneDrive and Google Drive built on top of object-based storage services, make cloud object stores widely used to store user data. This fact makes file synchronization and sharing an increasingly adopted use case among common users.

Web-scale applications: As access to cloud object stores is through HTTP APIs, web applications fit naturally by storing the static web files (HTML) and their resources (Javascript, stylesheets, images and videos).

2.4.5 Examples and Existing Systems

As aforementioned in Section 2.4.1, object-based storage systems can be differentiated by whether they are *file systems* for HPC or *cloud-based storage systems*. Object-based file systems are usually distributed file systems. In this section we provide examples of both approaches.

Distributed File Systems

As in file-based storage, scale-out NAS can also incorporate a distributed object-based file system that can scale with a single namespace across many nodes. Normally, these file systems are designed for high performance, reliability, and manageability in HPC environments. Also, they suppoort thousands of parallel clients at over a TB/s of aggregate I/O, and providing tens of Petabytes (PBs) of storage. These file systems include: *Lustre* [32] and *Panasas Filse System* [33].

Cloud Storage Systems

Cloud object stores can be differentiated by whether they are products offered by cloud providers, or open source software packages. Typically, all of these systems offer similar characteristics, such as highly scalability, reliability, availability, durability, event notifications, cross-region replication, versioning and sometimes encryption. In this sense, commercial cloud object storage solutions include: *IBM Cloud Object Storage* [7], *Amazon S3* [8], *Google Cloud Storage* [9], *Azure Blob Storage* [10], and *EMC Elastic Cloud Storage* [11].

On the other hand, open source cloud object stores include a handful of different systems with different characteristics. However, we next provide a couple of systems, which from our perspective, are the most adopted open source implementations thanks to the large community that offers support to them. These systems are: *Ceph* [12] and *OpenStack Swift* [13].

2.5 OpenStack Swift

In this section, we provide an architectural and operational view of OpenStack Swift object store. We took Swift as a paradigmatic example of a cloud object store since it is widely adopted by the community. Moreover, the systems we designed in this thesis are all based on it due the built-in extensibility it offers in contrast of another cloud object storage systems. Thus, it is essential to know how Swift is organized, its characteristics and particularities, and how it internally works, among other aspects.

Swift is the object store project of the OpenStack [34] open source cloud computing platform. It offers cloud storage software, so that you can store and retrieve large amounts of data through its simple API (Section 2.5.2). Its built-in characteristics include scalability, durability, high availability and concurrency across the entire data set. It is designed to manage the storage of large amounts of unstructured data that grows without bound in the current days. This, in a cost-effectively way on a long-term basis, across clusters of standard server hardware. Swift is enabled for storing large amount of metadata alongside the objects. Moreover, it is defined as a highly available, distributed, eventually consistent object store.

2.5.1 Swift Architectural Overview

Figure 2.1 shows a high-level overview of the components of a common OpenStack Swift deployment. These components [35], described below, include proxy, account, container and object servers, a hash ring which determines the data placement, and some standalone services (e.g. object audition, object replication).

The first component of a Swift architecture is the Proxy Server. It is responsible for tying together the rest of the Swift architecture. It acts as a gateway against the storage nodes where the actual data is stored. Moreover, it contains a public API where the clients make the requests. For each request, it will look up

the location of the account, container, or object in the ring and route the request accordingly. The ring is a mapping between the names of entities stored on disk and their physical locations. Once built, a ring represents a ready to use storage policy, which provide a way to differentiate service levels.

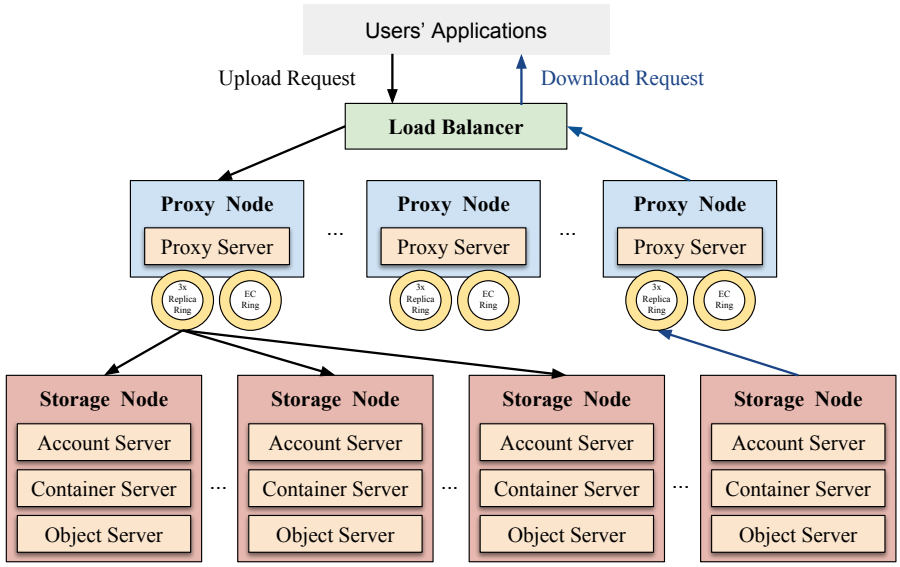


Fig. 2.1 High-level architecture overview of an OpenStack Swift deployment

On the other hand, Account Servers are responsible for listings of containers, while Container Servers primary job are to handle listings of objects. Finally, an Object Server is a very simple blob storage server that can store, retrieve and delete objects stored on local devices. Objects are stored as binary files on the filesystem with metadata stored in the file's extended attributes (xattrs). Each object is stored using a path derived from the object name's hash and the operation's timestamp.

2.5.2 Supported API Calls

OpenStack Swift provides full object lifecycle management through its RESTful API. Details about all the available calls are in [36]. Each object's access path consists of exactly three elements: `/account/container/object`. The object is the exact data input by the user. Accounts and containers provide a way of grouping objects. Nesting of accounts and containers is not supported.

With the right user permissions, Swift API allows calls for managing the Swift *accounts*, and *containers* where the data is finally stored. These calls include, for example, listing or updating the metadata of an account, or creating, listing, and deleting containers. Moreover, from the object perspective, the API allows to create, update, get and delete objects, and add, update, or delete metadata.

2.5.3 Extensibility

Extensibility is another key advantage of OpenStack Swift. Both the Proxy, Account, Container and Object servers allow to introduce *middlewares* in their storage flow. A middleware is a software that sits between the client-side request on the front-end and the back-end resource being requested. In the storage context, a middleware intercepts the requests to the storage system to run, for example, data transformations. In this sense, middlewares extend the capabilities and add new functionalities to the object storage service. Swift includes many of middlewares in its catalog [37]. For example, it is possible to introduce an *encryption* middleware in the data pipeline in order to guarantee the confidentiality of the data upon an upload request. Moreover, Swift also includes the *object versioning* middleware, which allows to keep multiple versions of the objects in case of overwrite or deletion, or the *rate limiter* middleware, which can limit the number of transactions per second at the account or container levels.

2.6 The Active Storage Technique

Nowadays, the common scenario in cloud computing is the disaggregation of the storage capacity from the compute, which among other things, eases the deployment of the services and their scalability. This approach, however, has two main drawbacks. On the one hand, when data has to be processed, the disaggregation forces to move all data from the storage cluster to the compute cluster. Finally, sometimes, the disaggregation also forces to move the results back to the storage cluster. As a consequence, this inevitably consumes a huge quantity of network bandwidth. On the other hand, disaggregation leaves in the storage infrastructure an important amount of underutilized resources (CPU and memory), since, usually, storage systems are not compute-intensive services.

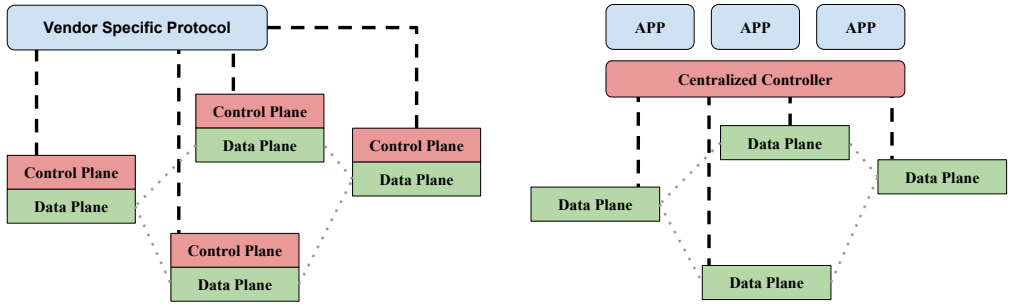
In this sense, active storage [38] is the technique proposed to reduce the bandwidth requirement, and to leverage the underutilized system resources, by

moving computation, or some parts of computation, closer to the storage servers. By offloading some computing tasks to the storage nodes, close to the data that they manage, active storage makes it possible to substantially reduce the data movement across the network and, hence, the overall network traffic. At the same time, active storage leverages CPU and memory resources that, otherwise, they would never be used. In other words, the key benefit of this technique is that by performing data processing at the source, data does not need to be moved between storage and compute clusters.

There are many active storage frameworks in the research community (see Section 3.3). One example of them, which works integrated within OpenStack Swift deployments, is the OpenStack Storlets [39] framework, developed by IBM [40]. It allows to run computation tasks where the data is, leveraging storage nodes' underutilized resources, thus reducing data movement between storage and compute clusters.

2.7 The so-called 'Software-defined' Paradigm

The 'software-defined' paradigm can be defined in general terms as that management software that operates independently of the underlying hardware. It abstracts the hardware by a layer of software, which manages the hardware, usually, in an automated manner by means of high-level policies.



(a) Traditional systems (b) Software-defined abstraction

Fig. 2.2 High-level comparison between traditional and software-defined systems

Software-defined technologies usually includes two main architectural entities, the *control plane* and the *data plane*. The control plane is the entity that orchestrates and makes intelligent decisions about the management policies. It is

mainly integrated by a centralized controller. On the other hand, the data plane is where the policies are enforced, modifying the hardware through which the data passes. While in traditional systems the control plane is collocated with the data plane (Fig. 2.2a), in the software-defined paradigm the control plane is abstracted from the data plane with a centralized controller (Fig. 2.2b).

Leading the paradigm is the Software-defined Networking (SDN) technology, built to abstract network architecture and make network devices dynamic and programmable (Fig. 2.3). In this sense, SDN suggests to centralize network intelligence in one network component by disassociating the forwarding process of network packets (data plane) from the routing process (control plane).

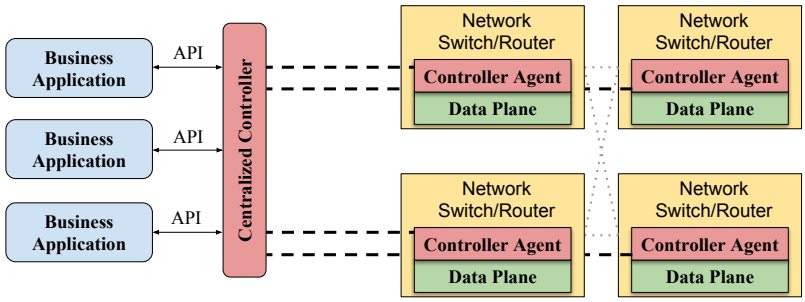


Fig. 2.3 High-level overview of the software-defined networking abstraction

Following SDN trend, they appeared more software-defined technologies for different areas of the computing. For example, the Software-defined Storage (SDS) technology is designed to abstract and orchestrate the underlying storage hardware disks, arrays, or servers through a software management layer (see Section 2.7.1). On the other hand, the Software-defined Data Center (SDDC) is designed to manage an entire data center, where services such as compute, network, storage, security and availability are pooled, aggregated and managed by intelligent policy-driven software, providing self-service, automation, and application and business management.

2.7.1 Software-defined Storage

As the name implies, SDS can be described, at high-level, as the storage that is defined by software. However, its characteristics go further of this high-level description, lacking several essential aspects of the currently intended meaning. The key cornerstone of any 'software-defined' technology is the abstraction of

the control software from the underlying hardware, which allows for centrally manage the infrastructure in an easy way. SDS follows the same paradigm, using a software-based control layer abstracted from the physical storage servers, discs or arrays. However, the actual meaning of SDS in today's context is even more complex. SDS is described as the technology that supports overall architectural definition, configuration, and operation of an storage system [41]. SDS provides optimized, and automated, storage system control and administration to allow effective and efficient resource utilization.

SDS usually uses virtualization to abstract and control the storage system. This eases the deployment and redeployment of infrastructure resources, and provides optimal alignment of available resources to users' application requirements. Virtualization enables support for multiple Service Level Agreements (SLAs). Moreover, SDS is designed to simplify the storage architecture, thus reducing specialized components and skills requirements, and it is optimized for interoperability across hardware and software platforms. Hence, it provides greater storage infrastructure flexibility to share resources while maintaining the required SLAs, allowing users to better use their data.

In short, SDS is a storage architecture for a wide variety of storage requirements based on a set of software and hardware components, dynamically configurable to meet customers' workload requirements. To fully realize the potential of this technology, SDS implementations should incorporate these characteristics:

- **Programmatically administered**: Programmable interfaces to support dynamic storage deployment, configuration and management, enabling policy-based automation of infrastructure storage resources.
- **Automation**: Realization of autonomic data storage capabilities (provisioning, reconfiguration, etc.) to provide dynamic SLA configuration.
- **Monitoring**: Metric collection for the automation of the storage in order to guarantee and validate that the users SLAs are met.
- **Scalability**: High scalability of the storage virtualization, and pooling, which is essential to adapt the storage resources to the demand.
- **Interoperability**: Generic storage infrastructure and control components. The abstraction of functionality across underlying hardware eases systems integration and configuration of infrastructure components.

Chapter 3

State of the Art

This chapter aims at bringing the reader closer to the concrete research problems that motivate this thesis. As described below, we identified three main research areas: *data management and processing in cloud object stores*, *software-defined storage* and *active storage*. We discuss, for each one, the current state of the available systems and research works.

For **Contribution 1** (*Add a multi-tenant management layer*), we review Software-defined Storage (SDS) technologies, designed to manage the storage system automatically. Moreover, we also investigate the current state of the active storage research works, as SDS could derive in computation tasks placed close to the data, such as compression, encryption or any other data transformation process required to manage the storage system.

For **Contribution 2** (*Add self-management support for tenants*), we review the current state of the data management and processing by tenants in cloud object stores. Furthermore, we also review SDS technologies to consider if their abstractions can help to overcome this challenge. As object-based management policies could require computation tasks over the objects, it is also interesting to review the current state of the active storage systems and research works.

For **Contribution 3** (*Scale-out policy enforcement in the data plane*), we review active storage technologies, paying special attention to whether they take care about the resources of the storage cluster. Moreover, it is also interesting to investigate the current state, scope, and limitations of an emerging technology called *serverless computing*, especially in what refers to managing and processing objects from a cloud object store.

3.1 Data Management and Processing

In our attempt to enhance the programmability of cloud object stores, we first review the current state of these systems in what refers to object management and processing by tenants, since object stores are the main scope of this thesis. In this sense, as these systems offer object management and processing techniques in different ways, we classify them in two main groups: First, those that a tenant can perform directly to the object store, as standalone service: *Internal Management*. And second, those where a tenant needs the involvement of other cloud services to perform actions over the objects: *External Data Processing*.

3.1.1 Internal Management

Thanks to the potential of object stores, all of the big players in cloud services offer this type of systems in their service catalog. These systems, described in Section 2.4.5, provide lifecycle management of objects in different ways. For example, Amazon S3 [8] allows changing the replication level and the replica placement (storage class), and setting expiration rules to the objects by means of simple policies. Similar to S3, EMC Atmos [42] allows the introduction of customizable metadata to determine the placement and the protection of the data. Following this trend, object management in Google Cloud Storage (GCS) [9] refers to lifecycle actions, which as before, they can change the storage class, set expiration dates to objects, or even delete them. To this end, GCS has a set of limited lifecycle conditions (e.g. age of the object) [43] that, once met, they can run these actions. In Microsoft, as in GCS, the Azure Blob Storage [10] lifecycle management consists of rule-based policies which can be used to move data to the best access tier or storage class, and to expire data at the end of its lifecycle.

Regarding open-source solutions, in systems like Ceph [12] and OpenStack Swift [13], the object lifecycle management by tenants is limited to set the expiration time of objects, normally based on a specific date or a certain number of days after object creation. However, open source systems are usually extensible in this sense. This is the case of Swift, which allows to introduce middlewares [37] in the storage path (see Section 2.5.3). This approach allows storage administrators to inject new ways of data management, such as object versioning or data encryption, among others. Moreover, middleware extensions enable to integrate more powerful systems for processing data objects, commonly cataloged as active storage frameworks. As active storage represents by itself a research field, we

extensively discuss these systems and others in Section 3.3. On the other hand, research works are focused on the security of the objects. For instance, in [44], the authors propose a novel content-level access control for objects in Swift. It allows filtering the content of an object depending on who is accessing in each moment, although the system only allows processing JSON files. Other researchers focused their efforts on handle data sharing security in cloud object stores [45–48], based on different techniques, such as through the use of public-key encryption.

3.1.2 External Data Processing

Traditionally, processing data from cloud object stores has been done in disaggregated clusters with compute capabilities. The main advantage of the disaggregation is that it ensures the correct scalability and elasticity of the system. Thus, from common on-premise servers to virtual machines, compute clusters allow tenants to deploy software that can process objects from a cloud object store. For example, the virtualization model consists of creating a VM, selecting the desired flavor (CPU, memory and hard disk size), and installing a guest operating system on it. Once ready, then it is possible to install all the required software to process objects from an object store. Examples of these systems include Amazon EC2 [49], and IBM Cloud Virtual Servers [50]. However, the main drawback at this level of abstraction is that tenants are responsible to install and maintain all the servers or virtual machines, the operating system, and the required software.

Serverless Computing

Nowadays, a new technology called *serverless computing* emerged to run computation tasks in an easy way. It is based on a computing abstraction called *function*. A function is a piece of code that runs as a reaction to an event triggered by a cloud service, such as the upload of an object to an object store. This abstraction greatly simplifies running computing tasks over objects. Unlike traditional models, in serverless computing there is no need to manage servers or virtual machines, all of this is automatically provisioned by cloud vendors, and tenants just have to worry about the code of the function.

Following disaggregation trend, highly scalable serverless platforms such as AWS Lambda [51], and the open-source Apache OpenWhisk [52] deployed in IBM Cloud Functions [53], are becoming very popular these days to run asynchronous computing tasks over disaggregated object stores like Amazon

S3 or IBM COS, respectively. Even recent research works are using serverless computing frameworks for data-intensive tasks over disaggregated object storage services [54–60]. Moreover, to complement and add more value to serverless computing frameworks, Amazon, IBM, and other cloud providers offer a service called *API Gateway*. API Gateway enables to easily expose functions as RESTful endpoints. It is a mechanism to synchronously call functions, which might read data from an object store and output transformed content to the client.

Other technologies have appeared that tap into the simplicity of serverless computing. One of these is Amazon Lambda@Edge [61] which allows running functions near to the users. In this case, Edge functions are mainly designed for header and metadata manipulation, and for lightweight computations and data transformations. The main application of this technology is to accelerate those use cases that do not require strictly communication with the back-end, thus providing low-latency services. For example, web site data, which is commonly stored in a cloud object store, can benefit from these edge functions to dynamically adapt their content without the need to communicate with the storage back-end in each request. On the other hand, some research works propose novel systems to perform data analytics at the edge [62, 63].

Interactive Queries

Inevitably, there is an still increasing adoption of cloud object stores as data back-ends, mainly motivated for their built-in characteristics, such as simplicity, high scalability and low costs. In this sense, nowadays it is very common to put large datasets directly to cloud object stores, instead of loading them into databases. Thus, because of the need to run analytics, new services have emerged to process this data from cloud object stores [64, 65].

Commonly, most of these services are specially designed to query data from cloud object stores in an SQL-like fashion. For example, IBM Cloud SQL Query [66] is an interactive query service that makes it easy to directly analyze data on IBM Cloud Object Storage (COS) using standard SQL. It means that it is possible to store data on IBM COS and query that data as with an SQL database. This approach is similar to Amazon Athena [67] that operates over S3. Moreover, Amazon Redshift Spectrum [68] or Facebook Presto [69] can also provide interactive queries over large scale object repositories like S3. In particular, Presto offers SQL interactive queries where all processing is in memory

and pipelined across the network between stages. Usually, these service are serverless, meaning that there is no infrastructure to manage, no setup, servers, or data warehouses.

3.1.3 Discussion

Reviewing the most important cloud object stores, we can see that these systems, as standalone services, have a limited set of customizable characteristics regarding the objects' lifecycle management. In most of these services, tenants can only set expiration times for the objects. On the other hand, those solutions that offer multiple storage tiers (e.g. data archival) also allow tenants to move data between them, either manually or by means of policies. This permits to change the replication level and the replica placement of the objects. Normally, cloud object stores are lightweight services deployed on commodity hardware servers. With this fact, one can infer that adding new capabilities in the storage system may inevitably increase the resources usage, which may derive in an inconsistent storage operation.

Computing on objects from cloud object stores has always been done in disaggregated compute clusters. However, the complexity of the traditional model, where computation takes place on physical servers or virtual machines, is clearly huge for those non-experts tenants, as they are responsible to install and maintain all the software stack. For adding simplicity to the cloud computing area, a new computing abstraction called serverless computing has emerged recently. Serverless platforms like AWS Lambda are mainly designed for asynchronous event-driven computing tasks over disaggregated storage resources. Concretely, it is only possible to launch functions on the HTTP PUT event, that is, when an object is uploaded. For example, imagine a simple *image resizing* function. In this case, when an image is uploaded to S3, the upload event triggers a Lambda function that resizes the image. All of this is done in a disaggregated compute cluster, eventually storing the resulting resized image back to the object store. On the other hand, API Gateway allows to proactively invoke functions. This mechanism can be useful for a number of applications. But also, it incurs extra overhead, as functions are not in the read/write path from/to the object store. In contrast, in interactive queries frameworks, the pipelined execution avoids unnecessary I/O and associated latency overhead. However, these systems cannot provide generic function computations since they are focused on a specific task.

Unfortunately, disaggregation means intensive network usage. Currently, serverless platforms do not support interception of cloud object storage requests and their corresponding inline processing. As a result, this approach does not allow managing the complete lifecycle of objects, which is what we pursue to overcome the challenges of this thesis. Nevertheless, serverless computing principles and abstractions may be a good approach in our intent to *scale-out the policy enforcement in the data plane*.

Open source cloud object storage solutions, as it is obvious, allow more flexibility in terms of extensibility. In addition to all we aforementioned about object lifecycle management and object processing, open source permits researchers to create frameworks that can extend the programmability of object stores like Swift. Some of these systems, as active storage frameworks, extend the capabilities of the storage systems allowing tenants to perform computation tasks directly within the storage cluster, where the data is (see Section 3.3). Thus, we can conclude that, despite all of the computing abstractions we reviewed in this section, either on-premise computing, VMs, functions, or active storage tasks, cloud object stores, as standalone services, lack of the necessary flexibility on the programmability, concretely on the storage and data management.

3.2 Software-defined Storage

Software-defined Storage (SDS) is nowadays the main paradigm in what refers to the storage management. Similar to what happened with Software-defined Networking (SDN) [70, 71], adopting software-centric, policy-based management models for storage systems is increasingly accepted as an effective technique to reduce storage management costs [72]. For this reason, in recent years SDS has been applied to different storage substrates, such as file-systems, block storage, and object stores. In this section, we study some of these SDS systems, grouping them in two categories. On the one hand, the *Commercial SDS Systems*, and on the other hand, the *Research SDS Systems*.

3.2.1 Commercial SDS Systems

In the industry, most systems provide a simple rule-based policy model. For instance, EMC ScaleIO [73] enables administrators to build pools of hardware-specific storage (e.g., volume $p \rightarrow SSD$) and IBM Spectrum [74] offers to adjust

the data redundancy scheme enforced on a particular storage volume (e.g., volume $v \rightarrow 3$ replicas). Normally, these commercial products offer a graphical interface for administrators to define policies.

Regarding automation, such products have focused more intensely on solving the resource automation problem in block storage, which has traditionally been one of the most time-consuming administration tasks in multi-tenant storage platforms. Among others, systems like EMC ScaleIO, VMWare vSAN [75, 76], IBM Spectrum, Nutanix ECP [77], EMC ViPR [78, 79], or its open-source branch named CoprHD [80] offer advanced storage virtualization technologies that enable administrators to aggregate new hardware to virtual storage pools, create containers and volumes on top of them, and specify the static data layout properties such as redundancy (e.g., geo-replication, coding). Often, the automation capabilities of these systems also encompass the configuration of the network within client VMs and the storage back-end.

Many commercial SDS products also enable administrators to set up data volumes with specific services to be executed on the data. For example, VMWare vSAN can execute inline deduplication, compression, and data encryption at rest on volumes if necessary. Nutanix ECP and IBM Spectrum also incorporate data services such as data protection and configurable availability policies. To deliver such computing services on data, commercial SDS products provide a proprietary software layer that implements such features and/or exploits some native functionalities already built-in in storage appliances.

On the other hand, despite that many SDS products integrate monitoring services of the storage system, dynamic provisioning is a less common feature. Only IBM Spectrum seems to offer features related to automatically scaling cluster resources depending on the demand and performance of workloads [81]. Other commercial systems, such as OpenIO [82], perform dynamic orchestration of computing tasks on top of the most suitable storage resources to optimize workloads. OpenIO builds a control loop that monitors the storage resources to execute an efficient scheduling decision.

3.2.2 Research SDS Systems

In the research community, software-defined storage has been studied in a plethora of works. However, in this section we review some of those systems that provide full SDS capabilities on storage clusters.

IOFlow [83] was the first research work presenting a complete SDS architecture. IOFlow enables end-to-end (e2e) policies to specify the treatment of I/O 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. In this sense, IOFlow provides rule-based policies to enforce potentially complex routing primitives (create/remove queues, configure token bucket) on specific storage flows at the data plane. The automation capabilities of IOFlow are mainly related with data services that can be implemented in the form of "stages". Authors demonstrate the flexibility of IOFlow with two use cases: distributed bandwidth control, which demonstrates the coordinated flow control capabilities across several stages, and a malware scanning service that requires stage I/O routing capabilities. Thus, the first use case represents an example of dynamic resource provisioning, whereas the second use case can be classified as static data service automation. IOFlow requires changes at the storage server and the hypervisor level to intercept flows of I/O requests, meaning that it is not designed to be system transparent. The control plane of IOFlow is centralized but extensible, as it can accommodate new algorithms or bandwidth policies to control I/O routing across the different stages. Accordingly, the data plane is extensible, as new stages can be added and discovered by the control plane.

sRoute [84] presents an advanced platform for executing computing services on data flows based on sSwitches (an evolution of the "stages" notion in IOFlow). The control plane can enforce rule-based policies that describe forwarding rules to route IOs from clients to the storage back-end through an arbitrary number of sSwitches. By intercepting and classifying I/O flows, sSwitches may embed data services such as customized replication (static data service automation) or tail latency control (dynamic provisioning), to name a few. The control plane of sRoute is extensible as it can accommodate new algorithms to enforce policies, such as bandwidth control or caching [85]. Interestingly, sRoute proposes the notion of "delegate function" as a mean of distributing control rules across data plane stages, thus partially decentralizing the control plane. At the data plane, I/O flows can be intercepted at several sSwitches to perform operations on them, normally related to the classification, redirection, and prioritization of block requests without impacting client VMs. The data plane of sRoute is extensible, as new sSwitches can be integrated in the data plane following an API. In this sense, while sRoute still requires modifications at the storage stack to add new sSwitches in a target storage server (which prevents system transparency), a

difference with IOFlow is that sSwitches may also reside in other servers. This means that some sSwitches do not require code changes on the target storage system as they are added as external services where I/O flows may be routed.

Finally, **Retro** [86] is a framework for implementing resource management policies in multi-tenant distributed systems. Retro is not classified as SDS, although it may be, since 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. Thanks to a flexible DSL, in Retro administrators can program complex policies that achieve resource fairness or latency guarantees. Retro is focused on solving dynamic resource management problems across multiple tenants, including typical problems of storage management (bandwidth, latency, etc.). However, it is not designed to offer other storage automation or provisioning services such as configurable data layouts or computations on storage requests (e.g., compression, encryption) commonly found in SDS systems. Retro's control plane is extensible not only with new programmable policies, but also with new resources added at the data plane. In this sense, Retro enables "control points" to be easily incorporated in a system for managing generic system resources thanks to AOP interception. This means that the target system does not require changes in its code and it remains oblivious to the fact that a certain resource is being orchestrated by Retro (system transparency). Moreover, this makes Retro capable of controlling different target systems (HDFS, HBase, etc.) instead of being tailored to a specific one.

3.2.3 Discussion

As we introduced in Sections 2.7.1, the absence of a common definition and standard for SDS has promoted the development of diverse system designs with different conceptions and objectives [87–89].

Reviewing most commercial SDS systems, we may conclude that they are mainly focused on storage provisioning and virtualization for block-based storage systems. In general, commercial products do not provide system transparency. That is, their main objective is to make it easier for administrators to virtualize, configure, and provision storage resources and services. However, to achieve this objective, such SDS products require installing a proprietary storage stack containing the SDS functionality. Such a stack does not only involve the storage back-end; in some systems, it is also required for client VMs to install a proprietary

software to operate against the storage back-end. Regarding dynamic provisioning, only IBM Spectrum offers automatic scalability based on the demand. Perhaps, due to its complexity, these products are still far from incorporating approaches already existing in the literature related to SLO-based auto scaling [90–93]. Similarly, extensibility is a property not present in the aforementioned systems, as they offer a packaged software instead of a platform to develop new data services or control algorithms on top of it.

On the other hand, research SDS systems are centered around the concept of I/O interception and dynamic provisioning. Whereas commercial systems lack of publicly available technical information about their internals, SDS research works provide more technical specifications about them, thus allowing us to better analyze the particularities of each one. In these sense, we reviewed three different SDS systems which provide policy-based storage automation. Each one designed for slightly different purposes, such as I/O interception in IOFlow, or multi-tenant dynamic resource management in Retro. However, as in the case of commercial systems, these works are also focused on providing SDS capabilities for block storage devices, mainly used by virtual machines. Although the scope of this thesis differs from block-based storage, the characteristics and abstractions of these systems may provide a helpful perspective for our research challenges.

To conclude, we reviewed how SDS systems provide, in a greater or lesser extent, transparency, extensibility, programmability, I/O interception, and dynamic provisioning. We think that they are all essential in a complete SDS solution, and should be present in our system. Moreover, as we already discussed, all of the previous SDS systems are mainly focused on block-based storage systems, and unfortunately, their characteristics are not directly applicable to object-based stores. In this sense, in the first contributions of this thesis we present the first multi-tenant SDS framework for cloud object stores.

3.3 Active Storage

Moving computation close to the data to benefit from data locality is nowadays a commonplace idea. In databases, stored procedures and co-processors [94–96] have standard interfaces, and have already been in use in many production environments. Moreover, the early concept of *active disk* [97–100], that is, a HDD with computational capacity, was borrowed by distributed file system designers in HPC environments two decades ago.

In the context of storage systems, computation close to the data has been studied as active storage. Active storage [97] has led to a plethora of research works proposing platforms for different storage systems [101–104]. For example, for object-based distributed file systems and for cloud-based object stores, which we study in this section. Many of these works proved significant data transfer reductions and optimizations derived from data locality.

3.3.1 Object-based Distributed File Systems

Most of the works in active storage have been focused on the context of object-based distributed file systems, which use Object Storage Devices (OSDs) as data back-end [105, 106]. For example, [38] presented an active storage implementation integrated in the Lustre file system that provides flexible execution of code close to data in the user space. In another work [107], the authors created a prototype on top of a parallel file system (PVFS) to carry out data analytic computations as part of I/O operations. This work is focused on HPC environments, using the Message Passing Interface (MPI) [108] on the client side for communication. In [109], the authors created an active storage framework for OSDs on top of the Panasas file system, which allows to run sync/async tasks over the objects. This framework uses VM engines to execute tasks downloaded from client applications.

Regarding the self-management of data by tenants, in [110–112], the authors proposed a framework for file systems and object stores that consists of a programmable storage stack through which all the files pass before storing them into disk. This stack is extensible by adding plugins, including compression, encryption, snapshots, redundancy and integrity checking. Although the authors do not explicitly define the framework as active storage, these plugins are in fact active storage tasks since they run close to where the data sits. To take benefit of the storage stack, users can activate the plugins by means of file attributes. Attributes are set on each file and each directory to convey storage policy decisions to the storage system. Moreover, the system allows users to modify the files' attributes, thus permitting users to have full control of their data, not only during the creation, but also throughout the existence of the file.

Resource aware systems

While the research works discussed above propose active storage frameworks for different distributed file systems, a common point in all of them is the lack of

resource management (CPU and Memory), when running active storage tasks over the objects. They all argue that offloading computation close to the data improves the overall computation execution time, reducing significantly the data movement between clusters. However, most of these systems are not aware about the available resources in OSDs. Existing studies have neglected the impact of resource contention when processing concurrent request I/O operations by the same OSD, which happens frequently in practice.

In this sense, Oasis [113] enables users to transparently process OSD objects, and supports different processing granularity: at per-volume or per-object levels. Oasis can partition computation tasks between the client host and the OSD dynamically, depending upon the OSD workload, which is not a common approach in active storage research works. This feature allows preventing somehow the resource contention in OSDs. On the other hand, in [114] the authors analyzed the impact of resource contention on active storage systems. Based on their analysis, they created a framework called *Dynamic Operation Scheduling Active Storage* (DOSAS), which dynamically offloads the active processing operations between storage and compute nodes, according to the state of the system environment. They built the framework using the PVFS2 [115] parallel file system. Finally, by evaluating their architecture, the authors observed that resource contention is a critical problem for active storage systems.

Policy-based systems

Most of the research works in active storage are based on the request-driven model, where the tasks are executed upon an object request to an OSD, intercepting the data-flow of the object. In the request-driven model, active storage tasks can be executed in three different circumstances: 1. Implicit invocation, where the users are not aware of the active storage tasks enabled by a system administrator; 2. Associative invocation, where the users have to associate the objects to the desired active storage tasks. Then, upon an object request, the task is always executed; and 3. Transient invocation, where the users have to explicitly specify the active storage task to run in the object request.

Altough the request-driven model allows to make computation close to the data upon an object request, in [116] it is introduced the concept of *policy objects* to extend the functionality of OSDs. In this work, the authors proposed a hybrid approach to combine request-driven and policy-driven models. With policy-driven

model, and through the usage of policy objects with an associative approach, it is possible to specify under which OSD conditions active storage tasks are executed over the objects. This work is focused on the *self-management of storage*, and tasks are executed offline, like batch applications, when the conditions of the policy objects are met. A policy object is the set of conditions and can be evaluated as a Boolean value. The system allows to associate a method with one or more policies, but a policy can be associated with only one method. This policy-driven model keeps the associations between policy objects and active storage tasks in a centralized table, and the system has to query this table each certain period of time to evaluate the conditions of policy objects.

3.3.2 Cloud Object Stores

The literature shows how active storage has been well studied in the context of object-based distributed file systems. However, it should be noted that, as we already discussed in Section 2.4.1, these distributed file systems differ from cloud object stores on how the data is accessed (for example, iSCSI vs. HTTP), and their internal architecture. So, although they store the data as objects, they are different variants of object-based storage. In this sense, there are some research works that propose active storage frameworks for cloud object stores. In particular, we review those ones proposed for OpenStack Swift.

On the one hand, we have the **OpenStack Storlets** framework. It extends Swift with the ability to run user-defined computations, called Storlets, close to the data, in a secure and isolated manner [40, 39]. A Storlet is a compiled and packaged code, currently in JAVA or Python programming language, that can be uploaded to Swift as any other object. Once uploaded, Storlets can be proactively invoked over data objects, that is, the user must explicitly indicate in the object request headers which Storlet to run every time. In this sense, one single Storlet at a time is allowed to execute upon an object request. Storlets framework intercepts the data pipeline and run computations inline, modifying the data-flow when the objects are uploaded or downloaded from Swift. Storlets run within containers, thus providing of the sufficient program isolation and a guarantee of security that any arbitrary code will run safely without compromising the storage system. Moreover, Storlets may run both on proxy and storage nodes.

On the other hand, there is the **ZeroVM** framework. It is a lightweight container-based virtualization platform that provides deterministic process execu-

tion and isolation [117]. ZeroVM consists of the virtualization of applications, to then place them where the data sits, thus leveraging data locality. This provides the ability to transform or process data in situ, instead of moving data to where the application is. The main difference with Storlets is that in ZeroVM applications run asynchronously, upon an explicit invocation (like stored procedures or batch tasks). In contrast, Storlets run synchronously, processing the data stream of the requested object. As in the case of the Storlets, two of the key benefits of running applications on containers are: 1) ZeroVM instances can start in around 5ms. This means that the system is very quick in the application spawning phase. And 2) ZeroVM applications are isolated and secured within containers, so it is not possible to interfere with the storage system.

3.3.3 Discussion

Active storage may help to improve the self-management of data by tenants of object stores. Although active storage is designed to perform computations close to the data, we can leverage the abstractions proposed by these systems to run object management techniques over the objects in a cloud object store.

In the context of distributed file systems, we reviewed some works that enable active storage capabilities in object stores. However, most of them are designed to be transparent to the users, who are not aware about the tasks that are going to be applied. Other systems allow the involvement of users, who can proactively invoke the appropriate active storage tasks over the objects. Although it is a step towards the self-management of data, these tasks are focused on performing data transformations, and not for data management. In the context of cloud object stores, Storlets and ZeroVM frameworks enable OpenStack Swift deployments to perform computation close to the data in different ways. They are a good point to start our research, and provide a good vision on how active storage is managed in cloud object stores. However, most of these systems, both for distributed file systems and cloud object stores, follow the request-driven model, where active storage tasks are executed upon an object request, or by proactivelly invoking the task itself. Despite this, we found one research work where active storage tasks are executed when some conditions are met (policy-driven) [116]. Although the authors propose a batch-oriented model for self-managing an OSD-based storage system, its abstractions may provide a good perspective on how to overcome the research challenge of adding self-management support for tenants.

Nevertheless, one drawback of these systems is that they can inevitably produce resource contention in the storage cluster. In this sense, distributed object-based file systems and cloud object stores follow similar approaches. For example, ZeroVM internally determines which machines in the cluster contain a replicated copy of an object and then randomly chooses one to execute a ZeroVM process. On the other hand, in OpenStack Storlets, an Storlet application is executed in one of those servers which contain a copy of the object, or alternatively, in a proxy server. While all of these systems work well as a proof-of-concept for data-local computing, it is quite possible for active storage tasks to be spread unequally across the storage cluster, thus resulting in an inefficient use of the available compute resources, and even worse, producing resource contention against the storage service. In the challenge of *scale-out the policy enforcement in the data plane*, we pursue for solve these problems. In this sense, we reviewed some interesting related research works that provide different ways for facing the resource contention [113, 114]. These works tried to solve resource contention issues by limiting the execution of active storage tasks, or by delegating them to external services. They advocate to run tasks both at the server side and/or at the client side, depending on the state of the storage cluster. Anyway, they cannot solve the scalability problem that makes these systems less attractive for elastic cloud settings.

By analyzing the cloud object storage catalog, we can see that cloud providers do not offer active storage capabilities in their cloud object storage services. At this point, one may infer that active storage has inherently some downsides that preclude its general applicability in the Cloud: 1. Compute tasks are limited by the available resources at storage nodes, which hinders scalability; and 2. It requires of resource management to limit access to scarce or sensitive resources at storage nodes. As a consequence, it is more than likely that these problems regarding scalability and resource contention have prevented cloud providers to adopt active storage techniques in their object storage services. They advocate for the disaggregation of compute from storage, which provides simplicity and better scalability. Moreover, the fact that these services are proprietary solutions has prevented researchers to investigate active storage solutions for them. This final reflections provide a good perspective on how we have to overcome the challenges of this thesis. Specially, in the third contribution, since we want to provide compute capabilities within the storage infrastructures, at the same time that we mitigate scalability and resource contention problems.

UNIVERSITAT ROVIRA I VIRGILI
ENHANCING THE PROGRAMMABILITY OF CLOUD OBJECT STORAGE
Josep Sampé Domenech

Chapter 4

Extending Multi-tenant Management

4.1 Introduction

Despite their growing popularity, cloud object stores are not well prepared for heterogeneity. Usually, they use static configurations without distinction between tenants with time-varying requirements [119, 120]. This inevitably causes all tenants to share the same service level, even when the different workloads from the different applications can dynamically vary over time. For example, the storage requirement of a social network can dramatically differ from a big data analytics application. While a social network, such as Instagram, stores large amounts of small-medium sized photos and videos (KB- to MB-sized objects), a big data analytics framework would probably be a read-intensive application for large files (GB-sized objects).

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 infrastructure to eliminate data movement and enable in-place analytics capabilities [121, 119], thus benefiting of the data placement. Supporting such a variety of requirements in a cloud object store is challenging, because in current systems, normally custom functionalities

The results presented in this chapter are published in [118]

are hard-coded into the storage system implementation due to the *absence of a true programmable layer*, making it difficult to manage and maintain as the system evolves.

4.1.1 Scope and Challenges

In this chapter, 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 [83, 122, 85, 123, 84]. 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 I/O 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 [83, 85, 84]. 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 [83, 85, 84], 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.

4.1.2 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 I/O 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 4.5.2). 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 I/O 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 [124]. 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 4.5.1). 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 I/O 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.

4.2 Crystal Design

Crystal seeks to efficiently handle workload heterogeneity and applications with evolving requirements in a shared object store. To achieve this, Crystal separates high-level policies from the mechanisms that implement them at the data plane, to avoid hard-coding the policies in the system itself. To do so, it uses three abstractions: *filter*, *inspection trigger*, and *controller*, in addition to *policies*.

4.2.1 Abstractions in Crystal

Filter. It is a piece of code that a system administrator can inject into the data plane to perform custom computations inline, on incoming object requests. 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. 4.1). 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.

	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

Fig. 4.1 Structure of the Crystal DSL

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 I/O 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 If-This-Then-That (IFTTT) service [125]. This service allows users to express small rule-based programs, called "recipes", using *triggers* and *actions*. For example, a simple compression policy could be set as in Listing 4.1.

Listing 4.1 Example of a trigger, action and recipe

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

An IFTTT-like language can reflect the extensibility capabilities of the I/O 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. 4.1 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

UNIVERSITAT ROVIRA I VIRGILI
ENHANCING THE PROGRAMMABILITY OF CLOUD OBJECT STORAGE
Josep Sampé Domenech

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.

4.2.2 System Architecture

Fig. 4.2 presents Crystal's architecture, which consists of a *control plane* and a *data plane*.

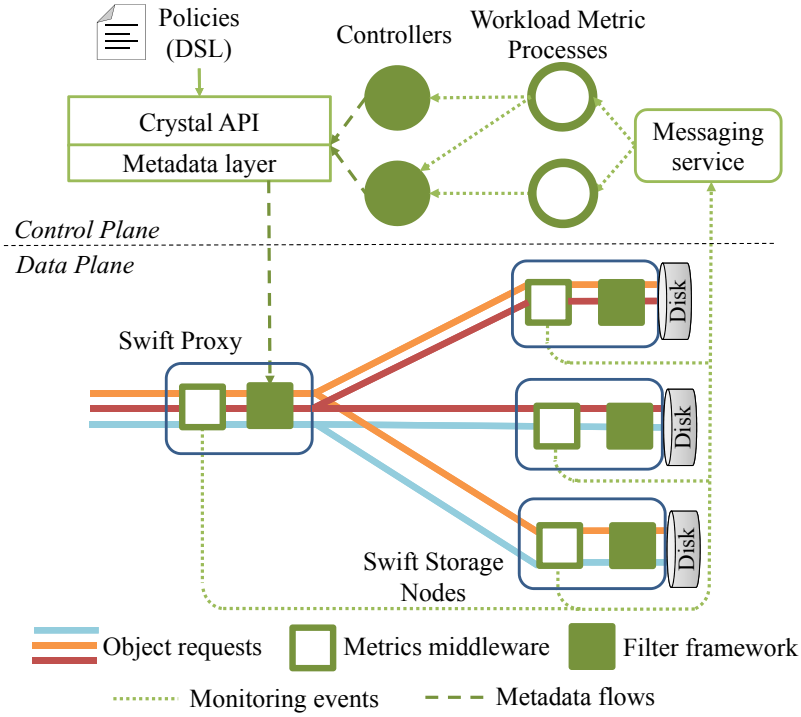


Fig. 4.2 High-level overview of Crystal's architecture on top of OpenStack Swift

Control Plane. Crystal provides administrators with a system-agnostic Domain-specific Language (DSL) 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 lifecycle and metadata of controllers, filters and metrics (see Table 4.1). Moreover, the control plane is built upon a distributed model. Although logically centralized,

the controller is split into a set of autonomous micro-services, each running a separate control algorithm. Other micro-services, called *workload metrics*, 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 metrics 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.

4.3 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 the distributed controllers.

4.3.1 Crystal DSL

Crystal's DSL hides the complexity of low-level policy enforcement, thus achieving simplified storage administration (Fig. 4.1). Its structure contains:

Target: The target of a policy represents the recipient of a policy's action, and it is mandatory to specify it on every policy definition. To meet the specific needs of object storage, targets can be *tenants* and *containers*. This enables high management and administration flexibility.

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

Action clause: The action clause of a policy defines how a filter should be executed on an object request once the policy takes place. The action clause may accept parameters after the `WITH` keyword in form of key/value pairs that will be passed as input to customize the filter execution. Retaking the example of a compression filter, we may decide to enforce compression using a `gzip` or an `lz4` engine, and even their compression level. To cope with object stores formed by proxies/storage nodes like 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. 4.1).

The vocabulary of our DSL can be extended on-the-fly to accommodate new filters and inspection triggers. That is, in Fig. 4.1 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. Moreover, the Crystal DSL has other features:

1. *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)

2. *Pipelining* several filters on a single request (e.g., compression + encryption) ordered as they are defined in the policy, similar to stream processing frameworks [124]

3. *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.

Table 4.1 shows the available API calls to Crystal, which include calls to manage the Crystal controller, calls to manage the filter framework, and calls to manage the workload metrics. In this sense, Crystal offers a DSL compilation service via API calls. Crystal compiles simple automation policies as *target→filter* relationships at the metadata layer. Next, we show how dynamic policies, with `WHEN` clause, use controllers to enforce filters.

Crystal Controller	**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 lifecycle of distributed controllers and workload metric processes in the system.

Filter Framework	**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 Metrics	**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 4.1 Main calls of Crystal controller, filter framework and workload metrics management APIs

4.3.2 Distributed Controllers

Crystal resorts to distributed controllers, in form of micro-services, which can be deployed in the system at runtime to extend the control plane [126–128].

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

automation controllers (e.g., `P2` in Fig. 4.1). Their lifecycle 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. 4.1). To this end, the base global controller class encapsulates the logic i) to ingest monitoring events, ii) to disseminate the computed assignments across nodes[1], and iii) to get Service-Level Objectives (SLO) to be enforced from the metadata layer (see Table 4.1). This allowed us to deploy distributed I/O bandwidth control algorithms (Section 4.5).

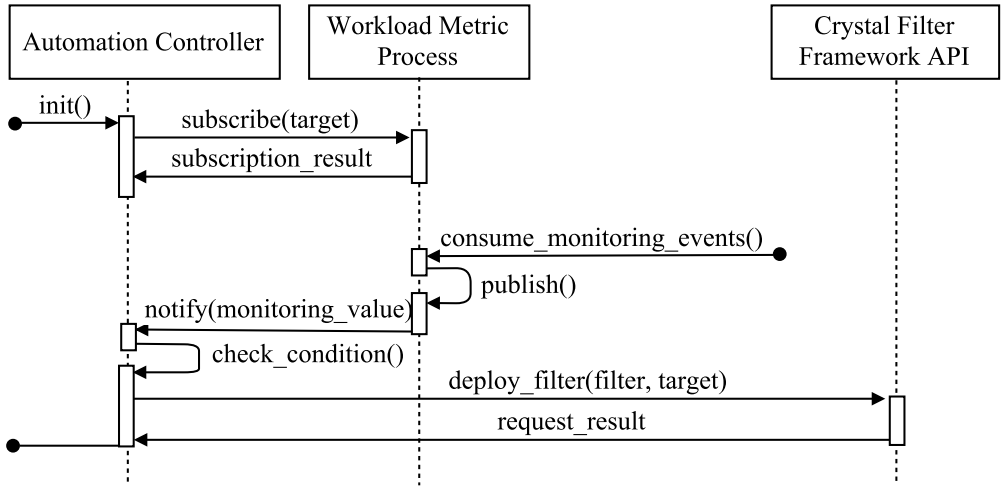


Fig. 4.3 Interactions among automation controllers, workload metric processes and the filter framework

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

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

UNIVERSITAT ROVIRA I VIRGILI
ENHANCING THE PROGRAMMABILITY OF CLOUD OBJECT STORAGE
Josep Sampé Domenech

simplicity [128], 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 [129]. For instance, Fig. 4.3 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.

4.4 Data Plane

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

4.4.1 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 4.4.2) 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 4.1). Our metrics framework allows developers to plug-in metrics that inspect both the type of requests and their contents (e.g., compressibility [130]). We provide the logic to abstract developers from the complexity of request interception and event publishing.

4.4.2 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 [97, 38] 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 [131]. 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 such as compression, which change the content of data objects, may receive a special treatment. 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 [39] 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 [132]. 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 lifecycle, 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.

4.5 Extending Crystal

In this section we describe the benefits of Crystal's design by extending the system with new data management filters, and a novel distributed control of I/O bandwidth for OpenStack Swift.

4.5.1 New Storage Automation Policies

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

Listing 4.2 Example of DSL policies

```
1 P1: FOR TENANT T1 WHEN OBJECT_TYPE=DOCS DO SET COMPRESSION ON
      PROXY, SET ENCRYPTION ON STORAGE_NODE
2 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.

Listing 4.3 Java Storlet skeleton

```
1 public class StorletName implements IStorlet {
2    public void invoke(ArrayList<StorletInputStream> inputStream,
3          ArrayList<StorletOutputStream> outputStream,
4          Map<String, String> parameters, StorletLogger log)
5          throws StorletException {
6               //Develop filter logic here
7    }
8 }
```

Data plane (Monitoring): Via the Crystal API (see Table 4.1), we deployed metrics that capture various workload aspects (e.g., `PUT`s/`GET`s 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.

4.5.2 Global Management of IO Bandwidth

Goal: To provide Crystal with means of defining policies that enforce a global I/O bandwidth SLO on `GET`s/`PUT`s:

Listing 4.4 Example of a DSL bandwidth policy

```
1 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 [86]. We implemented a native bandwidth control filter that enables the enforcement at both stages.

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

Data plane (Monitoring): For building the control loop, our bandwidth control service integrates individual monitoring metrics per type of traffic (i.e., `GET`, `PUT`, `REPLICATION`); this makes it possible to define policies 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 [133, 134].

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.

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), spareBwSN(SLOAssignments));
8:      spareBwSLOs ← {};
9:
    ▷ Distribute spare bandwidth equally across all tenants
10:     for tenant in info do
11:         spareBwSLOs[tenant] ← spareBW / numTenants(info);
12:     end for
13:
14:
    ▷ Calculate assignments to achieve spare bw shares for tenants
15:     spareAssignments ← spareSLO(SLOAssignments, spareBwSLOs);
16:
    ▷ Combine SLO and spare bw assignments on tenants
17:     return SLOAssignments ∪ spareAssignments;
18:
19: end function
```

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. 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.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 [135], our goal in Algorithm 1 is to offer an attractive simplicity/effectiveness trade-off, validating our bandwidth differentiation framework.

UNIVERSITAT ROVIRA I VIRGILI
ENHANCING THE PROGRAMMABILITY OF CLOUD OBJECT STORAGE
Josep Sampé Domenech

4.6 Prototype Implementation

The code of Crystal is publicly available on GitHub [136]. The prototype has been tested in different releases of OpenStack Swift, concretely from Kilo to Ocata versions. The Crystal API is implemented with the Django framework. The API manages the system's metadata from Redis 3.0 in-memory store [137]. 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 PyActor [138] for building distributed controllers and workload metric processes that can communicate among them. For fault tolerance, the PyActor 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 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.

4.7 Evaluation

Next, we evaluate the prototype of Crystal in an OpenStack Swift deployment (Ocata version). Our evaluation addresses the challenges of Section 4.1.1 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 I/O bandwidth SLOs on different tenants; iv) Finally, Crystal has low execution/monitoring overhead.

4.7.1 Testbed Characteristics

We ran our experiments in a 13-machine cluster formed by 9 Dell PowerEdge 320 nodes; 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. Nodes in the cluster are connected via 1 Gigabit Ethernet (GbE) switched links.

4.7.2 Workload

We resort to well-known benchmarks and replays of real workload traces. First, we use `ssbench` [139] 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 of these parameters can be configured.

To evaluate Crystal under real-world object storage workloads, we collected the following traces [140]: ii) The first trace captures 1.28 Terabytes (TBs) 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 [141] to create realistic contents for data objects based on the file types described in the workload traces.

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

Storage management capabilities of Crystal. Fig. 4.4 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, composed by 4 PUT operations per 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. 4.4 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 [39] (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).

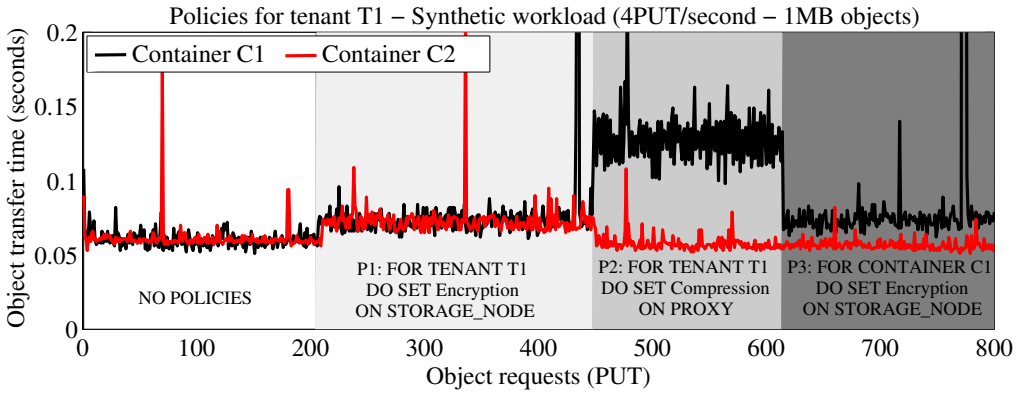


Fig. 4.4 Enforcement of compression/encryption filters

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 4.4, 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` 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. 4.5 shows a dynamic caching policy (`P1`) on one tenant. The filter implements the Least Recently Used (LRU) eviction algorithm, 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.

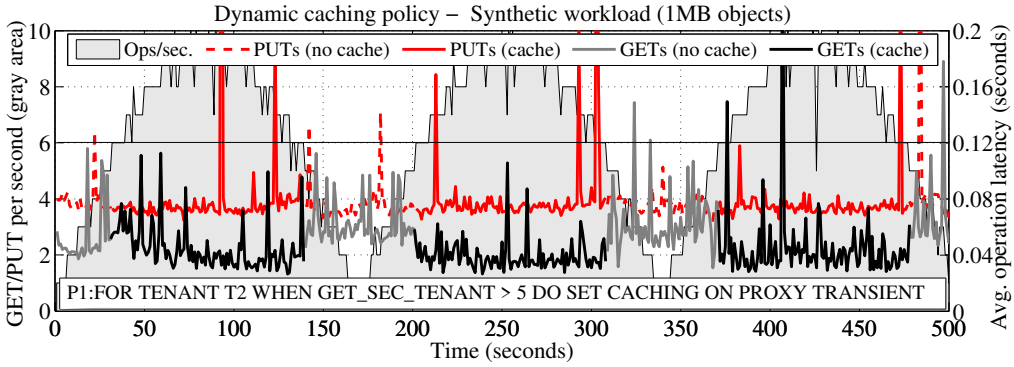


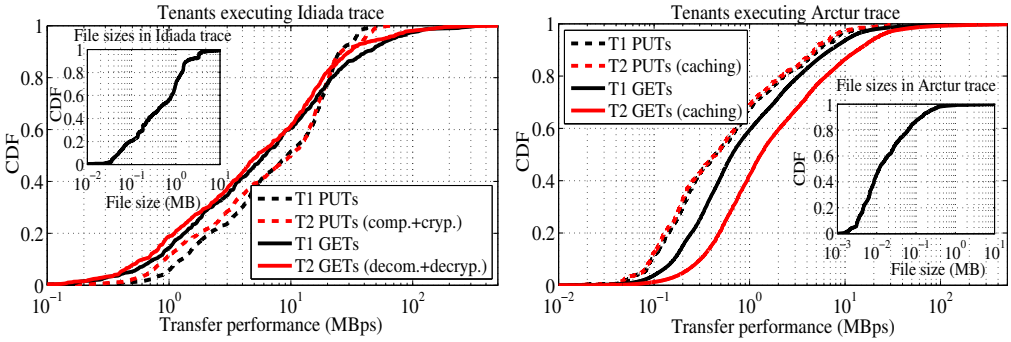
Fig. 4.5 Dynamic enforcement of caching filter

In Fig. 4.5, 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. In this sense, the usage of Crystal for dynamic storage automation provides some benefits. 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 on a replay of the Idiada trace (write-dominated), whereas we enforce caching of small files on a replay of Arctur workload (read-dominated).

Fig. 4.6a 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 higher data confidentially thanks to encryption.



(a) Idiada workload (b) Arctur workload

Fig. 4.6 Policy enforcement on real trace replays

Fig. 4.6b 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.

4.7.4 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. 4.10 we use 3 replicas), the rest of experiments were performed using one replica rings.

Request types. Fig. 4.7 plots two different SLO enforcement experiments on three different tenants for `PUT` and `GET` requests, respectively (enforcement at proxy node). Appreciably, the execution of Algorithm 1 exhibits a near exact behavior for both `PUT` and `GET` requests. Moreover, we observe that tenants obtain their SLO plus an equal share of spare bandwidth, according to the expected policy behavior defined by colored areas. This demonstrates the effectiveness of our bandwidth control middleware for intercepting and limiting both requests types. We also observe in Fig. 4.7 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.

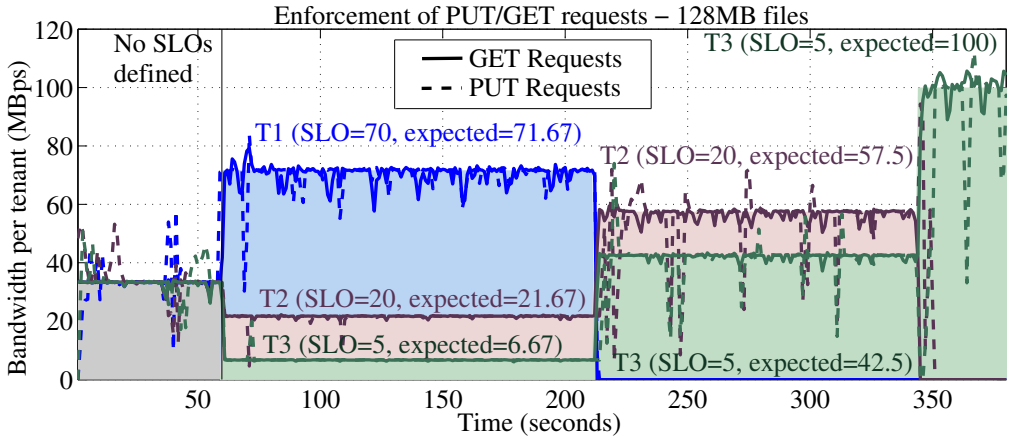


Fig. 4.7 Performance of the Crystal bandwidth differentiation service. 1 proxy/3 storage nodes, bandwidth control at proxy

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. 4.8 shows the enforcement SLOs

on `GET` requests at both stages. At first glance, we observe in Fig. 4.8 that our framework makes it possible to enforce bandwidth limits at both stages. However, Fig. 4.8 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.

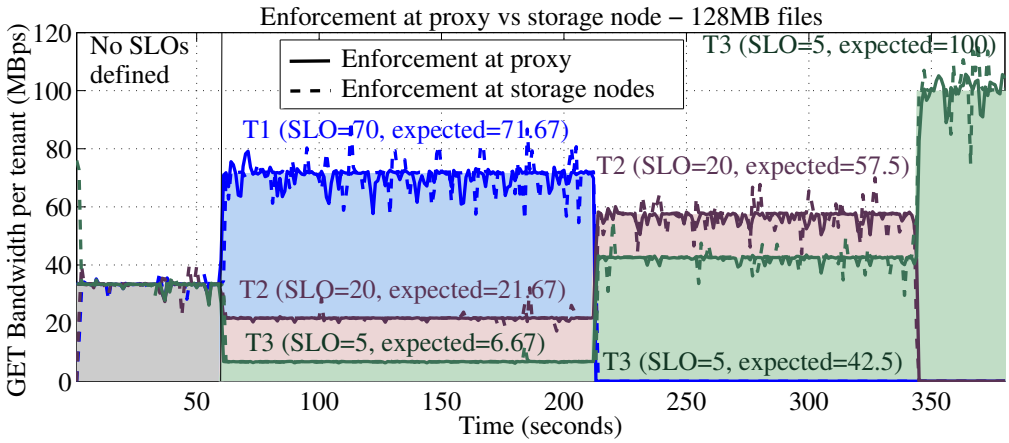


Fig. 4.8 Performance of the Crystal bandwidth differentiation service. 1 proxy/3 storage nodes

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. 4.9 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. 4.9 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.

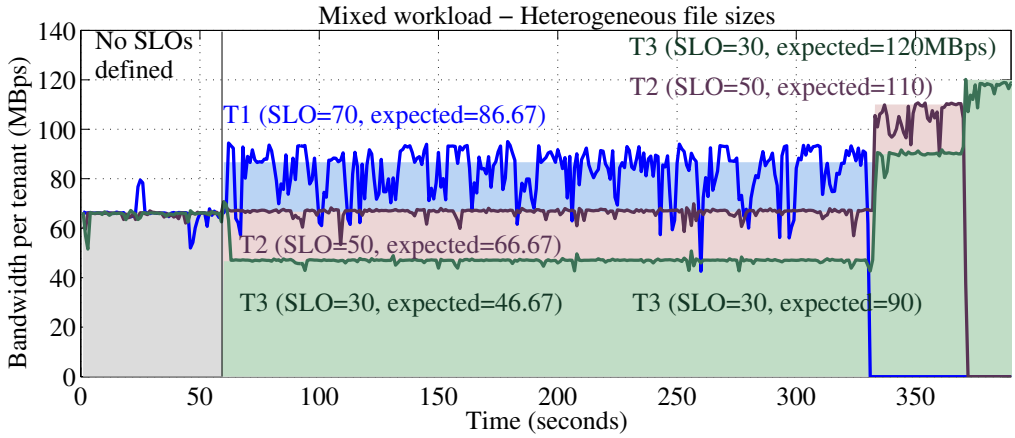


Fig. 4.9 Performance of the Crystal bandwidth differentiation service. 2 proxy/6 storage nodes, bandwidth control at proxies

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. 4.10 illustrates the impact of replication tasks on multi-tenant workloads. In Fig. 4.10, 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.

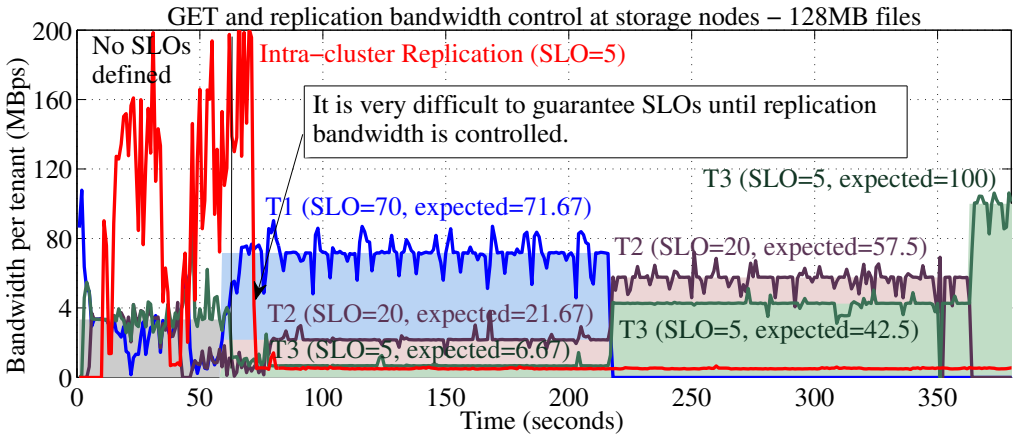


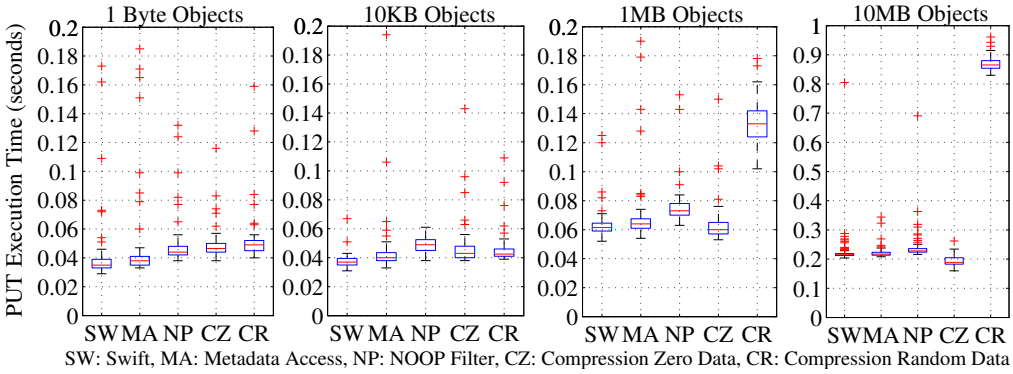
Fig. 4.10 Performance of the Crystal bandwidth differentiation service. 1 proxy/3 storage nodes, bandwidth control at storage nodes

4.7.5 Crystal Overhead

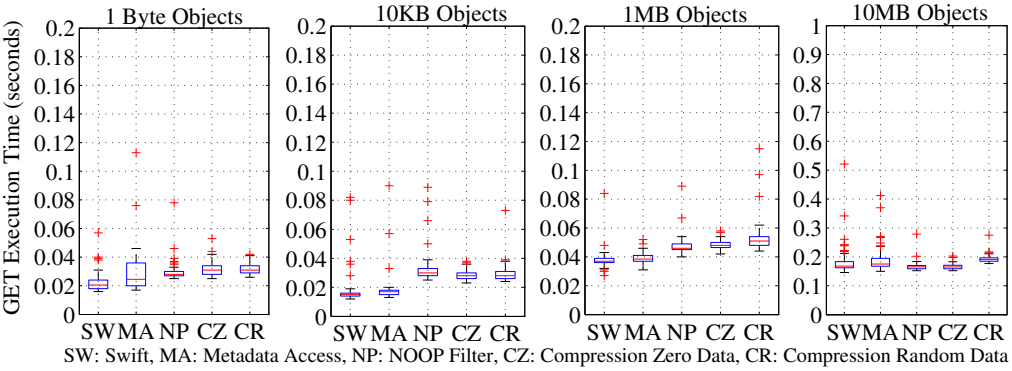
Filter framework latency overheads. A relevant question to answer is the performance costs that our filter framework introduces to the regular operation of the system. Essentially, the filter framework may introduce overhead at i) *contacting the metadata layer*, ii) *intercepting the data stream through a filter* and iii) *managing extended object metadata*. We show this in Fig. 4.11.

Compared to base Swift (`SW`), Fig. 4.11 shows that the metadata access (`MA` in boxplots) of Crystal incurs a median latency penalty between 1.5ms and 3ms. 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 10 Kilobytes (KBs) objects). This confirms that our filter framework minimizes communication with the metadata layer (i.e., 1 query per request). Moreover, Fig. 4.11 shows that an

in-memory store like Redis fits the metadata workload of Crystal, specially if it is co-located with proxy nodes.



(a) PUT Requests filters execution times.



(b) GET Requests filters execution times.

Fig. 4.11 Performance overhead of filter framework metadata interactions and isolated filter enforcement.

Next, we focus on the isolated interception of object requests via Storlets, which trades off performance for higher security guarantees (see Section 4.4). Fig. 4.11 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 [141]. Conversely, decompression in `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 4.4). 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. 4.12 depicts the latency overhead of pipeling multiple `NOOP` Storlet filters. As pipelining is a new feature of Crystal, it required a separate evaluation.

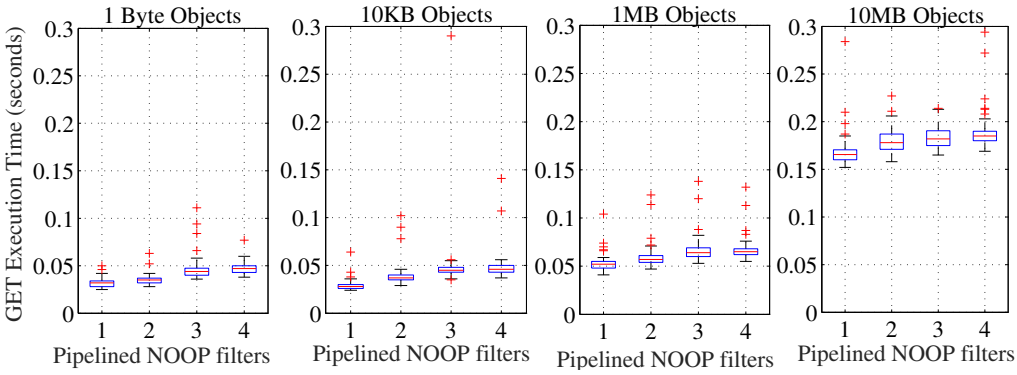


Fig. 4.12 Pipelining performance for isolated filters.

Fig. 4.12 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

UNIVERSITAT ROVIRA I VIRGILI
ENHANCING THE PROGRAMMABILITY OF CLOUD OBJECT STORAGE
Josep Sampé Domenech

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

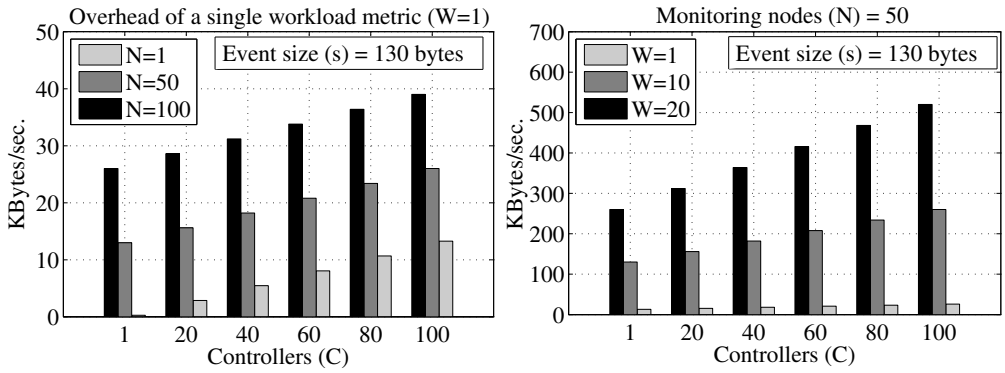


Fig. 4.13 Traffic overhead of Crystal depending on the number of nodes, controllers and workload metrics.

Fig. 4.13 shows that the estimated monitoring overhead of a single metric is modest; in the worst case, a single workload metric generates less than 40KBps in a 100-machine cluster with $|\mathcal{C}| = 100$ subscribed controllers. Clearly, the dominant factor of traffic generation is the number of workload metrics. However, even for a large number of workload metrics ($|\mathcal{W}| = 20$), the monitoring requirements in a 50-machine cluster do not exceed 520KBps. These overheads seem lower than existing SDS systems with advanced monitoring [86].

4.8 Summary

Crystal is a Software-defined Storage 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.

UNIVERSITAT ROVIRA I VIRGILI
ENHANCING THE PROGRAMMABILITY OF CLOUD OBJECT STORAGE
Josep Sampé Domenech

Chapter 5

Adding Self-management Support for Tenants

5.1 Introduction

Nowadays, commercial cloud object stores offer a limited set of actions for the management of the data by tenants. Typically, tenants can only decide the storage class where the data will be stored, and the expiration time of the objects, either manually or by means of policies [8, 42, 43, 10]. Although some cloud object stores also offer the posibility to activate encryption mechanisms [7], these are all the self-management techniques available to tenants. On the other hand, open source systems like Swift provide more flexibility for incorporating new characteristics to the storage service through middlewares [37, 44]. In the same manner, but in the context of file systems, there are some research works [110, 112] that enable administrators to introduce middlewares in the storage stack where all the files pass through before being stored on disk. However, in any case, these middlewares are usually static functionalities installed by storage administrators, so tenants cannot fine tune the storage service with new management and computing tasks adapted to their own specific requirements.

On the other hand, as we demonstrated in Chapter 4, SDS abstracts the storage service and the storage infrastructure from its management by adding a centralized control layer that orchestrates the data plane. This approach significantly increases the flexibility and programmability of cloud object stores.

The results presented in this chapter are published in [142]

However, SDS systems [74, 77, 78], including our previous work, are not designed to manage the data that they store. Generally, SDS technologies are architected for storage administrators [83, 84, 86], which decide the most appropriate storage management policies. Typically, the granularity of these policies are at per-tenant or even per-bucket levels. Thus, although the controller in all SDS systems is assumed to be easy scalable, adapting a "classical" SDS solution to accept policies at per-object level significantly increases its complexity. The number of storage policies grows up considerably when the granularity changes, which becomes a way harder to manage from an administrator perspective. Therefore, SDS abstractions are not suitable to manage tenants' data at such a fine-grained granularity. It does not make much sense for storage administrators to to deal with the data management particularities of hundreds or thousands of tenants.

In spite of this, other technologies provide a complementary layer of data management for cloud object stores. This is the case of *serverless computing* [51, 53]. It allows, at a certain level, the self-management of data by tenants. However, this model does not allow to control the full lifecycle of the objects, as functions are not located in the storage path, but in a disaggregated compute cluster, thus losing the advantages of data locality. With them, it is possible to run functions automatically when the objects are uploaded to the storage service, or by proactively invoking them through an external gateway. In any case, serverless computing leaves a wide range of possible applications unmanageable, concretely those that require synchronous handling.

5.1.1 Scope and Challenges

In this chapter, we argue that a fine-grained data management layer, with a new object-based policy abstraction, is a compelling solution to overcome the challenge of adding self-management support for tenants in cloud object stores. In contrast to SDS, which is not suitable for handling data at per-object level due to its built-in characteristics, our system must provide tenants the ability to manage the particularities of their data. Different objects may require different treatment, and thus, different management. For example, a specific dataset provided by a tenant may need 1) a personalized data processing pipeline of different tasks, such as data obfuscation and encryption, and 2) a specific management, to guarantee, for example, data availability over a concrete period of time. To achieve this goal, it is required to address the following research challenge.

Flexible and programmatic support for data self-management by tenants: This entails the design of a lightweight and extensible per-object management distributed architecture model. Furthermore, to manage the objects at low level, this new model requires a new policy abstraction, flexible enough to manage any kind of object, and programmatic-based in order to grant tenants the appropriate flexibility to code anything they require. Due to performance aspects, these programmatic policies must be executed where the data is, leveraging the underutilized storage resources (data locality). Also, to enable different types of data management, the platform should enable synchronous and asynchronous policy execution in an isolated execution environment to guarantee, as far as possible, the no disturbance of the storage service. Finally, as object storage systems usually store 3 copies of the objects, the system should have high consistency in order to guarantee the correct operation when the same object is accessed at the same time through different copies, which is not trivial to implement in object storage systems, since normally, they are eventually consistent systems.

5.1.2 Contributions

To increase the automation, flexibility, and programmability of object stores, we propose here Vertigo, a novel distributed framework that allows tenants to manage their data in a flexible and dynamic way, making use of a new programmatic policy abstraction called *microcontroller*. Microcontrollers enable tenants to manage the singularities of the objects in a flexible way. Thus, they can be attached to the objects for the appropriate management. Microcontrollers act as object wrappers that control objects behavior. It is for this reason that we introduce the concept of *smart objects* in cloud object stores.

On the other hand, Vertigo leverages the underutilized computing resources of the storage nodes to deploy microcontrollers that intercept objects' lifecycle. The distinguishing feature of microcontrollers 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, for example, on its access history. In addition, microcontrollers can be used to orchestrate active storage tasks, and even to manipulate objects based on their content. Typically, object stores operate at the bucket or object levels, mainly acting as simple repositories of data. One of the outstanding features of Vertigo is that it

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.

Finally, we evaluate the design principles of Vertigo by showing five use cases on top of OpenStack Swift. Theses use cases demonstrate the feasibility of our system and the extensible programmability that microcontrollers provide to cloud object stores, allowing tenants to self-manage their data in a flexible way.

5.2 The Management Policies: Microcontrollers

The entire architectural model revolves around the concept of microcontroller, and therefore, it is critical to define what a microcontroller is. As a working definition: *a microcontroller is a data behavior or management policy that is associated to one or more objects, and to one or more storage operations.* As cloud object stores operate through HTTP RESTful APIs, microcontrollers can be associated to any of their available HTTP methods, that is, GET, PUT, POST, HEAD, and DELETE. Thus, upon an object request to one of them, the associated microcontrollers (if any) are executed in a sandboxed environment. Objects in an object store are composed by an identifier, the actual data of the object, and the metadata that describes its content. Vertigo objects can also be composed by microcontrollers, which could be seen as wrappers at the very high level.

Microcontrollers follow the active storage approach, that is, they are executed close to the data within the storage infrastructure. However, they are not actually active storage tasks. Active storage tasks usually modify the current input or output data flow of the objects. Instead, microcontrollers are the policies that define the behavior of the objects. It should be noted that, as we will describe below, one characteristics of microcontrollers is that they are able to instrument more than one active storage task for object processing.

The programmability that microcontrollers provide, in contrast with descriptive high-level policies, allows tenants to create advanced forms of object control. Vertigo provides two different types of microcontrollers: `synchronous` and `asynchronous`. A synchronous microcontroller blocks the incoming or outcoming request until it completes the execution. In the synchronous model, microcontrollers are executed in real-time, which means that the lifecycle of ob-

jects is intercepted. In contrast, asynchronous microcontrollers are event-driven, that is, they run when an event occurs. The request that triggered the event does not wait for their completion.

5.3 Design Overview

The complete architecture is depicted in Fig. 5.1. As expected, the data plane includes the traditional cloud object storage system which manages objects and their associated metadata.

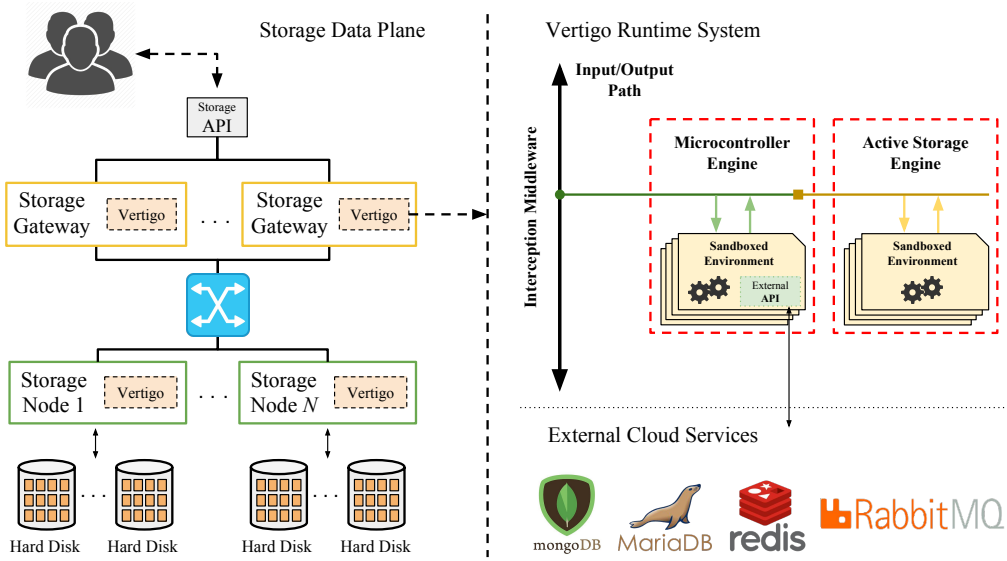


Fig. 5.1 High-level architecture overview of Vertigo integrated alongside a common cloud object storage architecture

Within the data plane, each node, either storage gateways where tenants connect to perform operations, and storage nodes where the actual data is stored, includes the Runtime System. The Runtime System is the entity that intercepts all types of an object's lifecycle requests. It contains the microcontroller and the active storage engines, responsible for running microcontrollers and active storage tasks in a safe and sandboxed environment, respectively. Moreover, the Microcontroller Engine has direct access to external cloud services through an extensible gateway module (API), including the object storage system itself. Thanks to this gateway, a microcontroller, and therefore tenants, can take benefit

UNIVERSITAT ROVIRA I VIRGILI
ENHANCING THE PROGRAMMABILITY OF CLOUD OBJECT STORAGE
Josep Sampé Domenech

of external cloud services, for example, for: 1) Persisting the state of the objects in an external database; 2) Operating over other objects in the object store; 3) Storing an access log of an specific object; 4) Sending a message to a message queue in order to notify other services of the performed actions over the objects, among others. Microcontrollers have access to all the relevant request information, such as the user name or id, the tenant name, and all the headers of the request. Moreover, they can access to the metadata of the requested object. In brief, microcontrollers have access to all of the environment information except the actual data of the object. This is because microcontrollers are management policies and not active storage tasks.

Regarding the associative approach of Vertigo, a microcontroller can be associated to more than one object. Also, an object can have more than one microcontroller associated. In the same way, in the storage service there may be microcontrollers not associated to an object, and also, objects with no associated microcontroller. Thus, the relationship between microcontrollers and objects is (N:M). In this sense, following the previous example, Fig. 5.2 illustrates the deployment of a microcontroller that deletes an object after x `gets`.

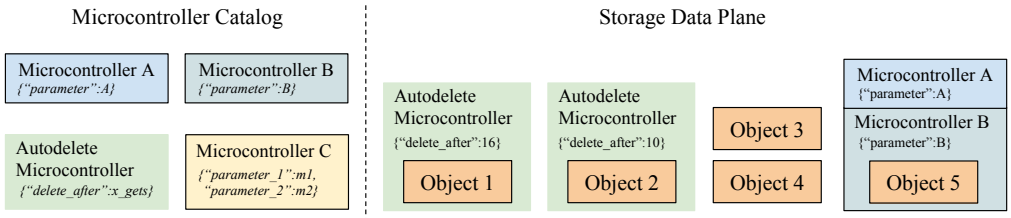


Fig. 5.2 Microcontroller deployment example

The idea behind our approach is to develop a generic delete microcontroller, and then associate it with different objects. In this example, what changes is the parameter value associated with the target object and the *autodelete microcontroller*: `{"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 microcontroller for different objects.

Examples. To better understand the role of the Vertigo components, consider a microcontroller that counts the number of GET requests to one object. To deploy it, the code of this microcontroller must be first loaded to the object store

as a common object, and then tell the Vertigo Runtime System that an *onGet* trigger is associated with this registered microcontroller for the targeted object `(KEY=value)` or objects. From that moment, every GET request to this object will be intercepted and routed to this microcontroller, eventually executed within the Microcontroller Engine of the specific tenant. A more advanced example could be a microcontroller that performs actions on the data plane when some condition is met. For example, the previous microcontroller could delete an object after 5 GET requests. In this case, the microcontroller would instrument the object store API to perform such delete action on the data plane. Finally, to get a full perspective of the potential of Vertigo, a more sophisticated example could be a microcontroller that instruments 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. In this case, when the appropriate conditions are met, the microcontroller executes the compression task within the Active Storage Engine to perform such operation.

5.3.1 The Core: Runtime System

The Vertigo Runtime System contains two major components for managing the different types of microcontrollers: the *Microcontroller Engine* and the *Active Storage Engine*. Both components are located within the storage system nodes, either gateways or storage nodes, and work on an isolated and sandboxed environment (Linux containers) that intercepts calls to the object store.

Microcontroller Engine. It takes care of the management (installation, deployment and configuration) of microcontrollers. *Installation* means uploading and registering the microcontroller code in the cloud object storage system. The microcontroller is stored as a common object but in a specific storage bucket, specially created for storing microcontrollers. *Deployment* refers to linking an installed microcontroller to one or more objects with one or more triggers. Triggers include the operations that may be intercepted in the selected object or objects: PUT, GET, POST, HEAD or DELETE. Finally, *configuration* refers to setting up the parameters and the necessary metadata of an already installed microcontroller, e.g., the maximum number of reads, or the compression ratio if the microcontroller orchestrates a compression task.

Once deployed, microcontrollers are triggered in reaction to lifecycle events of objects in the data plane. By default, microcontrollers do not have arbitrary

external Internet access. However, one fundamental part of the *Microcontroller Engine* is the external gateway module, which enables microcontrollers to communicate with external cloud services. The gateway provides an extensible API to access, for example, a Redis database, a MongoDB database, or a message broker like RabbitMQ. As stated before, thanks to this API, microcontrollers can store information in a persistent way by transparently benefiting from object and request metadata. For example, a microcontroller can maintain an access counter or store the timestamp of the last access in the object metadata. In short, a microcontroller can take one or all of the following actions after intercepting a lifecycle request of an object:

- It can add, update, or delete the metadata of the current object.
- It can `cancel` the request, or `rewire` the request to another object.
- It can orchestrate one or multiple active storage tasks. That is, when the appropriate conditions are met, tasks such as compression, encryption or filtering can be executed for data reduction or protection techniques.
- It can perform POST or DELETE requests to the cloud object store. For example, for deleting or for updating the metadata of any other object within the storage system.
- It can communicate with external services through the external gateway module. For example, for accessing a RDMS or a NoSQL database.

Active Storage Engine. It takes care of the management of active storage tasks. It is responsible of the installation, deployment and configuration of them. These tasks are stateless components that process the input and output data flows from and to an object. Typically, active storage tasks, like compression, are used as data reduction techniques in cloud object stores. They also include extraction, transformation and load (ETL) mechanisms. Moreover, other tasks like data encryption or data obfuscation are used for data protection.

5.4 Prototype Implementation

We implemented Vertigo [143] on top of OpenStack Swift [13], and modified the open source OpenStack Storlet framework [40] to suit our requirements (see Section 5.4.3 for further details).

As aforementioned, the core component of Vertigo is the *Runtime System*. Fig. 5.3 shows the integration of this component within a common Swift deployment. For security reasons, each tenant in the object store has its own isolated Runtime System. Taking Swift as basis, it is mainly composed by three elements: the *Interception Middleware*, the *Microcontroller Engine*, and the *Storlets Engine*.

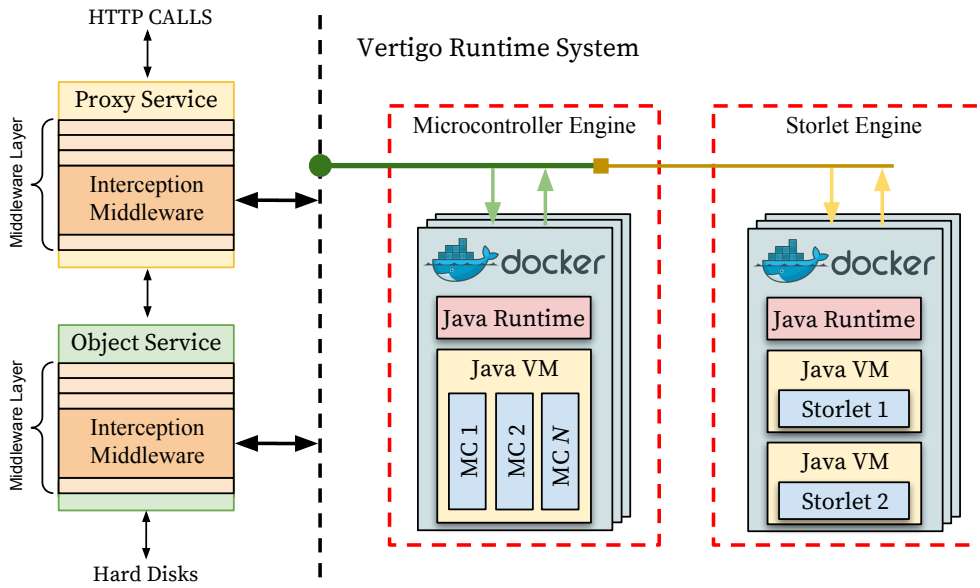


Fig. 5.3 Vertigo Rtuntime System integrated in OpenStack Swift

5.4.1 Interception Middleware

We built a new Swift interception middleware with two major tasks. First, to deploy (and undeploy) microcontrollers to a particular object(s), and second, to load microcontrollers to the Microcontroller Engine. For the deployment of a microcontroller, a user must issue a POST request to the object to be managed. Upon this action, a trigger header of the type: `onGet`, `onPut`, `onPost`, `onHead`, or `onDelete`, must be appointed to tell the framework which lifecycle event must be intercepted. Moreover, a microcontroller may have specific parameters related with the managed object. This information is uploaded to the system attaching to the request the parameter names and their values. Finally, this information is stored alongside the object, in its metadata.

To better understand this, consider a tenant that wants to deploy the previous automated deletion microcontroller to a given object. To do so, a Swift POST request with the header "`X-Vertigo-Onget:delete-1.0.jar`" is needed to deploy the microcontroller `delete-1.0.jar` with the `onGet` trigger. In this case, the microcontroller needs to know how many readings are allowed to the object with the parameter "delete_after". The parameters list is a JSON formatted string, easily readable by the microcontroller. For example, with the `curl` Unix tool, the command would be the following, in Listing 5.1.

Listing 5.1 Example of deploying a Vertigo microcontroller

```
curl ... $STORAGE_URL/bucket_name/object.csv
         -X POST
         -H "X-Vertigo-Onget:delete-1.0.jar"
         -H "X-Vertigo-mctype:sync"
         -d "{‘delete_after’:‘16’}"
```

The interception middleware is the responsible for reading the JSON information which contains the parameters, and for storing its content in the metadata of the object. In the same POST request it must be specified whether the microcontroller will run synchronously or asynchronously to the main request. Note that it is possible to deploy more than one microcontroller per object. In this case, the order of microcontroller execution would be the order of the deployment. On the other hand, a microcontroller can be undeployed with a special header added in a POST request: "`X-Vertigo-Onget-Delete:delete-1.0.jar`". This delete operation removes the microcontroller from the trigger list, and its parameters list from the metadata of the object.

Microcontroller deployment granularity. Although OpenStack Swift stores the objects in a flat name space, it is possible to simulate a hierarchical structure within a single bucket by adding forward slash characters (/) in the object name. Each folder between the bucket name and the object name is called *pseudo-folder* in Swift. In Vertigo, we take advantage of this logical hierarchy upon the deployment of a microcontroller to an object. Thus, Vertigo allows the deployment operation to act at different granularities: at per- object, pseudo-folder and bucket levels. In this sense, when a tenant associates a microcontroller to a bucket or pseudo-folder, the deployment operation is offloaded to all objects inside them. As an example of this process, if a user requires to associate a microcontroller to

all the objects inside "`acc_1/cont_2/pf_3`", the POST operation should point to this `pseudo-folder`, adding an *asterisk* at the end, as shown in Listing 5.2.

Listing 5.2 Example of associating a microcontroller to a pseudo-folder

```
curl ... $STORAGE_URL/acc_1/cont_2/pf_3/*
         -X POST
         -H "X-Vertigo-Onget:counter.jar"
```

For all the available triggers, except for the `onPut` type, when a microcontroller is deployed on a bucket/pseudo-folder, the deployment information is stored in the metadata of the available objects in it, in addition to the metadata of the the bucket/pseudo-folder itself. This enables the inheritance of microcontrollers, in such a way that when a user uploads an object to that bucket/pseudo-folder with microcontrollers deployed, all of them are inherited to the new object. Therefore, After the deployment operation (Listing 5.2), when a user uploads an object to "`acc_1/cont_2/pf_3`", the microcontroller `counter.jar` would be automatically deployed on it. The same occurs with the undeployment operation. It works at per- object, pseudo-folder, and bucket levels. When a users undeploy a microcontroller from a bucket/pseudo-folder, the operation is offloaded to all the objects inside them, and the inheritance of the microcontroller is deleted.

Regarding PUT operations (`onPut` trigger), the deployment information of the bucket/pseudo-folder is stored in two different places. First, as before, it is stored in the metadata of the the bucket/pseudo-folder itself. Second, the deployment information is kept in the memory of the proxy nodes. The interception middleware makes use of `Memcached` [144] for keeping the associations between objects and microcontrollers in a sort of cache. This approach speeds up the information retrieval time. In this way, with `Memcached`, Vertigo does not need to make a sub-request to Swift in order to know the microcontroller execution list when an object enters the storage cluster.

After the microcontroller deployment process, when a request arrives to an object, the middleware checks if the object has microcontrollers associated in the trigger related to the request. For example, if the request is a GET, the middleware checks the `onGet` trigger of the object. If there are microcontrollers deployed, the middleware automatically starts the Microcontroller Engine and sends to it the list of microcontrollers to execute, as well as the object metadata, the request metadata and the microcontrollers specific parameters.

One of the main characteristics of Vertigo is that a microcontroller can run synchronously or asynchronously to the object request. Asynchronous model permits to reduce the overhead produced by Vertigo. Imagine a microcontroller that stores a log of users that access to an object, in an external database. In this case, it is possible to launch the microcontroller asynchronously, and continue the main object request execution, following the default storage path. Thus, the main request does not need to wait for the overhead produced by the microcontroller when a value is stored in an external database. Moreover, with Vertigo it is also possible to run microcontrollers in a semi-synchronous way. This means that a microcontroller can run part of the code synchronously, return the control to the interception middleware, and then run the rest of the code asynchronously. This approach also reduces the overhead produced by the Microcontroller Engine

5.4.2 Microcontroller Engine

The Microcontroller Engine is a daemon process that provides isolation and safety thanks to an isolated environment using Linux `Docker` containers. As a result, the microcontrollers are sandboxed: They have no direct network access, no system execution capabilities, no thread creation capabilities, and no access to the file system. In the current implementation of Vertigo, the daemon process is based on Java. In this sense, each daemon process consists of a Java Virtual Machine (JVM) with a thread pool, where each microcontroller is executed in a different thread inside the same JVM.

The Microcontroller Engine is shared by all the users of the same tenant, but for each request there is one different instance of the microcontroller. When it receives the microcontroller list and all the metadata and parameters, it runs all the microcontrollers in the appropriate order. As stated above, the isolation provided by the Docker containers does not allow microcontrollers to have any kind of external access. However, we built a gateway that provides access to external cloud services. The basic microcontroller skeleton is as follows:

Listing 5.3 Vertigo microcontroller Java skeleton

```
1 public class mc implements IMicrocontroller {
2    public void invoke(Context ctx, Api api) {
3       // microcontroller code
4    }
5 }
```

Microcontrollers have two main parameters, the `Context` and the `API`. On the one hand, the `Context` provides access to the related request information. As stated in the previous section, this information includes the object metadata, the request metadata and the microcontrollers' specific parameters. Moreover, by using the context, it is possible to control the current request with one of these commands: *forward*, *cancel* and *rewire*. The *forward* command tells the interception middleware to continue the request normally, without doing any action. The *cancel* command cancels the request and returns an error message. Finally, the *rewire* command allows to rewire the request to another object.

On the other hand, the `API` parameter provides access to external services. By default, it includes access to: 1) Storlet Engine, for executing Storlets (by the `run-storlet` command); 2) Logging engine, which is useful for debugging microcontrollers; 3) OpenStack Swift, which allows microcontrollers to perform POST and DELETE operations over it. Although the first implementation of the `API` provides basic communication to microcontrollers, we built it to be easily extensible. It allows to add support to other external services, for example: RabbitMQ, Redis, MySQL and a large etc., even if it is necessary to add support to services out of the local cluster, for example, Amazon Redshift [145].

By default, when a synchronous microcontroller is executed, the interception middleware remains waiting until it receives one command to continue the request execution (*forward*, *cancel*, *rewire* or *run-storlet*). Once the microcontroller sends one of them to the interception middleware, then the request is processed accordingly. In such a case that more than one microcontroller is executed, the priority of the received commands would be the next:

Listing 5.4 Microcontroller response priority

```
1 cancel > rewire > run-storlet > forward
```

As a simple example of this functionality, we can implement a synchronous microcontroller that controls the access of an object to an specific user. With this example, we can see the flexibility of our framework to bring out a lot of extra functionalities in cloud object stores. This simple microcontroller provides finer-grained access control than the default *Swift bucket ACLs*. In this case, after the microcontroller is associated with the *onGet* trigger of an object, then only the specified user will be able to access to the object. Any other user will get the "Error 401" message, as shown in Listing 5.5.

Listing 5.5 Example of an access control microcontroller

```
public class ac implements IMicrocontroller {
    public void invoke(Context ctx, Api api) {
        if (ctx.request.user == "admin")
            ctx.request.forward();
        else:
            ctx.request.cancel("Error 401");
    }
}
```

Moreover, following the previous example, we can implement more advanced access control microcontrollers. For example, to only allow access to an object on specific hours of the day, or a given number of times. Also, a microcontroller can be used to manage Storlets, thus enabling implicit active storage tasks over the objects. For example, to run a compression Storlet on an object when a user performs a PUT request. Finally, a simple example of asynchronous microcontroller could be one which stores the username of all the users that access an object, to an external database, thus maintaining an access log of an object. In this case, the microcontroller would be associated to the *onGet* trigger of the object. As the request does not need to be neither modified nor controlled in any form, the microcontroller can run its code asynchronously:

Listing 5.6 Example of a microcontroller that writes to an external database

```
public class log implements IMicrocontroller {
    public void invoke(Context ctx, Api api) {
        api.db.store(ctx.request.user_name)
    }
}
```

Object metadata cache. One characteristic of microcontrollers is that they allow to modify objects' metadata. By default, Swift stores 3 copies of the objects. However, Swift is an eventually consistent system. This means that in a particular time, the replicas of an object may have different data and metadata. As microcontrollers are executed where the data is, modifying only the local copy of the metadata implies waiting Swift to replicate the modified value to all the copies of the object. This is not suitable in terms of request parallelism, since there may be multiple requests to the same object at the same time, and even

in different storage nodes. Moreover, this is a problem if this modified value is required in the parallel requests to the same object.

To overcome this issue, the Microcontroller Engine uses an external Redis database to manage the metadata of the objects. Thus, when a microcontroller is executed, it initially loads the object metadata into Redis, and then, makes all metadata modifications over it. If there are other requests to the same object, the other microcontrollers will use this previously-loaded metadata. When the microcontroller terminates, it evicts the metadata from Redis to all the object replicas. The metadata is kept into cache following the LRU replacement algorithm. With this distributed cache, we guarantee that the object metadata added by the microcontrollers is always consistent in all replicas, even if there are multiple requests at the same time. With it, it is even possible to do atomic operations over the metadata fields.

5.4.3 Storlets Engine

This is the third component of the Runtime System, and it is responsible of processing the actual data of the objects. We leveraged the OpenStack Storlets framework to implement this component. Storlets framework extends Swift with the capability to run computations close to the data in a secure and isolated manner, making use of `Docker` [132] 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 tasks in Vertigo, we were forced to extend their functionality. As the current Storlets framework only supports one (explicit) Storlet per request, we modified it to enable the pipelining of Storlets in the same request. Also, we enabled *implicit calls*, *metadata management*, and *orchestration*, among other things. For instance, a simple composite function like the one in the Listing 5.7 cannot be implemented with the current Storlets framework. However, it can be easily achieved with Vertigo and its original microcontroller abstraction.

Listing 5.7 Storlet composition example

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

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. Moreover, with our modified version, the Storlets can also run in parallel; it is not necessary to finish one to run another. For example, as we show in Section 5.7.2, with Vertigo it is possible to run a Storlet that transcodes a PDF document into plain-text, while, at the same time, another Storlet runs a GREP command over the resulting data flow to find, for example, some relevant phrases.

5.5 Extensibility

In Section 5.3.1 we described some relevant actions that a microcontroller can perform to manage the objects. However, its functionalities can be extend in diverse ways by an storage administrator. On the one hand, as we already discussed, the external gateway can be extended to accept any other cloud service, exposing its communication API to the microcontroller. On the other hand, the functionalities of microcontrollers can be extended by adding new middlewares in the Swift pipeline. Middlewares enable new features in the storage service, such as encryption or object versioning (see Section 2.5.3). In this sense, microcontrollers can directly benefit of Swift middlewares for managing the data. Thus, it is possible to extend the storage functionalities by adding new middlewares in the storage path, but with the capability to interact with microcontrollers.

In this direction, any of the available Swift middlewares in the official catalog [37] are able to be managed by a microcontroller. However, to demonstrate their integrability with Vertigo, we created two middlewares which add new capabilities to the storage service. These middlewares are the *object prefetching* [146], for prefetching objects in the Swift proxy servers, and the *object linking* [147], for making *soft links* between Swift objects. They are activated and managed by some request metadata headers, so microcontrollers can easily interact with them. Moreover, it is possible to limit the operation of these middlewares to work only from microcontrollers, and not from external user requests, which prevents non-experts users to wrongly use them.

Object prefetching. The main objective of prefetching objects is for speeding up GET operations, thus improving the overall object request time. This process is done before the object is known to be needed, and it loads the object in a low latency cache. Storage nodes commonly use HDDs to store the information,

however, their mechanical nature limits their overall performance. Is for this that caches store objects in memory or SSDs. Anyway, OpenStack Swift, by default, does no provide ways for prefetching objects in a proxy server cache.

This middleware enables object prefetching managed by microcontrollers. Thanks to our approach, it is possible to preload objects in a proxy server cache before users request them. Caching objects in the proxy server improves substantially the overall request time. For instance, a microcontroller can prefetch some related objects when other one is requested, or when a microcontroller detects that the object will be highly requested.

Object linking. In Swift, data replication and data placement is provided by the Rings. Each Ring is a storage policy that contains the storage nodes and the hard disks that it can use to store the information. In Swift, each bucket has a storage policy associated, assigned when the bucket is created. However, it is not possible to change it during the bucket lifetime. The main drawback of this approach is that, if it is needed to change the replication level or the data placement of an object, that object must be moved to another bucket. Doing this operation means that its original ID changes. That is, the object_id of *acc/bucket_1/object* would change to *account/bucket_2/object*. This is not a drawback if the tenant is aware of the change, but in such cases where the data movement has to be transparent to the tenants and users, it implies that they cannot access anymore to the object, due to the original ID is lost.

This middleware provides a way to move objects between buckets, leaving a symbolic link to the original location of the object. This symbolic link is a zero-bytes object that points to the new object location. Then, when a user accesses to the symbolic link, the request will be rewired to the original object. This simple feature enables microcontrollers to perform automatic object migration, based on, for example, its access history. We evaluated the middleware in Section 5.7, showing a minimum overhead when Vertigo gets the link object to obtain the location of the original object.

5.6 Applications

In this section, we show some of the applications that our framework can support. 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 data 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[1] 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[2]. 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. Moreover, because an object may have dependencies on other objects, the microcontroller can also delete them when the main object meets the conditions to be self-destructed.

As a basic example, we built a microcontroller 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.

Active Storage Orchestration. 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 Vertigo. This capability has been inherited by the OpenStack Storlet Engine. However, our framework permits the *pipelining* of consecutive computations that is not possible to perform with the current 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`

[1]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.

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

microcontroller 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. Moreover, these type of microcontrollers may be very useful to orchestrate *active storage* tasks which implement ETL (extract, transform and load) functions. With our microcontrollers, we can provide an intermediary transformation layer between raw object storage service and (big) data analytics frameworks.

As a simple example, we built a pipeline of two *active storage* tasks orchestrated by an `onGet` microcontoller. 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 [148], which shows the potential of our framework for data transformations.

Content Level Access Control. With Vertigo, it is very easy to implement advanced forms of access control. Typically, access control in object stores operates at the granularity of buckets, 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 bucket is enforced through access control lists. 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 microcontroller 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 [149], 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`[3] 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 a "police agent" should be able to access to all fields and issue an SQL query:

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

[3]As the dataset does not contain explicit identifiers, we added a random SSN to each row.

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.csv

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

This simple access policy is stored as the parameters of the microcontroller, as explained in Section 5.4.1. When a GET request comes for the object `adult_data.csv`, the target object server first checks the Swift ACL. If the object is accessible by that user, the microcontroller then reads the content level policy, and orchestrates an Storlet (like the previous use case) which execute the SQL query over the data, 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 microcontrollers that specify how the object behaves.

Automated Object Migration. By using the *object linking middleware* described above, Vertigo also provides a platform for migrating objects in an automated way. One motivation to move the data to another replication levels could be to reduce the storage costs. For example, as previously explained, in Swift is not possible to change the replication level of a bucket, so when we desire to change the replication of an object or a group of objects, it is needed that tenants explicitly move all the objects to another bucket.

One example of automated migration could be to move that data which is infrequently used, after few days, to a less-replication bucket (archiving). With the decentralized architecture of Vertigo, each object is managed itself by microcontrollers. In this case, we implemented a basic microcontroller that stores the last access timestamp of an object. As said, this metadata (the timestamp) is stored with the object itself, and will remain with it during its lifetime in the storage cluster. The microcontroller also keeps the access ratio as the state of the object. Moreover, with the microcontroller parameters, a tenant can set an specific access ratio threshold to the object, so that when the ratio decay below the threshold, it moves the object to a less-replication bucket, leaving a soft link in the original location. Normally, less-replication tiers have higher latencies, thus, the overhead added with our soft link implementation is not important.

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 I/O operations. The distinguishing aspect of Vertigo is that prefetching can be done per object to suit the specific application requirements instead of system-wide. This flexibility is useful for applications that put different degrees of emphasis on performance and latency. For instance, an access to an HTML file may preload objects of other pages.

As a basic example, we implemented a simple Web prefetching mechanism. When a user stores an HTML file for the first time, a microcontroller deployed with an `onPut` trigger instruments a Storlet for parsing the object and identifying the embedded objects. The result of this process is a list of Swift objects which compose each specific HTLM document. Such a list is stored in JSON format in the metadata of the object to enable the `onGet` microcontroller to preload all the embedded objects into `Memcached` 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.

5.7 Evaluation

We integrated the built prototype of Vertigo in an OpenStack Swift deployment, and evaluated it in terms of flexibility, performance and overhead. In this sense, we ran micro-benchmarks and measured the resources usage of the Microcontroller Engine, in addition to the overheads associated to microcontrollers, both in the GET and PUT requests. Moreover, we implemented the necessary microcontrollers to test all the previous applications. We evaluated them by running each one with a specific setup described in Section 5.7.2. To do so, we measured the execution times and bandwidth improvements associated to the execution of the related microcontrollers.

5.7.1 Testbed Characteristics

Our experimental testbed consisted of a client host with 2VCPUs and 4GB RAM. On the server side, we deployed Vertigo in an 8-machines rack with an OpenStack Swift (Ocata version) installation formed by 2 proxy node Dell PowerEdge R320 with 32GB RAM, and 7 storage nodes Dell PowerEdge R320 with 16GB RAM.

All of these machines, including the client, were connected via GbE network. All the server-side machines ran Ubuntu Server 16.04. The client host ran Ubuntu 16.04.1 CloudImage [150].

5.7.2 Workload

Here, we describe the specific workload configuration used for each application.

Automated Deletion. We utilized a random 100MB file, even thought, actually, in this case, is not important the size of the file because there is no data modification. It only removes the object where some condition are met. This means that the overhead of this application will be the same independently of the size of the object. We established in the microcontroller metadata to delete the object after 100 reads.

Active Storage Orchestration. For this experiment, we used a single PDF document of 100MB. More precisely, the `onGet` microcontroller 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 (due to the `grep` library that we have used in the Storlet), so our interception middleware remains waiting until this Storlet starts to return the answer. For this application we implemented a single microcontroller deployed to the *onGet* trigger of the PDF object. Its functionality consist in orchestrate both Storlets.

Content Level Access Control. We used the dataset described in Section 5.6, but we extended it to 100MB. As explained before, the dataset content is restricted to the type of user that makes a GET request over the file. To this aim, we used Swift user roles to return only specific fields. In this case the microcontroller, deployed to the *onGet* trigger of the dataset, reads from his metadata the fields allowed for the user role (e.g. `age`, `education`, `capital-gain`, `native-country`), and then orchestrates the SQL Storlet to filter it.

Automated Object Migration. For this experiment we used a random document of 100MB, and we created two buckets with different storage policies (in Swift, the storage police determines the replication level and the replica placement). The microcontroller is configured to calculate the access ratio of the

UNIVERSITAT ROVIRA I VIRGILI
ENHANCING THE PROGRAMMABILITY OF CLOUD OBJECT STORAGE
Josep Sampé Domenech

past 10 minutes, and we set a threshold of 1 (all of this information is stored in the microcontroller metadata). That is. when the access ratio of the object be less than 1 in the past 10 minutes, the object will be automatically moved to the another bucket, leaving a soft link in the original location.

Automated Prefetching. We took as an HTML file the Google HTML5 slide template [151], which consists of a single HTML document (`index.html`) and several embedded objects, including 10 image files of around 3.5MB and some JavaScript scripts. 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 sufficient to show the inventive aspects of Vertigo. To generate the workload, we used the Apache JMeter [152] toolkit.

5.7.3 Application characteristics

Table 5.1 shows information about the necessary microcontrollers for running the previous applications. The first column gives the name of the microcontroller, the second column gives the approximate number of *lines of code* or Java instructions required to execute it. Moreover, the third column (*type*) gives its synchronicity: *sync*hronous or *async*hronous. Finally, the fourth column gives the *size* of each compiled microcontroller. From this table, it can be seen that our microcontrollers are very lightweight for the proposed applications, and with few lines of code, it is possible to add new object management functionalities in a cloud object store.

Table 5.1 Application microcontrollers information

Microcontroller	**Lines of code**	**Type**	**Size**
Automated Deletion	≈ 5	sync	1.6K
Active Storage Orchestration	≈ 3	sync	1.4K
Content level Access Control	≈ 8	sync	1.7K
Automated object migration	≈ 24	async	2.2K
Resource Extractor	≈ 4	sync	1.5K
Automated Prefetching	≈ 6	async	1.7K

5.7.4 Results

All the tests were run 200 times and the results were averaged to create the plots in the next figures. Moreover, for those applications that can be compared to how they would work without a microcontroller, such as the content level access control, we evaluated the following two configurations: Traditional storage or baseline configuration (TS), namely, Swift without Vertigo, and the microcontrolled version of Swift (MC).

Base Overhead. As stated in Section 5.4.2, the Microcontroller Engine is executed inside a Docker container, therefore, we evaluated the base overhead of running containers in a Swift storage node. It is critical to measure it because the microcontrollers are object wrappers which are executed where the data is. Thus, the framework can only make use of the storage node resources where the microcontroller is executed. Because our implementation launches one Engine for each different tenant of Swift, this experiment consists of launching a new container each 5 seconds, and measure the impact on both the memory and the CPU. We used for this experiment the ubuntu:16.04 [153] docker image as a base image of the Microcontroller Engine.

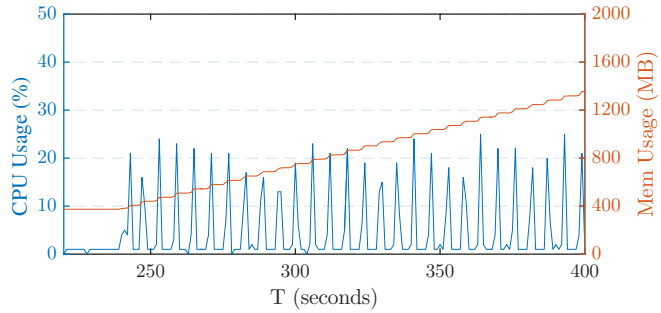


Fig. 5.4 Memory and CPU consumption for running a new docker container with the Microcontroller Engine

Fig. 5.4 shows the results of this experiment. As we can see, launching a new container entails a peak of around 20% of CPU utilization, but almost null CPU utilization to keep it running. In contrast, the memory consumption increases linearly to the number of launched containers. We measured that each one consumes, at least, 30MB of memory. Although it is not an important memory overhead, this result means that the number of different tenants that can execute

microcontrollers in a storage node, at a given instant, will be limited by the total RAM memory. In one of our storage nodes, width 16GB of RAM we could launch around 450 Docker containers.

As another base experiment, we measured the overhead added by launching a microcontroller with Vertigo. The overhead is the time needed by the interception middleware to launch a microcontroller inside the Microcontroller Engine (docker), and get a response from it. For this experiment we used the microcontroller listed in Listing 5.8, where the only instruction returns the control to the interception middleware without doing any other action.

Listing 5.8 Example of a No-Operation Vertigo Microcontroller

```
1 public class base implements IMicrocontroller {
2     public void invoke(Context ctx, Api api) {
3       ctx.request.forward();
4     }
5 }
```

We first evaluated the overhead added in the GET requests. To do so, we launched 200 requests to different objects with this microcontroller deployed on the `onGet` trigger. The results depicted in Fig. 5.5, show that almost all requests have 4 Millisecond (ms) of overhead.

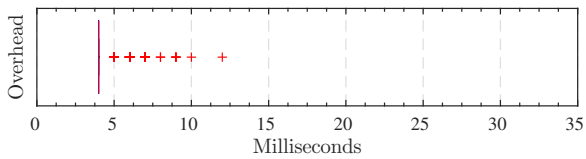
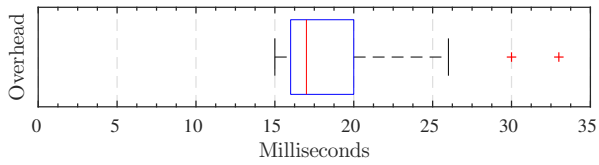


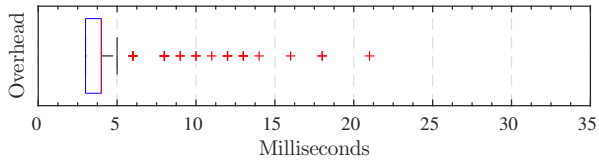
Fig. 5.5 Microcontroller base overhead on GET requests

In contrast, for measuring the overhead of the PUT requests, we deployed the microcontroller to the `onPut` trigger of a bucket, and we made 200 object PUT requests to it. For this experiments we calculated the overhead of both using the `Memcached` system and without using it. The results depicted in Fig. 5.6a, show that almost all requests have 17ms of overhead without using the metadata caching system. The extra overhead regarding to the GET requests, as explained in Section 5.4.1, is due to that the interception middleware needs to query, for each PUT request, the parent container in order to know the microcontrollers

to be applied (Query means make a HEAD request to the Swift container or pseudo-folder). In the second case, we used the `Memcached` system which stores the metadata that the interception middleware needs to put into execution the microcontrollers. With `Memcached`, the interception middleware only needs to make one HEAD request to the Swift parent bucket, then, it stores the information in memory, so that the rest of the requests that comes in to Swift, and to the same bucket, can use this cached information. As depicted in Fig. 5.6b, the median overhead with the caching system is around 4ms. Base overheads shown in Fig. 5.5 and Fig. 5.6 concern to all applications whose microcontrollers interfere in the object request, that is, synchronous microcontrollers. Asynchronous microcontrollers will not produce overhead as the request is not blocked.



(a) Without Memcached

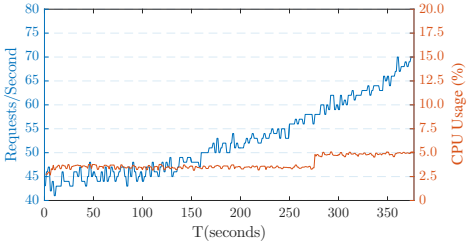


(b) With Memcached

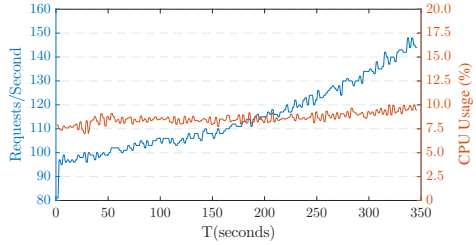
Fig. 5.6 Microcontroller base overhead on PUT requests

Microcontroller resource consumption. Second, we evaluated the resource consumption of microcontrollers. As they are executed where the data is, and as they can only leverage the storage node resources, it is critical to measure the resource consumption to show the impact of microcontrollers to the system. For this experiment, we used the SSBench [139] toolkit, which is built to test Swift deployments. We also prepared a special cluster configuration for the experiments, leaving only one proxy server and one storage node, in such a way that all requests go to the same storage node. This configuration will provide a better perspective of the resource usage of microcontrollers in a single node. To do so, we used the automated prefetching microcontroller, since it is the one

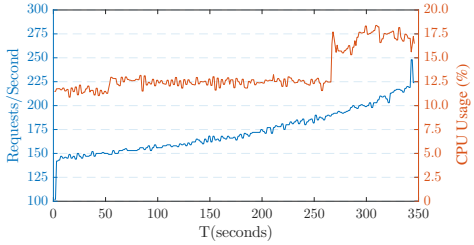
which spends more time in its execution, and therefore consumes more resources during more time. We deployed this microcontroller to the `onGet` trigger of a bucket where the SSBench will PUT and GET the objects for the benchmark. Fig. 5.7 shows the results. As we did not observe high memory usage after doing all experiments, we only depicted the CPU usage in the plot.



(a) From 40 to 70 req./s



(b) From 80 to 150 req./s



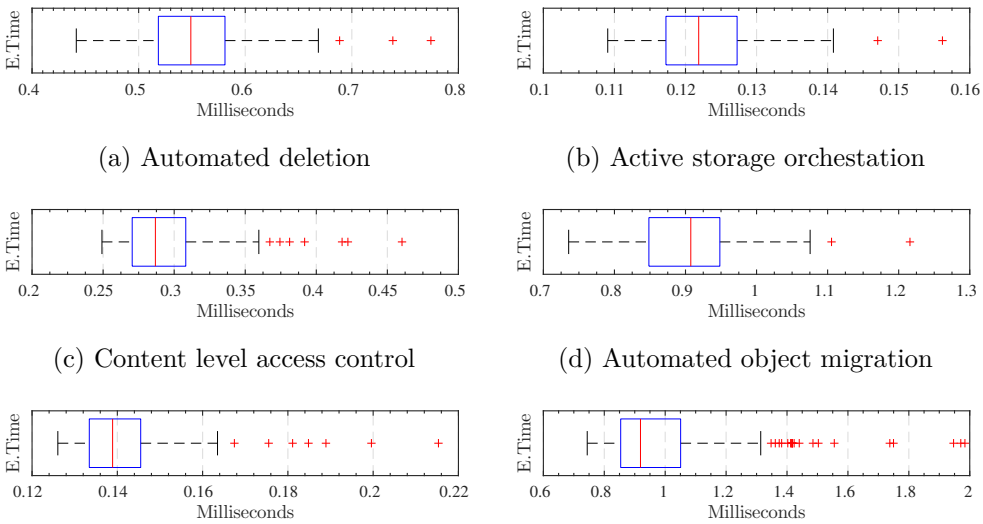
(c) From 150 to 250 req./s

Fig. 5.7 Microcontroller CPU usage for different workloads bursts

We first configured the SSBench in such a way that it performs from 40 up to 70 requests per second. In total it made 10000 GET requests against Swift for this experiment. Fig. 5.7a shows the results of this experiment. As depicted, the Docker container for this configuration consumes between 3% and 5% of the total available CPU. Due to the low CPU consumption observed, we decided to increase the workload. For the second experiment, we set the SSBench in such a way that it performs from 80 up to 150 requests per second. In this case, the workload contains a total of 20000 GET requests. As depicted in Fig. 5.7b, the launched microcontrollers consume between 7.5% and 10% of CPU, which may still be a low CPU consumption. Trying to found a limit in the number of microcontroller executions, we set another workload in such a way that it performs from 150 up to 250 requests per second. In this case, the workload

contains a total of 30000 GET requests. As depicted in Fig. 5.7c, with this configuration the microcontrollers consumes between 10% and 18% of CPU. This experiments clearly demonstrate how light are microcontrollers.

Microcontroller execution time. Third, we evaluated the microcontrollers execution time, that is, the whole time that the Microcontroller Engine spends to execute all the microcontroller code. The plots depicted in Fig. 5.8 are the results of 200 executions of each microcontroller. Overall, the median execution time is very low, in none of them reach 1ms. For automated deletion (Fig. 5.8a), the microcontroller spends 0.55ms on its execution, as well as the active storage orchestration microcontroller (Fig. 5.8b) spends around 0.12ms, the content level access control (Fig. 5.8c) 0.28ms, and the static resource extractor (Fig. 5.8e) 0.14ms. Otherwise, the automated object migration (Fig. 5.8d), and the automated prefetching (Fig. 5.8f) microcontrollers are closer to 1ms, both spend around 0.9ms. The reason of this longer execution times is that, in contrast of previous microcontrollers, these last two ones are more complex, and they need to do more actions to complete their execution. More complex instructions becomes in a longer execution times. Anyway, this experiment shows the lightweight of the microcontrollers for the proposed applications.



(a) Automated deletion

(b) Active storage orchestation

(c) Content level access control

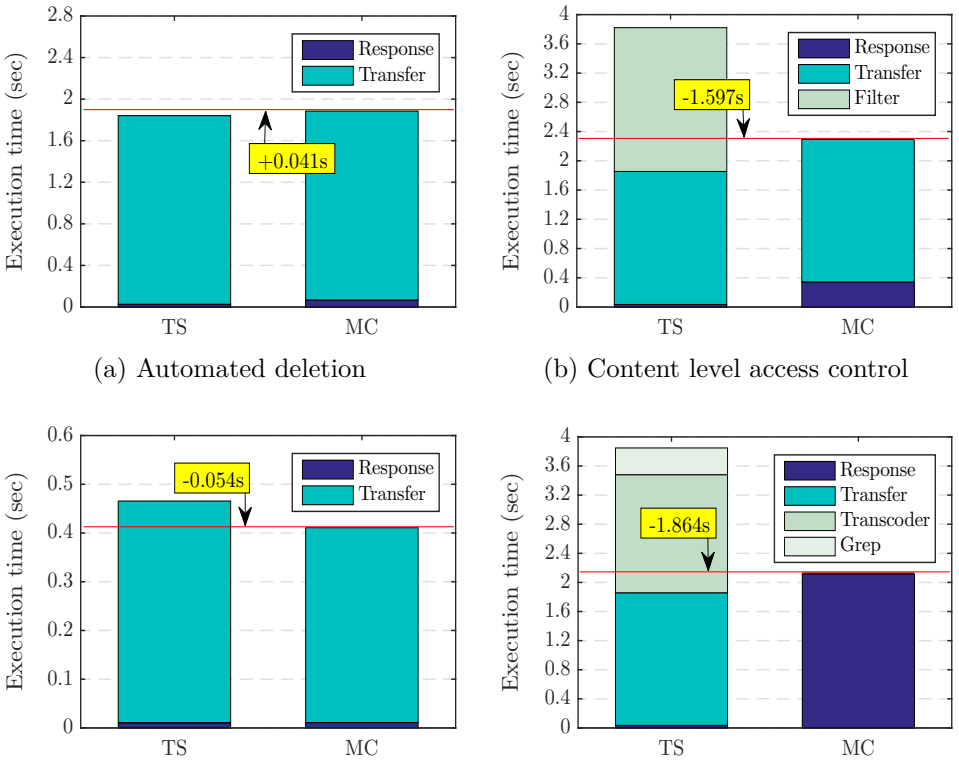
(d) Automated object migration

(e) Static resource extractor

(f) Automated prefetching

Fig. 5.8 Microcontroller execution times for the different applications

Application execution time. Fig. 5.9 shows the execution time breakdown for the different applications. The plots also show a comparison between how the applications behave with a traditional storage (TS) system (without Vertigo), versus how they behave by using microcontrollers (MC). We evaluated here the synchronous applications, in addition to the automated object migration (asynchronous), because the automated prefetching (asynchronous) does not provide any improvement compared to TS. 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.



(a) Automated deletion

(b) Content level access control

(c) Automated prefetching

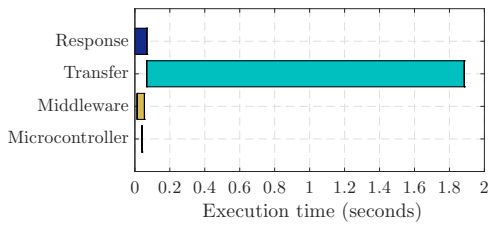
(d) Active storage orchestation

Fig. 5.9 Request execution time breakdown for the different applications

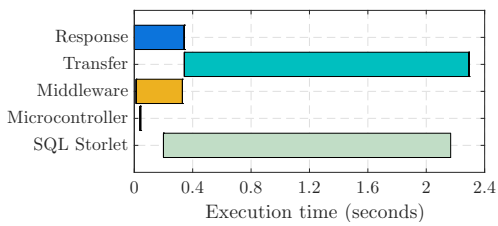
Overall, the system overhead is very low in all applications. For automated deletion, the overhead is only of 41ms. For the content level access control use case, the TS configuration needs to download all the file before filtering it. However, the MC configuration only requires to transfer 30% of the 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. 5.9b. Despite this, the MC configuration saves 1.6 seconds in comparison with the TS one.

In the automated prefetching application, the fact that all the images are preloaded into the proxy server saves around 54ms in the whole execution time (Fig. 5.9c). Finally, Fig. 5.9b reports the execution time of the active storage orchestration use case. With no microcontrollers (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 host saves around 1.86 seconds, as Swift only needs to transfer the result of the `grep` Storlet.

Timeline. Timeline refers to all what happens when a requests arrives to the cluster. We discuss the behavior of the requests both from the perspective of the client side (response and transfer) and from the server side (middleware, microcontroller and Storlets). In this case, we measured the automated deletion, active storage orchestration, content level access control, and automated prefetching applications. The rest of applications that use synchronous microcontrollers and do not manage any Storlet (as is the case of the automated object migration), the timeline would be similar to the automated deletion. The rest of applications that use asynchronous microcontrollers, the timeline would be similar to the automated prefetching. The results are depicted in Fig. 5.10 - 5.11.



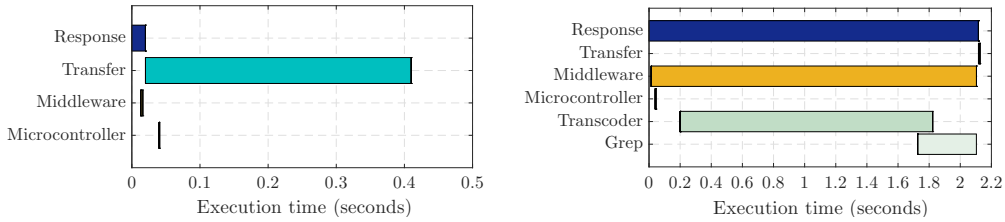
(a) Automated deletion



(b) Content-level access control

Fig. 5.10 Execution timeline breakdown for automated deletion and content-level access control applications

For *automated deletion*, the `onGet` microcontroller does not need to call any Storlet, as shown in Fig. 5.10a. Hence, when the microcontroller informs the interception middleware about this fact, Swift can start sending data to the user. In parallel, the microcontroller updates the number of accesses and checks if it is necessary to delete the object. For the *content level access control* use case, the interception middleware must wait for the SQL Storlet to start, introducing some overhead into the system as shown in Fig. 5.11a. Despite this, the microcontrolled version of this use case is still more efficient due to the data reduction that the Storlet provides as discussed before.



(a) Automated prefetching (b) Active storage orchestration

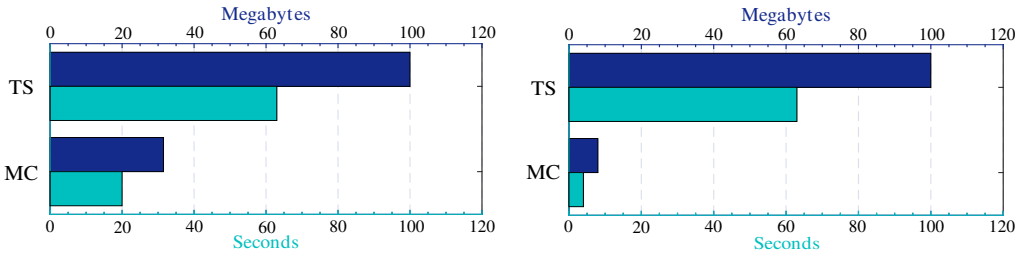
Fig. 5.11 Execution timeline breakdown for automated prefetching and active storage orchestration applications

For the automated prefetching use case, the resulting timeline plot (on GET requests) is shown in Fig. 5.11a. This use case is particular because the microcontroller is only executed if the web page is not in cache, adding 4ms of latency only in the first GET request. As shown in the figure, the microcontroller is executed asynchronously. Upon the first GET request, the microcontroller will load the static resources to the proxy server cache. 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 cache, saving I/O bandwidth and time.

In the timeline of the *active storage orchestration* use case, the microcontroller adds very low overhead, however, in this case, it is needed to take into account the Storlets that it puts into execution. 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, adding a little bit overhead when a user requests an object.

Bandwidth Usage. In those applications where a microcontroller orchestrates data reduction techniques through Storlets, the usage of Vertigo becomes in

bandwidth savings, and consequently, in a reduction of the total request time. This is the case of the content level access control and the active storage orchestration applications. Fig. 5.12 illustrates the total volume of data received by the client host, and the total request time in both configurations (Traditional Storage vs. Microcontrolled). As clearly shown in this figure, Vertigo can save significant bandwidth in both applications. For instance, for content level access control, it can avoid transferring 64,5 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.



(a) Content level access control (b) Active storage orchestration

Fig. 5.12 Comparison of bandwidth usage and total request time (TS vs. MC)

Automated object migration. The automated object migration application is an offline management tasks, and as a consequence it does not offer any kind of improvement beyond the automatic movement of the objects. However, it allows us to measure the overhead added by a Vertigo soft link.

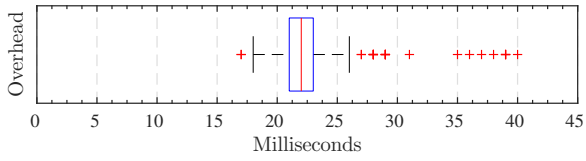


Fig. 5.13 Soft link overhead

Fig. 5.13 shows that getting a zero-byte object and route the request to the original one, swift spends around 23ms. As explained in section 5.4.1, the information of the new location of the object is stored for a time in a cache, so for the next requests, the overhead would be near to 0ms.

UNIVERSITAT ROVIRA I VIRGILI
ENHANCING THE PROGRAMMABILITY OF CLOUD OBJECT STORAGE
Josep Sampé Domenech

5.8 Summary

This chapter presents a new distributed architecture for cloud object storage that enables the self-management of the data by tenants. Our architecture introduces the novel concept of object-based microcontrollers as a decentralized mechanism to transparently extend and intercept object stores. Our microcontrollers are executed in a sandboxed environment in the storage nodes and they can intercept any lifecycle request in the desired objects. We demonstrated how microcontrollers increase the programmability and data management of object stores with concrete examples: prefetching, active storage orchestration, content-level access control, and automated deletion. Furthermore, we demonstrated that our interception framework is very lightweight achieving low overheads. In this sense, object-based microcontrollers can become a useful programming abstraction for extending object storage systems.

UNIVERSITAT ROVIRA I VIRGILI
ENHANCING THE PROGRAMMABILITY OF CLOUD OBJECT STORAGE
Josep Sampé Domenech

Chapter 6

Scaling-out Policy Enforcement in the Data Plane

6.1 Introduction

In Chapters 4 and 5 we aimed to extend the programmability of cloud object stores by adding a multi-tenant management layer for administrators, and a data self-management layer for tenants, respectively. The resulting systems make use of a technique called active storage, which benefits of the data locality of the storage service by placing computation tasks close to the data, where the data is stored. In particular, active storage has two major benefits: On the one hand, it leverages the underutilized compute resources of the storage infrastructure, and on the other hand, it considerably reduces the network traffic.

In spite of this, when the compute requirements of the well proven practice of collocating compute and storage are high, active storage has been shown to be not suitable for all cloud storage infrastructures such as object stores, principally due to two main reasons. First, the *Elasticity*: By collocating compute and storage, computing resources can only scale out with the number of storage nodes. It is thus impossible to provision compute power independently of the storage, and hence, fulfill the promise of elasticity. And second, the *Resource contention*: One of the main cares of tenants is response time, in particular, for inline real-time services. However, running multiple application functions at the same time in the storage nodes can lead to resource contention problems. And what is worse, it

The results presented in this chapter are published in [154]

UNIVERSITAT ROVIRA I VIRGILI
ENHANCING THE PROGRAMMABILITY OF CLOUD OBJECT STORAGE
Josep Sampé Domenech

can terribly affect the performance of other tenants who share the same storage infrastructure. While resource contention has been addressed to a certain level by limiting the resources used by compute tasks [114, 113], it cannot be solved with storage resources alone.

6.1.1 Scope and Challenges

To overcome these limitations, in this chapter we argue that the new promise of *serverless computing* can represent an alternative yet powerful solution to these problems. A key insight is that a serverless execution model is ideal to rapidly scale up and down compute capabilities without the need to manage any server. Moreover, it usually makes use of Linux containers to run tasks in an isolated manner, which at the same time simplifies the scalability. In this sense, serverless computing fits perfectly in terms of portability, since our previous works are also built on Linux containers.

Regarding cloud object stores, serverless computing [51, 52, 155, 156] is designed following the event-driven model, where functions are triggered in reaction of PUT requests. Once a file or files are already uploaded to a bucket, the associated functions are executed. Functions run in disaggregated compute clusters, having to move the data from the storage cluster to process it, eventually storing back the results. Although functions can also be proactively invoked by using a gateway, and through a customizable API, these main characteristics represent a problem regarding the systems built in Chapters 4 and 5.

Both Crystal and Vertigo are designed to intercept all the object lifecycle requests, and to process the data inline and in situ, thus running computation tasks close to the data. In this sense, our focus is on a *data-driven serverless computing system* integrated in the object storage infrastructure, and able to intercept all the object lifecycle requests, which is not possible with the current systems. As the main challenge of this chapter, this ambitious goal requires the design of the first data-driven serverless computing platform for object stores.

Design of a data-driven serverless computing model: Although nowadays serverless computing frameworks allow to process data from cloud object stores, they do not allow for inline processing, thus not taking advantage of the data locality. In this sense, characteristics such as data locality, programmability, simplicity, extensibility, elasticity and scalability should be the cornerstone of our system. This entails the design of a novel serverless framework capable to be

integrated within the storage infrastructure, in the storage path, but decoupled from the storage nodes. First, it requires the capability of synchronously process the data, with no timeout limitations[1]. Second, it requires a distributed monitoring system for 1) routing the requests to the appropriate computing node, and 2) provisioning the correct level of elasticity to the system.

Other characteristics are shared with the previous systems built in this thesis. For example, it requires the system to be easily usable by non-expert cloud tenants. Also, it must provide a simple API in order to create, modify and delete functions, and to attach and detach them to the objects. And finally, as in Vertigo, as object stores usually store 3 copies of the objects, the system must have high consistency in order to guarantee the correct operation when the same object is accessed at the same time through different copies, which is not trivial to implement in an object storage system.

6.1.2 Contributions

In this chapter, we present a novel *data-driven* serverless computing framework for cloud object stores called Zion. We aim to solve the scalability and resource contention problems of active storage, while benefiting from data locality to reduce latency by placing computations close to the data. Our model is data-driven and not event-driven, because tasks are located in the data pipeline, and intercept the data flows that arrive and return from the object store.

Our computing framework makes previous systems built in this thesis less intrusive in the storage infrastructure, while keeping data locality. In practice, there are many jobs that require functions to be part of the data pipeline to provide optimal response times. For example, those that require of synchronous interaction between tenants and cloud object stores as is the case of Crystal and Vertigo. Examples of these use cases include (but are not limited to) dynamic content generation, interactive queries, content verification, access control, which are better suited for a data-driven serverless computing model. Most of these use cases are hard or impossible to implement in current serverless platforms due to operational requirements, for instance, because they need to transparently intercept incoming requests to the object store, or because they require interactive communications through the standard object store's API.

[1] Among other limits, serverless computing platforms have function timeouts. Usually, they allow configurable execution times up to 10 minutes.

UNIVERSITAT ROVIRA I VIRGILI
ENHANCING THE PROGRAMMABILITY OF CLOUD OBJECT STORAGE
Josep Sampé Domenech

In contrast, our model is a lightweight solution that allows tenants to create small, stateless functions that intercept data flows in a scalable manner without the need to manage a server or a runtime environment. We coded a prototype implementation of Zion for OpenStack Swift, where the serverless compute layer lies between the proxy and storage nodes. This has allowed us to maximize write and read per-worker performance to storage nodes. Finally, we demonstrate the feasibility of our model, and we show how our solution scales up with minimum overhead and no resource contention through different practical use cases. With free containers available, Zion's overhead is around 9ms, which is well amortized by the functions' execution time, typically in the seconds (or minutes) range, for the possible data-driven use cases.

6.2 Design Overview

Zion has been designed for scalable, data-driven execution of small functions in object stores. And thus, all components revolve around the object storage service. Zion's design therefore assumes that the underlying object store follows the "classical" architecture of load balancers distributing the workload evenly across the gateways or proxies, and a large pool of storage nodes, which are in charge of storing the objects in the hard disks.

As a first overview, Fig. 6.1 shows a diagram of Zion's architecture. To not interfere with both plain `PUT/GET` requests and storage management tasks, such as replication and failover, Zion lays out a disaggregated computing layer between the storage and gateway nodes for executing the functions. Zion also integrates a metadata service and an interception software running in the storage gateways, which inspects incoming requests and reroutes them to the compute tier if necessary. Moreover, as we can see, this model is even less intrusive than Crystal or Vertigo, as it does not require to integrate or modify any component in the storage nodes.

6.2.1 Interception Software and Metadata Service

The first component of the system is the interception layer, which is integrated in the storage gateways (depicted as a router in Fig. 6.1). The major aim of this software is to manage the deployment of functions, the association of triggers to these functions, and their execution when a request matches a trigger.

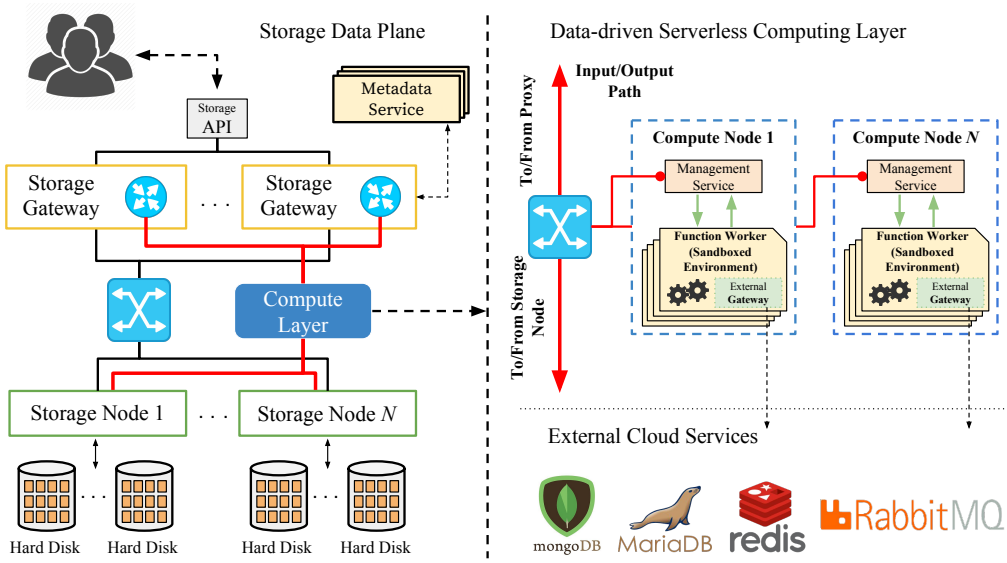


Fig. 6.1 High-level architecture overview of Zion integrated alongside a common cloud object storage architecture

A trigger is a combination of a URL, with prefix and suffix filters (similar to AWS Lambda for Amazon S3) and a HTTP method (GET, PUT, POST, HEAD, or DELETE). This interception mechanism is enough for many use cases. By specifying the suffix `.txt` as a filter, for instance, Zion can run a compression function to all GET requests for text objects. The list of available triggers is the following.

- **onPut**, **onPost**, **onHead**, and **onDelete**, which cause the execution of the associated function whenever a PUT, POST, HEAD, or DELETE request is received, respectively. As an example, the `onPut` trigger can be useful to process an object before its write to the object store, and even discard its storage, as processing is part of the write path, and not asynchronous as in AWS Lambda.

- **onBeforeGet**, a function associated to this trigger is ran when a user performs a GET request to the storage service. This is one of the two cases associated to the GET requests. In this case, the function is executed before forwarding the request to the storage node, and hence, the function cannot process the targeted data object. However, this trigger can be useful in

UNIVERSITAT ROVIRA I VIRGILI
ENHANCING THE PROGRAMMABILITY OF CLOUD OBJECT STORAGE
Josep Sampé Domenech

many use cases like HTTP headers processing, URL rewriting, temporarily redirects, etc.

- **onAfterGet**, which causes any function associated to this trigger to run on an incoming GET request to the storage service. In this case, the function intercepts the storage node's response, and therefore, it can dynamically manipulate the object's content.

The metadata service is the second architectural component of Zion. Metadata for triggers is pre-processed and indexed efficiently in the metadata service to guarantee a small, $\mathcal{O}(1)$ request matching overhead, of the order of μsecs. If there is any match, this layer is also responsible for redirecting the input flow as the object is read to an available *worker*. Non-intercepted data flows rapidly follow the default storage path and bypass the serverless compute layer as usual.

It is worth to note here that the fact that a serverless execution model can quickly spawn new *workers* is what makes it possible to intercept and process data flows "on-the-fly" without collocation. With VMs, interception will be significantly more difficult to achieve as VMs can take minutes to start.

6.2.2 Computation Layer

The third architectural component is the serverless computation layer. The computation layer is a pool of containers which puts the functions into execution.

Functions

A function is the computation code unit which can process the data. In our model, functions are data-driven, that is, they are focused to intercept the data flow, and process the data inline, as the object comes-in or comes-out from the storage cluster. Because of this, in our model, the response time from the functions (time to first byte) must be fast to not affect the user experience.

In addition to processing the data stream, we integrated in functions the flexibility of microcontrollers. Thus, functions can perform most of the same actions, such as store information in a persistent way (e.g., an access counter, the timestamp of the last access, etc.) Concretely, a function can take one or all of the following actions after intercepting a lifecycle request of an object:

- It can add, update, or delete the metadata of the current object.

- It can generate new requests to the object storage service. This includes: GET an object, for example, a dependency needed to process the main stream. PUT an object, for example, PUT a subset of the main object as it is processed. Delete an object, and POST metadata to another object;
- It can generate new requests to other services (e.g. Rabbit, Mongodb, etc.), for example, to store some relevant information extracted of the main data object stream.
- It can update the request/response headers of the request.
- It can `cancel` the request, or `rewire` the request to another object.

Functions may make use of third-party library dependencies in order to achieve a specific behavior. Once developed, functions should be packed with them within a TAR file. Therefore, in conjunction, functions and their dependencies must be lightweight, in such a way to minimize the time needed to transfer the function package from the storage system to the compute node.

Once packed, a function is uploaded as a regular object to the object store. An interesting feature of Zion is that it allows the user to set up the CPU and memory requirements, and a *timeout* value for every function. The timeout is the amount of time the system waits to receive the first byte of the function's output. If the function times out, the request is automatically canceled. This information allows functions to be configured differently in order to better manage certain jobs. This information is not mandatory, and Zion has the last word, assigning default values when necessary.

Further, Zion's functions accept parameters. Parameters can be explicit or implicit. Explicit parameters are provided as headers in the request. Implicit parameters are default parameters that a user specifies ahead of time in the moment of associating a trigger with a function. Explicit parameters take precedence over implicit ones in the case of collision. As an example, consider a function to resize an image and the image resolution as a parameter. If no argument was passed in the request, the implicit image resolution would be taken from the function's metadata, or an error would be thrown accordingly. The same function can have different parameter sets for different triggers.

A final important remark is that two different functions cannot intercept the same data flow in Zion, unless they do it in a pipeline fashion, one after another, which raises no consistency issues.

Compute Node

In a compute node, functions are executed inside isolated environments or Linux containers:

Containers. Each function has its own Linux container in order to not interfere with the other cloud functions. A container with a function running inside is called a *worker*. A function may have zero, one, or more workers running at the same time, depending on the workload. In the traditional function model, starting a new worker takes around $6-7$ seconds [157]. One requirement of our model is that functions have to start running as soon as possible, for this reason we leverage a ready-to-use pool of containers. In any case, our experiments verify that starting a new container takes around 0.9 seconds, which is practical enough for many synchronous and near-real-time applications. After a certain period of time, the idle workers are stopped and the corresponding containers are recycled in order to better optimize resource consumption.

Zion Service. This service manages the requests forwarded from the interception layer (6.2.1). When a new request arrives, it takes a container from the pool, installs the libraries and the function code, and sends the execution command to the runtime. As functions may have specific CPU and memory requirements, this service is also in charge of establishing the resource limits to containers according to their configuration parameters. It also load balances the workload across already started workers, and starts new workers if necessary, thus providing of the elasticity of the system.

Runtime. The runtime is integrated into the containers. It accepts functions for a specific programming language, and puts them into execution. Our prototype oz Zion currently supports Java, but other languages such as Python would be easy to integrate.

6.3 Prototype Implementation

We implemented a prototype of our serverless computing framework [158] on top of OpenStack Swift [13]. We have decided to make Zion the less intrusive as possible. As a result, the only modification in the base Swift architecture is a Swift middleware which intercepts the requests at the proxy servers side. The other elements are decoupled from the main Swift architecture, which makes Zion easier to deploy.

6.3.1 Interception Software and Metadata Service

In Swift, the simplest way to intercept requests is to create a Swift middleware. We built a new Swift interception middleware for Zion to accomplish two primary tasks: 1. The management of function code deployment and libraries, including the triggers that cause the functions to be run; and 2. Redirection of requests and responses through the computation layer when they need to be processed by any function.

Upon the assignment of a function, a trigger header of the type: `onPut`, `onBeforeGet`, `onAfterGet`, `onPost`, `onHead`, and `onDelete`, must be appointed to tell the framework which lifecycle events to intercept. Zion uses Redis [137], a quick in-memory key-value store, as distributed metadata service to maintain this information. To optimize request matching, Redis is collocated with the proxy nodes. Recall that as part of the metadata, Zion also includes configuration information for the functions such as the CPU and memory requirements, and cancellation timeouts, as we already discussed in the preceding section.

Function assignment granularity. Although OpenStack Swift stores the objects in a flat name space, is possible to simulate a hierarchical structure within a single bucket by adding forward slash characters (/) in the object name. Each folder between the bucket and object names is called a *pseudo-folder* in Swift. For example, in the object name: `images/zion.jpg`, the prefix `images/` is the pseudo-folder. In Zion, we take advantage of this logical hierarchy to enable function assignment at per-object, pseudo-folder and bucket levels. Moreover, we also enable mappings at suffix level, for example, to run functions to all objects whose name ends with `.jpg`.

Function execution. After the function assignment process, when a request arrives for an object, the Zion's middleware with the help of the metadata service checks if that request triggers the execution of a function. For example, if the request is a PUT, the middleware will launch an `onPut` trigger for the request. If there are functions that respond to this trigger, the middleware will immediately forward the request to an available container. Otherwise, the request will follow the default read/write path.

Parallel processing with functions. In all object stores, there is a limit in the maximum allowed object size. In Swift, this limit is 5GB. To bypass this limitation, Swift uses a special object called `Static Large Object [159]` (SLO). SLOs are objects split into different parts. The user must upload these parts,

together with a special object called `manifest`, which contains the location of the object parts. Getting an object is totally transparent to the users, who make a request to the manifest, and the whole object is returned as if it was stored as a single object in Swift.

This Swift SLO approach enables Zion to associate a different function to the manifest and to the parts. With this mechanism, it is possible to create a highly parallel and distributed computational substrate by executing a function to each dataset part, and finally, by running a reduction function to the filtered parts. In Section 6.4, we present a Hadoop-like use case for object storage implemented with our functions. This is aligned with the recent trend of large scale data analytics with serverless computing [55, 54].

6.3.2 Compute Layer

In our implementation, the computation layer is composed by a pool of compute nodes. They are located between the proxies and the storage nodes as shown in Fig. 6.1. It should be noted that the compute nodes assigned to Zion are not shared with OpenStack computing project Nova [160]; they are exclusively managed by Zion.

A general overview of how Zion operates is the following. Each function is run inside a separate container, what is called "worker". At the time of this writing, Zion's runtime is Java-based. And consequently, every function is run in a Java Virtual Machine (JVM). At a very high level, a worker can be viewed as a container running a specific function. Every new invocation to the function is handled by a new thread inside the JVM.

Functions

As the Zion's runtime is based on Java, Zion's functions are also written in Java. This means that functions are plain Java classes. They have an entry point called `invoke`, which contains the main code of the function and that it is called upon every new request. The `invoke` method has two arguments of type `Context` and `API`, which are made available automatically by the Zion's runtime on every new request.

The `Context` encapsulates the access to the request headers, the object's metadata, and the object's data stream. It also includes a logger facility for logging the execution of the function. The `API` enables access to external services.

By default, the isolation level of containers precludes functions from having any access to external resources. However, through the `API`, a function can access to some external services such as RabbitMQ and Redis in addition to Swift itself. The `API` class is extensible and has been framed to facilitate external interaction to future services. In Listing 6.1 we have the implementation code of a simple function that iterates over the data with no further processing.

Listing 6.1 A function that echoes the data passed to it.

```
1 public class Handler implements IFunction {
2    public void invoke(Context ctx, API api) {
3       while((data = ctx.object.in_stream.read()))
4          ctx.object.out_stream.write(data);
5     }
6 }
```

Compute Node

The complete architecture for compute nodes is depicted in Fig. 6.2, and it consists of Docker containers, the Zion Service, and the runtime itself.

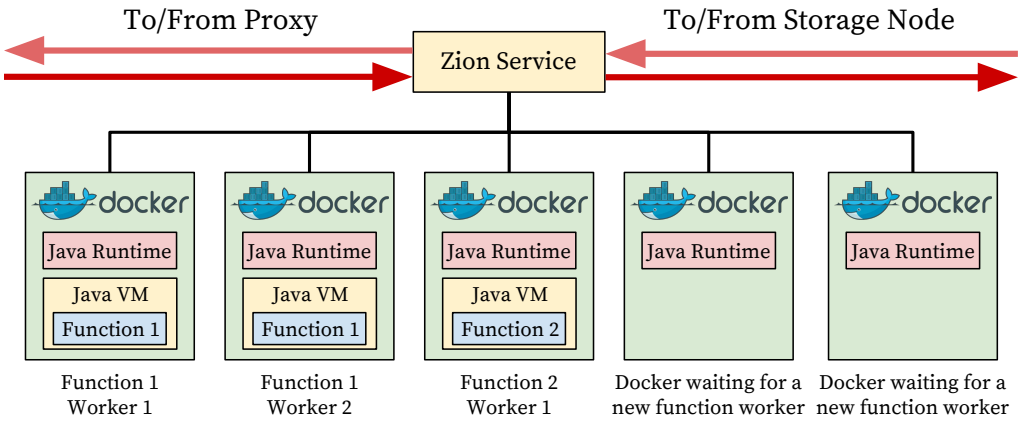


Fig. 6.2 High-level architecture overview of a Zion Compute Node

Containers. We use Docker containers to sandbox functions, so they have neither direct network access, nor system execution and thread creation capabilities, nor access to the local file system. Only a minimum ephemeral hard disk space is

provided under the `/tmp` directory to be used by functions. We used the Ubuntu 16.04 Docker image [153] for our implementation.

Zion Service. The Zion service is a Web Server Gateway Interface (WSGI) server whose mission is to spawn workers for processing the request forwarded by the interception middleware. Consequently, it must be up and running in all compute nodes, because as described in 6.2.2, it is responsible for managing containers and executing functions. When a function is running, this service waits until receiving a response from it. The waiting time is limited by the default system's timeout or by a user-defined timeout specified at deployment time.

Runtime. The runtime is integrated within Docker containers. With the Java runtime installed on them, this allows to rapidly spawn new containers and execute functions. As stated in previous section, each different function is executed in a separate container. If the compute node does not have any worker available for that function, the Zion service takes a Docker container from the pool of containers, executes it, and finally loads the function code and libraries into the runtime. Subsequent requests will then be able to reuse the same worker when processing the same "hot" function.

One feature of our serverless computing framework is that it is possible to modify the object's metadata (`{key:value}`). As we discussed in Section 5.4.2, Swift is an eventually consistent system that stores 3 copies of each object. This means that at a given time, the replicas of an object may have different data and metadata. As functions are *stateless*, that is, there is no relation between the different function invocations even if they occur in the same worker, modifying the local copy of an object's metadata implies waiting for Swift to update all the replicas. This behavior is clearly undesirable in terms of function's parallelism.

In this sense, as with microcontrollers, each function worker is directly attached to a distributed metadata caching system based on Redis. It is an internal feature, totally transparent to the users who develop functions. When a function is executed, the Runtime initially loads the object's metadata into the cache, and then, performs all the modifications over it. If there are other requests to the same object, the other functions will use this previously-loaded metadata. When the function terminates, it offloads the metadata from the cache to all object's replicas. With this distributed cache, we ensure that the objects' metadata touched by a function is always consistent in all replicas, even if there are multiple requests at the same time, and even atomically.

6.4 Applications

Here, we show some of the applications that Zion can support, and the ease with which these applications can be built on top of Swift. As expected, the described use cases are not arbitrary; they have been chosen to show the potential of our data-driven model, and in particular, for synchronous interactions with the object store. All of them have been implemented over Swift and evaluated in the next section.

Content-level Access Control. With Zion, it is extremely easy to implement sophisticated forms of access control based upon the contents of the objects themselves. As we already discussed in Section 5.6, typically, access control in object stores, such as Swift, operates at the granularity of buckets, and hence, once an object is accessible to some party, he gets the full content of the object. In this chapter, we show how content level access control can also be realized very easily in Swift thanks to our function abstraction in a similar fashion but cleaner than [44]. By "content level", we mean that Swift users will be able to access to certain parts of an object based on their credentials.

This example is ideal to show the limitations of AWS Lambda. First, it is clear that access control requires the interception of the GET requests, which can only be done indirectly with the help of the API Gateway service. Second, in AWS Lambda the response body payload it is limited to a determinate size (Nowadays: 6 MB), so it is not possible to provide large results from big datasets. Third, the processing of the object's content to satisfy the access control policy must be done inline, as the user needs a timely notification about the status of her request, which again cannot be realized with AWS Lambda. Finally, Zion's design enables access control to leverage data locality and improve response time, as functions go through the content of the object as it is read from the storage node.

To give a concrete example, consider the publicly available Adult dataset from the UCI Machine Learning Repository [161], described in 5.6. As stated in previous chapter, it 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` 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 a "police agent" should be able to access to all fields: `SSN`,

`age`, `education`, `marital-status`, `race`, `sex`, `relationship`, `capital-gain` and `native-country`, a "census analyst" could be restricted to get only a smaller view: `age`, `education`, `capital-gain`, `native-country`.

To implement this example, we have linked a function to the `onAfterGet` object trigger to enforce content level access control on the object `adult_data.csv`. We have defined a simple access policy that depending on the use role, "police agent" or "census analyst", allows to get all the fields or just an smaller projection view. This simple access policy has been stored as implicit parameter of the function, that is, in a JSON formatted string uploaded when we linked the object with the function, as explained in 6.2.2. When a GET request comes for the object `adult_data.csv`, the proxy first checks the Swift ACL. If the object is accessible by that user, the function then reads the content level policy, and filters the data, only if the user has the appropriate role.

Compression. A typical data reduction task is the compression of objects. In general, any dynamic content filtering that involves inline transformations of the content is ideally suited for our data-driven model. A data-driven task could compress/decompress a file dynamically "on-the-fly". As Zion acts directly on the data pipeline, that is, as the object is read/write to the object store, the result will be either latency improvement or space reduction depending upon whether Zion intercepts a GET request or a PUT request.

Here we will merely consider the case of compressing incoming objects "on-the-fly", which means that upon a GET request by the user, the target object will have to undergo decompression. The common approach for doing so in Swift is to implement an ad-hoc compression middleware. However, this approach is problematic. First, by running compression on the storage nodes, compression is repeated as many times are replicas there are. By offloading it to the proxies, we can disrupt Swift's normal operation under heavy load, since the proxies are responsible for looking up the location of objects in the rings and routing the requests accordingly. looking up the location of objects in the rings and routing

With Zion, we can easily write down a compression function, and execute it between the proxies and the storage nodes in a scalable way, without worrying about resources, or repeating the task many times. In addition to the possibility to intercept GET requests, Zion has another advantage over AWS Lambda. It is capable to run compression over objects whose total compression time exceed the five minutes' limit. This is because Zion cancels a request only if the time

for the receipt of the first byte from the function exceeds a timeout value (see Section 6.2.2 for details). This model is ideal for operations such as compression that can be run as data is read/write, which have been the focus of active storage for a long time.

To implement the compression function, we utilized `gzip` and then, we mapped it to a bucket with the `onPut` trigger. As such, all writes to this bucket will be compressed. We did the reverse process for the `onAfterGet` trigger, so that when a user requested an object of this bucket, she would get the original uncompressed version. To do so, we made use of an implicit parameter to tell the function what to do: either to compress or decompress. That is, for the `onPut` trigger, the implicit parameter was set to "compression". For the `onAfterGet` trigger, the implicit parameter value was set to "decompression".

Image processing. One of the archetypal use cases of serverless computing is that of image resizing, for we found it interesting to evaluate it here. It is ideal to show the potentials of asynchronous, event-based functions such as AWS Lambdas and also very useful for tenants that use the object store as a back-end for web image storage. In this case, when the images are uploaded, a function is triggered, resizing and creating all the needed images, for example, for the different possible devices that can request the main web (e.g. smartphone, PC, tablet, etc.).

We did the same with Zion, and coded a function that resizes an image to an arbitrary percentage. To intercept the PUT requests, we linked it to an `onPut` trigger and specified `.jpg` as the suffix of the object name. As Zion allows to create new objects as part of the function's output, the function stores the original object and its resized version(s).

Because of interception, one interesting feature of Zion is that it does not require to fully store the image before the generation of its resized version(s), since it is done "on-the-fly" prior to storage, saving storage bandwidth. Although of not much concern at first glance, this property is very interesting for concurrently processing a vast collection of images as in [54], because IO savings add up rapidly for a large amount of objects.

Signature verification. To demonstrate the versatility of Zion, we proposed the signature verification use case. Online content checking is again a perfect use case for data-driven functions since it requires an immediate response to the client. The main objective of this function is to verify that the objects (documents) that

are uploaded, are signed by the user, and to verify that the signature is valid, that is, the documents are authentic. With Zion is possible to do this task in near-real time, and notify the user instantly in the case of rejection. Also, in the case of rejection, we prevent the archival of an inauthentic document, thereby saving storage space. For the same reasons as above, it is readily evident that this use case cannot be implemented with AWS Lambdas.

The scenario is as follows: The user obtains the public key from her RSA key pair, and she uploads it to a public bucket in the object storage system. Then, the user signs the document with the private key, and uploads it, with the signature in a special header to the object storage system, which puts into execution the function and verifies the document. Note that in this case, the function uses an explicit parameter (signature) described above.

To do so, we coded a signature verification function. The function is mapped with a bucket to the `onPut` trigger. Therefore, all object PUTs to this bucket will be enforced. The function first gets the public key from the object store based on the user who is uploading the object. Then, it loads the document content and it verifies the signature. If it is valid, the document is stored, otherwise the document is rejected, sending an error message to the user.

Interactive queries and result aggregation. Finally, interactive queries is a use case that perfectly matches our data-driven model. When we want to perform fast data queries over existing data repositories, our data-driven model avoids moving the entire dataset to a computing cluster.

For example, object storage services are commonly used to archive data like log files. Businesses that want to extract fast insights from these data repositories using big data analytic tools must choose between two strategies: 1. Moving data to the computation cluster to take advantage from data locality; or 2. Using a connector to allow data analytic tools to read data directly from the remote object storage. With Zion, we offer a third strategy: Compute in the storage cluster using functions that filter data and aggregate results inline.

The case example we present is a top-k query on access logs of UbuntuOne [162], a personal cloud service. We want to obtain a list of the most active users and the total number of requests each user sent. The implementation of this query in Zion has two functions. The first one receives a fragment of the log file and filters requests logs maintaining a counter for each user, thus exploiting parallelism. The other function receives as input the partial counters of the

UNIVERSITAT ROVIRA I VIRGILI
ENHANCING THE PROGRAMMABILITY OF CLOUD OBJECT STORAGE
Josep Sampé Domenech

various instances of the first function, and it performs the aggregation and sorting in order to produce the desired result. To implement this mapreduce-like example, we tapped into Swift's SLO-based data partitioning (see Section 6.3.1).

6.5 Evaluation

For confirming our suspicions, we first studied how Swift behaves when some computation tasks are collocated in the same storage node. Then, we integrated the prototype of Zion in an OpenStack Swift deployment, and we ran micro-benchmarks to measure the behavior and overheads of our functions. We did so by running standalone experiments and the applications discussed in Section 6.4.

6.5.1 Testbed Characteristics

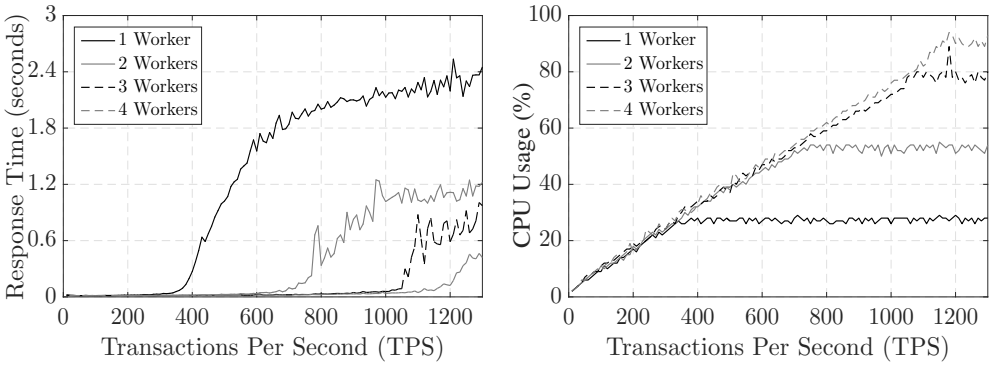
Our experimental testbed consisted of a host (or client) with 2VCPUs and 4GB RAM. On the server side, we deployed Zion in an 8-machines rack with an OpenStack Swift (Ocata version) installation formed by 2 proxy nodes Dell PowerEdge R320 with 32GB RAM and 7 storage nodes Dell PowerEdge R320 with 16GB RAM (each one with 4 CPU cores). At the compute side, the computation layer is composed by 3 nodes Dell PowerEdge R430 with 32GB RAM (each one with 24 CPU cores). All the server-side machines ran Ubuntu Server 16.04. The client host ran Ubuntu 16.04.1 CloudImage [150].

6.5.2 Swift Resource Contention

We first studied how collocation of compute and data affects Swift. To do so in "ideal" conditions, we restricted this experiment to a single storage node[2]. For this measurement, we used a single proxy node Dell PowerEdge R320 with 12GB RAM and 1 storage node Dell PowerEdge R320 with 8GB RAM.

Base Swift. The first experiment consisted of evaluating how Swift normally works. We first stored a bunch of random 10kB files in Swift. Next, using the UNIX `httperf` tool, we ran distinct workloads, each one differing in the number of transactions per second (TPS). We measured the resultant per-transaction response time (Fig. 6.3a) and CPU usage (Fig. 6.3b) of the storage node.

[2] We note that the results are easy to extrapolate to larger Swift deployments



(a) Response times (b) CPU Usage

Fig. 6.3 Usual Swift behavior in a given storage node

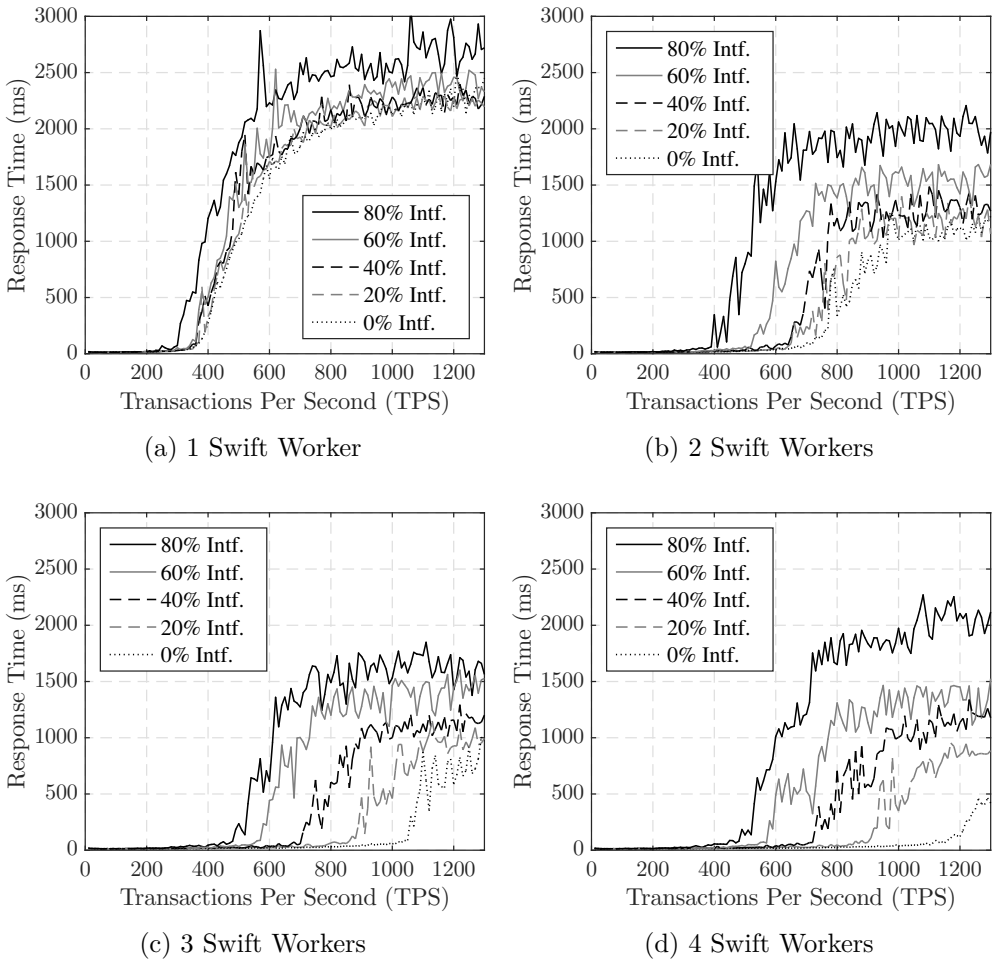
Technically, Swift uses workers to handle a workload. In Swift, a worker is nothing but a thread that accepts requests. Each worker normally accepts 1,024 concurrent requests, but it is a configurable parameter. This simple experiment confirmed us that when a worker exhausts the 100% of its core's resources, the response time steeply increases due to the queuing delays. For instance, with 1 worker, the core's usage reaches 100% around 380 TPS, the point beyond which the requests start to accumulate as shown in Fig. 6.3a. This effect can be alleviated by starting new Swift workers as seen in this figure. However, since storage nodes have 4 cores only, Swift is limited to 4 workers. And therefore, a storage node will eventually end up experiencing resource contention sooner or later even if there are no computations collocated with the data.

Collocated Computation. We repeated the same experiment but collocating computations with the storage service. The results in Fig. 6.4 show how the Swift's processing capacity diminishes as a function of the CPU usage borrowed from the collocated tasks. For example, if Swift was restricted to use only 1 worker for request processing, collocated tasks would have almost no impact on Swift. However, as the number of Swift workers increase to match the number of CPU cores, the resource contention begins to produce negative effects in the storage system due to the interference from the collocated computations.

Swift's best practices recommend to use as many workers as CPU cores. Under a heavy use of the CPU by the collocated tasks, this figure shows that Swift request processing would be severely diminished. For instance, when the

UNIVERSITAT ROVIRA I VIRGILI
ENHANCING THE PROGRAMMABILITY OF CLOUD OBJECT STORAGE
Josep Sampé Domenech

collocated tasks consumed 80% of the total CPU, Swift processing fell to a 35% in this experiment. This result suggests that *the right path to go is to move computations outside the storage nodes which is what we did with Zion.*



(a) 1 Swift Worker

(b) 2 Swift Workers

(c) 3 Swift Workers

(d) 4 Swift Workers

Fig. 6.4 Swift interference measurement in a given storage node

6.5.3 Workload

Here, we describe the specific workload used to test all the example applications described in Section 6.4. This specific evaluation setup is specially designed to test both the elasticity of Zion and the performance of each application function.

In this sense, as usually object stores can contain a huge diversity of object types and sizes [162], we set up heterogeneous workloads for all use cases in order to verify the behavior of Zion under different scenarios.

Content-level access control. For testing this application, we used the dataset described in Section 6.4, but trimmed down to different object sizes: 100kB, 1MB, 10MB and 100MB. As stated before, the dataset content is filtered out according to the type of user that requested the object. To this aim, we used Swift user roles to return only specific fields to each user. The function, associated to the *onAfterGet* trigger, reads from an implicit parameter the JSON string containing the allowed fields for a specific user's role (e.g. `age`, `education`, `marital-status`), and then returns them to the user.

Compression. It is well-known that the compression ratio of objects affects resource consumption. Objects that contain only zeros will be compressed more quickly and consume less resources than compressing a binary object. To get rid of this issue, we chose to use text documents with a similar compression ratio of around 60%. The documents were of sizes: 10kB, 100kB, 1MB and 10MB.

Image processing. As in the previous case, we tried to find the most common scenario for testing this function. We focused on those image sizes commonly used in static web pages. Finally, we used different `.jpg` files of 90kB, 400kB, 800kB and 1200 kB, and we set an implicit parameter so that the function resizes the image to the 50% of its original size.

Signature verification. A sample usage of this function may be to verify official documents and forms in a public institution. For this experiment we used text documents also of different sizes: 10 kB, 100 kB, 1 MB, and 10 MB. These documents are signed with a RSA private key. The experiment operates on PUT requests, verifying the correction of the signature.

Interactive data queries and result aggregation. For this use case, we used different sizes of the publicly available UbuntuOne's log file [162]: 100MB, 1GB, and 10GB, respectively. We compared Zion's functions execution time to those obtained using Hadoop. We built a 10-node Hadoop cluster of commodity workstations: 4-core i5 at 2.53 GHZ and 16 GB of RAM. For this application, we issued a Pig query against: 1. The log files stored in HDFS; and 2. Swift using the Apache Hadoop-Swift connector [163]. For Zion first-stage filtering functions, we picked chunk sizes of 10MB, 32MB, and 200MB for the 100MB, 1GB and 10GB files, respectively.

6.5.4 Application characteristics

Table 6.1 shows information of our four Zion application functions. The second column gives the number of *lines of code* required to execute the function. And the third column gives the *function size*. From this table, it can be seen that our functions are very lightweight for the proposed applications.

Table 6.1 Application function information

Application Function	Lines of Code	Function Size
Content-level Access Control	≈ 29	2.7 kB
Compression	≈ 8	1.8 kB
Image processing	≈ 17	2.3 kB
Signature verification	≈ 43	2.9 kB
Interactive data query	≈ 203	6 kB

6.5.5 Results

We first measured the base overheads produced by Zion. They include the runtime engine startup time, and the overheads associated to run functions in the storage path, in a decoupled compute layer.

Docker characteristics. The Java runtime, and then the functions, are executed inside Docker containers in our prototype. Consequently, the first validation to do was to assess the impact of starting our runtime within Docker containers. If the startup time was too large, it would hamper the inline processing capabilities of Zion. Our experiments, however, revealed that this is not the case.

For this experiment, we utilized a pre-compiled Docker image with our Java runtime integrated in it. The experiment consisted of launching $1,000$ containers at different rates in the compute cluster and measure the startup time and memory, along with the amount of memory for keeping the runtime up and running over time.

In this sense, Fig. 6.5 shows the results of this experiment. As depicted, starting a new Docker with the Java runtime takes between $0.85 - 0.95$ seconds. Regarding RAM consumption, we got that each container consumes around

35MB of memory. These results show how our runtime takes 5X times less to start in comparison with the AWS Lambda's runtime, which proves that Zion is lightweight enough for elastic inline processing. As Zion utilizes a large pool of already started containers, the start up time is typically negligible in many cases, and only amounts to 0.9 seconds if there are no free containers in the pool.

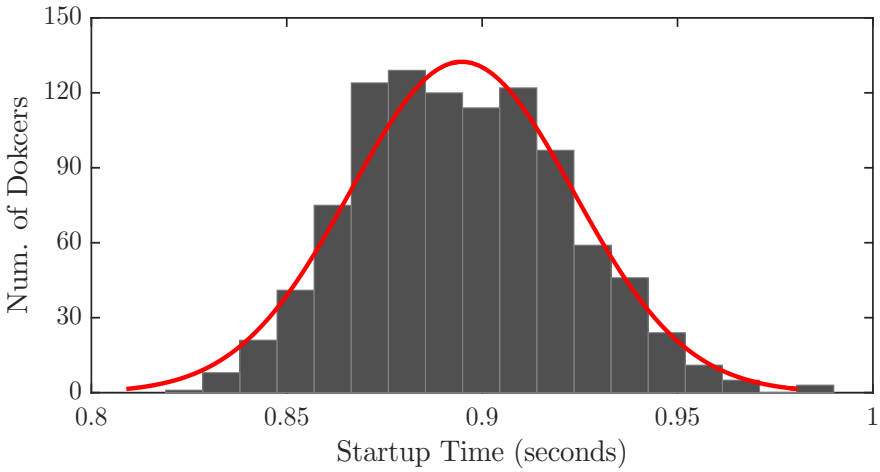
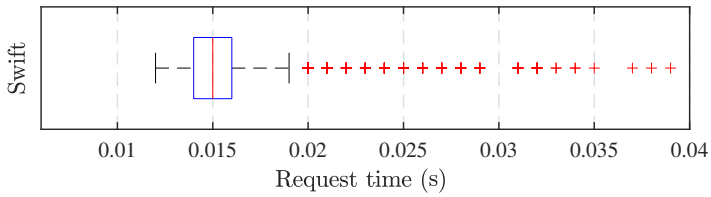


Fig. 6.5 Zion runtime startup time

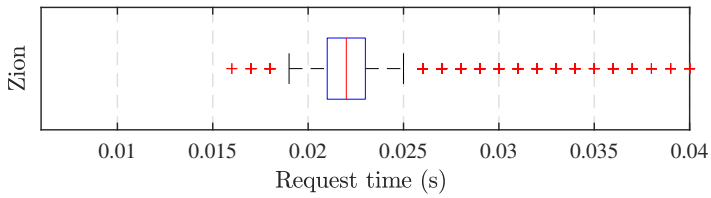
Zion overhead. As another basic experiment, we studied the extra overhead that Zion adds when running a function. The overhead is the time needed by the interception middleware to redirect the object to a compute node, take an available container, launch a function inside the Docker, and pass the object data through it. To do so, we made use of the function listed in Listing 6.1.

This simple function iterates over the data without processing it. For measuring the overhead, we used a set of 10kB objects. We first launched 5,000 plain GET requests to measure the base time needed to complete a GET request. Then, we launched another 5,000 GET requests for the same objects, but in this case, we set up the function in Listing 6.1 to respond upon the `onAfterGet` trigger of the objects.

The results are plotted in Fig. 6.6. This figure shows that the Zion's overhead is 9ms. This time includes the penalty of 5ms for the addition of an internal hop to Swift (redirect the data through a compute node), plus 4ms to determine whether and which function to run inside a free Docker container in the pool when the request touches a compute node.



(a) Base Swift request time



(b) Swift+Zion request time

Fig. 6.6 Zion base overhead

Performance of collocated functions. We evaluated here the performance of our application functions. For this evaluation, we conducted a stress test for each combination of application, object size and number of function workers. As our storage nodes have 4 CPU cores, we ran each function on 1, 2, 3 and 4 function workers, respectively. The Docker containers were set up with 1 CPU core and 512MB of RAM per worker. The results are depicted in Fig. 6.7-6.10.

The first experiments were done collocating the functions in the storage nodes without using Zion. The main objective of this experiment was: 1. To show how collocation limits the number of TPS; and 2. To show the performance of Zion's functions using 1 CPU core per function worker. As a side effect, notice that the latter makes it possible to faithfully generalize our results to any number of cores. We empirically verified this for the image resize use case. As the cluster is made up of 6 storage nodes, the function with 1 worker was able to process around 9 images per second for the 400kB image case, and around 54 images per second in the entire cluster. For the rest of applications, we utilized only one storage node. The reason is that to tap into the performance of the entire cluster, we would have to replicate 6 times each object to ensure that a copy of it is available at every storage node. In active storage, the total compute power for an object is proportional to the number of replicas it has.

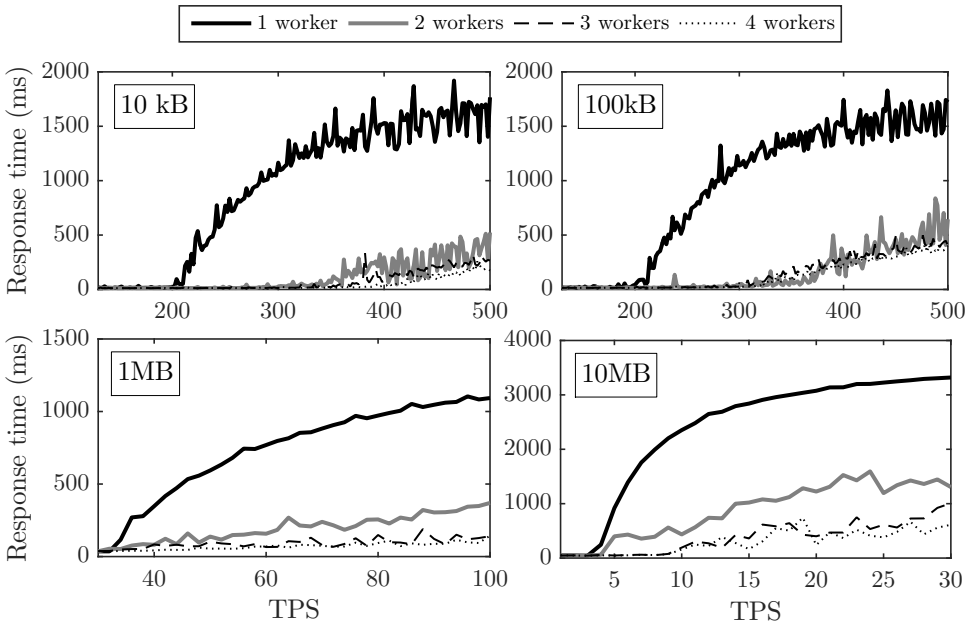


Fig. 6.7 Compression function performance

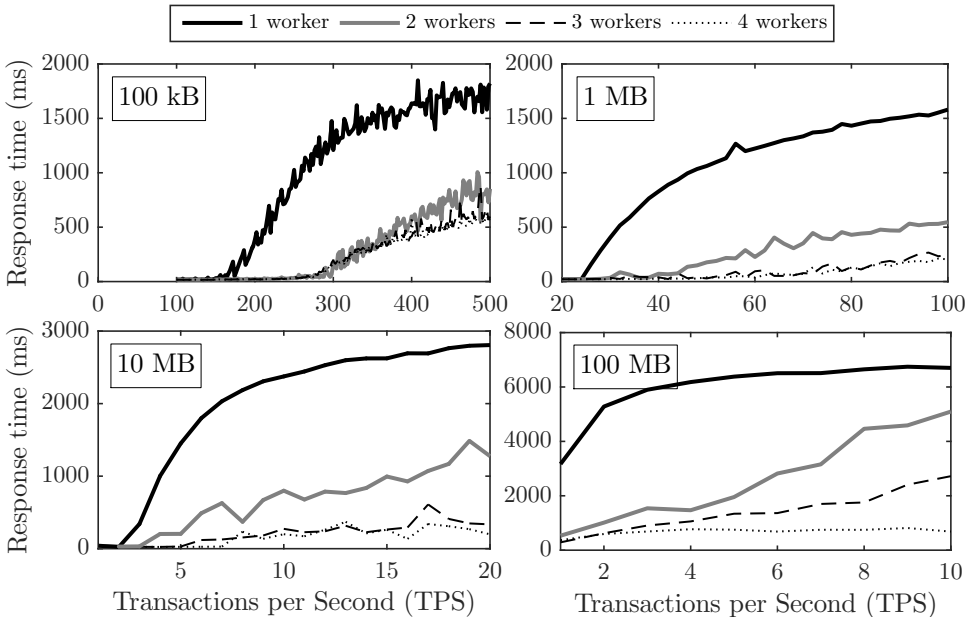


Fig. 6.8 Content-level access control function performance

Notice that in Fig. 6.7- 6.10, the curves representing the response time start to increase when the resources of the Docker container are overloaded, that is, when all the CPU cores are at the 100% of their capacity. These experiments also show how the object size is very important. In all experiments, the higher the object size, the lower the number of TPS a function can handle. To wit, for the content-level access control application (Fig. 6.8), the function is unable to handle more than 1 TPS for the 100MB object, irrespective of the number of workers. Concretely, this task takes around 1.6 seconds to complete. The same occurs with the image resizing function. We saw that resizing an image of 1200kB takes around 2.36 seconds, leading to < 1 TPS.

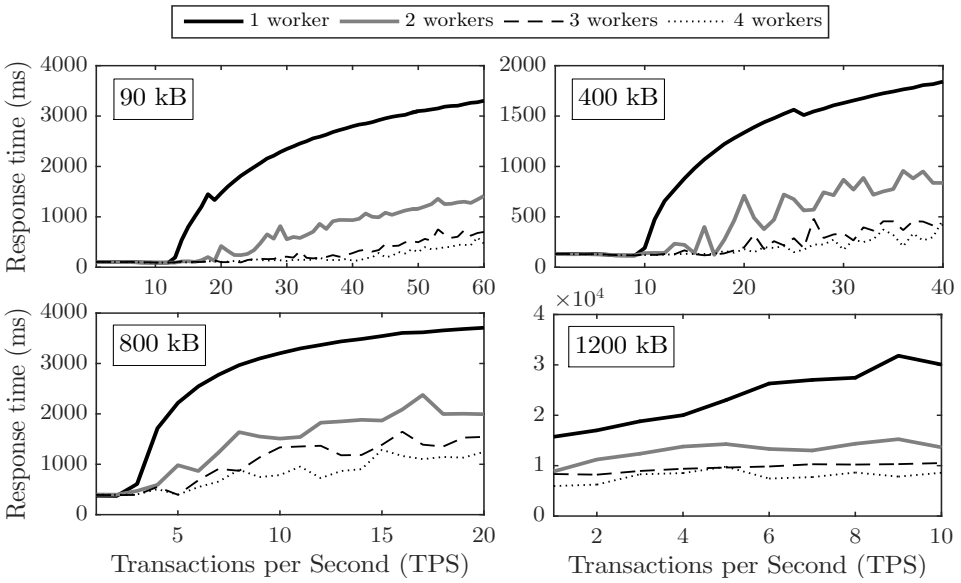


Fig. 6.9 Image resizer function performance

A function that needs more than 1 second to process an object, it can only handle in parallel as many objects as functions workers are running. These examples show that *function collocation at the storage nodes would be, in any case, insufficient to absorb a burst of transactions for more than one second.* Also, they demonstrate that it is almost impossible to predict resource consumption ahead of time, because resources depend on the object size. Consequently, *resource management for collocated functions should be dynamic*, which is hard to achieve in practice, instead of a simple resource management policy such as one single worker per CPU core.

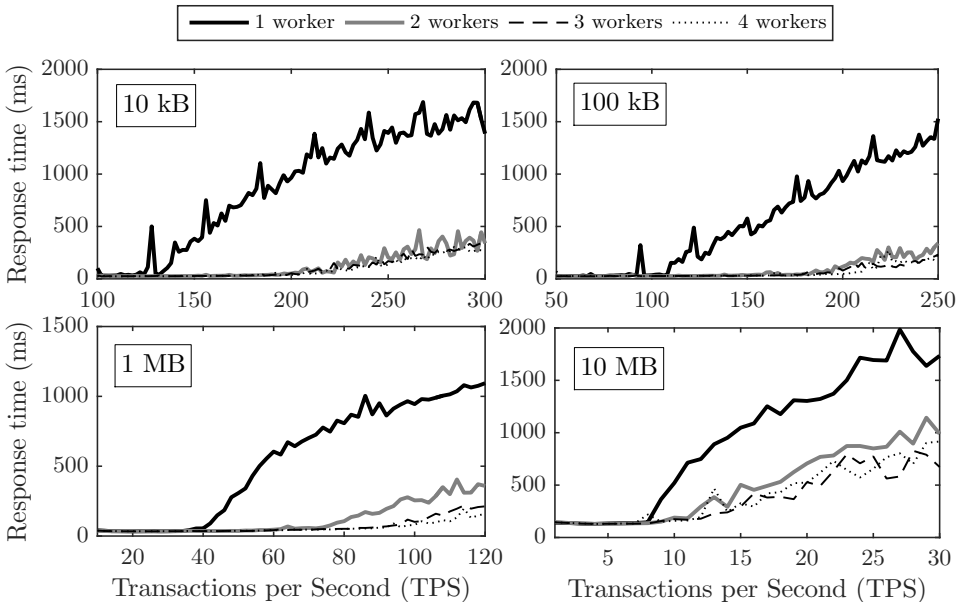


Fig. 6.10 Signature verification function performance

Performance of non-collocated functions. As resource contention is not an issue with non-collocated functions, what is key here is to verify that Zion's storage-disaggregated functions are indeed scalable. In order to test how the Zion's compute layer provides better scalability than the storage nodes, we extended the prior evaluation to non-collocated functions. Our compute nodes have 24 cores each one. This means that is possible to start up to 24 function workers in each node. However, to make the comparative fair with the storage nodes, we utilized between 1 to 8 function workers. Also, as in the previous experiments, we used 1 proxy and 1 storage node in addition to 1 compute node, which is enough to assess the scalability of Zion. The results are shown in Fig. 6.13- 6.12. In this case, we recorded the number of maximum transactions per second (Max. TPS) that each worker was able to handle with and without collocation.

First, we can see how in almost all cases, with 1 to 4 function workers, non-collocated functions can handle more transactions than the storage nodes. This is due to the fact that the compute nodes are more powerful than the storage nodes in terms of CPU. Therefore, the capacity of ingestion is higher. Second, the maximum TPS that a storage node can handle is always limited above by the number of available cores. That is, spawning more workers has no benefit

because they will contend for the available resources. This is the reason why the curve for collocated functions (gray line) flattens out beyond 4 workers.

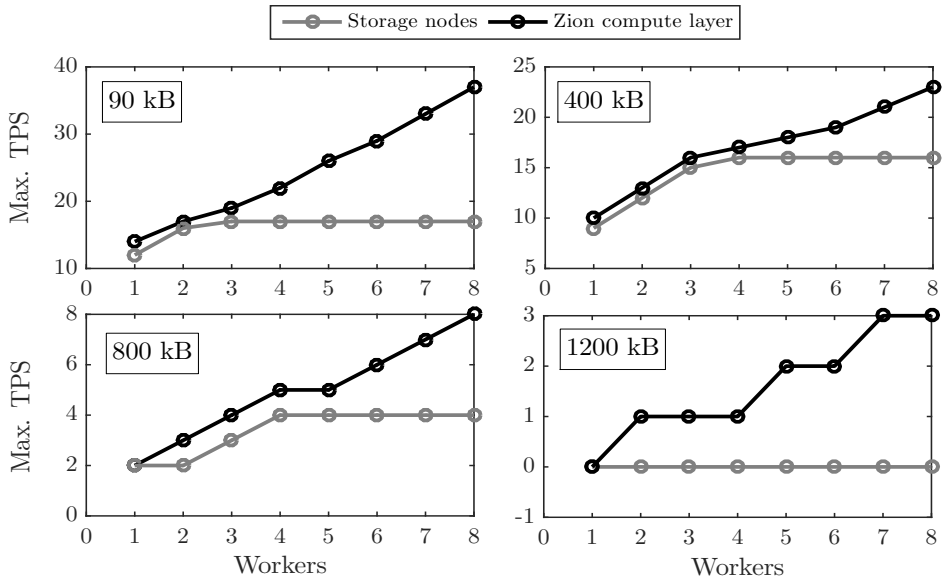


Fig. 6.11 Image resizer function scalability

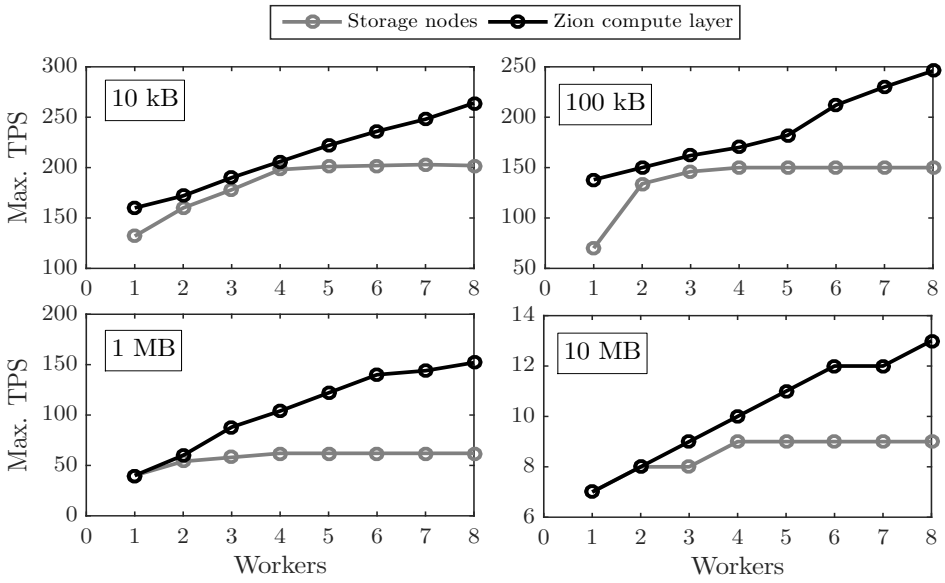


Fig. 6.12 Signature verification function scalability

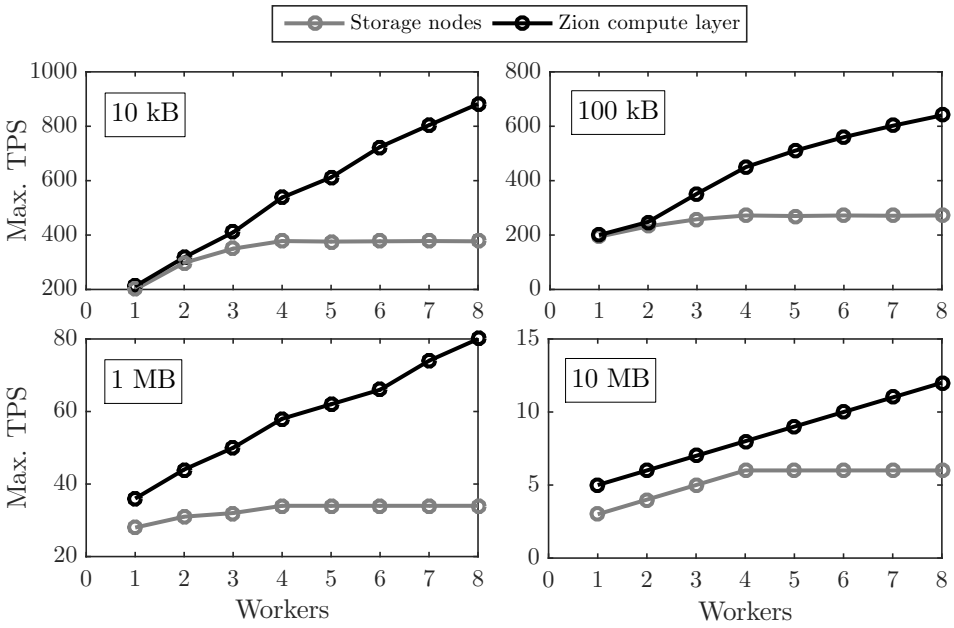


Fig. 6.13 Compression function scalability

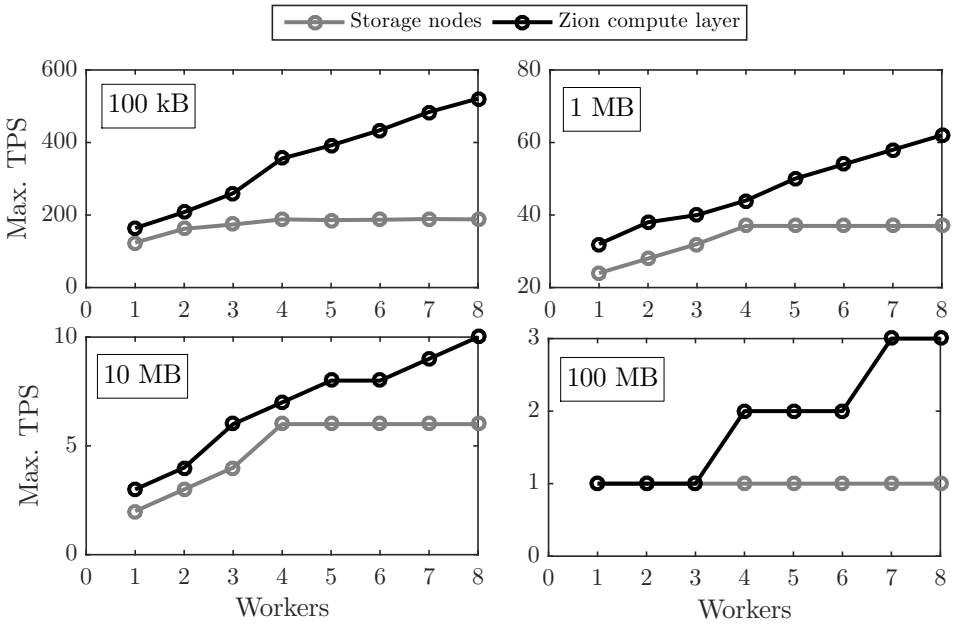


Fig. 6.14 Content-level access control function scalability

However, non-collocated functions (black line) can continue to process transactions by spawning more workers to respond to the demand. By a quick inspection of all the figures, the scalability of non-collocated functions is practically linear with the number of workers. This suggests that *disaggregating storage and compute is a practical solution to scale out computation in cloud object stores.*

Interactive data queries. Table 6.2 compares the execution times for the same query for Hadoop and Zion (Listing 6.2). The entries of this table were obtained by running 30X each configuration, and then, averaging the results.

Listing 6.2 Evaluated query in Hadoop and Swift + Zion clusters.

```
select user_id, count(*) total
where (req_t='GetContentResponse' or req_t='PutContentResponse')
      and msg='Request done'
group by user_id
order by total DESC
limit 10
```

Table 6.2 Interactive data queries execution times

Configuration	File size	Chunk size	Time
Pig query - Swift	100 MB	10 MB	81.6s
Pig query - HDFS	100 MB	10 MB	71.4s
Zion - 12 Workers	100 MB	10 MB	0.510s
Zion - 24 Workers	100 MB	10 MB	0.348s
Pig query - Swift	1 GB	32 MB	156s
Pig query - HDFS	1 GB	32 MB	75.4s
Zion - 12 Workers	1 GB	32 MB	4.183s
Zion - 24 Workers	1 GB	32 MB	2.256s
Pig query - Swift	10 GB	200 MB	985s
Pig query - HDFS	10 GB	200 MB	94.6s
Zion - 12 Workers	10 GB	200 MB	26.253s
Zion - 24 Workers	10 GB	200 MB	13.392s

Non-surprisingly, the Hadoop (Pig) configuration that ingests the data from the remote Swift service is the one that presents the highest mean execution time. This occurred due to network contention. Both clusters were geographically far apart, so communication went through some FAST Ethernet links within our institution's LAN. More concretely, it took ≈ 917 seconds to transfer the entire 10GB dataset, and ≈ 86 seconds to transfer the 1GB dataset. On the other hand, the configuration that read the data from HDFS had better times, specially for the large datasets. However, it has the disadvantage that it requires pre-loading all the data in HDFS, which may be not practical in many cases.

For Zion functions, the execution times were comparatively small despite using at most 24 workers. This suggests that *better times can be easily attained with more workers*. This is clearly seen when comparing Zion with 12 and 24 workers, respectively. For the 10GB dataset and 12 workers, it took 2X much more time than with 24 workers. Also, this experiment confirms the scalability of Zion without incurring in resource contention in the storage cluster. If we had executed this interactive query with 24 workers in the storage nodes, the storage cluster CPU would had reached 100% in all storage nodes for 13.4 seconds, leading to resource contention.

6.6 Summary

This chapter presents Zion, an innovative data-driven serverless computing framework for cloud object storage. Unlike commercial event-driven serverless models, our data-driven functions intercept and operate over objects as they are read/write from/to the object store. Since Zion follows serverless computing abstractions, it overcomes the scalability and resource contention problems of active storage, and without the need to manage any server or runtime environment. Moreover, by injecting computations in the data pipeline, Zion is ideal for use cases that require synchronous interactions with external users, which is the case of the previous systems designed in this thesis (Crystal and Vertigo). More concretely, examples of these use cases include dynamic content generation, interactive queries, content verification, and access control. In many of them, the data locality of our inline computations contributes to optimize latency and to reduce data transfers outside the data center.

Chapter 7

Conclusions & Future Work

Programmability unlocks the full computing power of a system, and significantly increases the innovation on it. Thus, in this thesis we demonstrated how programmability enables unprecedented flexibility and dynamism in cloud object stores. For example, we have shown how a novel Software-defined Storage (SDS) architecture for cloud object storage considerably improves the management of a storage system. In addition, we have also shown how our novel programmatic microcontroller abstraction permits tenants to adapt objects behavior to their specific requirements.

7.1 Overview of Contributions

In our effort to overcome the main research challenge of *enhancing the programmability of cloud object storage*, we posed three major research questions.

Question 1: *Is there a lack of flexibility for handling multi-tenant workloads?*

In Chapter 4 we designed the first SDS architecture for multi-tenant cloud object stores. It is the first one that separates the control plane from the data plane, by abstracting the control layer from the software and hardware of the storage system. With our novel design, we improve the flexibility, simplicity, programmability, manageability and extensibility of cloud object stores. Moreover, with our system it is possible to differentiate multi-tenant workloads, what is not common in most of today's cloud object stores, where tenants are treated indistinctly. To do so, we created a new centralized control layer that allows to

dynamically manage the data plane. This is carried out by means of high-level descriptive policies, based on the If-This-Then-That (IFTTT) paradigm. Thus, establishing new policies is as simple as telling the controller that, for example, *when the number of request per second be greater than five, apply the compression filter to the incoming workload*. Then, once condition(s) are met (If-This), it eventually enforces the filter at the data plane (Then-That). One of the key benefits of our control layer is that it is extensible at run-time. This fact enables adding any new functionality to the cloud object store at any time, with zero storage service down-time.

Our architecture is mainly designed to provide an efficient use of multi-tenant cloud stores. Moreover, this challenge has led us to create new SDS abstractions, such as what we called *controllers*. Controllers are programmatic micro-services that allow more advanced storage management, beyond the descriptive policies. For example, with a controller it is possible to implement novel algorithms for bandwidth control, which enforce the appropriate bandwidth for each tenant at the data plane. We finally created a prototype called Crystal on top of OpenStack Swift to demonstrate the feasibility of our design through multiple storage management applications. For example, we tested the enforcement of filters (compression, encryption), and bandwidth differentiation capabilities among tenants. The performance evaluation and use cases validation show how our novel architecture and abstractions are suitable and relevant enough to be integrated into any shared multi-tenant object store, such as Swift or Ceph.

The next challenge addressed in this thesis was related to how tenants use cloud object storage systems. It can be summarized as follows:

Question 2: *Are tenants able to manage their data in cloud object stores?*

In Chapter 5, we created a novel policy abstraction called microcontroller, which allows tenants to individually manage their objects programmatically. Microcontrollers are designed to handle the particularities of the objects, since different object may require different treatment, and thus, different management. Therefore, our microcontrollers enable the construction of new applications directly in the cloud object stores. To enable this fine-grained object management, microcontrollers act as object wrappers, intercepting the desired lifecycle requests to the objects, and processing them accordingly to the provided behavior. Moreover, microcontrollers can run synchronously or asynchronously upon receipt of the main storage request. They are executed in a sandboxed environment, and in

UNIVERSITAT ROVIRA I VIRGILI
ENHANCING THE PROGRAMMABILITY OF CLOUD OBJECT STORAGE
Josep Sampé Domenech

the storage nodes where the data is stored. This guarantees the security of the storage system, at the same time that they benefit from the data locality and inline processing.

We finally created a prototype on top of OpenStack Swift to evaluate our object-based microcontrollers, and we demonstrated their low overheads in the lifecycle interception layer. Moreover, through different use cases, we validated the programmability and flexibility of microcontrollers for managing objects in the data plane. In concrete, we evaluated applications like object prefetching, active storage orchestration, content-level access control, and automated deletion. The experiments and evaluations show how microcontrollers can become a flexible, dynamic and programmatic object management abstraction for enhancing cloud object stores.

The last challenge addressed in this thesis is related to the resource contention and elasticity problems of both previous contributions, as follows:

Question 3: *Are there enough compute resources in the storage layer for enhancing the programmability?*

In Chapter 6, we designed the first data-driven serverless computing platform for cloud object stores. This novel platform can be integrated within the storage cluster, but in an independent processing layer. This provides elasticity and prevents resource contention problems derived from running computations right away in the storage nodes. Our approach is different than existing serverless computing frameworks that are usually event-based. In our case, we propose the novel concept of *data-driven serverless computing for cloud object stores*, where functions are placed in the storage path, and executed synchronously, intercepting the lifecycle of the objects. By placing serverless functions in the storage path, our model is ideal for those use cases that require of synchronous interactions. Functions integrate the flexibility and programmability of microcontrollers, in addition to a powerful data-flow processing mechanism to process data in real-time, as it comes in/out from the storage cluster. This allows to manage and process data objects with a single piece of code.

Moreover, we created a prototype of our platform on top of OpenStack Swift to validate the feasibility of the inline functions. We proposed some applications which require of synchronous request interception, such as dynamic content generation, interactive queries, content verification, and access control. Finally, we evaluated the platform, getting low overheads when requests are routed to

the computing layer. Also, we tested the previous applications, demonstrating how our data-driven serverless functions placed in the storage path can mitigate resource contention and elasticity problems, while they benefit from the data locality of the storage system.

7.2 Future Research Directions

Enhancing the programmability in cloud object stores is a promising approach thanks to the flexibility and dynamism it provides. Thus, each contribution of this thesis separately opens new research lines. Next we are going to describe some topics and ideas that deserve future work.

1. **Experimental research framework for storage algorithms.** Typically, object stores are non-programmable systems, which difficulties investigation on them. Thus, the *flexible multi-tenant management control layer* we have designed provides, beyond the dynamic management of the storage, a new programmable layer that could allow researchers to easily investigate, implement and test new algorithms for example, for data placement or read-optimized paths. Furthermore, in this sense, our data-driven serverless model allows to test them in an elastic way within the storage cluster, without interfering the storage service.

2. **Security and privacy.** The fine-grained object interception of our microcontrollers enable future research works on object security and data privacy in cloud object stores. Traditionally, in many systems the security layer is located in front of them due to their complexity, and normally they manage the security at higher granularities. The advanced object management that microcontrollers provide enables a future research direction on how to offload this centralized layer and put the appropriate security policies directly to the objects. With our novel model, it is possible to enable *smart secure objects* in object stores.

3. **Rich storage system.** Our data-driven serverless computing layer located in the storage path, close to the data, enable future research works in different directions. First, for *unified storage* (multiprotocol storage). Companies want to move to cloud storage solutions due to their characteristics and low costs. However, their legacy applications require using local file systems to store the information. In this sense, each time more companies use file systems with a cloud object store as data back-end. Thus, it would be interesting to investigate

how to manage and process this data coming for diverse sources with our data-driven model. Second, for *rich-specialized databases*. Flexibility, simplicity and scalability of cloud object stores can enable in a future the design of databases on top of them. Thus, our rich data-driven computing platform can be an important actor to process this data elastically within the storage cluster. Finally, for *web services*. Cloud object stores are widely used for storing static web content. One future research direction would focus on investigating how our elastic serverless computing layer may produce and generate dynamic web content. This could potentially create an alternative and scalable stack for web applications.

UNIVERSITAT ROVIRA I VIRGILI
ENHANCING THE PROGRAMMABILITY OF CLOUD OBJECT STORAGE
Josep Sampé Domenech

Bibliography

[1] Data age 2025, Tech. rep., IDC/Seagate (2017).

[2] The growing role of object storage in solving unstructured data challenges, Tech. rep., 451 Research (2017).

[3] DELL, Magic quadrant for distributed file systems and object storage, https://www.gartner.com/doc/3816966/magic-quadrant-distributed-file-systems (2017).

[4] IDG, 2016 data & analytics research, https://www.idg.com/tools-for-marketers/tech-2016-data-analytics-research/ (2016).

[5] M. Factor, K. Meth, D. Naor, O. Rodeh, J. Satran, Object storage: The future building block for storage systems, in: Local to Global Data Interoperability-Challenges and Technologies, 2005, IEEE, 2005, pp. 119–123.

[6] J. Lindsay, Programmability, http://progrium.com/wiki/Programmability/ (2016).

[7] IBM, Cloud object storage, https://www.ibm.com/cloud/object-storage.

[8] Amazon, S3, https://aws.amazon.com/documentation/s3/.

[9] Google, Cloud storage, https://cloud.google.com/storage/.

[10] Microsoft, Azure blob storage, https://azure.microsoft.com/en-us/services/storage/.

[11] EMC, Ecs overview and architecture, Tech. rep., H-14071.11 (2018).

[12] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, C. Maltzahn, Ceph: A scalable, high-performance distributed file system, in: Proceedings of the 7th Symposium on Operating Systems Design and Implementation, OSDI '06, USENIX Association, Berkeley, CA, USA, 2006, pp. 307–320.
URL http://dl.acm.org/citation.cfm?id=1298455.1298485

[13] OpenStack, Swift, https://docs.openstack.org/swift/.

[14] Is object storage right for your organization?, Tech. rep., Taneja Group (2016).

[15] S. Whitehouse, The gfs2 filesystem, in: Proceedings of the Linux Symposium, Citeseer, 2007, pp. 253–259.

[16] F. B. Schmuck, R. L. Haskin, Gpfs: A shared-disk file system for large computing clusters., in: FAST, Vol. 2, 2002.

[17] IBM, Cloud block storage, https://www.ibm.com/cloud/block-storage.

[18] Amazon, Elastic block store (ebs), https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AmazonEBS.html.

[19] Microsoft, Premium storage, https://docs.microsoft.com/en-us/azure/virtual-machines/windows/premium-storage.

[20] Google, Persistent disk, https://cloud.google.com/functions/docs/.

[21] A. S. Tanenbaum, M. Van Steen, Distributed systems: principles and paradigms, Prentice-Hall, 2007.

[22] Glusterfs, https://docs.gluster.org/en/latest/.

[23] Amazon, Elastic file system, https://aws.amazon.com/documentation/efs/.

[24] IBM, Cloud file storage, https://www.ibm.com/cloud/file-storage/.

[25] Microsoft, Azure files, https://docs.microsoft.com/en-us/azure/storage/files/.

[26] B. Welch, Object storage technology, https://www.snia.org/sites/default/education/tutorials/2013/spring/file/BrentWelch_Object_Storage_Technology.pdf.

[27] M. Mesnier, G. R. Ganger, E. Riedel, Object-based storage, IEEE Communications Magazine 41 (8) (2003) 84–90.

[28] E. Riedel, S. Iren, Object storage and applications, Tech. rep., SNIA (2007).

[29] D. Nagle, M. Factor, S. Iren, D. Naor, E. Riedel, O. Rodeh, J. Satran, The ansi t10 object-based storage standard and current implementations, IBM Journal of Research and Development 52 (4.5) (2008) 401–411.

[30] P. Deutsch, Deflate compressed data format specification version 1.3, Tech. rep. (1996).

[31] A. B. Primer, Erasure codes for storage systems.

[32] Lustre: A scalable, high-performance file system, Tech. rep., Cluster File Systems, Inc (2003).

[33] B. Welch, M. Unangst, Z. Abbasi, G. A. Gibson, B. Mueller, J. Small, J. Zelenka, B. Zhou, Scalable performance of the panasas parallel file system., in: FAST, Vol. 8, 2008, pp. 1–17.

[34] OpenStack, Cloud computing platform projects, https://www.openstack.org/software/project-navigator.

[35] OpenStack, Swift architectural overview, https://docs.openstack.org/swift/latest/overview_architecture.html.

[36] OpenStack, Swift api, https://developer.openstack.org/api-ref/object-store/.

[37] OpenStack, Swift middleware catalog, https://docs.openstack.org/swift/latest/middleware.html.

[38] J. Piernas, J. Nieplocha, E. J. Felix, Evaluation of active storage strategies for the lustre parallel file system, in: SC, 2007, p. 28.

[39] OpenStack, Storlets, https://docs.openstack.org/storlets/.

[40] S. Rabinovici-Cohen, E. Henis, J. Marberg, K. Nagin, Storlet engine: performing computations in cloud storage, Tech. rep., IBM H-0320 (2014).

[41] L. Coyne, J. Dain, P. Gilmer, P. Guaitani, I. Hancock, A. Maille, T. Pearson, B. Sherman, C. Vollmar, et al., IBM Software-Defined Storage Guide, IBM Redbooks, 2017.

[42] EMC, Emc atmos cloud storage architecture, Tech. rep., H-9505.1 (2014).

[43] Google, Cloud storage - object lifecycle management, https://cloud.google.com/storage/docs/lifecycle.

[44] P. Biswas, F. Patwa, R. Sandhu, Content level access control for openstack swift storage, in: Proceedings of the 5th ACM Conference on Data and Application Security and Privacy, ACM, 2015, pp. 123–126.

[45] D. Mishin, D. Medvedev, A. Szalay, R. Plante, M. Graham, Data sharing and publication using the scidrive service, Proceedings of Astronomical Data Analysis Software and Systems XXIII 465.

[46] N. Kaaniche, M. Laurent, M. El Barbori, Cloudasec: A novel public-key based framework to handle data sharing security in clouds, in: Security and Cryptography (SECRYPT), 2014 11th International Conference on, IEEE, 2014, pp. 1–14.

[47] E. Bacis, S. De Capitani di Vimercati, S. Foresti, D. Guttadoro, S. Paraboschi, M. Rosa, P. Samarati, A. Saullo, Managing data sharing in openstack swift with over-encryption, in: Proceedings of the 2016 ACM on Workshop on Information Sharing and Collaborative Security, ACM, 2016, pp. 39–48.

[48] E. Bacis, S. D. C. di Vimercati, S. Foresti, S. Paraboschi, M. Rosa, P. Samarati, Access control management for secure cloud storage, in: International Conference on Security and Privacy in Communication Systems, Springer, 2016, pp. 353–372.

[49] Amazon, Amazon elastic compute cloud (ec2), https://aws.amazon.com/documentation/ec2/.

[50] IBM, Cloud virtual servers, https://www.ibm.com/cloud/virtual-servers.

[51] Amazon, Lambda, https://aws.amazon.com/lambda/.

[52] Apache, Openwhisk, https://openwhisk.apache.org/.

[53] IBM, Cloud functions, https://www.ibm.com/cloud/functions.

[54] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, K. Winstein, Encoding, fast and slow: Low-latency video processing using thousands of tiny threads, in: NSDI, 2017.

[55] E. Jonas, S. Venkataraman, I. Stoica, B. Recht, Occupy the cloud: Distributed computing for the 99%, arXiv preprint arXiv:1702.04024.

[56] J. S. Gil Vernik, Process large data sets at massive scale with pywren over ibm cloud functions, https://www.ibm.com/blogs/bluemix/2018/04/process-large-data-sets-massive-scale-pywren-ibm-cloud-functions/ (2018).

[57] G. V. I. Josep Sampé, Pywren for ibm cloud, https://github.com/pywren/pywren-ibm-cloud (2018).

[58] Y. Kim, J. Lin, Serverless data analytics with flint, arXiv preprint arXiv:1803.06354.

[59] Qubole, Spark on lambda, https://github.com/qubole/spark-on-lambda (2018).

[60] B. Congdon, Corral a mapreduce framework, https://github.com/bcongdon/corral (2018).

[61] Amazon, Lambda edge, https://aws.amazon.com/es/lambda/edge/.

[62] S. Nastic, T. Rausch, O. Scekic, S. Dustdar, M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Ristov, R. Prodan, A serverless real-time data analytics platform for edge computing, IEEE Internet Computing 21 (4) (2017) 64–71.

[63] S. Yi, Z. Hao, Q. Zhang, Q. Zhang, W. Shi, Q. Li, Lavea: Latency-aware video analytics on edge computing platform, in: Proceedings of the Second ACM/IEEE Symposium on Edge Computing, ACM, 2017, p. 15.

[64] J. Sampé, Towards cooperative analytics in disaggregated big data clusters, in: 2nd URV Doctoral Workshop in Computer Science and Mathematics, Publicacions URV, 2015, pp. 9–13.

[65] Y. Moatti, E. Rom, R. Gracia-Tinedo, D. Naor, D. Chen, J. Sampe, M. Sanchez-Artigas, P. Garcıa-Lopez, F. Gluszak, E. Deschdt, et al., Too big to eat: Boosting analytics data ingestion from object stores with scoop, in: Data Engineering (ICDE), 2017 IEEE 33rd International Conference on, IEEE, 2017, pp. 309–320.

[66] IBM, Cloud sql query, https://www.ibm.com/cloud/sql-query.

[67] Amazon, Athena, https://aws.amazon.com/athena/.

[68] Amazon, Redshift spectrum, https://aws.amazon.com/redshift/spectrum/.

[69] Facebook, Presto: Distributed sql engine for big data, https://prestodb.io/.

[70] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner, Openflow: enabling innovation in campus networks, ACM SIGCOMM Computer Communication Review 38 (2) (2008) 69–74.

[71] A. Tootoonchian, Y. Ganjali, Hyperflow: A distributed control plane for openflow, in: USENIX INM/WREN'10, 2010, pp. 3–3.

[72] Software-defined storage: A pervasive approach to IT transformation driven by the 3rd platform, https://www.emc.com/collateral/analyst-reports/idc-sds-3rd-platform.pdf.

[73] Dell emc - software-defined storage solutions, https://www.emc.com/en-us/storage/software-defined-storage/solutions.htm.

[74] Ibm software-defined storage, http://www-03.ibm.com/systems/storage/software-defined-storage.

[75] VMware, Vmware vsan 6.5 technical overview, Tech. rep. (2016).

[76] Software-defined storage - vmware products, http://www.vmware.com/products/software-defined-storage.html.

[77] Nutanix enterprise cloud platform, https://www.nutanix.com/products/.

[78] EMC, Emc vipr controller. automate and simply storage management, Tech. rep., EMC H-1430.8 (2014).

[79] EMC, Emc vipr suite. multivendor storage automation and self-service, Tech. rep., EMC H-1430.8 (2014).

[80] Microsoft, Coprhd, https://coprhd.github.io/.

[81] 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 58 (2/3) (2014) 5–1.

[82] Openio, http://www.openio.io/.

[83] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, T. Zhu, Ioflow: a software-defined storage architecture, in: ACM SOSP'13, 2013, pp. 182–196.

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

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

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

[87] S. Peter, J. Li, I. Zhang, D. R. Ports, D. Woos, A. Krishnamurthy, T. Anderson, T. Roscoe, Arrakis: The operating system is the control plane, ACM Transactions on Computer Systems (TOCS) 33 (4) (2016) 11.

[88] R. Raghavendra, P. Dewan, M. Srivatsa, Unifying hdfs and gpfs: Enabling analytics on software-defined storage, in: ACM/IFIP/USENIX Middleware'16, 2016, p. 3.

[89] J. Wires, A. Warfield, Mirador: An active control plane for datacenter storage., in: USENIX FAST'17, 2017, pp. 213–228.

[90] G. A. Alvarez, E. Borowsky, S. Go, T. H. Romer, R. Becker-Szendy, R. Golding, A. Merchant, M. Spasojevic, A. Veitch, J. Wilkes, Minerva: An automated resource provisioning tool for large-scale storage systems, ACM Transactions on Computer Systems (TOCS) 19 (4) (2001) 483–518.

[91] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, A. C. Veitch, Hippodrome: Running circles around storage administration., in: USENIX FAST'02, Vol. 2, 2002, pp. 175–188.

[92] H. V. Madhyastha, J. McCullough, G. Porter, R. Kapoor, S. Savage, A. C. Snoeren, A. Vahdat, scc: cluster storage provisioning informed by application characteristics and slas., in: USENIX FAST'12, 2012, p. 23.

[93] B. Trushkowsky, P. Bodík, A. Fox, M. J. Franklin, M. I. Jordan, D. A. Patterson, The scads director: Scaling a distributed storage system under stringent performance requirements., in: USENIX FAST'11, 2011, pp. 163–176.

[94] H. Vashishtha, E. Stroulia, Enhancing query support in hbase via an extended coprocessors framework, in: ServiceWave, 2011, pp. 75–87.

[95] Apache, Hbase coprocessors, https://blogs.apache.org/hbase/entry/coprocessor_introduction (2012).

[96] Google, Bigtable coprocessors, https://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf (2009).

[97] E. Riedel, G. A. Gibson, C. Faloutsos, Active storage for large-scale data mining and multimedia, in: VLDB, 1998, pp. 62–73.

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

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

[100] M. Uysal, A. Acharya, J. Saltz, Evaluation of active disks for decision support databases, in: IEEE HPCA'00, 2000, pp. 337–348.

[101] L. Zeng, S. Chen, Q. Wei, D. Feng, Sedas: A self-destructing data system based on active storage framework, in: APMRC, 2012 Digest, IEEE, 2012, pp. 1–8.

[102] Z. Istvan, D. Sidler, G. Alonso, Active pages 20 years later: Active storage for the cloud, IEEE Internet Computing 22 (4) (2018) 6–14.

[103] R. Wickremesinghe, J. S. Chase, J. S. Vitter, Distributed computing with load-managed active storage, in: HPDC, 2002, pp. 13–23.

[104] X. Ma, A. N. Reddy, Mvss: An active storage architecture, IEEE Transactions on Parallel and Distributed Systems 14 (10) (2003) 993–1005.

[105] Q. Xu, K. M. M. Aung, Y. Zhu, K. L. Yong, Building a large-scale object-based active storage platform for data analytics in the internet of things, The Journal of Supercomputing 72 (7) (2016) 2796–2814.

[106] T. M. John, A. T. Ramani, J. A. Chandy, Active storage using object-based devices, in: IEEE Cluster'08, 2008, pp. 472–478.

[107] S. W. Son, S. Lang, P. Carns, R. Ross, R. Thakur, B. Ozisikyilmaz, P. Kumar, W.-K. Liao, A. Choudhary, Enabling active storage on parallel i/o software stacks, in: MSST, 2010, pp. 1–12.

[108] Message passing interface (mpi), https://computing.llnl.gov/tutorials/mpi/.

[109] M. T. Runde, W. G. Stevens, P. A. Wortman, J. A. Chandy, An active storage framework for object storage devices, in: MSST, 2012, pp. 1–12.

[110] S. Narayan, J. A. Chandy, Attest: Attributes-based extendable storage, J. Syst. Softw. 83 (4) (2010) 548–556.

[111] O. Momin, C. Karakoyunlu, M. T. Runde, J. A. Chandy, Creating a programmable object storage stack, in: ACM PFSW'14, 2014, pp. 3–10.

[112] S. Narayan, J. A. Chandy, Extendable storage framework for reliable clustered storage systems, in: IEEE IPDPSW'10, 2010, pp. 1–4.

[113] Y. Xie, D. Feng, Y. Li, D. D. Long, Oasis: an active storage framework for object storage platform, Future Generation Computer Systems 56 (2016) 746–758.

[114] C. Chen, Y. Chen, P. C. Roth, Dosas: Mitigating the resource contention in active storage systems, in: CLUSTER, 2012, pp. 164–172.

[115] R. B. Ross, R. Thakur, et al., Pvfs: A parallel file system for linux clusters, in: 4th annual Linux showcase and conference, 2000, pp. 391–430.

[116] L. Qin, D. Feng, Active storage framework for object-based storage device, in: AINA'06, 2006, pp. 97–101.

[117] P. Rad, V. Lindberg, J. Prevost, W. Zhang, M. Jamshidi, Zerovm: secure distributed processing for big data analytics, in: WAC, 2014, pp. 1–6.

[118] R. Gracia-Tinedo, J. Sampé, E. Zamora, M. Sánchez-Artigas, P. García-López, Y. Moatti, E. Rom, Crystal: software-defined storage for multi-tenant object stores, in: Proceedings of the 15th Usenix Conference on File and Storage Technologies, USENIX Association, 2017, pp. 243–256.

[119] A. Anwar, Y. Cheng, A. Gupta, A. R. Butt, Mos: Workload-aware elasticity for cloud object stores, in: ACM HPDC'16, 2016, pp. 177–188.

[120] A. Anwar, Y. Cheng, A. Gupta, A. R. Butt, Taming the cloud object storage with mos, in: Proceedings of the 10th Parallel Data Storage Workshop, 2015, pp. 7–12.

[121] G. V. Michael Factor, 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).

[122] M. Murugan, K. Kant, A. Raghavan, D. H. Du, Flexstore: A software defined, energy adaptive distributed storage framework, in: IEEE MASCOTS'14, 2014, pp. 81–90.

[123] R. Gracia-Tinedo, P. García-López, M. Sánchez-Artigas, J. Sampé, Y. Moatti, E. Rom, D. Naor, R. Nou, T. Cortés, W. Oppermann, et al., Iostack: Software-defined object storage, IEEE Internet Computing 20 (3) (2016) 10–18.

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

[125] Ifttt, https://ifttt.com.

[126] G. A. Agha, Actors: A model of concurrent computation in distributed systems, Tech. rep., The MIT Press (1985).

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

[128] R. Stutsman, C. Lee, J. Ousterhout, Experience with rules-based programming for distributed, concurrent, fault-tolerant code, in: USENIX ATC'15, 2015, pp. 17–30.

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

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

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

[132] D. Merkel, Docker: lightweight linux containers for consistent development and deployment, Linux Journal 2014 (239) (2014) 2.

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

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

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

[136] Crystal, https://github.com/Crystal-SDS.

[137] R. Labs, Redis, https://redis.io.

[138] D. Barcelona, P. Garcia, Pyactor, https://github.com/pedrotgn/pyactor.

[139] SwiftStack, Ssbench, https://github.com/swiftstack/ssbench.

[140] Iostack datasets, http://iostack.eu/datasets-menu.

[141] R. Gracia-Tinedo, D. Harnik, D. Naor, D. Sotnikov, S. Toledo, A. Zuck, SDGen: mimicking datasets for content generation in storage benchmarks, in: USENIX FAST'15, 2015, pp. 317–330.

[142] J. Sampé, P. García-López, M. Sánchez-Artigas, Vertigo: Programmable micro-controllers for software-defined object storage, in: Cloud Computing (CLOUD), 2016 IEEE 9th International Conference on, IEEE, 2016, pp. 180–187.

[143] J. Sampé, Vertigo framework, https://github.com/JosepSampe/vertigo.

[144] B. Fitzpatrick, Distributed caching with memcached, Linux journal 2004 (124) (2004) 5.

[145] Amazon, Redshift, https://docs.aws.amazon.com/redshift/.

[146] J. Sampé, Swift prefetching middleware, https://github.com/JosepSampe/swift-prefetching-middleware.

[147] J. Sampé, Swift linking middleware, https://github.com/JosepSampe/swift-linking-middleware.

[148] 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, B. Qiu, Bigdatabench: A big data benchmark suite from internet services, in: IEEE HPCA'14, 2014, pp. 488–499.

[149] U. of California, Adult dataset, http://archive.ics.uci.edu/ml/datasets/Adult.

[150] Ubuntu, Cloud image repository, https://cloud-images.ubuntu.com/.

[151] Google, html5 slides, https://code.google.com/archive/p/html5slides/.

[152] Apache, Jmeter, http://jmeter.apache.org/.

[153] Ubuntu, Docker image, https://hub.docker.com/r/_/ubuntu/.

[154] J. Sampé, M. Sánchez-Artigas, P. García-López, G. París, Data-driven serverless functions for object storage, in: Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference, ACM, 2017, pp. 121–133.

[155] Google, Cloud functions, https://cloud.google.com/functions/docs/.

[156] Microsoft, Azure functions, https://azure.microsoft.com/en-us/services/functions/.

[157] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, Serverless computation with openlambda, in: HotCloud, 2016.

[158] J. Sampé, Zion framework, https://github.com/JosepSampe/zion.

[159] OpenStack, Swift large objects, https://docs.openstack.org/swift/latest/overview_large_objects.html.

[160] OpenStack, Nova, https://docs.openstack.org/nova.

[161] I. University of California, Adult data set, http://archive.ics.uci.edu/ml/datasets/Adult (1996).

[162] R. Gracia-Tinedo, Y. Tian, J. Sampé, H. Harkous, J. Lenton, P. García-López, M. Sánchez-Artigas, M. Vukolic, Dissecting ubuntuone: Autopsy of a global-scale personal cloud back-end, in: IMC, 2015, pp. 155–168.

[163] Apache, Hadoop openstack support: Swift object store, https://hadoop.apache.org/docs/stable/hadoop-openstack/index.html.

UNIVERSITAT ROVIRA I VIRGILI
ENHANCING THE PROGRAMMABILITY OF CLOUD OBJECT STORAGE
Josep Sampé Domenech

UNIVERSITAT ROVIRA I VIRGILI
ENHANCING THE PROGRAMMABILITY OF CLOUD OBJECT STORAGE
Josep Sampé Domenech

UNIVERSITAT ROVIRA I VIRGILI
ENHANCING THE PROGRAMMABILITY OF CLOUD OBJECT STORAGE
Josep Sampé Domenech

UNIVERSITAT
ROVIRA i VIRGILI