UNIVERSITAT ROVIRA I VIRGILI

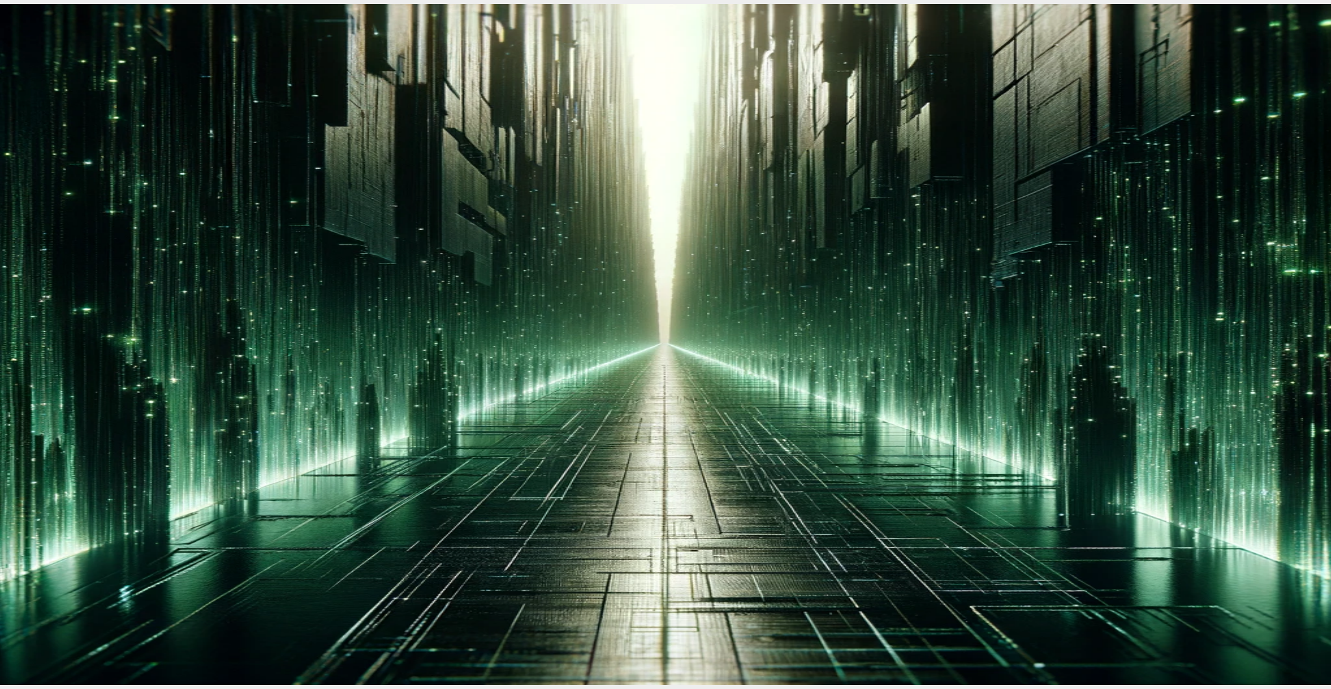
ON IMPROVED PERFORMANCE AND SECURE DATA MANAGEMENT IN CLOUD STORAGE SERVICES

Raul Saiz Laudo

UNIVERSITAT ROVIRA i VIRGILI

On Improved Performance and Secure Data Management in Cloud Storage Services

RAÚL SÁIZ LAUDÓ



DOCTORAL THESIS
2024

Raúl Sáiz Laudó

On Improved Performance and Secure Data Management in Cloud Storage Services

DOCTORAL THESIS

Supervised by

Dr. Marc Sánchez-Artigas

Department of Computer Engineering and Mathematics

UNIVERSITAT ROVIRA i VIRGILI

Tarragona

2024

UNIVERSITAT
ROVIRA i VIRGILI

Departament d'Enginyeria
[DΣIM]
Informàtica i
Matemàtiques

FAIG CONSTAR que aquest treball, titulat “On Improved Performance and Secure Data Management in Cloud Storage Services”, que presenta Raúl Sáiz Laudó 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.

HAGO CONSTAR que el presente trabajo, titulado “On Improved Performance and Secure Data Management in Cloud Storage Services”, que presenta Raúl Sáiz Laudó para la obtención del título de Doctor, ha sido realizado bajo mi dirección en el Departamento de Ingeniería Informática y Matemáticas de esta universidad.

I STATE that the present study, entitled “On Improved Performance and Secure Data Management in Cloud Storage Services”, presented by Raúl Sáiz Laudó for the award of the degree of Doctor, has been carried out under my supervision at the Department Computer Engineering and Mathematics of this university.

Tarragona, 16 de Gener/16 de Enero/January 16, 2024

El director de la tesi doctoral

El director de la tesis doctoral

Doctoral thesis supervisor

Dr. Marc Sánchez-Artigas

Abstract

The rise of cloud storage systems in both personal and enterprise domains has brought a significant shift in data management practices. This transition encompasses not only raw object storage but also personal cloud-based storage services like Dropbox, which enable users to synchronize their files across multiple devices, fostering collaboration and accessibility. Both types of storage services, namely object storage and personal cloud storage, are the primary focus of this thesis.

On one hand, this thesis focuses on personal cloud storage services. The cornerstone of these services is to implement an efficient file synchronization protocol that automatically maps the changes in the local file system of a user to the cloud via a series of network communications in a timely manner. If not sufficient care is taken, an ill-designed file synchronization protocol may generate huge network traffic, causing tremendous financial losses and performance penalties to both service providers and users. To increase the network efficiency of file synchronization protocols, this thesis proposes a novel file synchronization deferment method named "rate-based sync deferment". Synchronization deferment consists of batching edits to a file to deliberately defer the synchronization process for some time. In this way, the client can artificially increase the amount of useful data per synchronization operation, and thus, diminish the network overhead.

On the other hand, this thesis focuses on raw object storage services. At the same time, cloud storage is becoming a more integral part of our digital ecosystem, it is also developing into a large repository of private and sensitive data. However, the tools for protecting data in object storage services is very limited. For instance, Amazon S3, the market leader, only protects data while it is in transit by using Secure Socket Layer/Transport Layer Security (SSL/TLS) or at rest, using either client-side or server-side encryption. But it does not allow, for instance, to use homomorphic encryption in place to provide more expressive functionalities for encrypted data. To overcome this, this thesis explores the benefits of applying the software-defined principles to secure data management in object stores and proposes a new system called EGEON. In EGEON, the control plane holds the intelligence of the system and consist of a logically centralized controller. The data plane, implemented as a serverless programmable framework, enables the injection of programming logic such as encryption primitives to perform custom computations over object `GET` requests.

In both cases, the results of the thesis have been satisfactory, advancing the state of the art in two clear-cut directions. On one hand, the new synchronization deferment mechanism improves synchronization delays between 2X to 12X relative to the state of the art. On the other hand, this thesis demonstrates the software-defined principles are a powerful tool to circumvent the rigidity of object storage APIs and bring data manipulation in place, eliminating unnecessary external data movements to protect data.

Acknowledgements

I would like to express special thank you to my advisor, Dr. Marc Sánchez-Artigas, for his invaluable guidance, unwavering patience, and expert advice. His mentorship was critical in shaping both this research and my growth as a scholar. I also extend my gratitude to the faculty members of the Research Group CloudLab for their support throughout this process.

My deepest appreciation goes to my family, especially my parents, Juan and Empar, my brother Guillermo, my wife Sory, and my two children, Raul and Diego, for their endless love, understanding, and support. Their belief in me has been a constant source of strength, inspiring me to persevere even in the most challenging times.

Finally, to everyone who has been a part of my thesis journey, both directly and indirectly, your support has been invaluable. Thank you for contributing to this significant milestone in my life.

Raúl Sáiz Laudó
Cambrils January 2024

Thesis Publications

- Raul Saiz-Laudo. "Reducing Network Overhead on Personal Cloud Systems". In: *3rd URV Doctoral Workshop in Computer Science and Mathematics*. Ed. by Sergio Gómez and Aïda Valls. Tarragona, Spain: Publicacions URV, Nov. 2016, pp. 27–31. ISBN: 978-84-8424-495-0 [80].

- Raúl Sáiz-Laudó, Marc Sánchez-Artigas, and Pedro García-López. "RSD: Rate-Based Sync Deferment for Personal Cloud Storage Services". In: *IEEE Communications Letters* 21.11 (2017), pp. 2384–2387. DOI: `10.1109/LCOMM.2017.2731848`. JCR Impact Factor: 2.723 (Q2 Telecommunications), SJR Index: 0.589 (Q1 Electrical and Electronics Engineering) [82].

- Raul Saiz-Laudo and Marc Sánchez-Artigas. "Egeon: Software-Defined Data Protection for Object Storage". In: *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 2022, pp. 99–108. DOI: `10.1109/CCGrid54584.2022.00019`. Ranked Core A. Acceptance rate: 24% (75/302 *submissions*) [81].

Table of Contents

List of Figures

Listings

List of Tables

List of Equations

List of Abbreviations

ACLs	Access Control Lists
API	Application Programming Interface
ASD	Adaptative Sync Deferment
AWS	Amazon Web Services
AZs	Availability Zones
BaaS	Backend as a Service
CDC	Content-Defined Chunking
CLAC	Content-Level Access Control
CPU	Central Processing Unit
CRT	Chine Remainder Theorem
CRUSH	Controlled Replication under Scalable Hashing
CSV	Comma-separated values
EWMA	Exponentially Weighted Moving Average
FaaS	Function as a Service
FPGA	Field-Programmable Gate Array
Gb	Gigabits
GbE	Gigabit Ethernet

GB	Gigabyte (10^9 bytes)
GHz	Gigahertz
HPC	High-Performance Computing
HTTP	Hypertext Transfer Protocol
IDS	Identity Service
I/O	Input/Output
IoT	Internet of Things
ITR	Interrupted Transfer Resumption
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
KB	Kilobyte (10^3 bytes)
KiB	Kibibyte (2^{10} bytes)
LaBAC	Label Based Access Control
LAN	Local Area Network
LRU	Least Recently Used
LTS	Long Time Service
Mb	Megabits
Mbps	Megabits per second
MB	Megabyte (10^6 bytes)
MiB	Mebibyte (2^{20} bytes)
ms	Millisecond
NDP	Near Data Processing
NOOP	No Operation
NVMe	Non-Volatile Memory Express
OSDs	Object Storage Daemons
PBs	Petabytes ($10^1$5 bytes)
PEKS	Public Key Encryption with Keyword Search
PII	Personal Identifiable Information
P2P	Peer-assisted Offloading
RSD	Rate-Based Sync Deferment

RAM	Random Access Memory
s	Second
SDN	Software-Defined Networking
SDS	Software-Defined Storage
SR	Slowdown Ratio
SSD	Solid State Disk
SSF	Serverless Storage Functions
S3	Amazon Simple Storage Service
TUE	Traffic Usage Efficiency
TTFB	Time to First Byte
TTLB	Time to Last Byte
UB1	Ubuntu One
UDF	User-Defined Function
VM	Virtual Machine
XML	Extensible Markup Language
4G	4th Broadband Cellular Network

Chapter 1

Motivation

In today's digital landscape, cloud storage has become an integral component, revolutionizing how we store, access, and manage data. With the exponential increase in data generation from various sources, including personal devices, businesses, and IoT systems, the traditional means of storage such as local hard drives (HDDs) and solid state drives (SSDs) have become insufficient to guarantee high availability of data and durability.Recent projections suggest that by 2025, the worldwide datasphere will swell to an immense 163 Zettabytes (ZiB) [77]. This represents a staggering tenfold increase from the volume of data produced in 2016.

Cloud storage services address this challenge by offering scalable, flexible, and efficient data storage solutions that are accessible from anywhere in the world. This technology not only ensures high data availability but also facilitates collaboration and data sharing across different geographical locations. Among the available cloud storage incarnations, object storage has become the storage of choice in the cloud, and of other everyday storage services such as file synchronization systems (e.g., Dropbox). Services like Amazon S3, the market leader, are growing at an amazing clip. The reason is that object storage is cheap, reliable and scalable, but also because it eliminates a historically big challenge in bringing new and compelling products to market. But S3, and other object storage services such as OpenStack Swift, are schemaless and lack the tools to extract value from the massive amounts of data that they house. Their APIs are rather poor, operate at the object level, and offer almost no tools, e.g., to enable data manipulation in-place, requiring physical transformations or external data movements. Simply put, they lack the ability to code directly on the object storage infrastructure, building virtual data pipelines and transformations [84]. Companies are pouring so much data into the cloud but then have to move it to other services to draw insights and perform even basic analytics. That is expensive, time consuming and in many cases complete overkill. The insufficient programmability of object storage, together with the need of supporting data security policies that govern how users and apps can interact with a given content, have become one of the main motivations behind this thesis. We recall that the security support available in object storage systems is strikingly inadequate to support complex, collaborative environments such as of those file synchronization systems. For instance, managing access control at scale based on the content of the objects themselves is impossible in today's cloud object stores [14].

1.1 Problem statement

Public clouds have democratized access to storage resources for a variety of workloads by offering a plethora of services to users. These services range from raw object storage up to more sophisticated personal cloud storage services such as Dropbox, where users can synchronize their files across multiple devices, fostering collaboration and accessibility. Both types of storage services, namely object storage and personal cloud storage, are the primary focus of this thesis.

On one hand lies personal cloud storage. These services provide users with a very convenient way to store and share data from anywhere, on any device, and at any time. However, the linchpin in providing this functionality is to implement an inefficient file synchronization protocol that automatically maps the changes in the local file system of a user to the cloud via a series of communications over the network in a timely manner. We say "timely", because slow synchronization may lead to file conflicts in case that multiple users are editing the same very file concurrently. Typically, only one copy of the file will be saved as the original copy and other updates will generate new copies of the file as "conflict version". This is burdensome for users. But this only half of the story. The other half of the story is that an efficient file synchronization protocol may generate huge network traffic, and cause tremendous financial losses and performance penalties to both service providers and users. An example of this was the case of the Ubuntu One (UB1) service. A key problem for the survival of UB1 was the growing costs of outsourcing data storage [93]. This was indeed related to a poor data management. For instance, the fact that file updates were responsible for 18.5 of upload traffic in UB1, mostly due to the lack of delta updates[1] in the desktop client, gives an idea of the what may happen when the file synchronization protocol is built without proper care [30].

In this context, this thesis explores a new file synchonization deferment method that optimizes the use of network bandwidth between clients and service providers. The root of the problem is the presence of frequent edits to files. Frequent edits may well incur in abundant overhead traffic that far exceeds the amount of useful synchronization data sent by the client over time, which is referred to as the *traffic overuse problem* [47]. As observed in [23], 8.5% of Dropbox users, more than 10% of their synchronization traffic is caused by frequent edits, which suggests that this is a relevant problem in personal cloud storage services. Unfortunately, some cloud storage services such as iCloud Drive as of 2019 still treated frequent edits as full file synchronization operation[2], thus aggravating the traffic overuse. To overcome this problem, some cloud storage services deliberately the synchronization process for a certain period of time to batch the file updates and save network traffic. This observation leads to the first question tackled by this thesis:

[1] Delta compression is also noteworthy, as it involves storing only the changes made to a file, rather than the entire file after each edit, thus conserving storage space and bandwidth.

[2] A full upload of the file to the cloud servers

Question I: Could file synchronization deferment be optimized to help mitigate the traffic overuse problem?

On the other hand, mainstream personal cloud storage services abide by a two-cloud system architecture [47], one cloud for storage, and another for control. The control cloud, typically inhouse, is responsible for the indexing of file blocks, file metadata such as folders, user accounts, and so forth. The storage cloud stores the synchronized files, along with small-sized thumbnail views for certain files, and use object storage services such as Amazon S3 for this aim. Example of these sevices included Dropbox [23] and UB1 [30]. That is, these services use raw object storage as storage substrate for files, and file blocks in case the cloud storage service supports incremental synchronization. The reason is that object storage is cheap and reliable, e.g., the standard service level agreement (SLA) of Amazon S3 offers 99.99% data availability. Although Amazon S3 gives any developer access to a highly scalable, reliable, fast, inexpensive data storage infrastructure, its has a poor interface, operating at the level of entire objects, with a few exceptions such as multipart uploads[3] and small set of services (e.g., caching, queueing, data placement). This makes it difficult to implement collaborative scenarios in personal cloud storage services that require advanced security policies. Just to illustrate, Amazon S3 protects data at rest using server-side encryption, which means that Amazon is who manages the encryption keys and can access all the information uploaded by users. Personal cloud storage services may alternatively use client-side encryption to protect data. With client-side encryption, the user client could encrypt the data client-side and upload it to Amazon S3. However, this approach would require the participation of the control cloud much more intensively. The control cloud would need to manage encryption keys and security policies to enable the secure sharing of data among multiple users (e.g., collaborative file editing).

[3] Multipart upload enables a client to upload a single object as a set of independent parts and in any order. Each part is a contiguous portion of the contents of the object. If transmission of any part fails, the client can retransmit that part without affecting the rest of the parts.

To circumvent the rigidity of object storage APIs, the *software-defined storage (SDS)* paradigm has emerged as a compelling solution to this problem [98, 53]. In this new paradigm, the control and the data flows are split into two major components, namely the control and data planes. Among other things, this separation ensures better modularity of the storage stack, and introduces differentiated I/O treatment of data flows. SDS inherits legacy concepts from Software-Defined Networking (SDN) [42] and applies them to storage-oriented environments, bringing new insights to storage stacks, such as improved system programmability and extensibility [89]. Although there exist some SDS systems for object storage such as Crystal [29], the existing solutions have so far put the spotlight on performance and data management, such as compression, cache optimization, and bandwidth differentiation, leaving software-defined security aside and out of the picture. This observation set off the second main question of this thesis:

Question II: In collaborative scenarios such as those of personal cloud storage services, could security data management be outsourced to the storage cloud by applying the principles of software-defined storage?

Put another way, the second question of this thesis explores the benefits of applying the software-defined principles to secure data management in object stores. In this sense, it is easy to envision a system where the control plane would hold the intelligence of the system and consist of a logically centralized controller. And a data plane, e.g., implemented as a serverless[4] programmable framework, which would allow the injection of programming logic such as encryption primitives to perform custom computations over object requests.

[4] By "serverless", we refer to the so-called Function as a Service (FaaS) model that helps developers work better by removing the need for them to maintain application infrastructure, letting them to focus on writing the programming logic.

Finally, this thesis provides some preliminary results of porting the software results of Question II to the edge. One of the primary drivers of modern storage systems is the proliferation of Internet of Things (IoT) devices, or more specifically, what is known as edge computing. Unfortunately, building an edge object storage system is far from trivial due to numerous challenges, such as the stringent resources of the edge servers. For a storage system to be viable at the edge, it must provide speed, resilience and security with very little compute and storage resources. Certainly, this attribute led us to study whether or not object storage can be the storage of choice for the edge. To answer this question, we provide some preliminary results to show to what extent this idea is feasible by providing a unified platform that can handle both storage and data transformations in a single edge server.

Chapter 2

State of the Art and Background

This chapter reviews the state of the art relative to our contributions with the aim to bring the reader closer to the research problems that motivate this thesis.

Data varies in its access patterns, and as a result, it is optimally stored on different types of storage systems. There are three primary categories of data storage: block storage, file storage, and object storage [7].

This thesis is exclusively focused on the study and proposal of improvements for open-source Software-Defined Storage (SDS) systems, specifically those that employ a data-plane based on storlets [65].

As such, the structure of this chapter will be as follows:

Our discussion will begin with an explanation of the term ’object storage’, outlining its basic concept and distinguishing features. Following that, we will delve into the internal structure of object storage, highlighting its key components and how they interact. Next, we will explore a specific application of object storage, focusing on personal cloud storage systems to illustrate its practical use. Finally, we will examine software-defined storage (SDS), discussing its definition, structural components, and the various options available in the current technological landscape.

2.1 Generic Object Storage

Object storage does not allow access to raw data blocks or file-based access. Instead, it grants access to entire objects or blobs of data, typically through a system-specific API (see §2.1.3 for further details). These objects can be accessed through URLs using HTTP protocols, similar to accessing websites in web browsers. Object storage uses URL abstraction to allow the storage system to expand and scale independently of the underlying storage mechanisms. This characteristic makes object storage particularly suitable for systems requiring scalability in terms of capacity, concurrency, or both.

One of the primary benefits of object storage is its capability to spread requests for objects across numerous storage servers. Major cloud service providers, including AWS with their S3 service, Microsoft Azure's Blob Storage, IBM's Cloud Object Storage (COS), Oracle's OCI Object Storage, and Google's Cloud Storage, offer disaggregated storage solutions [3, 36, 44]. This distribution enables reliable, scalable storage solutions for vast data quantities at a comparatively low cost.

2.1.1 Overview

In object storage, each piece of data is treated as a distinct object. Each object includes the data itself, a unique identifier, and metadata. The metadata is extensive and can include details about the data's security, retention policies, and other attributes. Figure 2.1 showns main parts of an object storage system. Now we are going to highlight the most important components:

(i) ***Unique Identifiers:*** Objects are stored in a flat address space and are accessed through unique identifiers (like a URL in web-based storage). This differs from file systems which use a hierarchical structure of directories and file names (see §2.1.3 for further details).

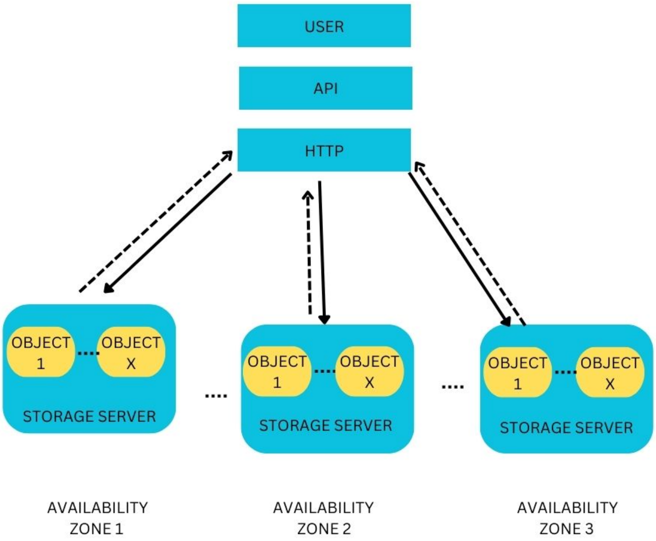


FIGURE 2.1: Object storage high level generic structure.

(ii) ***Scalability:*** Object storage is highly scalable. It can manage vast amounts of unstructured data, making it ideal for cloud storage solutions and big data applications. For example, S3 uses a system of prefixes, akin to unique file paths in operating systems. The design of S3 partitions these prefixes[1] a feature that allows the system to efficiently handle thousands of requests per second as stated on [92, 51].

(iii) ***Metadata:*** The rich metadata allows for more sophisticated management and data analytics. It can be used for automating tasks like data indexing, archiving, or tiering.

(iv) ***Data Durability and Reliability:*** Object storage often includes built-in features for data durability, such as replication and erasure coding, making it highly reliable and suitable for long-term data retention. Recently, in 2020, AWS updated S3 to

[1] You can use prefixes to organize the data that you store in Amazon S3 buckets. A prefix is a string of characters at the beginning of the object key name [4]

function as a strongly consistent[2] system [5, 92], aligning with the capabilities of other cloud providers who already offered strong consistency.

(v) ***API Access:*** Access to object storage is typically provided through APIs (see §2.1.3 for further details), such as the S3 API from Amazon Web Services, which has become a de facto standard.

2.1.2 Swift: example as infrastructure

Swift is a multi-tenant, highly scalable, and durable object storage system, engineered to store substantial volumes of unstructured data at a low cost. It is utilized by a diverse range of entities, including businesses of various sizes, service providers, and research organizations around the globe. Swift is commonly used for storing unstructured data like documents, backups, images,etc.

Swift [7] enables the storage, retrieval, and deletion of objects along with their related metadata in containers using a RESTful HTTP API. Developers have the flexibility to either interact directly with the Swift API or utilize one of the numerous client libraries available for widely-used programming languages, including Java, Python, and more.

Figure 2.2 shows a high-level overview of the components of a common OpenStack Swift deployment. These components [67], 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).

At the core of Swift's architecture lies the Proxy Server, which serves as the central connecting component for the rest of the architecture. Functioning as a gateway to the storage nodes where data is physically stored, the Proxy Server also houses the public API that clients use to submit requests. For every request, it identifies the location of the relevant account, container, or object within the ring and directs the request appropriately. The ring itself is a crucial component that maps the names of stored entities to their physical locations on disk. Once established, a ring constitutes a fully operational storage policy, enabling the differentiation of service levels.

[2] Amazon S3 delivers strong read-after-write and list consistency automatically for all applications. What you write is what you will read.

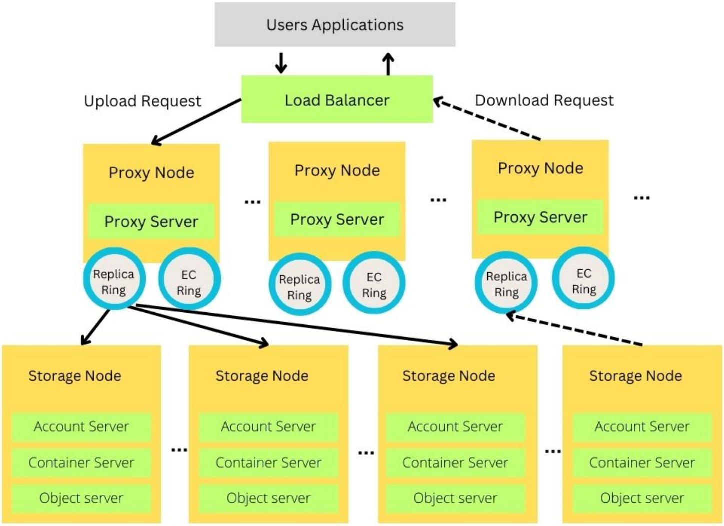


FIGURE 2.2: Swift Architecture.

Conversely, the role of Account Servers is to manage listings of containers, whereas Container Servers primarily deal with the listings of objects. The Object Server, on its part, is a straightforward blob storage server capable of storing, retrieving, and deleting objects on local devices. These objects are stored as binary files in the filesystem, with their metadata kept in the file's extended attributes (xattrs). Each object's storage path is determined based on the hash of the object's name and the timestamp of the operation.

Key characteristics

Swift is engineered to scale linearly, adapting to the quantity of data that requires storage and the number of users it needs to accommodate. To scale up, the system expands in the necessary areas – by incorporating more storage nodes to boost storage capacity, adding proxy nodes to handle an increase in requests, and enhancing network capacity to alleviate identified bottlenecks. To enhance its reliability and availability, Swift not only replicates and distributes object copies within a cluster but also across different data centers.

Utilizing a shared-nothing approach[3], Swift is able to fully leverage the available server capacity to handle numerous requests concurrently. This architecture allows it to operate efficiently on a range of commodity hardware. Additionally, Swift offers storage policies enabling operators to utilize hardware optimally, tailored to the specific demands and constraints of different scenarios.

Moreover, Swift supports the development and implementation of middleware directly on the storage system. This capability allows for the customization and enhancement of storage functionalities, catering to specific use cases and requirements (see §2.1.4 for further details).

2.1.3 Swift API

The Swift RESTful HTTP API encompasses a defined set of rules that outline the various types of HTTP requests you are permitted to send to a Swift cluster. These rules also specify the kinds of success and failure responses you can expect to receive under different circumstances, along with the details of the data contained in those responses. For each incoming request, Swift must first verify the identity of the requester and determine whether they have the necessary permissions before it processes the request and provides a response. Within the Swift cluster, the proxy server process is the exclusive component that interacts with external clients. This exclusivity stems from the fact that only the proxy server process implements the Swift API. In simple terms, the Swift API is an HTTP-based protocol consisting of a specific set of rules and terminology that the proxy server process utilizes to communicate with external clients.

[3] In a share-nothing architecture, each node (which could be a server, or a cluster) operates independently and autonomously, with its own private resources, such as memory, disk storage, and processing power.

For example, a storage URL for an object in a Swift cluster might look like this:

`https : //server.com/v1/account/container/object`

From this URL we can extract the next information:

(i) **server.com/v1** This part is an endpoint into the cluster. Proxy server on a node can now handle your request.

(ii) **/account/container/object** This part is formed with one or more identifiers that make up the unique location of the data. This hierarchical structure dictates that to reach a particular object, you should sequentially specify the account, followed by the container, and finally the object within that container.

Swift utilizes the standard HTTP verbs for different actions. These verbs and their corresponding actions in Swift are as follows:

(i) **GET** This is used to download objects (along with their metadata), or to list the contents of containers or accounts.

(ii) **PUT** This verb is employed to upload objects, create containers, or overwrite metadata headers.

(iii) **POST** It is used for updating metadata for accounts or containers, overwriting metadata for objects, or creating containers if they don't already exist.

(iv) **DELETE** This command deletes objects or containers that are empty.

(v) **HEAD** Used to retrieve header information, including metadata, for an account, container, or object.

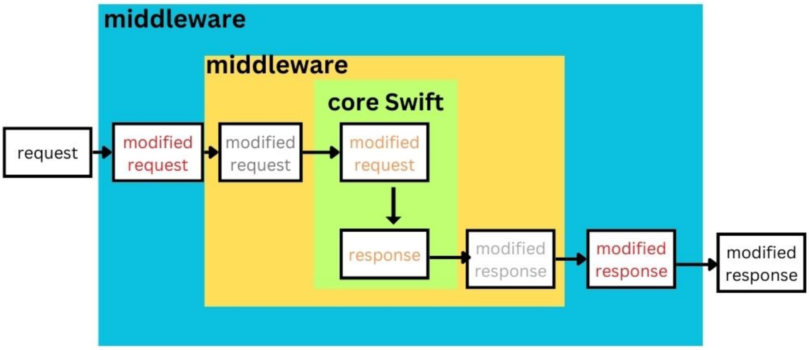
2.1.4 Swift Middleware

Swift comes equipped with various middleware packages that enhance the functionality of its installation. Middleware presents an excellent opportunity to tailor Swift to meet your specific requirements.

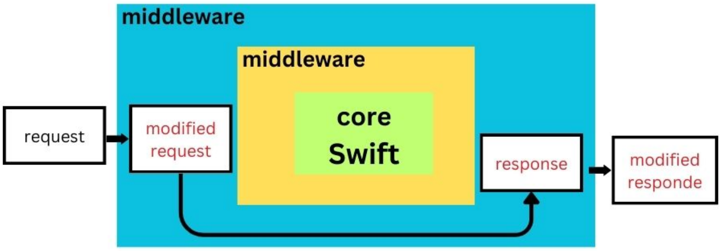
Swift is built upon Python's Web Services Gateway Interface (WSGI) model and is configured using the Python Paste framework. One of the key strengths of WSGI is its middleware which effectively envelops around other middleware layering down to the core application at the center, which in this context is Swift itself.

A significant advantage of this structure is that each middleware layer operates independently, without needing to be aware of the other layers, including the core application layer (Swift in this instance). This independent functionality allows middleware code to be relatively straightforward and modular. This modularity facilitates easy customization and extension of Swift's capabilities to suit various use cases and operational environments. In Figure 2.3 you can observe the interaction between the middleware layers and how a request or response is bypassed through the system. This illustration provides a clear visual representation of the middleware workflow, demonstrating the path of a request as it navigates through the various layers, and how certain responses can effectively shortcut the remaining stages in the pipeline.

In summary, each middleware layer in Swift's architecture has the option to inspect a request and modify it if necessary. After this inspection and modification, the middleware can either pass the request along to the next layer in the pipeline or provide a response that effectively bypasses the remaining middleware layers. Similarly, on the return trip of a request, each middleware layer has the option to once again inspect and potentially modify the request before it proceeds further. This flexible middleware structure allows for a high degree of customization and control over how requests are processed within the Swift system.



(A) Middleware pipeline. Each layer has the option to inspect a request and modify it if necessary.



(B) Middleware bypass. Certain responses can effectively shortcut the remaining stages in the pipeline.

FIGURE 2.3: Middleware pipeline and bypass mechanism.

2.1.5 Access control

Swift provides two levels of access control: account-level and container-level, each with its own set of privileges and functionalities.

Account-Level Access Control. This access control grants privileges across an entire account and is typically used for broad account's resources management which is suitable for administrators or users needing extensive access across the account.

- A user with `read-only` account-level access can download any object in the account, list objects in any container, view all containers, and access any non-sensitive metadata about objects, containers, or the account.
- `Read-write` account access includes all read-only privileges, plus the ability to create and overwrite objects, create new containers, delete objects or empty containers, and set any non-sensitive container or object metadata.
- `Admin` account access provides full ownership of the account to the authorized groups, allowing complete control over the account and its resources.

Container-Level Access Control. This access control level grants privileges within a specific container. Container-level access can be configured to allow permissions even to unauthenticated users for reading and/or writing objects within the container. Container-level access control is ideal for more granular control over specific containers, useful in scenarios where different containers within the same account have varying access requirements.

The ACL for a container is formatted as a comma-separated list that may include:

- Authorization groups: Specified groups that are granted access based on the permissions set.
- Referrers: External entities or domains that may be granted access.
- The keyword ".rlistings" indicates that clients can retrieve a list of objects within the container.

```
1 X-Account-Access-Control:{
2                              "admin":["AUTH_alice","sysadmins"],
3                              "read-only":["AUTH_bob"]
4                            }
```

LISTING 2.1: Account Access Control ACL Swift example.

In Listing 2.1, as illustrated in the example from Swift ACL [7], we see the configuration of access control settings. Specifically, in line 2, user `Alice` and the group `sysadmins` are granted full control over a resource, which could be an account, a container, or an object. This level of control includes capabilities like modifying, deleting, or adding resources. Additionally, in line 3, user `Bob` is provided with permissions to list, download, and view the metadata of that same resource. This setup showcases the flexibility of Swift's ACL system in defining varying levels of access for different users or groups to the same resource.

2.2 Personal Cloud Storage

Personal Cloud Storage (PCS) services are data-intensive applications accessible via the Internet that enable users to synchronize files with servers in the cloud and across various devices. These services have become a popular tool for distributed and collaborative work, allowing files to be automatically shared and synchronized among users, thereby introducing additional requirements for near real-time data access and synchronization [2].

2.2.1 Overview

The convenience of storing personal files, synchronizing devices and sharing content with ease has significantly increased the appeal of these services. As a result, more individuals are utilizing them for their personal and professional needs. This widespread popularity has led to a competitive cloud storage market, with a multitude of providers vying for users' attention [60].

The market includes established players like Dropbox and Box [24, 16], as well as tech giants such as Apple, Google, and Microsoft. These larger companies are able to offer substantial amounts of storage space at increasingly affordable prices, further intensifying the competition in the cloud storage sector. This competitive landscape challenges new entrants to differentiate themselves and offer compelling services to attract and retain users.

PCS services offer users highly user-friendly tools, particularly valuable in the context of the growing variety of user devices that need synchronization. Owing to the availability of these resources and tools, many of which are free, there is a tendency for users to upload increasingly substantial volumes of personal and private data [31].

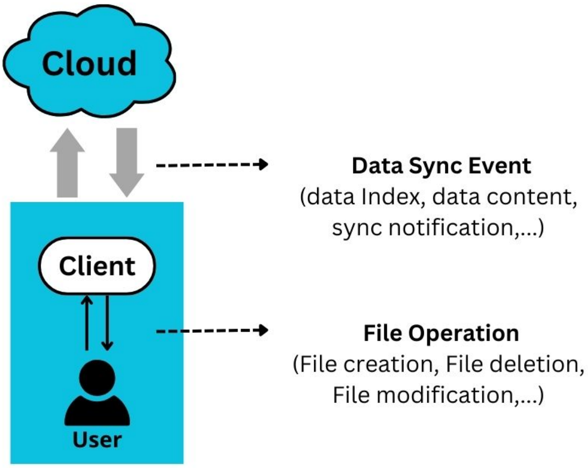


FIGURE 2.4: Personal cloud storage communication schema.

The core function of cloud storage services is data synchronization (sync), a process that automatically reflects changes made in users' local file systems to the cloud. Synchronizing a file in cloud storage services encompasses a sequence of data sync events. These events include transferring the data index, the actual data content, a sync notification, sync status / statistics, and finally, a sync acknowledgement. Each of these data sync events generates network traffic, contributing to the overall communication load between the local system and the cloud. This process that is visually depicted and explained in Figure 2.4.

2.2.2 Synchronization on personal cloud storage

In this section, we address another common issue in the development of cloud storage: the overuse of network resources during the synchronization process. The synchronization (sync) traffic in these services accounts for more than 90% of the total service traffic, indicating a significant portion of data transfer within these systems. Interestingly, it is noted that this total service traffic is equivalent to one-third of the traffic generated by a major online platform like YouTube [23]. This comparison highlights the substantial amount of data involved in sync processes, suggesting that a part of this sync traffic might be avoidable or reducible with more efficient synchronization strategies.

Mainstream cloud storage services — for example Dropbox or Baidu Netdisk [8, 24]— typically adopt a two-cloud system architecture. One cloud, the object storage cloud, serves as the hosting platform for users' file content. The index/control cloud, on the other hand, is utilized to maintain users' account information, online status, file metadata (such as directories), block index (when applicable), and other related data. This structure is depicted in Figure 2.5, providing a visual representation of how the object storage cloud and the index/control cloud function together in the system.

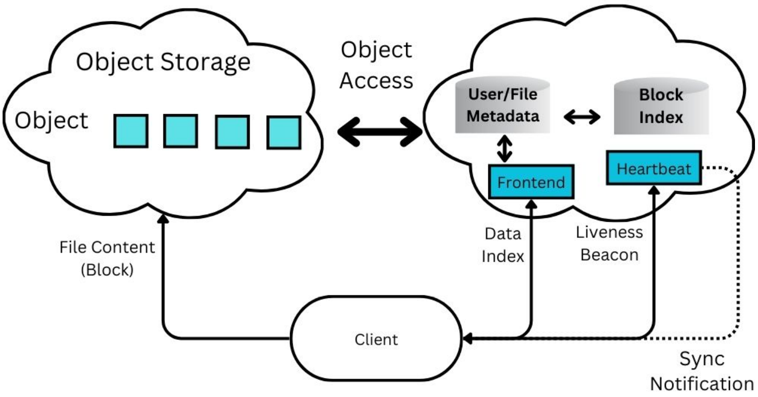


FIGURE 2.5: Personal cloud storage basic communication schema.

In Figure 2.5, three distinct types of sync traffic are exchanged to facilitate standard data synchronization activities:

(i) Connection with Front-End Server:

  (a) Each client maintains a connection to a front-end server.

  (b) This server authenticates the user's account and stores metadata about their files, including a list of files, their sizes, attributes, and locations in the object storage cloud.

  (c) If files are stored as blocks in the object storage cloud, a corresponding block index is linked with the user/file metadata index.

(ii) Connection to Object Storage Server:

  (a) Clients establish connections with an object storage server to transfer the content of files or their blocks.

(b) For transferring large files, clients may create additional connections to multiple object storage servers, enabling parallel transfers and increased throughput.

(c) Clients capable of data compression compress files or blocks before uploading to the object storage cloud.

(d) With modifications in synced files, and if incremental sync is supported, clients will first check a local cache or consult the front-end server to identify file updates. These updates are then uploaded as compressed binary diffs to the object storage cloud.

(iii) Connection to Heartbeat Server:

(a) Clients also keep a connection to a heartbeat server.

(b) They periodically send a status signal to the heartbeat server to confirm their online presence and receive sync notifications from the index/control cloud (like updates made to shared files by other users).

In addition to these three sync traffic types, there's also inter-cloud traffic between the two clouds, facilitating the sync process. For instance, during incremental sync, the front-end server might need to access data from the object storage cloud to determine file differences. This multi-layered approach ensures comprehensive and efficient data synchronization across the network.

2.2.3 Minimizing synchronization network use

In examining the sync mechanisms in [46], we identified three main streams: intra-file approaches, cross-file approaches and batching approaches. Each stream represents a distinct method of optimizing the data synchronization process explained below:

Intra-File Approaches

This includes techniques like varying the data compression level for each file, handling interrupted file transfers, and applying data deduplication within a single file to minimize redundancy.

- **Compression:** varying levels of compression are applied depending on the access method. Notably, the compression level on the server side is generally set higher than on the client side, ranging from 5% to 30% more compression. This means that data downloads typically consume less traffic than uploads from a user's client perspective, due to the more efficient compression applied on the server side for stored data. Moreover, studies reveal that the compression technique is applied differently by each cloud storage provider [46]. For instance, Dropbox [24] applies compression both during file upload and download. In contrast, Seafile [87] only uses compression during file uploads, while SugarSync [96] does not employ compression in either uploads or downloads.

- **Incremental sync:** can greatly reduce the sync traffic of file edits by transferring only the altered bits or blocks. There are three granularities in data synchronization techniques: full-file, block-level, and chunk-level. Google Drive exemplifies full-file sync, where the entire file is synchronized. Seafile, on the other hand, employs Content-Defined Chunking-based sync (CDC)[4]. Dropbox [24], iCloud [37] , and SugarSync [96] utilize `delta-sync` on the client side, but they differ in the chunk sizes they use for synchronization.

- **Interrupted transfer resumption (ITR):** Most current services support ITR by employing different sequences (either sequentially or in parallel) and granularities for block transfers, which typically range between $128KB$ and $32MB$ [46].

[4] Dynamic chunking mechanism [69]

Cross-File Approaches

This might include protocols for efficient data delivery between files, such as choosing between a Client/Server or Peer-to-Peer model, and methods for identifying and eliminating duplicate data across different files.

- **Deduplication:** can avoid the transmission of data already stored both on the server side and client side. Block-level dedup exhibits trivial superiority (in terms of traffic saving) to full-file dedup but incurs much higher computation complexity [46].

- **Peer-assisted offloading (P2P) :** can considerably cut down the cloud-side traffic cost by around 25% for delivering a popular file shared by multiple users. Only Baido Netdisk [8] adopted this mechanism for the moment [46].

Batching Approaches

This includes strategies like file or chunk bundling and sync deferment, which delay the synchronization process until a significant batch of data is accumulated, thereby reducing the frequency of sync operations and potentially saving network resources.

- **Bundling:** consolidates several small data modifications into a larger combined change for network transmission. This method is effective in decreasing the overhead linked to sending continuous minor changes to keep all data synchronized.

- **Sync deferment:** postponing synchronization tasks until a more suitable moment, which helps in decreasing the network overhead. There are adaptive techniques in place that adjust the timing of data synchronization not at predetermined intervals, but in response to the prevailing system conditions, the extent of data alterations, network traffic, and other relevant factors. When handling frequent edits, iCloud [37], Drive [56] and SugarSync [96] degrade from incremental sync to full-file sync, thus aggravating the traffic overuse. However, fixed sync deferments are inefficient in certain scenarios [46].

2.2.4 Synchronization Deferment

Some studies indicate that for approximately 8.5% of Dropbox [24], users frequent edits are responsible for more than 10% of their sync traffic [**13**]. This suggests that a significant portion of the network resources utilized by these users is due to the continuous files updates. Moreover, these frequent edits can lead to a substantial amount of overhead traffic which may greatly surpass the volume of useful data updates transmitted by the user client over time. This issue is known as the traffic overuse problem [**13**]. To address this challenge, some cloud storage services deliberately delay the synchronization process for a fixed duration. This deferral aims to batch file updates together, thereby reducing the frequency of sync operations and consequently, the associated overhead traffic.

Consider a situation where multiple apps on a single device are frequently updating a shared key-value store. If every change immediately triggers a synchronization with the server, it would result in an overwhelming number of network requests. This could significantly strain the network bandwidth, potentially causing higher latency and reduced battery life in mobile devices due to persistent network activity. As the frequency of these updates increases, the system begins to defer certain changes to alleviate network traffic. However, this deferral means that updates may not appear instantly on other devices. Finding the right balance in this scenario is challenging. It is critical to maintain network efficiency and ensure the system remains responsive and functional, but doing so without overloading network resources or compromising data immediacy is a complex task.

Network overhead

Addressing the issue of overuse of network resources requires an initial understanding of what constitutes network overhead. Throughout the thesis, we will utilize the Traffic Usage Efficiency (TUE) [5] metric [48] to quantify the network overhead, which is defined

[5] When a file is updated (created, modified, or deleted) at the user side, the data update size denotes the size of altered bits relative to the original local file (or the cloud-stored file if it is not compressed). From the users' point of view, the data update size is an intuitive and natural signifier about how much traffic should be consumed.

as:

$$TUE = \frac{\text{Total sync traffic}}{\text{Data update size}}, \tag{2.1}$$

where the *data update size* refers to the size of the altered bits in the update due to file creation, modification or removal. It intuitively signals the network overhead from the user side.

Adaptive Sync Deferment

We describe the *adaptive sync deferment* (ASD) algorithm presented in [48]. To the best of our knowledge, ASD is the only algorithm in the literature that *adaptively* adjusts the deferment time. We briefly revisit it here. It works as follows:

Upon the ith update at time t_i, the idea behind ASD is to adaptively tune the sync deferment time window T_i, such that if the next update falls within the range t_i to $t_i + T_i$, then it is deferred and marked as pending in the client. Otherwise, all the pending updates are pushed to the cloud backend. Specifically, T_i is tuned in an iterative manner as an ***E****xponentially* ***W****eighted* ***M****oving* ***A****verage* (EWMA) controller:

$$T_i = \min\left((1-\omega)T_{i-1} + \omega\Delta t_i + \epsilon, T_{max}\right), \tag{2.2}$$

where Δt_i is the inter-update time between the (i-1)th and the ith data updates, and $\epsilon \in$ (0,1.0) is a small constant that ensures T_i to be slightly longer than Δt_i in a small number of iteration rounds. T_{max} is a constant representing an upper bound on T_i, to prevent a large T_i from harming user experience due to long sync delays.

By a simple inspection of (2.2), it is easy to see that ASD does not account for the size of updates. It focus only on one single dimension: *inter-update time*. Although this metric is sufficient to reduce the TUE, it does not always minimize the sync delay. An improvement of this mechanism is proposed on this thesis on chapter §3.

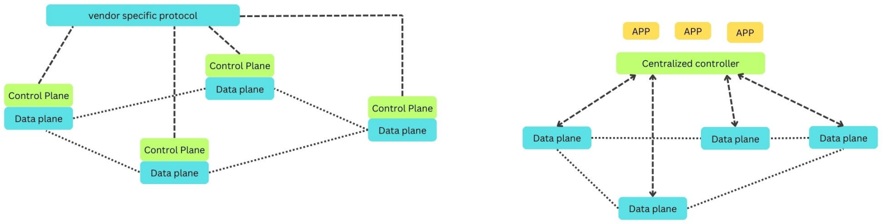
2.3 Software-Defined Storage

Software-defined Storage (SDS) can be succinctly described as storage orchestrated by software. The core concept behind 'software-defined' technology lies in detaching the control software from the hardware layer, which facilitates centralized and streamlined management of the infrastructure. SDS conforms to this paradigm by using a software-centric control layer that operates independently from the physical components like storage servers, disks or arrays. Currently, this technology is viewed as a facilitator for the architectural design, configuration and operation of a storage system [19], providing optimized and automated control and administration to improve the efficiency of resource utilization.

2.3.1 Overview

SDS restructures the I/O stack from traditional storage frameworks. This reorganization separates the control and data flows into two distinct layers: control and data. In conventional storage systems, each I/O layer necessitates the development of specific control tasks, such as coordination, queueing, metadata management, and monitoring at each layer. However, SDS adopts a more unified system abstraction.

In this model, basic control operations such as I/O bandwidth allocation [89, 70, 98] are handled by a logically centralized control plane. These control operations are expressed using policies and implemented in form of control applications. This centralized control platform enables the implementation of various control functionalities, like I/O prioritization, data placement strategies, and rate limiting. These functionalities can be applied across a broad range of control granularities (such as per-user, per-tenant, or per-request) and in different environments like cloud computing, high-performance computing (HPC), and specialized application-specific storage stacks. The importance of the software-defined approach is that the aforementioned functionalities can be provided in a transparent way using control plane (micro-controllers, data policies) and data plane abstractions [49].



(A) Traditional system structure.

(B) Software-defined structure abstraction.

FIGURE 2.6: Comparison between high-level structure on traditional and SDS systems.

Figure 2.6 contrasts a monolithic approach with a Software-Defined Storage (SDS) approach, emphasizing the stark differences in flexibility and adaptability. In the monolithic model, storage components are tightly integrated, forming a single, unified system. This integration, while robust, makes modifications and scalability challenging, as any change often requires significant overhaul of the entire system. Conversely, the SDS approach is represented as a modular and dynamic structure, where storage resources and services are abstractly separated from the underlying hardware.

Fundamentally, SDS embodies a storage architecture designed to cater to a wide range of storage requirements, consisting of dynamically configurable software and hardware components. To fully realize the capabilities of SDS, implementations need to embrace these features, thereby creating a storage environment that is responsive, efficient, and adaptable:

(i) ***Programmatically administered:*** Programmable interfaces to support dynamic storage deployment, configuration and management, enabling policy-based automation of infrastructure storage resources.

(ii) ***Automation:*** Realization of autonomic data storage capabilities (provisioning, reconfiguration, etc.) to provide dynamic SLA configuration.

(iii) ***Monitoring:*** Metric collection for the automation of the storage in order to guarantee and validate that the users SLAs are met.

(iv) ***Scalability:*** High scalability of the storage virtualization, and pooling, which is essential to adapt the storage resources to the demand.

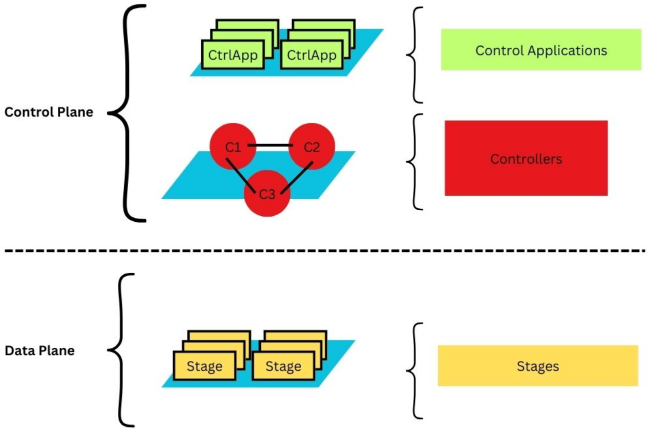
(v) ***Interoperability:*** Generic storage infrastructure and control components. The abstraction of functionality across underlying hardware eases systems integration and configuration of infrastructure components.

SDS Architecture

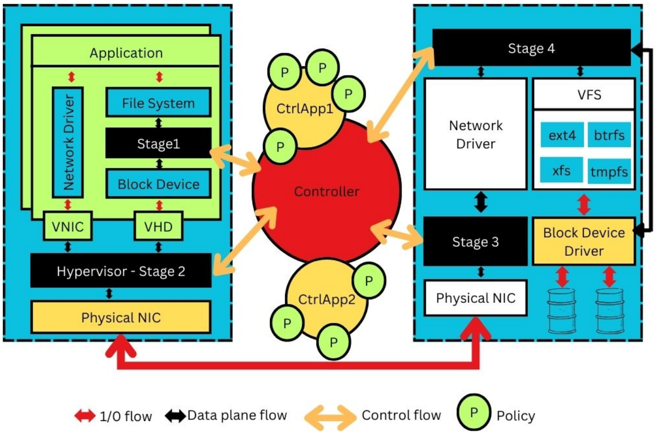
A SDS architecture is characterized by two principal functional planes. This architecture is visually represented in Fig. 2.7a and 2.7b , which illustrate a layered perspective of SDS functionality and an SDS-enabled storage infrastructure, respectively. The control plane, a key component of this architecture, is divided into two parts: first, it includes the global control building blocks essential for creating system-wide control applications. Secondly, it is the core of the SDS system's intelligence, featuring a logically centralized controller. This controller provides a comprehensive view of the entire system and centralizes control functions. Additionally, a number of control applications are developed on top of this centralized controller, further enhancing the system's capabilities [33].

Control applications act as the primary interface for the control environment, as shown in Fig.2.7b (`CtrlApp1` and `CtrlApp2`). They are the standard method for SDS users, such as system designers and administrators, to implement various storage policies across the storage infrastructure. These policies, comprising a series of rules, govern the management of I/O flow. They are established within the control applications, distributed by the controllers, and executed within the data plane.

The data plane in an SDS architecture is a programmable, multi-stage component that is distributed along the I/O path, as illustrated in Figure 2 (`Stage1 . . . Stage4`). It contains sophisticated, fine-grained storage services that are dynamically adaptable to changes in the infrastructure. Each stage within the data plane can be positioned within or in parallel to different I/O layers. These stages are designed to implement



(A) Layered view of the SDS planes of funcionality.



(B) SDS-enabled on top of generic multi-tenant infrastructure.

FIGURE 2.7: Generic SDS infrastructure. Control applications are the entry point of the control environment (*CtrlApp1* and *CtrlApp2*), and the de facto way of SDS users to express different storage policies (*P*) to be enforced over the storage infrastructure. Policies are a set of rules that declare how the I/O flow is managed, being defined at control applications, disseminated by controllers, and installed at the data plane., which is a programmable multi-stage component distributed along the I/O path. Each stage can be placed within or horizontally aligned with I/O layers, and respects to a distinct storage service to be employed over intercepted data flows, such as data management (e.g., caching, rate limiting), data transformations (e.g., compression, encryption), and data routing activities (e.g., flow customization, replica placement).

specific storage services on intercepted data flows. Services include data management activities like caching, rate limiting, and I/O prioritization; data transformations such as compression, encryption, and deduplication; and data routing functions like flow customization and replica placement.

For any intercepted I/O requests, applicable policies trigger the execution of associated storage services on the filtered data. Subsequently, the processed data is redirected back into the original I/O flow, exemplified in Figure 2 by the data flow between the File System, `Stage1`, and the Block Device.

2.3.2 SDS Systems: state of the art

As highlighted in earlier chapters, we will now explore in greater detail the systems that serve as the foundation for our study.

IOFlow

IOFlow [98] introduced the first comprehensive SDS architecture. It allows for the specification of end-to-end (e2e) policies that dictate how I/O flows are managed from virtual machines (VMs) to shared storage. This is achieved through a queuing abstraction in the data plane, which translates high-level policies into specific queuing rules. IOFlow offers rule-based policies that enable complex routing primitives (like creating/removing queues, configuring token buckets) for particular storage flows at the data plane. The system's automation capabilities mainly involve data services implemented as "stages".

Implementing IOFlow requires modifications at both the storage server and hypervisor levels to intercept I/O request flows, indicating that it is not inherently system transparent. Its control plane is centralized yet adaptable, capable of integrating new algorithms or bandwidth policies for controlling I/O routing across various stages. Similarly, the data plane is designed to be extendable, allowing the addition and discovery of new stages by the control plane.

sRoute

sRoute [94] introduces an innovative platform for executing computing services on data flows, utilizing sSwitches, which are an evolution of the stages concept found in IOFlow. In this system, the control plane enforces rule-based policies that outline the forwarding rules for directing I/Os from clients to the storage backend, passing through one or more sSwitches. These sSwitches, by intercepting and categorizing I/O flows, can incorporate data services such as custom replication (static data service automation) or tail latency control (dynamic provisioning).

A notable feature of sRoute is the concept of a "delegate function", which serves to distribute control rules across data plane stages, thereby achieving a degree of decentralization in the control plane. On the data plane, sSwitches can intercept I/O flows at multiple points, performing operations related to the classification, redirection, and prioritization of block requests, all without affecting the client VMs. The data plane is also designed to be extensible, allowing for the integration of new sSwitches via an API.

While sRoute still necessitates changes in the storage stack to add new sSwitches to a target storage server, thus limiting system transparency, a key distinction from IOFlow is that sSwitches can also reside in other servers. This implies that some sSwitches do not require code alterations in the target storage system, as they are added as external services where I/O flows can be rerouted.

Retro

Retro [52] offers effective resource management in multi-tenant distributed storage systems, allowing system designers to independently orchestrate and refine resource management policies, separate from the base system implementation. A centralized controller with a logically flat structure offers a comprehensive global view of resources and orchestrates various queue-based data plane stages. Retro [52] operates independently of specific systems and resources, with its data plane designed to abstract various resources (such as storage devices, CPUs, thread pools, and networks). It incorporates management-based storage features like I/O prioritization and rate limiting, which are implemented at predetermined locations along the I/O path.

Crystal

Crystal, as referenced in [29], enhances the resource management capabilities established by Retro [52]. It introduces an SDS architecture designed for object storage, aiming to ensure efficient resource sharing and performance isolation in environments with multi-tenant heterogeneous workloads. Within its control plane, Crystal employs a set of distributed controllers with a flat architecture. These controllers are designed to dynamically adjust storage components to accommodate the evolving needs of different tenants. Functioning as separate micro-services, each controller can be deployed during runtime and applies a unique control algorithm to implement policies at designated levels within the storage stack.

In this system, global controllers possess the ability to monitor and consistently administer the data plane stages across the entire system. Their primary duties involve enhancing monitoring events and disseminating storage policies throughout the controllers and data stages. On the other hand, automation controllers handle set monitoring metrics, employing these to accurately and immediately adjust stages using a trigger-based method. This approach is particularly used for managing actions related to object requests.

Regarding the data plane, as referenced in [64], the system utilizes customizable storlet-based stages within OpenStack Swift instances. These stages play a crucial role in delivering management and transformation services for storage. Equipped with adaptable filters, known as storlets [65], they enable developers to implement and execute tailored storage services that operate on incoming object requests. This framework [63] significantly boosts the flexibility and effectiveness of the storage service provision. A storlet [65] is the binary code deployed as a Swift object. 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 associated metadata. Moreover, the computation has no access to disks, network or to the Swift request environment. An intuitive illustration of this structure is presented in Figure 2.9, and its `invoke` interface shown on Listing 2.2, which class diagram is shown on Figure 2.8.

```java
public void invoke(ArrayList<StorletInputStream> inStreams,
                   ArrayList<StorletOutputStream> outStreams,
                   Map<String,String> parameters,
                   StorletLogger logger) throws StorletException;
```

LISTING 2.2: Storlet invoke interface.

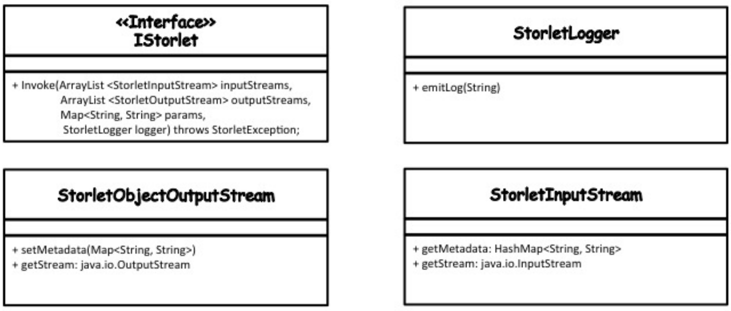

«Interface»
IStorlet
+ Invoke(ArrayList <StorletInputStream> inputStreams,
ArrayList <StorletOutputStream> outputStreams,
Map<String, String> params,
StorletLogger logger) throws StorletException;
StorletLogger
+ emitLog(String)
StorletObjectOutputStream
+ setMetadata(Map<String, String>)
+ getStream: java.io.OutputStream
StorletInputStream
+ getMetadata: HashMap<String, String>
+ getStream: java.io.InputStream

FIGURE 2.8: Class diagram illustrating the classes involved in the Storlet API.

Vertigo

Vertigo [83] offers a distributed framework that enables tenants to manage their data in a flexible and dynamic way. It features a programmable policy abstraction, termed a microcontroller. These microcontrollers are crafted to enable tenants to manage the distinct attributes of each object with versatility. Acting as wrappers, they regulate the behavior of these objects.

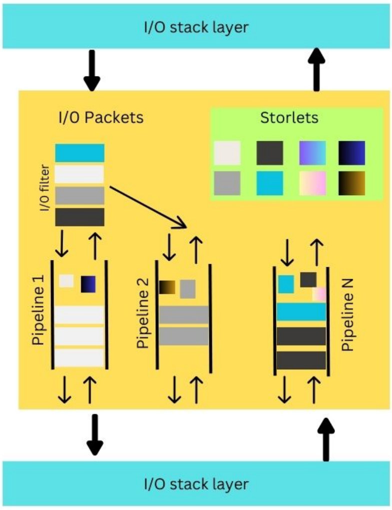


FIGURE 2.9: Abstract architecture of a storlet-based data plane stage. Stages comprehend a set of pre-installed storlets that comply with initial storage policies. An stage forms several storlet-made pipelines to efficiently enforce storage services over data flows. At runtime, I/O flows are intercepted, filtered, classified, and redirected to the respective pipeline.

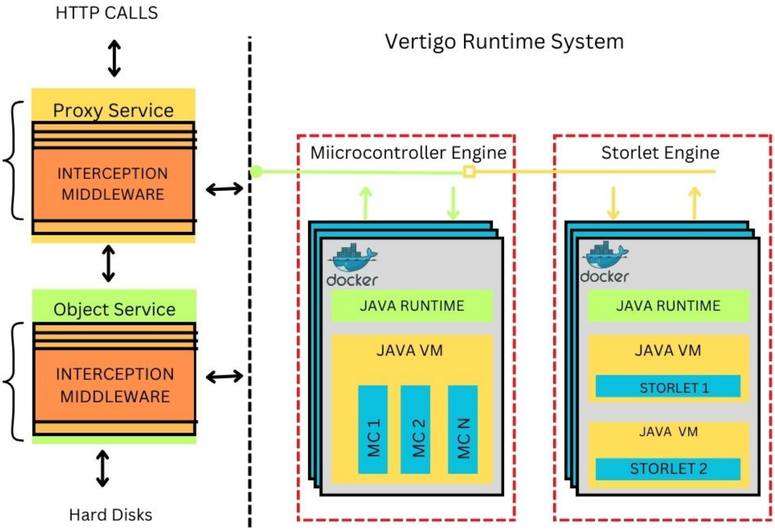


FIGURE 2.10: Vertigo and Microcontrollers structure.

Vertigo [83] utilizes microcontrollers that engage with and influence the lifecycle of objects within these nodes. A distinctive feature of these microcontrollers is their capacity to react to changes in the state of an object. Such reactivity facilitates the execution of intricate management policies. A high level abstraction of this structure is shown on Figure 2.10.

Additionally, microcontrollers go beyond mere monitoring functions. They are capable of orchestrating active storage tasks, adding an extra dimension of dynamic management to the system.

2.3.3 Near Data Processing in object storage systems

Serverless cloud computing [39, 33] adds another layer of abstraction to conventional cloud computing paradigms. This innovation effectively eliminates the necessity for developers to handle server-side management, as stated in [10].

To be classified as a serverless service, as outlined in [39, 33], it must scale automatically without the requirement for manual provisioning and charge based on actual usage. Currently, cloud functions represent a fundamental component in serverless computing and are pioneering a more simplified and versatile programming model for cloud-based environments.

Serverless cloud computing provides two primary services: Backend as a Service (BaaS) and Function as a Service (FaaS):

BaaS includes a range of services such as storage, messaging, and user management. BaaS offers developers a set of ready-to-use backend services that can simplify and accelerate the development process by managing the server-side operations.

FaaS allows developers to deploy and execute their code on cloud computing platforms without having to manage the underlying infrastructure. FaaS heavily relies on BaaS services like databases, messaging systems, and user authentication mechanisms. It is often considered the most dominant model in serverless computing and is synonymous with “event-driven functions” [27, 43]. This model is designed to respond to specific events, executing code in response to triggers such as HTTP requests, database changes, or message queue actions.

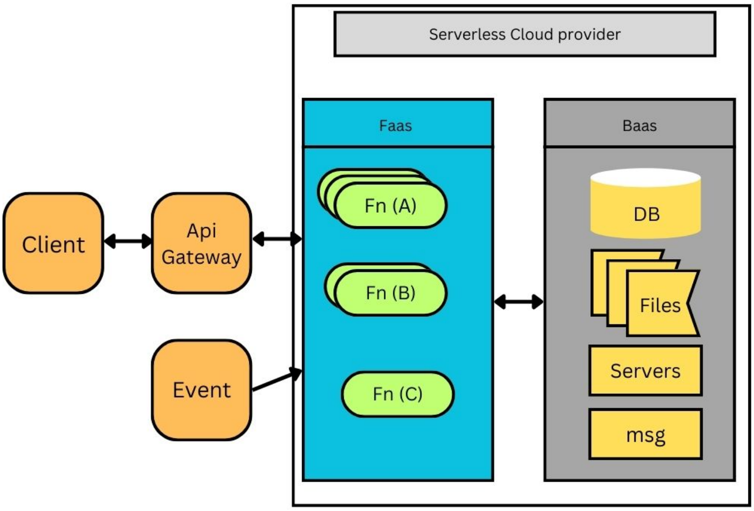


FIGURE 2.11: Serverless cloud computing offers *backend as a service (BaaS)* and *function as a service (FaaS)*. The *BaaS* includes services like storage, messaging, and user management. While, the *FaaS* enables developers to deploy and execute their code on computing platforms. The *FaaS* relies on the services provided by the *BaaS* such as a database, messaging, user authentications.

As an example, developers have the capability to deploy their applications onto the serverless cloud in the shape of functions, as depicted in Figure 2.11. This methodology streamlines the development process by automating the management of the underlying infrastructure. Consequently, developers can concentrate predominantly on coding and enhancing the functional aspects of their applications.

2.3.4 NDP as serverless storage functions

The initial wave of research in NDP emerged in the 1970s and 80s, marked by the development of "database machines" that integrated computing functions into the storage/memory systems. This period saw references [15, 22, 34, 68, 95] highlighting these advancements. The 1990s experienced a renewed interest in NDP, particularly in the fields of databases, along with expansion into image processing and data mining, frequently utilizing "Active Disks" [1, 40, 79, 78]. During this time, data was typically presumed to be located on the same device where computations occurred, often overlooking the complications related to sharding in distributed storage systems.

The initial mainstream adoption of NDP systems emerged in the mid-2000s with the development of Hadoop and the HDFS ecosystems, as referenced in [21]. Although HDFS and Hadoop are capable of managing certain simple data types that may be distributed across blocks and nodes, the emergence of computational storage devices reintroduces a fundamental challenge. This challenge involves records that are divided across devices and are unable to utilize in-situ computations effectively without extra complexity or the need for data collation.

Ceph [107] and OpenStack Swift [66] each implemented their unique NDP features. For Ceph, this capability was facilitated through dynamic classes [106]. In contrast, Swift introduced storlets, which are compact applications designed to be executed within a Swift deployment [25, 62, 65]. For more detailed information about OpenStack Swift, please refer to Section 2.1.2.

Ceph [107] is an object storage based free software storage platform that stores data on a single distributed computer cluster, and provides interfaces for $object-level$, $block-level$ and $file-level$ storage. At its core, Ceph features RADOS [6], a fully distributed, reliable, and self-managing object store. The fundamental components of Ceph are the OSDs. These OSDs are tasked with storing objects on local file systems and collaboratively working to replicate data. They play a crucial role in detecting and recovering from failures, as well as managing data migration when OSDs are added to or removed from the cluster. The design philosophy of Ceph is grounded in the understanding that failures are a common occurrence in large-scale storage systems. Therefore, Ceph is engineered to guarantee both reliability and scalability, primarily through the intelligent functionalities of the OSDs. Ceph storage clusters are designed to run on commodity hardware, using an algorithm called Controlled Replication under Scalable Hashing

[6] Reliable Autonomic Distributed Object Store

(CRUSH) to ensure data is evenly-level distributed across the cluster and that all cluster nodes can retrieve data quickly without any centralized bottlenecks.

As llustrative examples of severless NDP storage systems, one can refer to systems like: Zion [84], Shredder [110] and Glider [11]. The following is a brief description of each of these systems.

ZION

As representative examples of serverless NDP storage systems, we can refer to Zion [84], which is a data-driven serverless computing framework designed for cloud object stores. Its development aims to address the challenges of scalability and resource contention. A key advantage of Zion is its utilization of data locality to minimize latency, achieved by positioning the serverless compute layer strategically between the proxy and storage nodes.

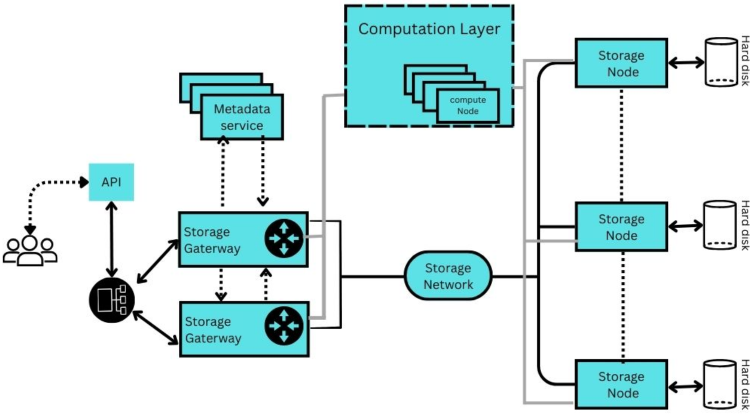


FIGURE 2.12: Zion presents a *disaggregated computing layer* between the storage and gateway nodes for executing the functions, a *metadata service* for triggers management and an *interception software* (depicted as a router) running in the storage gateways, which inspects incoming requests and reroutes them to the compute tier if necessary.

Zion [84] is implemnted on top of OpenStack Swift [64] and has three main components: a disaggregated computing layer, a metadata service and an interception software. All of them depicted on Fig. 2.12 and explained below.

The computing layer, which is disaggregated, operates as a collection of containers. These containers are responsible for executing the functions. A function, in this context, is defined as a computation code that intercepts the data flow. It processes the data inline, either as the object is incoming or outgoing from the storage cluster.

Each function, a simple example is shown Listing.2.3, is allocated a separate Linux container to ensure it does not disrupt other cloud functions. These containers, each with a running function, are termed 'workers'. Depending on the workload, a function can have varying numbers of workers active simultaneously.

```java
public class Handler implements IFunction {

  public void invoke ( Context ctx , API api ) {

    while((data = ctx.object.stream.read())){
      ctx . object . stream . write ( data );

    }
  }
}
```

LISTING 2.3: A simple function that echoes the data passed to it.

The interception software, that is seamlessly incorporated into the storage gateway (illustrated as a router in Fig.2.12). This software's primary purpose is to handle the deployment of functions, link triggers to these functions, and facilitate their activation upon a request aligning with a trigger. A trigger consists of a URL accompanied by prefix and suffix filters, akin to those in AWS Lambda for Amazon S3, and is associated with an HTTP method, which includes GET, PUT, POST, and DELETE.

The metadata service, which efficiently pre-processes and indexes trigger metadata. It also plays a crucial role in redirecting the input flow towards an available worker as the object is read. Data flows that are not intercepted continue rapidly along the default storage path, bypassing the serverless compute layer as is standard practice.

Shredder

Internally, Shredder [110] is composed of three distinct layers: a networking layer, a storage layer, and a function layer. Each CPU core independently operates all three layers. This is facilitated by a shared-nothing architecture, ensuring that each layer's state is partitioned across CPU cores to minimize contention and synchronization overheads. This architecture and its components is shown on Figure 2.13.

The **storage layer** is responsible for hosting all tenants' data in memory, offering a basic key-value interface through GET and PUT functions. The **network layer** takes charge of managing network connections, processing protocols, and handling requests from all tenants. It directly interacts with the storage layer for reading and writing data using the key-value interface. For requests requiring specific storage functions, the network layer forwards these to the function layer.

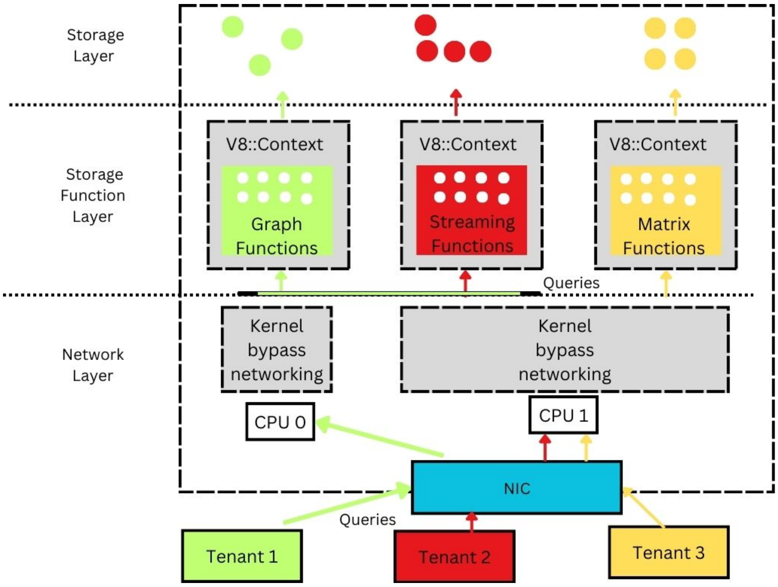


FIGURE 2.13: Three layers Shredder's architecture.

The serverless **function layer** plays a crucial role by matching incoming requests with the appropriate storage function code and its context, which includes the environment and state linked with that storage function. It executes these operations within a dedicated per-core instance of the V8[7] [100] runtime. To streamline interactions between the function runtime and the storage layer, each V8 runtime incorporates a set of embedded trusted access methods, thereby avoiding costly inter-layer calls.

Shredder's storage functions are designed to enhance the storage data model, allowing safe utilization of low-level hardware. These functions are also safeguarded by cost-effective language-level guarantees, ensuring efficient and secure operations.

Glider

Concurrently with the writing of this thesis, D. Barcelona et. al have proposed Glider [11] which introduces a novel approach to addressing the challenge of data movement in serverless computing, particularly concerning the intermediate data generated during compute stages. This approach revolves around the idea of minimizing data transfer by reversing the conventional process: instead of moving data to the code (compute stages), it proposes moving the code to where the data resides. This is achieved through a unique service model for ephemeral computational storage, facilitating near-data computation. By enabling data-bound operations to be executed close to the data storage, Glider significantly reduces both the number of connections required from serverless functions to storage and the volume of data transmitted.

[7] Google's open source high-performance JavaScript and WebAssembly engine, written in C++.

Chapter 3

Rate-based Sync Deferment for Personal Cloud Storage Services

Cloud storage services like Dropbox, Google Drive, and OneDrive, to cite a few, are becoming an increasingly “vital” tool in our everyday life. Unluckily, these services can incur large network overhead in different usage scenarios. To reduce it, these systems utilize several techniques like source-based deduplication, chunking, delta compression, etc. One of these techniques is sync deferment, which relies on the packing of updates to intentionally defer the synchronization process for some time, and increase the volume of useful data per overhead byte. The scientific literature has shown this technique to be very helpful, though there are still some limitations on current solutions. To resolve them, we present here a new adaptive sync deferment method, that is comparable to the current state of the art in terms of network overhead, but is also able to minimize the file synchronization time up to 12X.

3.1 Introduction

As a tool for personal storage and file synchronization, cloud storage services like Dropbox, OneDrive or Google Drive have become part of our everyday life. In these systems, the process of synchronizing a file requires of several metadata and data transfers between the cloud servers and the end user devices. To reduce the network overhead, these services use a suite of tools like delta compression, chunking, etc. [48, 50].

One of these optimizations, and the subject of this chapter, is *sync deferment*. Sync deferment consists of batching updates to intentionally defer the sync process for some time. In this way, the client can artificially increase the amount of useful data per sync operation, and hence, diminish the network overhead [48]. This has proven to be very effective to cope with frequent file modifications [47]. The authors of [48], however, discovered that simple sync deferment based on static thresholds could be not helpful. To wit, they found that by using a *fixed* sync deferment time (not tunable by the user), the network overhead could be of several orders of magnitude larger than the amount of useful data in some situations. To address this issue, they proposed an *adaptive sync deferment* (ASD) technique that adjusts the sync deferment time based on the *inter-update time*.

As a part of our ongoing process of implementing an open-source cloud storage service based on [50], we carefully studied the ASD algorithm, and found that in some cases, it may render long synchronization delays. Such a finding spurred us to come up with a novel deferment algorithm that we contribute here. While it is comparable to ASD in terms of network overhead, its distinguishing feature is that it is also capable of *minimizing the synchronization delay*, up to 12X compared with ASD. This translates into a better user experience, and less conflicts. The reason is that we study another dimension beyond inter-update time: *the data size of updates*. Combining both metrics, we can operate with the data rate instead, and adjust the deferral time to the minimum necessary to presumably collect enough useful data, and thus deliver a low network overhead.

3.2 Rate-based sync deferment

Although it has been seen in [48] that ASD outperforms sync deferment algorithms based on static values (e.g., the number of uncommitted bytes), it only considers an EWMA estimator of the inter-update time as the deferment criteria. And hence, it neglects an important dimension: *the size of updates*. The TUE metric, however, measures the traffic overhead. Consequently, what should be key for a sync deferment scheme should be to accumulate a sufficiently large number of unsynced bytes such that TUE was close to 1, yet delivering a short synchornization delay. With time alone, it is very difficult to achieve this, as we demonstrate in the evaluation. It is also necessary to consider the count of deferred bytes. Put another way, when the number of deferred bytes guarantees a small TUE, it does not yet make sense to wait for a new update, but to trigger a sync operation with the cloud backend as soon as possible.

3.2.1 Trade-off between TUE and synchronization time

It is clear that a sync deferment algorithm able to hold good levels on both parameters is still missing. This task is not easy. There is a trade-off between both parameters: when one tries to minimize one dimension, the other can grow uncontrollably.

To better understand this, pretend now that the dimension to optimize is the synchronization delay. Clearly, this will benefit user experience, since any modification to a file will be quickly propagated to the unsynced devices. However, it will impose a huge overhead on the system, because the count of unsynced bytes will be typically small. On the other hand, suppose now that objective is to decrease the network overhead. This would require deferring updates in order to transmit more useful bytes per overhead byte. Depending on to what extent, however, the synchronization delay could become intolerably long, and for instance, preclude services such as collaborative file editing.

To give a sense of this trade-off, we investigated the impact of sync deferment delay on TUE in Ubuntu One (UB1), a real cloud storage service. Concretely, we randomly picked $10,000$ client sessions from the publicly available trace [30]. Then, for each client session, we recorded the resulting TUE obtained by varying the sync deferment threshold from 30 to 120 seconds. The resulting TUE values were averaged to produce Fig. 3.1.

Since the UB1 service shut down on July 2014, we could not use its desktop client to empirically measure the TUE. Instead, to approximate the resulting TUE as a function of the amount of deferred data, we performed a small measurement analysis of Dropbox similar to that in [48]. The idea was pretty simple. To approximate its real TUE, we measured the resulting TUE when adding files of different sizes to the sync folder. To avoid any bias, we used binary, *non-compressible* files, ranging from 1 B to 100 MB, so that the TUE can be simply approximated as the ratio between the monitored network traffic after every file addition and the file size. Actually, we got similar results to those listed in [47, 48], with a TUE of $\approx$ 37 for 1 KB files, and of $\approx$ 1.1 for 100 MB files. As a result of this measurement, we could return a TUE value for every potential size of deferred data by means of curve interpolation, and thus produce Fig. 3.1. We also used the interpolated TUE curve in our evaluation[1].

As can be seen in this figure, the longer the client waits to send the pending updates to the cloud, the shorter the TUE is. The important observation to be made is that deferring updates too much is not of much utility. Beyond a certain point, TUE does not reduce significantly. This suggests that *as soon as the size of the deferred updates yields a small TUE, it is better off to push them to the cloud.* Almost surely, waiting for the next update will bring no much benefit. ASD cannot exploit such a trade-off well. For this reason, we devised the RSD algorithm.

3.2.2 The RSD algorithm

Given a target network overhead $\overline{TUE}$, our major objective is to minimize the synchronization time. To this end, instead of adapting the sync deferment time according to the inter-update time as in ASD, we do so by turning attention onto the *update rate*, defined as $R = b/\Delta t$, where b is the total number of bytes accumulated from local updates over certain time interval Δt. By estimating the rate R, we can dynamically adjust the sync

[1] All the experiments in this letter were performed on a commodity machine: Intel Core i5-4440 3.10GHz CPU, 8 GB RAM, connected to a 1 GbE LAN.

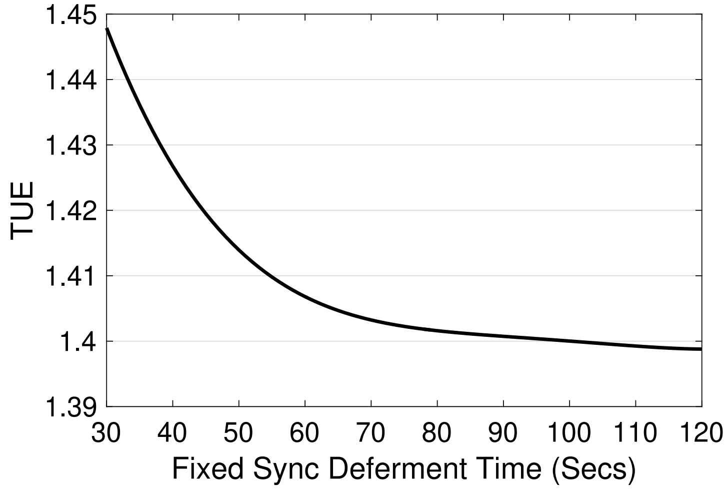


FIGURE 3.1: *TUE* as a function of sync deferment time in UB1.

deferment delay to the data generation rate of a user, so that it can be shortened if it is expected that the targeted $\overline{TUE}$ will be fulfilled soon according to the predicted rate. This is the reason why our algorithm is called *rate-based sync deferment* (RSD).

In practice, our scheme expresses the overhead objective as the number of unsynced bytes necessary to accumulate in order to yield an overhead equal or lower than $\overline{TUE}$. We denote this quantity as $B^{\overline{TUE}}$. Note that $B^{\overline{TUE}} = (\text{Total sync traffic})/\overline{TUE}$.

To estimate the rate, RSD utilizes an EWMA predictor. This means that upon the ith update, RSD estimates the current rate R_i as an average between the value of the last estimation R_{i-1}, and the current observation $b_i/\Delta t_i$ such that:

$$R_i = (1-\omega)R_{i-i} + \omega\frac{b_i}{\Delta t_i}, \tag{3.1}$$

where Δt_i is the inter-update time between the (i-1)th and the ith data updates, b_i is the size of the ith update in bytes, and ω is the weighting factor that shapes its memory. With the value of R_i at hand, then RSD calculates the sync deferment time T_i as
$T_i = \frac{B^{\overline{TUE}} - B^{\text{ACC}}}{R_i}$, i.e., as the number of bytes still needed to amass from future updates

$(B^{\overline{TUE}} - B^{\text{ACC}})$ divided by R_i, where B^{ACC} is a byte counter that tracks the size of all updates since the last sync operation with the cloud backend. After syncing the pending updates, B^{ACC} is always reset to 0.

As in ASD, a sync operation is started with the cloud servers when the next update falls outside the range t_i to $t_i + T_i$. Also, to ensure that updates do not remain unsynced for a long time, RSD uses a timer $\mathcal{T}$. When $\mathcal{T}$ expires, a new sync operation is triggered, irrespective of whether the target $\overline{TUE}$ is met or not. Note that this is different from limiting the maximum allowed value for each individual T_i as in ASD. Indeed, ASD does not ensure that the deferred updates are eventually applied. This is easy to see by simply inspecting (2.2). If inter-update times were always $< T_{max}$ (e.g., as a result of a frequent file modification), ASD could defer the updates forever if EWMA converged to a stationary value. In RSD, we addressed this issue from the start by using an independent timer, which is re-programmed upon the arrival of the first update after the last sync operation. The complete pseud code is listed in Algorithm 1.

Algorithm 1 RSD algorithm

```
upon $i$th update happens do
    if $B^{\text{ACC}} = 0$ then
        set timer $\mathcal{T}$ to $T_{max}$ seconds
    $\Delta t_i := t_i - t_{i-1}$                              ▷ compute the inter-update time
    $R_i := (1-\omega)R_{i-i} + \omega \frac{b_i}{\Delta t_i}$     ▷ update the EWMA estimator of the rate
    $B^{\text{ACC}} := B^{\text{ACC}} + b_i$
    if $T_{i-1} < \Delta t_i$ then
        push the deferred $B^{\text{ACC}}$ bytes to the cloud; $B^{\text{ACC}} := 0$
    else
        $T_i := \frac{B^{\overline{TUE}} - B^{\text{ACC}}}{R_i}$
upon timer $\mathcal{T}$ expires do
    push the deferred $B^{\text{ACC}}$ bytes to the cloud; $B^{\text{ACC}} := 0$
```

3.2.3 Analytical Comparison: RSD vs. ASD

Here we compare analytically the performance of RSD with ASD to better understand the benefits of sync deferment based on the data generation rate. For this purpose, we will adopt the "X KB/X sec" pattern that was originally posited in [48, 45] to validate

ASD. As in [45], we will set the weight factor $\omega = 1/2$ to simplify our discussion. From the analysis in [45], it follows easily the following corollary:

Corollary 1 *Under the"X KB/X sec" pattern,* ASD *triggers at most* $k = \lg X + 1$ *data transfers to the cloud.*

Indeed, it is easy to see that these data transfers occur at the first k updates, i.e., while the value of T_i converges to X. From that point onwards, (2.2) guarantees that $T_i > X$, for $i > k$[2], and hence, subsequent file updates may stay in pending state at the desktop client for a long time, even forever, yielding very long synchronization times.

With RSD, however, the value of T_i decreases progressively towards 0, to ensure that when the size of the deferred updates reaches $B^{\overline{TUE}}$ bytes, a sync operation with the cloud backend is always started. This minimizes the synchronization time yet delivering a small TUE:

Theorem 1 *Under the"X KB/X sec" pattern,* RSD *delivers the minimum synchronization delay to satisfy* $\overline{TUE}$.

Corollary 2 *Let* $B^{\overline{TUE}}$ *denote the number of bytes necessary to accumulate to meet the objective* $\overline{TUE}$. *To verify that* RSD *minimizes the synchronization time, we must prove that there is no data transfer to the cloud until the target* $\overline{TUE}$ *is fulfilled. First, we note that the observed rate* $r_i = \frac{X}{X} = 1$ *for all updates. Similar to [45], note that we assume* $B^{\overline{TUE}} < T_{max}$. *Otherwise, a sync operation would be triggered without reaching the target* $\overline{TUE}$. *Then, according to (3.1), we will have the series* $R_k = \left(1 - \frac{1}{2^k}\right)$, *which gives the corresponding time window series:*

$$T_k = \frac{B^{\overline{TUE}} - kX}{R_k} = \left(B^{\overline{TUE}} - kX\right)\left(\frac{2^k}{2^k - 1}\right). \tag{3.2}$$

Recall that a sync operation is triggered only if $T_k < X$ *(where X is the inter-update time). From (3.2), it is easy to see that this inequality does not hold for* $k \leq \frac{B^{\overline{TUE}}}{X} - 1$. *Hence, a new sync operation with the cloud will not be started until at least* $B^{\overline{TUE}}$ *bytes are accumulated from the deferred updates, thus meeting the target* $\overline{TUE}$ *while yielding the minimum sync delay.*

[2] Under the "X KB/X sec" pattern, the inter-update time is $\Delta t_i = X$ sec $\forall i$.

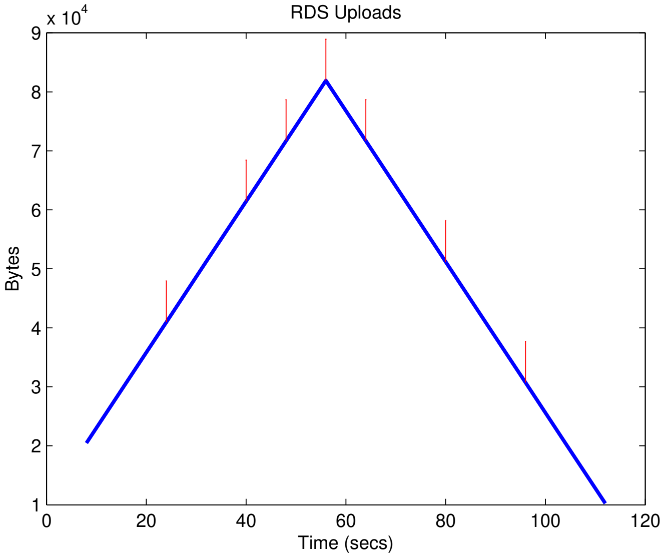


FIGURE 3.2: RSD under a regular triangular pattern.

To sum up, under a regular pattern, RSD is able to minimize the synchronization time, triggering a new sync operation with the cloud only when the overhead is optimal. In contrast, ASD triggers a logarithmic number of sync operations until reaching a stable state, and then, it defers updates forever until there is a significant change in the inter-update time. This behavior is easy to see in the triangular pattern shown in Fig. 3.2, where the bitsize of updates increases gradually up to 80KBs to decrease with the same speed. While RSD triggers a new sync operation as soon as $B^{\overline{TUE}}$ bytes has been buffered, spreading them over time, ASD only triggers a final sync operation after adjustment of the deferral time to the regular inter-update time, remaining out-of-sync for 110 seconds.

TABLE 3.1: Details of Workloads.

		Inter-update time (secs)		Update size (MB)	
Workload	No. Updates	90%-tile	Skewness	Median	CV
1	1,653	16	3.24	2.74	1.41
2	5,133	1	30.12	0.03	6.19
3	3,463	7	18.2	0.40	4.53
4	736	61	1.02	0.81	0.28
5	505	49	3.13	5.54	0.67
6	441	234.6	1.96	5.01	0.67
7	730	125.1	−0.11	0.01	0.51

3.3 Experimental Comparison

In practice, however, data update patterns are not so regular as the "X KB/X sec" pattern. For this reason, we compared the performance of both algorithms using real sessions from UB1 users [30]. From this trace[3], and after some pre-processing, we extracted 7 random workloads[4] corresponding to 7 different users. For clarity, we numbered them from 1 to 7. Workloads 1 – 5 corresponded to active users with inter-updates times of a few seconds. Their purpose was to evaluate both algorithms in the face of frequent of file modifications, which is the major driving force for sync deferment techniques. Workloads 6 and 7 corresponded to "warm" users, and concretely, to users who often made changes to their sync folder with a frequency that exceeded 120 sec. The goal of the last two workloads was to evaluate the performance of RSD when the timer $\mathcal{T}$ expires. More details about workloads can be found in Table 3.1.

Metrics. Besides TUE, we compared both algorithms in terms of synchronization delay, which is the aspect that distinguishes RSD from ASD. To this aim, we utilized the slowdown ratio SR defined as $SR = \frac{Time_{\text{ASD}}}{Time_{\text{RSD}}}$, where $Time_{\text{ASD}}$ and $Time_{\text{RSD}}$ are the average lengths of deferment periods delivered by ASD and RSD, respectively. Note that $SR = 1$ if both algorithms perform identically. A value of $SR > 1$ quantifies how many times the synchronization delay is larger in ASD compared with RSD.

[3] The UB1 log trace contains the timestamp and the size of every update, among other information.

[4] The workloads are available upon request.

Setup. The experimental setup was as follows. In both algorithms, we set the weighting factor ω to 1/2 in order to strike a perfect balance between memory and agility. T_{max} was set to 120 sec in both algorithms. For RSD, this meant that the timer $\mathcal{T}$ was reset to "120" sec at the beginning of each deferment period. As RSD depends also on the targeted $\overline{TUE}$, we ran it for 3 different values of $\overline{TUE}$: 1.2, 1.3 and 1.4, respectively. This gave us a sense of the sensitivity of RSD to $\overline{TUE}$.

In the experiment, we replayed the sequence of updates in each workload. During each replay, we recorded two measures on a per-deferment period basis: the length and resulting TUE of each sync deferment interval. For each workload, the results of both metrics were finally averaged to produce Fig. 3.3-3.4.

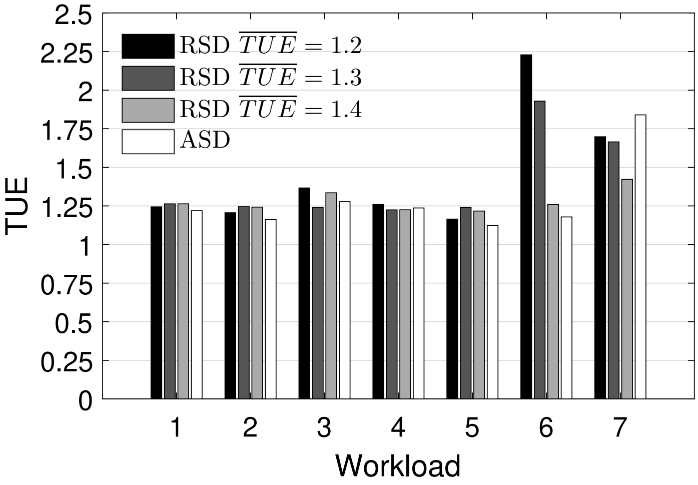


FIGURE 3.3: Empirical TUE of ASD and RSD for different target $\overline{TUE}$ values.

Results. As shown in Fig. 3.3, the TUE values of both ASD and RSD are very similar in the workloads 1 – 5. These workloads are very intense, and show that both mechanisms are equally effective in the reduction of the network overhead in the face of frequent file modifications. For these workloads, RSD does not appear to be sensitive to the prespecified $\overline{TUE}$. This is a "good news", since it reduces the amount of potential "fine tuning" decisions to be made in order to optimize RSD's performance.

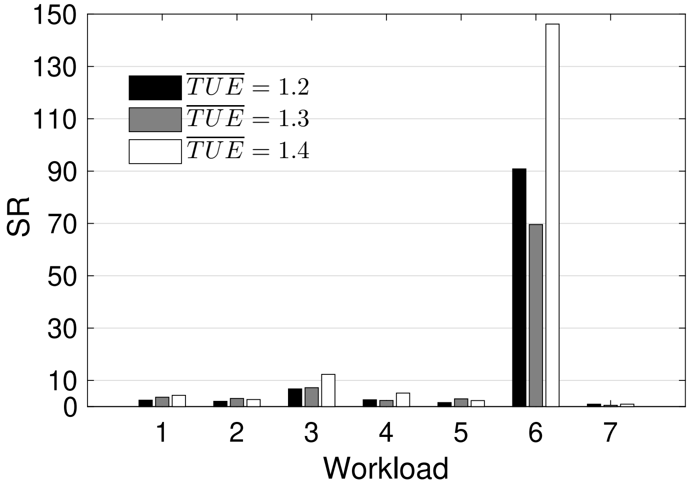


FIGURE 3.4: Slowdown ratio (*SR*) of ASD relative to RSD for different $\overline{TUE}$s.

For workload 6, RSD is only comparable to ASD for the most loosen $\overline{TUE}$ value. The reason is that inter-update times in this workload often exceeded the 120 sec, and RSD ended up issuing a new sync operation with the cloud servers without attaining the targeted $\overline{TUE}$ in many occasions. However, such a behavior is desirable as confirmed in Fig. 3.4. Thanks to timer $\mathcal{T}$, RSD yielded synchronization times between 70X to 150X shorter than ASD, which incurred intolerably long sync delays.

For workload 7, both algorithms reported bad TUE values. The reason behind this is that times between updates were very variable, alternating between long and short inter-update times, which caused a slow response of the EWMA controller in both algorithms. Even in this pessimistic setting, RSD was capable of decreasing a little bit the sync time as can be seen in Fig. 3.4.

For workloads 1 – 5, RSD improved sync delays between 2X to 12X relative to ASD with equivalent TUE values, which undeniably demonstrates the superior performance of RSD.

3.4 Conclusion

Cloud storage services like Dropbox and Google Drive are becoming very popular these days. To optimize network traffic, these storage services rely on techniques like sync deferment. Literature so far has proven this technique to very useful in the face of frequent file modifications. However, there still exist some performance weaknesses on current implementations. To cope with them, this letter presents an innovative adaptive sync deferment algorithm, which is comparable to the current state of the art in terms of overhead, but as a distinguishing feature, it also optimizes file synchronization delays. Our experimental results report improvements between 2X to 12X in sync delay.

Chapter 4

Software-Defined Data Protection for Object Storage

With the growth in popularity of cloud computing, object storage systems (e.g., Amazon S3, OpenStack Swift, Ceph) have gained momentum for their relatively low per-GB costs and high availability. However, as increasingly more sensitive data is being accrued, the need to natively integrate privacy controls into the storage is growing in relevance. Today, due to the poor object storage interface, privacy controls are enforced by data curators with full access to data in the clear. This motivates the need for a new approach to data privacy that can provide strong assurance and control to data owners. To fulfill this need, this paper presents EGEON, a novel software-defined data protection framework for object storage. EGEON enables users to declaratively set privacy policies on how their data can be shared. In the privacy policies, the users can build complex data protection services through the composition of data transformations, which are invoked inline by EGEON upon a read request. As a result, data owners can trivially display multiple views from the same data piece, and modify these views by only updating the policies. And all without restructuring the internals of the underlying object storage system. The EGEON prototype has been built atop OpenStack Swift. Evaluation results shows promise in developing data protection services with little overhead directly into the object store. Further, depending on the amount of data filtered out in the transformed views, end-to-end latency can be low due to the savings in network communication.

4.1 Introduction

With the rapid growth in popularity of Cloud services, object storage systems (e.g., Amazon S3 [3], IBM COS [36] or OpenStack Swift [64]) have gained momentum. These storage systems offer consolidated storage at scale, with high degrees of availability and bandwidth at low cost. Proof of that is the recent trend of serverless computing. Due to the high difficulty of function-to-function communication[1], many serverless systems use object storage for passing data between functions [85, 74, 59, 86, 54], which has revived the interest in this type of object storage.

Although very useful for cloud applications, object storage systems offer a small number of options to keep sensitive data safe. Few (or no) efforts have been realized on security issues such as data confidentiality, data integrity, or access control, to mention a few. For instance, online storage services such as Amazon S3, or IBM COS, only provide server-side encryption for protecting objects at rest [3, 36]. Similar words can be said for the access control of individual objects[2], which is currently either realized via simple object ACLs (Access Control Lists) as in S3 [3], or not possible at all as in OpenStack Swift [14].

This poor interface is insufficient for many applications. For instance, it does not enable in-place queries on encrypted data, transparent and secure data sharing, and access control based on the content of an object. But also, it is very difficult to make it

[1] Some works have shown that cloud functions can communicate directly using NAT (Network Address Translation) traversal techniques. However, direct communication between functions is not supported by cloud providers.

[2] In general terms, cloud object stores enforce access at the container level rather than at the object level.

evolve to meet the changing needs of applications and withstand the test of time. In practice, most of these object storage systems leave no other choice to modifying the system internals to incorporate new security mechanisms at the object level. This requires a deep knowledge of the system, extreme care when modifying critical software that took years of code-hardening to trust, and significant cost and time (see, for instance, [46], where it is described the "daunting" task of deploying new erasure coding solutions in object stores such as OpenStack Swift [64] and Ceph [107]).

Rather than relying on object storage systems to change, we advocate in this paper to "work around" the traditionally rigid object storage APIs by embracing a *software-defined* approach. Similarly to software-defined networking (SDN), we argue that the separation of the "control logic" from the "data protection logic" can give the needed flexibility and ease of use to enable users, programmers and sysadmins to custom-fit access control and object protection. To give an example, pretend that upon certain conditions, parts of an encrypted object need to be re-encrypted to share it with the mobile users of the application. Further, these conditions may depend on the contents of the object itself (e.g., on sensitive data such as sexual orientation), which must always remain confidential from the server. Simply put, what we pursue is to offer users the ability to succinctly express this behavior at the object storage level, and enforce it by calling the corresponding re-encryption modules.

Nevertheless, a software-defined solution to enhance object storage data protection requires solving several issues at two levels:

- At the control plane, by enabling the composition of per-object protection services. These compositions should be expressed concisely and in a manner agnostic to the data protection code and the remaining storage stack.
- At the data plane, by making it truly programmable. To put it baldly, the data plane should not only allow to plug-in new protection logic in the critical I/O path, but to run it safely. In addition, it should enable the re-usability of the protection capabilities, so that users can compose new data protection controls.

In this research work, we present EGEON, a novel software-defined data protection framework for object storage. EGEON exports a scripting API to define privacy policies for protecting objects. These policies enable data owners to declaratively set complex data protection services through the composition of user-defined transformations run in a serverless fashion. These functions represent the elementary processing units in EGEON, and may be re-used and linked together to implement complex privacy policies. The major feature of these transformations is that they are executed "inline" by EGEON upon a standard `Get` request. In this way, users can trivially display multiple views from the same object, and modify these views by updating the policies. If some functionality to implement a view is missing, EGEON provides a simple API to deploy new transformations and customize the access to data objects.

In this sense, one of the primary contributions of EGEON is the ability to provide privacy-compliant transformed views of the underlying data on the fly. Perception of privacy can vary broadly across applications. As an example, a dataset created by a hospital may include personally identifiable information (PII) that is not needed when it is processed by a data analytics engine. Nonetheless, if the same dataset is accessed by medical personnel, a richer view of it should be given. Thus, a practical system needs to support a range of privacy preferences.

By adopting a software-defined storage architecture, EGEON allows users to express their privacy preferences as policies in the control plane and produce views conforming to the policies by running transformations in the data plane. In this work, we focus on (cryptographic) transformations that process data as streams, that is, as data is being retrieved from object storage nodes. Consequently, the first bytes of transformed views are received as soon as possible, which permits EGEON to scale to arbitrary object sizes without important penalty on end-to-end latency.

The EGEON prototype we present in this research has been implemented atop OpenStack Swift [64]. We took Swift because it is open source and a production quality system. Its sizable developer community ensures that our new properties are built on code that is robust and that will be soon evaluated. It must be noticed that the design concepts underpinning EGEON are generic and could be easily ported to other storage substrates.

For example, object classes allow to extend Ceph by loading custom code into Object Storage Daemons (OSDs), which can be run from within a `librados` application [57]. Thus, with some effort, it would be possible to leverage object classes to implement transformed views of data.

Our performance evaluation of EGEON shows promise in developing data protection services with low overhead directly into the object storage. For instance, the overhead of a `NOOP` policy, where a storage function simply echoes the input data, is of around 9 ms. Also, depending on the amount of protected data filtered out in the transformed views, end-to-end latency can be even lower with EGEON due to the savings in network communication (up to 72.1x for a 4G mobile use case).

4.2 Related Work

Software-defined storage systems. A first category of related work comprises software-defined approaches for storage, and in particular, for object storage systems. The common feature of these approaches is that they break the vertical alignment of conventional storage infrastructures by reorganizing the I/O stack to decouple the control and data flows into two planes of functionality—control and data. A number of proposals have followed this approach, including IOFlow [98], sRoutes [94], Retro [52], Vertigo [83] and Crystal [29, 32].

Among them, only Vertigo and Crystal have been tailored to object storage. As EGEON, both systems have been deployed atop OpenStack Swift. But unlike EGEON, their data plane is based on OpenStack Storlets [65]. A Storlet is a piece of Java logic that is injected into the data plane to run custom storage services over incoming I/O requests. As in EGEON, this design increases the modularity and programmability of the data plane stages, fostering reutilization. However, Storlet-made pipelines are not reactive, wasting resources when we are only interested in specific elements of the data stream. Moreover, the Storlet-enabled data plane of Vertigo and Crystal emphasizes control-flow over data-flow, making it hard to explicitly represent the (cryptograhic) transformations of objects. Per contra, EGEON's data plane is driven solely by the events showing up in

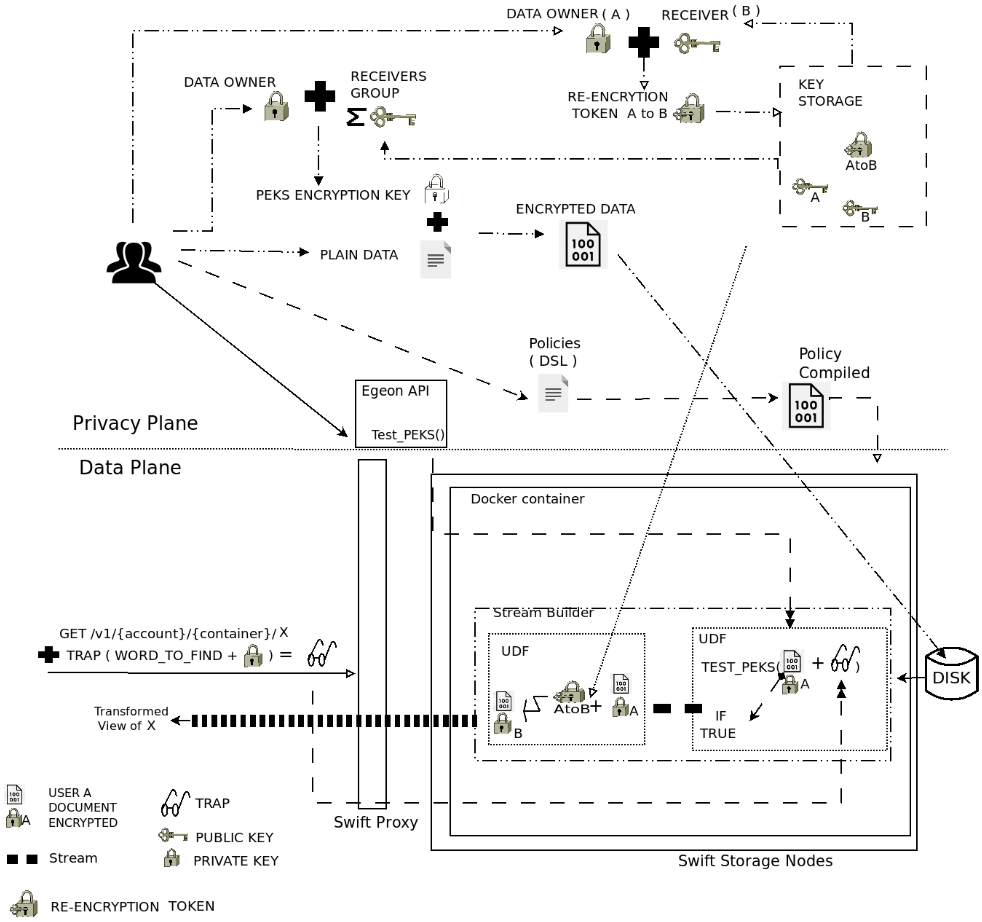


FIGURE 4.1: EGEON's software-defined architecture. The system is divided into two planes: the *privacy plane*, specialized for data protection, which offers an API to allow data owners to manage life-cycle of their data protection policies, and the *data plane* responsible for generating the privacy-compliant data views delivered to final user.

the data streams, which makes it easy to reuse the same transformations over and over again on different data. Only the events must be re-defined in the policies.

Finally, it is worth to note that to the best of our knowledge, we are not aware of another software-defined storage system that automatically enforces privacy policies along the I/O path as EGEON. Software-defined security has remained within the boundaries of software-defined networking (see, for instance, Fresco [91]). The only exception is the recent vision paper [38] on software-defined data protection. Like EGEON, [38] argues that the key ideas of software-defined storage can be translated to the data protection domain. However, the approach of [38] is radically different. Instead of adding privacy controls to the I/O stack of a disk-based object storage system, [38] assumes all in-storage processing to occur on FPGAs and "smart storage" devices, for we see [38] as an orthogonal work to us.

Privacy Policy Enforcement. There exist many systems that enforce privacy policies automatically. Most of these systems resort to Information Flow Control (IFC) as a means to control how information flows through the system. See, for instance, Riverbed [104], which uses IFC to enforce user policies on how a web service should release sensitive user data. In contrast to these systems, EGEON follows a software-defined approach to leverage the storage resources and enforce the privacy policies where data is. Similar to our transformation chains, Zeph [17] proposes to enforce privacy controls cryptographically but over encrypted stream processing pipelines. Specifically for storage, Guardat [101], at the block level, and Pesos [41], at the object level, enable users to specify security policies, for instance, to stipulate that accesses to a file require a record be added to an append-only log file. Nonetheless, these systems do not permit the composition of advanced privacy controls as EGEON, and thus, fall short to empower users with strong data controllers.

4.3 Design

EGEON is a software-defined data protection framework that augments object storage with composable security services to enforce users' privacy preferences over protected data. These services are built up as pipelines of serverless functions. Fig. 4.1 illustrates an overview of EGEON's architecture. Our objective is to enable authorized users or applications to access protected data without violating the privacy policies of data owners. We have designed EGEON to make it easy the leverage of state-of-the-art privacy solutions (such as content-level access control, homomorphic encryption, encrypted keyword search, ...) while preserving the normal data flow in the consuming applications. Concretely, we achieve this by introducing a logical separation between the *privacy plane*, where data owners set their privacy preferences, and the *data plane*, where the creation of privacy-compliant transformed views happens. This separation allows for heterogeneous policies atop the same data without having to modify the system internals to enforce advanced policies at the object level.

EGEON's architecture consists of the following components:

Privacy Plane. The privacy plane in EGEON corresponds to the control plane of a software-defined architecture [29, 32], but specialized for data protection. In practice, this means that EGEON provides its own script language to assist data owner in composing data protection services from elementary serverless functions in the *data plane*. The textual, JSON-based language supports conditions and compositions to build up inline privacy transformers (e.g.,see Listing 4.5). Moreover, EGEON offers an API to allow data owners to manage the life-cycle of their data protection policies.For performance reasons, once uploaded, the policies are automatically compiled into Java bytecode and stored in the *Metadata Service*. For fast access, this service has been built on top of Redis [76], an in-memory, low-latency key value store.

Data Plane. In EGEON, the data plane has a critical role. The data plane is responsible for generating the privacy-compliant data views. And hence, it must be extensible to accommodate new functionality that enables privacy transformations on data. Particularly, in this realization of EGEON, we have focused on inline privacy transformations as data is retrieved from storage nodes HDDs. As mandated by the policies in the privacy plane, privacy transformations are constructed from pipelines of user-defined functions executed as serverless functions by EGEON. Namely, a user integrating a new transformation only needs to contribute the logic. Resource allocation and execution of the chain of transformations is automatically handled by EGEON, bringing a true serverless experience to users.

To minimize execution overhead, since many cryptographic operations are CPU-intensive, EGEON abides by the principles of reactive programming and runs a transformation only when is strictly needed, instead of continuously on the data streams. More concretely, EGEON extends the observer pattern [26] and execute a certain transformation in the pipeline when an event occurs [9]. Consequently, EGEON better utilizes the available resources in the storage servers by balancing the load across the chain of transformations. To better understand this, pretend that a user wants to compute the average salary of employees in a department `X`. Now suppose that all the employee records have been saved in a single JSON document with all the salary values homomorphically encrypted. Thanks to EGEON reactive core, the transformation to average the salary will only be run when the event "`employee of department X`" comes through the data stream, thereby saving CPU resources.

4.3.1 Threat Model

Specifically, EGEON enforces user's privacy preferences via function composition. That is, users are ensured that their data is transformed as it goes through a pipeline of transformation functions before it is released to applications. In the meantime, the original data remains end-to-end encrypted.

We assume an *honest-but-curious* [71] storage servers, i.e., the server performs the computations correctly but will analyze all observed data to learn as much information as possible. We also presume the existence of an identity service (IDS) such as OpenStack Keystone for user authentication. We leverage this service for authentication of the storage functions. An IDS is a standard requirement in multi-user systems and can even be a trustworthy external entity.

Consumers of shared data are semi-trusted, in that they do not collude with the servers to leak the data or keys. This is a reasonable assumption for groups of data consumers that are acquainted with each other. Further, EGEON assumes that the applications behave correctly and do not hand out user keys to malicious parties. Finally, we assume state-of-the-art security mechanisms to be in place for user devices, and that all parties communicate over secure channels.

In this setting, EGEON enforces *data confidentiality*, making sure that the adversary learns nothing about the data streams, except what can be learned from the transformed views.

Robustness. While EGEON is able to handle various types of failures in practice, provable robustness against misconfigured, or even malicious privacy policies, and data producers is out of scope for this research. A malicious user sending corrupted tokens cannot compromise privacy but could alter the output of a transformed view.

4.3.2 Privacy Plane

In the privacy plane, EGEON provides the capabilities for data owner to set their privacy preferences —i.e., user-centric privacy—, and what transformations will be required to apply to enforce a privacy policy. These transformations are specified by their unique name, and their existence is verified when the privacy policy is to be compiled. A privacy policy applies to a single object. In this paper, we do not consider the question of how to set privacy policies for group of objects and how they should look like. This question has been left for future work.

In EGEON, targeted data objects in the privacy policies are specified by its full resource path. Following OpenStack Swift specs [64], the access path to an object is structured into three parts: `/account/container/object`. As an example, for the `rose.jpg` object in the `images` container in the 1234 account, the resource path is: `/1234/images/rose.jpg`.

Data owners can translate their preferences over an object to a set of transformations by mapping them in a JSON-based schema language. In addition to some meta-information (e.g., policy identifier), this schema permits data owners to formalize conditions at the policy level using a rich set of operators such as “`StringLike`", “`NumericLessThan`", etc. Importantly, the language enables data owners to build date expressions using operators like “`DateNotEquals`", which makes it possible to express temporal restrictions. For instance, pretend that a data owner wishes to prevent that a document can be accessed on weekends. She could indicate this through the date expression:
“`DateNotEquals`" : {“`Day`" : [“`Sat`", “`Sun`"]}.

More interestingly, this schema allows composing complex data protection transformations from elementary UDFs (User-defined Functions). This can be achieved by adding each individual UDF as a step in the transformation pipeline defined in the JSON object “`Action`". This object contains two name-value pairs: “`StartAt`", which indicates the first transformation in the pipeline, and “`Steps`", which is another JSON object that specifies the transformation UDFs along with their input parameters. Since transformations run only when the corresponding events come through the data stream, this schema allows data owners to specify the observed events for each transformation to execute. To do so, there exists a field named “`EventType`" to signal the event to be observed by a particular transformation UDF. A typical “`EventType`" block looks like one in Listing 4.1.

```
{
  "Type": "<type_of_event>",
  "Input": [<parameter_block>, ...],
}
```

LISTING 4.1: EventType example.

where the “`Type`" field specifies the type of an event (e.g., an XPath [103] event) and the array “`Input`" lists the parameters that are required for this type of event. For instance, if a data owner wanted to apply a transformation UDF over all `salary` elements of an XML document, she could do so by setting an XPath event as as shown in Listing 4.2.

```
{
  "Type": "XPathEvent",
  "Input": [{"Predicate": "//salary"}],
}
```

LISTING 4.2: XPath Event example.

Similarly, a transformation UDF block is shown on Listing 4.3.

```
{
  "Id": "<identifier_of_UDF>",
  "EventType": "<event_block>",
  "Input": [<parameter_block>, ...],
  "Next": (<identifier_of_UDF>|"End")
}
```

LISTING 4.3: UDF Block example.

The “Id" field is a string which uniquely identifies the UDF, while the “Input" field permits to specify the parameters for the transformation UDF (e.g., the targeted security level of a cryptosystem). Finally, the “Next" field indicates the next step to follow in the pipeline, or “End" to indicate the end of the chain of transformations.

An example of a real policy can be found in Listing 4.5. This policy provides transformed views over a JSON file containing employee records of the format shown on Listing 4.4.

```
{
  "employee": {
  "name": "Alice",
  "identification": {
    "SSN": "32456677"
  },
  "salary": 50000
  }
  ...
}
```

LISTING 4.4: JSON record example.

As a first transformation in the chain, this policy uses content-level access control [14] (CLAC). Very succinctly, CLAC works as follows. It assigns targeted JSON elements an object label (`olabel`) and each user a user-label (`ulabel`). Then, it allows to define rules in the form of (`ulabel`, `olabel`), which means that the JSON elements labeled with any of the `olabels` are allowed to be read by the users labeled with the corresponding `ulabels`. To indicate what JSON items to protect, CLAC uses JSONPath in this case.

In this policy, we assume two types of users: treasurers with user label “`treasurer`" and regular users with label “`user`". We protect the salaries with the object label “`sensitive`" and specify the single rule
(“`treasurer`",“`sensitive`"), which means that only treasurers will have access to the salaries. The field “`salary`" is identified using the JSONPath expression: ‘`$.employee.salary`’ (Listing 4.5,line 14).

The second transformation applies homomorphic encryption on the field “`salary`" to prevent the servers from learning the employee salaries [90], while computing the average salary of employees (Listing 4.5, lines 20-30).

As a final transformation in the chain, the policy uses proxy re-encryption [90] to convert the homomorphically encrypted average salary to a ciphertext under the receiver's key. Thus, the transformed view can be decrypted by the receiver, without the data owner having to share her private key nor performing any encryption for the receiver on her personal device.

To wrap up, this policy will generate two transformed views of the same data. For regular users, it will only be executed the first step (Listing 4.5, lines 10-19): the CLAC transformation. Due to lack of permissions, the CLAC module will eliminate the encrypted “`salary`" field, and output a transformed view with the rest of information. For a treasurer, it will output the same view as any regular user (without individual salaries), but enriched with the average salary encrypted under her public key thanks to proxy re-encryption (Listing 4.5, lines 31-37).

We want to note that the different transformations UDF only execute when the corresponding JSONPath events come along. To wit, proxy re-encryption will only be run one time, after the JSON field named "`average_salary`" is added at the end of the response by the second transformation in the chain.

As a final word, this example clearly displays how EGEON is capable of performing real-time privacy transformations atop the same data object for a variety of application scenarios, thus enhancing the rigid interface of object storage systems.

4.3.3 Data Plane

The focus of EGEON is on inline privacy transformations, where data streams are "observables" and user-defined privacy transformations are "observers" subscribed to the data streams. As as soon as an event is observed, it will be delivered to the subscribed observers. If there are no events on the data stream then the original data stream is pushed back to the user. In this sense, a privacy transformation is nothing but a function taking an observable as input and returning another observable as its output. This design has the advantage that transformations can be chained together to generate complex data views compliant with the policies in the privacy plane.

Runtime. Currently, EGEON's runtime is Java-based. Thus, the chain of transformations is run within a Java Virtual Machine (JVM) wrapped within a Docker container to guarantee a high level of isolation between two different data transformations.

Policy Enforcement. Upon a new `Get` request, EGEON starts up a thread inside the JVM to perform three tasks. We refer to this thread as the "master thread". The three tasks in order of execution are:

1. *Policy loading*, where the master thread loads the policy into memory and evaluates its conditions clauses.

2. *Observable setting*, where the master thread opens a data stream to the target object, namely, the observable, if the policy conditions are fulfilled. To this aim, it creates an instance of the appropriate subclass of the abstract class `StreamBuilder` that the EGEON's engine leverages to start parsing the object. Subclasses are required since the specific logic to parse the data stream and generate the events depends on the type of file. Specifically, EGEON picks up the proper `StreamBuilder` subclass

```
{
 "Id": "employee.policy",
 "Object": "v1/{account}/{container}/employees.json",
 "Condition": {
    "DateNotEquals": { "Day": ["Sat","Sun"] }
 },
 "Action": {
  "StartAt": "Step1",
  "Steps": {
    "Step1": {
     "Id": "CLAC",
     "EventType": {
        "Type":  "JSONPathMarkerEvent",
        "Input": [{"Predicate":"$.employee.salary", "olabel":
    "sensitive" }]
     },
     "Input": [{"ulabel": "treasurer",
                "olabel": "sensitive" }],
     "Next": "Step2"
    },
    "Step2": {
     "Id": "SUM",
     "EventType": {
        "Type":  "JSONPathEvent",
        "Input": [{"Predicate":"$.employee.salary"}]
     },
     "Input": [{"average": true},
         {"keyOwner": "meta://Alice/keys/hom",
         {"output":"$.average_salary"}],
     "Next": "Step3"
    },
    "Step3": {
     "Id": "PRE",
     "EventType": {
        "Type":  "JSONPathEvent",
        "Input": [{"Predicate":"$.average_salary"}]
     },
     "Next": "End"
}}}
```

LISTING 4.5: A sample policy to process employee records.

based on the object extension (e.g., “.json" for JSON documents) using *factory* methods.

3. *Transformations setting*, where the master thread makes an instance of each transformation UDF in the pipeline and chains them together. The StreamBuilder subclass generates events as the data is being parsed, and notifies the subscribed transformation UDFs in the same order as dictated in the privacy policy. To wit, in Listing 4.5, upon the JSONPath event ‘$.employee.salary’, EGEON will execute first the CLAC transformation, followed by the SUM transformation.

In EGEON, we assume that an observable can only handle one event at a time. We adopted this design to minimize compute resources at the storage layer, so that both the data stream and all its transformations operate in the same thread, in our case, the master thread. Nevertheless, this can be easily changed by switching to a different thread and integrating some additional logic for coordinating the threads.

Since each transformation UDF acts as an observer, another responsibility of the master thread is to subscribe each privacy transformation to the events specified in the policy. To this aim, it invokes the method
install(Event event,UDF observer) in the StreamBuilder class, where the parameter observer is the transformation UDF bound to the event.

Extensibility. At the time of this writing, EGEON implements three types of data sources: XML, JSON and CSV documents, a number of events including XPath and JsonPath expressions, CSV field and records, etc., and multiple transformation UDFs (see §4.3.4 for further details). However, EGEON is extensible, and new events, observables and observers can be incorporated by extending the abstract classes Event, StreamBuilder and UDF, respectively.

We show here an example of a transformation UDF to perform summations on ciphertexts [90] to see how easy it is to code a transformation UDF (Listing 4.6).

```
package com.urv.egeon.function;

import com.urv.egeon.runtime.api.crypto.Homomorphic;
import com.urv.egeon.runtime.api.parser.UDF.ContextUDF;
import com.urv.egeon.runtime.api.parser.UDF.UDF;
import com.urv.egeon.runtime.api.parser.event.Event;

public class Sum extends UDF {
    private Homomorphic accum = new Homomorphic();

    @Override
    public Event update(Event e, ContextUDF ctx) {
       try {
          this.accum.add((String) e.getValue());
       } catch (Exception ex) {
          // first execution of the UDF
          this.accum.setKeys((String)
   ctx.getParameter("keyOwner"));
          this.accum.setCipher(this.accum.fromSerial((String)
   e.getValue()));
       } finally {
          return e;
       }
    }
    @Override
    public Object complete(ContextUDF ctx) { ... }
}
```

LISTING 4.6: A transformation UDF to perform summations on ciphertexts.

As shown in Listing 4.6, a UDF has two methods: the method `update`, which is invoked every time a new subscribed event is emitted, and the method `complete`, which is called when the data stream is finalized (empty). The `update` method has two arguments of type `Event` and `ContextUDF`. The first argument encapsulates the details of the emitted event. For instance, for the homomorphic summation UDF of Listing 4.6, this may mean a JSONPath event, alongside the value of the selected field by the JSONPath expression (e.g., an encrypted salary value), and accessible via the method `getValue` (Listing 4.6, line 14). The `ContextUDF` encapsulates the access to the request metadata, such as HTTP headers and cryptographic keys and tokens sent out by the client, the object's metadata, and specific parameters required for the UDF to work, such as the data owner's public key to operate on the encrypted values (Listing 4.6, line 17). All this information is automatically made available by EGEON to the transformation UDF. This includes the input parameters set in the privacy policy (e.g., Listing 4.5,, lines 26-28).

It is worth to mention here that the input arguments whose values have the format of "`meta : //key`" are automatically downloaded from the Metadata Service using the key "`key`". In this way, a data owner can change the input parameters (e.g., her homomorphic public key) without having to re-compile the policy. An example of this can be found in Listing 4.5, line 27, for the argument "`keyOwner`", whose value is retrieved from the Metadata Service behind the scenes, and made accessible to the summation UDF via the context (Listing 4.6, line 17).

The purpose of the method `complete` is to provide a hook for developers to perform final computations and append their result at the end of the data stream. For instance, this could be useful to compute the average over encrypted data and append the result as new item at the end of a JSON document.

4.3.4 Data Transformation UDFs

EGEON comes up with a library of reusable functions (UDFs) for inline data protection. While some of these functions apply cryptographic transformations, others act on raw data, e.g., by filtering out protected parts of a JSON object to non-privileged users [14]. All functions are composable to provide complex transformed data views. These are the following:

Homomorphic encryption (HOM). HOM is a cryptosystem (typically, IND-CPA secure) that allows the server to perform computations directly on encrypted data, the final result being decrypted by the user devices. For general operations, HOM is prohibitively slow. However, it is efficient for summation. To support summation, along with proxy re-encryption (PRE), we adopted the homomorphic cryptosystem of [90]. We chose this scheme, because it allows secure data sharing and is tailored to mobile platforms and constrained IoT devices. Concretely, we implemented two UDFs:

- Summation (`SUM`): This UDF supports the summation of ciphertexts, such that the result is equal to the addition of the plaintext values: $\texttt{Enc}(m_1) \cdot \texttt{Enc}(m_2) = \texttt{Enc}(m_1 + m_2)$.

- Re-Encryption (`PRE`). Succinctly, `PRE` allows the storage servers to convert ciphertexts under the data owner's key to ciphertexts under authorized users keys without leaking the plaintext. Therefore, the data owner can securely share data with other users, i.e., without sharing her private key nor performing any encryption for them on her personal device. To do so, the data owner d solely needs to issue a re-encryption token for a user u based on his public key pk_u as the $\texttt{Token}_{d \to u}$. With this token, the `PRE` UDF can automatically re-encrypt data on behalf of the data owner without her intervention. Further, the re-encryption tokens are unidirectional and non-transitive.

For both UDFs, we assume integers of $\leq$ 32 bits with 128-bits of security. The implementation makes use of the optimal Ate pairing [102] over Barreto-Naehrigopera elliptic curve [13], and also applies the Chine Remainder Theorem (CRT) to optimize decryption [90]. For all this cryptographic processing, we use the RELIC toolkit [6].

Keyword search (SEARCH) To allow keyword searches (as the SQL "`ILIKE`" keyword), we make use of a cryptographic scheme for keyword searches on encrypted text [35]. As above, we have chosen this scheme because it is multi-user. To put it baldly, before storing an object, the data owner first selects the users with whom she wants to share her data and then encrypts it with their public keys. To search for keywords in the shared object, a user makes a trapdoor $\texttt{Trap}_W$ for the keyword set W using his private key, and then sends it to the server. The server runs the `SEARCH` UDF, which takes the public key of the user, the trapdoor $\texttt{Trap}_W$ and the encrypted text, and returns "yes" if contains W

or "no" otherwise. As expected, this scheme is proved secure against chosen keyword attacks (IND-CKA).

One major advantage of this scheme is its short ciphertext size. Concretely, for n users, it requires $(n + \ell + 1) \cdot L1$ bits, where ℓ is the number of keywords and $L1$ is the bit length of the underlying finite field $\mathbb{F}_q$, which is much smaller than the deployment of n separate instances of a public key searchable encryption scheme, one for each user. For the implementation of this scheme, we use the "SS512" elliptic curve, a symmetric curve with a 512-bit base field, which provides a security level of 80-bits, from the jPBC library [20].

Content-level access control (CLAC). As an example of a non-cryptographic function, we decided to implement content-level access control [14]. The central idea of `CLAC` is to enforce access control at the content level to restrict who reads which part of a document. To give a concrete example, consider that a hospital stores its patient records as big JSON object. These records should be accessed differently by different personnel. For example, a "doctor" could see health information from her patients, while a "receptionist" should only view basic profile information about the patients. With `CLAC`, a data owner can define content-level policies to censor access to parts of a data object. Remember that in object storage systems such as Swift or Amazon S3, once an object is made accessible to someone, she retrieves the full content of the object. So, there is no way to hide out sensitive information to that user.

Our implementation of `CLAC` overcomes this limitation. As introduced in §4.3.2, `CLAC` borrows the LaBAC model (Label Based Access Control Model) [14]. Either be an XML element, a JSON element, or a CSV column, an object label (`olabel`) is assigned on the targeted item. Similarly, each authorized user is given a user label (`ulabel`). Then, the LaBAC model works by specifying tuples of rules in the form of (`ulabel`,`olabel`), which tell that only the users labeled with `ulabel` can access the items labeled with that `olabel`. For instance, if only users with the user label "`manager`" were authorized to access items labeled with "`restricted`", a data owner should have to set the rule ("`manager`", "`restricted`") to effectively formalize access control. As in the LaBAC model, our implementation admits hierarchies of both `ulabels` and `olabels` to rank users and objects.

One interesting side effect of our reactive data plane is that our `CLAC` implementation is generic, and not tied to a specific file type. What changes is the mechanism to identify the items, which depends on the object type. That is, for XML, an XPath expression should be used to indicate that a certain element has “`restricted`" access, while for a JSON document, a JSON predicate should be used in its stead. We have abstracted this coupling by the definition of specific marker events as shown in Listing 4.5,lines 22-25.

4.4 Implementation

We have constructed EGEON by extending Zion [84], a data-driven serverless computing middleware for object storage. In particular, we have implemented EGEON on top of OpenStack Swift [64], a highly-scalable object store.

OpenStack Swift is split into several components. The main components are the object and proxy servers. While the object servers are responsible for the storage and management of the objects, the proxy servers expose the RESTful Swift API (e.g., `GET /v1/account/container/object` to get object content) and stream objects to and from the clients upon request.

To be as non-intrusive as possible, the only modification we perform in the default Swift architecture is the deployment of a custom Swift middleware to intercept `GET`, or read, requests at the proxy servers [63]. This middleware also provides a simple API to manage the life-cycle of privacy policies. Essentially, it communicates with the Metadata Service to store and retrieve the privacy policies for protected data objects. Recall that the Metadata Service leverages Redis [76] to yield sub-millisecond access latency to metadata. To optimize request matching even further, we have collocated the Redis instance with the proxy server. As Zion, EGEON uses containers to sandbox the execution of the chain of privacy transformations. That is shown on Fig. 4.2. However, contrary to Zion, which is a general-purpose serverless platform, EGEON employs a single, optimized serverless function to produce the privacy-compliant views of the underlying data. This function deserializes the compiled privacy policy, loads it into

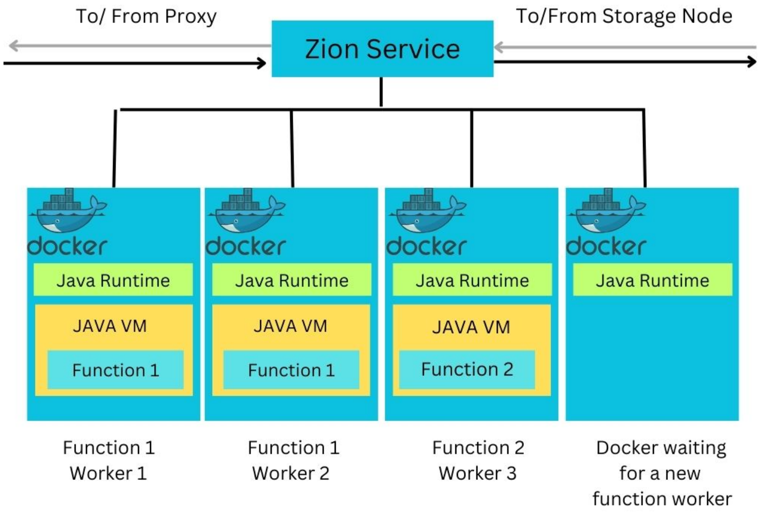


FIGURE 4.2: Docker, controller and Functions Structure.

memory, evaluates the conditions from the clauses, and if it “applies”, it runs the the pipeline of UDF transformations. This design has two main benefits. On the one hand, EGEON runtime starts up faster. On the other hand, UDF transformations enjoy of a great level of isolation. Simply put, they have neither direct network access nor access to the local Linux file system, among other namespaced resources, which protects the whole system from malicious transformations.

Also, to enhance response time for policies that are accessed frequently, EGEON runtime deploys a cache for policies using the Google Guava caching library [28] configured with a least-recently-used (LRU) eviction policy.

Resource allocation is managed by Zion. EGEON does not contribute any optimization at this level. When a read request comes along, our Swift middleware contacts the Zion service, which manages the containers in the object servers, and starts up a new one if necessary.

4.5 Evaluation

In this section, we evaluate key aspects of EGEON's design and our prototype implementation. First, we begin with a series of microbenchmarks to judge aspects such as system overhead, throughput and the performance of the cryptographic operators in isolation. Finally, we assess EGEON's flexibility to compose complex data views.

System setup. All the experiments have been conducted in a cluster of 8 machines: 2 Dell PowerEdge R320 machines with 12GB RAM, which operate as Swift Proxy servers, and 6 Dell PowerEdge R320 machines with 8GB RAM and 4-core CPUs, which act as object servers. The version of OpenStack Swift is Stein 5.2.0. All the machines run Ubuntu Server 20.04LTS and are interconnected through 1GbE links. The client machine for the experiments is equipped with a Intel Core i5-4440 CPU with 4 cores and 8GB RAM.

Competing systems. In some tests, we have compared EGEON against plain Zion [84] and Vertigo [83], all deployed in the same Swift cluster as above. Concretely, we have used Zion as a baseline to assess the overhead added by EGEON's software-defined architecture to the original Zion design, while we have chosen Vertigo as an example of a general-purpose, software-defined object storage system. As EGEON, Vertigo allows users to create pipelines of storage functions, each implemented as an OpenStack Storlet [65]. Hence, it is a good representative to act as a proving ground for the performance of EGEON against a similar software-defined architecture. It must be noticed that the control plane in Vertigo is programmatic, while in EGEON is declarative through the use of JSON-based privacy policies.

4.5.1 Microbenchmarks

We have run several microbenchmarks:

Cryptographic operations. To better understand the sources of overhead incurred by EGEON, we examined the throughput of the individual transformation UDFs, since different privacy policies may result in various transformation mixes. For each type of cryptographic transformation, we measured the number of operations per second that the EGEON runtime can perform on a object server in the data plane, as well as the latency. The meaning of each operation depends on the specific type of the cryptographic transformation UDF. For `HOM`, this refers to the summation of two encrypted 32-bit integers. For `PRE`, it refers to the proxy re-encryption of a single encrypted 32-bit integer, while for `SEARCH`, it represents the search of a keyword in an encrypted document with the same keyword. The results of this experiment are given in Table 4.1. As expected, we can observe that the latency of the cryptographic operations is in the order of a few milliseconds, which is acceptable for many use cases. Due to the added latency of the cryptographic transformations, a reactive data plane such as that available in EGEON, which is driven exclusively by the events appearing in the data streams, can be of great help to define privacy policies that minimize the number of cryptographic operations (for instance, by skipping unneeded data in the first steps of the transformation chain).

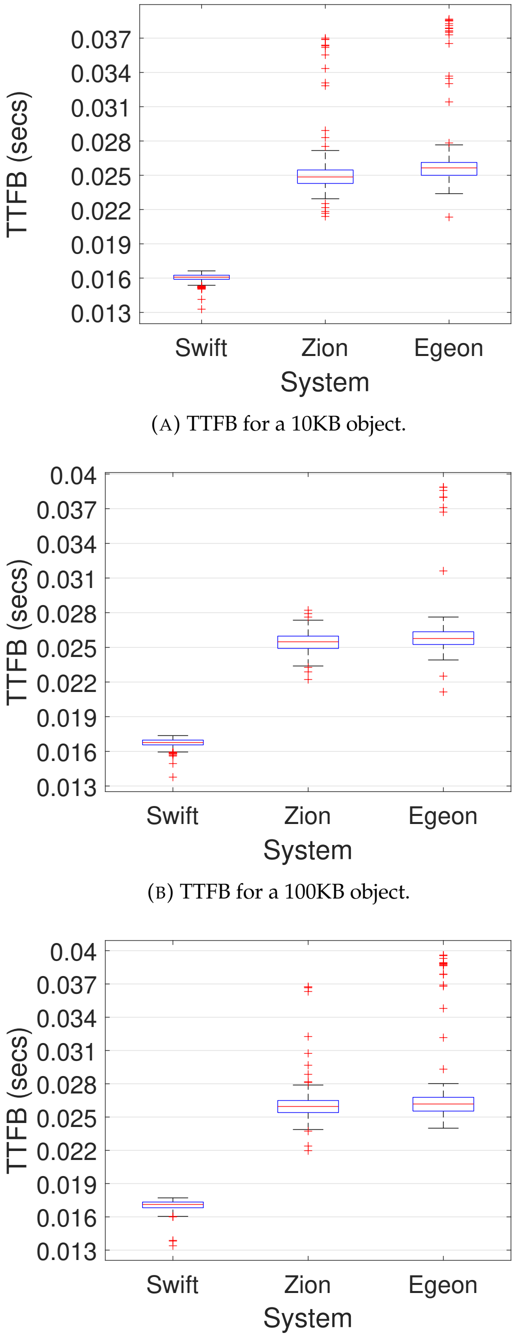
Overhead. To provide a full picture of the overheads incurred by EGEON, we measured the latency introduced by EGEON to the I/O path with respect to Zion and vanilla Swift. To measure the overhead, we utilized the Time to First Byte (TTFB), which captures how long the client needed to wait before receiving its first byte of the response payload from the Swift servers. To make measurements more precise, we colocated the client with one of the Swift proxy servers. As a client, we used `pycurl` to generate the `Get` requests for three different data file sizes. For each object size, we performed 1K requests. For Zion, each request caused the invocation of a `NOOP` function that simply echoes the input stream to the output (see Listing 1 in [84] for further details). For EGEON, we set up a `NOOP` policy which includes a single `NOOP` UDF in the transformation chain. An UDF that does nothing can be shown on Listing 4.7.

```java
package com.urv.egeon.function;

import com.urv.egeon.runtime.api.parser.UDF.ContextUDF;
import com.urv.egeon.runtime.api.parser.UDF.UDF;
import com.urv.egeon.runtime.api.parser.event.Event;

public class Noop extends UDF {
    @Override
    public Event update(Event e, ContextUDF ctx) { return e; }
    @Override
    public Object destroy(ContextUDF ctx) { return null; }
}
```

Listing 4.7: A no-operation (NOOP) UDF.

The results are given in Fig. 4.3. As seen in this figure, Zion and EGEON are on par, which demonstrates that EGEON software-defined architecture adds little overhead to Zion. With respect to vanilla Swift, both systems add around 9ms of extra latency, which can be considered very small. To wit, serverless function invocation in major cloud providers usually take between 25 to 320ms in warm state [105].

Since Zion does not support the pipelining of functions, we compared EGEON against Vertigo. Recall that Vertigo enables users to chain several Storlets together, where each Storlet can implement some reusable storage function such as decryption, compression, etc. We repeated the same experiment as above, but evaluating chains of increasing length. For EGEON, we set up chains of `NOOP` UDFs, while for Vertigo, we did the same, but for pipelines of `NOOP` Storlets. The results are depicted in Fig. 4.4. We can see that while the overhead keeps constant in EGEON, Vertigo shows a linear increase in latency. The reason for such a difference is the reactive core of EGEON, which does nothing if the transformations are not subscribed to any event. Vertigo, however, wires each Storlet with their neighbors in the chain, which takes some time, albeit each of them just copies the input to the output. This strongly reinforces the idea that for a software-defined data protection system to be useful, it is not a good idea to route the data streams through

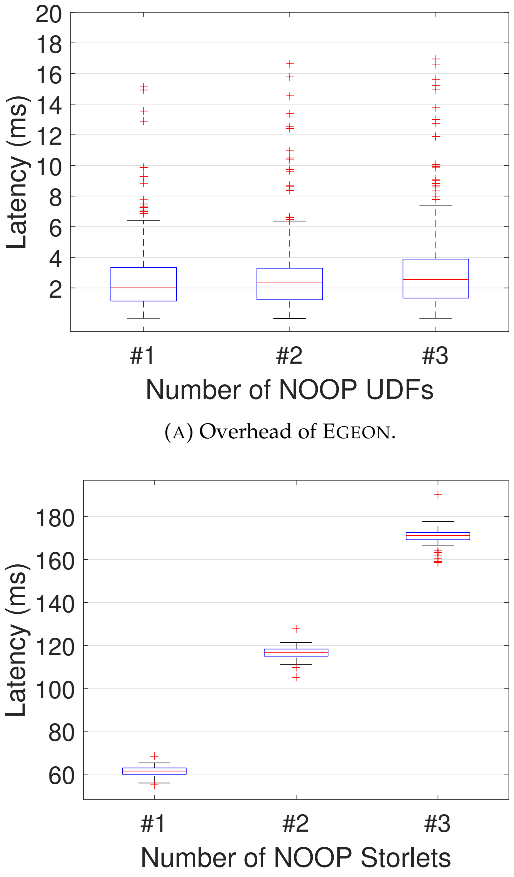


(A) TTFB for a 10KB object.

(B) TTFB for a 100KB object.

(C) TTFB for a 1MB object.

FIGURE 4.3: Time to First Byte (TTFB) for different object sizes.



(A) Overhead of EGEON.

(B) Overhead of Vertigo.

FIGURE 4.4: Overhead of chain setup of EGEON versus Vertigo.

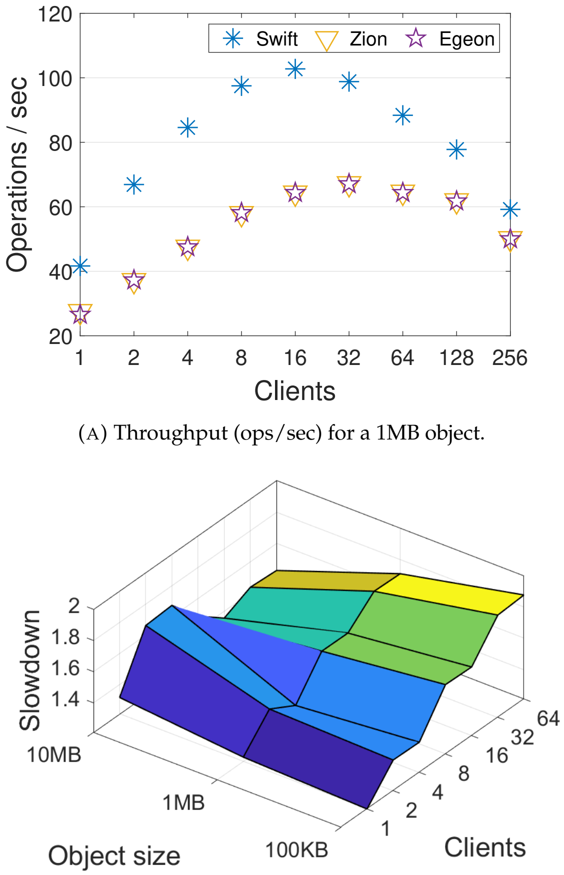
TABLE 4.1: Performance of cryptographic primitives available in EGEON.

Transformation UDF	**Throughput** (ops/sec)	**Latency** (ms)
Homomorphic Addition (SUM)	616	1.62
Proxy Re-Encryption (PRE)	137	7.29
Keyword Search (SEARCH)	166	6.02

a pipeline of functions, but rather to act on them when it is strictly needed. Indeed, Vertigo's overhead is more than one magnitude higher than EGEON's overhead as shown in Fig. 4.4.

Throughput. As a final microbenchmark, we quantified the impact of EGEON on the system throughput. As above, EGEON was compared to Zion and vanilla Swift to give real sense of its performance. For this experiment, we utilized the `getput` benchmarking tool suite [88] for Swift. And in particular, the `gpsuite` to conduct parallel tests with multiple clients. More concretely, we run `gpsuite` in one of the Swift proxy servers for 10 seconds and for different object sizes. We considered a replication factor of 3, and instrumented `gpsuite` to stress 3 out of the 6 object servers in the data plane. As in the overhead test, a NOOP function call per request was made for Zion and a NOOP policy plus NOOP UDF for EGEON. Table 4.2 reports the maximum throughput in operations per second attained by each system. Similarly to what was observed for the overhead, EGEON and Zion perform in similar terms. More interestingly, as the object size increases, the gap between both EGEON and Zion and Swift grows. We investigated this issue and we found that this happens due to a higher CPU interference caused by the JVM used to run the EGEON logic and the NOOP code in Zion, respectively.

Also, Fig. 4.5a plots the throughput for an increasing number of emulated clients for a 1MB object, which exhibits the same behavior as before. That is, EGEON and Zion showing a similar performance, while Swift delivering a much higher throughput due to the absence of any computation in the I/O path. Finally, Fig. 4.5b illustrates EGEON's slowdown relative to vanilla Swift, calculated as Slowdown = $\frac{\text{EGEON download time}}{\text{Swift download time}}$, as a function of the object size (x-axis) and the number of concurrent clients (y-axis). As can be seen in the figure, the slowdown factor does not increase steeply. Rather, it increases



(A) Throughput (ops/sec) for a 1MB object.

(B) Slowdown of EGEON over vanilla Swift.

FIGURE 4.5: Throughput and latency of EGEON under a multi-client setting.

TABLE 4.2: Maximum throughput (ops/sec) for different object sizes.

System	**Object size**		
	100KB	1MB	10MB
Swift	157,81	111,43	31,62
Zion	140,72	79,69	21,17
Egeon	140,56	79,59	21,26

gradually in both axes, never doubling the latency. This indicates that pushing down data protection logic to the storage may be acceptable for many applications.

4.5.2 Applications

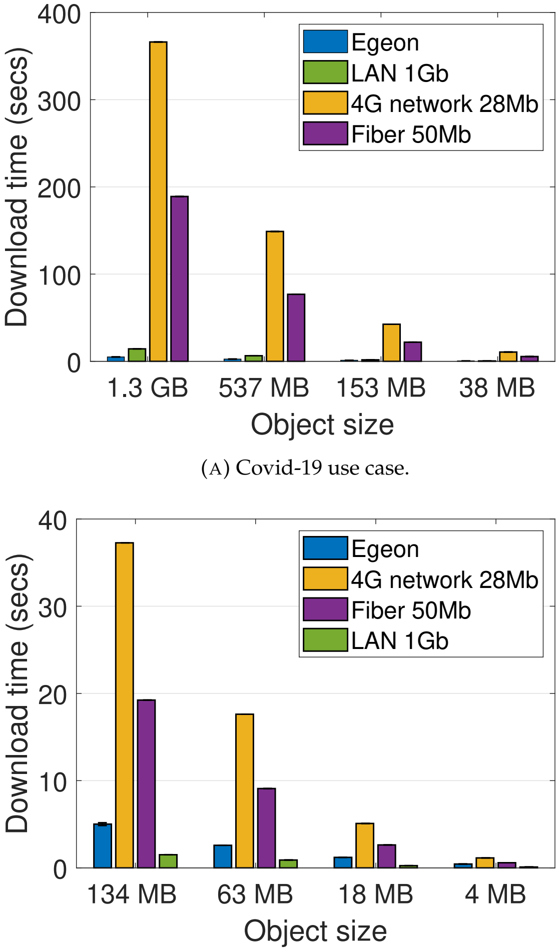
To evaluate the composability of EGEON, we have designed two privacy policies that capture the different complexities of real-world applications. These applications are the following:

Covid-19 **use case.** In this use case, we demonstrate the same policy of Listing 1, but applied to healthcare. We use the JSON file reported by the US government that summarizes the patient impact on healthcare facilities caused by Covid-19 [99] (April 2021). Specifically, we exchange the user label “`treasurer`" by “`state coordinator`", and label the field that reports the sum of patients hospitalized in a pediatric inpatient bed in 7-day periods [FAQ-10.b)] as “`sensitive`", as it reveals which hospitals may be collapsed. The rest of information is ignored. As a result, the transformed view for state officials only bears a single homomorphically encrypted value that aggregates the sum of all healthcare facilities. As in Listing 1, the policy links 3 UDFs: `CLAC`→`HOM`→`PRE`. To play out with the file size, we split this dataset into three smaller files based on increasingly smaller time periods: year, month and week.

Adult dataset [61] use case. This dataset in CSV format from the UCI Machine Learning Repository [61] has 48842 records and 14 attributes. Some of these attributes can leak sensitive information such as race and occupation. For this use case, we have decided to protect the attribute #7: occupation, with the `SEARCH` scheme to allow for type-of-employment searches on encrypted text. To prove composability, we have used `CLAC` to protect three attributes out of the four attributes chosen for this experiment. We have used one object label: “`sensitive`", and one user label: “`HR manager`", so that only a human resources manager can retrieve the three protected columns. Further, we have encrypted other fields to increase the file size to 134MB, to later split it into 3 smaller files based on the attribute #6: marital-status. The policy is as follows (some fields have been omitted for brevity):

```
{
 "Object": "v1/{account}/{container}/adult.csv",
 "Action": {
  "StartAt": "Step1",
  "Steps": {
    "Step1": {
     "Id": "CLAC",
     "EventType": {
        "Type":  "ColumnMarkerEvent",
        "Input": [{"columns": [2,6,7], "olabel": "sensitive" }]
     },
     "Input": [{"ulabel": "HR manager", "olabel": "sensitive" }],
     "Next": "Step2"
    },
    "Step2": {
     "Id": "SEARCH",
     "EventType": {"Type": "ColumnEvent", "Input": [{
   "column":7}]},
     "Next": "End"
 }}}
```

LISTING 4.8: A policy definition example.



(A) Covid-19 use case.

(B) Adult dataset use case.

FIGURE 4.6: Performance of EGEON in two composite policies.

Experiment. In this test, we measure the time to download the raw files directly from Swift against the time to download the transformed views generated by EGEON. The goal is to decide if it is worth to push the privacy transformations into the object store instead of running them on the user devices and VMs, so that software-defined data protection is within reach. To do so, we have capped the ingoing bandwidth of our client machine to emulate different network speeds and customary scenarios: Fiber network speeds to emulate home and business users, 4G network bandwidth to emulate mobile users, and finally, LAN to simulate a scenario where the client and Swift servers reside on the same local area network (e.g., a university intranet). The exact network speeds are listed in Table 4.3. We performed 1K executions per object size and network speed.

TABLE 4.3: Network Speeds.

Network	4G	Fiber	LAN
Median speed (Mbps)	28.9 [109]	55.98 [108]	887 (SpeedTest)

The results are plotted in Fig. 4.6a for the Covid-19 use case, and in Fig. 4.6b for the Adult dataset. Error bars display the standard deviations of results, which are indeed very narrow. Non-surprisingly, we can see that EGEON lowers the download time significantly for the slow 4G and fiber connections, which means that pushing down privacy transformations to storage is a good deal better than the naive approach of encrypting data on the client side and retrieve the whole file as alleged by cloud providers such as AWS for S3. The savings in some scenarios can be dramatic such as in the Covid-19 use case, where just a few bytes (e.g., aggregates such as `SUM`, `COUNT` and `AVG`, etc.) are consumed by the application, reaching 72.1X speedup for 4G mobile terminals. For the LAN setting, the benefits are not so clear, and for the Adult dataset, EGEON shows a slowdown factor of 3.3X in the worst case due to the heavy computations associated with keyword search—actually, the test primitive of `SEARCH` requires three pairing operations [35]. Either way, we believe that EGEON's fine-grained data protection capabilities outweigh the slight loss of performance.

4.6 Work-In-Progress: porting EGEON to the edge.

Although our original aim was on cloud object storage, we felt that moving EGEON to the edge can provide a cost-effective and secure data infrastructure for many applications. With 5G around the corner and the Internet of Things (IoT) economy exploding, the truth is the current cloud infrastructure will struggle to keep up and can quickly accumulate multiple PBs of data in no time. PBs of data that can be processed close to the devices [12, 73]. One of the key challenges in the development of applications at the edge is the efficient data sharing between the multiple edge clients. In this sense, edge storage can greatly improve data access and enable latency-sensitive applications [97, 55]. However, the dynamic and heterogeneous environment in the edge, along with the diverse application requirements poses several challenges such as:

- Stringent resources of the edge servers;
- Preprocessing of different data storage formats and data types, often involving Extract, Transform, Load (ETL) operations;
- Execution of low-latency data analytics applications;
- Security and privacy concerns; and
- Quality of service (QoS) guarantees.

Despite recent advancements in the pursuit of practical edge storage solutions, there is no solution to address all the above challenges in a holistic manner. To support a variety of data formats, such as documents, images, and other unstructured data types, the object storage model is the right abstraction. Certainly, object storage is the storage of choice in the cloud, then:

Why couldn’t object storage be the storage of choice for the edge?

The answer depends on many factors. Above all, the edge storage model is useful in those situations where the objective is to perform processing and analytics at the edge, filtering out unnecessary data and retaining or sending only the insights and relevant data to the cloud. In this model, the compute and storage components are co-located at the edge and are specifically designed to store and process data on-site. The primary aim is not to store petabytes of data at the edge. Instead, this model envisions managing data ranging from a hundreds of gigabytes up to a terabyte.

Unfortunately, as discussed in this thesis, solutions such as OpenStack Swift, AWS S3 and MinIO [58] , the most prominent technical cloud solutions, fall short to address all the above challenges. Just to illustrate, the above solutions are general-purpose and have not been designed with NDP in mind. NDP refers to a computational paradigm where processing elements are co-located "next to" data (see section §**??**sec:ndpexplain for further details). The absence of NDP support implies that the processing of a datum requires shipping it out of the local storage subsystem to a separate software stack for data analysis, often located at a distance, which can lead to increased data movement, latency, and potential performance bottlenecks. Even storage system such as Swift, which provides some incipient NDP support in form of storlets [62], is not well-prepared for edge computing. Storlets are small Java applications that can be launched within a Swift deployment. However, the lack of resources problem still remains: the execution of a simple storlet requires launching a Java Virtual Machine (JVM), which may be too heavyweight for many edge servers.

Based on the above observations, we have started to adapt EGEON to become an edge object storage solution with NDP support, or put another way, with UDF support for data transformations. The new edge storage solution must be "all-in-one" and fit in a single server. In principle, the current architecture of EGEONallows this, providing a unified platform that can handle both storage and data transformations. However, its architecture has a large memory and storage footprint, mostly due to its Java-based runtime. To mitigate this, we have re-written the whole Java executor into C++, which allows us to have a fine-grained control of the memory, along with a series of modifications to make it more efficient. Since this research is still in progress, here only describe most important modifications undertaken to make EGEONmore edge-compliant, together with some preliminary results.We call this new variant "EGEON-EDGE".

4.6.1 Modifications to the runtime

Here we discuss the main changes to the original system. First off, we have implemented a new C++ multi-threaded executor for EGEON. The new executor is more lightweight, and is compiled to the underlying system architecture (e.g., x86_64) for improved performance when EGEONis deployed on an edge server. As in the Java executor, the new executor is containerized, but the new containers is built from a "distroless" image. Distroless images contain only your application and its runtime dependencies. They do not contain package managers, shells or any other programs you would expect to find in a standard Linux distribution. Concretely, we used the container image `gcr.io/distroless/cc`, maintained by Google. This image contains a minimal Linux, `glibc` runtime and has a size of around 24 MiB. As we will see in Section 4.6.2, the use of small container image allowed us to minimize the memory footprint, as well as to reduce significantly the cold start time, i.e., the time it takes for a container instance to boot up. Fig.4.7 shows an overview of the new C++ runtime of EGEON.

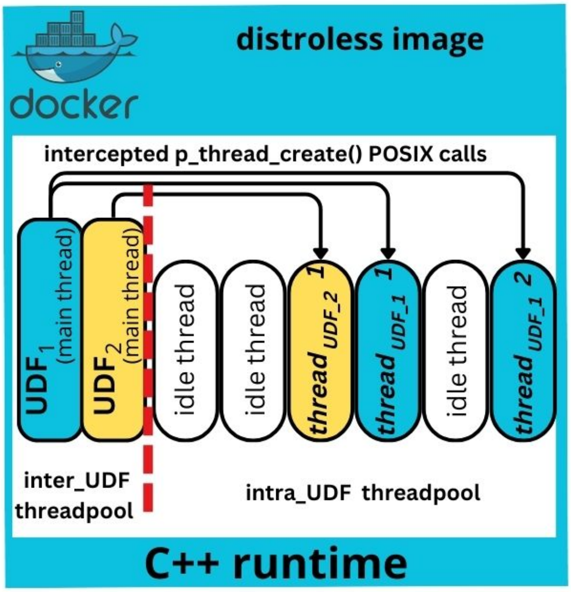


FIGURE 4.7: EGEON-EDGE new runtime. Two main concurrent UDFs (UDF$_1$ and UDF$_2$) from *inter_UDF threapool* are executing various threads, which are grabbed from *intra_UDF threadpool* on each intercepted `pthread_create` POSIX call.

Executor-to-object server communication. To communicate the object server with a multi-threaded executor instance, we use Linux pipes. The object server reads the file from disk and passes the file to the executor using a pipe. The executor processes the file as a stream and sends the transformed file chunks back to the object server using another pipe. In this back and forth, multiple data copies take place between the user space and the kernel space, hurting performance. To alleviate this, EGEON-EDGE uses the `vmsplice` system call to attach the virtual memory pages from the executor process directly into the kernel pipe buffer, avoiding to copy the transformed data chunks from user space to kernel space. When the object server instance calls the `readv` system call, the data is then directly copied from these pages to the destination buffer. In this way, the data is transferred using a single memory copy. Communication between an object server and a process server continues to be HTTP-based.

By default the Linux kernel has a compile-time limitation of 16 pages per pipe, i.e., 4 KiB/page, for a total limit of 64 KiB transferred per call to `vmsplice` or `readv`. At first glance this limitation may seem cumbersome. However, in practice, our preliminary results show a significant improvement with the default values, with no need to use huge pages, which may provide unsatisfactory performance on important applications [75].

Hierarchical thread pool. To reuse the same executor for handling concurrent requests, the new executor is also multi-threaded. However, the multi-thread support has been implemented as a thread pool to eliminate the cost of thread creation and destruction. Upon a new object request, a thread is borrowed from the pool to execute the corresponding UDF. When the UDF finishes execution, the thread is automatically returned to the pool for later usage. An interesting feature of the thread pool is that is hierarchical. By "hierarchical", we mean that the thread pool has been split into two regions. The first region, or inter-UDF thread pool, is used to execute the main thread of the UDFs; however, if the UDF in turn creates new threads, the new threads are grabbed from the second region of the pool, or intra-UDF thread pool. We have adopted this structure to enable UDFs to leverage intra-UDF parallelism, which is very interesting for commonplace CPU-intensive operations such as compression and encryption. To stick with the standards, the intra-UDF parallelism adopts the POSIX Thread model, which is available in many operating systems, including Windows.

To implement it, we have replaced and rewritten the corresponding function such as `pthread_create` in the POSIX Thread API with the `LD_PRELOAD` library. With the `LD_PRELOAD` library, the executor intercepts the calls to `pthread_create` and replaces it by a custom implementation that borrows a thread from the second region of the pool. If there are not sufficient threads in this region, the executor traps and cancels the execution of the whole threaded UDF. In case there are no sufficient threads in the first region, the incoming object request is enqueued until a thread becomes available, or the request simply times out. We believe that this design is well consistent with object storage semantics.

Last but not least, it must be noted that the size of the thread pool is configurable. By now, the system administrator can decide how many threads will be allocated to the first and second regions of the pool. We hope in a near future to provide a simple API with calls such as `set_num_intraop_threads` to dynamically resize the second region of the pool.

4.6.2 Preliminary results

In this section, we provide some preliminary results of EGEON-EDGE. The results in this section show how the new system behaves in terms of scalability, resource consumption, and response time to the end user. This data provides valuable insights into the efficacy of the implemented changes and helps in identifying areas that may require further optimization or adjustment.

Setup. For the experiments, all the systems were deployed on a single VM instance with the following specs: 4GiB of RAM and 2 vCPUs, equipped with the Linux distribution 20.04 Ubuntu 64 bits, which emulates a small edge server. In addition to the original EGEON system, which we call it now EGEON-CLOUD to differentiate it from the new variant, we compare EGEON-EDGE to ZION.

Efficiency of EGEON-EDGE. We first focus on the difference in the resource footprint and cold-start initialization latency between EGEON-EDGE and ZION. To this aim, we leveraged various tools such as Prometheus [72] and cAdvisor [18] to extract metrics such as CPU utilization and memory utilization for both the EGEON-EDGE and ZION executors. To measure memory consumption, we used the resident set size (RSS) metric, which reflected the number of pages each executor had in real memory. To have a broader view, we got these metrics for both executors at rest (i.e., when they are not running any UDF), and when running the `NOOP` storage function, that is, a UDF that simply echoes the input object to the output. With this, we can report the minimal footprint in both the passive and active states.

We also investigated cold-start initialization times. To do so, we measured the time to create a new container and have the corresponding executor up and running. Finally, we observed the capacity as the maximum number of concurrent running executors that the VM can sustain before running out of memory. These metrics give a sense of the elasticity of both systems, with the one with more capacity and lower cold-start times exhibiting a higher degree of elasticity.

Results are given in Table 4.4. This table shows several orders of magnitude improvement in the resource footprint. This comes from the thinning of the container image and the design of a lighter executor written in C++, tailored to the execution of storage functions. Altogether, this results in a significantly lower CPU utilization, of up to 20 times less under similar conditions. Furthermore, RAM utilization has markedly decreased with the new incarnation of EGEON, reaching a reduction factor of up to $2,600$ times compared to ZION.

In terms of elasticity, the VM can support up to 5.76X more executors in EGEON than in ZION, while exhibiting a 70% lower cold start initialization time. These two metrics are very important in a serverless context. A higher capacity means a higher packing density of concurrent storage functions on a given host. Low initialization times reduce cost and latency for the user, through their mitigation of the cold-start problem.

Performance of EGEON-EDGE. To get a first sense of the performance of EGEON-EDGE, we coded and executed a UDF that performs a `grep` to search the keyword `IGMP` in a 1.97GB network log file. We chose this function because it is an IO-bound task, more amenable to what would be run on an edge server, where some preprocessing is typically performed to sanitize the data before further processing it in the cloud. As a baseline, we

TABLE 4.4: Efficiency comparison of ZION and EGON-EDGE. RSS stands for Resident Set Size. Capacity is measured as the number of containerized executors concurrently running in both systems.

Metric	**Zion**	**EGEON-EDGE**	Improvement Factor
Cold Start	1.8709s	1.0960s	1.70X
image size	422MB	34MB	17.51X
# containers	135	778	5.76X
RSS idle	48,345,088B	16,384B	2,646X
RSS NOOP	59,805,696B	151,552B	395X
CPU idle	0.580%	0.029%	19.5X
CPU noop	0.812%	0.035%	22.6X

also ran the NOOP UDF. To obtain meaningful results, we ran both functions 100 times in pre-warm state. As performance indicators, we measured the Time to First Byte (TTFB) and the Time to Last Byte (TTLB), or download time.

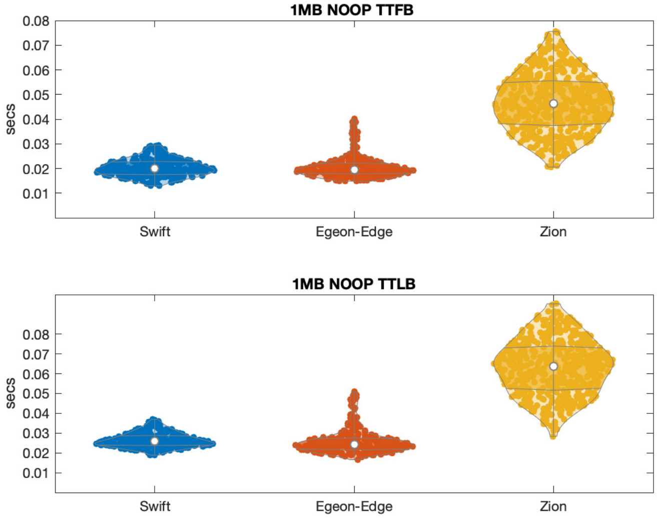


FIGURE 4.8: Time to First Byte (TTFB) and Time to Last Byte (TTLB) of the NOOP UDF comparison between Swift, EGEON-EDGE (C++ Executor) and EGEON-CLOUD (JAVA executor).

The results for the NOOP UDF can be found in Fig. 4.8. As can be seen in this figure, the TTFB is almost equivalent to that of Swift, and more than 2X shorter than ZION. Likewise, the TTLB of EGEON-EDGE was very similar to that of Swift, and 3X shorter

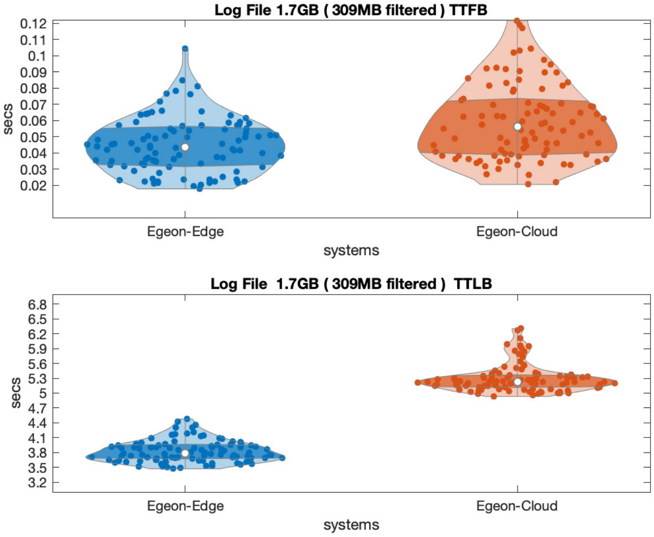


FIGURE 4.9: Time to First Byte (TTFB) and Time to Last Byte (TTLB) of the `grep` UDF comparison between EGEON-EDGE to EGEON-CLOUD.

than that of ZION. The reason for this improvement is attributable to the use of the `vmsplice` system call that allowed us to avoid copying data in and out of an opaque in-kernel pipe buffer, and the development of a lighter executor compared to ZION, which is managed by the JVM garbage collector. This trend also turns up for the `grep` UDF, albeit smaller in magnitude, as shown in Fig. 4.9. The reason is that the this UDF is not purely I/O and CPU costs reduce the benefit of a single-copy IPC. This result could be improved by letting EGEON pipeline the write of a data chunk to the pipe with the processing of the next data chunk in another thread. We left this improvement as future work. Either way, the performance results seem promising, and research path to continue exploring in the next months with more evaluation and enhancements.

4.7 Conclusions

As increasingly much more sensitive data is being collected to gain valuable insights, the need to natively integrate privacy controls into the storage systems is growing in importance. In particular, the poor interface of object storage systems, which lacks of sophisticated data protection mechanisms, along with the inherent difficulties to refactor them, have motivated us to design and implement EGEON. To put in a nutshell, EGEON is a novel software-defined data protection framework for object storage.

It allows data owners to define privacy policies on how their data can be shared, which permit the composition of data transformations to build sophisticated data protection controls. In this way, data owners can specify multiple views from the same data piece, and modify these views by only updating the policies (e.g., by modifying the chain of transformations that produce a particular view), leaving the system internals intact. The EGEON prototype has been coded atop OpenStack Swift. Our evaluation results demonstrate that EGEON adds little overhead to the system, yet empowering users with the needed controls to ensure strong data protection.

As a work in progress (WIP), we are porting EGEON to the edge by rewriting critical parts of the software stack in C++ such as the executor, getting promising results. For instance, the new enhancements have shown several orders of magnitude improvement in the resource footprint and cold start initialization times compared to the original system implementation.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

The increasing popularity of cloud storage services like Dropbox and Google Drive underscores the importance of optimizing network traffic. Sync deferment has been identified as a crucial technique in this context, particularly effective in scenarios of frequent file modifications. Current literature affirms the convenience of this technique,yet it also highlights performance shortcomings in existing implementations.

The first key question of this thesis appears here: **Could file synchronization deferment be optimized to help mitigate the traffic overuse problem?.**

Based on the research and experimental results, the answer is affirmative. We have developed an innovative adaptive sync deferment algorithm named Rate-based Sync Deferment (RSD) which matches the current state of the art in terms of overhead but also stands out by reducing file synchronization delays.

The adaptive nature of RSD allows it to dynamically adjust to varying conditions, thereby enhancing efficiency. This is particularly crucial in cloud storage environments where file modification frequencies can be unpredictable and varied. The experimental results are compelling, showing improvements in sync delay by factors ranging from 2X to 12X which mitigates network traffic overuse and also enhances user experience by ensuring more timely updates of their stored data.

On another note, in the context of personal cloud storage services, especially in collaborative scenarios, the second key question of this thesis arises: **Could security data management be effectively outsourced to the storage cloud by applying the principles of software-defined storage?**

To answer this question, a novel software-defined data protection framework for object storage named EGEON has been developed, demonstrating its practical applicability and effectiveness by introducing minimal overhead to the system. EGEON asnswers the growing need for integrated privacy controls in storage systems, particularly in collaborative environments where data security is paramount.

EGEON allows data owners to define privacy policies that control how their data can be shared. This capability enables the composition of data transformations to create sophisticated data protection controls. By specifying multiple views from the same data and modifying these views through policy updates (for example, altering the chain of transformations that produce a particular view), data owners can manage their data security without altering the system's internals.

EGEON introduces a minimal overhead of just 9ms, which is quite modest when compared to the 25 to 320 ms overhead typically associated with cloud providers [105]. In terms of performance, EGEONexhibits a certain degree of slowdown, but is relatively small. In practical terms, this means that the performance degradation, while noticeable, is not excessively burdensome. It never exceeds double the time it would take without it.

For a wide range of applications, the overhead level introduced by EGEON is generally acceptable. In most cases, the advantages offered by EGEON, particularly in terms of enhanced data security and secure cooperation scenarios, significantly outweigh the minor performance reduction it causes. This situation presents a strategic balance between functionality and efficiency. For numerous users and organizations, this trade-off could be considered worthwhile.

5.2 Future Work

As a work in progress, we are exploring object storage as a potential storage solution for edge computing. It's important to emphasize that the decreased resource footprint and quicker initialization times are key elements in edge environments. In these settings, where resources are typically limited, and responsiveness is crucial, these factors become even more critical.

In this sense, EGEON is being adapted to edge computing by rewriting critical parts of its software stack in C++, focusing on components like the executor. This has yielded promising results, improving by several orders of magnitude both the resource footprint and cold start initialization times compared to the original Cloud-based system implementation. This suggests that object storage could indeed be a viable choice for edge computing.

However, it is important to consider that the journey to make object storage suitable for the edge involves overcoming inherent challenges such as limited computing power, storage capacity and network connectivity. The ongoing work on EGEON indicates that these challenges are not insurmountable but require changes, for example, to resilence.

Resilience is a critical factor for storage in edge computing environments. In such settings, IoT devices and edge data centers are often difficult to physically access and maintain. Unlike drives in traditional data centers, those in IoT and edge data centers face more challenging physical conditions, increasing the likelihood of drive failures. Given this reality, it is s imperative for these architectures, particularly the storage components, to be designed to fail in place. This means accepting that drive failures are inevitable and ensuring that these failures do not result in data loss. This scenario underscores the importance of a self-healing and automated storage architecture. Such a system can safeguard data even in the event of drive failures and is capable of automatically failing over to alternate data centers if all drives at a specific edge location fail.

Therefore, it would be desirable to have a sort of localized cloud at the edge, complemented by the backup of central cloud storage. By adopting this approach, we enhance the resilience of data. In the event of a failure, we would be supported by our nearby edge cloud, while still having the reassurance of backup from the main cloud storage. This dual-layered strategy ensures data robustness, providing a fail-safe mechanism where the nearby edge cloud offers immediate support and the central cloud storage serves as a reliable safety net.

In that sense, we are planning to implement a dual cloud strategy, combining the localized processing power of edge cloud capabilities with the comprehensive backup and robustness offered by central cloud storage.

Bibliography

[1] Anurag Acharya, Mustafa Uysal, and Joel Saltz. “Active disks: Programming model, algorithms and evaluation”. In: *ACM SIGOPS Operating Systems Review* 32.5 (1998), pp. 81–91.

[2] Mahmuda Akter et al. “Performance Analysis of Personal Cloud Storage Services for Mobile Multimedia Health Record Management”. In: *IEEE Access* 6 (2018), pp. 52625–52638. DOI: `10.1109/ACCESS.2018.2869848`.

[3] Amazon. *Amazon S3 Developer Guide*. `http://aws.amazon.com/documentation/s3`. 2018.

[4] Amazon. *AWS S3 Prefixes*. `https://docs.aws.amazon.com/AmazonS3/latest/userguide/using-prefixes.html`.

[5] Amazon. *AWS S3 Strong Read after Write Consistency*. `https://aws.amazon.com/es/blogs/aws/amazon-s3-update-strong-read-after-write-consistency`.

[6] Diego F. Aranha and Conrado P.L. Gouvêa. *RELIC is an Efficient LIbrary for Cryptography*. `https://github.com/relic-toolkit/relic`. 2017.

[7] Joe Arnold. *Openstack swift: Using, administering, and developing for swift object storage.* " O’Reilly Media, Inc.", 2014.

[8] *Baidu*. Baidu. 2024. URL: `https://www.baidu.com` (visited on 01/01/2024).

[9] Engineer Bainomugisha et al. "A Survey on Reactive Programming". In: *ACM Comput. Surv.* 45.4 (Aug. 2013).

[10] Ioana Baldini et al. "Serverless computing: Current trends and open problems". In: *Research advances in cloud computing* (2017), pp. 1–20.

[11] Daniel Barcelona-Pons, Pedro García-López, and Bernard Metzler. "Glider: Serverless Ephemeral Stateful Near-Data Computation". In: *Proceedings of the 24th International Middleware Conference*. 2023, pp. 247–260.

[12] Luciano Baresi and Danilo Filgueira Mendonça. "Towards a serverless platform for edge computing". In: *2019 IEEE International Conference on Fog Computing (ICFC)*. IEEE. 2019, pp. 1–10.

[13] Paulo S. L. M. Barreto and Michael Naehrig. "Pairing-Friendly Elliptic Curves of Prime Order". In: *Selected Areas in Cryptography (SAC)*. 2006, pp. 319–331.

[14] Prosunjit Biswas, Farhan Patwa, and Ravi Sandhu. "Content Level Access Control for OpenStack Swift Storage". In: *ACM CODASPY*. 2015, pp. 123–126.

[15] Haran Boral and David J DeWitt. "Database machines: An idea whose time has passed? A critique of the future of database machines". In: *Database Machines: International Workshop Munich, September 1983*. Springer. 1983, pp. 166–187.

[16] Box. *Box Cloud Storage*. `https://www.box.com/`.

[17] Lukas Burkhalter et al. "Zeph: Cryptographic Enforcement of End-to-End Data Privacy". In: *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, July 2021, pp. 387–404.

[18] *cAdvisor*. URL: `https://github.com/google/cadvisor` (visited on 01/01/2024).

[19] Larry Coyne et al. *IBM Software-Defined Storage Guide*. IBM Redbooks, 2018.

[20] Angelo De Caro and Vincenzo Iovino. "jPBC: Java pairing based cryptography". In: *16th IEEE Symposium on Computers and Communications, ISCC 2011*. 2011, pp. 850–855.

[21] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified data processing on large clusters". In: (2004).

[22] David J DeWitt and Paula B Hawthorn. "A performance evaluation of database machine architectures". In: (1981).

[23] Idilio Drago et al. "Inside dropbox: understanding personal cloud storage services". In: *Proceedings of the 2012 internet measurement conference*. 2012, pp. 481–494.

[24] Dropbox. *Dropbox Cloud Storage*. `https://www.dropbox.com/`.

[25] Michael Factor. "Storlets: Turning Object Storage into a Smart Storage Platform". In: *IBM Research Blog* (2014).

[26] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[27] Xinzhou Geng et al. "Research on early warning system of power network overloading under serverless architecture". In: *2018 2nd IEEE Conference on Energy Internet and Energy System Integration (EI2)*. IEEE. 2018, pp. 1–6.

[28] Google. *Guava: Google Core Libraries for Java*. `https://github.com/google/guava`. 2021.

[29] Raúl Gracia-Tinedo et al. "Crystal: Software-Defined Storage for Multi-Tenant Object Stores". In: *15th USENIX Conference on File and Storage Technologies (FAST 17)*. 2017, pp. 243–256.

[30] Raúl Gracia-Tinedo et al. "Dissecting ubuntuone: Autopsy of a global-scale personal cloud back-end". In: *Proceedings of the 2015 Internet Measurement Conference*. 2015, pp. 155–168.

[31] Raúl Gracia-Tinedo et al. "Giving wings to your data: A first experience of Personal Cloud interoperability". In: *Future Generation Computer Systems* 78 (2018), pp. 1055–1070.

[32] Raúl Gracia-Tinedo et al. "Software-defined object storage in multi-tenant environments". In: *Future Generation Computer Systems* 99 (2019), pp. 54–72.

[33] Hassan B Hassan, Saman A Barakat, and Qusay I Sarhan. "Survey on serverless computing". In: *Journal of Cloud Computing* 10.1 (2021), pp. 1–29.

[34] David K. Hsiao. “Data base machines are coming, data base machines are coming!” In: *Computer* 12.03 (1979), pp. 7–9.

[35] Yong Ho Hwang and Pil Joong Lee. “Public key encryption with conjunctive keyword search and its extension to a multi-user system”. In: *International conference on pairing-based cryptography*. Springer. 2007, pp. 2–22.

[36] IBM. *IBM Clod Object Storage*. `https://cloud.ibm.com/docs/services/cloud-object-storage`. 2018.

[37] *iCloud*. iCloud. 2024. URL: `https://www.icloud.com` (visited on 01/01/2024).

[38] Zsolt István, Soujanya Ponnapalli, and Vijay Chidambaram. “Software-Defined Data Protection: Low Overhead Policy Compliance at the Storage Layer is within Reach!” In: *Proc. VLDB Endow.* 14.7 (2021), pp. 1167–1174.

[39] Eric Jonas et al. “Cloud programming simplified: A berkeley view on serverless computing”. In: *arXiv preprint arXiv:1902.03383* (2019).

[40] Kimberly Keeton, David A Patterson, and Joseph M Hellerstein. “A case for intelligent disks (IDISKs)”. In: *Acm Sigmod Record* 27.3 (1998), pp. 42–52.

[41] Robert Krahn et al. “Pesos: Policy Enhanced Secure Object Store”. In: *Thirteenth EuroSys Conference (EuroSys ’18)*. 2018.

[42] Diego Kreutz et al. “Software-defined networking: A comprehensive survey”. In: *Proceedings of the IEEE* 103.1 (2014), pp. 14–76.

[43] Sameer G Kulkarni et al. “Living on the edge: Serverless computing and the cost of failure resiliency”. In: *2019 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*. IEEE. 2019, pp. 1–6.

[44] Leo Leung. *Core Concepts, Oracle Cloud Infrastructure*. Youtube. 2019. URL: `https://www.youtube.com/watch%20v=BCj4cLYB8NU&list=PLvlciYga5j3zcvx_1BgVaPk1-BX7t2YcK` (visited on 01/01/2024).

[45] Zhenhua Li, Zhi-Li Zhang, and Yafei Dai. “Coarse-grained cloud synchronization mechanism design may lead to severe traffic overuse”. In: *Tsinghua Science and Technology* 18.3 (2013), pp. 286–297.

[46] Zhenhua Li et al. "A quantitative and comparative study of network-level efficiency for cloud storage services". In: *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)* 4.1 (2019), pp. 1–32.

[47] Zhenhua Li et al. "Efficient batched synchronization in dropbox-like cloud storage services". In: *Middleware 2013: ACM/IFIP/USENIX 14th International Middleware Conference, Beijing, China, December 9-13, 2013, Proceedings 14*. Springer. 2013, pp. 307–327.

[48] Zhenhua Li et al. "Towards network-level efficiency for cloud storage services". In: *Proceedings of the 2014 Conference on Internet Measurement Conference*. 2014, pp. 115–128.

[49] Pedro Garcia Lopez, Raul Gracia Tinedo, and Alberto Montresor. "Towards Data-driven software-defined infrastructures". In: *Procedia Computer Science* 97 (2016), pp. 144–147.

[50] Pedro Garcia Lopez et al. "Stacksync: Bringing elasticity to dropbox-like file synchronization". In: *Proceedings of the 15th International Middleware Conference*. 2014, pp. 49–60.

[51] Oleg Lvovitch and Sally Guo. *AWS re:Invent 2022 - Deep dive on Amazon S3 (STG203)*. Youtube. 2022. URL: https://www.youtube.com/watch?v=FJJxcwSfWYg (visited on 01/01/2024).

[52] Jonathan Mace et al. "Targeted resource management in multi-tenant distributed systems". In: *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*.

[53] Ricardo Macedo et al. "A survey and classification of software-defined storage systems". In: *ACM Computing Surveys (CSUR)* 53.3 (2020), pp. 1–38.

[54] Ashraf Mahgoub et al. "SONIC: Application-aware Data Passing for Chained Serverless Applications". In: *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 2021, pp. 285–301.

[55] Antonios Makris et al. "Towards a Distributed Storage Framework for Edge Computing Infrastructures". In: *Proceedings of the 2nd Workshop on Flexible Resource and Application Management on the Edge*. 2022, pp. 9–14.

[56] *Microsoft*. Microsoft. 2024. URL: https://www.microsoft.com/en-us/microsoft-365/onedrive/online-cloud-storage (visited on 01/01/2024).

[57] Leonardo Militano. *Experimenting on Ceph Object Classes for Active Storage*. https://blog.zhaw.ch/icclab/experimenting-with-ceph-object-classes-for-active-storage/. 2019.

[58] *Minio Private Cloud Storage*. https://min.io/.

[59] Ingo Müller, Renato Marroquín, and Gustavo Alonso. "Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure". In: *2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20)*. 2020, pp. 115–130.

[60] Maurizio Naldi and Loretta Mastroeni. "Cloud storage pricing: A comparison of current practices". In: *Proceedings of the 2013 international workshop on Hot topics in cloud services*. 2013, pp. 27–34.

[61] C.L. Blake D.J. Newman and C.J. Merz. *UCI Repository of machine learning databases*. https://archive.ics.uci.edu/ml/index.php. 1998.

[62] OpenStack. *Github Storlets*. https://github.com/openstack/storlets.

[63] OpenStack. *Middleware and Metadata*. https://docs.openstack.org/swift/latest/development_middleware.html. 2021.

[64] OpenStack. *Swift*. https://docs.openstack.org/swift/. 2021.

[65] OpenStack. *Welcome to storlets' documentation!* https://docs.openstack.org/storlets/latest/. 2020.

[66] openstack. *openstack Object Storage Middleware*. openstack. 2023. URL: https://docs.openstack.org/swift/ussuri/middleware.html (visited on 01/01/2024).

[67] openstack. *openstack Swift Architecture*. openstack. 2023. URL: https://docs.openstack.org/swift/stein/overview_architecture.html (visited on 01/01/2024).

[68] Esen A Ozkarahan, Stewart A Schuster, and Kenneth C Smith. “RAP: an associative processor for data base management”. In: *Proceedings of the May 19-22, 1975, national computer conference and exposition*. 1975, pp. 379–387.

[69] Calicrates Policroniades and Ian Pratt. “Alternatives for Detecting Redundancy in Storage Systems Data.” In: *USENIX Annual Technical Conference, General Track*. 2004, pp. 73–86.

[70] Rogério Pontes et al. “Safefs: A modular architecture for secure user-space file systems: One fuse to rule them all”. In: *Proceedings of the 10th ACM International Systems and Storage Conference*. 2017, pp. 1–12.

[71] Raluca Ada Popa et al. “CryptDB: Protecting Confidentiality with Encrypted Query Processing”. In: *Twenty-Third ACM Symposium on Operating Systems Principles (SOSP’11)*. 2011, pp. 85–100.

[72] *Prometheus . Monitoring system and time series database.* URL: https://prometheus.io/ (visited on 01/01/2024).

[73] Ioannis Psaras et al. “Mobile data repositories at the edge”. In: *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)*. 2018.

[74] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. “Shuffling, fast and slow: scalable analytics on serverless infrastructure”. In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI’19)*. 2019, pp. 193–206.

[75] *Recommendation to disable huge pages for MongoDB.* 2019. URL: https://docs.mongodb.com/manual/tutorial/transparent-huge-pages/ (visited on 01/01/2024).

[76] *Redis*. https://redis.io/.

[77] David Reinsel, John Gantz, and John Rydning. "Data age 2025: the evolution of data to life-critical don't focus on big data; focus on the data that's big". In: *International Data Corporation (IDC) White Paper* (2017).

[78] Erik Riedel, Garth Gibson, and Christos Faloutsos. "Active storage for large-scale data mining and multimedia applications". In: *Proceedings of 24th Conference on Very Large Databases*. Citeseer. 1998, pp. 62–73.

[79] Erik Riedel and Garth A Gibson. "Active Disks: Remote Execution for Network-Attached Storage (CMU-CS-97-198)". In: (1997).

[80] Raul Saiz-Laudo. "Reducing Network Overhead on Personal Cloud Systems". In: *3rd URV Doctoral Workshop in Computer Science and Mathematics*. Ed. by Sergio Gómez and Aïda Valls. Tarragona, Spain: Publicacions URV, Nov. 2016, pp. 27–31. ISBN: 978-84-8424-495-0.

[81] Raul Saiz-Laudo and Marc Sánchez-Artigas. "Egeon: Software-Defined Data Protection for Object Storage". In: *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 2022, pp. 99–108. DOI: `10.1109/CCGrid54584.2022.00019`.

[82] Raúl Sáiz-Laudó, Marc Sánchez-Artigas, and Pedro García-López. "RSD: Rate-Based Sync Deferment for Personal Cloud Storage Services". In: *IEEE Communications Letters* 21.11 (2017), pp. 2384–2387. DOI: `10.1109/LCOMM.2017.2731848`.

[83] Josep Sampé, Pedro García-López, and Marc Sánchez-Artigas. "Vertigo: Programmable Micro-controllers for Software-Defined Object Storage". In: *2016 IEEE 9th International Conference on Cloud Computing (CLOUD'16)*. 2016, pp. 180–187.

[84] Josep Sampé et al. "Data-Driven Serverless Functions for Object Storage". In: *18th ACM/IFIP/USENIX Middleware Conference (Middleware'17)*. 2017, pp. 121–133.

[85] Josep Sampé et al. "Serverless Data Analytics in the IBM Cloud". In: *19th ACM/IFIP Middleware Conference Industry (Middleware'18)*. 2018, pp. 1–7.

[86] Marc Sánchez-Artigas et al. "Primula: A Practical Shuffle/Sort Operator for Serverless Computing". In: *21st International Middleware Conference Industrial Track (Middleware'20)*. 2020, pp. 31–37.

[87] *SeaFile*. SeaFile. 2024. URL: `https://www.seafile.com/en/home/` (visited on 01/01/2024).

[88] Mark Seeger. *GetPut benchmarking suite*. `https://github.com/markseger/getput`. 2021.

[89] Michael A. Sevilla et al. "Malacology: A Programmable Storage System". In: *EuroSys*. 2017, pp. 175–190.

[90] Hossein Shafagh et al. "Secure Sharing of Partially Homomorphic Encrypted IoT Data". In: *15th ACM Conference on Embedded Network Sensor Systems (SenSys'17)*. 2017.

[91] Seungwon Shin et al. "FRESCO: Modular Composable Security Services for Software-Defined Networks." In: *20th Annual Network and Distributed System Security Symposium NDSS'13*. 2013.

[92] Matt Sidley and Sally Guo. *AWS re:Invent 2021 - Deep dive on Amazon S3*. Youtube. 2021. URL: `https://www.youtube.com/watch?v=v3HfUNQ0JOE` (visited on 01/01/2024).

[93] Jane Silber. *Shutting down Ubuntu One file services*. canonical. 2014. URL: `https://canonical.com/blog/shutting-down-ubuntu-one-file-services` (visited on 01/01/2024).

[94] Ioan Stefanovici et al. "sRoute: treating the storage stack like a network". In: *USENIX FAST'16*. 2016, pp. 197–212.

[95] Stanley YW Su and G Jack Lipovski. "CASSM: A cellular system for very large data bases". In: *Proceedings of the 1st International Conference on Very Large Data Bases*. 1975, pp. 456–472.

[96] *SugarSync*. SugarSync. 2024. URL: `https://www1.sugarsync.com` (visited on 01/01/2024).

[97] Theodoros Theodoropoulos et al. “Cloud-based XR services: A survey on relevant challenges and enabling technologies”. In: *Journal of Networking and Network Applications* 2.1 (2022), pp. 1–22.

[98] Eno Thereska et al. “IOFlow: a software-defined storage architecture”. In: *ACM SOSP’13*. 2013, pp. 182–196.

[99] U.S. Department of Health & Human Services. *COVID-19 Reported Patient Impact and Hospital Capacity by Facility*. https://healthdata.gov/Hospital/COVID-19-Reported-Patient-Impact-and-Hospital-Capa/anag-cw7u. Apr. 2021.

[100] *V8*. URL: https://v8.dev (visited on 01/01/2024).

[101] Anjo Vahldiek-Oberwagner et al. “Guardat: Enforcing Data Policies at the Storage Layer”. In: *Tenth European Conference on Computer Systems (EuroSys ’15)*. 2015.

[102] Frederik Vercauteren. “Optimal Pairings”. In: *IEEE Transactions on Information Theory* 56.1 (2010), pp. 455–461.

[103] W3C. *XML Path Language*. https://www.w3.org/TR/xpath-30/. 2020.

[104] Frank Wang, Ronny Ko, and James Mickens. “Riverbed: Enforcing User-defined Privacy Constraints in Distributed Web Services”. In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 2019, pp. 615–630.

[105] Liang Wang et al. “Peeking Behind the Curtains of Serverless Platforms”. In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 2018, pp. 133–146.

[106] Noah Watkins and Michael Sevilla. “Using Lua in the Ceph distributed storage system”. In: *Proceedings of the Lua Workshop*. 2017, pp. 16–17.

[107] Sage A. Weil et al. “Ceph: A Scalable, High-performance Distributed File System”. In: *USENIX OSDI*. 2006, pp. 307–320.

[108] www.fcc.gov. *Measuring Fixed Broadband - Tenth Report | Federal Communications Commission*. https://www.fcc.gov/reports-research/reports/measuring-broadband-america/measuring-fixed-broadband-tenth-report.

[109] www.opensignal.com. *Benchmarking the global 5G user experience – October update | Opensignal*. `https://www.opensignal.com/2020/10/13/benchmarking-the-global-5g-user-experience-october-update`.

[110] Tian Zhang et al. "Narrowing the gap between serverless and its state with storage functions". In: *Proceedings of the ACM Symposium on Cloud Computing*. 2019, pp. 1–12.