BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

MONITORING THE OPENSTACK SWIFT OBJECT STORE USING BEANSTALK EVENTS

SLEDOVÁNÍ OBJEKTOVÉHO ÚLOŽIŠTĚ OPENSTACK SWIFT POMOCÍ BEANSTALK UDÁLOSTÍ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. NEMANJA VASILJEVIĆ**
AUTOR PRÁCE

SUPERVISOR **RNDr. MAREK RYCHLÝ, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2022

Brno University of Technology
Faculty of Information Technology

Department of Information Systems (DIFS) Academic year 2021/2022

Master's Thesis Specification

Student: **Vasiljević Nemanja, Bc.**
Programme: Information Technology and Artificial Intelligence
Specialization: Information Systems and Databases
Title: **Monitoring the OpenStack Swift Object Store Using Beanstalk Events**
Category: Databases
Assignment:

1. Explore OpenStack Swift object storage, especially its architecture and activities. Study also MinIO object storage. Learn about the object storage OpenIO Software Defined Storage and in which way it uses Beanstalk to monitor and distribute events over the storage.
2. Design a service that will monitor activities in OpenStack Swift and, following the pattern of OpenIO, publish Swift events using the Beanstalk protocol. Consider also the ability to monitor and publish events from MinIO.
3. After consulting with the supervisor, implement the proposed service over OpenStack Swift/MinIO so that compatibility with OpenIO is guaranteed. For verification, also implement a sample client that will be able to subscribe to events using Beanstalk from both OpenIO and OpenStack Swift/MinIO.
4. Test the solution, evaluate and discuss the results. Publish the resulting software as open-source.

Recommended literature:

- Raúl GRACIA-TINEDO, Josep SAMPÉ, Gerard PARÍS, Marc SÁNCHEZ-ARTIGAS, Pedro GARCÍA-LÓPEZ and Yosef MOATTI: Software-defined object storage in multi-tenant environments. *Future Generation Computer Systems*. 99, 54-72, 2019. ISSN 0167-739X. Available at [https://doi.org/10.1016/j.future.2019.03.020]
- OpenStack Docs: Object Storage monitoring. The OpenStack project [online]. 2021 [seen 2021-09-29]. Available at [https://docs.openstack.org/swift/ussuri/admin/objectstorage-monitoring.html]
- Send notifications on PUT/POST/DELETE requests - swift-specs 0.0.1.dev82 documentation. OpenStack Foundation [online]. 2016 [seen 2021-09-29]. Available at [https://specs.openstack.org/openstack/swift-specs/specs/in_progress/notifications.html]

Requirements for the semestral defence:

- Items 1 and 2 finished and item 3 in progress.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Rychlý Marek, RNDr., Ph.D.**
Head of Department: Kolář Dušan, doc. Dr. Ing.
Beginning of work: November 1, 2021
Submission deadline: May 18, 2022
Approval date: October 21, 2021

Abstract

The goal of this thesis is to create software that can monitor and publish event notifications from Openstack Swift and OpenIO Software-Defined Storage (SDS) to a Beanstalk queue. In addition, this thesis also proposes a solution for publishing event notifications from MinIO to a Beanstalk queue.

In order to accomplish this goal, new middleware is proposed that can be run inside a pipeline of Proxy Server in OpenStack Swift and inside the pipeline of OIO-Swift inside OpenIO SDS.

Proposed middleware allows users to specify if they are interested in publishing event notifications for specific objects/containers using metadata. For example, users can specify a set of rules involving object properties, such as name (prefix, suffix) and size, and only events satisfying those rules will be published.

The contribution of this thesis is unique software capable of event monitoring from both OpenIO SDS and Openstack Swift.

Abstrakt

Cílem této práce je vytvořit software, který je schopen monitorovat a publikovat notifikace o události z Openstack Swift i z OpenIO Software-Defined Storage (SDS) do fronty Beanstalk. Tato práce také navrhuje řešení pro publikování notifikaci o událostech z MinIO do fronty Beanstalk.

K dosažení tohoto cíle je navržen nový middleware, který lze spouštět uvnitř pipeline proxy serveru v OpenStack Swift a uvnitř pipeline OIO-Swift serveru v OpenIO SDS.

Navržený middleware umožňuje uživatelům určit, zda mají zájem o publikování notifikaci o události pro konkrétní objekty/kontejnery pomocí metadat. Uživatel může specifikovat sadu pravidel zahrnující vlastnosti objektu, jako je název (prefix, přípona, podřetězec) a velikost, a budou publikovány pouze události splňující tato pravidla.

Přínosem této práce je unikátní software schopný monitorování událostí z OpenIO SDS i Openstack Swift.

Keywords

OpenIO Software-Defined Storage, Openstack Swift, MinIO, Beanstalk queue, Event monitoring, Event notification, Amazon S3 event notification, Object storage

Klíčová slova

OpenIO Softwarově definované úložiště, Openstack Swift, MinIO, Beanstalk fronta, Monitorování událostí, Oznámení o událostech, Amazon S3 oznámení o události, Objektové úložiště

Reference

VASILJEVIĆ, Nemanja. *Monitoring the OpenStack Swift Object Store Using Beanstalk Events*. Brno, 2022. Master’s thesis. Brno University of Technology, Faculty of Information Technology. Supervisor RNDr. Marek Rychlý, Ph.D.

Rozšířený abstrakt

Současný stav je že uživatelé v OpenStack Swift nemají možnost získat informace když se provede určitá událost v častí objektovém úložišti které vlastni, nebo ke kterým máji přístupová práva. Například, OpenStack Swift neumožňuje odeslat notifikaci uživateli když dojde k smazání, nahrávání nebo čtení objektu.

Výsledkem teto práce je program pojmenovaný jako ENOSS - Event Notifications in OpenStack. ENOSS je implementován ve tvaru Python WSGI middleware a je zaražen do popelíne výchozí brány(gateway) objektového úložišta Swift a OpenIO SDS. Toto umístění umožňuje ENOSS programu přístup ke všem vstupním (uživatelské žádostí) a výstupním (odpovědí objektových úložišť) informaci.

Program umožňuje každém uživateli specifikovat o které události má zájem, tj které události se máji publikovat. Middleware silně využívá metadata vyšších vrstev (container a account). Konfigurace definující které události se máji publikovat se ukládá do systémových metadata, která jsou přístupna jen v interních procesech objektového úložišta. ENOSS rozšiřuje Swift API o koncový body pro vkládání nových konfiguraci a čtení uložených.

Uživatel může specifikovat typ události (čtení, aktualizace, zápis, mazání) a/nebo může definovat sadu filtrovacích pravidel která musí byt splněna aby událost byla publikována. Momentálně ENOSS podporuje filtrovací pravidla na prefix, sufix, maximální velikost, minimální velikost, typ internetového media, uživatelé a HTTP kód odpovědí objektového úložišta. ENOSS umožňuje publikaci notifikaci do následujících cílů: Beanstalkd fronta, Apache Kafka a Elasticsearch. Uživatelům je umožněno vybrat do kterého cíle se notifikace má odeslat.

Klíčova vlastnost ENOSS programu je podpora vlastních cílů, filtrovacích pravidel a obsahu notifikace. ENOSS specifikuje rozhraní a pravidla, která pokud se dodržuji vedou ke snadné integraci nové vlastni třídí s ENOSS systémem.

Další klíčová vlastnost je kompatibilita s Amazon S3 Event Notifications. Specifikování událostech které se májí publikovat je realizováno pomoci konfigurace která je kompatibilní s S3. Zároveň, výchozí struktura a obsah notifikace je taky kompatibilní s AWS S3.

Výchozí nastaveni ENOSS programu publikuje jenom úspěšně ukončené událostí. Oproti AWS S3, ENOSS lze konfigurovat aby publikovat události které nebyly úspěšně ukončené (např. neoprávněný přístup, interní chyba).

Z analýzy chování ENOSS programu lze vyvést že pří zjištění konfiguraci notifikaci, uložených ve vyšších vrstev v architektuře, ENOSS má všechna potřebná data v cache pamětí. To znamená ze ENOSS nemá dopad na latence žádostí uživatelů který nemají nastavené notifikace. Při vytvoření obsahu notifikace a provedeni filtru ENOSS nemusí mít všechny nutné informace dostupné, a musí přečíst data z objektového úložišti, což zvyšuje latence. Ovšem získaná data z objektového úložišti jsou vložena do cache pamětí, a diky tomu lze dojit k maximálně jednom dodatečném čtení dat z objektového úložišti pří publikování události.

Výsledný program má velké množství použití. ENOSS umožňuje detekci anomálii (vyfiltrovat události které mají návratový kód 5xx), odcizeni dat (notifikace když došlo k přístupu dat uživatelem který by nemel mít právo přístupu), prevence odcizeni dat (notifikace filtrující událostí s návratovým kódem 401) a postprocessing (např odeslání metadat do Elasticsearch a následné vyhledavání objektu pomoci metadata).

Monitoring the OpenStack Swift Object Store Using Beanstalk Events

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of RNDr. Marek Rychlý Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Nemanja Vasiljević
May 18, 2022

Acknowledgements

I would like to thank my thesis supervisor, RNDr. Marek Rychlý Ph.D. for professional leadership, time, willingness and valuable advice. The door to RNDr. Marek Rychlý Ph.D. office was always open whenever I ran into a problem or had questions regarding my research or writing.

I would also like to thank Mr. Christian Schwede, a principal engineer working at Red Hat, core reviewer and contributor to Swift, for providing me with additional information and guidance.

Finally, I would like to express my gratitude to my parents and my family for providing me with support and continuous encouragement throughout my years of study and thought process of writing this thesis.

Contents

1 Introduction **3**

2 Background **5**
- 2.1 Object storage . . . 5
- 2.2 Software-Defined storage . . . 7
- 2.3 Beanstalk queue . . . 8
  - 2.3.1 Beanstalkd elements . . . 9
  - 2.3.2 Job Lifecycle . . . 9
  - 2.3.3 Key characteristics . . . 10
- 2.4 Event notifications . . . 11
  - 2.4.1 CloudEvents . . . 11
  - 2.4.2 Amazon S3 event notifications . . . 12

3 Object storages **14**
- 3.1 OpenIO SDS . . . 14
  - 3.1.1 Key characteristics . . . 14
  - 3.1.2 Data organization . . . 15
  - 3.1.3 Serverless computing . . . 16
  - 3.1.4 OIO-Swift . . . 18
- 3.2 OpenStack Swift . . . 19
  - 3.2.1 Key characteristics . . . 19
  - 3.2.2 Data model . . . 20
  - 3.2.3 Middlewares . . . 21
- 3.3 MinIO . . . 24
  - 3.3.1 Introduction . . . 24
  - 3.3.2 Key features . . . 24
  - 3.3.3 Architecture . . . 25
  - 3.3.4 Event notifications . . . 27

4 Solution draft **28**
- 4.1 Current state . . . 28
  - 4.1.1 OpenIO SDS . . . 28
  - 4.1.2 OpenStack Swift . . . 28
  - 4.1.3 MinIO . . . 29
- 4.2 Middleware for OpenStack Swift and OpenIO SDS . . . 29
  - 4.2.1 Location . . . 29
  - 4.2.2 Design . . . 29
  - 4.2.3 Structure of published event . . . 33

4.2.4 Event Notification configuration . . . 35
4.3 Proxy for MinIO . . . 35

5 Implementation **38**
5.1 ENOSS . . . 38
5.1.1 Middleware . . . 38
5.1.2 Notification configuration . . . 42
5.1.3 Filters . . . 43
5.1.4 Notification payload . . . 44
5.1.5 Destinations . . . 45
5.1.6 Custom filters/payloads/destinations . . . 46
5.1.7 OpenIO SDS compatibility . . . 47
5.2 MinIO proxy . . . 48

6 Testing, benchmark and possible applications **49**
6.1 Testing . . . 49
6.1.1 Unit tests . . . 49
6.1.2 Functional tests . . . 50
6.2 Performance analysis . . . 51
6.3 Experiments . . . 52
6.4 Use cases and applications . . . 57

7 Conclusion **59**

Bibliography **61**

A Contents of the included storage media **64**

B Repository and Usage Guide **65**

C Excel@FIT Article **67**

Chapter 1

Introduction

In the current world, cloud computing has become the most popular way of delivering different services on the Internet. One of the most popular cloud services is cloud storage, allowing users to store data in remote locations maintained by a third party. Based on how cloud storage manages data, cloud storage can be divided into three types: Block storage, File storage, and Object storage. Object storage manages data as objects, and each object typically includes data itself and some additional information stored in object metadata. Since data are stored in remote locations, to which users do not have direct and complete access, some users or external services might want to receive information about specific events (for example, change of content) in storages where their data are located.

The importance of this thesis is to provide event information to users in OpenIO SDS and OpenStack Swift, which will allow users to react to those events, create more sophisticated backend operations and postprocessing, or possibly prevent/detect unwanted actions. In addition, providing event notifications will allow users to have a better picture of what is going on in their storage and improve monitoring in these object storages.

There were two attempts[20][22] to solve this issue within OpenStack Swift which were not officially accepted, and their solution is outdated. So currently, there is no official solution for publishing event notifications in OpenStack Swift nor OpenIO Software-Defined Storage (from now on SDS).

My interest in this topic stems from its possible impact on the extensive amount of users that OpenStack Swift and OpenIO have. Furthermore, I have always wanted to contribute to open-source projects. The possibility to improve user experience in OpenStack Swift and OpenIO SDS and allow these storags to be even more competitive against commercial storage (Amazon, Google, ...) is another reason why I choose this topic.

This thesis aims to create a program/middleware which will publish event notifications to user-specified destinations. One of the supported destinations will be the Beanstalk queue, but the program can be easily configured to support other destinations (for example, Kafka) using a predefined interface. The proposed program will allow users to specify, using object metadata (such as name prefix/suffix and object size) and type of event, which event notification should be published. The program will be able to run within OpenStack Swift and OpenIO SDS. This thesis will strive to find such a solution that could be officially accepted as part of OpenStack Swift and OpenIO SDS.

This work consists of six chapters. Chapter 1 introduces the motivation, objectives, and proposed solutions of this work. Chapter 2 briefly describes the technologies and general areas that this work relates to. Chapter 3 covers object storage used in this work. Chapter introduces OpenIO SDS and describes its data organization and key services providing

events. Chapter 3 introduces OpenStack object storage Swift, its data model, server processes and describes middlewares within OpenStack Swift. Lastly, chapter 3 briefly covers MinIO storage and how publishes event notifications. Chapter 4 describes the current state of event notifications in OpenIO SDS, OpenStack Swift, and MinIO, proposes a solution for publishing event notifications in OpenIO SDS and OpenStack Swift, and a solution for publishing event notifications from MinIO to Beanstalk queue. Chapter 5 describes the process of implementation, validation notification configurations, supported filters, notification payloads, and destinations. Furthermore, chapter 5, explains compatibility issues between OpenIO SDS and Swift and explains the implementation process of the proxy program allowing publishing notifications from MinIO to beanstalkd. Chapter 6 describes the testing process, benchmarking, and applied experiments. Last chapter 7 describes the overview of the achieved results.

Chapter 2

Background

This chapter introduces Object storage, its core concepts, and the underlying technologies. After introducing the Object storage, for sufficient understanding of this master thesis topic, it is essential to explain how Software-defined storage manages data and what event types can occur inside. The last part of this chapter describes the concept of event notifications, why they are essential, and the current interfaces for publishing them to users.

2.1 Object storage

Object storage, also known as *object-based storage (OBS)*, handles data as objects instead of the hierarchical methods used in file systems[37]. The object storage is designed to handle data as whole objects, making it an ideal solution for any unchanging data. Data in object stores are changed by replacing objects or files, and therefore object stores are the preferred mechanism for storing such files[38].

Key concepts

Key concepts of object storage are[39]:

- **Objects** - An object typically consist of user data and metadata uploaded to object storage.
- **Containers/Buckets** - represents logical abstraction used to provide a data container in object storage. An object with the same name in two different containers represents two different objects. This concept segregates data using bucket ownership and a combination of public and secret keys bound to object storage accounts, allowing users and applications to manipulate with authorized data for specific types of manipulation (read/write/update).
- **Metadata** - Additional information about data, such as date of creation and last modification, size, and hash.
- **Access Control Lists(ACLs)** - used as primary security construct in object storage, stored in account or bucket level, and allows owners to grant permissions for certain operations based on UUID, email, ...
- **Object Data protection** - two primary data protection schemes in object storage are Replication and Erasure Coding.

Replication is a method used to ensure data resilience. Data are copied into multiple locations/disks/partitions. In case of failure, data are used from a secondary copy to recreate the original copy or as a primary copy.

Erasure coding is a process through which the data is separated into fragments. Then fragments are expanded and encoded with redundant pieces and stored across different storage devices. Erasure coding adds redundancy and allows object storage to tolerate failures.

Object data

With object storage techniques, each object contains[39]:

- **Data** - user-specified data that needs to be stored in persistent storage. Such data can be binary data, text file, image, etc.
- **Metadata** - additional data that describe objects data. Metadata can be divided into two types: *Device-managed metadata* is additional information maintained by a storage device and used as part of object management in physical storage[37]. The second type is *Custom metadata*, where users can store additional information in key and value pairs. In object storage, metadata is stored together with the object.
- **A universally unique identifier (UUID)** - This ID, created using a hashing process based on object name and other additional information, is assigned to each object in object storage. Using ID object storage systems can tell apart objects from one another. ID is also used to extract data in a system without knowing their physical location/drive and offset.

Access to object storage

Object storage services provide a `RESTful` interface [41] over `HTTP` protocol to store and access objects. This approach allows users to create, read, delete, update, or even query objects anytime and anywhere simply by referencing UUID (or using specific attributes for querying), usually with a proper authentication process. The most popular interfaces for communicating with object storages are *Amazon S3 (Simple Storage Service) API* and *OpenStack Swift API*.

Pros and cons of object storage

Pros:

- Capable of handling a large amount of unstructured data
- Reduced TCO and cheap COTS - Object storage is designed to utilize cheap COTS(Commercial off-the-shelf) components. As a result Total Cost of Ownership(TCO) is lower than owning homemade Network-Attached Storage(NAS)[38].
- Unlimited scalability - Since object storages are built on distributed systems, they scale very well compared to traditional storages, where they often have an upper limit[34].
- Wide-open metadata - allows users to store custom metadata and the possibility of creating metadata-driven policies, such as compression and tiering.

Cons:

- No in-place update - object must be manipulated as a whole unit.
- No locking mechanism - object storage does not manage object-level locking, and it is up to applications to solve concurrent PUT/GET.
- Slower - this makes object storages a poor choice for applications that need rapid and frequent access to data.

2.2 Software-Defined storage

Software-Defined Storage(SDS) is a storage architecture that separates software storage from hardware allowing greater scalability, flexibility, and control over the data storage infrastructure. With the growth of *Software-Defined Networks(SDN)* and the need for *Software-Defined Infrastructure(SDI)*, which aims to virtualize network resources and separate the control plane from the data plane, this principle was needed to be applied on Object storage as well[32].

To overcome limitations of traditional storage infrastructures, the Software-Defined Storage is imposed as a proper solution to simplify data and configuration management while improving the end-to-end control functionality of conventional storage systems[36]. Furthermore, while traditional storages like storage area networks (SAN) and network-attached storage (NAS) provides scalability and reliability, SDS provides it with much lower cost by utilizing industry-standard or x86 system and therefore removing dependency on expensive hardware[26].

Principles

There is no clear definition on criteria for defining software-defined storage, although several fundamental principles can be deducted[30]:

- **Scale-out** - SDS should enable low-cost horizontal scaling (by adding new commodity hardware to existing infrastructure) compared to vertical scaling with more powerful (and expensive) hardware.
- **Customizable** - SDS should offer system storage customization to meet specific storage QoS requirements. This will allow users to choose storage solution based on their requirements/performance and avoid unnecessary overpaying.
- **Automation** - once QoS is defined process of deployment and monitoring on object storage should be automated and done without the need for human resources.
- **Masking** - SDS can mask an underlying storage system and distributed system as long as they provide common storage API and meet required QoS. SDS can offer Block or File API even though data are saved in object storage (like Ceph[1] does).
- **Police Management** - SDS Software must manage storage according to specified policies and QoS requirements despite being in multi-tenant space. SDS must be capable of handling failures and autoscale in case of change in workloads.

[1] Ceph - distributed object, block, and file storage platform https://ceph.io

Architecture

As previously described, the main characteristic of SDS is to separate storage functions into a *control plane* and *data plane*.

Control plane - the control plane is a software layer with the main goal to virtualize storage resources. The control plane manages data provision and provides orchestration of data services across object storage. Solutions that are part of the control plane allow policy automation, analytics and optimization, backup and copy management, security, and integration with the API services, including other cloud provider services[31].

Data plane - the data plane encompasses the infrastructure where data is processed. The data plane provides an interface to the hardware infrastructure and defines how the storage is accessed. It provides access methods to storage, such as *Block I/O* (for example, iSCSI), *File I/O* (NFS, SMB, or Hadoop Distributed File System (HFDS)), and *object storage*. It defines storage management functions, such as virtualization, RAID protection, tiering, encryption, compression, and data deduplication that can be requested by the control plane[31].

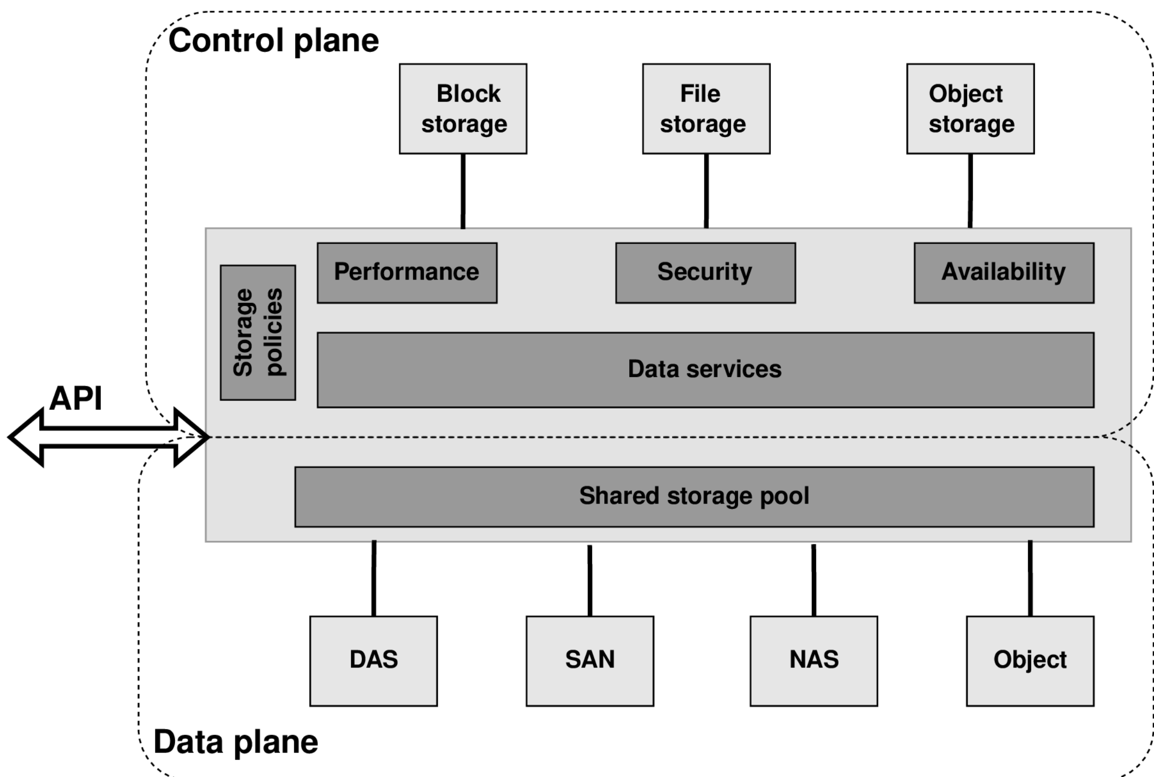

Figure 2.1: SDS data and control plane (source: [23], remade).

2.3 Beanstalk queue

Beanstalk queue or shorter **beanstalkd** is a fast, simple and lightweight working queue[3]. The primary use case is to manage workflow between different parts of workers of application

through working queues and messages. Beanstalkd was developed for the need of Facebook application in order to reduce average response time[3]. Provided by simple protocol design, heavily inspired by Memcached, implemented in programming language C, Beanstalkd offers lean architecture, which allows it to be installed and used very simply, making it perfect for many use cases[19].

2.3.1 Beanstalkd elements

Beanstalkd is a priority queue with server-client architecture. The server represents queues where jobs are saved based on priority. Beanstalkd architecture is composed of several components:

- **Jobs** - tasks stored by the client
- **Tubes** - used for storing tasks, each tube contains a ready queue and a delay queue.
- **Producer** - creates and sends jobs to beanstalkd using command `put`.
- **Consumer** - process „listening" on an assigned tube, reserves and consumes jobs from the tube.

2.3.2 Job Lifecycle

Each job is uniquely assigned to one worker at a time. The client creates a job and inserts it into a beanstalkd tube using the `put` command. While being in the tube, the job can be in next states[4]:

- `Ready` - the task is free and can be executed immediately by the Consumer.
- `Delayed` - the task has assigned delay time that needs to expire before execution. After delay time expires, beanstalkd will automatically change its state to `Ready`.
- `Reserved` - the task is reserved and is being executed by the *Consumer*. Beanstalkd is responsible for checking whether the task is completed in time (**TTR** - Time to run).
- `Buried` - reserved task, the task will not be removed nor executed until the client decides. This state is often used for further inspection in debugging process when failure or undefined behavior occurs during task execution.
- `Deleted` - the task is deleted from the tube, beanstalkd no longer maintains these jobs.

Figure 2.2 describes the life cycle of a job in a beanstalkd tube. Job is created by *Producer* using put command. Beanstalkd allows the *Producer* to add delay time before the task is ready for execution, setting the job state to `Delayed`. After delay time expires, beanstalkd will automatically change job state to `Ready`. The Producer can specify job priority and jobs with the `Ready` state are stored in the priority queue. A job with the biggest priority is reserved and executed by a *Consumer*. After successfully executing the task, the *Consumer* will delete the job from beanstalkd. If some error occurs, the *Consumer* can **bury** the task. The Consumer can decide that he is not interested in completing the reserved task. Using the `release` command (with optional delay) job state will be changed

back to `Ready` (or `Delay` if delay exists). Jobs with the `Burried` state will not be touched by the beanstalkd server until the client „`kicks`" them to `READY` state.

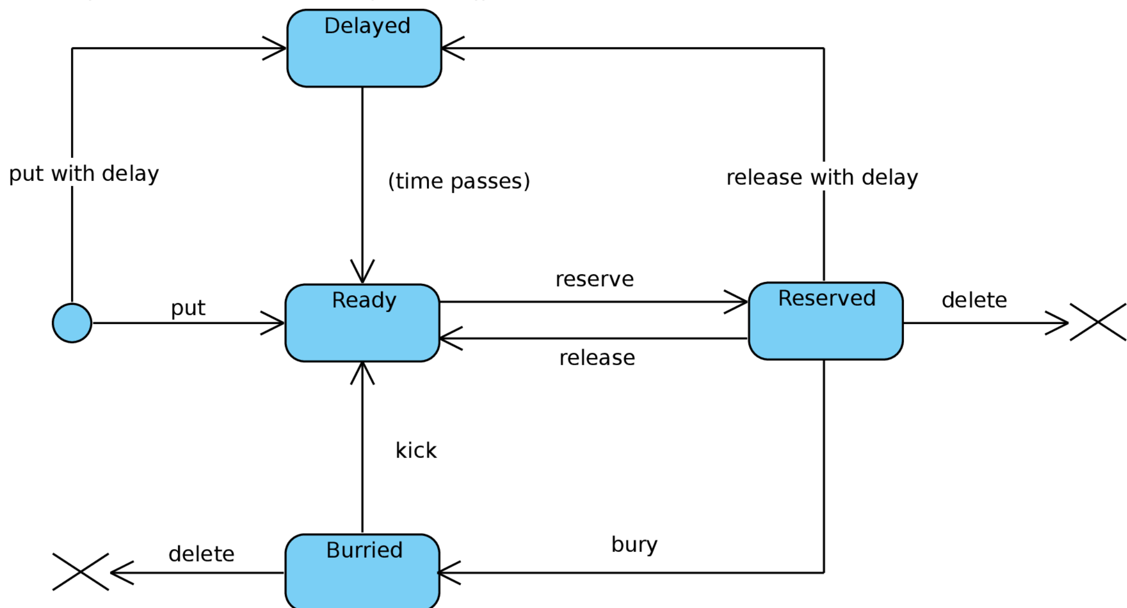

Figure 2.2: State machine diagram of job in Beanstalkd tube.

2.3.3 Key characteristics

Key beanstalkd characteristics are:

Asynchronous - beanstalkd allows producers to put jobs in the queue, and workers can process them later.

Distributed - in the same way as *Memcached*[2], beanstalkd can be distributed, although this distribution is handled by clients. The beanstalkd server does not know anything about other beanstalkd running instances.

Persistent - beanstalkd offers support for persistent jobs during which all jobs are written to binlog. In case of a power outage, after restarting a beanstalkd instance, it will recover jobs content from the logs.

Not secured - beanstalkd is designed to be run in a private/secure network. Therefore it **does not support authentication or authorization**.

Scalability - beanstalkd can be scaled horizontally, although it must be done on the client side, where each client would connect to multiple servers and then use specific algorithms(e.g., Round-robin) to switch between the different servers.

[2]Memcached - in-memory key-value store `https://memcached.org/`

2.4 Event notifications

An event is a runtime operation executed by a software element, representing a significant change or occurrence in a system. Event is created in order to make some information available to other software elements not specified by the operation[40].

Event notification is a message created by a system in order to notify other parts of the system that an event has taken place[27]. Event notifications are usually used for monitoring and asynchronous job processing.

In object storage, event notifications are used to notify users or tenants about specific changes and occurrences in their bucket or account. Typical event notifications include creating new (or updating existing) objects in the bucket. In addition, most object vendors offer **publish/subscribe** notifications, allowing users to subscribe to certain types of event notifications using predefined rules. Information about rules specifying event notifications is usually stored in the upper-level metadata (bucket or account).

2.4.1 CloudEvents

Publishers tend to describe event data differently due to non-existing standards or formats. The lack of a common way to describe events means developers have to learn how to handle events from each event source. To solve this problem, CloudEvents was created.

CloudEvents is a specification for describing event data in common way[5] hosted by CNCF[3]. CloudEvents goal is to dramatically simplify event specification and delivery across services, platforms and beyond. CloudEvents has been integrated by many popular object storage vendors, such as Oracle Cloud, IBM Cloud Code Engine, Azure, Google Cloud, etc.

Attributes in CloudEvents specification can be divided into three categories:

Required attributes - set of attibures that are required to be included in all events[6]:

- id (string) - event identifier, must not be empty.
- source (URI-reference) - identifies context in which event occured, must not be empty.
- specversion (string) - the version of CloudEvents specification, must not be empty.
- type (string) - value describing the type of occurred event. Often this attribute is used for policy enforcement, routing and monitoring.

Event data attirbutes - attibures containing and describing event data:

- datacontenttype (string) - content type of data value (allows data to carry any type of content).
- dataschema (URI) - identifies the schema that data adheres to.
- data - data payload

[3]CNCF - Cloud Native Computing Foundation https://www.cncf.io/

Optional attributes :

- time - timestamp
- subject (string) - the subject of the event in the context of the event producer.
- extension attributes - custom attibutes allowing external systems to attach metadata to an event.

```
{
    "specversion" : "1.0",
    "type" : "com.github.pull_request.opened",
    "source" : "https://github.com/cloudevents/spec/pull",
    "subject" : "123",
    "id" : "A234-1234-1234",
    "time" : "2018-04-05T17:31:00Z",
    "comexampleextension1" : "value",
    "comexampleothervalue" : 5,
    "datacontenttype" : "text/xml",
    "data" : "<much wow=\"xml\"/>"
}
```

Listing 2.1: Example of event described using CloudEvents specification in JSON format.

2.4.2 Amazon S3 event notifications

Amazon Simple Storage Service (S3) is one of the most popular cloud object storages providing a REST web service interface. Amazon S3 is reliable, scalable, commercial and one of the most popular object storage that manages Web-Scale computing by itself[33]. As a result, Amazon S3 has a big impact on object storage and most other object storage vendors crated compatible **S3 API** for their services.

One of the monitoring features that Amazon S3 provides is **Event Notification**, which offers users to receive notifications when certain events happen in their S3 bucket. To enable such notifications, users need to create a notification configuration that identifies which events Amazon S3 should publish[2]. Notifications are configured at the bucket level and then applied to each object in the bucket.

Amazon S3 provides limited event destinations to which event notification messages can be send[1]:

- *Amazon Simple Notification Service (Amazon SNS)* - flexible, fully managed push messaging service, can be used to send messages to mobile phones or distributed services.
- *Amazon Simple Queue Service (Amazon SQS)* queues - reliable and scalable hosted queues for storing messages as they travel between computers.
- *AWS Lambda* - serverless, event-driven compute service. Lambda can run custom code in response to the Amazon S3 bucket event (if the lambda function writes to the same bucket that triggers the notification, it can create an execution loop).
- *Amazon EventBridge* - serverless event bus service used to receive events from AWS. It allows users to define rules to match events and deliver them to defined targets.

By this date, Amazon S3 **does not support CloudEvents** specification and describes event data in its own way. Some of the event types that Amazon S3 can publish are displayed in table 2.1.

Event type	**Desription**
s3:TestEvent	after enabling the event notifications, Amazon S3 publishes a test notification to ensure that topic exist and bucket owner has permissions to publish specified topic.
s3:ObjectCreated:*	An object was created (regardless on operation).
s3:ObjectCreated:Put	An object was created by an HTTP PUT operation.
s3:ObjectCreated:Post	An object was created by HTTP POST operation.
s3:ObjectCreated:Copy	An object was created an S3 copy operation.
s3:ObjectCreated: CompleteMultipartUpload	An object was created by the completion of a S3 multi-part upload.
s3:ObjectRemoved:*	An object was removed (regardless on operation).
s3:ObjectRemoved:Delete	An object was deleted by HTTP DELETE operation.
s3:ObjectRemoved: DeleteMarkerCreated	An versioned object was marked for deletion.

Table 2.1: Subset of Amazon S3 Event Types [1]

Chapter 3

Object storages

3.1 OpenIO SDS

This section introduces OpenIO Software-defined storage, its key features, its data organization along with the underlying technologies. Furthermore, this section introduces Grid For Apps framework (3.1.3) and event publishing in OpenIO (3.1.3).

OpenIO Software-defined storage is open source object storage that is perfectly capable of traditional use cases (such as archiving, big data, cloud). However, at the same time, combined with Grid for Apps (3.1.3), it opens the door for users to create an application that needs much more sophisticated back-end operations. These applications include industrial IoT, machine learning and artificial intelligence, as well as any other applications whose workflow can benefit from automated jobs or tasks[25]. In addition, OpenIO SDS is event-driven storage with the ability to intercept events seamlessly and transparently to the rest of the stack.

3.1.1 Key characteristics

Hardware agnostic OpenIO SDS is fully software-defined storage capable of running on x86 or ARM hardware with minimal requirements. Cluster nodes can be different from each other, allowing different generations, types, and capacities to be combined without affecting a performance or efficiency[11]. OpenIO has built-in support for heterogeneous hardware allowing every node to be used at its maximum performance.

No SPOF architecture Every single service used to serve data is redundant from object chunks stored in a disc to the directory level, every information is duplicated. As a result, there is no single point of failure (SPOF) in the cluster and a node can be shut down without affecting overall availability or integrity[21].

Cluster organization Instead of a traditional cluster ring-like layout, OpenIO SDS is based on a grid of nodes 3.1. It is flexible and resource-conscious. Compared to other object storage solutions, cluster organization is not based on static data allocation that usually use Chord peer-to-peer distributed hash table algorithm. Instead, OpenIO SDS uses distributed directory for organizing data and metadata hash tables, which allows the software to attain the same level of scalability but with better and more consistent performance[11].

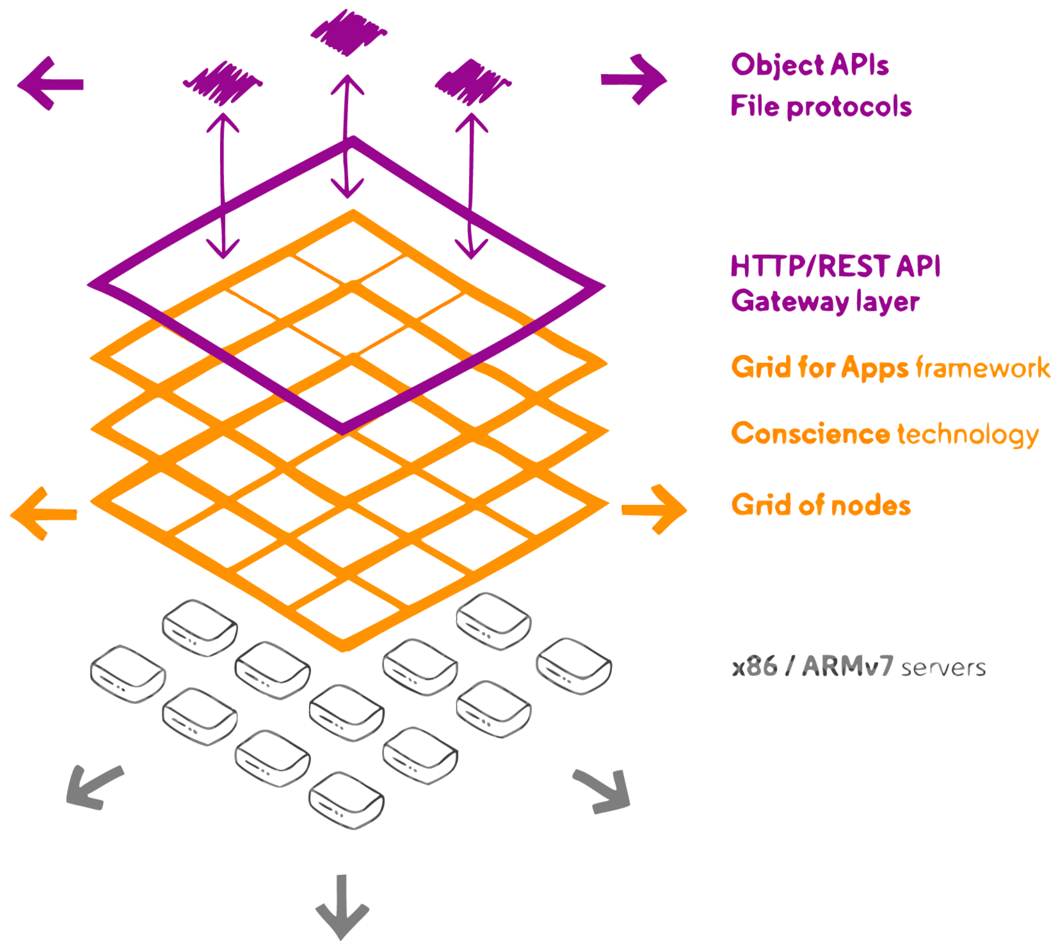

Figure 3.1: Layered view on OpenIO SDS architecture (source: [18]).

Tiering With tiering, OpenIO SDS offers users to configure a pool containing a group of hardware that can then be used to store specific types of objects. For example, users can create a pool of high-performance hard disks (e.g. SSDs) and use the pool to store objects that require low latency. This feature is realized by a mechanism called **storage policies**. Multiple storage policies can be defined in one particular namespace. Storage policies can also be used for specifying how many replicas should be created for a specific dataset[21].

ConsciousGrid ConsciousGrid is an OpenIO technology that uses real-time metrics from the nodes(CPU, I/O, capacity) automatically discover and place data in the most appropriate place. It provides **load balancing** and computes a score for each node and then provides weighted random selection[13].

3.1.2 Data organization

Multi-tenancy is one of the core concepts in OpenIO SDS. Data objects are stored within following hierarchy: `Namespace/Account/Container/Object` 3.2. Multiple namespaces can be configured in each cluster, providing multi-region/zone logical layouts for applications and segregated workloads depending on a tenant or geo-distribution need[12]. There is no

classic subdirectory tree. Instead, objects are stored in a flat structure in the container level. However, like many other object storages, there is a way to emulate a filesystem.

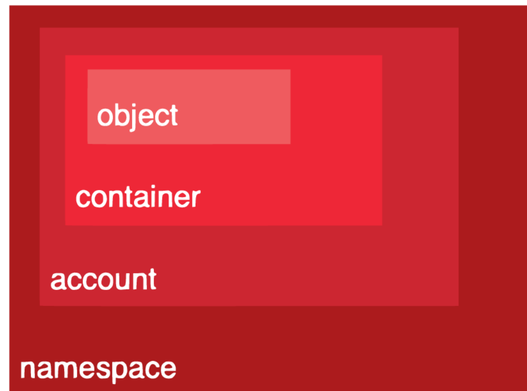

Figure 3.2: Object data organization in OpenIO SDS (source: [21]).

Namespace A coherent set of network services working together to run OpenIO's solutions. It hosts services and provides operations such as service configuration and monitoring.

Account An account usually represents a tenant and is the top level of data organization. Each account owns and manages a collection of containers. In addition, the account keeps track of namespace usage for each customer (i.e. bytes occupied by all of a customer's objects)[21].

Container Container represents an object bucket. Each container belongs to one (and only one) account and is identified by a unique name within the account. The container carries additional information specifying how to manage its objects (e.g. how to secure them)[21].

Object Object is the smallest data unit visible by a customer and represents a named BLOB with metadata. OpenIO SDS allows several objects to be stored in a container and are considered versions of the same object. Classic `API` operations (`PUT/GET/DELETE`) will be directed towards an object with the latest version. If the size of an object is larger than the specified limit at the namespace level, the object will be divided into chunks of data. This behavior allows capacity optimization as well as **distributed reads** that could be particularly useful for high-speed video streaming of large media[21].

3.1.3 Serverless computing

OpenIO offers Serverless computing in object storage cluster nodes using the framework Grid For Apps.

Grid For Apps Like Amazon AWS Lambda, OpenIO offers an **event-driven compute service** called Grid for Apps that works on top of OpenIO.

Grid for Apps intercepts all the events that happen in the storage layer, and based on user configuration, triggers specific applications or scripts to act on data (metadata) stored

in object storage[25]. The application is executed in cluster nodes and utilizes free unused resources available in the cluster. This improves efficiency (fewer data moving since object data are already available) and saves money (no need for external resources)[25].

Grid for Apps allows customers to perform operations such as metadata enrichment, data indexing and search (e.g. indexing metadata to Elasticsearch), pattern recognition, machine learning, data filtering, monitoring, etc[25].

Grid for Apps in OpenIO is realized using service `event-agent` and `beanstalkd` queue.

Event-agent Event-agent is an OpenIO service responsible for handling asynchronous jobs. It relies on beanstalkd backend to manage jobs. Event-agent key characteristics are[13]:

- Stateless
- CPU intensive
- Must be deployed on every server of the cluster

Every event that occurs in OpenIO is inserted in a beanstalkd tube. Event-agent is listening to the beanstalkd tube and consumes jobs from it. Consumers are produced using Eventlet Network Library [8]. The number of workers can be configured.

In event-agent, users can specify handlers for each type of event in event-handler.conf. Some of the event types in OpenIO are `storage.content.new` (e.g., new object in storage), `storage.container.deleted` (e.g., object has been deleted), etc.

Events handler is defined as a pipeline containing applications that will react to the event. In example 3.1, deleting an object will invoke `storage.content.deleted` event. Event-agent will handle the event using `content_cleaner` application, which deletes objects chunks from object storage.

OpenIO offers users to process events outside of the event-agent. In order to do that, users can use the application `notify` which will send an event to a specified beanstalkd tube. Then a user can create a custom consumer process that will execute the job from beanstalkd tube. An example of such configuration is displayed in listing 3.1.

```
[handler:storage.content.deleted]
pipeline = content_cleaner

[handler:storage.content.new]
pipeline = notify

[filter:content_cleaner]
use = egg:oio#content_cleaner

[filter:notify]
use = egg:oio#notify
tube = oio-rebuild
queue_url = ${QUEUE_URL}
```

Listing 3.1: Example of event-agent handler configuration

3.1.4 OIO-Swift

One of the key components in OpenIO SDS system is OIO-Swift service, which acts like gateway. Main responsibility of this component is to handle Swift/S3 user requests. Implementation of OIO-Swift is based on OpenStack Swift Proxy server. OIO-Swift key features include:

- Operations on objects, containers and accounts.
- Authentication support (using OpenStack Keystone).
- Metadata support - enables usage of system metadata.
- Swift Middleware support - allow running Swift middlewares within OpenIO SDS system.

3.2 OpenStack Swift

This section introduces OpenStack object storage (code name Swift) and describes its key features. Furthermore, this section elaborates OpenStack Swift architecture, introduces its main services and interfaces for communication with object storage.

OpenStack Swift is open-source object storage developed by Rackspace, a company that, together with NASA, created the OpenStack project. After becoming an open-source project, Swift became the leading open-source object storage supported and developed by many famous IT companies, such as Red Hat, HP, Intel, IBM, and others.

OpenStack Swift is a multi-tenant, scalable, and durable object storage capable of storing large amounts of unstructured data at low cost[29].

3.2.1 Key characteristics

Besides standard object storage characteristics (like scalability, durability, hardware agnostic, etc.), some of the keys OpenStack Swift characteristics:

Multi-regional capability OpenStack Swift has distributed architecture. Data can be distributed and replicated into multiple data centers, although the negative effect could be higher latency between them. Distribution can provide high availability of data and recovery site[29].

No SPOF With all data being replicated and distributed, there is no single proof of failure in OpenStack Swift architecture.

Developer-friendliness OpenStack Swift offers many built-in features that developers and users can use. some of the most interesting built-in features are[29]:

- **Automatically expiring objects** - Objects can be given expiration time, after which objects become invalid and deleted from object storage.
- **Quotas** - Storage limits can be configurated on container/account level.
- **Versioned objects** - User can store a new version of an object, while object storage keeps previous (older) versions.
- **Access control lists** - Users can configure access to their data to give or deny permission for reading or writing data to other users.

Middleware support - OpenStack Swift allows adding custom middlewares, which will be run directly on storage system[34]. This feature can be used for monitoring purposes, for example, informing users or other applications about new objects in storage using Webhook middleware.

Large object support - By default, OpenStack Swift has a limit on a single uploaded object, which is 5GB. However, using segmentation, the size of a single object can be virtually unlimited. This option offers a possible higher upload speed, in case of parallel upload[15].

Partial object retrival Users can retrieve part of an object, for example, just a portion of a movie object file[35].

3.2.2 Data model

OpenStack Swift allows users to store unstructured data objects with a canonical name containing *account*, *container* and *object* in given order[29]. The account names must be unique in the cluster, the container name must be unique in the account space, and the object names must be unique in the container. Other than that, if containers have the same name but belong to a different account, then they represent different storage locations. The same principle applies to objects. If objects have the same name but not the same container and account name, then these objects are different.

Account Accounts are root storage locations for data. Each account contains a list of containers within the account and metadata stored as key-value pairs. Accounts are stored in the account database. In OpenStack Swift, account is **storage account** (more like storage location) and **do not represent a user identity**[29].

Container Containers are user-defined storage locations in the account namespace where objects are stored. Containers are one level below accounts, therefore they are not unique in the cluster. Each container has a list of objects within the container and metadata stored as key-value pairs. Containers are stored in container database[29].

Object An object represents data stored in OpenStack Swift. Each object belongs to one (and only one) container. An object can have metadata stored as key-value pairs. Swift stores multiple copies of an object across the cluster to ensure durability and availability. Swift does this by assigning an object to *partition*, which is mapped to multiple drives, and each driver will contain object copy[29].

Server Processes The path towards data in OpenStack Swift consists of four main software services: **Proxy server**, **Account server**, **Continaer server** and **Object server**. Typically Account, Container and Object server are located on same machine creating **Storage node**.

Proxy server The proxy server is the service responsible for communication with external clients. For each request, it will look up storage location(node) for an account, container, or object and route the request accordingly[14]. The proxy server is responsible for handling many failures. For example, when a client sends a `PUT` request to OpenStack Swift, the proxy server will determine which nodes store the object. If some node fails, a proxy server will choose a hand-off node to write data. When a majority of nodes respond successfully, then the server proxy will return a success response code[29].

Account server Account server stores information about containers in a particular account to SQL database. It is responsible for listing containers. It does not know where specific containers are, just what containers are in an account[14].

Container server Container server is similar to account server, except it is responsible for listing objects and also does not know where specific objects are[14].

Object server The Object Server is blob storage capable of storing, retrieving, and deleting objects. Objects are stored as binary files to a filesystem, where metadata are stored in the *file's extended attributes (xattrs)*. This requires a filesystem with support of such attributes. Each object is stored using a hash value of object path (account/container/object) and timestamp. This allows storing multiple versions of an object. Since last write wins (due to timestamp), it is ensured that the correct object version is served[14].

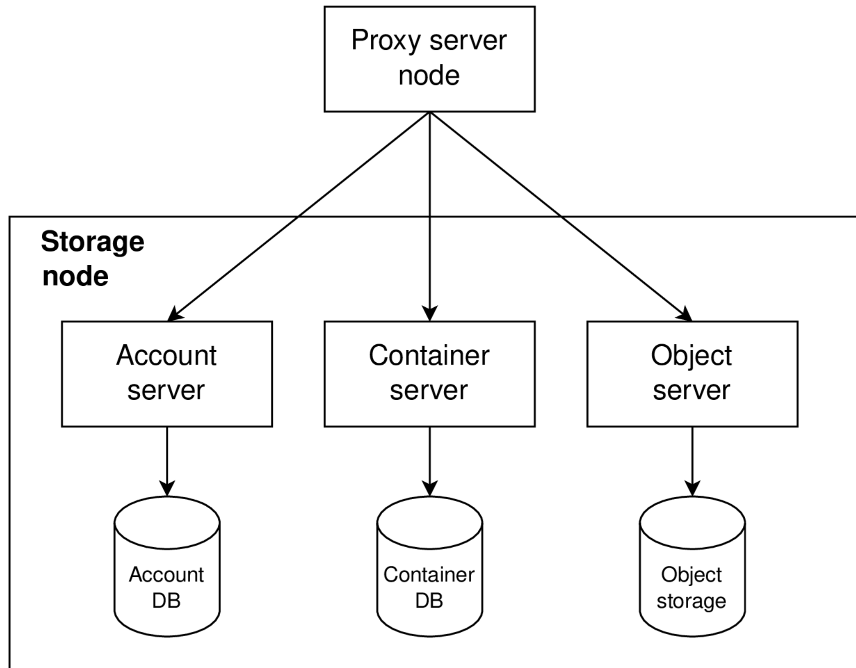

Figure 3.3: OpenStack Swift servers architecture.

3.2.3 Middlewares

Using Python WSGI middleware, users can add functionalities and behaviors to OpenStack Swift. Most middlewares are added to the Proxy server but can also be part of other servers (account server, container server, or object server).

Middlewares are added by changing the configuration of servers. In example 3.2 `webhook middleware` is added into the proxy server by changing its pipeline (*pipeline:main*). Middlewares are executed in the given order (first will be called webhook middleware, then proxy-server middleware).

Some of the middlewares are required and will be automatically inserted by swift code[16].

```
[DEFAULT]
log_level = DEBUG
user = <your-user-name>

[pipeline:main]
pipeline = webhook proxy-server

[filter:webhook]
```

```
use = egg:swift#webhook

[app:proxy-server]
use = egg:swift#proxy
```

Listing 3.2: Example of proxy server configuration (proxy-server.conf).

Interface OpenStack Swift servers are implemented using Python WSGI applications. Therefore only Python WSGI middlewares are accepted in OpenStack Swift.

In listing 3.3 is example of simplified healthcheck middleware. The constructor takes two arguments, the first is a WSGI application, and the second is a configuration of middleware defined using Python Paste framework in proxy-server.conf. Middleware must have a call method containing the request environment information and response from previously called middleware. Middleware can perform some operations and call the next middleware in the pipeline or intercept a request. In the healtcheck example, if the path directs to /healtcheck , the middleware will return HTTP Response, and other middlewares in the pipeline will not be called.

Method filter_factory is used by the Python Paste framework to instantiate middleware.

```python
import os
from swift.common.swob import Request, Response

class HealthCheckMiddleware(object):
    def __init__(self, app, conf):
        self.app = app

    def __call__(self, env, start_response):
        req = Request(env)
        if req.path == '/healthcheck':
            return Response(request=req, body=b"OK", content_type="text/plain")(env,
    start_response)
        return self.app(env, start_response)

def filter_factory(global_conf, **local_conf):
    conf = global_conf.copy()
    conf.update(local_conf)

    def healthcheck_filter(app):
        return HealthCheckMiddleware(app, conf)
    return healthcheck_filter
```

Listing 3.3: Example of healthcheck middleware in OpenStack Swift

Metadata OpenStack Swift separates metadata into 3 categories based on their use:

- **User Metadata** - User metadata takes form X-<type>-Meta-<key>: <value>, where <type> represent resource type(i.e. account, container, object), and <key> and <value> are set by user. User metadata remain persistent until are updated using new value or removed using header X-<type>-Meta-<key> with no value or a header X-Remove-<type>-Meta-<key>: <ignored-value>.

- **System Metadata** - System metadata takes the form of X-<type>-Sysmeta-<key>: <value>, where <type> represent resource type(i.e. account, container, object) and

`<key>` and `<value>` are set by internal service in Swift WSGI Server. All headers containing system metadata are deleted from a client request.

System metadata are visible only inside Swift, providing a means to store potentially sensitive information regarding Swift resources.

- **`Object Transient-Sysmeta`** - System metadata takes the form of `X-Object-Transient-Sysmeta-<key>:<value>`. Transient-sysmeta has a similar behavior as system metadata and can be accessed only within Swift, and headers containing Transient-sysmeta are dropped. If middleware wants to store object metadata, it should use transient-sysmeta[16].

3.3 MinIO

This section introduces MinIO object storage, describes its key features, most essential components, and event notifications in MinIO.

3.3.1 Introduction

MinIO is software-defined object storage that provides high performance and scalability. MinIO was designed to be the standard in private/hybrid cloud object storage. It runs on industry-standard hardware and is 100% open source[10].

MinIO software-defined object storage suite consists of a *MinIO server* and optional components.

MinIO Server - MinIO Server is distributed object storage server.

MinIO Client - Service provoding familiar UNIX commands like *ls, cat, cp, diff* in MinIO storage.

MinIO Console - Browser-based GUI offering all commands from MinIO Client in a design that feels more intuitive for DevOps and IT admins.

MinIO Kubernetes Operator - plugin allowing easy deployment and operation of MinIO object storage on Kubernetes.

3.3.2 Key features

MinIO was designed to multiple benefits to object storage:

Ease of use MinIO can be installed simply by downloading a single binary file and executing it. The configuration setup has been kept to a bare minimum. Upgrading to a newer version is done with a single command, which is non-disruptive and does not provoke any downtime[24].

Encryption and WORM MinIO provides per-object encryption using a unique object key protected by a master key managed key-management system (KMS).

MinIO supports object locking by enforcing *Write-Once-Read-Many(WORM)* immutability until the lock is expired or lifted. This mode prevents tempering with data once written[28].

Metadata architecture MinIO does not provide separate storage for metadata. All operations are performed on object-level granularity. This approach isolates any failures and does not allow any spillover to larger system failures[24].

High availability MinIO design allows a server to lose up to half its drivers and a cluster to lose up to half its servers, and MinIO will still be able to successfully process requests and serve objects. This is achieved by erasure code that protects data with redundancy[24].

3.3.3 Architecture

MinIO is designed to be cloud-native object storage to be run in lightweight containers managed by external orchestration service such as Kubernetes. The entire server is 40MB static binary and is highly effective in its use of CPU and memory resources. This allows co-hosting multiple numbers of tenants on shared hardware[10].

Usually, storages are built using multi-layer storage architecture, with a durable block layer at the bottom, a virtual file system in the middle layer, and multiple API gateways providing multiple protocols for emulating file, block, and object manipulation. The problem with this approach is that it has too many compromises[24].

MinIO decided on a completely different approach compared to other storage systems. Since MinIO's primary purpose is to serve only objects, it was built using **single-layer architecture** that provides all necessary functionalities without compromises. The advantage of this approach is object storage with high performance and lightweight[24].

In MinIO single-layer architecture, there is no such thing as Metadata server, but objects *data and metadata are stored together*, which eliminates the need for a metadata database. In addition, MinIO performs all functions (erasure code, bitrot check, encryption) as inline, strictly consistent operations. This metadata design allows, in case of damage of an object, the damage can be healed/corrected for the individual object[28].

Figure 3.4 visualize MinIO cluster architecture. Each MinIO cluster is a collection of distributed MinIO servers attached to local drivers (JBOD/JBOF). Drivers are grouped into erasure sets and objects are stored into these sets using a hashing algorithm[10].

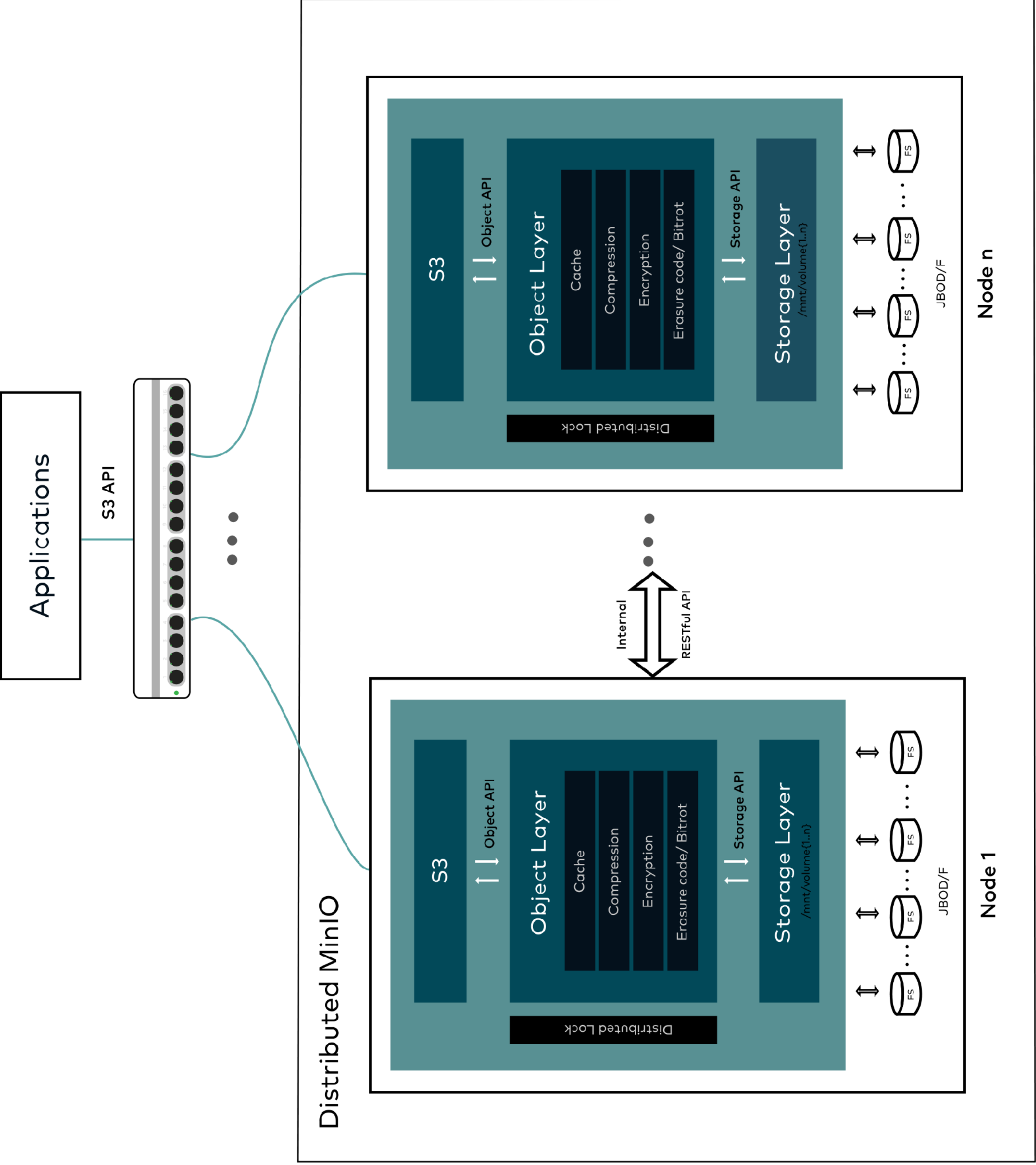

Figure 3.4: Overview of MinIO cluster architecture (source: [10], modified).

3.3.4 Event notifications

MinIO supports event notification for an event occurring to objects. MinIO provides Amazon S3 like events structure and API for defining which events will be published. Administrators can define bucket-level notification rules using MinIO client or provided MinIO SDK API, around which S3 events and objects will MinIO publish event notifications. MinIO Lambda Notifications are built into the MinIO object storage service and only require access to the remote notification target [9].

Supported event notification targets are AMQP, Redis, MySQL, LMQTT, NATS, Apache Kafka, Elasticsearch, PostgreSQL, Webhooks, and NSQ. Figure 3.5 provides an overview of events triggering and event publishing in MinIO.

Beside events occred on objects such as `s3:ObjectCreated:*` and `s3:ObjectRemoved:*`, MinIO offers event notification for access to storage `s3:ObjectAccessed:*` and event notification when bucket is created `s3:BucketCreated` and deleted `s3:BucketRemoved`.

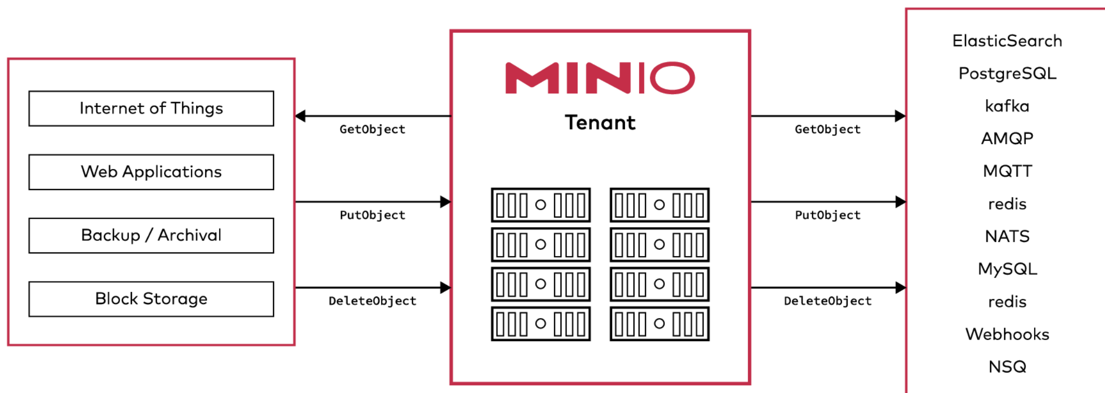

Figure 3.5: Overview of event notification in MinIO storage (source: [9]).

Chapter 4

Solution draft

This chapter describes the current state of event notifications in OpenIO SDS, OpenStack Swift, and MinIO. It describes proposed solutions for OpenIO SDS and OpenStack Swift in the form of middleware and for publishing events notification from MinIO to Beanstalkd in the form of an proxy application.

4.1 Current state

4.1.1 OpenIO SDS

OpenIO Software-Defined storage has event-driven architecture, capable of publishing events to Beanstalkd using *event-agent service* and *Notify filter*. The main disadvantage of the current event publishing state is that configuration describing what type of events should be published is applied to the whole storage. Since OpenIO SDS is a multi-tenant space, some tenants might be interested in different events inside storage. The best use-case solution would be to let tenants decide what kind of events should be published in storage assigned to them.

Second disadvantage is **lack of event filters**. Tenants might be interested in events involving specific objects or buckets that satisfy specific rules (e.g., object prefix, size).

The third disadvantage is that events are published only to a beanstalkd queue. OpenIO SDS does not support any other destinations for event publishing. Since events can be used for monitoring, there should be a proper interface, allowing users to define the destination to which events will be published (e.g., Kafka, Prometheus, MySQL).

4.1.2 OpenStack Swift

Currently, there is support for event publishing in OpenStack Swift. For example, there is no way to detect changes in a given container except by listing its content and comparing timestamps.

To partially solve this problem, OpenStack Swift created a specification[1] of middleware that would send out notifications to users if a new object was created, metadata updated, or data has been deleted. Two proposed solutions[20][22] lacked a standard interface for event publishing (no support for either Amazon S3 or CloudEvents), which were not accepted and are outdated.

[1] OpenStack Swift: Send notifications on PUT/POST/DELETE requests https://specs.openstack.org/openstack/swift-specs/specs/in_progress/notifications.html

4.1.3 MinIO

MinIO supports event publishing in the form of *Bucket notifications*. It can inform a user when an object is created, updated, or deleted. Besides events regarding objects, MinIO provides events notifications for replication events and evens regarding creating and deleting buckets. Furthermore, MinIO allows users to configure which events will be published using the Amazon S3 event notification structure.

MinIO offers various notification targets (e.g., MySql, Redis, Elasticsearch) but does not offer Beanstalkd as a notification target.

Minio is open-source object storage but **does not provide custom middlewares**. However, since MinIO is implemented in the Go programming language, any custom changes (tweaks) in MinIO source code means that the whole project needs to be compiled, which can result in incompatibility in future versions of MinIO.

4.2 Middleware for OpenStack Swift and OpenIO SDS

The goal is to create common application/middleware capable of running within OpenStack Swift and OpenIO SDS. The middleware will allow users to configure: which types of events will be published and a destination where given events will be published. Proposed middleware will be called **`ENOSS - Event Notifications in OpenStack Swift`**.

4.2.1 Location

For OpenIO SDS ideal place to run new middleware is inside the pipeline of event-agent. The main reason is that the event-agent has access to every event that occurs in OpenIO SDS and processes jobs in asynchronous mode, which means it will not impact the latency of client requests.

Most of the middlewares within OpenStack Swift are placed in the Proxy server since they can react to every client request. Therefore, the new proposed middleware will also be placed inside the Proxy server pipeline.

4.2.2 Design

The proposed middleware heavily utilizes containers/buckets and accounts metadata. Information about which event should be published and where will be stored in metadata of upper level. For publishing events regarding objects, the configuration will be stored in a container/bucket metadata (for container/bucket events, the configuration will be stored in the account level).

Compared to Amazon S3 Notifications, ENOSS middleware will publish events regarding containers/buckets. Furthermore, ENOSS middleware will publish events regarding access to object storage (HTTP GET/HEAD), where Amazon S3 only offers notifications about changes (PUST/POST) in a bucket.

ENOSS middleware can be configured so that specific even types will be forbidden for publishing in whole object storage. This option could be beneficial when there are many reads from object storage, and publishing those events could significantly impact object storage performance. Therefore such event types can be disabled for whole object storage.

Activity diagram of ENOSS middleware in container level is shown in figure 4.1. Container metadata contains event notification configuration for publishing objects in a given

container. Therefore the first step is to parse and validate such metadata. If the event notification configuration is not valid, then such configuration will be removed from metadata.

The next step is deciding if the event should be published. Since metadata about event publishing is stored in the upper level, ENOSS needs to read account metadata from storage. After reading and parsing account metadata, `ENOSS` middleware checks if the event satisfies a rule in configuration retrieved from account metadata. If yes, the event will be published to a specified destination in account metadata. A similar process is done for events involving objects, except objects do not carry information about event publishing, and configuration is stored in a proper container's metadata.

The figure 4.2 shows simplified class diagram of ENOSS middleware. ENOSS defines `EventDestination` inteface , which simply sends created event notification to specified destination. This allow new types of event destination to be added easily in future. Class `ENOSSMiddleware` is the core of middleware. Since OpenIO Proxy is compatible with OpenStack Proxy server, there is no need to divide ENOSS middleware into subclasses.

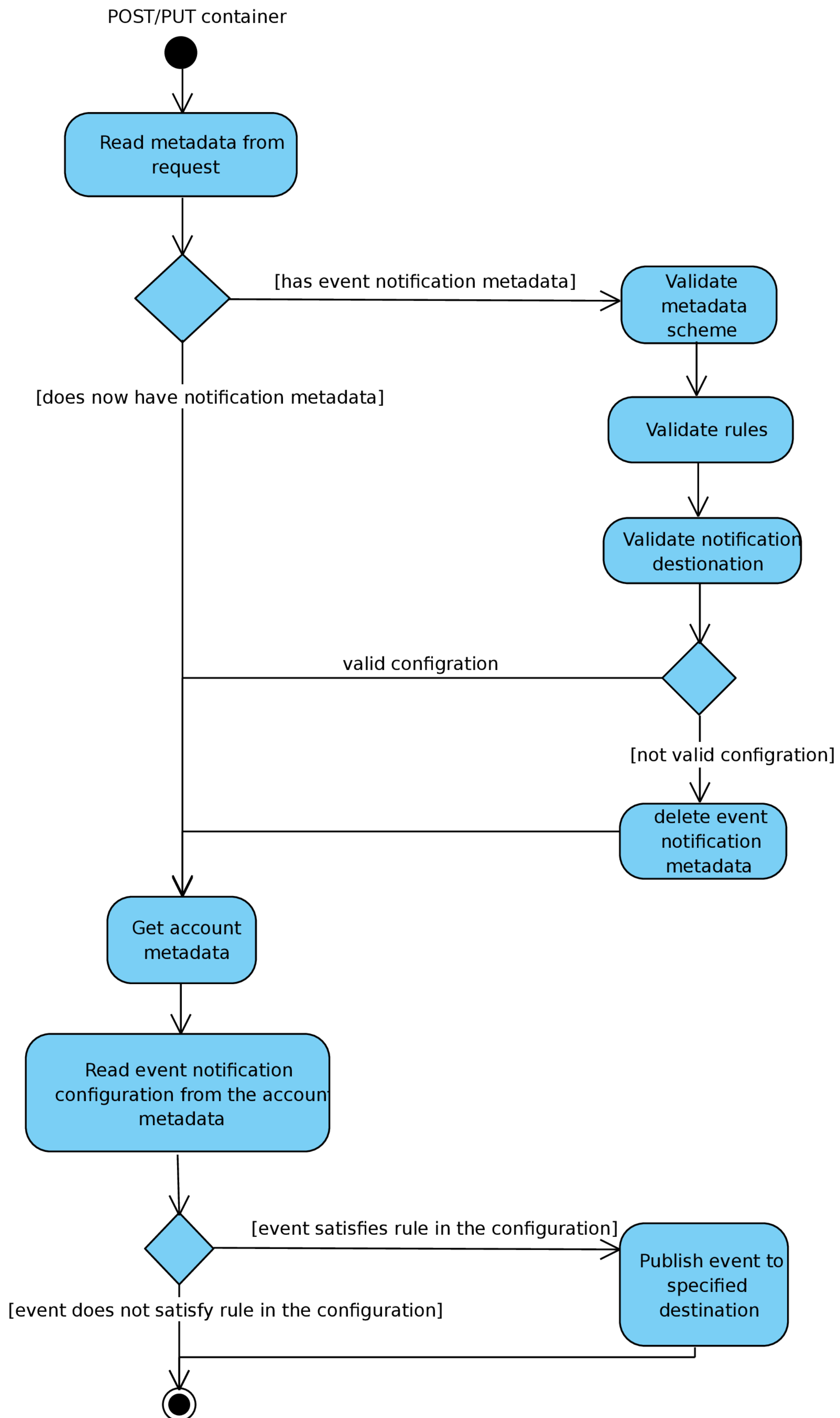

Figure 4.1: Activity diagram of ENOSS middleware.

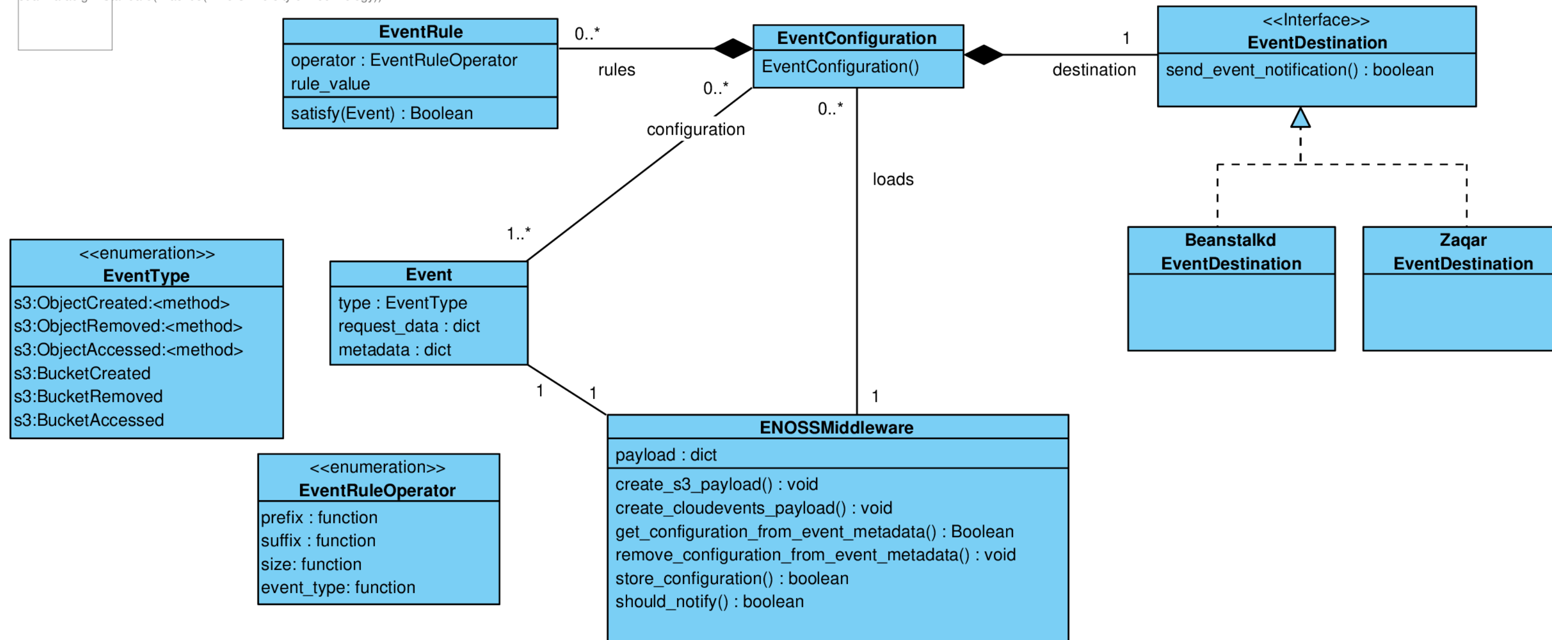

Figure 4.2: Class diagram of ENOSS middleware.

4.2.3 Structure of published event

ENOSS can publish event notification in Amazon S3 like structure and in structure following CloudEvents standard. Listing 4.1 and 4.2 describes event notification structure in JSON format.

```
{
    "specversion" : "1.0",
    "type" : "event type",
    "objectstorage" : "name of object storage (swift, openiosds)",
    "source" : "URI-reference where an event occured",
    "id" : "request id",
    "time" : "the time, in ISO-8601 format when event occured",
    "datacontenttype" : "application/json",
    "data": {
        "userid": "id of user that created event",
        "useripaddress": "ip addres of user that created event",
        "requestid": "request id",
        "transactionid": "transaction id",
        "configurationid": "id configuration that trigged notification",
        "resource": {
            "name": "name of the resource that triggered event (name of
                an object, container or account)",
            "hash": "hash value / internal id of resource",
            "metadata": "user metadata"
        }
    }
}
```

Listing 4.1: CloudEvents structure of event notification published by ENOSS middleware.

```
{
  "Records":[
    {
      "eventVersion":"2.2",
      "eventSource":"aws:s3",
      "eventTime":"The time, in ISO-8601 format, for example,
          1970-01-01T00:00:00.000Z, when an object storage finished
          processing the request",
      "eventName":"event-type",
      "userIdentity":{
        "principalId":"id of user who caused the event"
      },
      "requestParameters":{
        "sourceIPAddress":"ip address where request came from"
      },
      "responseElements":{
        "x-amz-request-id":"request ID"
      },
      "s3":{
        "s3SchemaVersion":"1.0",
        "configurationId":"ID found in the bucket notification
            configuration",
        "bucket":{
          "name":"bucket-name",
          "ownerIdentity":{
            "principalId":"if od bucket owner"
          },
          "arn":"bucket-ARN in format arn:aws:s3:::<bucket-name>"
        },
        "object":{
          "key":"object key/name",
          "size":"object-size in bytes",
          "eTag":"object eTag/hash",
          "versionId":"object version if bucket is versioning-
              enabled, otherwise null",
          "sequencer": "a string representation of a hexadecimal
              value used to determine event sequence, only used with
               PUTs and DELETEs"
        }
      }
    }
  ]
}
```

Listing 4.2: Amazon S3 structure of event notification published by ENOSS middleware.

4.2.4 Event Notification configuration

User can store event notification configuration using metadata with key **`EventNotificationConfiguration`** where value is configuration. EvenNotifcation middleware offers Amazon S3 like structure for configuring event notifications.

Listing 4.3 describes event notification configuration. `<Target>` represent targeted destination where event notifications will be sent (e.g., Beanstalkd, Elasticsearch). `<FilterKey>` is a unique name of a filter containing rules that must be satisfied in order to publish events.

Event type takes form `s3:<Type><Action>:<Method>` and are compatible with Amazon S3 event types. Type represents resource type (object, bucket), action represent action preformed by user and can have values: `Created, Removed, Accessed`. The method represents the REST API method performed by a user: `Get, Put, Post, Delete, Copy, Head`. For example, if a new object was created, even type would be described as `s3:ObjectCreated:Put`. To match event type regardless of API method assign value `*` to `<Method>`.

```
{
  "<Target>Configrations": [
    {
      "Id": "configration id",
      "TargetParams": "set of key-value pairs, used specify dynamic
          parameters of targeted destination (e.g., name of beanstalkd
          tube or name of the index in Elasticsearch)",
      "Events": "array of event types that will be published",
      "PayloadStructure": "type of event notification structure: S3 or
          CloudEvents (default value S3)",
      "Filter": {
        "<FilterKey>": {
          "FilterRules": [
            {
              "Name": "filter operations (i.e. prefix, sufix, size)",
              "Value": "filter value"
            }
            ...
          ]
        }
      }
    }
    ...
  ]
}
```

Listing 4.3: Strucute of event notification configuration

4.3 Proxy for MinIO

MinIO has support for event notifications. The main problem is that it does not support Beanstalkd as an event notification destination. Since any change in MinIO source code could lead to incompatibility with future versions and with no support for custom appli-

cations/middlewares inside MinIO, the safest solution to publish event notifications from MinIO to Beanstalkd would be a proper proxy application.

The proposing proxy application would connect to some of the supported event notifications destinations (e.g., MQTT[2]), subscribe to events coming from MinIO, and forward them to Beanstalkd.

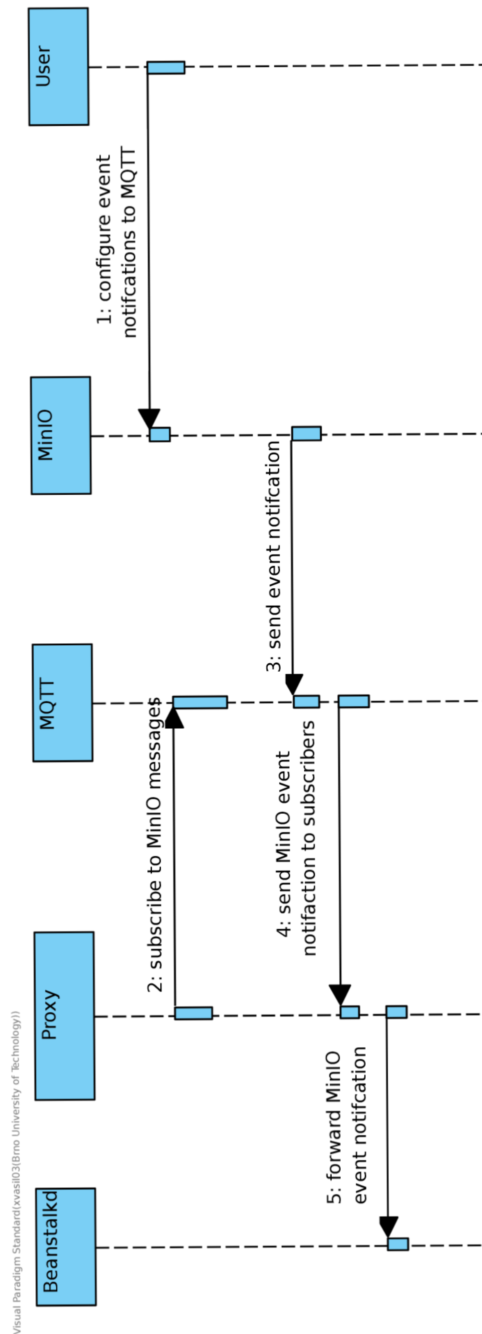

Figure 4.3: Sequence diagram of proxy application allowing publishing events from MinIO to Beanstalkd.

[2]MQTT - extremely lightweight publish/subscribe messaging protocol https://mqtt.org/

Figure 4.3 shows a sequence diagram of the proposed proxy application for MQTT. The user configures MinIO event notifications using *MinIO client* or *MinIO SDKs*. Proxy application subscribes for MinIO messages in MQTT. Once event notification is sent from MinIO to MQTT, MQTT will send the event notification to subscribers, in this case Proxy application. Proxy application will receive a message containing an event notification from MinIO and forward it to Beanstalkd.

Chapter 5

Implementation

This chapter discusses the implementation of a new OpenStack middleware - ENOSS. Frist, it summarises the implementation of the central middleware and describes the new API for user communication. The chapter then describes notification configuration processing and it's validation process in ENOSS. Furthermore, this chapter covers the implementation of handlers for different types of payloads, filters, and destinations with a big emphasis on ENOSS „openness“ to new custom handlers. The last part of this chapter discusses the possibility of running ENOSS inside OpenIO SDS.

The implementation of OpenStack middleware **ENOSS** is publicly avaliable at Github[1] and at official software repository for Python - Pypi[2].

5.1 ENOSS

ENOSS (Event notifications in OpenStack Swift) is implemented in the form of Python WSGI middleware and is located in the Proxy server pipeline. ENOSS is implemented using the flake8 coding style with the OpenStack hacking module.

5.1.1 Middleware

Central ENOSS component is middleware located in in source file `enoss/enoss.py`. Since WSGI middleware „wraps“ incoming user request and OpenStack Swift response, all informations about user request (user ip address, headers, request body, etc) and Swift response (headers, body, HTTP code, etc) are avaliable to ENOSS.

ENOSS workflow can be divided into 6 logical stages:

1. **Storing new notification configuration** (fig. 5.1) - Before incoming user request is passed on to the Proxy server pipeline, ENOSS needs to determine if user is trying to store notification configuration to OpenStack Swift. In order to allow users to store new notification configuration ENOSS offers API `POST /<account>?notification` for enabling notifications on accounts level and `POST /<account>/<container>?notification` for enabling notifications on containers level, where body of such requests contains notification configuration. Storing notification configuration to object level is forbidden and on such user request ENOSS will return HTTP Forbidden (401).

[1] ENOSS Github repo https://github.com/xvasil03/enoss
[2] ENOSS package published on Pypi https://pypi.org/project/enoss

If the incoming user request fits the specified API, ENOSS will read the request body and check if it contains a valid notification configuration. If body contains invalid configuration, ENOSS will return HTTP Bad Request (400), otherwise ENOSS will modify user request by storing notification configuration appropriate system metadata header - „`X-<type>-Sysmeta-notifications`" where `<type>` is either Account or Container. After this, the user's request will be passed down to the pipeline, and ENOSS will receive Swift's response to the user's request.

Since system metadata can be accessed only by applications running within Swift, users cannot change notification configuration on their own (by updating metadata using POST request) and must use ENOSS API.

ENOSS API uses the HTTP POST method; Swift will store notification configuration only if a user has write rights. This approach allows ENOSS to avoid ACL user checking and enforces that only users with write rights will be able to configure event notifications.

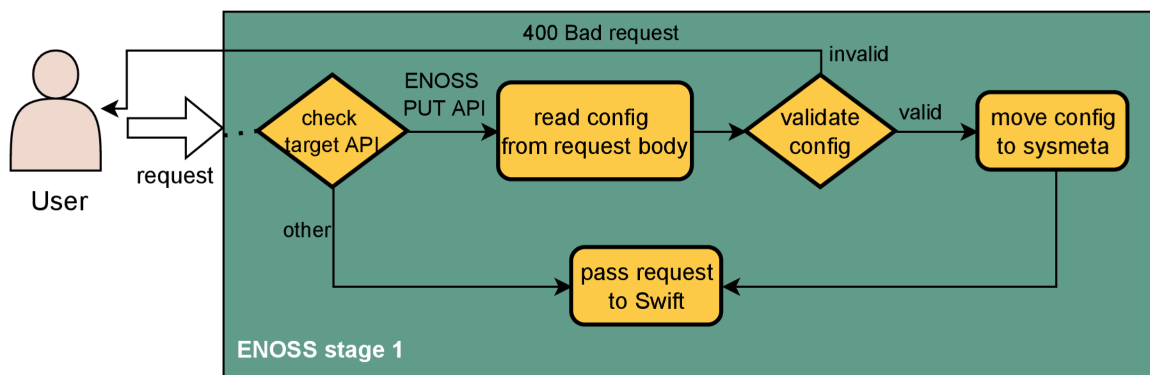

Figure 5.1: ENOSS middleware stage 1 - Storing new notification configuration.

2. **Reading notification configurations of upper levels** (fig. 5.2) - Swift processed a user's request, and ENOSS received Swift's response to the user request, which means that event occurred in Swift. The main task of this stage is to collect all notification configurations stored in the upper levels of the hierarchy. Admin notification configuration is already available (read during initialization of middleware) and remains to read configuration stored in accounts and containers level.

   Proxy server uses per request cache - internally called **infocache**, to which stores all metadata read from object storage during the processing of user's request. For reading informations(metadata) about containers and accounts Swift internally offers functions `get_account_info()` and `get_container_info()`. Those functions first check if wanted information is available in infocache, then check Memcache(if it is available), and only then do functions create another request to Swift for wanted information.

   In this specific case, metadata of upper levels are available in infocache. In addition, Admin's configuration is also available. Therefore, ENOSS does not need to create an additional request to the object storage and reads configurations from fully cached data.

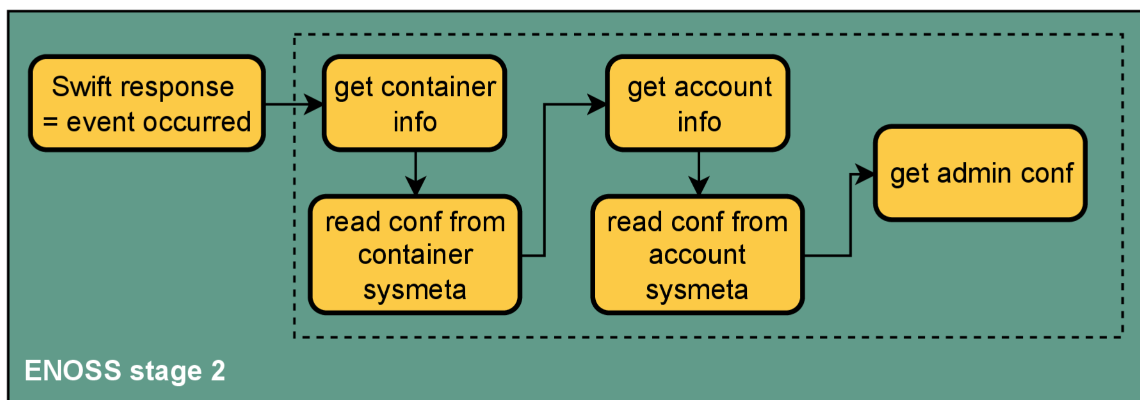

Figure 5.2: ENOSS middleware stage 2 - Reading notification configurations of upper levels.

3. **Evaluate satisfied configurations** (fig. 5.3) - Each notification configuration of the upper-level ENOSS needs to evaluate if the event should be published. In this stage, ENOSS checks if even type and filter rules are satisfied. In addition, ENOSS checks if a notification configuration allows publishing unsuccessful events (ENOSS by default publishes only successful events). It is important to note that some of the filters might need information that is not currently available(cached), resulting in an additional request to object storage.

   The results of this stage are configurations for whom an occurred event satisfies all rules and filters.

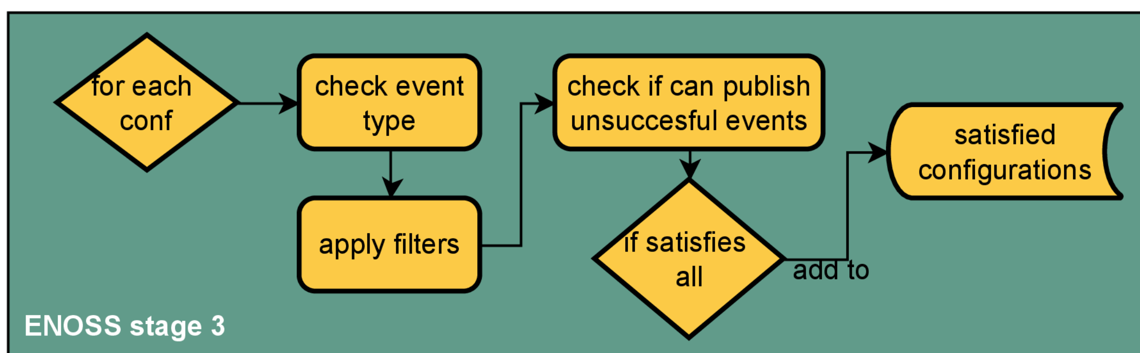

Figure 5.3: ENOSS middleware stage 3 - Evaluation of satisfied configurations.

4. **Creating notification payload** (fig. 5.4) - In this stage, ENOSS creates a payload of notification for the occurred event based on a specified type of payload in the notification configuration. By default payload type is AWS S3, but a user can specify other types. Like the previous stage, some information might not be available in the cache, and an additional object storage request might occur.

5. **Sending notification** (fig. 5.4) - Notification payload was created, and ENOSS will select a proper destination to where a notification will be sent. The target destination is specified in the notification configuration, which triggers a notification.

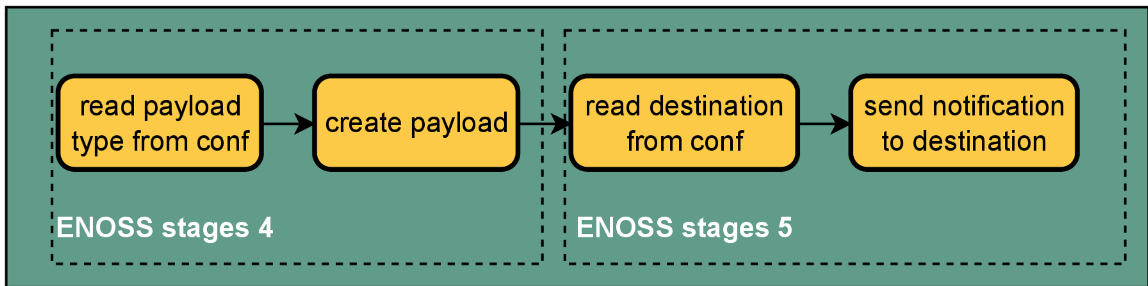

Figure 5.4: ENOSS middleware stages 4 and 5 - Creating notification payload and sending notification to destination.

6. **Reading stored configuration** (fig. 5.5) - In the last stage, ENOSS checks if a user's request was targeted towards ENOSS, i.e., if a user wanted to read stored notification configuration. For this purpose, ENOSS offers API: `POST /<account>?notification` and `POST /<account>/<container>?notification`. Suppose Swift responded successfully (HTTP 200). In that case, a user was successfully authorized to perform a read operation. Therefore ENOSS is „allowed" to send user asked information without worrying about a security breach. ENOSS will read account/container metadata using swift functions `get_account_info()` or `get_container_info()` (similarly to stage 1), and extract notification configuration from system metadata. Extracted notification configuration will be stored in a response's body of user request.

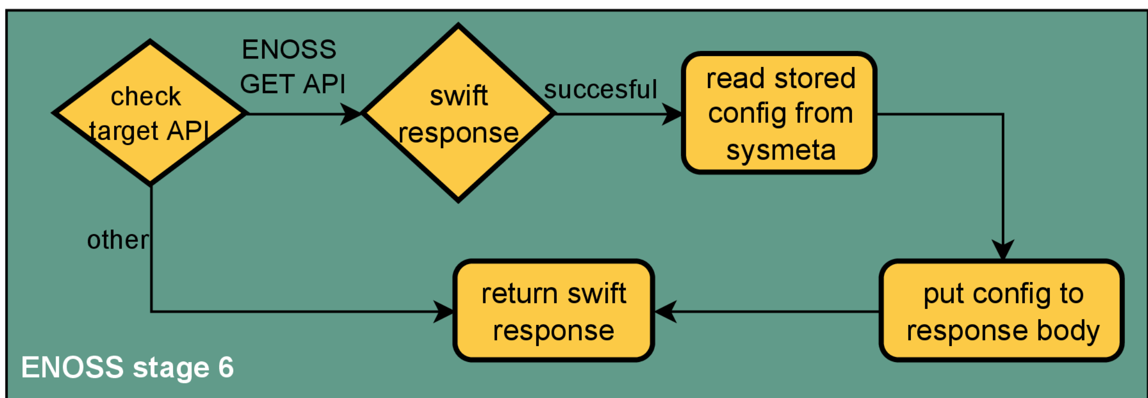

Figure 5.5: ENOSS middleware stages 6 - Reading stored configration.

Configuration - during the runtime process, the middleware needs to validate new notification configurations, create different types of notification payloads, and send notifications to various destinations. In order to do so, the middleware needs information about available destinations and destination connection configurations. Furthermore, the middleware needs information about available payload types and the validation process of a new notification configuration. Lastly, since ENOSS allows Swift admins to publish notifications, the middleware also needs access to the admin's notification configuration.

Part of the needed information is stored in the Proxy server's configuration. Listing 6.2 shows an example of middleware configuration, where:

- **destinations_conf_path** - is a path to a configuration file containing all information needed for ENOSS to connect to various destinations (mandatory).
- **use_destinations** - is a list of destinations (separated by a comma) that can be used during ENOSS runtime. Since ENOSS supports multiple destinations, not all of them must be used during run time. Therefore, ENOSS will create connections only to destinations specified in this list (mandatory).
- **s3_schema** - path to file containing JSON schema used during the validation process of new notification configurations (mandatory).
- **admin_s3_conf_path** - path to file containing admins notification configuration for publishing notifications (optional).

```
[filter:enoss]
destinations_conf_path = /etc/swift/enoss/destinations.conf
use_destinations=beanstalkd,elasticsearch
s3_schema = /etc/swift/enoss/configuration-schema.json
admin_s3_conf_path = /etc/swift/enoss/admin_s3_conf.json
paste.filter_factory = enoss.enoss:enoss_factory
```

Listing 5.1: Example ENOSS middleware configuration stored in the Proxy server configuration (proxy-server.conf).

During the **initialization process**, the middleware loads destinations configuration (`destinations_conf_path`), uses loaded configuration to initialize destinations handlers (which create connections to destinations), loads admin's notification configuration (`admin_s3_conf_path` and initialize handlers used for the creation of different types of notification payloads.

5.1.2 Notification configuration

Before storing a new notification configuration, ENOSS checks if the configuration is valid. For this purpose, ENOSS uses class **S3ConfigurationValidator** located in `enoss/configurationpy`. Middleware provides `S3ConfigurationValidator` with information about available destinations, payload, and new notification configuration that needs to be validated. The validation process is divided into five steps:

1. **Schema validation** - new notification configuration is validated using JSON schema validator.
2. **Event type validation** - since a user can specify for which even types notification should be published, it is needed to validate if specified event types are supported and named correctly.
3. **Filter validation** - a user can filter events by setting various rules that must be satisfied in order to send a notification. This step checks if specified filter rule operators are supported and if their input value is valid.
4. **Destination validation** - `S3ConfigurationValidator` checks if for specified destinations in a new configuration exists an available destination handler in ENOSS capable of publishing notifications to specified destinations.

5. **Payload validation** - Similar to the previous step, `S3ConfigurationValidator` checks if there is an available payload handler capable of creating a notification's payload type specified in a new notification configuration.

If a new notification configuration is invalid, `S3ConfigurationValidator` will raise an exception **`ConfigurationInvalid`**, which results in ENOSS returning HTTP Bad Request.

Once an event occurs in Swift, middleware reads notification configurations stored in system metadata of upper levels(containers or accounts). In order to easily manipulate notification configurations, middleware represents notification configuration using class **`S3NotifiationConfiguration`**, located in `enos/configuration.py`.

`S3NotifiationConfiguration` offers function `get_satisfied_destinations()`, which for an occurred event in Swift computes destination configurations where all specified criteria for publishing notification are satisfied. This function needs to:

- Checks if occurred event type matches specified event types.
- Performs event filtering using specified event rules.
- In case of unsuccessful events (Swift response HTTP code is not 2xx), check if a user allowed publishing notification for unsuccessful events.

5.1.3 Filters

ENOSS allows users to filter events using filter rules. Only events satisfying specified rules are published. ENOSS defines the interface of classes that perform various types of filtering. All filter rules classes are located in `enoss/filter_rules/`. It is essential to notice that in some cases filter rule might need information that is not available/not cached and it can result in an additional request to Swift storage.

`RuleI` - is an interface specifying class representing user-specified rules that must be satisfied in order to publish event notifications. The constructor receives a value, which is read from the notification configuration. The call method has access to all information about the request, which allows implementing rules about, e.g., user IP address, return code, object prefix/suffix/length. Function **`validate()`** is used during the validation process of new notification configurations. Its task is to validate if an input value can be used to initialize a given filter rule. This function can be used for data type checking and if an input value has a correct format.

```
1  @six.add_metaclass(abc.ABCMeta)
2  class IRule(object):
3      def __init__(self, value):
4          self.value = value
5
6      @abc.abstractmethod
7      def __call__(self, app, request):
8          raise NotImplementedError('__call__ is not implemented')
9
10     @staticmethod
11     def validate(value):
12         raise NotImplementedError('validate is not implemented')
13
```

Listing 5.2: Interface of class representing filter rule.

Currently, ENOSS offers the following filter rules:

- **Suffix**: input value is a string. Checks if user's request target(account/container/object) has specified suffix.
- **Prefix**: input value is a string. Similar to `Suffix`, except it checks targets prefix.
- **Maxsize**: input value is int representing size in bytes. If a request's target is an object, then checks if the object's size is not bigger than the specified size. In the case of an account or container, it checks if its used space (metadata `X-<target>-Bytes-Used`) is not bigger than the specified size.
- **Minsize**: input value is int representing size in bytes. Similar to `Maxsize`, except target's size must not be lower then specified size.
- **Contenttype**: input value is a string representing content type. If a request target is an object, then checks object type. In the case of `PUT`, `GET` or `HEAD` methods, the object's content type will be read from headers. Otherwise, an object's type will be read from object storage (if it is not in the infocache, it will result in an additional request to Swift).
- **Httpcodes**: input value is a list of strings representing HTTP code or group of HTTP codes (e.g. [„200“, „404“, „4xx“]). Checks if Swift response's HTTP code matches any of the input values. Users can use the wildcard character „x“ to specify a group of HTTP codes.
- **Usersin**: input value is a list of strings representing a list of users. If a user who created an event in Swift is in the input list, the rule is satisfied.
- **Usersout**: input value is a list of strings representing a list of users. Similar to `Userin`, except rule is satisfied in user that created event is not in the input list.

5.1.4 Notification payload

ENOSS is flexible regarding a payload of notifications. Notification payload might differ based on ENOSS applications and destination types to which notification should be sent. ENOSS specifies the interface of classes that create notification payload, and classes realizing defined interface are located in `enoss/payload/`.

PayloadI - is an interface specifying classes used for creating notification payload. When event notifications are configured on a container or account, ENOSS sends test notifications to all specified destinations in configuration. This way, it allows users to check if they successfully configured event notifications. Method `create_test_payload()` is used for this purpose. One of the parameters is `request`,which contains all information about the incoming request(e.g., user IP address, incoming headers) and information about Swift response(e.g., headers, status code). The `invoking_configuration` contains information about stored notifications configuration. When an event occurs on a container/account with enabled notifications, ENOSS checks if notification for such event should be published based on event notification configuration. If yes, the method `create_payload()` will be used to create a notification payload. Similar to filter rules, if a payload needs information that is currently not available/not in the cache, an additional request to Swift storage might occur.

```
1   @six.add_metaclass(abc.ABCMeta)
2   class IPayload(object):
```

```
    def __init__(self, conf):
        self.conf = conf

    @abc.abstractmethod
    def create_test_payload(self, app, request, invoking_configuration):
        raise NotImplementedError('create_test_payload is not implemented')

    @abc.abstractmethod
    def create_payload(self, app, request, invoking_configuration):
        raise NotImplementedError('create_payload is not implemented')

```

Listing 5.3: Interface of class used to create notification payload

Currently ENOSS support notification payload types:

- **S3Payload** - this class creates a notification payload compatible with Amazon AWS S3 notifications described in the listing 4.2. Class is optimized to read all needed information from available sources (infocache, headers, etc.). In some cases, for example, updating objects metadata using the PUT method, some information (i.e., object size) is not available in infocache or headers. Such cases will result in an additional request to Swift storage in order to retrieve needed information.

5.1.5 Destinations

ENOSS supports sending notifications to various types of destinations. Similar to filter rules and payloads, ENOSS defines interface class for this purpose. During ENOSS runtime, all available destination classes are initialized at the start of ENOSS, and their life ends with middlewares lifetime. All classes implementing destinations interface are located in `enoss/destinations/`.

DestinationI - is an interface specifying classes used for sending event notifications to the desired destination. The constructor receives configuration(dict), which can contain information needed for creating a connection with the desired destination(address, port, authentication,...). Configuration is loaded from ENOSS middleware configuration, which is loaded by the Proxy server. Method
`send_notification()` receives notification payload(dict), and its task is to send a notification to a desired destination.

```
@six.add_metaclass(abc.ABCMeta)
class IDestination(object):
    @abc.abstractmethod
    def __init__(self, conf):
        raise NotImplementedError('__init__ is not implemented')

    @abc.abstractmethod
    def send_notification(self, notification):
        raise NotImplementedError('send_notification is not implemented')

```

Listing 5.4: Interface of class used for sending notification message to desired destination

Available destinations - ENOSS currently supports the following destinations:

- **BeanstalkdDestination** - the destination is responsible for publishing notifications to Beanstalkd work queue. The destination is implemented using Python3 client library `Greenstalk`[3]. Beanstalkd server connection configuration is stored

[3]Greenstalk - Beanstalkd Python Client https://github.com/justinmayhew/greenstalk.

in the destination configuration file(which is specified in ENOSS section in proxy-server.conf) under section `[beanstalkd]`. From this section, information about server address, port, and beanstalkd tube are read and used to initialize the connection using the Greenstalk library. Sending notification to Beanstalkd is relatively simple, `BeanstalkdDestination` receives notification in the form of dict, transforms it to string, and sends it to a queue using Greenstalk connection.

- **KafkaDestination** - the destination is responsible for publishing notifications to Apache Kafka distributed event streaming platform. The destination is implemented using Kafka Python client library [4]. Library provides a high-level asynchronous message producer - `KafkaProducer`. Similar to `BeanstalkdDestination`, `KafkaDestination` reads server configuration from the destination configuration file and reads the name of Kafka topic to which notifications will be published. Before sending a notification, `KafkaDestination` transforms the notification from dict to bytes datatype and then sends it using KafkaProducer instance.

- **ElasticsearchDestination** - the destination is responsible for publishing notifications to Elasticsearch search engine. The destination uses official Python Client library [5]. `ElasticsearchDestination` supports HTTPS connection to Elasticsearch server. During the initialization phase, `ElasticsearchDestination` reads server connection configuration from the destination configuration file and makes a connection to Elasticsearch. After making a connection, `ElasticsearchDestination` will ping the Elasticsearch server to verify that the connection was successful. If not, an exception will be raised, and Proxy server will not be initialized. Information about targeted index and index mapping is also read from configuration. After a successful ping to the Elasticsearch server, `ElasticsearchDestination` will check if the targeted index exist. If it does not exist, an index will be created with specified index mapping(optional). In order to send a notification to Elasticsearch server, `ElasticsearchDestination` transforms the notification to a string and then uses `Elasticsearch Index API` with a specified index name.

5.1.6 Custom filters/payloads/destinations

Classes that implement individual rules, payloads, and destinations can be interpreted as **handlers** that handle/perform specific jobs - create notification payloads, filter events, or send notifications to desired destinations. Each handler type has its module. For example, filter rule handlers are located in module `enoss/filter_rules`, payload handlers in `enoss/payloads`, and destination handlers in `enoss/destination`.

ENOSS middleware imports handlers only from specified modules; therefore, all handler classes must be imported in `__init__.py` of specified modules. For ENOSS middleware to assign jobs to proper handlers, ENOSS defines the following rules:

- Handlers must implement/use predefined handler interfaces. For example, inn the case of filter rules, that is `RuleI`, for payload creation `PayloadI` and for sending notification to specific destination handler class must implement interface `DestinationI`.

[4] Kafka Python client https://github.com/dpkp/kafka-python
[5] Elasticsearch Python Client https://www.elastic.co/guide/en/elasticsearch/client/python-api/current/index.html

- Handler's name must be in format **<Type><Handler suffix>**, where handler suffix represents a type of job that handler performs. In particular, for payload creation handler suffix is Payload, for filter rules Rule and for sending notification to desired destination handler suffix is Destination. <Type> must correspond with a type of individual filter rule/payload/destination that a handler performs and is used by ENOSS during the handler selection process.
- All handlers must be imported in __init__.py file of a specified handler module.

Handler selection - during the initialization phase, ENOSS middleware will load all available handlers and separate them based on the type of the handler. ENOSS expects that the names of handlers match values in notification configuration. Figure 5.6 illustrates the handler section process and how names in notification configuration are connected to specific handlers. For example, if a user-specified notification payload is „s3", ENOSS will select handler S3Payload from module enoss/payloads. The same principle is applied to filter rules and destinations. It is essential to notice that loaded handlers are used during the validation process of new notification configurations. ENOSS will declare notification configuration as invalid if the handler cannot be found.

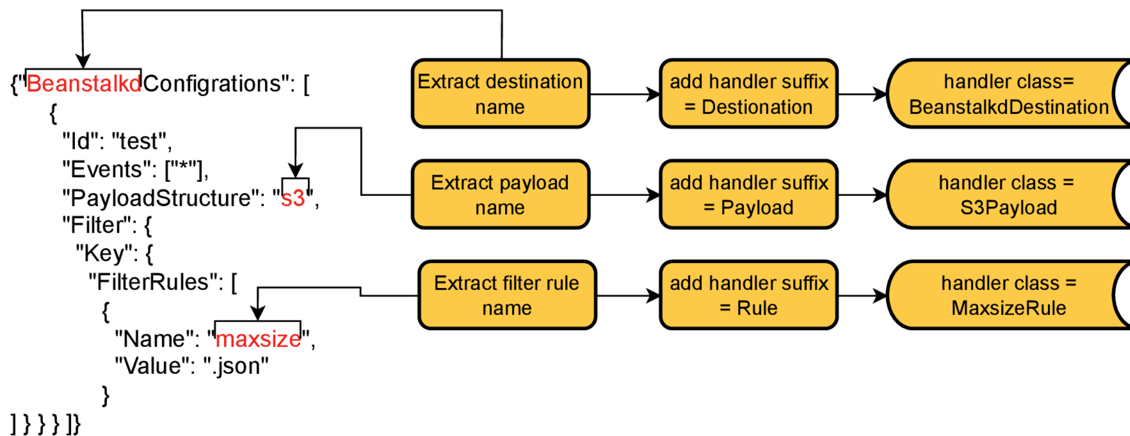

Figure 5.6: Process of handler selection for given notification configuration.

The creation of custom filters/payloads/destinations can be interpreted as a creation of a new handler, and its process is simple and intuitive. A new handler must follow the specified rules(interface, name, and location). Once a new handler is implemented and imported into a proper module, a custom filter/payload/destination can be used in notification configuration using the handler selection described above.

5.1.7 OpenIO SDS compatibility

OpenIO SDS allows running Swift middleware using the OIO-Swift gateway. Compared to the Swift Proxy server, OpenIO SDS offers a service called **oio-proxy**. oio-proxy is, in fact, a modified Swift Proxy server that allows communication with internal object storage services needed to handle user requests. The main difference between Swift and OpenIO Proxy server is that oio-proxy is implemented using **Python 2.7** while Swift proxy runs on **Python 3**.

ENOSS middleware, notification configuration, and validation, payload, and filter handlers implementation are compatible with Python 2 and Python 3. For this purpose, implementation was based on Python library `six`, which allows writing Py2/3 compatible codes.

The Problem arises in the implementation of destinations handlers, which utilize third-party libraries, where compatibility with Python 2 is not guaranteed. Since support for Python 2 has officially ended, all new client libraries are mainly implemented only using Python 3.

Kafka Python library is compatible with both Python 2 and Python 3. Therefore Kafka destination can be used when ENOSS is running within OpenIO SDS.

Beanstalkd destination will use Python library `greenstalk` when ENOSS is running in Swift. Since `greenstalk` offers only support for Python 3, Beanstalkd needed to use another library for OpenIO SDS. For this purpose, library `pybeanstalk` was used (offers only support for Python 2). Therefore, ENOSS needs to use two different libraries in order to publish notifications to beanstalkd, from Swift and OpenIO SDS.

Elasticsearch destination is compatible with Elasticsearch release version 8.2. However, this Elasticsearch release offers only Python 3 client library and is incompatible with lower versions of the Elasticsearch Python library. Therefore, when ENOSS runs in OpenIO, ENOSS cannot send notifications to Elasticsearch.

5.2 MinIO proxy

Since publishing notifications directly from MinIO to Beanstalkd queue is not possible due to possible incompatibility in newer versions of MinIO, a proxy program is needed.

The main idea of the proposed solution is that MinIO will publish notifications to supported destination, in this case, MQTT queue, the roxy program will read notification from MQTT and send it to Beanstalkd queue.

Proxy program was written in Python3. For communication with MQTT, program uses Eclipse Paho library, and for communication with Beanstalkd program utilizes Greenstalk library.

The program expects one argument - path to the configuration file in INI format. Configuration file must contain two sections:

- **beanstalkd** - information needed to connect to beanstalkd queue (address, port, and tube name).
- **mqtt** - information needed to connect to MQTT (broker, port, username, password, topic)

The program will read the configuration from the input file and then create connections to Beanstalkd and MQTT. MQTT API allows the definition of `on_message()` callback function, which will be called whenever a new message is read from MQTT. This function is defined to read incoming message payload from MQTT and forwards it to Beanstalkd queue.

Lastly, using function `loop_forever()`, the program will be run in the infinite loop, while reading messages from MQTT and forwarding to the Beanstalkd queue.

Chapter 6

Testing, benchmark and possible applications

This chapter testing ENOSS middleware to determine whether ENOSS behavior matches its specification. This process is done using two types of tests: **unit** and **functional** tests. Furthermore, this chapter explains the process of benchmarking, analyzes ENOSS latency for each stage, provides information about benchmark scenarios, and compares performance between different types of destinations.

6.1 Testing

OpenStack Swift project provides three types of tests: unit, functional, and probe tests. The first two types(unit and functional tests) are used to verify ENOSS's correct behavior, while probe tests are not used since they are designed to test much of Swift's internal processes. Tests in OpenStack Swift are standardized and automated using TOX automation project[1].

6.1.1 Unit tests

Swift's unit tests are designed to test small parts of the code in isolation [7]. Unit tests check that a small selection of the code is behaving correctly. Unit tests are implemented in Python using `unittest` framework. ENOSS unit test is located in `test/unit/common/middleware/test_`

Since the goal of unit tests is testing in isolation, all communication with external services should be excluded. In order to achieve this, all destination handlers are replaced with mock handler `MockDestination`(listing 6.1) using `unittest.mock.patch` decorator.

```
class MockDestination(object):
    def __init__(self, conf):
        self.reset()

    def reset(self):
        self.state = 'notification not sent'

    def send_notification(self, notification):
        self.state = 'notification sent'

```

Listing 6.1: Mock class used to replace destination handlers in unit tests.

[1] tox automation project https://tox.wiki/en/latest/

ENOSS unit test contains:

1. **Initialization test** - checks if ENOSS can be initialized.
2. **Handlers interface test** - checks if all available handler classes implement specified interfaces.
3. **Configuration validation tests** - checks if the validation process of notification configuration is behaving correctly. It uses various invalid configurations(unsupported event type, payload, destination, missing filter rule value, etc.) to verify the correct validation process.
4. **New valid notification configuration** - simulates a user trying to store a new valid notification configuration to Swift using ENOSS API. Test checks if new notification configuration is stored into system metadata.
5. **New invalid notification configuration** - simulates a user trying to store a new invalid notification configuration to Swift using ENOSS API. Test checks if return response from ENOSS is HTTP Bad request.
6. **Reading stored notification configuration** - a user wants to read a notification configuration from object storage. Test checks if Swift response body contains notification configuration.
7. **Invoke notification from containers level** - notification configuration is stored in the container level, and an event occurs in the object level. Test checks if notification is sent to Beanstalk queue using mocked handler `MockDestination`.
8. **Invoke notification from account level** - notification configuration is stored in the account level, and an event occurs in the containers level. Test checks if notification is sent to Beanstalk queue using mocked `MockDestination`.

6.1.2 Functional tests

The functional Swift tests are designed to validate that the entire Swift system is working correctly from an external perspective (they are „black-box“ tests). In the ENOSS testing context, functional tests are run against public Swift(and ENOSS) API endpoints. ENOSS functional test is located in `test/functional/test_enoss.py`.

Similar to unit tests, functional tests are implemented using `unittest` module. ENOSS functional test contains:

1. **Storing new notification configuration** - sends to Swift ENOSS API request with new valid notification configuration and checks if notification configuration is stored using additional request to Swift (using ENOSS GET API).
2. **Deleting existing notification configuration** - deletes stored notification configuration and check if notification configuration is indeed deleted using the same principle as the previous test.
3. **Storing invalid notification configuration** - sends to Swift ENOSS API request with invalid notification configuration, checks if Swift(ENOSS) responds with HTTP Bad request, and checks that notification configuration was not stored in Swift.

4. **ALC test for reading stored notification configuration** - simulates an unauthorized user trying to read stored a notification configuration without read rights. Expects HTTP Unauthorized response from Swift. The exact process is repeated for an authorized user, where the test expects that the user will receive notification configuration in Swift's response body.

5. **ALC test for storing new notification configuration** - simulates an unauthorized user trying to store a new notification configuration without write rights. Expects HTTP Unauthorized response from Swift. Then the same process does for an authorized user, where the test expects that the notification configuration will be successfully stored.

6.2 Performance analysis

As described in subsection 5.1.1, ENOSS can be divided into 6 logical stages. In order to understand how to measure ENOSS performance, firstly, an analysis of each logical stage is needed. The analysis mainly consists of determining whether ENOSS has all needed information to perform tasks in a stage. If not, where can ENOSS obtain such information, and how long would it approximately take for ENOSS to obtain or perform the given task.

Stages analysis:

1. **Storing new notification configuration** - information about new notification configuration is available in a user's request. ENOSS will either return HTTP Bad Request (if the configuration is invalid) or copy the configuration to the system metadata of the request. Therefore ENOSS performs this stage extremely fast, and no performance issue can occur during this stage.

2. **Reading notification configurations of upper levels** - while Swift processes incoming request, information about account and container are read from storage and then stored in infocache. ENOSS has access to infocache and will read configurations from infocache. Therefore, no performance issues can occur in this stage as well.

3. **Evaluate satisfied configurations** - in this stage, ENOSS performs event filtering based on stored notification configuration obtained in the previous stage. Some filter rules might need information that is not currently cached and available. For example, updating an object's metadata is done using the POST API method. The object size is not available in incoming request headers, Swift response headers, or infocache. If notification configuration contains an object size filter, ENOSS will obtain needed information using an additional request to object storage, which can cause additional latency.

4. **Creating notification payload** - similarly to the previous stage, in order to create a notification payload, ENOSS might need information that is not cached, and to obtain such information, ENOSS will have to make an additional request to object storage.

5. **Sending notification** - in this stage, ENOSS sends a notification to a specified destination using third-party libraries. The speed at which this task is performed depends on the type of connection with a destination (HTTP/HTTPS), the type of destination, and whether sending notifications is synchronous or asynchronous.

6. **Reading stored configuration** - similarly to the first stage, information about stored notification configuration is available in infocache. ENOSS will copy the configuration from infocache to the response body. Therefore no performance issues can occur in this stage.

Additionally, performance issues can occur during `DELETE` events. Since ENOSS creates event notifications after an event occurs, ENOSS needs to obtain information before it is deleted from object storage.

If ENOSS obtains information from object storage, such information will be saved into infocache. This results in a **maximum of one additional request** to object storage per user's request.

Therefore, computing ENOSS latency can be defined as:

$$ENOSS_latency = time(obtaining_needed_information) + time(sending_notification)$$

where obtaining needed information in the best scenario can be entirely from cached data, or in the worst scenario can result in **maximum one additional request** to object storage.

6.3 Experiments

Benchmarking was carried out on the FIT VUT university Kubernetes cluster. Each supported destination(Beanstalkd, Kafka, Elasticsearch) was created as a Kubernetes pod and service. OpenStack Swift was deployed using Nvidia Dockerfile for OpenStack Swift AIO(All in One), where Proxy, Account, Container, and Object servers are located on the same machine. OpenStack Swift pod was configured to have 16 VCPU (CPU Model: AMD EPYC 7282), 16GB RAM, and 50GB disk storage.

Since the goal is to benchmark ENOSS middleware, which is located in the Proxy server, the Proxy server was configured to have only one worker in order to saturate the Proxy server CPU. Furthermore, in order to avoid bottlenecks on other servers, four workers were assigned(12 in total) to each of the other servers(Account, Container, and Object servers).

The benchmarking process is done using the benchmark tool **`ssbench`**, designed by SwiftStack(Nvidia) for OpenStack Swift object storage system. `ssbench` has two main components - **ssbench-master** and **ssbench-worker**.The master component is in charge of creating and distributing benchmark jobs while workers preforms received jobs. A Benchmark test is defined using a **scenario**(sometimes called a "CRUD scenario"), which is a JSON-formatted file. The scenario contains the following information [17]:

- **name** - scenario's name
- **sizes** - list of „object size" classes. Each object size class has a name, minimum and maximum size of objects in bytes.
- **initia_files** - dictionary of initial file-counts per size class. Defines probability distribution of object sizes during the benchmark run itself.
- **run_seconds** - number of seconds the benchmark scenario should be run.
- **container_count** - number of containers in Swift used during the benchmark run.
- **user_count** - determines the maximum client concurrency during the benchmark run.

- **crud_profile** - determines the distribution of each kind of CRUD operations.

```
{
    "name": "Small test scenario",
    "sizes": [{
        "name": "tiny",
        "size_min": 4096,
        "size_max": 65536
      }, {
        "name": "small",
        "size_min": 100000,
        "size_max": 200000
    }],
    "initial_files": {
        "tiny": 100,
        "small": 10
    },
    "run_seconds": 300,
    "crud_profile": [3, 4, 2, 2],
}
```

Listing 6.2: Example of ssbench scenario.

All experiments were done using one ssbench master and 5 ssbench workers. During benchmarking, information about CPU and RAM usage used by the Proxy server worker was tracked. Connections to Beanstalkd and Kafka queue are done without authorization and using an unsecured connection, while the connection with Elasticsearch is made using TLS/HTTPS connection using a CA certificate.

Experiment 1: in order to fully saturate proxy CPU and avoid disk bottleneck, for this experiment ssbench scenario contains only tiny documents(size 4-6KB) and only read operations. This way, data will be serviced entirely from the buffer cache, and the disk bottleneck will be avoided. In this benchmark test, ENOSS is configured, with an admin notification configuration, to create and send notifications for all events to one destination during the benchmark test. In this scenario, ENOSS has all needed information(in infocache) and does not need to make additional requests to object storage.

The benchmark test is repeated for each supported destination, as well as when notification configuration is not enabled, and when ENOSS is enabled in the Proxy server. Each benchmark test during this experiment was run for 60 minutes.

Figure 6.1 contains information about the used CPU and RAM by the Proxy server worker(where ENOSS is located). In all cases, CPU usage was saturated, and maximum output from the Proxy server was achieved. Using information about RAM usage, a conclusion can be drawn that when ENOSS is enabled, the Proxy server worker will use about 20MB more RAM for this hardware configuration.

Table 6.1 contains benchmark results for this experiment. The first row(**ENOSS disabled**) in the table shows results when ENOSS is not enabled in the Proxy server pipeline. The second row(**No notifications**) contains benchmark results when ENOSS is enabled but was not configured to publish any notification. Other rows show results when ENOSS is enabled and configured to publish notifications to the specified destination.

When ENOSS is enabled but is not configured to publish notifications, ENOSS middleware still needs to read notification configurations of upper levels(stage 2 described in

subsection 5.1.1). Since there are no notification configurations in upper levels, ENOSS will return the received Swift response. Therefore, the latency of this example can be interpreted as the latency of the second stage. Comparing average latency when ENOSS is not enabled and when ENOSS is enabled but does not publish notifications, the statement that no performance issues can occur during reading notification configurations of upper levels is confirmed.

Comparing the results of each supported destination, a conclusion can be drawn that the beanstalkd queue outperforms the Kafka queue. At the same time, Elasticsearch has the worst performance, partially due to the HTTPS connection. Using results when ENOSS publishes notifications and when ENOSS is not enabled, the latency of publishing notifications can be computed.

In this experiment, when ENOSS had all needed information cached, ENOSS on average, took between 0.04(beanstalkd) and 0.16(Elasticsearch) seconds to publish a notification.

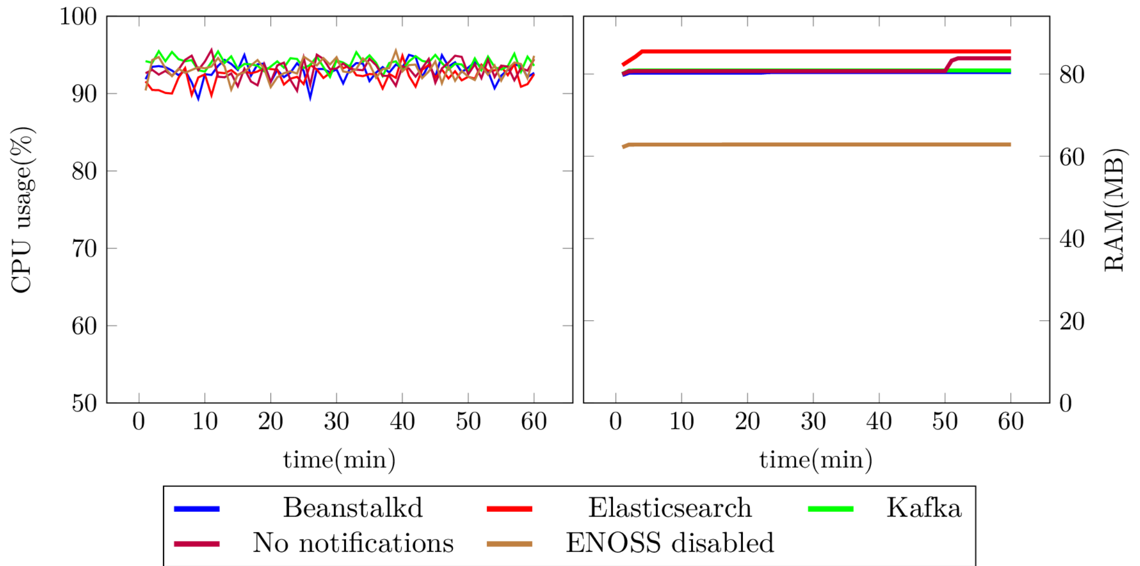

Figure 6.1: Left: CPU usage by Proxy server worker during experiment 1. Right: RAM usage by Proxy server worker during experiment 1.

Destination	Requests count	Requests per sec	Latency(seconds)				
			min	max	std dev	95%-idle	avg
ENOSS disabled	1 328 729	369	0.003	0.926	0.016	0.026	0.018
No notifications	1 159 416	322	0.003	1.138	0.018	0.030	0.021
Beanstalkd	1 140 780	316	0.003	0.861	0.015	0.031	0.022
Apache Kafka	916 787	254	0.004	0.953	0.017	0.039	0.027
Elasticsearch(https)	722 446	200	0.009	8.990	0.021	0.048	0.034

Table 6.1: Benchmark results for experiment 1.

Experiment 2: similarly to experiment 1, the ssbench scenario contains only tiny documents, and ENOSS is configured to create notifications for all events to one destination per event. This experiment targets situations when ENOSS does not have all needed information and needs to create one additional request to Swift.

Figure 6.2 contains information about CPU and RAM usage by the Proxy server worker. Comparing CPU and RAM usage during this experiment with the previous experiment, a conclusion can be drawn that CPU and RAM usage remained the same. Therefore, ENOSS did not need more RAM to gather needed information from object storage.

From table 6.2, containing benchmark results for Experiment 2, a deduction can be made that making an additional request to object storage impacts latency. The worst performance occurred when ENOSS published notifications to Elasticsearch, while Beanstalkd outperformed Kafka queue, the same as the previous experiment.

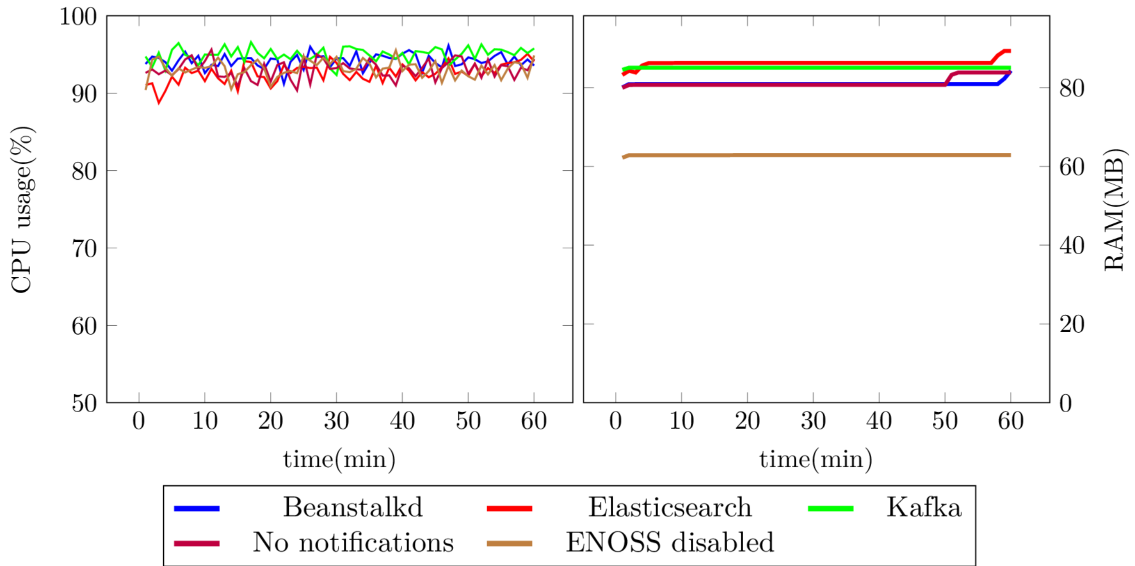

Figure 6.2: Left: CPU usage by Proxy server worker during experiment 2. Right: RAM usage by Proxy server worker during experiment 3.

Destination	Requests count	Requests per sec	Latency(seconds)				
			min	max	std dev	95%-idle	avg
ENOSS disabled	1 328 729	369	0.003	0.926	0.016	0.026	0.018
No notifications	1 159 416	322	0.003	1.138	0.018	0.030	0.021
Beanstalkd	732 845	203	0.006	0.638	0.016	0.046	0.034
Apache Kafka	570 528	158	0.007	0.909	0.018	0.056	0.043
Elasticsearch(https)	498 556	138	0.011	0.870	0.023	0.069	0.050

Table 6.2: Benchmark results for experiment 2.

Experiment 3: this experiment aims to simulate real-life usage of OpenStack Swift object storage. Scenario contains tiny(1-10KB), small(100KB) and medium(1MB) objects, CRUD operations distribution is 36% create, 27 % read, 18% update and 18% delete, where 52% of operations were with tiny objects, 35% with small and 13% with medium objects. The duration of each benchmark test during this experiment is 30 minutes.

In figure 6.3 can be seen that no CPU saturation was achieved since the scenario involves larger objects and non-read operations, which led to disks(and other components) bottlenecks. Figure 6.3 shows that during benchmark tests, when ENOSS was enabled, CPU usage increased. Computing(and then comparing) average CPU usage during different benchmark tests in this experiment was discovered that CPU usage increases from 10 to 13% when ENOSS is enabled.

In table 6.3 can be seen that the latency trend is similar to other experiments. Again, Beanstalkd shows the best performance and outperforms Kafka queue, while Elasticsearch is last.

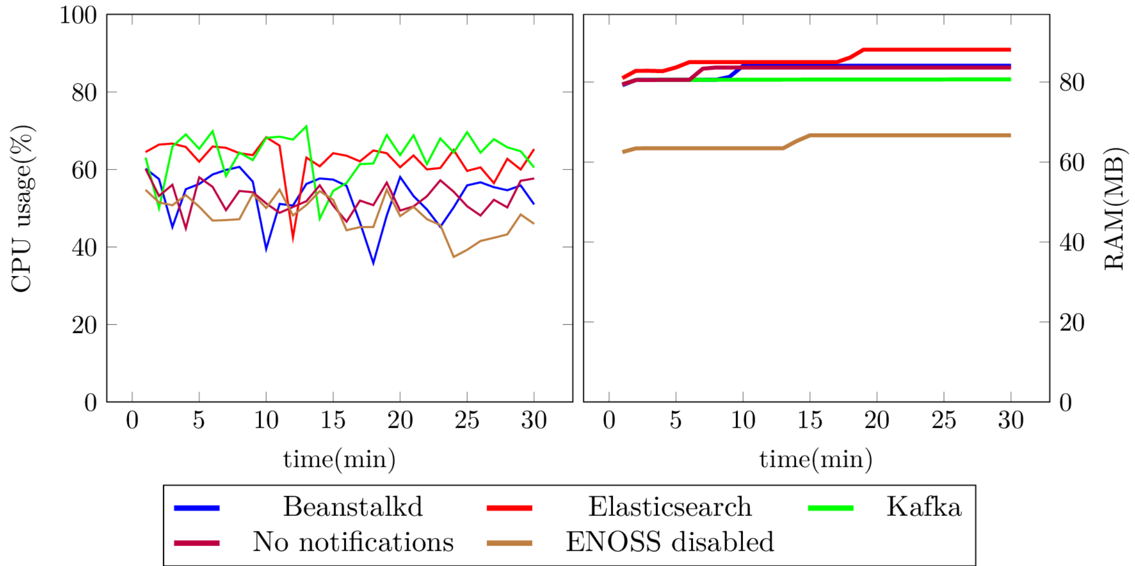

Figure 6.3: Left: CPU usage by Proxy server worker during experiment 3. Right: RAM usage by Proxy server worker during experiment 3.

Destination	Requests count	Requests per sec	Latency(seconds)				
			min	max	std dev	95%-idle	avg
ENOSS disabled	223 618	124	0.004	6.239	0.044	0.097	0.039
No notifications	204 998	113	0.004	3.682	0.042	0.105	0.043
Beanstalkd	204 894	113	0.004	3.866	0.046	0.102	0.042
Apache Kafka	193 471	107	0.004	5.104	0.047	0.104	0.045
Elasticsearch(https)	172 672	95	0.009	1.924	0.049	0.104	0.050

Table 6.3: Benchmark results for experiment 3.

Experiments results tracking average request count per second, and latency are combined and displayed in figures 6.4 and 6.5. From provided results can be concluded that:

- Beanstalkd has the best performance, while Elasticsearch is the slowest.
- ENOSS has a relatively small impact on latency for users that do not have enabled notifications.
- Making an additional request to Swift, in order to retrieve needed information, increases latency (in these experiments by 12-16 ms).
- Creating notification for each event in Swift resulted in only a 10-15 % performance decrease for Beanstalk and Kafka queue.

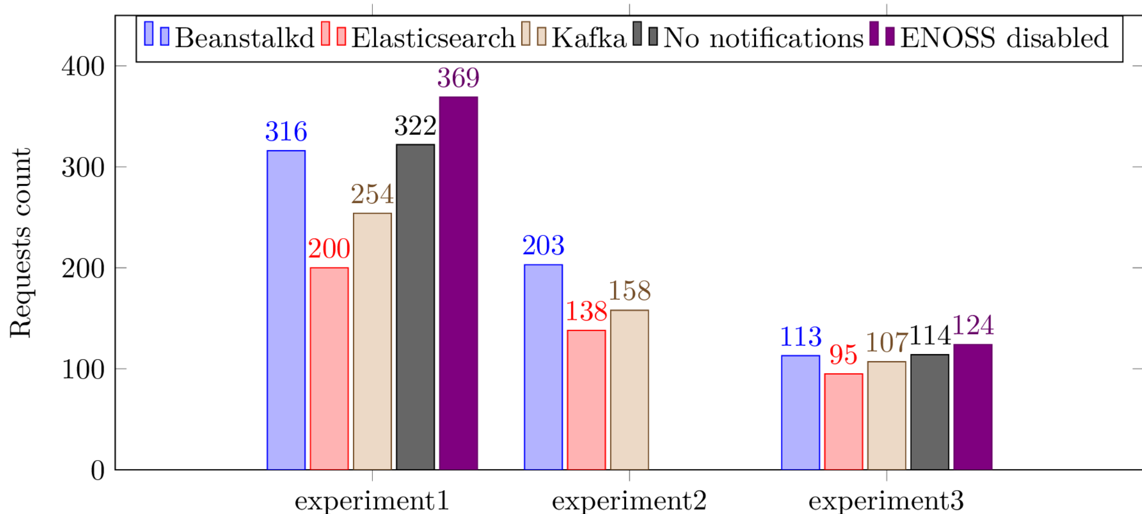

Figure 6.4: Combined experiments results tracking average number of requests per second. Bigger value is better.

6.4 Use cases and applications

ENOSS has multiple use cases. Some of possible use case scenarios where ENOSS can be applied are:

- **Event monitoring and alerting** - ENOSS can be expanded to publish notifications to more sophisticated event monitoring services with alerting (like Prometheus).
- **Anomaly detection** - since ENOSS is capable of publishing unsuccessful events, using filter **httpcodes** admin can set ENOSS to publish notifications about events involving internal errors (HTTP code 5xx) or any other unsuccessful HTTP codes.
- **Data theft detection** - user has a designated container for sensitive data. The container owner can „tell“ ENOSS which users should have access to the container,

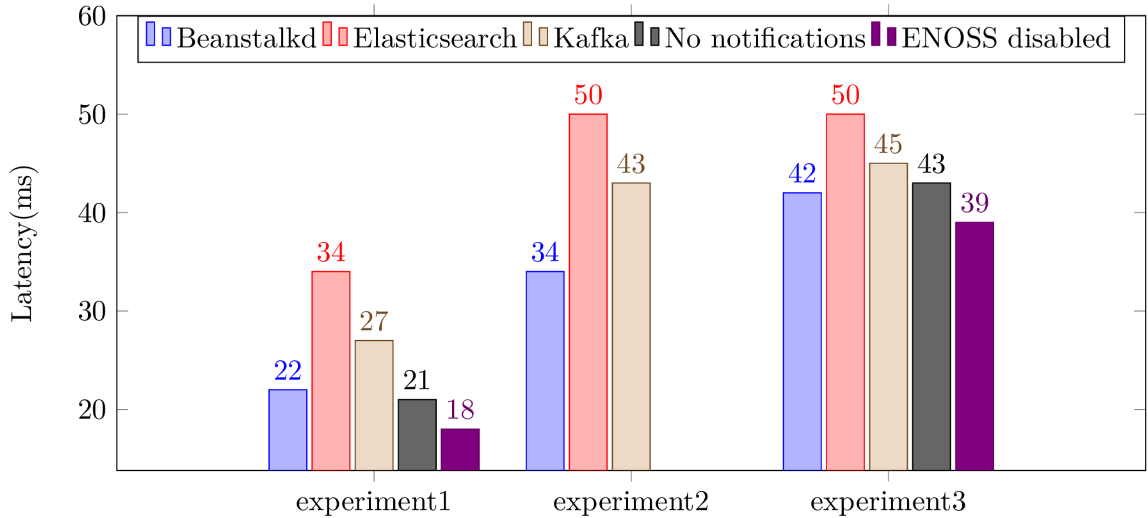

Figure 6.5: Combined experiments results tracking latency. Smaller value is better.

and if some unknown users, that are not in the list of users specified by the container's owner, somehow gain access to the container, then ENOSS will publish a notification about such event.

- **Data theft prevention** - user configures ENOSS to publish unauthorized events, which can result in the detection of the possible attempt of data theft.
- **Postprocessing** - user wants to search stored data using their metadata. The user configures ENOSS to publish events that store, modify and delete data in object storage. The destination of published events would be Elasticsearch or some other custom destination capable of full-text search.

Chapter 7

Conclusion

This thesis aimed to provide a way for users to retrieve information about events occurring in parts of object storage that they own/have access to. The primary destination for such information was the Beanstalkd queue. This goal was achieved by creating OpenStack Swift middleware called ENOSS(Event Notifications in OpenStack Swift).

ENOSS allows users to specify which event types should be published. Furthermore, ENOSS offers users additional even filtering using filter rules such as prefix, suffix, size, returned HTTP status code, and users list. All these filters, including event type, can be combined, allowing users to specify events that need to be published more precisely.

ENOSS offers several destinations to which notifications can be published. From popular stream-processing platform Apache Kafka to lightweight and extremely fast Beanstalkd working queue. ENOSS also supports publishing notifications to Elasticsearch, with many applications (from simple logging to more complex usage like indexing metadata).

Another ENOSS feature is the support of different payload types. Since ENOSS supports multiple destinations and has multiple applications, notification payloads may differ based on ENOSS usage.

One of the key advantages and ENOSS features is support for custom destinations, payloads, and filters. In addition, ENOSS offers effortless creation and integration of new destinations, payloads, and filters using predefined interfaces and rules.

ENOSS configuration and notification payload are compatible with AWS S3 Event Notification. This key feature will allow future users to easily manipulate ENOSS since S3 Notifications are well documented and widely used. Additionally, OpenStack Swift and OpenIO SDS will be more competitive in the market since notifications were not supported in these object storages until now. S3 compatibility will also allow more accessible transition users from AWS S3 to OpenStack Swift or OpenIO SDS.

Benchmarking showed that even during stress tests when all events were published, sending notifications had a pretty minor impact on object storage performance. In the worst-case scenario, ENOSS needs to make an additional request to object storage to decide if an event should be published or to create a notification payload. From the experiment results simulating everyday daily use of object storage, it was concluded that publishing notifications had a pretty minor impact on latency - between 3ms (for Beanstalkd queue) and 11ms for Elasticsearch.

The work of the thesis was presented at the student conference of innovation, technology, and science Excel@FIT2022 under the name „ENOSS - Event Notifications in OpenStack Swift“. The work received an award by the expert commission for the beneficial extension of the platform OpenStack.

Possible future work and ENOSS improvements include creating support for new destinations(MQTT, Redis, NSQ), payloads, and filters. One of the current ENOSS limitations is that some supported destinations can publish notifications to a single server address. Support for secondary destination servers would be beneficial.

Bibliography

[1] *Amazon S3 Event notification types and destinations* [online]. [cit. 2021-12-27]. Available at: https://docs.aws.amazon.com/AmazonS3/latest/userguide/notification-how-to-event-types-and-destinations.html.

[2] *Amazon S3 Event Notifications* [online]. [cit. 2021-12-27]. Available at: https://docs.aws.amazon.com/AmazonS3/latest/userguide/NotificationHowTo.html.

[3] *Beanstalkd* [online]. [cit. 2021-12-27]. Available at: https://beanstalkd.github.io/.

[4] *Beanstalkd protocol* [online]. [cit. 2021-12-27]. Available at: https://raw.githubusercontent.com/beanstalkd/beanstalkd/master/doc/protocol.txt.

[5] *CloudEvents* [online]. [cit. 2021-12-27]. Available at: https://cloudevents.io/.

[6] *CloudEvents Specification* [online]. [cit. 2021-12-27]. Available at: https://github.com/cloudevents/spec/blob/v1.0.1/spec.md.

[7] *Contributing to OpenStack Swift* [online]. [cit. 2022-05-11]. Available at: https://docs.openstack.org/swift/latest/contributor/contributing.html.

[8] *Eventlet* [online]. [cit. 2021-12-27]. Available at: https://eventlet.net/.

[9] *MinIO - Object Storage Monitoring* [online]. [cit. 2021-12-27]. Available at: https://min.io/product/object-storage-performance-monitoring.

[10] *MinIO Object Storage* [online]. [cit. 2021-12-27]. Available at: https://min.io/product/overview.

[11] *OpenIO - Key Characteristics* [online]. [cit. 2021-12-27]. Available at: https://docs.openio.io/latest/source/arch-design/overview.html.

[12] *OpenIO SDS: Core Concepts* [online]. [cit. 2021-12-27]. Available at: https://docs.openio.io/latest/source/arch-design/sds_concepts.html.

[13] *OpenIO Services* [online]. [cit. 2021-12-27]. Available at: https://docs.openio.io/latest/source/arch-design/sds_services.html.

[14] *Swift Architectural Overview* [online]. [cit. 2021-12-27]. Available at: https://docs.openstack.org/swift/xena/overview_architecture.html.

[15] *Swift: Large object support* [online]. [cit. 2021-12-27]. Available at: https://docs.openstack.org/swift/xena/admin/objectstorage-large-objects.html.

[16] *Swift Middleware and Metadata* [online]. [cit. 2021-12-27]. Available at: https://docs.openstack.org/swift/xena/development_middleware.html.

[17] *SwiftStack Swift Benchmarking Suite* [online]. [cit. 2022-05-11]. Available at: https://pypi.org/project/ssbench/.

[18] *Teratec OpenIO* [online]. [cit. 2021-12-27]. Available at: https://teratec.eu/gb/qui/membres_Openio.html.

[19] *How To Install and Use Beanstalkd Work Queue on a VPS* [online]. 2013 [cit. 2021-12-27]. Available at: https://www.digitalocean.com/community/tutorials/how-to-install-and-use-beanstalkd-work-queue-on-a-vps.

[20] *OpenStack Swift proposed solution 1* [online]. 2015 [cit. 2021-12-27]. Available at: https://review.opendev.org/c/openstack/swift/+/196755.

[21] *OpenIO Core Solution Description*. OpenIO, 2016. Available at: https://www.openio.io/resources/.

[22] *OpenStack Swift proposed solution 2* [online]. 2016 [cit. 2021-12-27]. Available at: https://review.opendev.org/c/openstack/swift/+/388393.

[23] *Software Defined Storage (SDS)* [online]. 2016 [cit. 2021-12-27]. Available at: http://www.sjaaklaan.com/?e=167.

[24] *Build a High-Performance Object Storage-as-a-Service Platform with Minio*. Intel Corporatio, 2017. Available at: https://min.io/resources/docs/CPG-MinIO-reference-architecture.pdf.

[25] *OpenIO Next-Generation Object Storage and Serverless Computing Explained*. OpenIO, 2018. Available at: https://www.openio.io/wp-content/uploads/2018/03/OpenIO-NextGenObjectStorageAndServerlessComputingExplained.pdf.

[26] *What is software-defined storage?* [online]. 2018 [cit. 2021-12-27]. Available at: https://www.redhat.com/en/topics/data-storage/software-defined-storage.

[27] *What is event-driven architecture?* [online]. 2019 [cit. 2021-12-27]. Available at: https://www.redhat.com/en/topics/integration/what-is-event-driven-architecture.

[28] *High Performance Object Storage*. MinIO Inc., 2021. Available at: https://min.io/resources/docs/MinIO-high-performance-object-storage.pdf.

[29] Arnold, J. *OpenStack Swift: Using and Administering and and Developing for Swift Object Storage*. O'Reilly Media, Inc., 2014. ISBN 978-1-4919-0082-6.

[30] Chen and Robin, Y.-F. The Growing Pains of Cloud Storage. *IEEE Internet Computing*. 2015, vol. 19, no. 1, p. 4–7. DOI: 10.1109/MIC.2015.14.

[31] Coyne, L., Dain, J., Forestier, E., Guaitani, P., Haas, R. et al. *IBM Software-Defined Storage Guide*. IBM, 2018. ISBN 0738457051. Available at: https://www.redbooks.ibm.com/redpapers/pdfs/redp5121.pdf.

[32] Gracia Tinedo, R., Sampé, J., París, G., Sánchez Artigas, M., García López, P. et al. Software-defined object storage in multi-tenant environments. *Future Generation Computer Systems*. 2019, vol. 99, p. 54–72. DOI: https://doi.org/10.1016/j.future.2019.03.020. ISSN 0167-739X. Available at: https://www.sciencedirect.com/science/article/pii/S0167739X18322167.

[33] Gulabani, S. *Amazon S3 Essentials*. Packt Publishing, 2015. ISBN 9781783554898.

[34] Kapadia, A., Rajana, K. and Varma, S. *OpenStack Object Storage (Swift) Essentials*. Packt Publishing, 2015. ISBN 978-1-78528-359-8.

[35] Kapadia, A., Varma, S. and Rajana, K. *Implementing Cloud Storage with OpenStack Swift*. Packt Publishing, 2014. ISBN 9781782168058.

[36] Macedo, Ricardo, Paulo, ao, J., Pereira et al. A Survey and Classification of Software-Defined Storage Systems. New York, NY, USA: Association for Computing Machinery. 2020, vol. 53, no. 3. DOI: 10.1145/3385896. ISSN 0360-0300. Available at: https://doi.org/10.1145/3385896.

[37] Mesnier, M., Ganger, G.R., Riedel et al. Object-based storage. *IEEE Communications Magazine*. 2003, vol. 41, no. 8, p. 84–90. DOI: 10.1109/MCOM.2003.1222722.

[38] O'Reilly, J. *Network Storage: Tools and Technologies for Storing Your Company's Data*. Elsevier, 2017. ISBN 9780128038635; 0128038632.

[39] Patil, A., Rangarao, D., Seipp, H., Lasota, M. and Santos..., R. M. dos. *Cloud Object Storage as a Service: IBM Cloud Object Storage from Theory to Practice*. IBM, 2017. ISBN 0738442453.

[40] Raj, P., Raman, A. and Subramanian, H. *Architectural Patterns*. Packt Publishing, 2017. ISBN 9781787287495.

[41] Zheng, Qing, Chen, Haopeng, Wang et al. *COSBench: A Benchmark Tool for Cloud Object Storage Services*. 2012. 998-999 p. ISBN 978-1-4673-2892-0.

Appendix A

Contents of the included storage media

```
/
├─ thesis -- thesis documentation source codes
├─ xvasil03.pdf -- Thesis document
└─ src -- source codes
   ├─ enoss -- ENOSS repository
   │  ├─ benchmark -- benchmark results
   │  │  ├─ expr1 -- benchmark resuls for experiment 1
   │  │  ├─ expr2 -- benchmark resuls for experiment 2
   │  │  ├─ expr3 -- benchmark resuls for experiment 3
   │  │  └─ k8s -- k8s sources used for benchmarking
   │  ├─ demo -- OpenIO SDS and OpenStack Swift demo with enabled ENOSS
   │  ├─ enoss -- ENOSS source codes
   │  │  ├─ enoss.py -- ENOSS middleware
   │  │  ├─ destinations -- destination handlers
   │  │  ├─ payloads -- payload handlers
   │  │  └─ filter_rules -- filter rule handlers
   │  ├─ etc/swift/enoss -- configuration
   │  ├─ test -- ENOSS tests
   │  │  ├─ functional
   │  │  └─ unit
   │  └─ examples -- Several examples of notifications configurations
   └─ mqtt-to-beanstalkd -- source codes for MinIO proxy from MQTT to Beanstalkd
```

Appendix B

Repository and Usage Guide

ENOSS repository is publicly avaliable at the Github https://github.com/xvasil03/enoss.

ENOSS demo is stored in `enoss/demo`. Demo contains following containers:

- **Beanstalkd listener** - contains beanstalkd service, receives notifications from OpenIO SDS and Swift and prints then to stdout (docker logs).
- **ENOSS Swift** - container with OpenStack Swift with enabled ENOSS. Can publish notifications to beanstalkd. After container init runs UNIT and FUNCTIONAL ENOSS tests.
- **ENOSS OpenIO** - container with OPENIO SDS with enabled ENOSS. Can publish notifications to beanstalkd.
- **Demo worker** - waits 2 minutes for other containers to initialize then runs demo scripts `enoss/demo/demo_openio.sh` and `enoss/demo/demo_swift.sh` Demo scripts will enable notifications on specific container, read stored configuration from object storage, and then create event which will trigger notification. Connect to this container and communicate with OpenStack Swift using hostname `swift-service` and with OpenIO SDS using hostname `openio-service`.

Run demo using `docker-compose up`.

Mqtt-to-Beanstalkd Demo

1. `cd mqtt-to-beanstalkd`
2. `docker-compose build`
3. `docker-compose run`

ENOSS Build - output located in `enoss/dist`

1. `cd enoss`
2. `pip3 install -U setuptools`
3. `pip3 install wheel`
4. `python3 setup.py sdist bdist_wheel`

Instalation - OpenIO SDS(Python 2) - requires pip2.

1. `cd enoss`
2. `pip install ./dist/*whl`
3. `pip install -r ./requirements-py2.txt`
4. `store configurations files from enoss/etc (needed for ENOSS configuration)`

Instalation - OpenStack Swift (Python 3)

1. `cd enoss`
2. `pip3 install enoss/`
3. `pip3 install -r ./requirements.txt`
4. `store configurations files from enoss/etc (needed for ENOSS configuration)`

Adding ENOSS to Proxy server - `enoss/etc` contains example of ENOSS configuration for OpenStack Swift.

1. `Add enoss to proxy server pipeline (behind s3api and bulk middleware) in proxy-server.conf./`
2. `Configure ENOSS using section [filter:enoss] in proxy-server.conf.`
3. `Configure destinations configuration using file speicified in destinations_conf_path.`
4. `Restart proxy server (swift-init proxy restart).`

Adding ENOSS to Proxy server

1. `Add enoss to proxy server pipeline (behind s3api and bulk middleware) in proxy-server.conf.`
2. `Configure ENOSS using section [filter:enoss] in oio-proxy-server.conf.`
3. `Configure destinations configuration using file speicified in destinations_conf_path.`
4. `Restart oio-proxy server.`

Enabling notification configuration on a container

1. `Store notification configuration using ENOSS POST API.`
2. `Check if test event was sent to specified destination in stored configuration.`

Appendix C

Excel@FIT Article

Article with title: „ENOSS - Event Notifications in OpenStack Swift“ published and presented on April 30, 2022 at student conference Excel@FIT2022.

BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2022
http://excel.fit.vutbr.cz

ENOSS - Event Notifications in OpenStack Swift

Nemanja Vasiljević*

Abstract

Currently, object storage OpenStack Swift does not provide any pieces of information to users about events that occurred in storage they own/have access to. For example, users do not have information when the content of their object storage is accessed, changed, created, or deleted. This paper aims to create a solution that will send notifications about events that occurred in OpenStack Swift to user-specified destinations. The proposed solution, using metadata, allows users to specify where and which event should be published based on even types (read, create, modify, delete) and other properties such as object prefix, suffix, size. It also offers multiple destinations(Beanstalkd queue, Kafka, etc.) to which notifications can be published. The solution is fully compatible with AWS S3 Event Notifications and, compared to AWS, supports more destinations, event types, filters and allows unsuccessful events to be published. Event notification can be used for monitoring, automatization, and serverless computing (similar to AWS Lambda).

Keywords: Event — Notifications — OpenStack Swift

Supplementary Material: Github repository

*xvasil03@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

Object storage is a data storage architecture that manages data as objects, and each object typically includes data itself and some additional information stored in objects metadata. Since object storage is often used in cloud computing, data are stored in remote locations where users do not have direct and complete access. Some users or external services might want to receive information about specific events in storage where their data are located. For example, there is no easy way to detect changes in a specific container except to list its content and compare timestamps, which can be complex, slow, and inefficient if there are many objects in storage.

The importance of this work is to provide event information to users in OpenStack Swift, which will allow users to react to those events, create more sophisticated backend operations, postprocessing and automatization, or possibly prevent/detect unwanted actions. In addition, providing event notifications will allow users to have a better picture of what is going on in their storage and improve monitoring in object storage.

Users can be interested in only specific events, for example, creating new objects in the container. Therefore, the proposed solution must allow event filtering based on event type and other properties (e.g., object name prefix/suffix/size). Furthermore, since object storage has multiple users, each user can have different requirements for event notification, and the proposed solution must be prepared for it.

Application of event notifications varies from sim-

ple monitoring or webhook to more sophisticated applications such as serverless computing like AWS Lambda. Therefore the structure of event notification may differ based on the application and destination to which it is published. Therefore, the proposed solution must be ready to publish event notifications to different destinations and event notification structures.

AWS S3 object storage is one of the most popular storage with their API, supported by many other object storages, including OpenStack Swift. Since AWS S3 supports event notifications, it would be ideal if the proposed solution in OpenStack is compatible with the S3 event notification protocol. As a result, not only that OpenStack Swift would offer the same functionality as AWS S3 (that currently lacks), but the protocol would be compatible with AWS S3, which would allow more accessible transfer users from AWS S3 to OpenStack Swift. Therefore, users would not have to learn additional protocols, instead can follow the existing AWS S3, which is most popular and well documented.

This work consists of six chapters. Chapter 1 introduces the motivation, defines problems and desired objectives. Chapter 2 describes object storage OpenStack Swift, its data model, main processes, and describes middlewares and metadata within OpenStack Swift. Chapter 3 analyzes and compares existing solution for given problem. Chapter 4 describes proposed solution - ENOSS, its key features, configuration and interfaces. Chapter 5 summarize proposed solution, highlights results of this work and its contributions. Chapter 5 contains acknowledgments to people that helped me to create this paper.

2. OpenStack Swift

OpenStack Swift is open-source object storage developed by Rackspace, a company that, together with NASA, created the OpenStack project. After becoming an open-source project, Swift became the leading open-source object storage supported and developed by many famous IT companies, such as Red Hat, HP, Intel, IBM, and others.

OpenStack Swift is a multi-tenant, scalable, and durable object storage capable of storing large amounts of unstructured data at low cost[1].

2.1 Data model

OpenStack Swift allows users to store unstructured data objects with a canonical name containing *account*, *container* and *object* in given order[1]. The account names must be unique in the cluster, the container name must be unique in the account space, and the object names must be unique in the container. Other than that, if containers have the same name but belong to a different account, they represent different storage locations. The same principle applies to objects. If objects have the same name but not the same container and account name, then these objects are different.

Accounts are root storage locations for data. Each account contains a list of containers within the account and metadata stored as key-value pairs. Accounts are stored in the account database. In OpenStack Swift, account is **storage account** (more like storage location) and **do not represent a user identity**[1].

Containers are user-defined storage locations in the account namespace where objects are stored. Containers are one level below accounts; therefore, they are not unique in the cluster. Each container has a list of objects within the container and metadata stored as key-value pairs. Containers are stored in container database[1].

Objects represent data stored in OpenStack Swift. Each object belongs to one (and only one) container. An object can have metadata stored as key-value pairs. Swift stores multiple copies of an object across the cluster to ensure durability and availability. Swift does this by assigning an object to *partition*, which is mapped to multiple drives, and each driver will contain object copy[1].

2.2 Main processes

The path towards data in OpenStack Swift consists of four main software services: **Proxy server**, **Account server**, **Continaer server** and **Object server**. Typically Account, Container and Object server are located on same machine creating **Storage node**.

Proxy server is the service responsible for communication with external clients. For each request, it will look up storage location(node) for an account, container, or object and route the request accordingly[2]. The proxy server is responsible for handling many failures. For example, when a client sends a `PUT` request to OpenStack Swift, the proxy server will determine which nodes store the object. If some node fails, a proxy server will choose a hand-off node to write data. When a majority of nodes respond successfully, then the server proxy will return a success response code[1].

Account server stores information about containers in a particular account to SQL database. It is responsible for listing containers. It does not know where specific containers are, just what containers are in an account[2].

Container server is similar to the account server, except it is responsible for listing objects and also does not know where specific objects are[2].

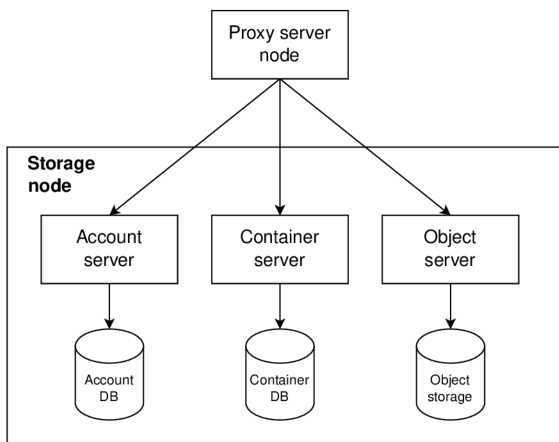

Figure 1. OpenStack Swift servers architecture.

Object Server is blob storage capable of storing, retrieving, and deleting objects. Objects are stored as binary files to a filesystem, where metadata are stored in the *file's extended attributes (xattrs)*. This requires a filesystem with support of such attributes. Each object is stored using a hash value of object path (account/container/object) and timestamp. This allows storing multiple versions of an object. Since last write wins (due to timestamp), it is ensured that the correct object version is served[2].

2.3 Middleware

Using Python WSGI middleware, users can add functionalities and behaviors to OpenStack Swift. Most middlewares are added to the Proxy server but can also be part of other servers (account server, container server, or object server).

Middlewares are added by changing the configuration of servers. Listing 1 is shows how to add *webhook middleware* to proxy server by changing its pipeline (*pipeline:main*). Middlewares are executed in the given order (first will be called webhook middleware, then proxy-server middleware).

Some of the middlewares are required and will be automatically inserted by swift code[3].

Listing 1. Example of proxy server configuration (proxy-server.conf).

```
[DEFAULT]
log_level = DEBUG
user = <your-user-name>

[pipeline:main]
pipeline = webhook proxy-server

[filter:webhook]
use = egg:swift#webhook

[app:proxy-server]
use = egg:swift#proxy
```

Interface - OpenStack Swift servers are implemented using Python WSGI applications. Therefore only Python WSGI middlewares are accepted in OpenStack Swift.

Listing 2 provides example of simplified *healthcheck middleware*. The constructor takes two arguments, the first is a WSGI application, and the second is a configuration of middleware defined using Python Paste framework in *proxy-server.conf*. Middleware must have a call method containing the request environment information and response from previously called middleware. Middleware can perform some operations and call the next middleware in the pipeline or intercept a request. In the healthcheck example, if the path directs to `/healthcheck` , the middleware will return `HTTP Response`, and other middlewares in the pipeline will not be called.

Method `filter_factory` is used by the Python Paste framework to instantiate middleware.

```
import os
from swift.common.swob import Request,
    Response

class HealthCheckMiddleware(object):
  def __init__(self, app, conf):
    self.app = app

  def __call__(self, env, start_response):
    req = Request(env)
    if req.path == '/healthcheck':
      return Response(request=req, body=b"
    OK", content_type="text/plain")(env,
    start_response)
    return self.app(env, start_response)

def filter_factory(global_conf, **
    local_conf):
  conf = global_conf.copy()
  conf.update(local_conf)

  def healthcheck_filter(app):
    return HealthCheckMiddleware(app, conf)
  return healthcheck_filter
```

Listing 2. Example of healthcheck middleware in OpenStack Swift

2.4 Metadata

OpenStack Swift separates metadata into 3 categories based on their use:

`User Metadata` - User metadata takes form

`X-<type>-Meta-<key>:<value>`

where `<type>` represent resource type(i.e. account, container, object), and `<key>` and `<value>` are set by user. User metadata remain persistent until are updated using new value or removed using header `X-<type>-Meta-<key>` with no value or a header

X-Remove-<type>-Meta-<key>:<ignored--value>.

System Metadata - System metadata takes form X-<type>-Sysmeta-<key>:<value> where <type> represent resource type(i.e. account, container, object) and <key> and <value> are set by internal service in Swift WSGI Server. All headers containing system metadata are deleted from a client request. System metadata are visible only inside Swift, providing a means to store potentially sensitive information regarding Swift resources.

Object Transient-Sysmeta - This type of metadata have form of X-Object-Transient--Sysmeta-<key>:<value>. Transient-sysmeta is similar to system metadata and can be accessed only within Swift, and headers containing Transient-sysmeta are dropped. If middleware wants to store object metadata, it should use transient-sysmeta[3].

3. Existing solutions

There is no official OpenStack solution that satisfies all requirements mentioned in section 1, although some of the existing programs can be used to solve some of the problems partially.

Webhook middleware described in 2.3 can be used for detection of new objects in specific container. With some tweaks, it could detect object deletion and modification too. One of the many limitations of this middleware is the lack of support for different destinations (it can publish notification only to one type of destination), no filtering, a single type of event notification structure, and incompatibility with AWS S3.

OpenStack Swift attempts - OpenStack Swift is aware of the lack of event notifications, and in order to solve it, they crated specification for this problem [4]. This specification was mainly focused on detection changes inside the specific container (creation, modifying, and deletion of objects). There were two attempts to solve this problem.

- **First attempt** [5] - allowed sending notifications only to Zaqar queue[1] and had very simple event notification strucuture. Notification contained only informations about names of account, container and object on which event occured and name of HTTP method.
- **Second attempt** [6] - was more sophisticated solution that was design to support multiple destinations to which notification can be published. The event notification structure was expanded for information such as eTag (MD5 checksum) and transaction id. The author introduced the concept of "notification policy" which represented the configuration of event notifications. One of the main critiques made by code reviewers was incompatibility with AWS S3 storage.

[1] Zaqar queue - OpenStack Messaging https://wiki.openstack.org/wiki/Zaqar

Both attempts are outdated, and due to a lack of interest from users/operators, OpenStack Swift halted development for this problem.

ENOSS - my solution, code name ENOSS, satisfies all requirements specified in section 1. Key features are events filtering, support of multiple destinations, AWS S3 compatibility, different event notification structure, the definition of interfaces for future expansions of filters, destinations, and event notification structure, and design that allows its effortless expansions.

4. ENOSS

ENOSS (Event Notifications in OpenStack Swift) is a program that enables publishing notifications containing information about occurred events in OpenStack Swift. It is implemented in the form of Python WSGI middleware and is located in the Proxy server pipeline. Since the Proxy server communicates with external users, by placing ENOSS in its pipeline, ENOSS can react to every user request to OpenStack Swift, which makes the Proxy server an ideal place for ENOSS.

4.1 Key featrues

The proposed middleware heavily utilizes containers/buckets and accounts metadata. Information specifying which event should be published and where is stored in metadata of upper level. For publishing events regarding objects, the configuration is stored in container metadata, and for container events, the configuration is stored at an account level.

Multi user environment - since many different users communicate with OpenStack Swift, each of them can be interested in different event notifications. ENOSS solves this problem by allowing each container and account to have its notification configuration.

Event filtering - one of the main requirements for event notifications is allowing users to specify for which events should notifications be published - i.e., event filtering. ENOSS allows users to specify which types of events should be published (object/container creation, deletion, access, ...). ENOSS goes a little further and allows users to specify rules that must be satisfied for event notification to be published. Some rule operators are object/container name prefix/suffix and object size. For example, using this feature,

users can select only events regarding objects bigger than 50Mb (operator: object size) or events regarding pictures (operator: object suffix).

Multiple destinations - since event notifications have multiple applications, from monitoring to automatization, it is essential that the proposed solution can publish a notification to multiple different destinations. ENOSS is fully capable of publishing event notifications to many different destinations (e.g., Beanstalkd queue, Kafka). In ENOSS, publishing notifications about a single event is not limited to only one destination. If a user wishes, it can be published to multiple destinations per single event. This feature allows event notification to be used for multiple applications simultaneously.

Event notification structure - depending on the application of event notification structure of notification may differ. Therefore, ENOSS supports several different notification structures, and using event notification configuration, ENOSS allows users can select a type of event notification structure.

AWS S3 compatibility - ENOSS puts a big emphasis on support and compatibility with AWS S3. The structure of event configuration and event names in ENOSS is compatible with AWS S3. ENOSS also supports all filtering rules from AWS S3, and the default event notification structure is compatible with AWS S3. This is all done to ease transfer users from AWS S3 to OpenStack Swift. Using the existing, well-documented protocol, users will have an easier time learning and using event notifications in OpenStack Swift.

4.2 Configuration

Setting event notification configuration - in order to enable event notifications on specific container, first step is to store its configuration. For this purpose ENOSS uses API:

```
POST /v1/<acc>/<cont>?notification
```

Figure 2 describes process of storing event configuration. Authorized user sends event notification configuration using request body, ENOSS perform validation, if configuration is valid, ENOSS will store configuration to container system metadata, otherwise it will return unsuccessful HTTP code.

Reading stored event notification configuration - Event notifications configuration can contain sensitive information. Since ENOSS stores configuration to storage using system metadata, which can be accessed only by application within OpenStack Swift, it disables reading stored configuration by simple GET/HEAD requests. For this purpose ENOSS offer API

```
GET /v1/<acc>/<cont>?notification
```

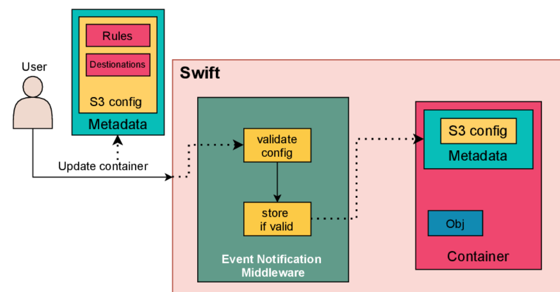

Figure 2. Process of setting event notification configuration in ENOSS.

For security reasons, ENOSS allow only users with write rights to read stored configuration.

Configuration structure - Listing 3 describes event notification configuration. `<Target>` represent targeted destination where event notifications will be sent (e.g., Beanstalkd, Elasticsearch). `<FilterKey>` is a unique name of a filter containing rules that must be satisfied in order to publish events.

Event type takes form :

```
s3:<Type><Action>:<Method>
```

and are compatible with Amazon S3 event types. Type represents resource type (object, bucket), action represent action preformed by user and can have values: `Created`, `Removed`, `Accessed`. The method represents the REST API method performed by a user: `Get`, `Put`, `Post`, `Delete`, `Copy`, `Head`. For example, if a new object was created, even type would be described as `s3:ObjectCreated:Put`. To match event type regardless of API method assign value $*$ to `<Method>`.

```
{
  "<Target>Configrations": [
    {
      "Id": "configration id",
      "TargetParams": "set of key-value
          pairs, used specify dynamic
          parameters of targeted
          destination (e.g., name of
          beanstalkd tube or name of the
          index in Elasticsearch)",
      "Events": "array of event types that
          will be published",
      "PayloadStructure": "type of event
          notification structure: S3 or
          CloudEvents (default value S3)",
      "Filter": {
        "<FilterKey>": {
          "FilterRules": [
            {
              "Name": "filter operations (i
                  .e. prefix, sufix, size)
                  ",
              "Value": "filter value"
            }, ...
    }
```

Listing 3. Strucute of event notification configuration

4.3 Interfaces

One of the use cases of ENOSS can be publishing event notifications to custom destinations / currently unsupported destinations. In order to ease future development and support of new destinations, as well as different message structures and filters, ENOSS defined class interfaces and a set of rules needed to be followed in order to integrate new destination/message structure/filter to ENOSS.

DestinationI - is an interface specifying class that will be used for sending event notifications to the desired destination. The constructor receives configuration(dict), which can contain information needed for creating a connection with the desired destination(address, port, authentication,...). Configuration is loaded from ENOSS middleware configuration, which is loaded by the Proxy server. Method `send_notification` receives notification(dict) and its task is to send notification to desired destination.

```
class DestinationI(object, metaclass=abc.ABCMeta):
    @abc.abstractmethod
    def __init__(self, conf):
        raise NotImplementedError('__init__ is not implemented')

    @abc.abstractmethod
    def send_notification(self, notification):
        raise NotImplementedError('send_notification is not implemented')
```

Listing 4. Interface of class used for sending notification message to desired destination

PayloadI - is an interface specifying class that will be used for creating notification payload. When event notifications are configured on a container or account, ENOSS sends test notifications to all specified destinations in configuration. This way, it allows users to check if they successfully configured event notifications. Method `create_test_payload` is used for this purpose. One of the parameters is `request`, which contains all information about the incoming request(e.g., user IP address, incoming headers) as well as information about Swift response(e.g., headers, status code). `invoking_configuration` contains informations about stored event notifications configuration. When an event occurs on a container/account with enabled event notifications, ENOSS checks if notification for such event should be published based on event notification configuration. If yes, method `create_payload` will be used to create notification payload.

```
class PayloadI(object, metaclass=abc.ABCMeta):
    def __init__(self, conf):
        self.conf = conf

    @abc.abstractmethod
    def create_test_payload(self, app, request, invoking_configuration):
        raise NotImplementedError('create_test_payload is not implemented')

    @abc.abstractmethod
    def create_payload(self, app, request, invoking_configuration):
        raise NotImplementedError('create_payload is not implemented')
```

Listing 5. Interface of class used to create notification payload

RuleI - is an interface specifying class that represents user-specified rule which must be satisfied in order to publish event notification. The constructor receives value, which is read from the event notification configuration. The call method has access to all information about the request, which allows implementing rules about, e.g., user IP address, return code, object prefix/suffix/length, etc.

```
class RuleI(object, metaclass=abc.ABCMeta):
    def __init__(self, value):
        self.value = value

    @abc.abstractmethod
    def __call__(self, app, request):
        raise NotImplementedError('__call__ is not implemented')
```

Listing 6. Interface of class representing filter rule.

4.4 Integration of new class implementing interface

- Often, implementation of new classes is way easier than its integration with a given system.

In the ENOSS case, where everything moves around event notifications configuration, which users specify, this problem can be challenging. ENOSS was designed with this problem in mind. In order to effortlessly integrate new classes that implement interfaces specified in 4.3, several steps/rules must be followed:

Class naming - To integrate classes with ENOSS and allow users to use them in event notifications configuration, the class name must have a proper suffix. Name of classes implementing interface `DestinationI` must have suffix `Destination` (e.g. name of class sending notifications to Kafka would be `KafkaDestination`). Same principle applies for other interfaces, for payload suffix is `Payload` and for filter rule suffix is `Rule`

Names in event notifications configuration - since class names in ENOSS must follow the above-specified rules, they are automatically integrated into ENOSS. Classes are connected with event notifications configuration using the class prefix name, i.e., without the class suffix described above.

In listing 7, `KafkaConfigurations` means that class `KafkaDestination` will be used for sending notification, `"PayloadStructure": "S3"` means that `S3Payload` will be used for creating notification payload, and filter rule with `"Name": "suffix"` will use class `SuffixRule`.

```
{
  "KafkaConfigurations": [
    {
      "Id": "kafka - example",
      "Events": "*",
      "PayloadStructure": "S3",
      "Filter": {
        "FilterExample": {
          "FilterRules": [
            {
              "Name": "suffix",
              "Value": ".jpg"
            }
]}}}]}
```

Listing 7. Example of event notifications configuration

4.5 Notification payload structure

Default notification payload structure is compatible with AWS S3. Listing 8 shows example of published S3 notification. It contains information about:

- event - name(type), time and source
- user - id and ip address
- request id
- container/bucket - name and owner
- object - name(key), size, eTag(id), version and sequencer

The sequencer key allows a way to determine the sequence of events. Since event notifications aren't guaranteed to arrive in the same order that the events occurred, sequencer can be used to determine the order of events for a given object key. Therefore events that create objects (PUTs) and delete objects contain a sequencer[7].


```
{
 "Records":[
   {
      "eventVersion":"2.2",
       "eventSource":"swift:s3",
      "eventTime":"2022-04-12T14
          :04:48.189110",
      "eventName":"s3:ObjectCreated:Put",
      "userIdentity":{
        "principalId":"test,test:tester,
            AUTH_test"
      },
      "requestParameters":{
        "sourceIPAddress":"::ffff
            :127.0.0.1"
      },
      "responseElements":{
        "x-amz-request-id":"
            tx9a657c6753dd475699128
            -0062558700"
      },
      "s3":{
        "s3SchemaVersion":"1.0",
        "configurationId":"admin_conf",
        "bucket":{
          "name":"current2",
          "ownerIdentity":{
            "principalId":"AUTH_test"
          },
          "arn":"arn:aws:s3:::current2"
        },
        "object":{
          "key":"curr_my_object",
          "size":"16",
          "eTag":"
              a87ff679a2f3e71d9181a67b7542122c
              ",
          "versionId":"1649772288.14729",
          "sequencer":"1649772288.14729"
        }
}}]}
```


Listing 8. Example of published S3 notification

4.6 Use Cases and Scenarios

ENOSS has multiple use cases. Some of possible use case scenarios where ENOSS can be applied are:

Anomaly detection - since ENOSS is capable of publishing unsuccessful events, using filter **httpcodes** admin can set ENOSS to publish notifications about events involving internal errors (HTTP code 500) or any other nonstandard HTTP codes.

Data theft detection - user has a designated container for sensitive data. The container owner can "tell" ENOSS which users should have access to the container, and if some unknown users, that are not in the list of users specified by the container's owner, somehow gain access to the container, then ENOSS will publish a notification about such event.

Data theft prevention - user configures ENOSS to publish unauthorized events, which can result in the detection of the possible attempt of data theft.

Postprocessing - user wants to search stored data using their metadata. The user configures ENOSS to publish events that store, modify and delete data in object storage. The destination of published events

would be Elasticsearch or some other custom destination capable of full-text search.

5. Conclusions

This paper presents a solution for publishing notifications about events that occurred in OpenStack Swift.

ENOSS is fully compatible with AWS S3 Event Notifications, offers multiple destinations to which notifications can be published, allows users to specify, using filters, which event notifications should be published. Furthermore, users can choose different types of notification payload (from standard AWS S3 payload structure to custom-defined structure) and offers a way for effortless expansions of new types of destinations, notification payloads, and filters.

ENOSS can be used for monitoring events in OpenStack Swift, automatization and postprocessing, and serverless computing capable of reacting to events that occurred in OpenStack Swift(similarly to AWS Lambda).

In the future, new destinations(Elasticsearch, MySql, Redis, etc.) will be added. A further plan is the support of various new filters (filtering using time when an event occurred, stored metadata, etc.). Last but not least, support of different notification standards, such as CloudEvents.

Acknowledgements

I would like to thank my supervisor RNDr. Marek Rychlý Ph.D. for his valuable advice and support during the creation of this work.

References

[1] Joe Arnold. *OpenStack Swift: Using, Administering, and Developing for Swift Object Storage*. O'Reilly Media, Inc., 2014.

[2] Swift architectural overview. online. https://docs.openstack.org/swift/xena/overview_architecture.html.

[3] Swift middleware and metadata. online. https://docs.openstack.org/swift/xena/development_middleware.html.

[4] Send notifications on put/post/delete requests. online. https://specs.openstack.org/openstack/swift-specs/specs/in_progress/notifications.html.

[5] Openstack swift - event notification first attempt. online. https://review.opendev.org/c/openstack/swift/+/196755.

[6] Openstack swift - event notification second attempt. online. https://review.opendev.org/c/openstack/swift/+/388393.

[7] Event message structure - amazon s3. online. https://docs.aws.amazon.com/AmazonS3/latest/userguide/notification-content-structure.html.