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 2021

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
Field of study: 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 as (filter) part of asynchronous service Event-Agent 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, substring) 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 jako (filtr) součást asynchronní služby Event-Agent uvnitř 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, 2021. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor RNDr. Marek Rychlý, Ph.D.

Monitoring the OpenStack Swift Object Store Using Beanstalk Events

Declaration

I hereby declare that this Bachelor'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
January 17, 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

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 objects 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[18][20] 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 notification 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 storages to be even more competitive against commercial storages (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 objects 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 technologies and general areas that this work relates to. Chapter 3 covers OpenIO SDS, describes its data organization and key services providing events. Chapter 4 introduces OpenStack object storage Swift, its

data model, server processes and describes middlewares within OpenStack Swift. Chapter 5 briefly covers MinIO storage and the way it publishes event notifications. Chapter 6 describes the current state of event notifications in OpenIO SDS, OpenStack Swift and MinIO, and proposes a solution for publishing event notifications in OpenIO SDS and OpenStack, and a solution for publishing event notifications from MinIO to Beanstalk queue.

Chapters 3, 4, and 5 deal with the first point of the assingment, while chapter 6 deals with the second point of assigment.

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 types of events can occur inside. The last part of this chapter describes the concept of event notifications, why they are essential, and 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 hierarchical methods used in file systems[36]. 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[37].

Key concepts

Key concepts of object storage are[29]:

- **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, 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[29]:

- **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[36]. The second type is *Custom metadata*, to which users can store any 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 [39] 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)[37].
- 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[28].
- 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[35]. 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[24].

Principles

There is no clear definition on criteria for defining software-defined storage, although several fundamental principles can be deducted[31]:

- **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[34].

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[34].

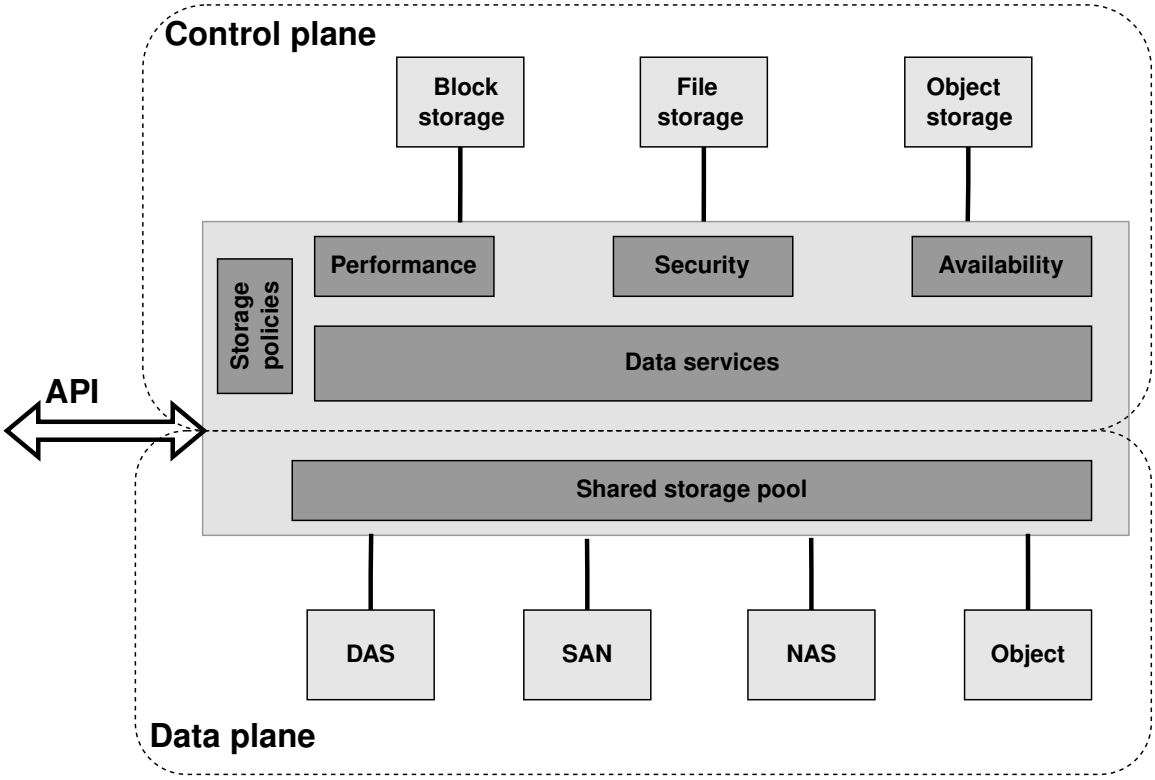

Figure 2.1: SDS data and control plane (source: [21], remade).

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[17].

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.

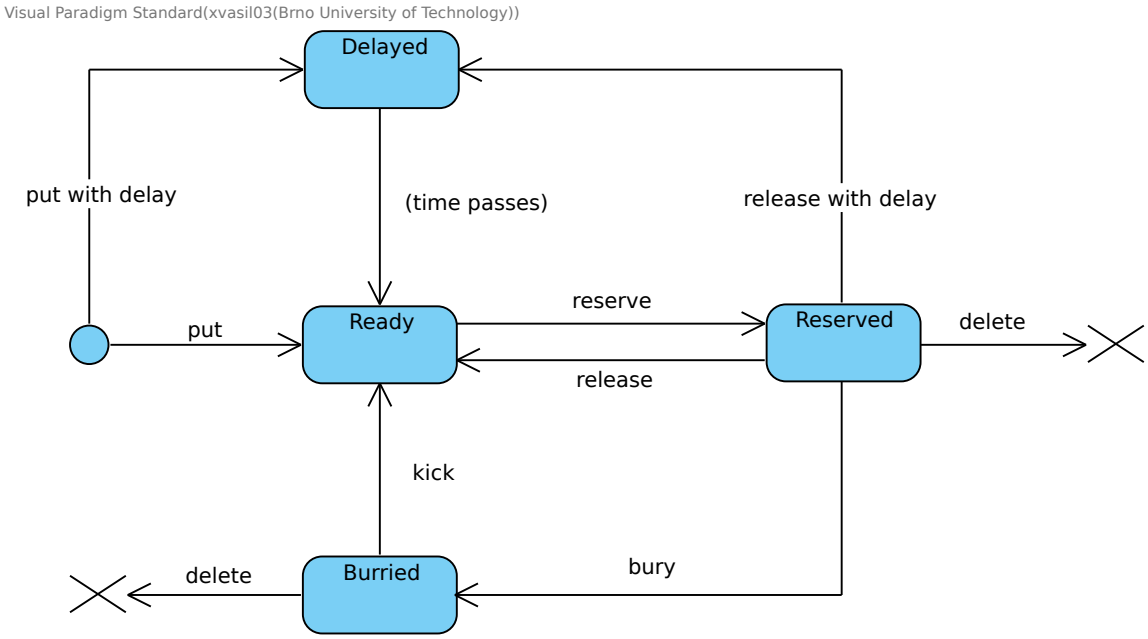

Figure 2.2: State machine diagram of job in Beanstalkd tube.

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[38].

Event notification is a message created by a system in order to notify other parts of the system that an event has taken place[25]. 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

OpenIO SDS

This chapter introduces OpenIO Software-defined storage, its key features, its data organization along with the underlying technologies. Furthermore, this chapter introduces Grid For Apps framework (3.3.1) and event publishing in OpenIO (3.3.2).

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 (3.3.1), 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[23]. 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 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[10]. 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[19].

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[10].

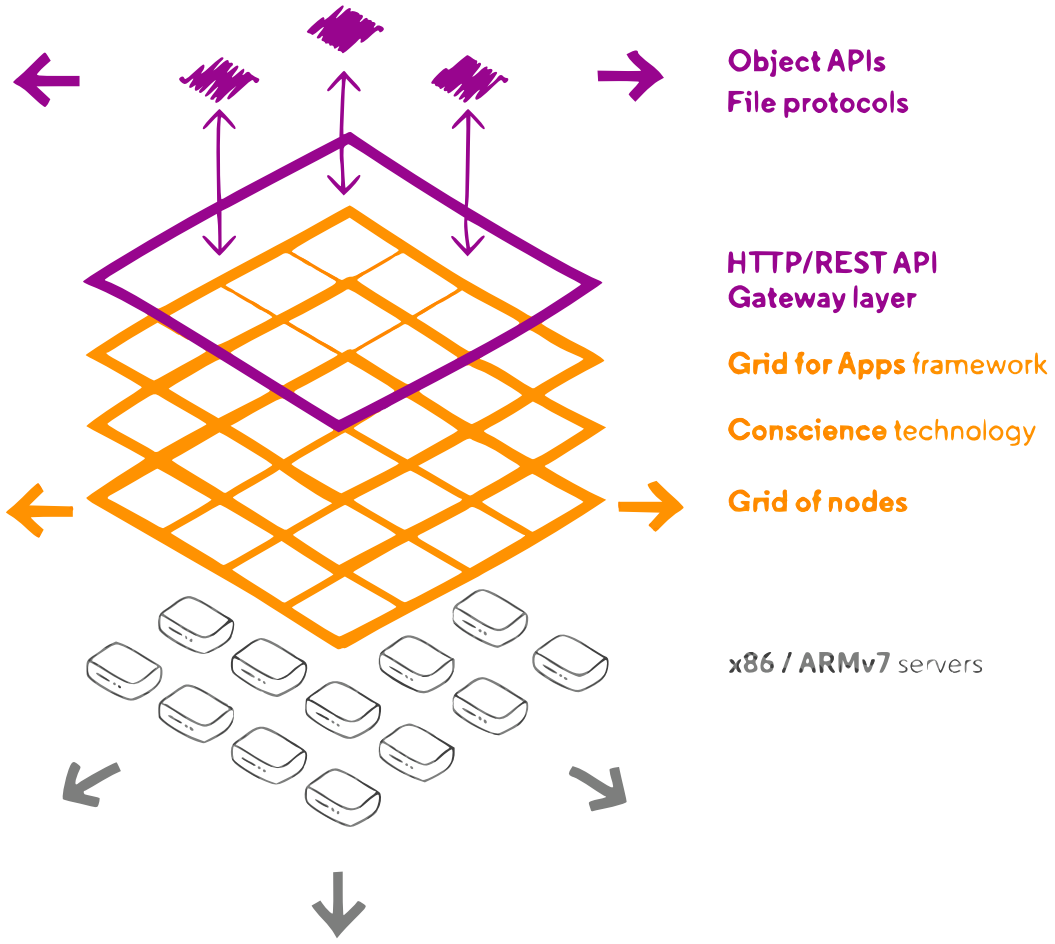

Figure 3.1: Layered view on OpenIO SDS architecture (source: [16]).

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[19].

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[12].

3.2 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[11]. 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.

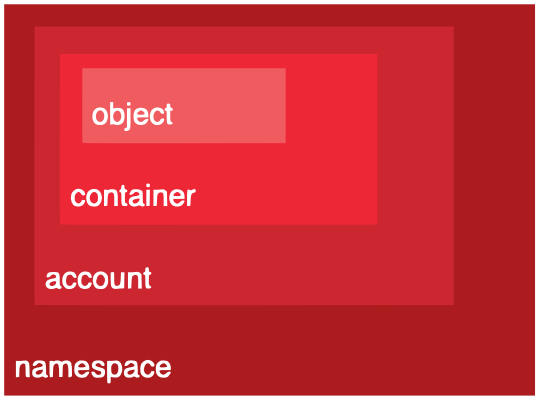

Figure 3.2: Object data organization in OpenIO SDS (source: [19]).

3.2.1 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.

3.2.2 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)[19].

3.2.3 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)[19].

3.2.4 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[19].

3.3 Serverless computing

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

3.3.1 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[23]. 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)[23].

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[23].

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

3.3.2 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[12]:

- 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 [7]. 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

Chapter 4

OpenStack Swift

This chapter introduces OpenStack object storage (code name Swift) and describes its key features. Furthermore, this chapter 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[30].

4.1 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[30].

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[30]:

- **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[28]. 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[14].

Partial object retrival Users can retrieve part of an object, for example, just a portion of a movie object file[27].

4.2 Data model

OpenStack Swift allows users to store unstructured data objects with a canonical name containing *account*, *container* and *object* in given order[30]. 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.

4.2.1 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**[30].

4.2.2 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[30].

4.2.3 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[30].

4.3 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**.

4.3.1 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[13]. 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[30].

4.3.2 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[13].

4.3.3 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[13].

4.3.4 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[13].

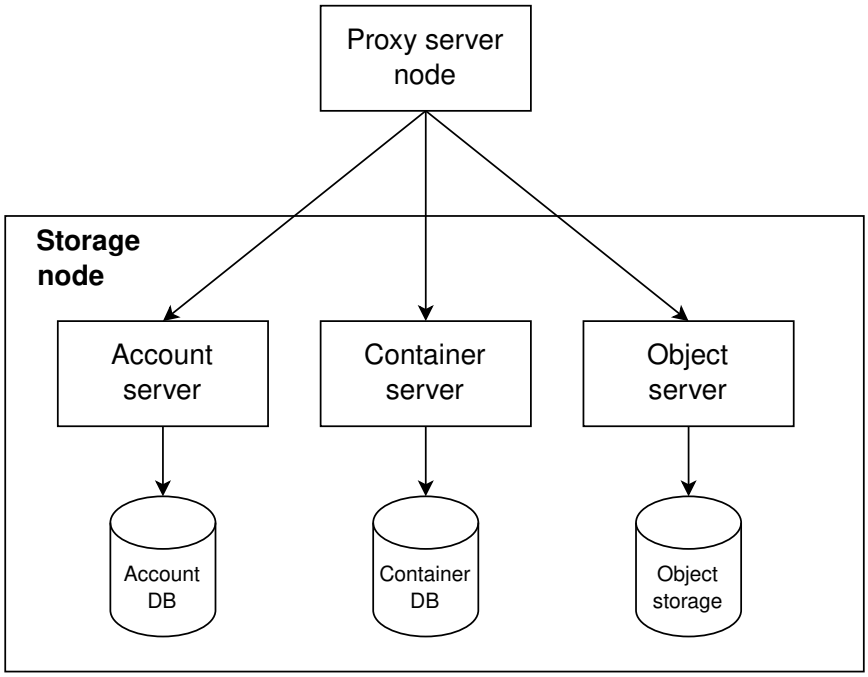

Figure 4.1: OpenStack Swift servers architecture.

4.4 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 4.1 `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[15].

```
[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 4.1: Example of proxy server configuration (proxy-server.conf).

4.4.1 Interface

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

In 4.2 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.

```
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 4.2: Example of healthcheck middleware in OpenStack Swift

4.4.2 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[15].

Chapter 5

MinIO

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

5.1 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[9].

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.

5.2 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[22].

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[26].

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[22].

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[22].

5.3 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[9].

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[22].

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[22].

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[26].

Figure 5.1 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[9].

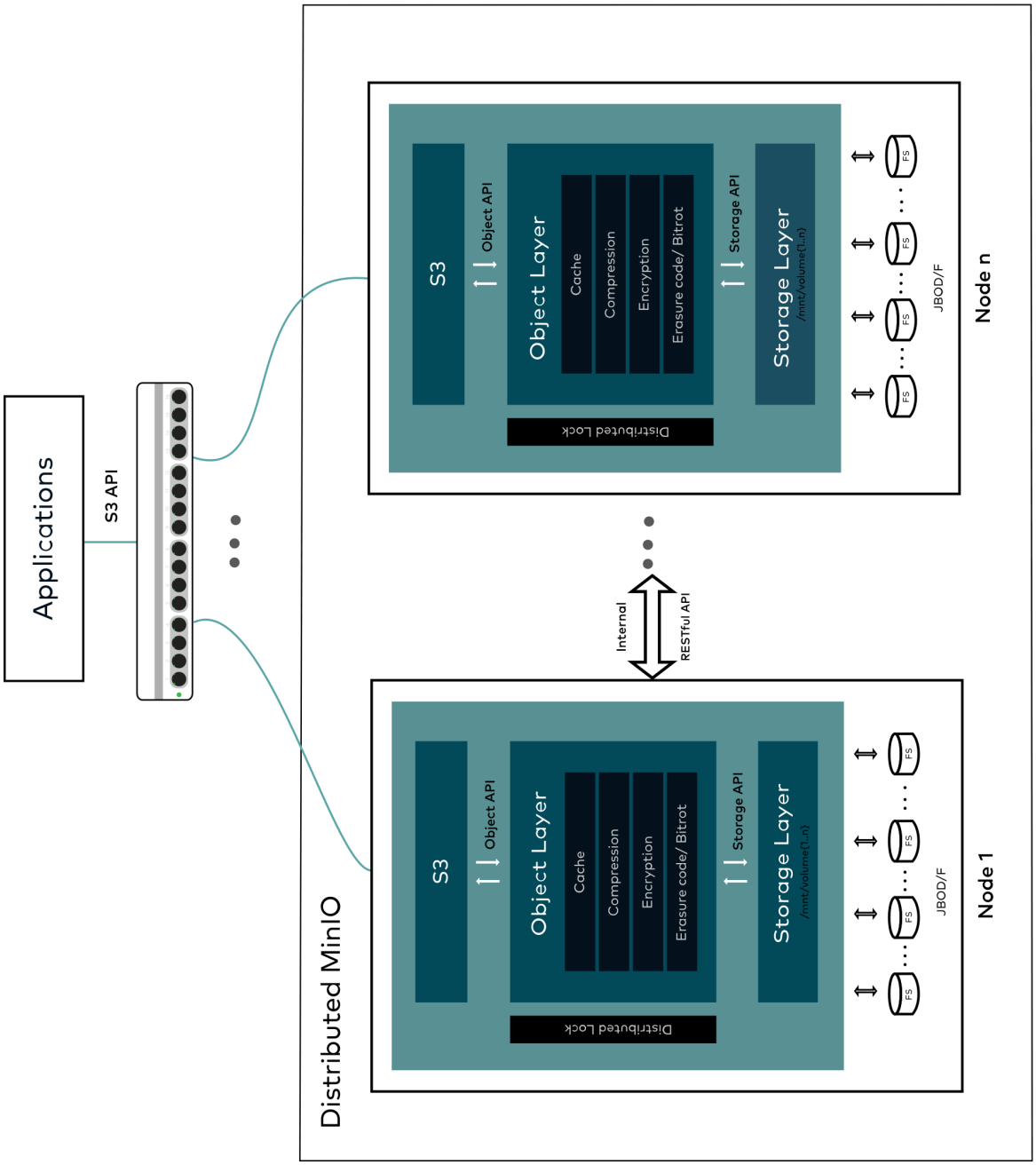

Figure 5.1: Overview of MinIO cluster architecture (source: [9], modified).

5.4 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 [8].

Supported event notification targets are AMQP, Redis, MySQL, LMQTT, NATS, Apache Kafka, Elasticsearch, PostgreSQL, Webhooks, and NSQ. Figure 5.2 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`.

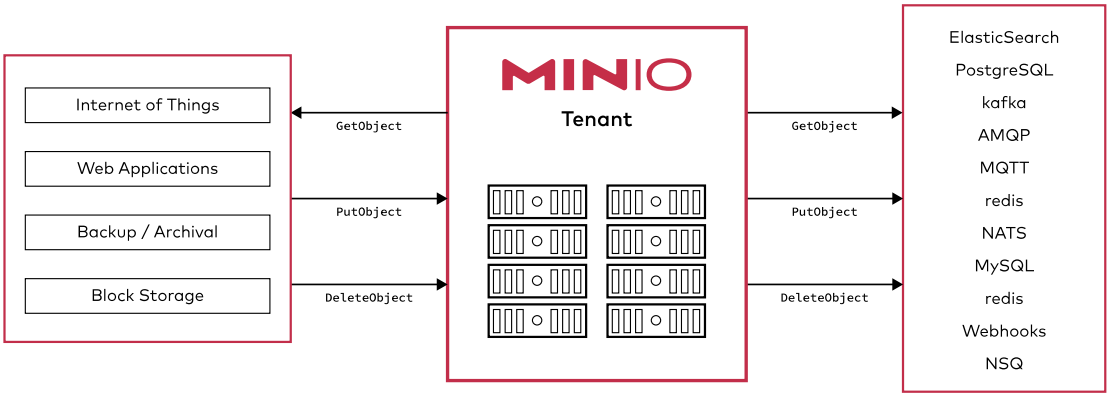

Figure 5.2: Overview of event notification in MinIO storage (source: [8]).

Chapter 6

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.

6.1 Current state

6.1.1 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).

6.1.2 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[18][20] 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

6.1.3 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.

6.2 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 `EventNotification`.

6.2.1 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.

6.2.2 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, EventNotification middleware will publish events regarding containers/buckets. Furthermore, EventNotification 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.

EventNotification 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 EventNotification middleware in container level is shown in figure 6.1. 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, EventNotification needs to read account metadata from storage. After reading and parsing account metadata, `EventNotification` 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 6.2 shows simplified class diagram of EventNotification middleware. EventNotification 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 `EventNotification` is the core of middleware. Since access to metadata of upper level (container, account) does not have same interface for OpenStack Swift and OpenIO SDS, subclasses `Swift EventNotification` and `OpenIO EventNotification` were introduces to solve this problem.

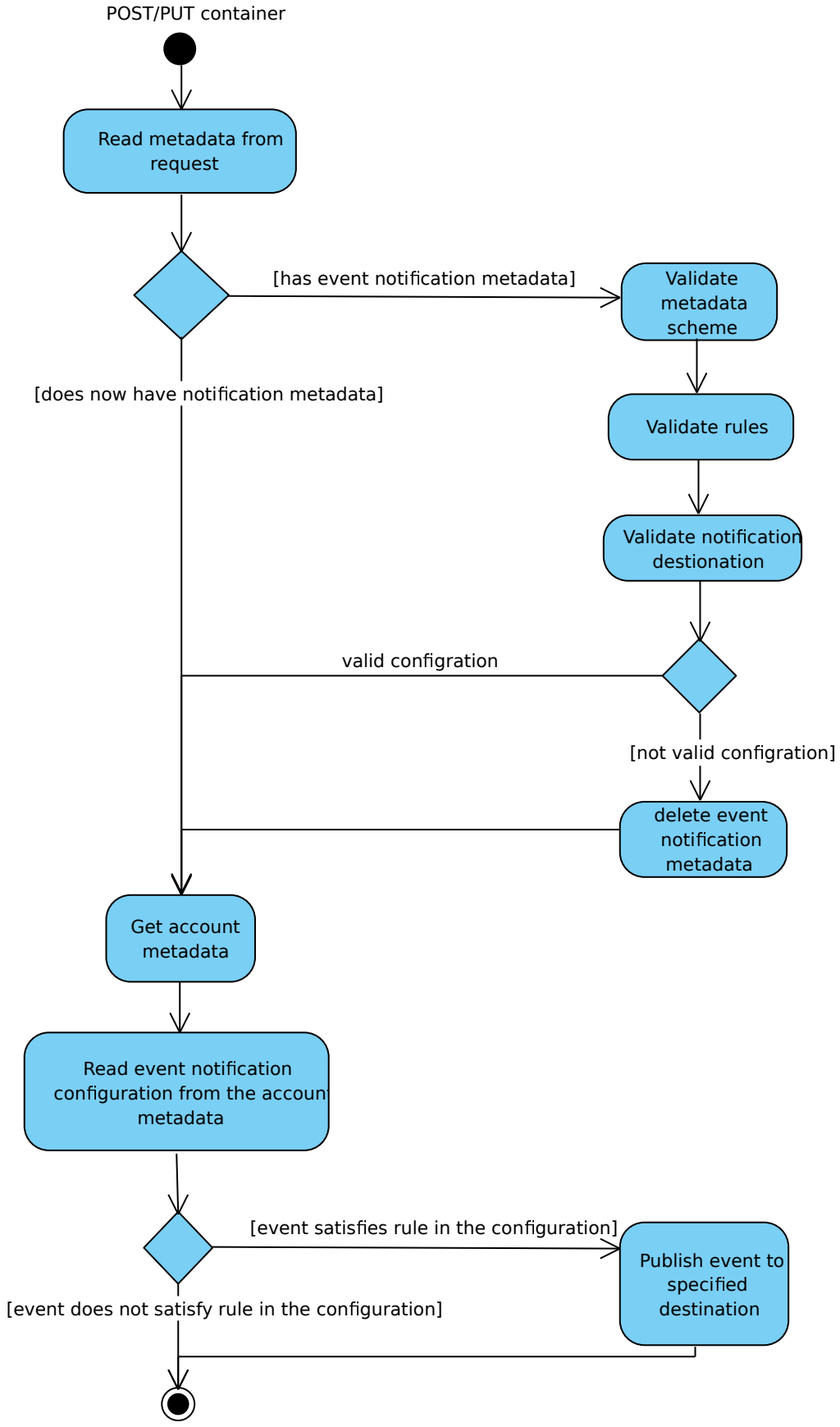

Figure 6.1: Activity diagram of EventNotification middleware.

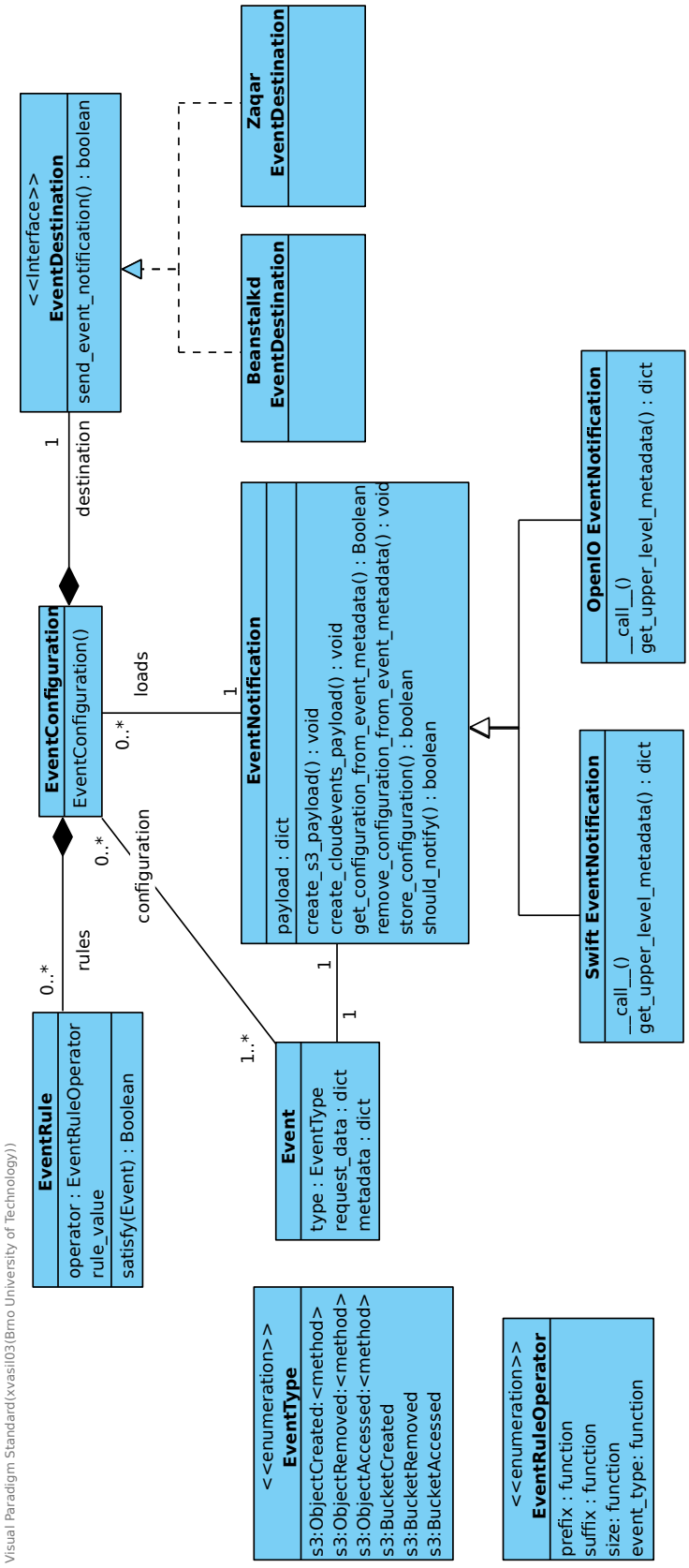

Figure 6.2: Class diagram of EventNotification middleware.

6.2.3 Structure of published event

EventNotification can publish event notification in Amazon S3 like structure and in structure following CloudEvents standard. Listing 6.1 and 6.2 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 6.1: CloudEvents structure of event notification published by EventNotification 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 6.2: Amazon S3 structure of event notification published by EventNotification middleware.

6.2.4 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 6.3 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.

Even 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",
      "StructureType": "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 6.3: Strucute of event notification configuration

6.3 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.

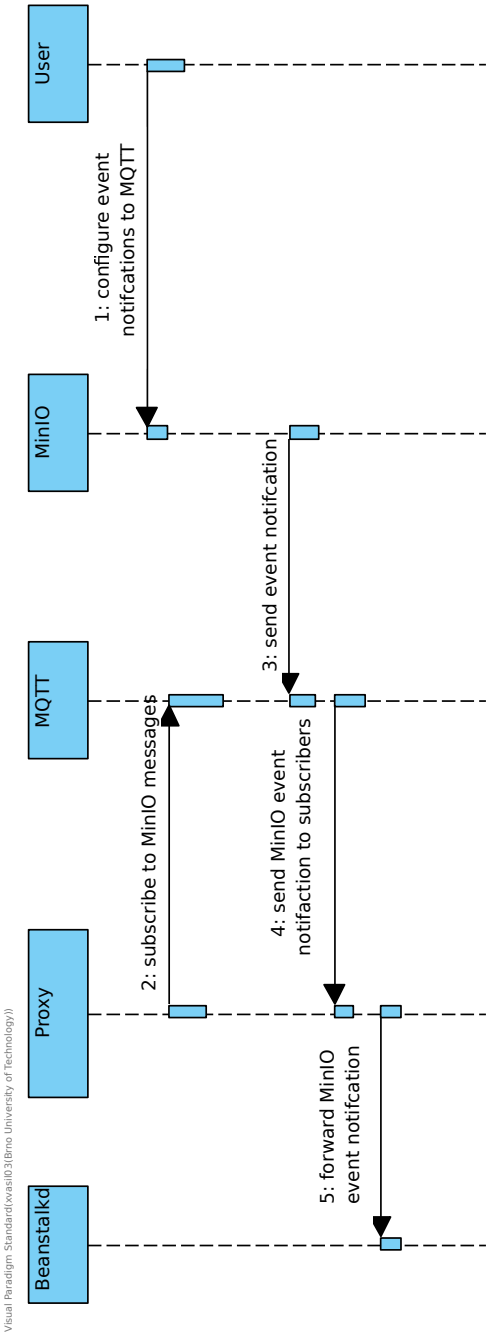

Figure 6.3: Sequence diagram of proxy application allowing publishing events from MinIO to Beanstalkd.

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

Figure 6.3 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 7

Implementation, experiments and assessment

Chapter 8

Conclusion

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

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

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

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

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

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

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

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

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

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

[17] *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.

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

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

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

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

[22] *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.

[23] *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.

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

[25] *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.

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

[27] Amar Kapadia, K. R. *Implementing Cloud Storage with OpenStack Swift*. Packt Publishing, 2014. ISBN 9781782168058.

[28] Amar Kapadia, S. V. *OpenStack Object Storage (Swift) Essentials*. Packt Publishing, 2015. ISBN 978-1-78528-359-8.

[29] Anil Patil, H. S. M. L. R. M. d. S. . *Cloud Object Storage as a Service: IBM Cloud Object Storage from Theory to Practice*. IBM, 2017. ISBN 0738442453.

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

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

[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] Larry Coyne, E. F. P. G. R. H. C. D. M. A. M. T. P. B. S. C. V. *IBM Software-Defined Storage Guide*. IBM, 2018. ISBN 0738457051. Available at: https://www.redbooks.ibm.com/redpapers/pdfs/redp5121.pdf.

[35] Macedo, R., Paulo, J. a., Pereira, J. and Bessani, A. 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.

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

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

[38] Pethuru Raj, H. S. *Architectural Patterns*. Packt Publishing, 2017. ISBN 9781787287495.

[39] Zheng, Q., Chen, H., Wang, Y., Duan, J. and Huang, Z. *COSBench: A Benchmark Tool for Cloud Object Storage Services*. 2012. 998-999 p. ISBN 978-1-4673-2892-0.