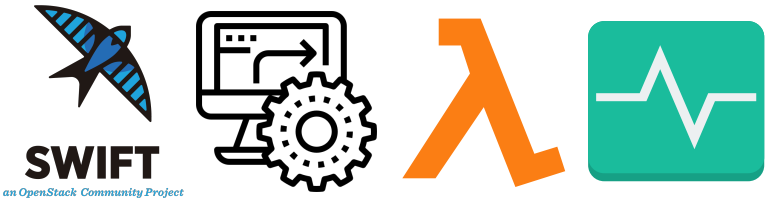
BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

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 **??** introduces the motivation, defines problems and desired objectives. Chapter **??** describes object storage OpenStack Swift, its data model, main processes, and describes middlewares and metadata within OpenStack Swift. Chapter **??** analyzes and compares existing solution for given problem. Chapter **??** describes proposed solution - ENOSS, its key features, configuration and interfaces. Chapter **??** summarize proposed solution, highlights results of this work and its contributions. Chapter **??** 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[**?**].

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[**?**]. 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**[**?**].

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[**?**].

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[**?**].

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[**?**]. 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[**?**].

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[**?**].

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

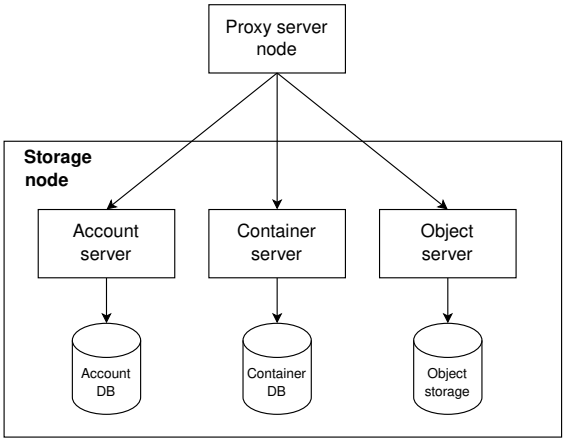


Figure 1. OpenStack Swift servers architecture.

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[**?**].

2.3 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 **??** 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[**?**].

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 **??** 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="
    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. Existing solutions

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

Webhook middleware described in **??** 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 [**?**]. 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** [**?**] - 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** [**?**] - 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.

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 **??**. 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,

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

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 **??** 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`

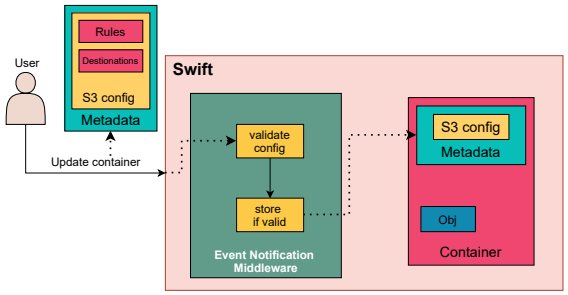


Figure 2. Process of setting event notification configuration in ENOSS.

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

Configuration structure - Listing **??** 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 $\star$ 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 **??**, 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 **??**, `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

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.