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 informations to users about events occured in storage they own/have access to. Users do not have information when content of their object storage is accessed, changed, created or deleted. The aim of this paper is to create solution that will send notifications about events occured in OpenStack Swift to user specified destinations. 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, etc. It also offers mutiple destinations(Beanstalkd queue, Kafka, etc.) to which notifications can be publsihed. 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 as well. Event notification can be used not just for monitoring, but also for automatization and serverless computing similarly to AWS Lambda.

Keywords: Event — Notifications — OpenStack Swift

Supplementary Material: Demonstration Video — Downloadable Code

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

1. Introduction

Object storage is 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 storages are often used in cloud computing, 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 in storages where their data are located. For example, currently there is no easy way to detect changes in specific container, except to list it's content and compare timestamps, which can be complex, slow and unefficient if there is a lot 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 of new objects in container, therefore it is important that proposed solution allows event filtering based on event type and other properties (e.g. object name prefix/suffix/size). Since object storage have multiple users, each user can have different requirements for event notification and proposed solution must be prepared for it.

Application of event notifications is various, from simple monotoring or webhook, to more sophisticated application as serverless computing like AWS Lambda. This means that structure of event notification may differ based on its application and destination to which is published. Proposed solution must be ready to pubslih event notification to different destinations as well as in different event notfication structure.

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

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, 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.

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

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[**?**].

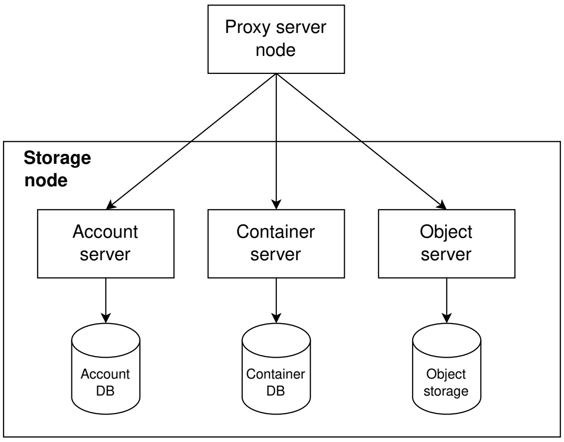

Figure 1. OpenStack Swift servers architecture.

2.3 Middleware

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

Middlewares are added by changing the configuration of servers. Listing 1 is shows how to add *webhook middleware* to proxy server by chaning 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 2 provides example of simplified *healthcheck middleware*. The constructor takes two arguments, the first is a WSGI application, and the second is a configuration of middleware defined using Python Paste framework in *proxy-server.conf*. Middleware must have a call method containing the request environment information and response from previously called middleware. Middleware can perform some operations and call the next middleware in the pipeline or intercept a request. In the 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 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 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[**?**].

3. Existing solutions

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

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

OpenStack Swift attempts - OpenStack Swift is aware of 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 specific container (creation, modifying and deletiton of objects). There were two attempts to solve this problem.

- **First attempt??** - allowed sending notifications only to Zahar[todo] queue 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 notfication can be published. Event notification structure was expanded for informations such as eTag (MD5 checksum) and transaction id. Author introduced concept of ”notification policy”, which represented configuration of event notifications. One of main critiques made by code reviewers was incompatibility with AWS S3 storage.

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

ENOSS - my solution, code name ENOSS, satisfies all requirements specified in section 1. Key features are: events filtering, support of multiple destinations, AWS S3 compatibility, different event notification structure, 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 program that enables pubslishing notifications containing information about occured events in OpenStack Swift. It is implemented in form of Python WSGI middleware and is located in Proxy server pipeline. Since Proxy server communicates with external users, by placing ENOSS in its pipeline ENOSS is able to react on every request made by users to OpenStack Swift, which makes Proxy server ideal places 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 a container metadata and for container events, the configuration is stored in a account level.

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

Event filtering - one of the main requirements for event notifications is allowing users to specify for which events should notifications be publsihed - i.e. event filtering. ENOSS allows user to specify which types of events should be published (object/continaer creation, deletion, access, etc.). ENOSS goes little bit futher and also allows users to specify rules that must be satisfied in order for event notification to be pubslished. Some of the rule operators are: object/container name prefix/suffix and object size. Using this feature user can select for example, only events regarding objects bigger then 50mb (operator: object size) or events regarding pictures (operator: object suffix).

Multiple destinations - since event notifications has multiple applications, from monitoring to automatization, it is important that proposed solution is able to publish notification to multiple diferent destinations. ENOSS is fully capable of publishing event notifications to many different destinations (e.g. Beanstalkd quque, Kafka). In ENOSS, publishing notification about single event is not limited to only one destina-

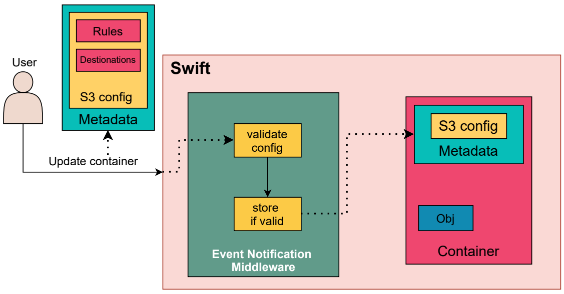

Figure 2. Process of setting event notification configuration in ENOSS.

tion, but, if user wishes, it can be publihed to multiple different destination per single event. This feature allows event notification to be used for mutliple applications simulteniosly.

Event notification structure - depending on application of event notification structure of notification may differ. ENOSS supports several different notification structures and using event notification configuration users are able to select type of event notification structure.

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

4.2 Configuration

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

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

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

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

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

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

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

Event type takes form :

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

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

Listing 3. Strucute of event notification configuration

```
{
  "<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"
            }
            ...
          ]
        }
      }
    }
    ...
  ]
}
```

4.3 Interfaces

Destination -

Payload -

Filter Rule -

Supported event types

4.4 Performance

5. Conclusions

[Paper Summary] What was the paper about, then? What the reader needs to remember about it? Lorem ipsum dolor sit amet, consectetur adipiscing elit. Proin vitae aliquet metus. Sed pharetra vehicula sem ut varius. Aliquam molestie nulla et mauris suscipit, ut commodo nunc mollis.

[Highlights of Results] Exact numbers. Remind the reader that the paper matters. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed tempus fermentum ipsum at venenatis. Curabitur ultricies, mauris eu ullamcorper mattis, ligula purus dapibus mi, vel dapibus odio nulla et ex. Sed viverra cursus mattis. Suspendisse ornare semper condimentum. Interdum et malesuada fames ac ante ipsum.

[Paper Contributions] What is the original contribution of this work? Two or three thoughts that one should definitely take home. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent posuere mattis ante at imperdiet. Cras id tincidunt purus. Aliquam erat volutpat. Morbi non gravida nisi, non iaculis tortor. Quisque at fringilla neque.

[Future Work] How can other researchers / developers make use of the results of this work? Do you have further plans with this work? Or anybody else? Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse sollicitudin posuere massa, non convallis purus ultricies sit amet. Duis at nisl tincidunt, maximus risus a, aliquet massa. Vestibulum libero odio, condimentum ut ex non, eleifend.

Acknowledgements

I would like to thank my supervisor X. Y. for her help.