DAOS Design Document

Goal

The goal of this project is to create an S3 layer to access DAOS backend. DAOS is an object storage system built by Intel that DAOS is uniquely optimized for high-performance devices (e.g. Optane). It is highly complementary to CORTX in datacenters, so Seagate and Intel share a desire to create a unified solution combining DAOS and CORTX under the same S3 layer.

Implementation

- You can find the implementation code at https://github.com/zalsader/ceph/tree/add-daos-rgw-sal.
- A PR was opened into ceph: https://github.com/ceph/ceph/pull/47709
- Libds3 PR into DAOS: https://github.com/daos-stack/daos/pull/8889
    - Libds3 API is found at https://github.com/daos-stack/daos/blob/libds3/src/include/daos_s3.h
- A script to compile and configure the code could be found here: Build a single host development environment for CORTX/DAOS
- The detailed steps to compile and configure the code could be found here: DAOS + RGW (DGW) HOWTO
- Open tasks in the project can be found here: https://github.com/users/zalsader/projects/3
- S3 Integration Status

Contents

- Versioning
- Object Index
- Caching
- Notifications
- Object Lock
- Existing Pools
- Design Challenges

Video Demos

More demos are found here: Demos

20 Apr 2022

Second demo: https://youtu.be/6EDCQmAbxW4

The demo showed more S3 operations: get object, delete object, and demonstrated S3 object versioning, and multipart upload

31 Jan 2022

A demo of the initial proof of concept implementation can be found here: https://youtu.be/cL_e-ld5WIU

The demo showed how to set up the S3 – DAOS system, in addition to some basic S3 operations: list-buckets, make-bucket, put object. It also demonstrated how DGW interacts with DFuse by creating objects as files and that slashes (/) in the object path are showing up as directories.

Glossary

DAOS:

DAOS: The Distributed Asynchronous Object Storage (DAOS) is an open-source object store designed from the ground up for massively distributed Non Volatile Memory (NVM). DAOS takes advantage of next-generation NVM technology, like Intel© Optane™ Persistent Memory and NVM express (NVMe), while presenting a key-value storage interface on top of commodity hardware that provides features, such as transactional non-blocking I/O, advanced data protection with self-healing, end-to-end data integrity, fine-grained data control, and elastic storage, to optimize performance and cost.

Pool: Persistent versioned list of storage target memberships, including storage topology tree. Features include: self-healing / rebuild, space rebalancing. A pool offers storage virtualization and is the unit of provisioning and isolation. DAOS allows attaching custom attributes to a pool.

Container: A container represents an object address space inside a pool. It is equivalent to S3 buckets. Each container is a private object address space, which can be modified transactionally

and independently of the other containers stored in the same pool. A container is the basic unit of transaction and versioning. DAOS allows attaching custom attributes to a container.

DAOS Object: DAOS Objects have128-bit object address, similar to Motr. DAOS objects are intentionally simple. No default object metadata beyond the type and schema is provided. Many object schemas (replication/erasure code, static/dynamic striping, and others) are provided. Objects can be accessed through different APIs:

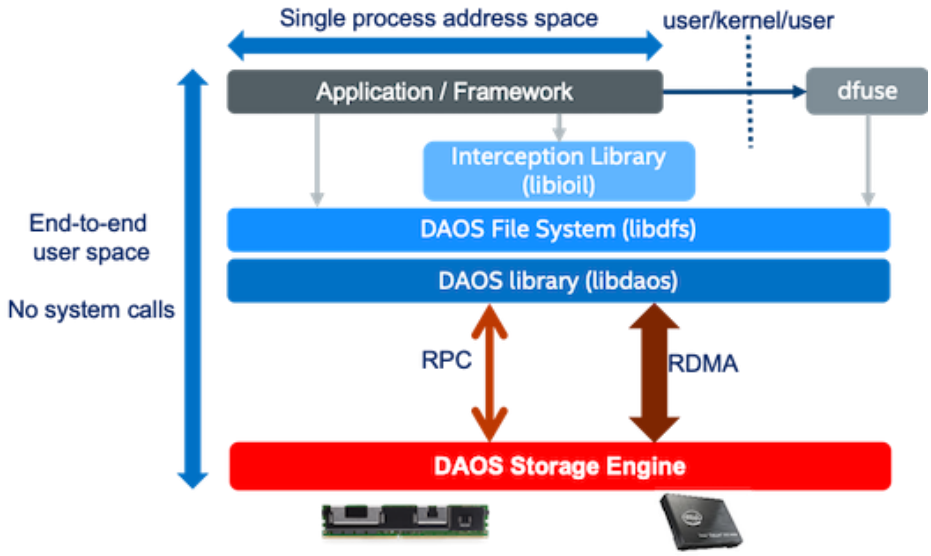
- **Multi-level key-array API** is the native object interface with locality feature. The key is split into a distribution (dkey) and an attribute (akey) key. Both can be of variable length and type.
    - Distribution Key (dkey): guaranteed to be collocated on the same target
    - Attribute Key (akey): The value associated with akey is a single variable-length value that cannot be partially overwritten or an array of fixed-length values.
- **Key-Value API** provides a simple key and variable-length value interface. It supports the traditional put, get, remove and list operations.
- **Array API** implements a one-dimensional array of fixed-size elements addressed by a 64-bit offset. A DAOS array supports arbitrary extent read, write and punch operations.

DAOS FS (DFS): allows mounting a container as a shared POSIX namespace on multiple compute nodes. This capability is provided by the **libdfs** library that implements the file and directory abstractions over the native `libdaos` library. The POSIX emulation can be exposed directly to applications or I/O frameworks (as is currently used in DAOS-RGW). It can also be exposed transparently via a FUSE daemon. No OS overhead is incurred if `libdfs` is used directly. It supports files, directories, and symbolic links.

LIBDS3: is a library built on top of `libdfs` implementing common S3 APIs over DAOS and DFS. It is part of the DAOS codebase to make it easier to include changes later on.

DFS Objects/files have **extended attributes**, a set of key-value attributes attached to an object. Internally, DFS stores directories as DAOS objects using the multi-level key-array API, using dkey for the file / subdirectory name, and the akeys for file attributes and extended attributes. Files are stored as DAOS objects using the Array API.

DFuse (DAOS FUSE): DFuse provides DAOS File System access through the standard libc/kernel/VFS POSIX infrastructure. DFuse builds heavily on DFS. Data written via DFuse can be accessed by DFS and vice versa.



Motr:

Motr Object: A client object is an array of blocks, which can be read from and written onto at the block granularity. Each object has unique fid, a 128-bit address.

Motr Index: A client index is a key-value store.

Container: a "place" where a particular storage application lives.

RGW:

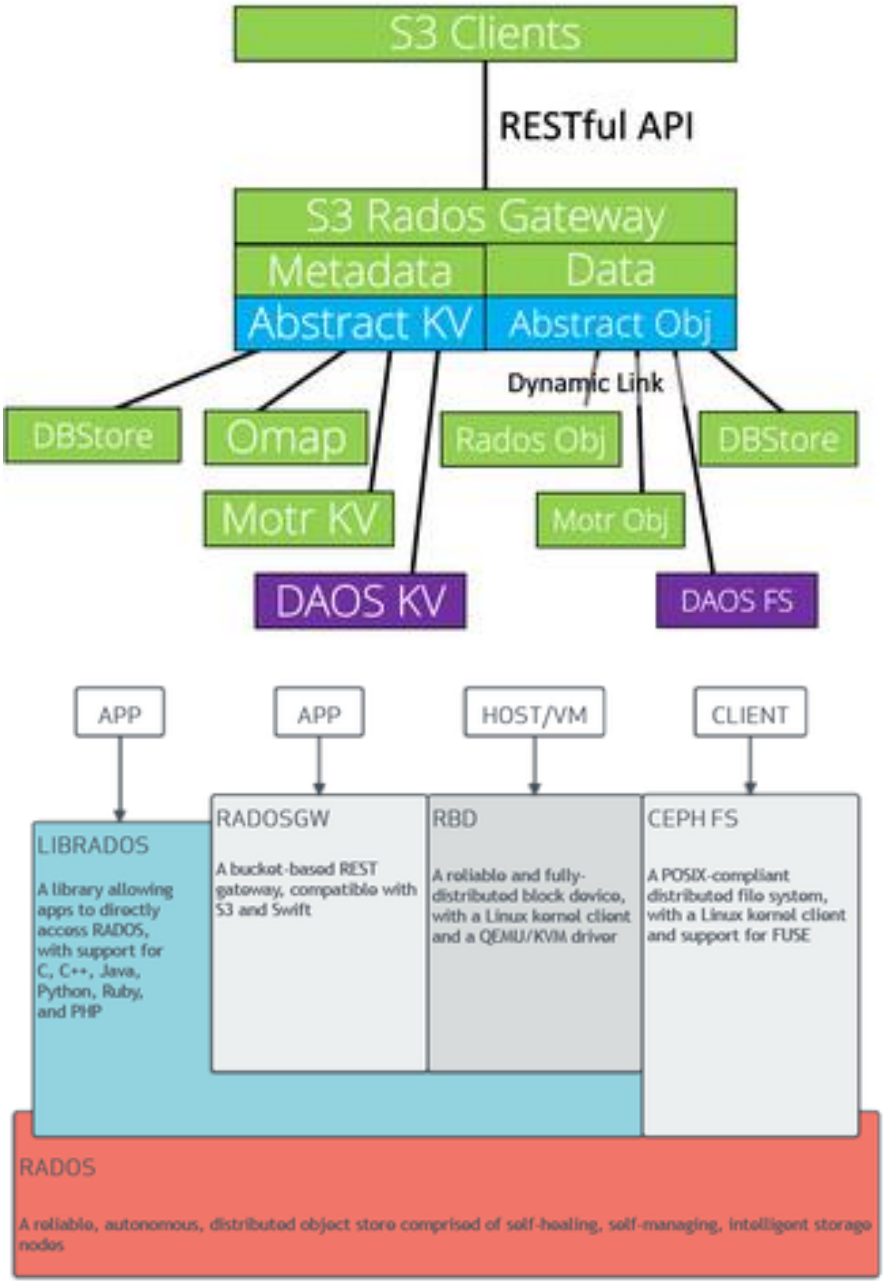
Bucket: A bucket is a mechanism for storing data objects. The term is used interchangeably with "**container**". Container names must be unique.

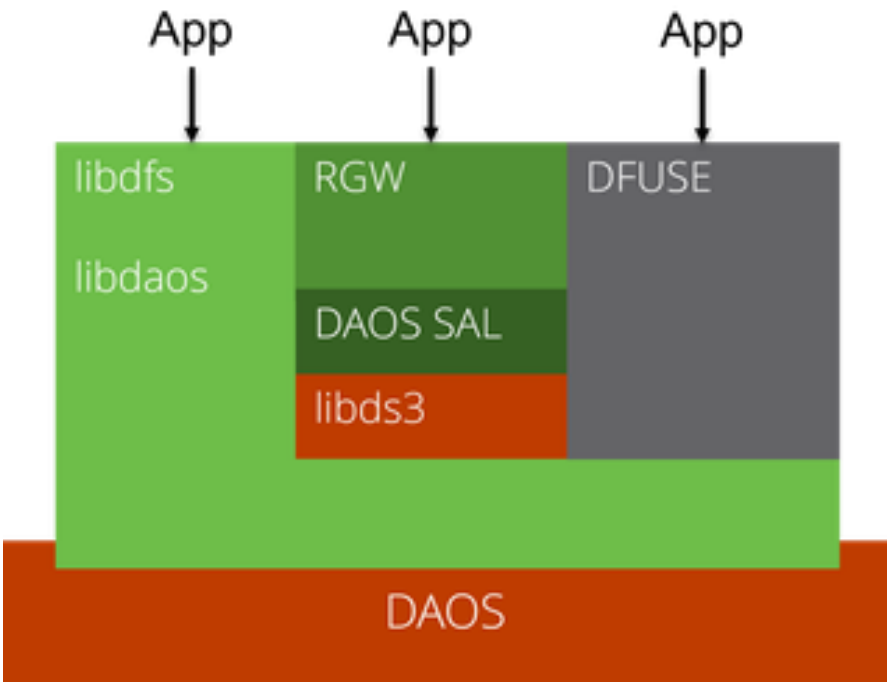
Bucket Index: The bucket index is an internal structure which holds a key-value map. Each key in the map is the name of an object, and the value holds some basic **metadata of that object** – metadata that shows up when listing the bucket.

RGW Object: Consists of metadata (such as manifest, ACLs, content type, ETag, and user-defined metadata) and the object value itself.

User: A user is either an individual or a system actor such as an application. Creating users allows you to control who (or what) can access your Ceph Storage Cluster, its pools, and the data within pools.

Design


S3 Clients
RESTful API
S3 Rados Gateway
Metadata
Data
Abstract KV
Abstract Obj
Dynamic Link
DBStore
Omap
Rados Obj
DBStore
Motr KV
Motr Obj
DAOS KV
DAOS FS
APP
APP
HOST/VM
CLIENT
LIBRADOS
A library allowing apps to directly access RADOS, with support for C, C++, Java, Python, Ruby, and PHP
RADOSGW
A bucket-based REST gateway, compatible with S3 and Swift
RBD
A reliable and fully-distributed block device, with a Linux kernel client and a QEMU/KVM driver
CEPH FS
A POSIX-compliant distributed file system, with a Linux kernel client and support for FUSE
RADOS
A reliable, autonomous, distributed object store comprised of self-healing, self-managing, intelligent storage nodes



Overview

CORTX switched from the Seagate-developed implementation of S3 to Ceph's Rados Gateway. This page offers a design that connects DAOS API to S3 via the same Rados Gateway, and closely resembles the Motr design for RadosGW. More details about the Motr design could be found in /wiki/spaces/PRIVATECOR/pages/771195409 and /wiki/spaces/PRIVATECOR/pages/776568839.

Ceph's RGW has a SAL (Storage Abstraction Layer) interface, which DAOS should implement, similar to Motr's WIP implementation, RADOS implementation and DBStore experimental implementation. SAL offers abstract interfaces to access buckets, objects and other S3 features without having to deal with the S3 formatting. This implementation offers a translation of these features into DAOS.

A new DAOS library: `libds3`

As part of this implementation, a new DAOS library was created called `libds3` to contain all the low-level DAOS code necessary for each operation. Because this library is in the DAOS repository, the implementation of each method in `libds3` can be altered without having to wait for the ceph process to merge code changes. This also allows some separation of concerns and the ability to add caching later. `libds3` internally uses the `libdfs` API. The new library API could be found at https://github.com/daos-stack/daos/blob/libds3/src/include/daos_s3.h

Why `libdfs`

An additional goal of the design is to allow S3 access to DAOS backend while maintaining as much layout interoperability with the POSIX API as possible. In other words, the POSIX API will be a lingua franca, where users are able to write a POSIX file and then read it as an object via S3 (and vice versa). In this design, we are translating slashes `/` in the object name to directories in the POSIX interface. Since the POSIX interface internally uses `libdfs` for the layout, it makes sense to also use it for the S3 layer and `libds3`. Additional benefits of this are that the `libdfs` API offers a straight-forward API to access and create files which should make development easier than using the lower-level API. Note that the lower level API is still usable, but one must be careful not to corrupt the layout to maintain interoperability. Additionally, compared to using the POSIX API directly in the implementation, `libdfs` is still in user-space and does not involve the Linux kernel.

Setup

Before launching DAOS + RGW (DGW), `daos_server` and `daos_agent` need to be running, and a pool needs to be created. The name of the pool is the only configuration needed. More details can be found here DAOS + RGW (DGW) HOWTO.

When DGW starts, in the `initialize` method, it initializes DS3 (`ds3_init`), which in turn initializes DAOS and DFS (`dfs_init`). Then it looks up the pool using the configured pool name (via `ds3_connect`) and stores the open DS3 handle in the `DaosStore` class to be used in later operations.

Buckets

S3 Buckets are mapped to DAOS containers. The DAOS containers need to be marked as POSIX compliant to maintain interoperability with DFS. This is done by setting the layout type property in the DAOS container to (`DAOS_PROP_CO_LAYOUT_POSIX`). The method `ds3_bucket_create` calls `dfs_cont_create_with_label`, which creates a DAOS container and initializes it to support DFS. Bucket information (metadata, number of objects, size, etc.) are then added as DAOS container attributes (using `daos_cont_set_attr`). The data is encoded by DGW and stored in one value (`"rgw_info"`). The bucket info could later be accessed by `ds3_bucket_get_info`, or changed using `ds3_bucket_set_info`

Listing buckets within ds3 (`ds3_bucket_list`) uses `daos_pool_list_cont` to list DAOS containers, returning only the containers that have labels, are DFS-compatible, and have the container info stored.

Objects

RGW objects map directly to DFS files, so we use `dfs_open` or `dfs_lookup` API calls inside `ds3_obj_open` and `ds3_obj_create`. The RGW object metadata is stored inside the extended attributes of the files, using `ds3_obj_[set|get]_info` , which in turn use `dfs_setxattr` and `dfs_getxattr`. Object attributed can be later changed using the same method calls. Slashes `/` in keys are translated into directories (created using `dfs_mkdir`) to make it compatible with POSIX API. Other delimiters are not supported at this time. Since this implementation uses the object's

extended attributes to store metadata, maintaining a separate bucket index should not be needed. This decision might be revisited in the future.

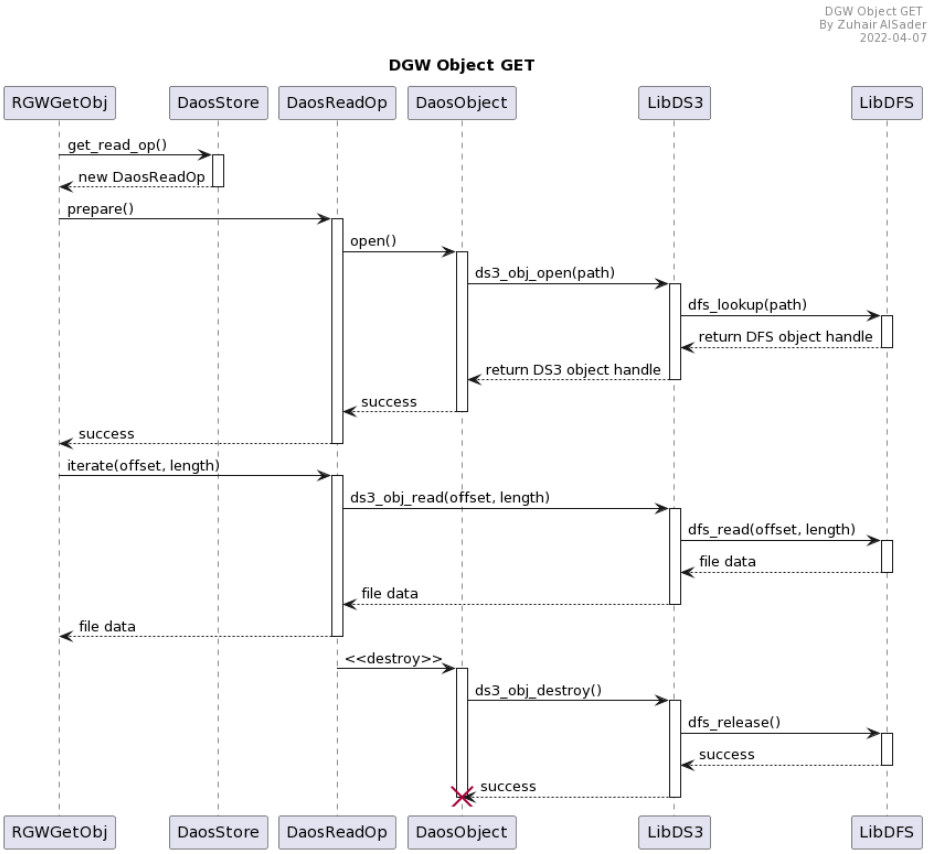
Basic Object Operations

When accessing objects, REST APIs provide DGW with three parameters: account information (access key in S3), bucket or container name, and object name (or key). After authentication, DGW looks up the user id from the given access keys and checks that the user can access the bucket. Then, it opens the bucket by its label using `ds3_bucket_open` and stores the open ds3 bucket handle in the bucket entity `DaosBucket`. Internally, `ds3_bucket_open` calls `dfs_connect` which returns a mounted dfs handle from a local cache.

When a `DaosBucket` instance is deconstructed, the ds3 bucket handle is released by `ds3_bucket_close`.

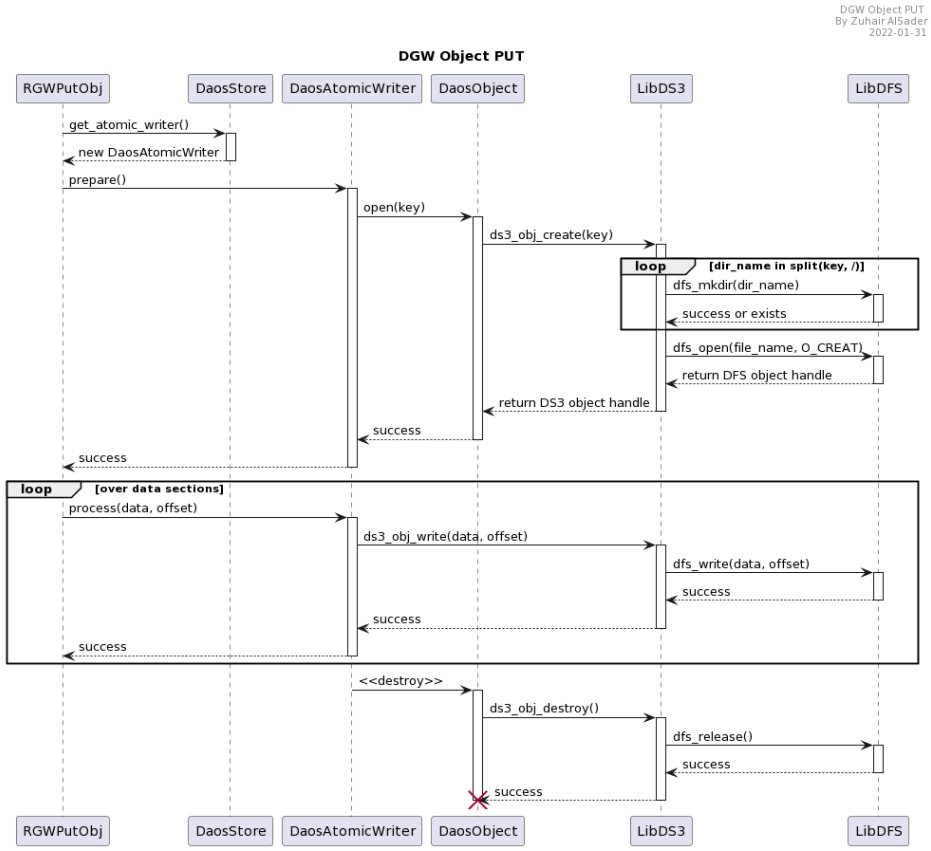
`GET` Operation

In a `GET` operation, the object is opened using `ds3_obj_open`, where the key is looked up within the directory structure using `dfs_lookup`, which internally traverses each directory in the key (split by `/`) until it finds the file, and an open DS3 object handle is returned and stored in the `DaosObject` entity. The handle is then used to read the object data using `ds3_obj_read` (via `dfs_read`). When the `DaosObject` is deconstructed, the DS3 object handle is closed using `ds3_obj_close`.



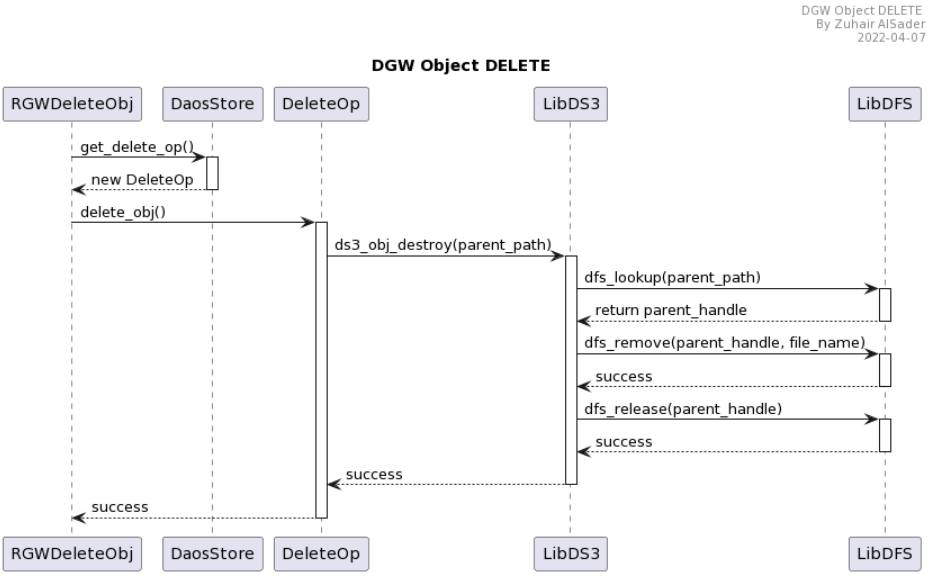
`PUT` Operation

In a `PUT` operation, `ds3_obj_create` creates the object along with all the directories in the object key. To achieve this, the key is split by `/`, then for each part of the key, a directory is looked up using `dfs_lookup_rel` or created using `dfs_mkdir` if it did not exist, then move on to the next part. For the last part of the key, a DFS file is created using `dfs_open` and the open DS3 file handle is stored in the `DaosObject` entity. The handle is then used to write data to the object using `ds3_obj_write` (via `dfs_write`). When the `DaosObject` is deconstructed, the DS3 object handle is closed using `ds3_obj_close`.



`DELETE` Operation

In a `DELETE` operation, `ds3_obj_destroy` looks up the object's parent directory within the directory structure using `dfs_lookup`, and an open DFS directory handle is returned. The handle is then used to remove the object using `dfs_remove`. When the operation is done, the DFS directory handle is released using `dfs_release`.



Users

In order to integrate with S3, DGW needs to store some user information and metadata. This includes user id, name, email, access keys, and the access policies for the user. This information is shared across the whole system, so it must be stored in a place accessible at the pool level. DAOS supports simple ACL policies on containers and pools, but S3 requires more user metadata and the complicated S3 policies do not map nicely to DAOS ACL policies. Users are thus stored in the metadata container as shown in the next section

The Metadata Container

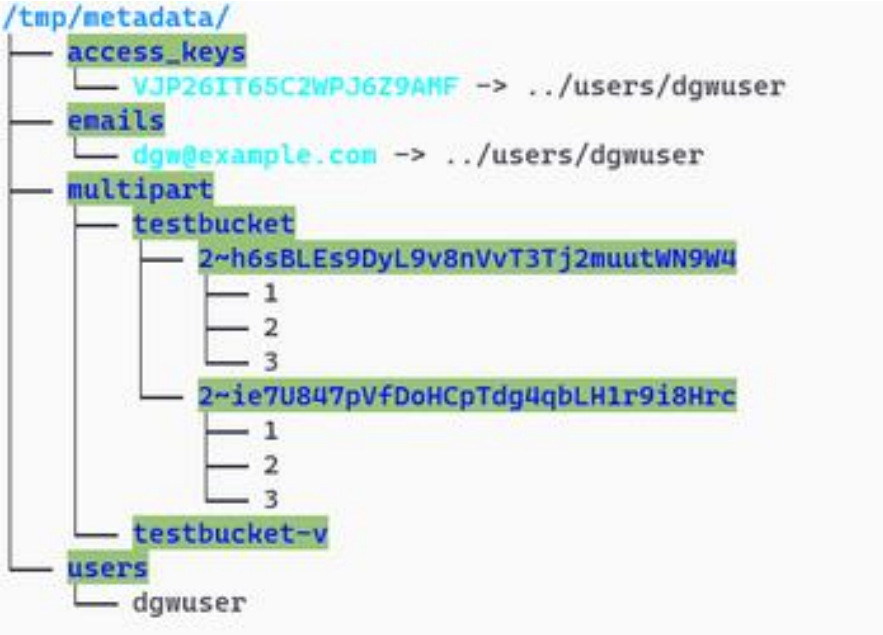
A new hidden DAOS container called `_METADATA` is used to store all pool metadata. The user list and other metadata are then stored as DFS objects within it. The same method could be used to store other miscellaneous metadata we might need in the future in the same bucket.

S3 bucket names cannot contain underscores or capital letters [https://docs.aws.amazon.com/AmazonS3/latest/userguide/bucketnamingrules.html], so the `_METADATA` bucket will not be accessible to the S3 interface. Additionally, a rule has been set in place to prevent the bucket from being loaded.

`_METADATA` directory structure

- `/users/` contains all the user data as files, with the user id as the file name and the user data as the file contents.
- `/emails/` contains soft link files. The name of each file is the email, while the file pointed to is the user data file in `/users/`.
- `/access_keys/` contains soft link files. The name of each file is the access key, while the file pointed to is the user data file in `/users/`.
- `/multipart/` contains multipart data for each bucket. See below for internal structure.

Example contents



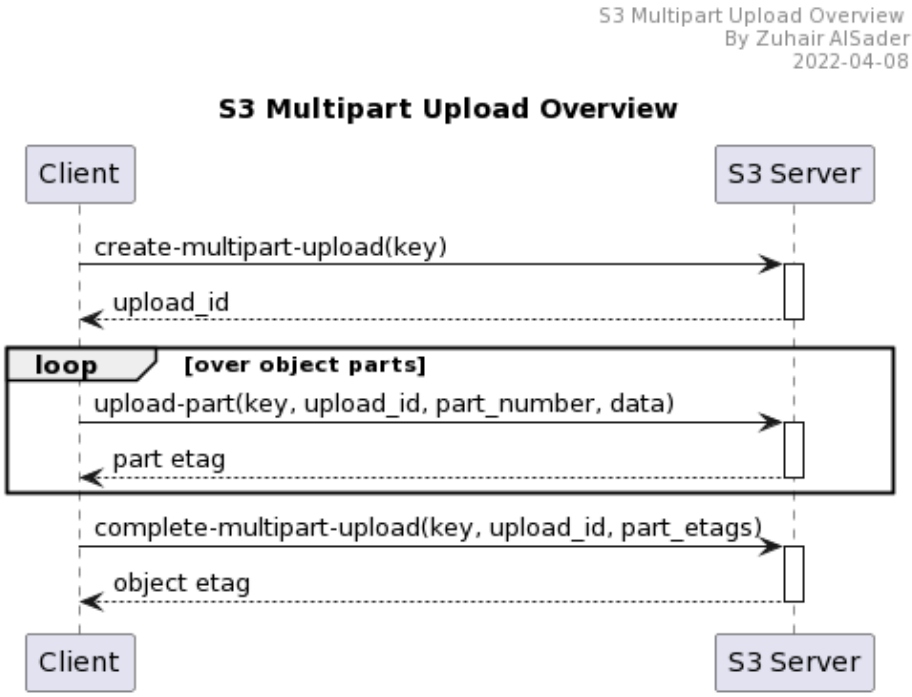
DS3 User API

- `ds3_user_set` adds or updates user information in the S3 user database.
- `ds3_user_remove` removes user from S3 user database.
- `ds3_user_get` looks up S3 user information by name.
- `ds3_user_get_by_email` looks up S3 user information by email.
- `ds3_user_get_by_key` looks up S3 user information by access key. Checking the user secret is done in RGW.

Multipart upload

Reference for multipart upload:
https://docs.aws.amazon.com/AmazonS3/latest/userguide/mpuoverview.html

Multipart upload allows uploading a single object as a set of parts. Each part is a contiguous portion of the object's data. The object parts could be uploaded independently and in any order. If transmission of any part fails, it can be retransmitted without affecting other parts. After all parts of the object are uploaded, the S3 layer assembles these parts and creates the object. The multipart upload is done on three steps: initiating the upload, uploading the object parts, then completing the upload. Upon completion, the object is constructed from the parts and the object is accessible like any other object in the bucket.



Design Summary:

- Multipart upload initiation (using `ds3_upload_init`)
  - Generate a unique `multipart_upload_id` that identifies the upload
  - Create the directory `upload_dir = multipart/<bucket_name>/<multipart_upload_id>` in the `_METADATA` bucket
  - Add any metadata and attributes related to the upload in the `rgw_entry` xattr of the `upload_dir` directory
  - Store the final object key in `rgw_key` xattr of the `upload_dir` directory
- Parts upload
  - In `ds3_part_open`, create or open and truncate a DFS object under `<upload_dir>/<part_id>`, and return opened part handle
  - Write the part data using `ds3_part_write`
  - `ds3_part_set_info` stores the metadata about each part in an xattr `rgw_part` of the part object
- Multipart upload completion
  - Create the object under the correct key and bucket, creating any missing directories in the process
  - Copy data from each part to the created object ordered by `part_id`, (uses `ds3_part_read`)
  - Move the metadata and attributes from the upload directory (read by `ds3_upload_get_info`) to the object
  - Delete the `upload_dir` directory created during initiation, including the parts, using `ds3_upload_remove`

Versioning

Versioning is provided by the SAL interface through setting the version id if it exists in the object name between brackets (`object/key[versionId]`), see examples below. The We maintain a symbolic link to the latest version of the object with the suffix `[latest]`. This is to allow easier access to the latest version of the object when reading the file or updating the latest version. During an upload, the function `ds3_obj_mark_latest` is used to mark the uploaded object as latest and make the `[latest]` link point to it.

Example contents

```
/tmp/dfuse/
├── 50KB
├── 50KB[KuHZRXp--pb9-RejoJ7GjtFfH7HeUyu]
├── 50KB[latest] -> 50KB[w82ZxwLNz0lPwgnMTGa.TRZhAxNqZHY]
├── 50KBsus
├── 50KB[w82ZxwLNz0lPwgnMTGa.TRZhAxNqZHY]
├── 98MB[l2q6EMMDF.bhJQhdBBrv4HHlHmzQEjv]
├── 98MB[latest] -> 98MB[l2q6EMMDF.bhJQhdBBrv4HHlHmzQEjv]
└── 98MB[QSLwVcE79afGVYDD84qHNGR8reQARVf]
```

TODO: Delete an object when its versioning is turned on. Do we delete all versions? Just latest?

Object Index

To solve the issue of ordering, markers and using other delimiters, a suggestion is to use a DAOS object with sorted dkeys (`DAOS_OT_DKEY_LEXICAL`). The details of this are being worked out.

Caching

Not yet implemented - more research is needed.

Goals:

- Cache open bucket handles
- Cache open object handles
- Avoid searching deep when doing bucket index.
- Cache metadata for objects

Candidates:

- boost::lru_cache; value can be any object, including something that holds open pointers
- ObjectCache in rgw_cache; value is anything that fits in a buffer list; no destructor
- Intel's gurt cache; used in `dfs_sys`. Maybe we should use `dfs_sys`

Notifications

To be researched.

Object Lock

To be researched.

Existing Pools

We can easily add all the necessary metadata to existing pools. We should add that in the initialize code.

Design Challenges

Ordering

DAOS DFS API do not support listing directory contents in order, so they will be returned in random ordering. This is a challenge when designing the list operation because we require being able to list the keys in order, and be able to return keys after a certain key marker.

Handling concurrent writes

The initial implementation of `PUT` has a bug where if a file is uploaded by two clients at the same time, the writes might interfere, causing the file contents to be corrupted. See this script for testing:

```
head -c 1GB /dev/urandom > /tmp/1GB1
head -c 1GB /dev/urandom > /tmp/1GB2

s3cmd put /tmp/1GB1 s3://testbucket/1GB --disable-multipart > /dev/null &
s3cmd put /tmp/1GB2 s3://testbucket/1GB --disable-multipart > /dev/null &

wait

s3cmd get s3://testbucket/1GB /tmp/1GB-get --force

md5sum /tmp/1GB1 /tmp/1GB2 /tmp/1GB-get

# Output:
WARNING: MD5 signatures do not match:
computed=92a0b0e643d278eab01a38315a3ef67b,
received=352c92e2e9958101635ff8758291e903
352c92e2e9958101635ff8758291e903  /tmp/1GB1
83c889ed0855d23c99c2ec83e1f757d9  /tmp/1GB2
92a0b0e643d278eab01a38315a3ef67b  /tmp/1GB-get
```

Multipart Upload Performance

The current design for multipart object uploads the parts to the metadata bucket then copies them to the object. This might cause some performance degradation on the complete multipart operation. Getting rid of the additional copy could improve performance.

Metadata Performance

The current design opens and closes a lot of object handles. For example, a list operation needs to access the metadata of each object in the bucket, which currently requires opening and closing each dfs object in the container that has the specified prefix.

Listing Objects with a Delimiter Other than `/`

The current implementation returns `EINVAL` when trying to list a bucket with a delimiter other than `/`. This is required because DGW internally translates `/` into a directories. To be fully S3 compatible, we need to support any prefix. One solution is to maintain a separate sorted full-key index for listing objects as mentioned above. Another is to use a tree searching algorithm (e.g. Depth-first Search) to find all objects up to a certain delimiter. We need to understand the performance implications of either decision.