scality-s3-server

Release 7.0.0

Jun 30, 2017

Documentation

CHAPTER 1

Contributing

Need help?

We're always glad to help out. Simply open a GitHub issue and we'll give you insight. If what you want is not available, and if you're willing to help us out, we'll be happy to welcome you in the team, whether for a small fix or for a larger feature development. Thanks for your interest!

Got an idea? Get started!

In order to contribute, please follow the Contributing Guidelines. If anything is unclear to you, reach out to us on slack or via a GitHub issue.

Don't write code? There are other ways to help!

We're always eager to learn about our users' stories. If you can't contribute code, but would love to help us, please shoot us an email at zenko@scality.com, and tell us what our software enables you to do! Thanks for your time!

CHAPTER 2

Getting Started

Installation

Dependencies

Building and running the Scality S3 Server requires node.js 6.9.5 and npm v3 . Up-to-date versions can be found at Nodesource.

Clone source code

```
git clone https://github.com/scality/S3.git
```

Install js dependencies

Go to the ./S3 folder,

```
npm install
```

Run it with a file backend

```
npm start
```

This starts an S3 server on port 8000. Two additional ports 9990 and 9991 are also open locally for internal transfer of metadata and data, respectively.

The default access key is accessKey1 with a secret key of verySecretKey1.

By default the metadata files will be saved in the localMetadata directory and the data files will be saved in the localData directory within the ./S3 directory on your machine. These directories have been pre-created within the repository. If you would like to save the data or metadata in different locations of your choice, you must specify them with absolute paths. So, when starting the server:

```
mkdir -m 700 $(pwd)/myFavoriteDataPath
mkdir -m 700 $(pwd)/myFavoriteMetadataPath
export S3DATAPATH="$(pwd)/myFavoriteDataPath"
export S3METADATAPATH="$(pwd)/myFavoriteMetadataPath"
npm start
```

Run it with multiple data backends

```
export S3DATA='multiple'
npm start
```

This starts an S3 server on port 8000. The default access key is accessKey1 with a secret key of verySecretKey1.

With multiple backends, you have the ability to choose where each object will be saved by setting the following header with a locationConstraint on a PUT request:

```
'x-amz-meta-scal-location-constraint':'myLocationConstraint'
```

If no header is sent with a PUT object request, the location constraint of the bucket will determine where the data is saved. If the bucket has no location constraint, the endpoint of the PUT request will be used to determine location.

See the Configuration section below to learn how to set location constraints.

Run it with an in-memory backend

```
npm run mem_backend
```

This starts an S3 server on port 8000. The default access key is accessKey1 with a secret key of verySecretKey1.

Run it for continuous integration testing or in production with Docker

DOCKER.rst

Testing

You can run the unit tests with the following command:

```
npm test
```

You can run the multiple backend unit tests with:

```
npm run multiple_backend_test
```

You can run the linter with:

```
npm run lint
```

Running functional tests locally:

The test suite requires additional tools, **s3cmd** and **Redis** installed in the environment the tests are running in.

- Install s3cmd
- Install redis and start Redis.
- Add localCache section to your `config.json`:

```
"localCache": {
    "host": REDIS_HOST,
    "port": REDIS_PORT
}
```

where `REDIS_HOST` is your Redis instance IP address (`"127.0.0.1"` if your Redis is running locally) and `REDIS_PORT` is your Redis instance port (`6379` by default)

- Add the following to the etc/hosts file on your machine:

```
127.0.0.1 bucketwebsitetester.s3-website-us-east-1.amazonaws.com
```

- Start the S3 server in memory and run the functional tests:

```
npm run mem_backend
npm run ft_test
```

Configuration

There are three configuration files for your Scality S3 Server:

1. `conf/authdata.json`, described above for authentication
2. `locationConfig.json`, to set up configuration options for

   where data will be saved
3. `config.json`, for general configuration options

Location Configuration

You must specify at least one locationConstraint in your locationConfig.json (or leave as pre-configured).

You must also specify 'us-east-1' as a locationConstraint so if you only define one locationConstraint, that would be it. If you put a bucket to an unknown endpoint and do not specify a locationConstraint in the put bucket call, us-east-1 will be used.

For instance, the following locationConstraint will save data sent to `myLocationConstraint` to the file backend:

```
"myLocationConstraint": {
    "type": "file",
    "legacyAwsBehavior": false,
    "details": {}
},
```

Each locationConstraint must include the `type`, `legacyAwsBehavior`, and `details` keys. `type` indicates which backend will be used for that region. Currently, mem, file, and scality are the supported backends. `legacyAwsBehavior` indicates whether the region will have the same behavior as the AWS S3 'us-east-1' region. If the locationConstraint type is scality, `details` should contain connector information for sproxyd. If the locationConstraint type is mem or file, `details` should be empty.

Once you have your locationConstraints in your locationConfig.json, you can specify a default locationConstraint for each of your endpoints.

For instance, the following sets the `localhost` endpoint to the `myLocationConstraint` data backend defined above:

```
"restEndpoints": {
     "localhost": "myLocationConstraint"
},
```

If you would like to use an endpoint other than localhost for your Scality S3 Server, that endpoint MUST be listed in your `restEndpoints`. Otherwise if your server is running with a:

- **file backend**: your default location constraint will be `file`
- **memory backend**: your default location constraint will be `mem`

Endpoints

Note that our S3server supports both:

- path-style: http://myhostname.com/mybucket
- hosted-style: http://mybucket.myhostname.com

However, hosted-style requests will not hit the server if you are using an ip address for your host. So, make sure you are using path-style requests in that case. For instance, if you are using the AWS SDK for JavaScript, you would instantiate your client like this:

```
const s3 = new aws.S3({
   endpoint: 'http://127.0.0.1:8000',
   s3ForcePathStyle: true,
});
```

Setting your own access key and secret key pairs

You can set credentials for many accounts by editing `conf/authdata.json` but if you want to specify one set of your own credentials, you can use `SCALITY_ACCESS_KEY_ID` and `SCALITY_SECRET_ACCESS_KEY` environment variables.

SCALITY_ACCESS_KEY_ID and SCALITY_SECRET_ACCESS_KEY

These variables specify authentication credentials for an account named “CustomAccount”.

Note: Anything in the `authdata.json` file will be ignored.

```
SCALITY_ACCESS_KEY_ID=newAccessKey SCALITY_SECRET_ACCESS_KEY=newSecretKey npm start
```

Scality with SSL

If you wish to use https with your local S3 Server, you need to set up SSL certificates. Here is a simple guide of how to do it.

Deploying S3 Server

First, you need to deploy **S3 Server**. This can be done very easily via our **DockerHub** page (you want to run it with a file backend).

> *Note: - If you don’t have docker installed on your machine, here are the ‘instructions to install it for your distribution <https://docs.docker.com/engine/installation/>‘__*

Updating your S3 Server container’s config

You’re going to add your certificates to your container. In order to do so, you need to exec inside your s3 server container. Run a `$> docker ps` and find your container’s id (the corresponding image name should be `scality/s3server`. Copy the corresponding container id (here we’ll use `894aee038c5e`, and run:

```
$> docker exec -it 894aee038c5e bash
```

You’re now inside your container, using an interactive terminal :)

Generate SSL key and certificates

There are 5 steps to this generation. The paths where the different files are stored are defined after the `-out` option in each command

```
# Generate a private key for your CSR
$> openssl genrsa -out ca.key 2048
# Generate a self signed certificate for your local Certificate Authority
$> openssl req -new -x509 -extensions v3_ca -key ca.key -out ca.crt -days 99999  -
↪subj "/C=US/ST=Country/L=City/O=Organization/CN=scality.test"

# Generate a key for S3 Server
$> openssl genrsa -out test.key 2048
# Generate a Certificate Signing Request for S3 Server
$> openssl req -new -key test.key -out test.csr -subj "/C=US/ST=Country/L=City/
↪O=Organization/CN=*.scality.test"
# Generate a local-CA-signed certificate for S3 Server
$> openssl x509 -req -in test.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out test.
↪crt -days 99999 -sha256
```

Update S3Server `config.json`

Add a `certFilePaths` section to `./config.json` with the appropriate paths:

```
"certFilePaths": {
    "key": "./test.key",
    "cert": "./test.crt",
    "ca": "./ca.crt"
}
```

Run your container with the new config

First, you need to exit your container. Simply run `$> exit`. Then, you need to restart your container. Normally, a simple `$> docker restart s3server` should do the trick.

Update your host config

Associates local IP addresses with hostname

In your `/etc/hosts` file on Linux, OS X, or Unix (with root permissions), edit the line of localhost so it looks like this:

```
127.0.0.1      localhost s3.scality.test
```

Copy the local certificate authority from your container

In the above commands, it's the file named `ca.crt`. Choose the path you want to save this file at (here we chose `/root/ca.crt`), and run something like:

```
$> docker cp 894aee038c5e:/usr/src/app/ca.crt /root/ca.crt
```

Test your config

If you do not have aws-sdk installed, run `$> npm install aws-sdk`. In a `test.js` file, paste the following script:

```
const AWS = require('aws-sdk');
const fs = require('fs');
const https = require('https');

const httpOptions = {
    agent: new https.Agent({
        // path on your host of the self-signed certificate
        ca: fs.readFileSync('./ca.crt', 'ascii'),
    }),
};

const s3 = new AWS.S3({
    httpOptions,
    accessKeyId: 'accessKey1',
    secretAccessKey: 'verySecretKey1',
    // The endpoint must be s3.scality.test, else SSL will not work
    endpoint: 'https://s3.scality.test:8000',
    sslEnabled: true,
    // With this setup, you must use path-style bucket access
    s3ForcePathStyle: true,
});

const bucket = 'cocoriko';

s3.createBucket({ Bucket: bucket }, err => {
    if (err) {
        return console.log('err createBucket', err);
    }
    return s3.deleteBucket({ Bucket: bucket }, err => {
        if (err) {
            return console.log('err deleteBucket', err);
        }
        return console.log('SSL is cool!');
    });
});
```

Now run that script with `$> nodejs test.js`. If all goes well, it should output `SSL is cool!`. Enjoy that added security!

CHAPTER 3

Clients

List of applications that have been tested with S3 Server

GUI

Cyberduck

- https://www.youtube.com/watch?v=-n2MCt4ukUg
- https://www.youtube.com/watch?v=IyXHcu4uqgU

Cloud Explorer

- https://www.youtube.com/watch?v=2hhtBtmBSxE

CloudBerry Lab

- https://youtu.be/IjIx8g_o0gY

Command Line Tools

s3curl

https://github.com/scality/S3/blob/master/tests/functional/s3curl/s3curl.pl

aws-cli

`~/.aws/credentials` on Linux, OS X, or Unix or `C:\Users\USERNAME\.aws\credentials` on Windows

```
[default]
aws_access_key_id = accessKey1
aws_secret_access_key = verySecretKey1
```

`~/.aws/config` on Linux, OS X, or Unix or `C:\Users\USERNAME\.aws\config` on Windows

```
[default]
region = us-east-1
```

Note: `us-east-1` is the default region, but you can specify any region.

See all buckets:

```
aws s3 ls --endpoint-url=http://localhost:8000
```

Create bucket:

```
aws --endpoint-url=http://localhost:8000 s3 mb s3://mybucket
```

s3cmd

If using s3cmd as a client to S3 be aware that v4 signature format is buggy in s3cmd versions < 1.6.1.

`~/.s3cfg` on Linux, OS X, or Unix or `C:\Users\USERNAME\.s3cfg` on Windows

```
[default]
access_key = accessKey1
secret_key = verySecretKey1
host_base = localhost:8000
host_bucket = %(bucket).localhost:8000
signature_v2 = False
use_https = False
```

See all buckets:

```
s3cmd ls
```

rclone

`~/.rclone.conf` on Linux, OS X, or Unix or `C:\Users\USERNAME\.rclone.conf` on Windows

```
[remote]
type = s3
env_auth = false
access_key_id = accessKey1
secret_access_key = verySecretKey1
region = other-v2-signature
endpoint = http://localhost:8000
location_constraint =
acl = private
```

```
server_side_encryption =
storage_class =
```

See all buckets:

```
rclone lsd remote:
```

JavaScript

AWS JavaScript SDK

```
const AWS = require('aws-sdk');

const s3 = new AWS.S3({
    accessKeyId: 'accessKey1',
    secretAccessKey: 'verySecretKey1',
    endpoint: 'localhost:8000',
    sslEnabled: false,
    s3ForcePathStyle: true,
});
```

JAVA

AWS JAVA SDK

```
import com.amazonaws.auth.AWSCredentials;
import com.amazonaws.auth.BasicAWSCredentials;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3Client;
import com.amazonaws.services.s3.S3ClientOptions;
import com.amazonaws.services.s3.model.Bucket;

public class S3 {

    public static void main(String[] args) {

        AWSCredentials credentials = new BasicAWSCredentials("accessKey1",
        "verySecretKey1");

        // Create a client connection based on credentials
        AmazonS3 s3client = new AmazonS3Client(credentials);
        s3client.setEndpoint("http://localhost:8000");
        // Using path-style requests
        // (deprecated) s3client.setS3ClientOptions(new S3ClientOptions().
↪withPathStyleAccess(true));
        s3client.setS3ClientOptions(S3ClientOptions.builder().
↪setPathStyleAccess(true).build());

        // Create bucket
        String bucketName = "javabucket";
        s3client.createBucket(bucketName);
        // List off all buckets
        for (Bucket bucket : s3client.listBuckets()) {
            System.out.println(" - " + bucket.getName());
        }
    }
}
```

Ruby

AWS SDK for Ruby - Version 2

```
require 'aws-sdk'

s3 = Aws::S3::Client.new(
  :access_key_id => 'accessKey1',
  :secret_access_key => 'verySecretKey1',
  :endpoint => 'http://localhost:8000',
  :force_path_style => true
)

resp = s3.list_buckets
```

fog

```
require "fog"

connection = Fog::Storage.new(
{
    :provider => "AWS",
    :aws_access_key_id => 'accessKey1',
    :aws_secret_access_key => 'verySecretKey1',
    :endpoint => 'http://localhost:8000',
    :path_style => true,
    :scheme => 'http',
})
```

Python

boto2

```
import boto
from boto.s3.connection import S3Connection, OrdinaryCallingFormat


connection = S3Connection(
    aws_access_key_id='accessKey1',
    aws_secret_access_key='verySecretKey1',
    is_secure=False,
    port=8000,
    calling_format=OrdinaryCallingFormat(),
    host='localhost'
)

connection.create_bucket('mybucket')
```

boto3

```
import boto3
client = boto3.client(
    's3',
    aws_access_key_id='accessKey1',
    aws_secret_access_key='verySecretKey1',
    endpoint_url='http://localhost:8000'
)

lists = client.list_buckets()
```

PHP

Should use v3 over v2 because v2 would create virtual-hosted style URLs while v3 generates path-style URLs.

AWS PHP SDK v3

```
use Aws\S3\S3Client;

$client = S3Client::factory([
    'region'  => 'us-east-1',
    'version'   => 'latest',
    'endpoint' => 'http://localhost:8000',
    'credentials' => [
         'key'    => 'accessKey1',
         'secret' => 'verySecretKey1'
    ]
]);

$client->createBucket(array(
    'Bucket' => 'bucketphp',
));
```

CHAPTER 4

Docker

For continuous integration with Docker

When you start the Docker Scality S3 server image, you can adjust the configuration of the Scality S3 server instance by passing one or more environment variables on the docker run command line.

Environment Variables

S3DATA=multiple

This runs Scality S3 server with multiple data backends. More info

```
docker run -d --name s3server -p 8000:8000 -e S3DATA=multiple scality/s3server
```

For running an S3 AWS backend, you will have to add a new section (with the `aws_s3` location type) to your `locationConfig.json` file:

```
"aws-test": {
    "type": "aws_s3",
    "details": {
        "awsEndpoint": "s3.amazonaws.com",
        "bucketName": "yourawss3bucket",
        "bucketMatch": true,
        "credentialsProfile": "default"
    }
}
```

You will also have to mount your AWS credentials file: `-v ~/.aws/credentials:/root/.aws/credentials` on Linux, OS X, or Unix or `-v C:\Users\USERNAME\.aws\credential:/root/.aws/credentials` on Windows

```
docker run --name s3server -p 8000:8000
-v $(pwd)/locationConfig.json:/usr/src/app/locationConfig.json
-v ~/.aws/credentials:/root/.aws/credentials -e S3DATA=multiple scality/s3server
```

HOST_NAME

This variable specifies a host name. If you have a domain such as new.host.com, by specifying that here, you and your users can direct s3 server requests to new.host.com.

```
docker run -d --name s3server -p 8000:8000 -e HOST_NAME=new.host.com scality/s3server
```

Note: In your `/etc/hosts` file on Linux, OS X, or Unix with root permissions, make sure to associate 127.0.0.1 with `new.host.com`

SCALITY_ACCESS_KEY_ID and SCALITY_SECRET_ACCESS_KEY

These variables specify authentication credentials for an account named “CustomAccount”.

You can set credentials for many accounts by editing `conf/authdata.json` (see below for further info), but if you just want to specify one set of your own, you can use these environment variables.

```
docker run -d --name s3server -p 8000:8000 -e SCALITY_ACCESS_KEY_ID=newAccessKey
-e SCALITY_SECRET_ACCESS_KEY=newSecretKey scality/s3server
```

Note: Anything in the `authdata.json` file will be ignored. Note: The old `ACCESS_KEY` and `SECRET_KEY` environment variables are now deprecated

LOG_LEVEL

This variable allows you to change the log level: info, debug or trace. The default is info. Debug will give you more detailed logs and trace will give you the most detailed.

```
docker run -d --name s3server -p 8000:8000 -e LOG_LEVEL=trace scality/s3server
```

SSL

This variable specifies the Common Name `<DOMAIN_NAME>` used to create the Certificate Signing Request using OpenSSL. This allows you to run S3 with SSL:

Note: In your `/etc/hosts` file on Linux, OS X, or Unix with root permissions, make sure to associate 127.0.0.1 with `<SUBDOMAIN>.<DOMAIN_NAME>`

Warning: These certs, being self-signed (and the CA being generated inside the container) will be untrusted by any clients, and could disappear on a container upgrade. That's ok as long as it's for quick testing. Also, best security practice for non-testing would be to use an extra container to do SSL/TLS termination such as haproxy/nginx/stunnel to limit what an exploit on either component could expose, as well as certificates in a mounted volume

```
docker run -d --name s3server -p 8000:8000 -e SSL=<DOMAIN_NAME> -e HOST_NAME=
↪<SUBDOMAIN>.<DOMAIN_NAME>
scality/s3server
```

More information about how to use S3 server with SSL here

LISTEN_ADDR

This variable instructs the S3 server, and its data and metadata components to listen on the specified address. This allows starting the data or metadata servers as standalone services, for example.

```
docker run -d --name s3server-data -p 9991:9991 -e LISTEN_ADDR=0.0.0.0
scality/s3server npm run start_dataserver
```

DATA_HOST and METADATA_HOST

These variables configure the data and metadata servers to use, usually when they are running on another host and only starting the stateless S3 REST server.

```
docker run -d --name s3server -e DATA_HOST=s3server-data
-e METADATA_HOST=s3server-metadata scality/s3server npm run start_s3server
```

REDIS_HOST

Use this variable to connect to the redis cache server on another host than localhost.

```
docker run -d --name s3server -p 8000:8000
-e REDIS_HOST=my-redis-server.example.com scality/s3server
```

REDIS_PORT

Use this variable to connect to the redis cache server on another port than the default 6379.

```
docker run -d --name s3server -p 8000:8000
-e REDIS_PORT=6379 scality/s3server
```

In production with Docker

Using Docker Volume in production

S3 server runs with a file backend by default.

So, by default, the data is stored inside your S3 server Docker container.

However, if you want your data and metadata to persist, you **MUST** use Docker volumes to host your data and metadata outside your s3 server Docker container. Otherwise, the data and metadata will be destroyed when you erase the container.

```
docker run -v $(pwd)/data:/usr/src/app/localData -v $(pwd)/metadata:/usr/src/app/
↪localMetadata
-p 8000:8000 -d scality/s3server
```

This command mounts the host directory, `./data`, into the container at /usr/src/app/localData and the host directory, `./metadata`, into the container at /usr/src/app/localMetaData. It can also be any host mount point, like `/mnt/data` and `/mnt/metadata`.

Adding modifying or deleting accounts or users credentials

1. Create locally a customized `authdata.json`.
2. Use Docker Volume

to override the default `authdata.json` through a docker file mapping. For example:

```
docker run -v $(pwd)/authdata.json:/usr/src/app/conf/authdata.json -p 8000:8000 -d
scality/s3server
```

Specifying your own host name

To specify a host name (e.g. s3.domain.name), you can provide your own config.json using Docker Volume.

First add a new key-value pair in the restEndpoints section of your config.json. The key in the key-value pair should be the host name you would like to add and the value is the default location_constraint for this endpoint.

For example, `s3.example.com` is mapped to `us-east-1` which is one of the `location_constraints` listed in your locationConfig.json file here.

More information about location configuration here

```
"restEndpoints": {
    "localhost": "file",
    "127.0.0.1": "file",
    ...
    "s3.example.com": "us-east-1"
},
```

Then, run your Scality S3 Server using Docker Volume:

```
docker run -v $(pwd)/config.json:/usr/src/app/config.json -p 8000:8000 -d scality/
↪s3server
```

Your local `config.json` file will override the default one through a docker file mapping.

Running as an unprivileged user

S3 Server runs as root by default.

You can change that by modifing the dockerfile and specifying a user before the entrypoint.

The user needs to exist within the container, and own the folder **/usr/src/app** for Scality S3 Server to run properly.

For instance, you can modify these lines in the dockerfile:

```
...
&& groupadd -r -g 1001 scality \
&& useradd -u 1001 -g 1001 -d /usr/src/app -r scality \
&& chown -R scality:scality /usr/src/app

...

USER scality
ENTRYPOINT ["/usr/src/app/docker-entrypoint.sh"]
```

CHAPTER 5

Integrations

High Availability

Docker swarm is a clustering tool developped by Docker and ready to use with its containers. It allows to start a service, which we define and use as a means to ensure s3server's continuous availability to the end user. Indeed, a swarm defines a manager and n workers among n+1 servers. We will do a basic setup in this tutorial, with just 3 servers, which already provides a strong service resiliency, whilst remaining easy to do as an individual. We will use NFS through docker to share data and metadata between the different servers.

You will see that the steps of this tutorial are defined as **On Server**, **On Clients**, **On All Machines**. This refers respectively to NFS Server, NFS Clients, or NFS Server and Clients. In our example, the IP of the Server will be **10.200.15.113**, while the IPs of the Clients will be **10.200.15.96 and 10.200.15.97**

Installing docker

Any version from docker 1.12.6 onwards should work; we used Docker 17.03.0-ce for this tutorial.

On All Machines

On Ubuntu 14.04

The docker website has solid documentation. We have chosen to install the aufs dependency, as recommended by Docker. Here are the required commands:

```
$> sudo apt-get update
$> sudo apt-get install linux-image-extra-$(uname -r) linux-image-extra-virtual
$> sudo apt-get install apt-transport-https ca-certificates curl software-properties-
↪common
$> curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
$> sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu
↪$(lsb_release -cs) stable"
```

```
$> sudo apt-get update
$> sudo apt-get install docker-ce
```

On CentOS 7

The docker website has solid documentation. Here are the required commands:

```
$> sudo yum install -y yum-utils
$> sudo yum-config-manager --add-repo https://download.docker.com/linux/centos/docker-
↪ce.repo
$> sudo yum makecache fast
$> sudo yum install docker-ce
$> sudo systemctl start docker
```

Configure NFS

On Clients

Your NFS Clients will mount Docker volumes over your NFS Server's shared folders. Hence, you don't have to mount anything manually, you just have to install the NFS commons:

On Ubuntu 14.04

Simply install the NFS commons:

```
$> sudo apt-get install nfs-common
```

On CentOS 7

Install the NFS utils, and then start the required services:

```
$> yum install nfs-utils
$> sudo systemctl enable rpcbind
$> sudo systemctl enable nfs-server
$> sudo systemctl enable nfs-lock
$> sudo systemctl enable nfs-idmap
$> sudo systemctl start rpcbind
$> sudo systemctl start nfs-server
$> sudo systemctl start nfs-lock
$> sudo systemctl start nfs-idmap
```

On Server

Your NFS Server will be the machine to physically host the data and metadata. The package(s) we will install on it is slightly different from the one we installed on the clients.

On Ubuntu 14.04

Install the NFS server specific package and the NFS commons:

```
$> sudo apt-get install nfs-kernel-server nfs-common
```

On CentOS 7

Same steps as with the client: install the NFS utils and start the required services:

```
$> yum install nfs-utils
$> sudo systemctl enable rpcbind
$> sudo systemctl enable nfs-server
$> sudo systemctl enable nfs-lock
$> sudo systemctl enable nfs-idmap
$> sudo systemctl start rpcbind
$> sudo systemctl start nfs-server
$> sudo systemctl start nfs-lock
$> sudo systemctl start nfs-idmap
```

On Ubuntu 14.04 and CentOS 7

Choose where your shared data and metadata from your local S3 Server will be stored. We chose to go with /var/nfs/data and /var/nfs/metadata. You also need to set proper sharing permissions for these folders as they'll be shared over NFS:

```
$> mkdir -p /var/nfs/data /var/nfs/metadata
$> chmod -R 777 /var/nfs/
```

Now you need to update your **/etc/exports** file. This is the file that configures network permissions and rwx permissions for NFS access. By default, Ubuntu applies the no_subtree_check option, so we declared both folders with the same permissions, even though they're in the same tree:

```
$> sudo vim /etc/exports
```

In this file, add the following lines:

```
/var/nfs/data        10.200.15.96(rw,sync,no_root_squash) 10.200.15.97(rw,sync,no_
↪root_squash)
/var/nfs/metadata    10.200.15.96(rw,sync,no_root_squash) 10.200.15.97(rw,sync,no_
↪root_squash)
```

Export this new NFS table:

```
$> sudo exportfs -a
```

Eventually, you need to allow for NFS mount from Docker volumes on other machines. You need to change the Docker config in **/lib/systemd/system/docker.service**:

```
$> sudo vim /lib/systemd/system/docker.service
```

In this file, change the **MountFlags** option:

```
MountFlags=shared
```

Now you just need to restart the NFS server and docker daemons so your changes apply.

On Ubuntu 14.04

Restart your NFS Server and docker services:

```
$> sudo service nfs-kernel-server restart
$> sudo service docker restart
```

On CentOS 7

Restart your NFS Server and docker daemons:

```
$> sudo systemctl restart nfs-server
$> sudo systemctl daemon-reload
$> sudo systemctl restart docker
```

Set up your Docker Swarm service

On All Machines

On Ubuntu 14.04 and CentOS 7

We will now set up the Docker volumes that will be mounted to the NFS Server and serve as data and metadata storage for S3 Server. These two commands have to be replicated on all machines:

```
$> docker volume create --driver local --opt type=nfs --opt o=addr=10.200.15.113,rw --
↪opt device=:/var/nfs/data --name data
$> docker volume create --driver local --opt type=nfs --opt o=addr=10.200.15.113,rw --
↪opt device=:/var/nfs/metadata --name metadata
```

There is no need to “”docker exec” these volumes to mount them: the Docker Swarm manager will do it when the Docker service will be started.

On Server

To start a Docker service on a Docker Swarm cluster, you first have to initialize that cluster (i.e.: define a manager), then have the workers/nodes join in, and then start the service. Initialize the swarm cluster, and look at the response:

```
$> docker swarm init --advertise-addr 10.200.15.113

Swarm initialized: current node (db2aqfu3bzfzzs9b1kfeaglmq) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join \
    --token SWMTKN-1-5yxxencrdoelr7mpltljn325uz4v6fe1gojl14lzceij3nujzu-
↪2vfs9u6ipgcq35r90xws3stka \
    10.200.15.113:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the␣
↪instructions.
```

On Clients

Simply copy/paste the command provided by your docker swarm init. When all goes well, you'll get something like this:

```
$> docker swarm join --token SWMTKN-1-
↪5yxxencrdoelr7mpltljn325uz4v6fe1gojl14lzceij3nujzu-2vfs9u6ipgcq35r90xws3stka 10.200.
↪15.113:2377

This node joined a swarm as a worker.
```

On Server

Start the service on your swarm cluster!

```
$> docker service create --name s3 --replicas 1 --mount type=volume,source=data,
↪target=/usr/src/app/localData --mount type=volume,source=metadata,target=/usr/src/
↪app/localMetadata -p 8000:8000 scality/s3server
```

If you run a docker service ls, you should have the following output:

```
$> docker service ls
ID            NAME  MODE        REPLICAS  IMAGE
ocmggza412ft  s3    replicated  1/1       scality/s3server:latest
```

If your service won't start, consider disabling apparmor/SELinux.

Testing your High Availability S3Server

On All Machines

On Ubuntu 14.04 and CentOS 7

Try to find out where your Scality S3 Server is actually running using the **docker ps** command. It can be on any node of the swarm cluster, manager or worker. When you find it, you can kill it, with **docker stop <container id>** and you'll see it respawn on a different node of the swarm cluster. Now you see, if one of your servers falls, or if docker stops unexpectedly, your end user will still be able to access your local S3 Server.

Troubleshooting

To troubleshoot the service you can run:

```
$> docker service ps s3docker service ps s3
ID                         NAME      IMAGE             NODE                              ␣
↪     DESIRED STATE  CURRENT STATE       ERROR
```

```
0ar81cw4lvv8chafm8pw48wbc  s3.1      scality/s3server  localhost.localdomain.
↪localdomain  Running        Running 7 days ago
cvmf3j3bz8w6r4h0lf3pxo6eu   \_ s3.1  scality/s3server  localhost.localdomain.
↪localdomain  Shutdown       Failed 7 days ago   "task: non-zero exit (137)"
```

If the error is truncated it is possible to have a more detailed view of the error by inspecting the docker task ID:

```
$> docker inspect cvmf3j3bz8w6r4h0lf3pxo6eu
```

Off you go!

Let us know what you use this functionality for, and if you'd like any specific developments around it. Or, even better: come and contribute to our Github repository! We look forward to meeting you!

S3FS

Export your buckets as a filesystem with s3fs on top of s3server

s3fs is an open source tool that allows you to mount an S3 bucket on a filesystem-like backend. It is available both on Debian and RedHat distributions. For this tutorial, we used an Ubuntu 14.04 host to deploy and use s3fs over Scality's S3 Server.

Deploying S3 Server with SSL

First, you need to deploy **S3 Server**. This can be done very easily via our DockerHub page (you want to run it with a file backend).

> *Note: - If you don't have docker installed on your machine, here are the ‘instructions to install it for your distribution <https://docs.docker.com/engine/installation/>‘__*

You also necessarily have to set up SSL with S3Server to use s3fs. We have a nice tutorial to help you do it.

s3fs setup

Installing s3fs

s3fs has quite a few dependencies. As explained in their README, the following commands should install everything for Ubuntu 14.04:

```
$> sudo apt-get install automake autotools-dev g++ git libcurl4-gnutls-dev
$> sudo apt-get install libfuse-dev libssl-dev libxml2-dev make pkg-config
```

Now you want to install s3fs per se:

```
$> git clone https://github.com/s3fs-fuse/s3fs-fuse.git
$> cd s3fs-fuse
$> ./autogen.sh
$> ./configure
$> make
$> sudo make install
```

Check that s3fs is properly installed by checking its version. it should answer as below:

```
 $> s3fs --version

Amazon Simple Storage Service File System V1.80(commit:d40da2c) with OpenSSL
```

Configuring s3fs

s3fs expects you to provide it with a password file. Our file is `/etc/passwd-s3fs`. The structure for this file is `ACCESSKEYID:SECRETKEYID`, so, for S3Server, you can run:

```
$> echo 'accessKey1:verySecretKey1' > /etc/passwd-s3fs
$> chmod 600 /etc/passwd-s3fs
```

Using S3Server with s3fs

First, you're going to need a mountpoint; we chose `/mnt/tests3fs`:

```
$> mkdir /mnt/tests3fs
```

Then, you want to create a bucket on your local S3Server; we named it `tests3fs`:

```
$> s3cmd mb s3://tests3fs

*Note:* *- If you've never used s3cmd with our S3Server, our README
provides you with a `recommended
config <https://github.com/scality/S3/blob/master/README.md#s3cmd>`__*
```

Now you can mount your bucket to your mountpoint with s3fs:

```
$> s3fs tests3fs /mnt/tests3fs -o passwd_file=/etc/passwd-s3fs -o url="https://s3.
↪scality.test:8000/" -o use_path_request_style

*If you're curious, the structure of this command is*
``s3fs BUCKET_NAME PATH/TO/MOUNTPOINT -o OPTIONS``\ *, and the
options are mandatory and serve the following purposes:
* ``passwd_file``\ *: specifiy path to password file;
* ``url``\ *: specify the hostname used by your SSL provider;
* ``use_path_request_style``\ *: force path style (by default, s3fs
uses subdomains (DNS style)).*
```

From now on, you can either add files to your mountpoint, or add objects to your bucket, and they'll show in the other. For example, let's‘ create two files, and then a directory with a file in our mountpoint:

```
$> touch /mnt/tests3fs/file1 /mnt/tests3fs/file2
$> mkdir /mnt/tests3fs/dir1
$> touch /mnt/tests3fs/dir1/file3
```

Now, I can use s3cmd to show me what is actually in S3Server:

```
$> s3cmd ls -r s3://tests3fs

2017-02-28 17:28         0   s3://tests3fs/dir1/
2017-02-28 17:29         0   s3://tests3fs/dir1/file3
2017-02-28 17:28         0   s3://tests3fs/file1
2017-02-28 17:28         0   s3://tests3fs/file2
```

Now you can enjoy a filesystem view on your local S3Server!

Duplicity

How to backup your files with S3 Server.

Installing

Installing Duplicity and its dependencies

Second, you want to install Duplicity. You have to download this tarball, decompress it, and then checkout the README inside, which will give you a list of dependencies to install. If you're using Ubuntu 14.04, this is your lucky day: here is a lazy step by step install.

```
$> apt-get install librsync-dev gnupg
$> apt-get install python-dev python-pip python-lockfile
$> pip install -U boto
```

Then you want to actually install Duplicity:

```
$> tar zxvf duplicity-0.7.11.tar.gz
$> cd duplicity-0.7.11
$> python setup.py install
```

Using

Testing your installation

First, we're just going to quickly check that S3 Server is actually running. To do so, simply run `$> docker ps .` You should see one container named `scality/s3server`. If that is not the case, try `$> docker start s3server`, and check again.

Secondly, as you probably know, Duplicity uses a module called **Boto** to send requests to S3. Boto requires a configuration file located in **‘‘/etc/boto.cfg‘‘** to have your credentials and preferences. Here is a minimalistic config that you can finetune following these instructions.

```
[Credentials]
aws_access_key_id = accessKey1
aws_secret_access_key = verySecretKey1

[Boto]
# If using SSL, set to True
is_secure = False
# If using SSL, unmute and provide absolute path to local CA certificate
# ca_certificates_file = /absolute/path/to/ca.crt
```

```
*Note:* *If you want to set up SSL with S3 Server, check out our
`tutorial <http://link/to/SSL/tutorial>`__*
```

At this point, we've met all the requirements to start running S3 Server as a backend to Duplicity. So we should be able to back up a local folder/file to local S3. Let's try with the duplicity decompressed folder:

```
$> duplicity duplicity-0.7.11 "s3://127.0.0.1:8000/testbucket/"

*Note:* *Duplicity will prompt you for a symmetric encryption
passphrase. Save it somewhere as you will need it to recover your
data. Alternatively, you can also add the ``--no-encryption`` flag
and the data will be stored plain.*
```

If this command is succesful, you will get an output looking like this:

```
--------------[ Backup Statistics ]--------------
StartTime 1486486547.13 (Tue Feb  7 16:55:47 2017)
EndTime 1486486547.40 (Tue Feb  7 16:55:47 2017)
ElapsedTime 0.27 (0.27 seconds)
SourceFiles 388
SourceFileSize 6634529 (6.33 MB)
NewFiles 388
NewFileSize 6634529 (6.33 MB)
DeletedFiles 0
ChangedFiles 0
ChangedFileSize 0 (0 bytes)
ChangedDeltaSize 0 (0 bytes)
DeltaEntries 388
RawDeltaSize 6392865 (6.10 MB)
TotalDestinationSizeChange 2003677 (1.91 MB)
Errors 0
-------------------------------------------------
```

Congratulations! You can now backup to your local S3 through duplicity :)

Automating backups

Now you probably want to back up your files periodically. The easiest way to do this is to write a bash script and add it to your crontab. Here is my suggestion for such a file:

```
#!/bin/bash

# Export your passphrase so you don't have to type anything
export PASSPHRASE="mypassphrase"

# If you want to use a GPG Key, put it here and unmute the line below
#GPG_KEY=

# Define your backup bucket, with localhost specified
DEST="s3://127.0.0.1:8000/testbuckets3server/"

# Define the absolute path to the folder you want to backup
SOURCE=/root/testfolder

# Set to "full" for full backups, and "incremental" for incremental backups
# Warning: you have to perform one full backup befor you can perform
# incremental ones on top of it
FULL=incremental

# How long to keep backups for; if you don't want to delete old
# backups, keep empty; otherwise, syntax is "1Y" for one year, "1M"
# for one month, "1D" for one day
OLDER_THAN="1Y"

# is_running checks whether duplicity is currently completing a task
is_running=$(ps -ef | grep duplicity  | grep python | wc -l)

# If duplicity is already completing a task, this will simply not run
if [ $is_running -eq 0 ]; then
    echo "Backup for ${SOURCE} started"

    # If you want to delete backups older than a certain time, we do it here
    if [ "$OLDER_THAN" != "" ]; then
        echo "Removing backups older than ${OLDER_THAN}"
        duplicity remove-older-than ${OLDER_THAN} ${DEST}
    fi

    # This is where the actual backup takes place
    echo "Backing up ${SOURCE}..."
    duplicity ${FULL} \
        ${SOURCE} ${DEST}
        # If you're using GPG, paste this in the command above
        # --encrypt-key=${GPG_KEY} --sign-key=${GPG_KEY} \
        # If you want to exclude a subfolder/file, put it below and
        # paste this
        # in the command above
        # --exclude=/${SOURCE}/path_to_exclude \

    echo "Backup for ${SOURCE} complete"
    echo "------------------------------------"
fi
# Forget the passphrase...
unset PASSPHRASE
```

So let's say you put this file in `/usr/local/sbin/backup.sh.` Next you want to run `crontab -e` and paste your configuration in the file that opens. If you're unfamiliar with Cron, here is a good How To. The folder I'm backing up is a folder I modify permanently during my workday, so I want incremental backups every 5mn from 8AM to 9PM monday to friday. Here is the line I will paste in my crontab:

Now I can try and add / remove files from the folder I'm backing up, and I will see incremental backups in my bucket.

CHAPTER 6

Architecture

Versioning

This document describes S3 Server's support for the AWS S3 Bucket Versioning feature.

AWS S3 Bucket Versioning

See AWS documentation for a description of the Bucket Versioning feature:

- Bucket Versioning
- Object Versioning

This document assumes familiarity with the details of Bucket Versioning, including null versions and delete markers, described in the above links.

Implementation of Bucket Versioning in S3

Overview of Metadata and API Component Roles

Each version of an object is stored as a separate key in metadata. The S3 API interacts with the metadata backend to store, retrieve, and delete version metadata.

The implementation of versioning within the metadata backend is naive. The metadata backend does not evaluate any information about bucket or version state (whether versioning is enabled or suspended, and whether a version is a null version or delete marker). The S3 front-end API manages the logic regarding versioning information, and sends instructions to metadata to handle the basic CRUD operations for version metadata.

The role of the S3 API can be broken down into the following:

- put and delete version data
- store extra information about a version, such as whether it is a delete marker or null version, in the object's metadata

- send instructions to metadata backend to store, retrieve, update and delete version metadata based on bucket versioning state and version metadata
- encode version ID information to return in responses to requests, and decode version IDs sent in requests

The implementation of Bucket Versioning in S3 is described in this document in two main parts. The first section, *"Implementation of Bucket Versioning in Metadata"*, describes the way versions are stored in metadata, and the metadata options for manipulating version metadata.

The second section, *"Implementation of Bucket Versioning in API"*, describes the way the metadata options are used in the API within S3 actions to create new versions, update their metadata, and delete them. The management of null versions and creation of delete markers are also described in this section.

Implementation of Bucket Versioning in Metadata

As mentioned above, each version of an object is stored as a separate key in metadata. We use version identifiers as the suffix for the keys of the object versions, and a special version (the *"Master Version"*) to represent the latest version.

An example of what the metadata keys might look like for an object `foo/bar` with three versions (with . representing a null character):

key
foo/bar
foo/bar.098506163554375999999PARIS 0.a430a1f85c6ec
foo/bar.098506163554373999999PARIS 0.41b510cd0fdf8
foo/bar.098506163554373999998PARIS 0.f9b82c166f695

The most recent version created is represented above in the key `foo/bar` and is the master version. This special version is described further in the section *"Master Version"*.

Version ID and Metadata Key Format

The version ID is generated by the metadata backend, and encoded in a hexadecimal string format by S3 before sending a response to a request. S3 also decodes the hexadecimal string received from a request before sending to metadata to retrieve a particular version.

The format of a `version_id` is: `ts` `rep_group_id` `seq_id` where:

- `ts`: is the combination of epoch and an increasing number
- `rep_group_id`: is the name of deployment(s) considered one unit used for replication
- `seq_id`: is a unique value based on metadata information.

The format of a key in metadata for a version is:

`object_name separator version_id` where:

- `object_name`: is the key of the object in metadata
- `separator`: we use the `null` character (`0x00` or `\0`) as the separator between the `object_name` and the `version_id` of a key
- `version_id`: is the version identifier; this encodes the ordering information in the format described above as metadata orders keys alphabetically

An example of a key in metadata: `foo\01234567890000777PARIS 1234.123456` indicating that this specific version of `foo` was the `000777`th entry created during the epoch `1234567890` in the replication group `PARIS` with `1234.123456` as `seq_id`.

Master Version

We store a copy of the latest version of an object's metadata using `object_name` as the key; this version is called the master version. The master version of each object facilitates the standard GET operation, which would otherwise need to scan among the list of versions of an object for its latest version.

The following table shows the layout of all versions of `foo` in the first example stored in the metadata (with dot `.` representing the null separator):

key	value
foo	B
foo.v2	B
foo.v1	A

Metadata Versioning Options

S3 Server sends instructions to the metadata engine about whether to create a new version or overwrite, retrieve, or delete a specific version by sending values for special options in PUT, GET, or DELETE calls to metadata. The metadata engine can also list versions in the database, which is used by S3 to list object versions.

These only describe the basic CRUD operations that the metadata engine can handle. How these options are used by the S3 API to generate and update versions is described more comprehensively in *"Implementation of Bucket Versioning in API"*.

Note: all operations (PUT and DELETE) that generate a new version of an object will return the `version_id` of the new version to the API.

PUT

- no options: original PUT operation, will update the master version
- `versioning: true` create a new version of the object, then update the master version with this version.
- `versionId: <versionId>` create or update a specific version (for updating version's ACL or tags, or remote updates in geo-replication) - if the version identified by `versionId` happens to be the latest

  version, the master version will be updated as well

  - if the master version is not as recent as the version identified by `versionId`, as may happen with cross-region replication, the master will be updated as well
  - note that with `versionId` set to an empty string `''`, it will overwrite the master version only (same as no options, but the master version will have a `versionId` property set in its metadata like any other version). The `versionId` will never be exposed to an external user, but setting this internal-only `versionID` enables S3 to find this version later if it is no longer the master. This option of `versionId` set to `''` is used for creating null versions once versioning has been suspended, which is discussed in *"Null Version Management"*.

In general, only one option is used at a time. When `versionId` and `versioning` are both set, only the `versionId` option will have an effect.

DELETE

- no options: original DELETE operation, will delete the master version
- `versionId: <versionId>` delete a specific version

A deletion targeting the latest version of an object has to:

- delete the specified version identified by `versionId`
- replace the master version with a version that is a placeholder for deletion
  - **this version contains a special keyword, 'isPHD', to indicate the** master version was deleted and needs to be updated
- initiate a repair operation to update the value of the master version:
  - involves listing the versions of the object and get the latest version to replace the placeholder delete version
  - if no more versions exist, metadata deletes the master version, removing the key from metadata

Note: all of this happens before responding to S3, and only when the metadata engine is instructed by S3 to delete a specific version or the master version. See section *"Delete Markers"* for a description of what happens when a Delete Object request is sent to the S3 API.

GET

- no options: original GET operation, will get the master version
- `versionId: <versionId>` retrieve a specific version

The implementation of a GET operation does not change compared to the standard version. A standard GET without versioning information would get the master version of a key. A version-specific GET would retrieve the specific version identified by the key for that version.

LIST

For a standard LIST on a bucket, metadata iterates through the keys by using the separator (`\0`, represented by `.` in examples) as an extra delimiter. For a listing of all versions of a bucket, there is no change compared to the original listing function. Instead, the API component returns all the keys in a List Objects call and filters for just the keys of the master versions in a List Object Versions call.

For example, a standard LIST operation against the keys in a table below would return from metadata the list of `[ foo/bar, bar, qux/quz, quz ]`.

key
foo/bar
foo/bar.v2
foo/bar.v1
bar
qux/quz
qux/quz.v2
qux/quz.v1
quz
quz.v2
quz.v1

Implementation of Bucket Versioning in API

Object Metadata Versioning Attributes

To access all the information needed to properly handle all cases that may exist in versioned operations, the API stores certain versioning-related information in the metadata attributes of each version's object metadata.

These are the versioning-related metadata properties:

- `isNull`: whether the version being stored is a null version.
- `nullVersionId`: the unencoded version ID of the latest null version that existed before storing a non-null version.
- `isDeleteMarker`: whether the version being stored is a delete marker.

The metadata engine also sets one additional metadata property when creating the version.

- `versionId`: the unencoded version ID of the version being stored.

Null versions and delete markers are described in further detail in their own subsections.

Creation of New Versions

When versioning is enabled in a bucket, APIs which normally result in the creation of objects, such as Put Object, Complete Multipart Upload and Copy Object, will generate new versions of objects.

S3 creates a new version and updates the master version using the `versioning:  true` option in PUT calls to the metadata engine. As an example, when two consecutive Put Object requests are sent to the S3 Server for a versioning-enabled bucket with the same key names, there are two corresponding metadata PUT calls with the `versioning` option set to true.

The PUT calls to metadata and resulting keys are shown below:

1. PUT foo (first put), versioning: `true`

key	value
foo	A
foo.v1	A

2. PUT foo (second put), versioning: `true`

key	value
foo	B
foo.v2	B
foo.v1	A

Null Version Management

In a bucket without versioning, or when versioning is suspended, putting an object with the same name twice should result in the previous object being overwritten. This is managed with null versions.

Only one null version should exist at any given time, and it is identified in S3 requests and responses with the version id “null”.

Case 1: Putting Null Versions

With respect to metadata, since the null version is overwritten by subsequent null versions, the null version is initially stored in the master key alone, as opposed to being stored in the master key and a new version. S3 checks if versioning is suspended or has never been configured, and sets the `versionId` option to `''` in PUT calls to the metadata engine when creating a new null version.

If the master version is a null version, S3 also sends a DELETE call to metadata prior to the PUT, in order to clean up any pre-existing null versions which may, in certain edge cases, have been stored as a separate version.[1]

The tables below summarize the calls to metadata and the resulting keys if we put an object 'foo' twice, when versioning has not been enabled or is suspended.

1. PUT foo (first put), versionId: `''`

key	value
foo (null)	A

(2A) DELETE foo (clean-up delete before second put), versionId: `<version id of master version>`

key	value

(2B) PUT foo (second put), versionId: `''`

key	value
foo (null)	B

The S3 API also sets the `isNull` attribute to `true` in the version metadata before storing the metadata for these null versions.

Case 2: Preserving Existing Null Versions in Versioning-Enabled Bucket

Null versions are preserved when new non-null versions are created after versioning has been enabled or re-enabled.

If the master version is the null version, the S3 API preserves the current null version by storing it as a new key `(3A)` in a separate PUT call to metadata, prior to overwriting the master version `(3B)`. This implies the null version may not necessarily be the latest or master version.

To determine whether the master version is a null version, the S3 API checks if the master version's `isNull` property is set to `true`, or if the `versionId` attribute of the master version is undefined (indicating it is a null version that was put before bucket versioning was configured).

Continuing the example from Case 1, if we enabled versioning and put another object, the calls to metadata and resulting keys would resemble the following:

(3A) PUT foo, versionId: `<versionId of master version>` if defined or `<non-versioned object id>`

key	value
foo	B
foo.v1 (null)	B

(3B) PUT foo, versioning: `true`

key	value
foo	C
foo.v2	C
foo.v1 (null)	B

[1] Some examples of these cases are: (1) when there is a null version that is the second-to-latest version, and the latest version has been deleted, causing metadata to repair the master value with the value of the null version and (2) when putting object tag or ACL on a null version that is the master version, as explained in *"Behavior of Object-Targeting APIs"*.

To prevent issues with concurrent requests, S3 ensures the null version is stored with the same version ID by using `versionId` option. S3 sets the `versionId` option to the master version's `versionId` metadata attribute value during the PUT. This creates a new version with the same version ID of the existing null master version.

The null version's `versionId` attribute may be undefined because it was generated before the bucket versioning was configured. In that case, a version ID is generated using the max epoch and sequence values possible so that the null version will be properly ordered as the last entry in a metadata listing. This value ("non-versioned object id") is used in the PUT call with the `versionId` option.

Case 3: Overwriting a Null Version That is Not Latest Version

Normally when versioning is suspended, S3 uses the `versionId: ''` option in a PUT to metadata to create a null version. This also overwrites an existing null version if it is the master version.

However, if there is a null version that is not the latest version, S3 cannot rely on the `versionId: ''` option will not overwrite the existing null version. Instead, before creating a new null version, the S3 API must send a separate DELETE call to metadata specifying the version id of the current null version for delete.

To do this, when storing a null version (3A above) before storing a new non-null version, S3 records the version's ID in the `nullVersionId` attribute of the non-null version. For steps 3A and 3B above, these are the values stored in the `nullVersionId` of each version's metadata:

(3A) PUT foo, versioning: `true`

key	value	value.nullVersionId
foo	B	undefined
foo.v1 (null)	B	undefined

(3B) PUT foo, versioning: `true`

key	value	value.nullVersionId
foo	C	v1
foo.v2	C	v1
foo.v1 (null)	B	undefined

If defined, the `nullVersionId` of the master version is used with the `versionId` option in a DELETE call to metadata if a Put Object request is received when versioning is suspended in a bucket.

(4A) DELETE foo, versionId: `<nullVersionId of master version>` (v1)

key	value
foo	C
foo.v2	C

Then the master version is overwritten with the new null version:

(4B) PUT foo, versionId: `''`

key	value
foo (null)	D
foo.v2	C

The `nullVersionId` attribute is also used to retrieve the correct version when the version ID "null" is specified in certain object-level APIs, described further in the section *"Null Version Mapping"*.

Specifying Versions in APIs for Putting Versions

Since S3 does not allow an overwrite of existing version data, Put Object, Complete Multipart Upload and Copy Object return `400 InvalidArgument` if a specific version ID is specified in the request query, e.g. for a `PUT`

`/foo?versionId=v1` request.

PUT Example

When S3 receives a request to PUT an object:

- It checks first if versioning has been configured
- If it has not been configured, S3 proceeds to puts the new data, puts the metadata by overwriting the master version, and proceeds to delete any pre-existing data

If versioning has been configured, S3 checks the following:

Versioning Enabled

If versioning is enabled and there is existing object metadata:

- If the master version is a null version (`isNull: true`) or has no version ID (put before versioning was configured):
  - store the null version metadata as a new version
  - create a new version and overwrite the master version
    - set `nullVersionId`: version ID of the null version that was stored

If versioning is enabled and the master version is not null; or there is no existing object metadata:

- create a new version and store it, and overwrite the master version

Versioning Suspended

If versioning is suspended and there is existing object metadata:

- If the master version has no version ID:
  - overwrite the master version with the new metadata (PUT `versionId: ''`)
  - delete previous object data
- If the master version is a null version:
  - delete the null version using the *versionId* metadata attribute of the master version (PUT `versionId: <versionId of master object MD>`)
  - put a new null version (PUT `versionId: ''`)
- If master is not a null version and `nullVersionId` is defined in the object's metadata:
  - delete the current null version metadata and data
  - overwrite the master version with the new metadata

If there is no existing object metadata, create the new null version as the master version.

In each of the above cases, set `isNull` metadata attribute to true when creating the new null version.

Behavior of Object-Targeting APIs

API methods which can target existing objects or versions, such as Get Object, Head Object, Get Object ACL, Put Object ACL, Copy Object and Copy Part, will perform the action on the latest version of an object if no version ID is specified in the request query or relevant request header (`x-amz-copy-source-version-id` for Copy Object and Copy Part APIs).

Two exceptions are the Delete Object and Multi-Object Delete APIs, which will instead attempt to create delete markers, described in the following section, if no version ID is specified.

No versioning options are necessary to retrieve the latest version from metadata, since the master version is stored in a key with the name of the object. However, when updating the latest version, such as with the Put Object ACL API, S3 sets the `versionId` option in the PUT call to metadata to the value stored in the object metadata's `versionId` attribute. This is done in order to update the metadata both in the master version and the version itself, if it is not a null version.[2]

When a version id is specified in the request query for these APIs, e.g. `GET /foo?versionId=v1`, S3 will attempt to decode the version ID and perform the action on the appropriate version. To do so, the API sets the value of the `versionId` option to the decoded version ID in the metadata call.

Delete Markers

If versioning has not been configured for a bucket, the Delete Object and Multi-Object Delete APIs behave as their standard APIs.

If versioning has been configured, S3 deletes object or version data only if a specific version ID is provided in the request query, e.g. `DELETE /foo?versionId=v1`.

If no version ID is provided, S3 creates a delete marker by creating a 0-byte version with the metadata attribute `isDeleteMarker: true`. The S3 API will return a `404 NoSuchKey` error in response to requests getting or heading an object whose latest version is a delete maker.

To restore a previous version as the latest version of an object, the delete marker must be deleted, by the same process as deleting any other version.

The response varies when targeting an object whose latest version is a delete marker for other object-level APIs that can target existing objects and versions, without specifying the version ID.

- Get Object, Head Object, Get Object ACL, Object Copy and Copy Part return `404 NoSuchKey`.
- Put Object ACL and Put Object Tagging return `405 MethodNotAllowed`.

These APIs respond to requests specifying the version ID of a delete marker with the error `405 MethodNotAllowed`, in general. Copy Part and Copy Object respond with `400 Invalid Request`.

See section *"Delete Example"* for a summary.

Null Version Mapping

When the null version is specified in a request with the version ID "null", the S3 API must use the `nullVersionId` stored in the latest version to retrieve the current null version, if the null version is not the latest version.

Thus, getting the null version is a two step process:

[2] If it is a null version, this call will overwrite the null version if it is stored in its own key (`foo\0<versionId>`). If the null version is stored only in the master version, this call will both overwrite the master version *and* create a new key (`foo\0<versionId>`), resulting in the edge case referred to by the previous footnote[1].

1. Get the latest version of the object from metadata. If the latest version's `isNull` property is `true`, then use the latest version's metadata. Otherwise,
2. Get the null version of the object from metadata, using the internal version ID of the current null version stored in the latest version's `nullVersionId` metadata attribute.

DELETE Example

The following steps are used in the delete logic for delete marker creation:

- If versioning has not been configured: attempt to delete the object
- If request is version-specific delete request: attempt to delete the version
- otherwise, if not a version-specific delete request and versioning has been configured:
  - create a new 0-byte content-length version
  - in version's metadata, set a 'isDeleteMarker' property to true
- Return the version ID of any version deleted or any delete marker created
- Set response header `x-amz-delete-marker` to true if a delete marker was deleted or created

The Multi-Object Delete API follows the same logic for each of the objects or versions listed in an xml request. Note that a delete request can result in the creation of a deletion marker even if the object requested to delete does not exist in the first place.

Object-level APIs which can target existing objects and versions perform the following checks regarding delete markers:

- If not a version-specific request and versioning has been configured, check the metadata of the latest version
- If the 'isDeleteMarker' property is set to true, return `404 NoSuchKey` or `405 MethodNotAllowed`
- If it is a version-specific request, check the object metadata of the requested version
- If the `isDeleteMarker` property is set to true, return `405 MethodNotAllowed` or `400 InvalidRequest`

Data-metadata daemon Architecture and Operational guide

This document presents the architecture of the data-metadata daemon (dmd) used for the community edition of S3 server. It also provides a guide on how to operate it.

The dmd is responsible for storing and retrieving S3 data and metadata, and is accessed by S3 connectors through socket.io (metadata) and REST (data) APIs.

It has been designed such that more than one S3 connector can access the same buckets by communicating with the dmd. It also means that the dmd can be hosted on a separate container or machine.

Operation

Startup

The simplest deployment is still to launch with npm start, this will start one instance of the S3 connector and will listen on the locally bound dmd ports 9990 and 9991 (by default, see below).

The dmd can be started independently from the S3 server by running this command in the S3 directory:

```
npm run start_dmd
```

This will open two ports:

- one is based on socket.io and is used for metadata transfers (9990 by

  default)
- the other is a REST interface used for data transfers (9991 by

  default)

Then, one or more instances of S3 server without the dmd can be started elsewhere with:

```
npm run start_s3server
```

Configuration

Most configuration happens in `config.json` for S3 server, local storage paths can be changed where the dmd is started using environment variables, like before: `S3DATAPATH` and `S3METADATAPATH`.

In `config.json`, the following sections are used to configure access to the dmd through separate configuration of the data and metadata access:

```
"metadataClient": {
    "host": "localhost",
    "port": 9990
},
"dataClient": {
    "host": "localhost",
    "port": 9991
},
```

To run a remote dmd, you have to do the following:

- change both `"host"` attributes to the IP or host name where the

  dmd is run.
- Modify the `"bindAddress"` attributes in `"metadataDaemon"` and

  `"dataDaemon"` sections where the dmd is run to accept remote connections (e.g. `"::"`)

Architecture

This section gives a bit more insight on how it works internally.

Metadata on socket.io

This communication is based on an RPC system based on socket.io events sent by S3 connectors, received by the DMD and acknowledged back to the S3 connector.

The actual payload sent through socket.io is a JSON-serialized form of the RPC call name and parameters, along with some additional information like the request UIDs, and the sub-level information, sent as object attributes in the JSON request.

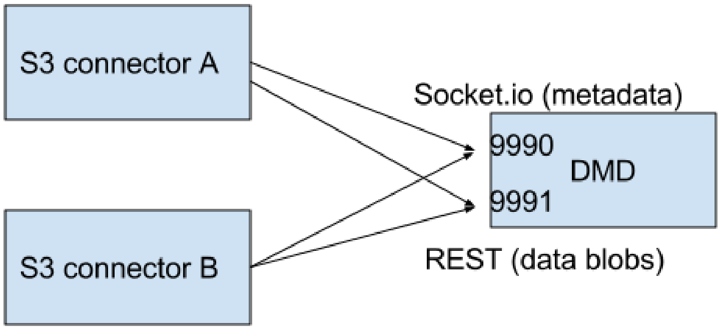

Fig. 6.1: ./images/data_metadata_daemon_arch.png

With introduction of versioning support, the updates are now gathered in the dmd for some number of milliseconds max, before being batched as a single write to the database. This is done server-side, so the API is meant to send individual updates.

Four RPC commands are available to clients: `put`, `get`, `del` and `createReadStream`. They more or less map the parameters accepted by the corresponding calls in the LevelUp implementation of LevelDB. They differ in the following:

- The `sync` option is ignored (under the hood, puts are gathered

  into batches which have their `sync` property enforced when they are committed to the storage)
- Some additional versioning-specific options are supported
- `createReadStream` becomes asynchronous, takes an additional

  callback argument and returns the stream in the second callback parameter

Debugging the socket.io exchanges can be achieved by running the daemon with `DEBUG='socket.io*'` environment variable set.

One parameter controls the timeout value after which RPC commands sent end with a timeout error, it can be changed either:

- via the `DEFAULT_CALL_TIMEOUT_MS` option in

  `lib/network/rpc/rpc.js`
- or in the constructor call of the `MetadataFileClient` object (in

  `lib/metadata/bucketfile/backend.js` as `callTimeoutMs`.

Default value is 30000.

A specific implementation deals with streams, currently used for listing a bucket. Streams emit `"stream-data"` events that pack one or more items in the listing, and a special `“stream-end”` event when done. Flow control is achieved by allowing a certain number of "in flight" packets that have not received an ack yet (5 by default). Two options can tune the behavior (for better throughput or getting it more robust on weak networks), they have to be set in `mdserver.js` file directly, as there is no support in `config.json` for now for those options:

- `streamMaxPendingAck`: max number of pending ack events not yet

  received (default is 5)
- `streamAckTimeoutMs`: timeout for receiving an ack after an output

  stream packet is sent to the client (default is 5000)

Data exchange through the REST data port

Data is read and written with REST semantic.

The web server recognizes a base path in the URL of `/DataFile` to be a request to the data storage service.

PUT

A PUT on `/DataFile` URL and contents passed in the request body will write a new object to the storage.

On success, a `201 Created` response is returned and the new URL to the object is returned via the `Location` header (e.g. `Location: /DataFile/50165db76eecea293abfd31103746dadb73a2074`). The raw key can then be extracted simply by removing the leading `/DataFile` service information from the returned URL.

GET

A GET is simply issued with REST semantic, e.g.:

```
GET /DataFile/50165db76eecea293abfd31103746dadb73a2074 HTTP/1.1
```

A GET request can ask for a specific range. Range support is complete except for multiple byte ranges.

DELETE

DELETE is similar to GET, except that a `204 No Content` response is returned on success.

Listing

Listing Types

We use three different types of metadata listing for various operations. Here are the scenarios we use each for:

- 'Delimiter' - when no versions are possible in the bucket since it is

  an internally-used only bucket which is not exposed to a user. Namely,

1. to list objects in the "user's bucket" to respond to a GET SERVICE

   request and

2. to do internal listings on an MPU shadow bucket to complete multipart

   upload operations.

- 'DelimiterVersion' - to list all versions in a bucket
- 'DelimiterMaster' - to list just the master versions of objects in a

  bucket

Algorithms

The algorithms for each listing type can be found in the open-source scality/Arsenal repository, in lib/algos/list.