Geo replication and disaster recovery for cloud object storage with Ceph rados gateway

Orit Wasserman
Senior Software engineer
owasserman@redhat.com
Linuxcon EU 2016
AGENDA

• What is Ceph?
• Rados Gateway (radosgw) architecture
• Geo replication in radosgw
• Questions
Ceph architecture
Cephalopod

A cephalopod is any member of the molluscan class Cephalopoda. These exclusively marine animals are characterized by bilateral body symmetry, a prominent head, and a set of arms or tentacles (muscular hydrostats) modified from the primitive molluscan foot. The study of cephalopods is a branch of malacology known as teuthology.
Ceph is a distributed object, block, and file storage platform [http://ceph.com](http://ceph.com).
Ceph

• Open source
• Software defined storage
• Distributed
• No single point of failure
• Massively scalable
• Self healing
• Unified storage: object, block and file
• IRC: OFTC #ceph,#ceph-devel
• Mailing lists:
  • ceph-users@ceph.com
  • ceph-devel@ceph.com
Ceph architecture

APP
  RGW
  A web services gateway for object storage, compatible with S3 and Swift

HOST/VM
  RBD
  A reliable, fully-distributed block device with cloud platform integration

CLIENT
  CEPHFS
  A distributed file system with POSIX semantics and scale-out metadata management

LIBRADOS
  A library allowing apps to directly access RADOS (C, C++, Java, Python, Ruby, PHP)

RADOS
  A software-based, reliable, autonomous, distributed object store comprised of self-healing, self-managing, intelligent storage nodes and lightweight monitors
Rados

• Reliable Distributed Object Storage
• Replication
• Erasure coding
• Flat object namespace within each pool
  • Different placement rules
• Strong consistency (CP system)
• Infrastructure aware, dynamic topology
• Hash-based placement (CRUSH)
• Direct client to server data path
OSD node

- 10s to 10000s in a cluster
- One per disk (or one per SSD, RAID group...)
- Serve stored objects to clients
- Intelligently peer for replication & recovery
Monitor node

- Maintain cluster membership and state
- Provide consensus for distributed decision-making
- Small, odd number
- These do not serve stored objects to clients
object placement

pool

hash(object name) % num_pg = pg

placement group (PG)

CRUSH(pg, cluster state, rule) = [A, B]
Crush

- pseudo-random placement algorithm
  - fast calculation, **no lookup**
  - repeatable, deterministic
- statistically uniform distribution
- stable mapping
  - limited data migration on change
- rule-based configuration
  - infrastructure topology aware
  - adjustable replication
  - allows weighting
Librados API

- Efficient key/value storage inside an object
- Atomic single-object transactions
  - update data, attr, keys together
  - atomic compare-and-swap
- Object-granularity snapshot infrastructure
- Partial overwrite of existing data
- Single-object compound atomic operations
- RADOS classes (stored procedures)
- Watch/Notify on an object
Rados Gateway
Rados Gateway

**APP**
- **RGW**
  A web services gateway for object storage, compatible with S3 and Swift

**HOST/VM**
- **RBD**
  A reliable, fully-distributed block device with cloud platform integration

**CLIENT**
- **CEPHFS**
  A distributed file system with POSIX semantics and scale-out metadata management

**LIBRADOS**
A library allowing apps to directly access RADOS (C, C++, Java, Python, Ruby, PHP)

**RADOS**
A software-based, reliable, autonomous, distributed object store comprised of self-healing, self-managing, intelligent storage nodes and lightweight monitors
Rados Gateway

APPLICATION

RADOSGW
LIBRADOS

RADOS CLUSTER

APPLICATION

RADOSGW
LIBRADOS

REST

socket
RESTful OBJECT STORAGE

- Data
  - Users
  - Buckets
  - Objects
  - ACLs
- Authentication
- APIs
  - S3
  - Swift
  - Librgw (used for NFS)
RGW vs RADOS object

• RADOS
  • Limited object sizes
  • Mutable objects
  • Not indexed
  • No per-object ACLs

• RGW
  • Large objects (Up to a few TB per object)
  • Immutable objects
  • Sorted bucket listing
  • Permissions
RGW objects requirements

- Large objects
- Fast small object access
- Fast access to object attributes
- Buckets can consist of a very large number of objects
RGW objects

**OBJECT**

<table>
<thead>
<tr>
<th>HEAD</th>
<th>TAIL</th>
</tr>
</thead>
</table>

• **Head**
  • Single rados object
  • Object metadata (acls, user attributes, manifest)
  • Optional start of data

• **Tail**
  • Striped data
  • 0 or more rados objects
RGW Objects

OBJECT: foo
BUCKET: boo
BUCKET ID: 123

head

123_foo

tail 1

123_28faPd3Z.1

tail 1

123_28faPd3Z.2
RGW bucket index

BUCKET INDEX

Shard 1
- aaa
- abc
- def (v2)
- def (v1)
- zzz

Shard 2
- aab
- bbb
-eee
-fff
- zzz
RGW object creation

• When creating a new object we need to:
  • Update bucket index
  • Create head object
  • Create tail objects
• All those operations need to be consist
RGW object creation

- **Write tail**: TAIL
  - aab
  - bbb
  - eee
  - fff (prepare)
  - zzz

- **prepare**
  - HEAD

- **Write head**

- **complete**
  - aab
  - bbb
  - eee
  - fff
  - zzz
RGW metadata cache

notify

notification

RADOS CLUSTER
Geo replication
Geo replication

- Data is replicated on different physical locations
- High and unpredictable latency between those locations
- Used for disaster recovery
Geo replication
Sync agent (old implementation)
Sync agent (old implementation)

- External python implementation
- No Active/Active support
- Hard to configure
- Complicate failover mechanism
- No clear sync status indication
- A single bucket synchronization could dominate the entire sync process
- Configuration updates require restart of the gateways
New implementation

• part of the radosgw (written in c++)
• Active/active support for data replication
• Simpler configuration
• Simplify failover/failback
• Dynamic reconfiguration
• Backward compatibility with the sync agent
Multisite configuration

- Realm
  - Namespace
  - contains the multisite configuration and status
  - Allows running different configurations in the same cluster
- Zonegroup
  - Group of zones
  - Used to be called region in old multisite
  - Each realm has a single master zonegroup
- Zone
  - One or more Radosgw instances all running on the same Rados cluster
  - Each zonegroup has a single master zone
Multisite environment example

ZoneGroup: us (master)
Zone: us-east (master)

ZoneGroup: eu (secondary)
Zone: eu-west (master)

Realm: Gold
Configuration change

- **Period:**
  - Each period has a unique id
  - Contains: realm configuration, an epoch and it's predecessor period id (except for the first period)

- Every realm has an associated current period and a chronological list of periods

- **Git like mechanism:**
  - User configuration changes are stored locally
  - Configuration updated are stored in a stagging period (using radosgw-admin period update command)
  - Changes are applied only when the period is commited (using radosgw-admin period commit command)

- Each zone can pull the period information (using radosgw-admin period pull command)
Configuration change – new master zone

- Period commit will result in the following actions:
  - A new period is generated with a new period id and epoch of 1
  - Realm's current period is updated to point to the newly generated period id
  - Realm's epoch is incremented
  - New period is pushed to all other zones by the new master
- We use watch/notify on the realm rados object to detect changes and apply them on the local radosgw
Configuration change

- Period commit will only increment the period epoch.
- The new period information will be pushed to all other zones
- We use watch/notify on the realm rados object to detect changes on the local radosgw
Sync process

- Metadata changes:
  - Bucket ops (Create, Delete and enable/disable versioning)
  - Users ops
- Metadata changes have wide system effect
- Metadata changes are rare
- Data changes: all objects updates
- Data changes are frequent
Metadata sync

- Metadata changes are replicated synchronously across the realm
- Each realm has a single meta master, the master zone in the master zone group
- Only the meta master can execute metadata changes
- Separate log for metadata changes
- Each Ceph cluster has a local copy of the metadata log
- If the meta master is down, the user cannot perform metadata updates till a new meta master is assigned
Metadata sync

- updates to metadata originating from a different zone:
  - forwarded request to the meta master
  - update the metadata log
  - meta master perform the change
  - meta master pushes metadata updates to all the other zones
  - Each zone will pull the updated metadata log and apply changes locally

- All zones check periodically for metadata changes
Data sync

- Data changes are handled locally and replicated asynchronously (eventual consistency)
- Default is Active/Active sync
- User can configure a zone to be read only for Active/Passive
- We first complete a full sync and then continue doing an incremental sync
- Each bucket instance within each zone has a unique incremented version id that is used to keep track of changes on that specific bucket.
Data sync

• Data sync run periodically
• Init phase: fetch the list of all the bucket instances
• Sync Phase:
  • for each bucket
    • If bucket does not exist, fetch bucket and bucket instance metadata from meta master zone. Create new bucket
    • Sync bucket
    • Check to see if need to send updates to other zones
• Incremental sync keeps a bucket index position to continue from
Sync status

- Each zone keeps the metadata sync state against the meta master.
- Each zone keeps the data sync state where it is synced with regard to all its peers.
### Sync status command

```
radosgw-admin sync status
```

<table>
<thead>
<tr>
<th>Realm</th>
<th>Zone Group</th>
<th>Zone</th>
<th>Metadata Sync Status</th>
<th>Data Sync Source</th>
<th>Data Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>f94ab897-4c8e-4654-a699-f72dfd4774df (gold)</td>
<td>9bcecc3c-0334-4163-8fbb-5b8db0371b39 (us)</td>
<td>153a268f-dd61-4465-819c-e5b04ec4e701 (us-west)</td>
<td>syncing</td>
<td>018cad1e-ab7d-4553-acc4-de402cfddd19 (us-east)</td>
<td>syncing</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>full sync: 0/64 shards</td>
<td></td>
<td>full sync: 0/128 shards</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>metadata is caught up with master</td>
<td></td>
<td>incremental sync: 64/64 shards</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>incremental sync: 64/64 shards</td>
<td></td>
<td>data is caught up with source</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A little bit of the Implementation

- We use co-routines for asynchronous execution based on boost::asio::coroutine with our own stack class.
- See code here: https://github.com/ceph/ceph/blob/master/src/rgw/rgw_coroutine.h
- We use leases for locking
What's next
WHAT'S NEXT

- Log trimming – clean old logs
- Sync modules – framework that allows forwarding data (and metadata) to external tiers. This will allow external metadata search (via elasticsearch)
THANK YOU!

Email: owasserm@redhat.com

IRC: owasserm OFTC #ceph, #ceph-devel