SEAGATE

Ceph RGW on CORTX Motr backend

Andriy Tkachuk, Sining Wu, Basavaraj Kirunge

Feb 2022

AGENDA

01 Introduction

02 Dive into the code: data-path

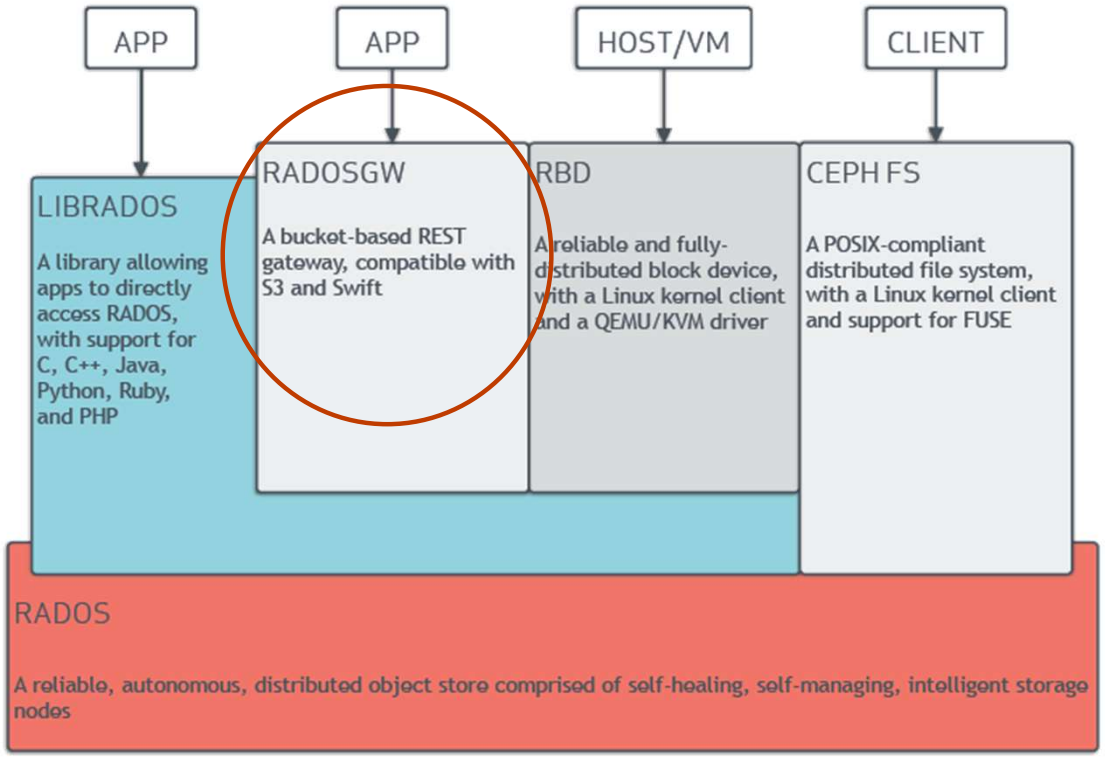
03 Latency and Concurrency

04 Lessons learned

05 Future plans, Q&A

Introduction – what is Ceph RGW?

From https://docs.ceph.com/en/latest/architecture/:

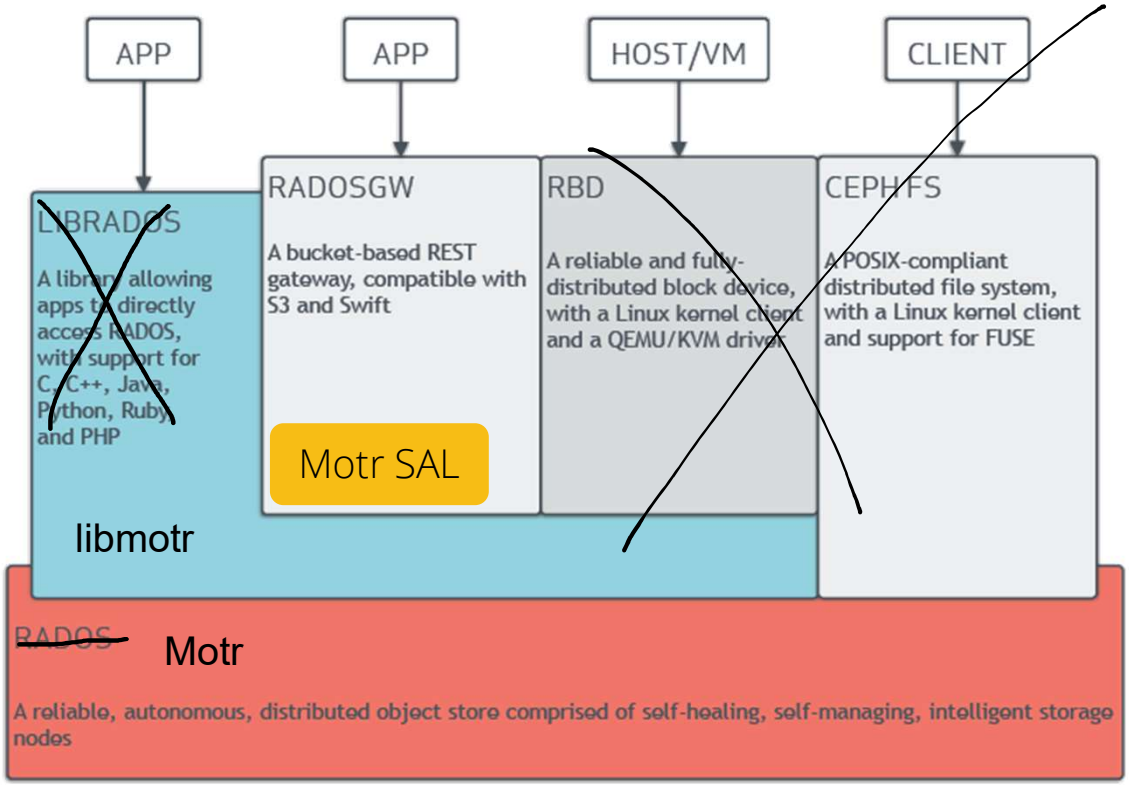


- S3 gateway to RADOS
- RGW == RADOSGW

Introduction – why not?..

- SAL - Storage Abstraction Layer
- Introduced recently at Zipper project

- MGW == Motr GW



Ok, let's dive into the code? – PUT data-path

- Motr objects can have different striping size (unit size) based on their total size
  - The stripping size is set on object creation and cannot change
  - Note: in RADOS the striping size is configured per pool, not per object
- For example, 64K object on 8+2 EC pool can have 64/8 == 8K striping units
  - and 1M object on the same pool can have 1M/8 == 128K units
- This allows to achieve an optimal storage utilisation and yet good performance for different object sizes
- So how do we learn the size of an object in Motr SAL?
  - Let's look at the API…

Dive into the code – PUT data-path

- There are two functions which control passing the data from RGW to Motr SAL:

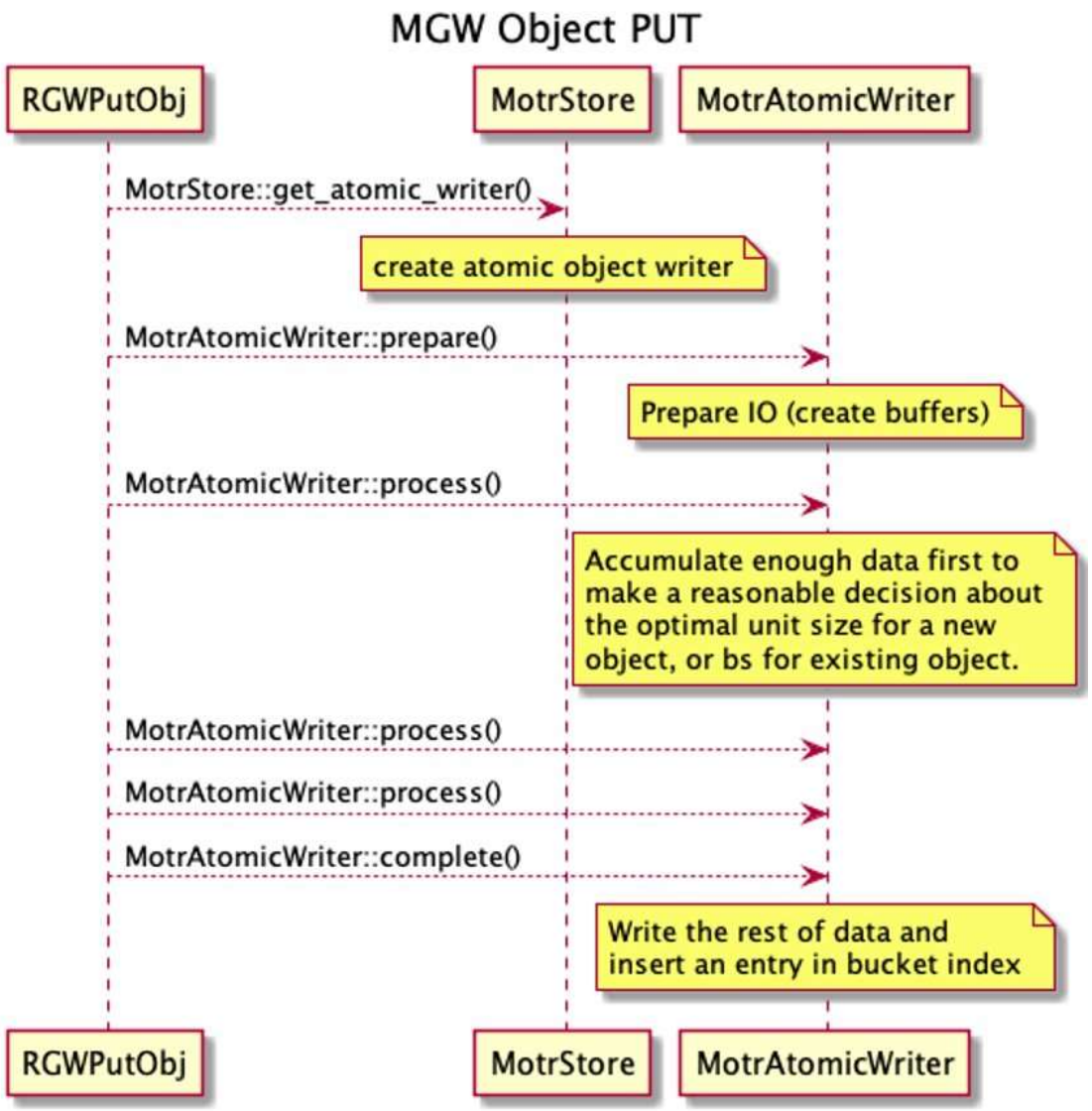
```
class Writer : public ObjectProcessor {

...

  /** Process a buffer.  Called multiple times to write different buffers. */
  virtual int process(bufferlist&& data, uint64_t offset) = 0;

  /** complete the operation and make its result visible to clients */
  virtual int complete(...) = 0;
```

- So we accumulate the data at process() (up to 32MB) and
- Create the object at complete() (or earlier, at process(), if object is bigger than 32MB)


MGW Object PUT
RGWPutObj
MotrStore
MotrAtomicWriter
MotrStore::get_atomic_writer()
create atomic object writer
MotrAtomicWriter::prepare()
Prepare IO (create buffers)
MotrAtomicWriter::process()
Accumulate enough data first to make a reasonable decision about the optimal unit size for a new object, or bs for existing object.
MotrAtomicWriter::process()
MotrAtomicWriter::process()
MotrAtomicWriter::complete()
Write the rest of data and insert an entry in bucket index
RGWPutObj
MotrStore
MotrAtomicWriter

RGW Data Layout

- **User info index** – stores all the user's buckets

Buckets	RGWBucketEnt
Bucket1	
Bucket2	

- **Bucket instance index** – stores all buckets' metadata

Buckets	MotrBucketInfo
Bucket1	

- **Bucket index** –
  stores bucket objects' metadata

Objects	Metadata
Object1	
Object2	

Object metadata:

- rgw_bucket_dir_entry
  - basic metadata that shows up when listing the bucket
- obj attrs
- Motr object specific info: obj_id, pver, layout_id

RGW object with RADOS backend

Bucket index (RADOS system object's omap)

objects	dir_entry
obj1	
obj2	

Header (Xattrs)

Chunk1 (4MB)

Chunk2 (4MB)

Chunk3 (4MB)

Chunk4 (4MB)

Metadata is stored in the Header

RGW object with Motr backend

Bucket index (Motr KV index)

objects	metadata
obj1	
obj2	

Metadata is stored in the Bucket index

Motr Object

Performance issues

- Although we got very good initial results for PUT, GET and TTFB (Time to First Byte) were very poor, especially with 64K objs
- As appeared, the problem was with the number of rgw worker threads configuration: the default value (512) is too high when there are many concurrent requests (like 20 reqs per node with 16 CPUs)
- The system went into thrashing

	Ceph: 3 replicas (24 drives)	Ceph: EC 4+2 (24 drives)	MGW: EC 4+2 (21 drives)
Object size, KiB	64	64	64
# of objects	7500	7500	7500
# of buckets	1	1	1
PUT avg MiB/sec	10.1	8.22	6.49
GET avg, MiB/sec	1395	877	4.33
TTFB Avg, ms	3	4	847
TTFB 99%, ms	6	9	5061

Performance issues

- Just setting rgw_thread_pool_size to 10 improved the result on more than 10 times!
- But we didn't stop on this…

	Ceph: 3 replicas (24 drives)	Ceph: EC 4+2 (24 drives)	MGW: EC 4+2 (21 drives)	MGW-Dec16 EC 4+2 (21 drives)
Object size, KiB	64	64	64	64
# of objects	7500	7500	7500	7500
# of buckets	1	1	1	1
PUT avg MiB/sec	10.1	8.22	6.49	18.4
GET avg, MiB/sec	1395	877	4.33	54
TTFB Avg, ms	3	4	847	69
TTFB 99%, ms	6	9	5061	136

Improving the GET Latency

- RGW does caching of S3 objects metadata (and even data up to rgw_max_chunk_size, 4MB by default – that's why they are so good at GET) and such data as user info and bucket info which are involved in the object lookup path
- So we tried to implement a simple time-based caching for user, bucket and object metadata – which gave us almost 3 times improvement!

Improving the GET Latency

	Ceph: 3 replicas (24 drives)	Ceph: EC 4+2 (24 drives)	MGW: EC 4+2 (21 drives)	MGW-Dec16 EC 4+2 (21 drives)	MGW-Dec29 EC 4+2 (21 drives)
Object size, KiB	64	64	64	64	64
# of objects	7500	7500	7500	7500	7500
# of buckets	1	1	1	1	1
PUT avg MiB/sec	10.1	8.22	6.49	18.4	17.7
GET avg, MiB/sec	1395	877	4.33	54	148
TTFB Avg, ms	3	4	847	69	25
TTFB 99%, ms	6	9	5061	136	53

Concurrency issues

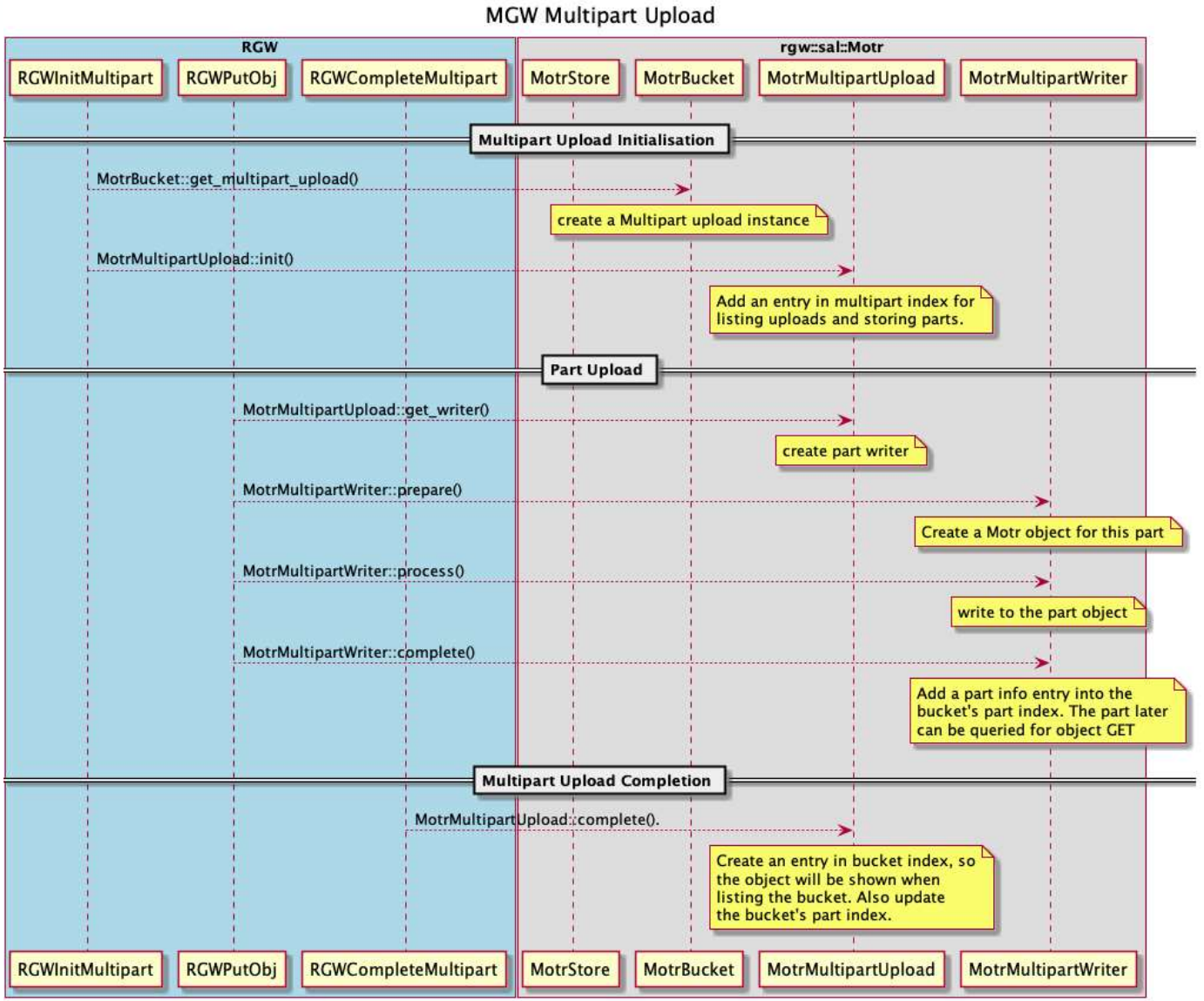
- How do we solve concurrency issues in distributed systems? What if two clients are writing to the same object at the same time?
- Distributed locks are very expensive, AWS is not using them also
- S3 protocol specifies that:
  - 1) object data should be consistent (not partial or mixed)
  - 2) last successful write is the one that remains
- In the early Motr SAL implementation each new S3 object had the same Motr object id (FID), that could lead to the violation of the 1st requirement (data consistency)

Concurrency issues – solution

- Always generate unique Motr object id (using Motr API) for any new S3 object
- Always add the new S3 object entry to the global GC queue in Motr KV (TBD)
- Write several Motr objects of the same S3 object concurrently to Motr
- Update bucket index – only the last successful one will remain
  - DTM makes sure the atomicity of this operation
- GC eventually will delete the data of all the previous S3 objects from Motr (TBD)

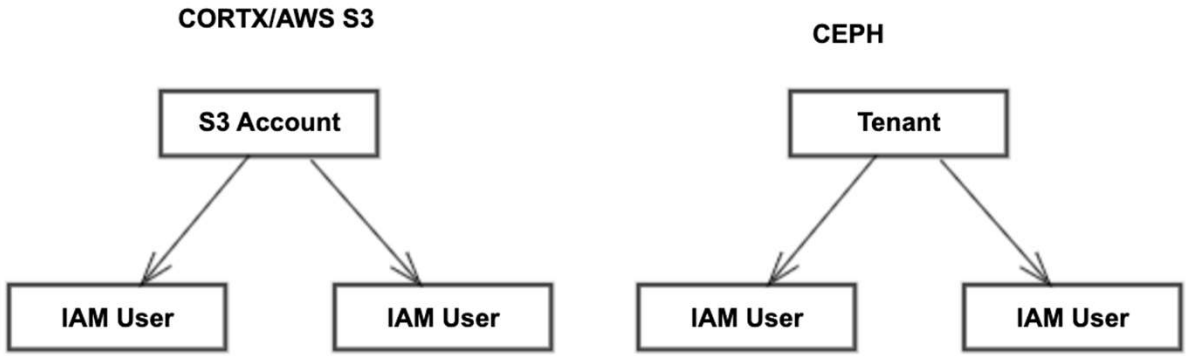
Multipart Upload

- Why multipart?
  - Because in case of a network failure only one part can be re-transmitted
  - You can upload the same object parts in parallel
  - You can start upload even before the total object size is known
- Implementation notes:
  - Each part is stored as a separate Motr object
  - Bucket multipart index for ongoing uploads contains metadata that answers questions such as which objects have started multipart upload and its upload id
  - Object parts index stores parts' details (size, oid, pvid,..) – similar to object metadata stored in the bucket index, but for parts


MGW Multipart Upload
RGW
rgw::sal::Motr
RGWInitMultipart
RGWPutObj
RGWCompleteMultipart
MotrStore
MotrBucket
MotrMultipartUpload
MotrMultipartWriter
Multipart Upload Initialisation
MotrBucket::get_multipart_upload()
create a Multipart upload instance
MotrMultipartUpload::init()
Add an entry in multipart index for listing uploads and storing parts.
Part Upload
MotrMultipartUpload::get_writer()
create part writer
MotrMultipartWriter::prepare()
Create a Motr object for this part
MotrMultipartWriter::process()
write to the part object
MotrMultipartWriter::complete()
Add a part info entry into the bucket's part index. The part later can be queried for object GET
Multipart Upload Completion
MotrMultipartUpload::complete().
Create an entry in bucket index, so the object will be shown when listing the bucket. Also update the bucket's part index.
RGWInitMultipart
RGWPutObj
RGWCompleteMultipart
MotrStore
MotrBucket
MotrMultipartUpload
MotrMultipartWriter

IAM* Differences

- No S3 Account in RGW
- Bucket owner is:
  - S3 Account in AWS
  - IAM User in RGW
- IAM Policy is not supported in RGW
- IAM API is not compliant with AWS

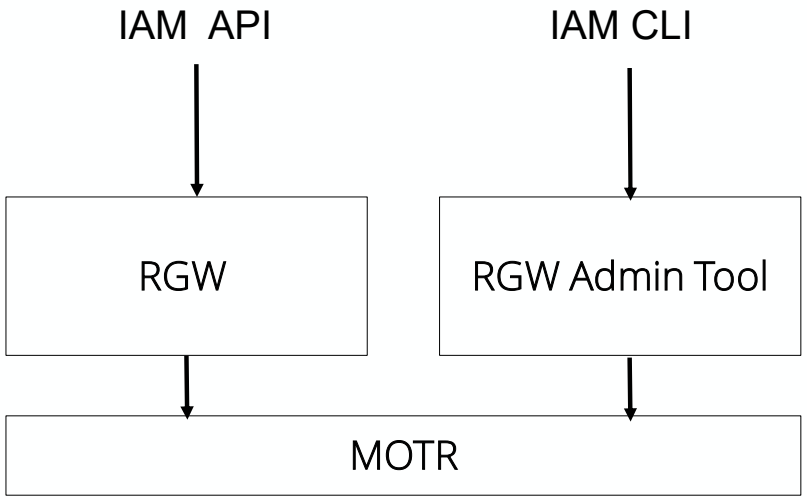


Status & Plans

- Add RGW user mgmt to MGW – in progress
- Discuss with Ceph community to add
  - S3 Account
  - IAM Policy
  - AWS Compliant IAM API

* IAM stands for Identity and Access Management

MGW Authentication – a bit more details



- RGW and Admin Tool to read auth data from Motr KV
- No changes are needed in authentication and authorisation steps, but in the storage of auth info only
- SAL API needs to be extended to store auth data in the backends

Auth Data:

- User Info
- Credentials
- Permissions/Policies
- Tenant/Account Info

What we've learned

- Being familiar with Motr client API was essential to implement PoC fast
- The RGW SAL API is still developing (it was changed several times even during our 1st PR was reviewed for landing!), but looks like it's in its final shaping stage
- You don't have to implement everything in RGW SAL API to make simple ops work
- It was relatively easy to implement also the basic versions of such advanced S3 features as: objects versioning and locking, multi-part I/O, objects' metadata caching
- Ceph community is quite friendly and welcoming

Future plans, Q&A

- Ceph RGW is going to be part of CORTX
- CORTX S3 developers to contribute back to Ceph RGW

- Now it's your time to speak! ☺

- Any further questions are welcome at our Slack channel

Thank you!