Handling Write Requests with OM HA

Introduction

This document discusses how to handle write requests received by OzoneManager HA so that all OM replicas arrive at consensus about mutations made to their respective states.

ShortHand Notations used in the document

1. OM DB. (Ozone Manager RocksDB)

2. OM (Ozone Manager)
3. startTransaction - which will be happening only on leader OM
4. applyTransaction - Applying transaction changes to OM. Which will be executed on leader and follower OM's.
5. Rpc Context - Client submit requests to OzoneManager server via RPC. RPC context means when client request is received at OzoneManagerServer End.
6. Request type - Considered as Write request.

When a write request is received in OM, the following steps are performed
1. First some pre-execution steps (e.g. allocateBlock, generate timestamp, sessionId, tokens, multipart uploadID)
2. Request validation (will request succeed like bucket exists, volume exists, key exists)
3. Write to Ratis log (get consensus and write to Raft Log)
4. Write to RocksDB (Persist to OM Rocks DB)
5. Execute request.
6. Send response (Response for the particular request)

All of these steps must be executed for each write request. However some of the steps can be reordered.

Late Validation

The 'late' in late validation means that the request execution is deferred until after consensus has been achieved via Raft.

1. Pre-execution steps like allocateBlock will be done when the RPC request is received.
2. In this approach, the request validation and execution will be done in the applyTransaction and once we get the response, immediately return the response to the client without applying to the OM DB.
3. In this approach :
    a. step 1 will be done at RPC level when the request is received, it can happen on any OM. (In OzoneManager HA we use Ratis Server API's to send requests, and if notLeaderException is thrown from Ratis, we use failOverProxyProvider in the ClientEnd to redirect the request to leader OM).
    b. Step 3 will be done once Ratis got quorum.
    c. Step 2,5,6 will be done in the applyTransaction. (We called this Late Validation because validation of the request is done in the applyTransaction instead of startTransaction).

Flow of the request

1. OM Receives a request for example let's take createBucket.

2. The creation time for the bucket will be set at RPC Level when the request is received, so that all OMs have the same data. (This should be done for correctness)
3. The request is written to Ratis log.
4. If consensus is achieved, then request will be sent to applyTransaction where it is divided into 2 steps:
    a. Validation of the request E.g. volume/bucket exist and create BucketInfo structure which we need to apply to OM DB. This validation may need to consult the OM DB as well as the request cache.
    b. Request and response details will be added to an in-memory request buffer and to an in-memory request cache.
    c. Immediately return response to the client without applying to the OM DB.
5. Requests will be flushed to disk asynchronously using a double-buffering approach similar to HDFS edit log flush.

In this approach also each request will be executed in 2 phase like but it will happen in applyTransaction. (In first phase we generate response to be sent to client, and add this to cache to do a batch commit).

Note: We need cache to be implemented here because we return response to client without applying to OM DB. Detailed design of cache is explained in this document.

Pros

1. OzoneManager will have better performance as we return the response to clients immediately without any disk write.
2. No expensive calls are happening in startTransaction.

Cons

1. In this each OM will be doing the same work as leader validation of the requests, maintaining the cache.
2. The pre-execution step like (creation of delegation token, allocation of Block etc.,) is done at RPC level so this can be executed by any OM. (Not only leader). (This is not a con, but this looks like we are doing some things which might not be needed as we are doing this step before any validation.)
3. This might be an issue during rolling upgrade where one of the OM got upgraded to the latest software, so it might return the new data for the request in to OM DB, where as other OM's which have not upgraded might write different data. So, we can see OM DB's divergence.
4. Invalid requests also will be written to Raft Logs, as validation is done at very later step.
5. Cache structure will be little complex as we need to store response of the request also. And for different requests it will be different responses.

Implementation details of Late Validation

Next we shall discuss how to implement a request type(Write request) in OM with late validation approach.

Adding a new Request type in OM:

When a new request type(Write request) is added to the OM, the developer must implement the following two interfaces.

1. OMClientRequest.
2. OMClientResponse.

OMClientRequest

Interface which defines the contract every write OM Request should implement.

```
public interface OMClientRequest {

 /**
  * Perform pre-execute steps on a OMRequest.
  *
  * Called from the RPC context, and generates a OMRequest object which has
  * all the information that will be either persisted
  * in RocksDB or returned to the caller once this operation
  * is executed.
  *
  * @return OMRequest that will be serialized and handed off to Ratis for
  *         consensus.
  */
 OzoneManagerProtocolProtos.OMRequest preExecute()
     throws IOException;

 /**
  * Validate the OMRequest and update the cache.
  * This step should verify that the request can be executed, perform
  * any authorization steps and update the in-memory cache.

  * This step does not persist the changes to the database.
  *
  * @return the response that will be returned to the client.
  */
 OMClientResponse validateAndUpdateCache();
```

```
}
```

OMClientResponse

Interface which defines the contract every OM Response should implement.

```
/**
* Interface for OM Responses, each OM response should implement this
interface.
*/
public interface OMClientResponse {

 /**
  * Implement logic to add the response to a batch. This method can be
  * be used to collect multiple updates into a single batch for writing
  * to disk.
  *
  * @param omMetadataManager
  * @param batchOperation
  * @throws IOException
  */
 default void addToRocksDBBatch(OMMetadataManager
omMetadataManager,BatchOperation batchOperation) throws IOException {
   throw new NotImplementedException("Not implemented, Each OM Response should
implement this method");
 }
}
```

Let’s take an example CreateBucketRequest, And see what needs to be done.

Implement OMClientRequest for CreateBucketRequest

```
public class OMBucketCreateRequest implements OMClientRequest {
 private OMRequest omRequest;

 OMBucketCreateRequest(OMRequest omRequest) {
   this.omRequest = omRequest;
 }
 @Override
 public OMRequest preExecute() {
   // Generate OM Request which sets the create time, generate
   // BucketEncryptionInfo, user information.
 }
```

```
 @Override
 public OMClientResponse validateAndUpdateCache() {
   // Validate volume/bucket exists or not.
   // Generate response data which needs to be added to OM DB.
 }
}
```

Implement OMClientResponse for CreateBucketRequest

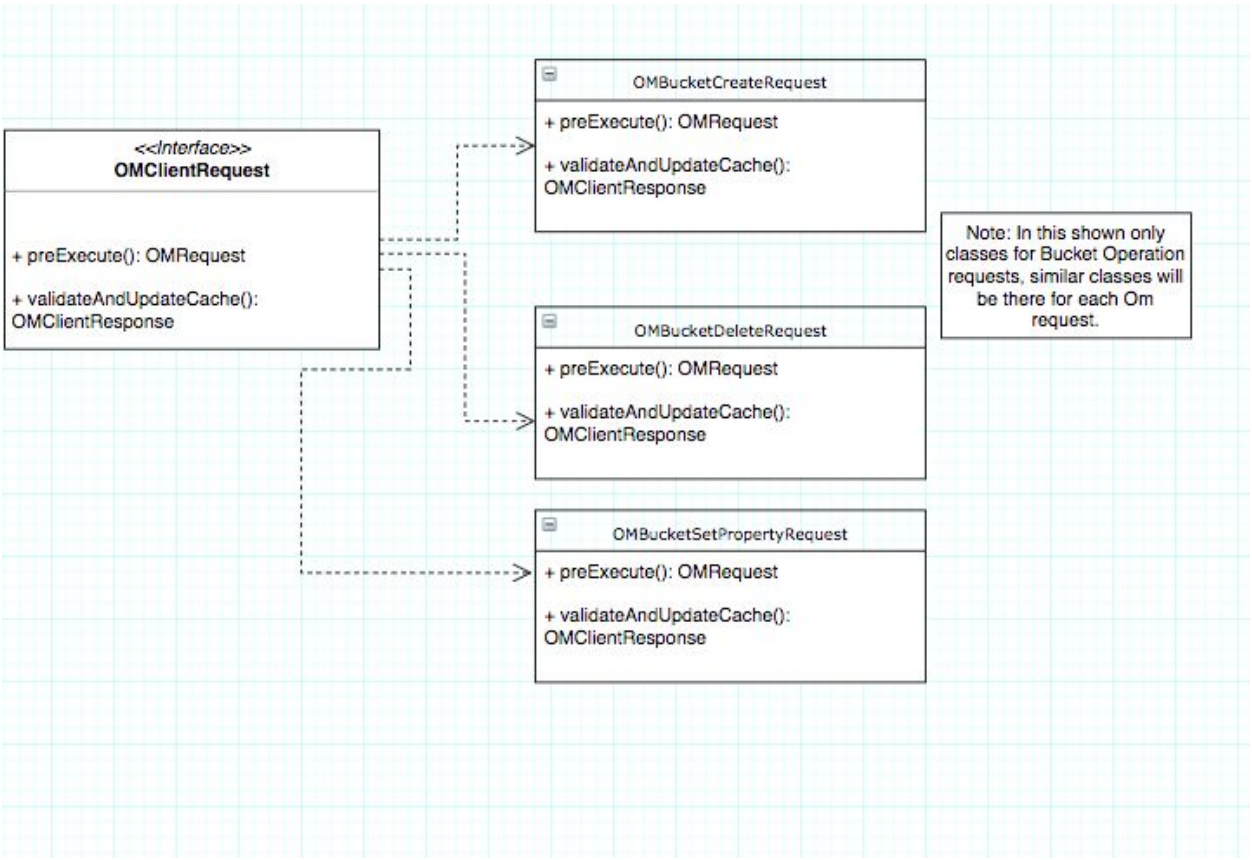
```
/**
* Response for CreateBucket request.
*/
public final class OMBucketCreateResponse implements OMClientResponse {

 private final OmBucketInfo omBucketInfo;

 public OMBucketCreateResponse(OmBucketInfo omBucketInfo) {
   this.omBucketInfo = omBucketInfo;
 }

 @Override
 public void addToRocksDBBatch() throws IOException {
      // Add logic to add omBucketInfo to rocksdb batch
 }
}
```

Class Diagram showing OMClientRequest

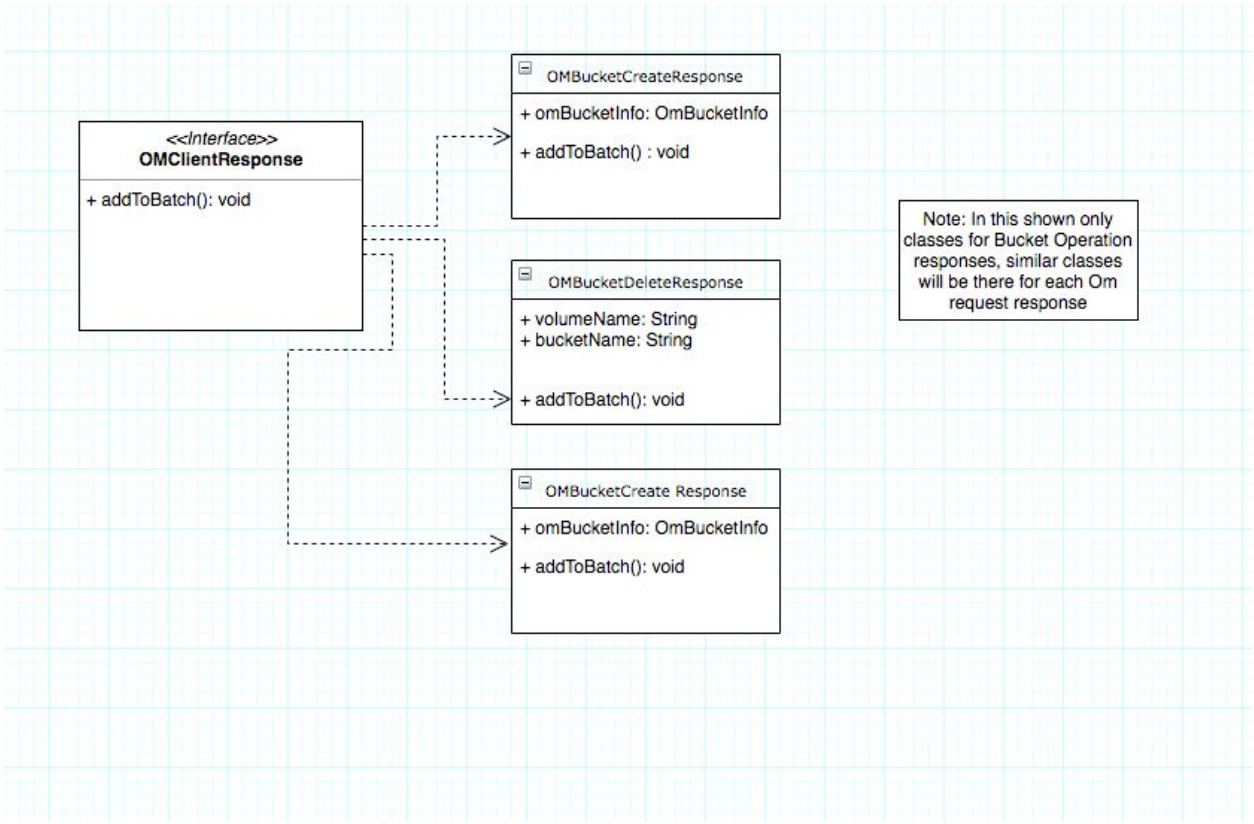


preExecute() - executed at RPC context level. Generate a OMRequest object with all the required information, which will be passed through RatisClient by calling sendAsync(). All the object information that will eventually be persisted to RocksDB and returned to the user must be generated in this step.

validateAndCreateResponse() - Executed in stateMachine applyTransaction. This should be an idempotent operation. This is a very light weight operation which will be executed on leader and follower OM. In general the following steps will be done in this method.

1. Acquire lock.
2. Validate the request can be executed E.g. check that the target volume and bucket exist, perform authorization checks.
3. Create a response object for the request to be added to double buffer.
4. Add the response information to TableCache. This makes the result of the operation visible to subsequent requests.
5. Release lock.

Class Diagram showing OMClientResponse

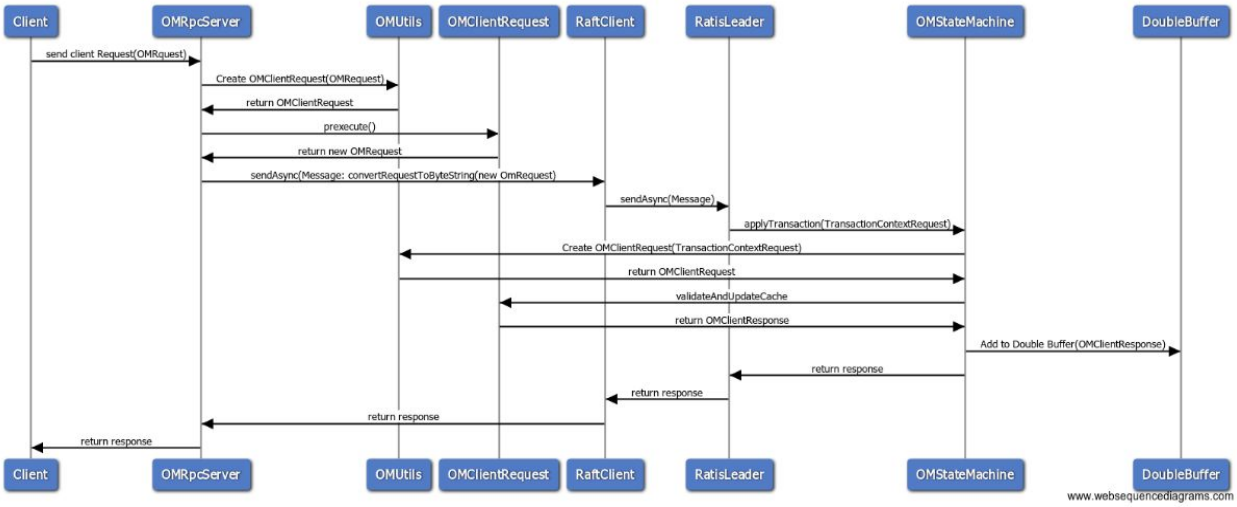


Once OM request is executed in applyTransaction, and response is generated, this is the object it generates, and gets added to the double buffer. Each Response Class will implement addToBatch to define their own implementation. (This is the method where we shall have logic to add the response to RocksDB batch)

Sequence Diagram

In the sequence diagram we have not shown ratis state transitions/calls, once after raft client sends request to Ratis Leader, it considered next call is to OMStateMachine. The main intention of the sequence diagram is to show how the OMRequest will be executed in applyTransaction.

Note: In a high level actual Ratis flow is Ratis leader write the request it to its own log, then send AppendEntries to followers in parallel, wait for the ACK for majority, then call applyStateMachine, applyStateMachine returns a future, wait for the future to be completed, and then return response to client.



Note: Link for the above diagram can be found here.

Appendix - Early validation

This is an alternate way to handle requests where the validation is performed in the startTransaction step. It is probably simpler to implement than Late Validation however it suffers from a performance issue.

The approach is detailed here for completeness.

1. Divide request in to 2 phase, where request will be executed and validated only on the leader OM, and the response will be added to the original request (This will be done in startTransaction). Next in the applyTransaction just it is apply to OM DB. (So all OM's will just apply the response which is given by leader OM to their DB.) We shall discuss this approach further in the next section.
2. In this approach:
    a. Step 1, 2 and 5 is done in startTransaction.
    b. Step 3 will be done after startTransaction and after Ratis got a quorum and steps 4 and 6 will be done in applyTransaction.

Flow of the request

1. OM Received a request for example let's take createBucket.
2. The request is sent to startTransaction where we check like bucket exists or not before creating the bucket, and then we execute the request and create BucketInfo structure

where we store information like creation time, metadata etc., and modify the original request to have this information.
3. In the applyTransaction we apply to OM DB.

StartTransaction of 2nd request can be done before applyTransaction of first request. This is one of the reasons for requiring the cache. We need cache to be implemented to validate the transactions, as when the request is received we validate in the startTransaction, and later in the applyTranasction it is applied to OM DB.

We need cache for two reasons:
a. Performance: To prevent slowing down startTransaction. (Expensive calls like allocateBlock where network call to SCM will happen)
b. Correctness: To avoid situations like inserting key into deleted bucket, deleting renamed key.

Pros

1. If any invalid requests like bucket create where bucket already exists, these requests will not enter in to Ratis log, and this will be ruled out at leader OM itself.
2. Looks cleaner, where all the validation and execution will happen only on leader and followers job is just apply the information to OM DB.
3. ApplyTransaction will be simpler. (We can also batch the requests, instead of doing rocksdb transaction for every request). Meaning for every request instead of doing rocksdb put, we can use rocksdb batching also.
4. Only one node does validation checks. Cannot arrive at different validation results, RocksDB cannot diverge.
5. Cache structure will be little simple, we don't need to maintain response in the cache.

Cons

1. Expensive to do disk reads for validation in startTransaction. (As startTransaction is executed in a serial fashion)
2. Cache will be rebuilt after leader election. As this is built in startTransaction.
3. Have to kill the OM node if request is successfully executed in startTransaction but failed to write in to ratis log, but the subsequent request succeeded to write to raft log. (As we are validating from the cache which is built in startTransaction). Cases like createBucket Transaction first and next we have a openKey request for createKey. If createBucket request failed to write to ratis log, but we have added in to our cache that bucket is created, and we shall execute the next request createKey for the bucket thinking that bucket will be created. SO, here we shall end up in a situation where our cache data is wrong and endup in creating a key for the bucket which does not exist.

Performing expensive operations like allocateBlock (which is a network call to SCM), disk read(which needs to be done for validating bucket/volume exists or not) in startTransaction, will significantly affect the OzoneManager performance as startTransactions() is not an asynchronous call in Ratis StateMachine. So, we shall go with late Validation approach.