OzoneManager HA

1. Introduction

OzoneManager (OM) is the Namespace manager for Ozone. It maps keys to blocks and containers. OM will be able to send an Ozone client to the right container location. OM interacts with clients and SCM only. It does not talk to the Datanodes. For containers, OM requests access from the SCM.

In an Ozone cluster, OM and SCM are two single points of failure. To ensure High Availability (HA), we should have HA for OM and for SCM. In this document, we propose an HA implementation for OzoneManager using Raft protocol. SCM HA will be designed separately.

2. Goals

1. Implement OM HA using Raft protocol.
   Apache Ratis provides a customizable Raft protocol. We will use a (2N+1) Ratis ring to achieve high availability. We will have one OM Leader and 2N OM Followers.
2. Fast failover between Leader and Follower OMs.
   The lag between Follower and Leader OMs should be minimized.
3. Strong Consistency.
   All the OMs must be in sync. If failover occurs, the new OM Leader must be in the same state as where the previous OM Leader left off.
4. HA failover should be transparent to the users of the Client library.
   The Ozone client should handle the transitions during failover triggered by failure of less than a quorum of OM nodes. It may not be possible to hide other classes of failures from the client e.g.
   - Loss of OM quorum
   - Network partition between client and leader OM.
   - OM leader being slow/unresponsive to client operations due to excessive load.

3. HA using Ratis

We propose implementing HA for OM using Apache Ratis. Apache Ratis is a highly customizable Raft protocol implementation in Java. Each Raft server has two main components - Raft Log and State Machine. State Machine is the fault-tolerant component. The inputs or state change on State Machine is done through commands in the Raft Log. Ratis provides support for a pluggable State Machine.

For OM HA implementation, we will have a (2N+1) node Ratis ring. Each OM will run an instance of the Ratis server. Ratis will write the Raft logs on disk and OM will act as the State Machine for its Ratis server. OM persists its state in a RocksDB store. The Ratis ring will have one Leader node and 2N follower nodes.

Applications/ clients contacting the OM would go through the RaftClient to redirect the request to the RatisServer Leader. Ratis replicates the state using a Write Ahead Log (WAL) i.e. we first persist a state change to the log before updating the in-memory state. Once Ratis server gets consensus from the followers that the request has been persisted to the log, it makes corresponding change to the State Machine i.e. OM.

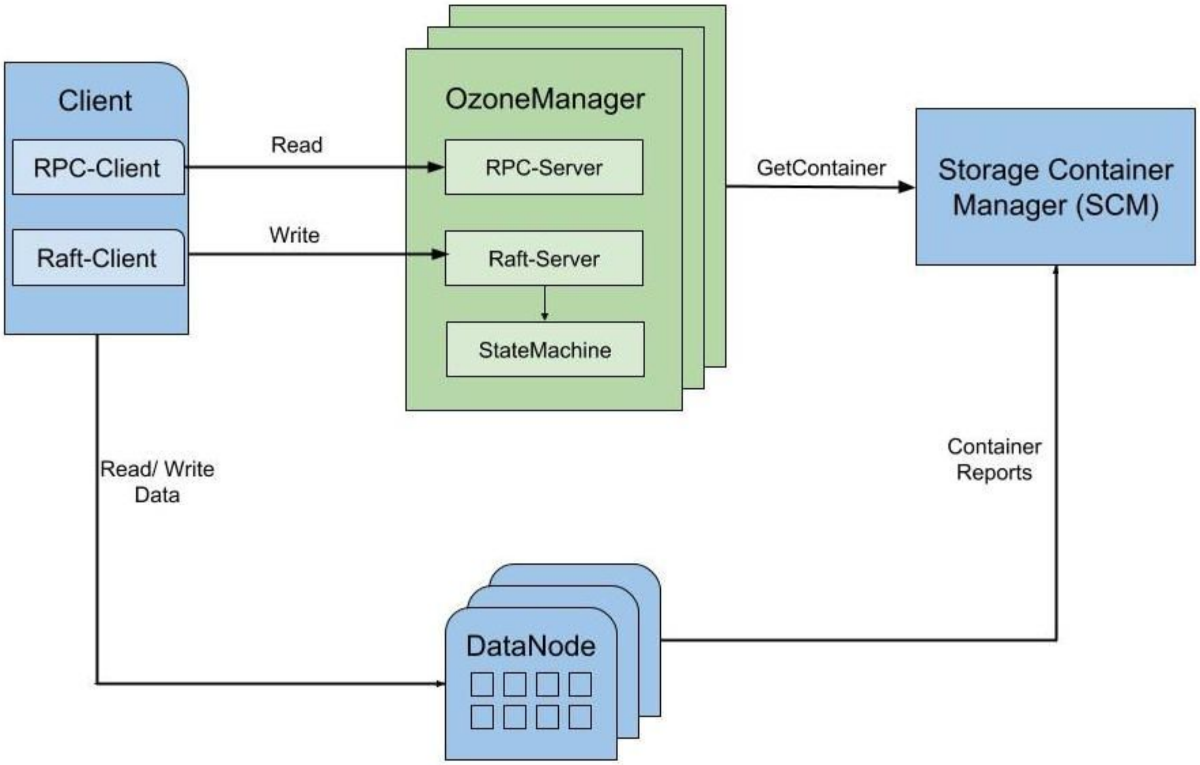
4. Terminology

We can think of the OM Leader as equivalent to Active and the followers to Standby in HDFS terminology. When referring to OM Leader, we refer to the OM with the Ratis instance in Leader state. And the OM with Ratis in Follower state will be referred to as OM Follower.

In this document, OM failover refers to the process of choosing a new Ratis Leader node through election.

For the rest of this document, we will consider a 3-node Ratis ring with one OM Leader and 2 OM Followers during no-election state.

5. Architecture



5.1. Configurations

Currently, we have the *ozone.om.address* configuration key to define the address of the OM service. Clients use this config key to discover OM. To enable HA for OM, we should set this config value to a comma separated list of all OM addresses. The OMs will also use the same configuration to discover other OMs in the ring.

Clients will also be able to discover all the OMs in the cluster by contacting one OM. When client contacts one OM, the OM would return a list of all other OMs in the cluster. This enables clients to discover other OMs not present in their config parameters.

5.2. OM Components

The OM will have two separate ports RPC server and Ratis server (RaftServer). The OM itself would act as the state machine for its Ratis server. All write requests from clients must land at the RaftServer. The RPC server can process read requests but not write requests.

5.3. OM State Changes

All state changes in OM should come through Ratis so that all the OMs are in sync. Clients cannot directly make state. For requests which can potentially modify the state of the OM, the client sends the request through RaftClient. The RaftClient passes the request to Leader RaftServer. The RaftServer replicates the request to its followers and after getting a quorum of acknowledgements from the followers, it applies the state change on the OM. OM itself acts as the state machine for its RaftServer.

5.4. Fast Failovers

To achieve fast failovers, the OM Follower's in-memory state should closely follow that of the Leader. When the Leader applies the state change on the OM, that is commits a transaction, it informs Ratis followers of the new *commitID*. The Ratis followers will apply transactions from their logs upto this *commitID* and catch up with the leader. This helps in reducing the lag between the active and follower OMs as the followers closely tail the state of the leader.

5.5. OM State

Each OM should always be aware of its state i.e. the state of its Ratis server. Ratis server should notify OM about any change to its state. There is an open Jira - RATIS-297 to add this support to Ratis. OM will also ping the Ratis server periodically to query its state.

5.6. Client-Ratis Communication

All the write requests from the client should come through the RaftClient. The RaftClient sends the request to the Leader RaftServer. The read requests from clients will be handled by the RpcServer. The read request

The OM will have separate ports

to communicate with clients for reads and writesand with Ratis. Clients can only talk to the OM and not to the Ratis server. Clients send their requests to OM and OM passes on the request to the Ratis server.

There is absolutely no direct communication between clients and OM's Ratis servers. Any exception arising from Ratis server should be transformed into user-friendly exceptions before throwing to the client.

5.7. Client Communications

All write requests from client to OM are sent via the RaftClient. The RaftClient sends the request to the Leader RaftServer. The RaftServer replicates the request to its followers and then updates the state machine data in the OM.

In case of read requests, the client first sends a request through RaftClient to the RaftServer. The RaftServer returns the most recent *commitID* and the leader information. The RpcClient uses this information to contact the RpcServer of the leader OM, which then services the read request from the client. The client should also be able to perform read operations from follower OMs upto the *commitID* returned by the leader RaftServer. This support would be added later on.

Currently, we would let only the leader OM to service read/ write requests from the client. If a client sends a request to an OM Follower, the OM should throw a *NonLeaderException*. The client should then try contacting the next OM to service the request. We will discuss more on how client discovers OM Leaders in the section Discovering OM Leader.

A failover can occur before the client receives an acknowledgement or response from the OM. So the client should retry its request after a timeout. The inbuilt retry cache in the RaftServer handles duplicate client requests. We will discuss more on how duplicate requests are identified and handled in the section Client Requests.

5.8. SCM Communications

Only the OM Leader must initiate state transitions and requests to SCM. If an OM Follower receives any response from the SCM, it should ignore it. The new OM Leader would resend the request to SCM and replicate this information to the followers. We will discuss more on OM-SCM interaction in the section OM-SCM Communications.

5.9. NonLeaderException

If an OM Follower or an OM Candidate receives any requests/ responses from either the client or the SCM, it should throw back a *NonLeaderException* to inform the client/ SCM that it is not the leader currently and that it should try contacting a different OM. Along with the exception, the OM should also return a list of other OMs in the cluster.

5.10. Handling Re-Elections

During a Ratis re-election, a new Leader is chosen from the Ratis ring. This leads to status changes in the OMs - one OM transitions from Leader to Follower and one OM transitions from Follower to Leader. All active transactions should be tracked by the Followers. Whenever there is any change in the Ratis state - Leader, Follower or Candidate, the Ratis server should notify the OM about this state change.

On a transition from Leader to Follower, we just need to update the state of the OM as follower so that it does not service any further requests. On transitioning an OM from follower to leader, Ratis automatically replays all the log records upto the committed *commitID*. RaftClients will redirect their requests to the new leader RaftServer.

There might arise a scenario such that a re-election happens but the old OM leader still thinks it is the leader even though a new leader has been elected. In such a scenario, the stale OM leader would not be able to service any write requests as it would not receive a quorum from the followers. But if there is a split-brain scenario and the old OM leader cannot heartbeat to the other OMs, it could potentially serve stale read requests upto three heartbeat intervals. After not receiving a response from the followers for three heartbeat intervals, the old OM would change its state to follower and no further stale reads will be serviced by this OM.

5.11. Ratis Log Purging

We need to periodically purge the Ratis Log to maintain a check on the log size. The OM state is persisted in RocksDB. We can periodically take a snapshot of the State Machine and possibly purge the Ratis logs upto the transaction the checkpoint has been taken. We will discuss more on checkpoints/ snapshots and log purging in the section on Bootstrapping OM.

6. Clients

6.1. Client Requests

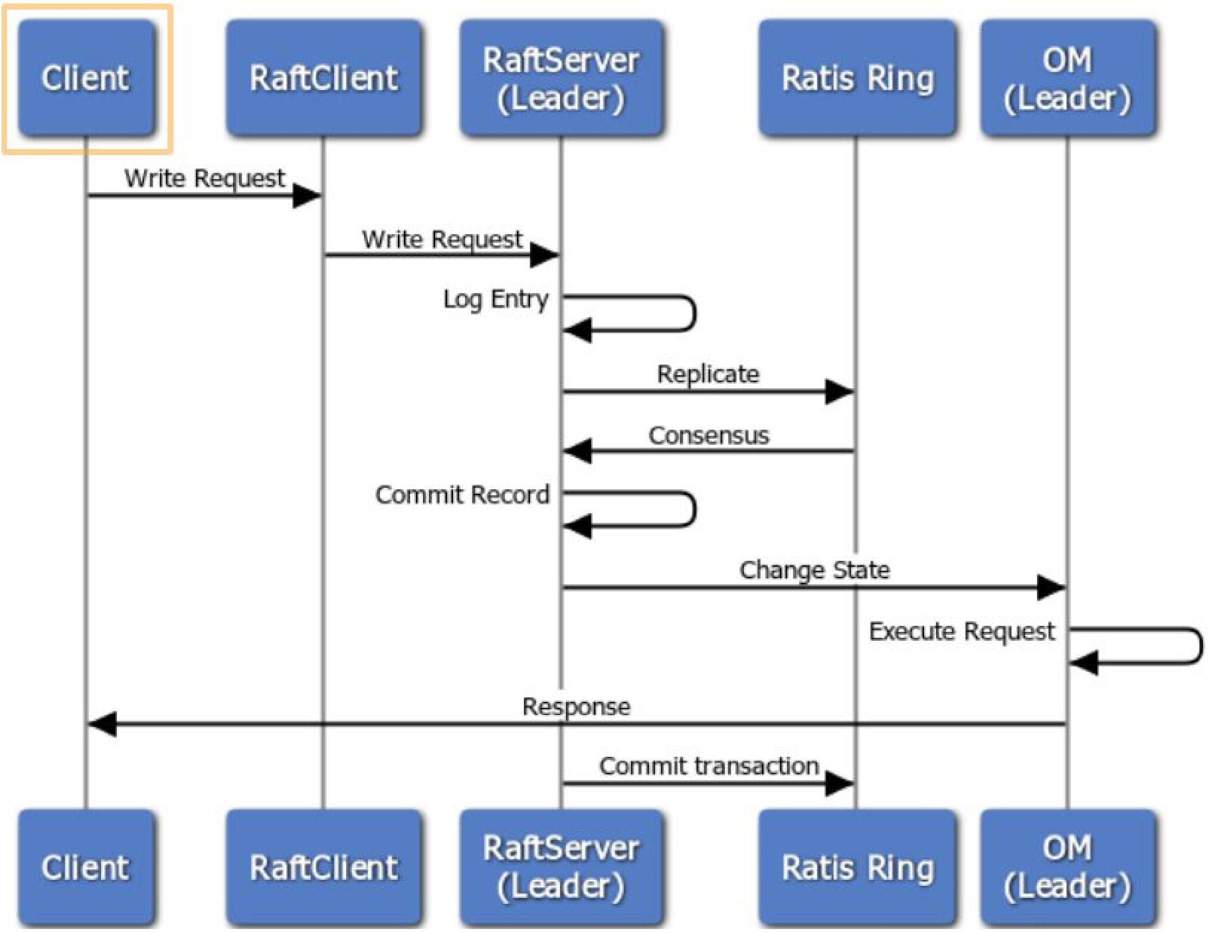
The following sequence diagrams show how requests from the client are handled. We show the handling of read and write requests separately. Any request that can potentially lead to a state change will be treated as a write request.

In these diagrams, *Ratis Ring* refers to the Followers in the Ratis Ring. Once an entry is logged on the Ratis Leader, it commands the Ratis Followers also to log the entry. Ratis then informs the OM to proceed with the request. After a request is executed and state change is successfully completed on the OM Leader, the Ratis Leader informs the Followers to commit the corresponding transactions on their OMs (that is *commitID* is updated).

6.1.1. Write Requests

When OM's RaftServer receives a state change request, it replicates this state change request to its own log and to the log of all the followers. After receiving a quorum of acknowledgements, it executes the request and applies the state change on the OM, updates it *commitID* and sends a response back to the client. The leader RaftServer then updates followers with the new *commitID*. The Ratis followers execute the request on their corresponding OMs and apply the state change.

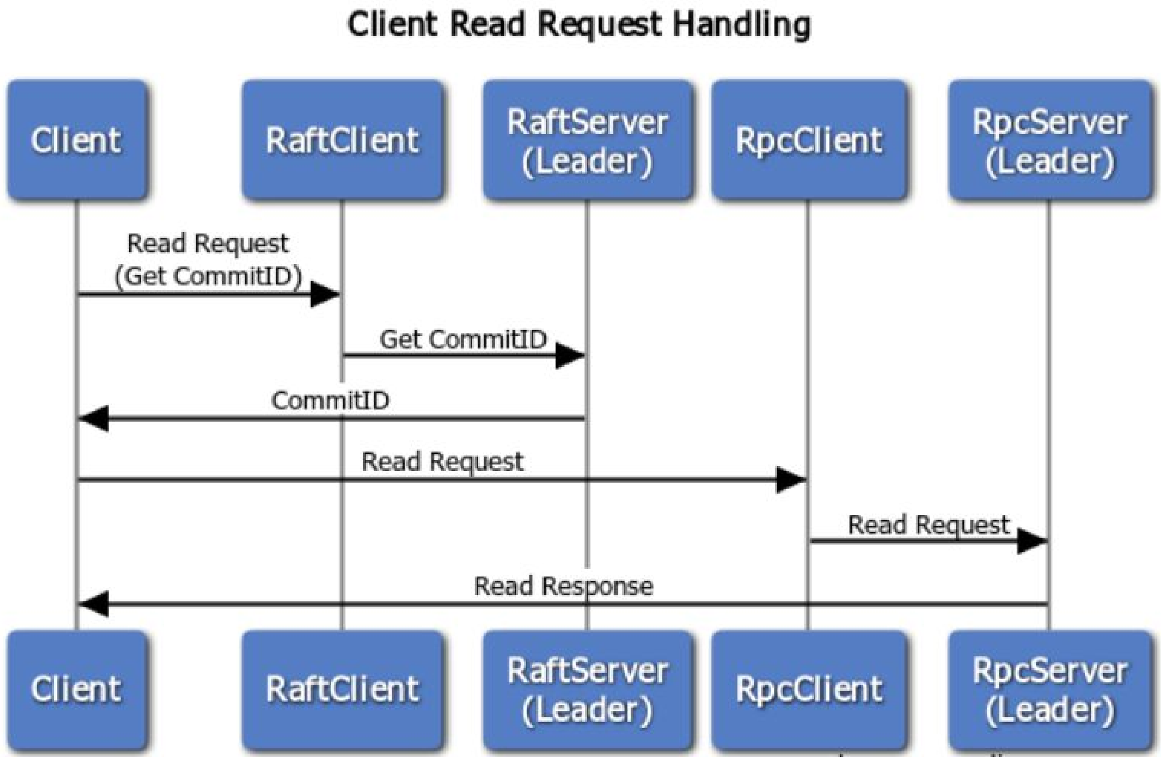
Client Write Request Handling



6.1.2. Read Requests

For read requests, clients should know which OM is the leader currently. To locate the OM leader, client sends the request to the RaftServer leader through RaftClient. The RaftServer responds with the current OM leader information along with the current *commitID*. Once client discovers the OM leader, it caches this information.

Read requests are routed through the RpcClient to the RpcServer of the OM Leader. The OM leader services these read requests. Currently, only the OM leader can process read requests. In case the state of the leader changes, it throws a *NonLeaderException* to the client and the client tries to discover the new OM leader. But we plan to support reads from OM Followers later on.



6.2. Client Retry Requests

6.2.1. Client Retry Request Types

Client request retries can be classified into two categories - Idempotent and AtmostOnce.

6.2.1.1. Idempotent

These client requests can be retried on failover. There are no repercussions of executing these requests more than once on the OM. For example, most read requests would qualify as *idempotent*.

6.2.1.2. AtmostOnce

These client requests must be executed at most once. This is ensured using the Ratis retry cache. Ratis maintains a retry cache and adds all incoming write requests here. Each write request is validated against the retry cache and only processed further if there is no corresponding entry in the cache. Duplicate requests are responded back with the same response which was cached for the original request. Because of this guarantee, a client can retry requests on OM failovers and other network failure conditions.

Ratis followers also maintain a retry cache which is populated when the follower applies the transaction. So even on OM failover to a new leader, client retry requests are not executed more than once.

6.3. Discovering OM Leader

The client reads the configuration key *ozone.om.address* to get the list of OMs in the cluster. On startup, the client pings all the OMs to figure out the OM Leader. The client will cache the first OM to respond as the Leader. All subsequent requests will be directed to the cached OM leader. Only when this OM stops responding as the leader or throws a *NonLeaderException* will the client start looking for the new leader.

6.3.1. Split-brain

During a re-election, there might arise a scenario where two OMs consider themselves as Leaders. This can happen if the old leader node loses contact with the other nodes and does not know that a new leader has been chosen. This old leader node would consider itself to be the leader until a *heartbeat timeout* occurs. During this duration, the old OM Leader might present itself as the leader to the client. But it cannot process any client write requests as it would not have the consensus from the ratis ring. The client, upon realizing that the leader is not able to process its write requests, pings the next OM in the ring in a round-robin fashion.

It is possible in case of split-brain scenario, the old OM leader gives out stale reads to clients. But this can happen only for the duration till *heartbeat timeout* occurs. Typically the heartbeat timeout happens after three heartbeats from leader are not responded to by a quorum of followers. After the timeout, the old OM would change its state to follower and no further stale reads will be serviced by this OM.

6.3.2. Backoff - Retry

When client sends a discover request to OMs and all OMs respond with Follower state or are unreachable, then the client should back off for some time before retrying as leader election might be in progress. The clients will use exponential backoff strategy for retries. This implies that the interval between retries will increase exponentially until a cut-off is reached. If the cut-off interval is reached without discovering the OM Leader, the client throws an exception that OM is unreachable.

7. OM-SCM Communications

OM communicates with the SCM through its RpcServer. But the response from the SCM might need to be replicated

7.1. Handling OM Failover

If the OM leader sends a request to the SCM and failsover before receiving a response back, the new OM leader would retry the same request to the SCM. All requests to SCM can be treated as idempotent. OM can resend duplicate requests to the SCM without any significant consequences. For instance, duplicate requests from OM might lead to unused block assignments or extra container allotments. But SCM is equipped to handle such situations. Hence, from OM's perspective, requests to SCM are idempotent.

7.2. Handling SCM Responses

Any message/ response from SCM that can potentially change the state of the OM must go through the RaftServer so that it is replicated to all the OMs. This should be handled on a case by case basis for each communication message between OM and SCM.

8. OM Snapshots

In case an OM falls behind in the ring or if an OM is restarted or if we add a new OM to the ring, we should have snapshots in place so that the OM does not have to replay all the log entries from the beginning. A snapshot upto *commitID X* should be sufficient to reconstruct the state of the OM upto that transaction.

8.1. Snapshots

Ratis currently supports snapshots/checkpoints. We can configure Ratis to auto trigger snapshots when log size exceeds limits and/or let the State Machine implementation take snapshots as required. The snapshots are stored in the State Machine and Ratis server can retrieve them when required.

When a snapshot is in progress, the OM will stop applying any state changes. The Ratis server will still accept incoming log entries and record them but will hold off on applying the changes to the state machine until the checkpointing is done. As such, we should configure Ratis such that only OM Followers take snapshots and let the OM Leader actively apply transactions. When the OM Follower completes taking the snapshot, it would catch up with the Leader by applying all the pending transactions.

8.1.1. Snapshot Contents

An OM snapshot would essentially be a snapshot of the LSM (RocksDB). When Ratis triggers a snapshot on the OM, the OM will take an image of the RocksDB at that point of time and persist it to a file.

8.1.2. Triggering Snapshots

OM Followers trigger snapshots on themselves based on the following parameters:

A. Time elapsed since the last snapshot was taken
B. Number of committed transactions since the last snapshot was taken

We also add a jitter to the time interval or the number of transactions so as to scatter snapshot triggers. This jitter will be generated using a combination of random numbers and configured interval.

The OM Leader does not take any snapshots. In case a failover occurs and an OM Follower transitions to Leader while checkpointing is in progress, the checkpointing will be interrupted. This way checkpointing does not interfere with the election process of Ratis.

The OM Leader should download snapshots from the followers periodically. Leader can follow similar criteria as followers to determine when it needs to download a snapshot from one of the followers. The Ratis Leader would send out a request to get the list of snapshots available with each follower. It would then choose the latest available snapshot and download it from the follower.

We propose that only the Ratis server should be able to trigger the snapshots. The state machine should not take snapshots on its own. To introduce a jitter for snapshot triggers on different nodes, the OM should be able to configure an initial jitter on the transaction number from when the snapshotting interval starts. For example, say we set the interval between snapshots to be 10,000 transaction. To ensure that not all Ratis servers trigger snapshots at exactly the same set of transactions, i.e. at *commitIDs* 10000, 20000, 30000 etc., each server should have a different initial jitter for the first snapshot and then take snapshot after configured interval from there on. So let’s say we set the initial jitter for first server to be 1000 and for the second to be 3000. Then the *commitIDs* at which the first server would take snapshots would be 11000, 21000, 31000 etc. and the for the second server, these numbers would 13000, 23000, 33000 and so on. We would need to introduce a new configurable parameter in Ratis for setting the initial transaction jitter. This value must be set before starting the Ratis server.

8.2. Purging Ratis Logs

Each Ratis server individually decides on when to purge logs. This decision will not be communicated by the Ratis Leader. Rather, the Ratis server would make this decision based on the snapshots available with its state machine. It periodically check with its OM on the latest available snapshot and the corresponding *commitID*.

9. Bootstrapping OM

When a stale or dead OM is restarted or a new OM is added to the cluster, we need to bootstrap that OM to bring it up to speed with the other OMs in the system. In this process, we need to bring the new OM state at par with the other OMs so that it can start participating in the Ratis replication process.

When a Ratis node is added to the Ratis ring and it receives heartbeats from the Ratis Leader, it informs the Leader about its current state of logs, that is the *commitID* up to which it is current. The Leader then sends this follower the latest snapshot and log entries as required. The Follower applies the snapshot and log entries to its state machine - OM.

We may be able to optimize this later to send incremental snapshots using the RocksDB *Transaction Log Iterator*. This can reduce the bootstrap time when an OM instance needs to catch up to a few hours or few days worth of updates. However it will not help when bootstrapping a brand new OM instance.

When an OM is bootstrapped, it discards its previous metadata in RocksDB, if present. The OMs metastore is rebooted with the snapshot of the RocksDB received from the Leader.

10. Diagnostics

Similar to *hdfs haadmin*, we can have an *ozone om haadmin* to help with diagnosing failovers and possible problems with OM servers. The *om haadmin* would support two subcommands - *getLeader* and *failoverInfo*.

1. ***ozone om haadmin getLeader***
   The *getLeader* subcommand would return the current OM Leader. If an election is in progress and there is no leader available currently, we would return a message indicating that an election is in progress.

2. ***ozone om haadmin failoverInfo <Number of previous failovers>***
   The *failoverInfo* subcommand would give us information about the last failover. This information includes the following
      i. Previous Leader
      ii. Current Leader
      iii. Time of the failover
   This subcommand will also support an optional parameter to specify the number of previous failovers to return, *N*. By default, we would return only the last failover. If this parameter is specified, we would return the above mentioned information for the last *N* failovers.