Ozone Cross-Region Bucket Replication

HDDS-12307
Author: Ivan Andika (ivandika@apache.org)

Context and Scope

There are cases where users might require syncing data from SG to US.

There are a few solutions that we have implemented internally

- Cross-DC Synchronous (Realtime) Writes
- Cross-DC Container Distributions: A placement policy that distribute the containers across nearby DCs after the container is closed

These solutions have the following characteristics:

- The DCs are close to each other so that the latency increase is still acceptable
    - SG <-> MY
    - These DCs also have dedicated network between them, guaranteeing their service
- The Datanodes are distributed across multiple DCs. However, they are still managed by a single SCM service

There are cases where users might require syncing data from SG to US. However, the current Cross-DC DR solutions will introduce unacceptable latencies with 100ms+ round trips. Therefore, Cross-DC Synchronous (Realtime) Writes cannot be used for this case. Moreover, these clusters do not share the same SCM, so there is no single unified view of both clusters' data (datanodes). Therefore, any Cross-DC Container Distributions solution will be complex since one datanode might need to be "owned" by more than one SCM service.

We need to find another replication solution to work under these constraints.

Terminologies

This document introduces the following terminologies to aid understanding

- Region: A single geographical region, as opposed to DC, different regions implies significant network latencies and instabilities between them
    - Unlike Cross-DC which might be within the same region
    - For example: SG and US are different regions, but SG and MY are not in different regions
- Cluster: This is a Ozone cluster with a unique SCM Service with a unique Cluster ID
    - OM and DNs will use the SCM Service ID to identify the cluster ID
- Source Cluster: The Ozone cluster where the replication originates from

- Target Cluster: The Ozone cluster which will be the target for the bucket replication from the Source Cluster
    - It is located in a different region such that there are significant latency from the Source Cluster
    - It is a completely separate Ozone cluster in the sense that does not share any namespace (OM) or blockspace (SCM / DNs) compared with the Source Cluster
- Source Bucket: The source bucket in the source cluster
- Target Bucket: The target bucket in the target cluster

Requirements

The following are the goals

- Provide near real-time replications between different Ozone clusters (SG -> US) between an active source and a passive target
    - The keys written / deleted in the source cluster should be written / deleted in the destination cluster within a specified time (should be in minutes, ideally seconds)
    - Passive target means that the target should not be modified at all and only acts as the receiver of the replication
- Both namespace (OM) and data blocks (SCM/DN) need to be replicated to the destination cluster
- Ensure that the source cluster traffic remains stable even when replications are happening
    - Bucket replications should not significantly affect user in the source cluster
    - Avoid full scan for buckets for each replication
    - The replication should not be done synchronously during the OM writes
        - The bucket replication should not be in the user's write / read critical path
- Ensure a predictable recovery point objective (RPO): the maximum length of time permitted that data can be restored from, which may or may not mean data loss
    - The goal is that a key created in the source bucket will be replicated to the target bucket within minutes (ideally seconds)
- Minimize the cross-region traffic
    - For each unique piece of data, we should only send it once across the region
    - If possible, support batching requests to increase cross-region throughput

For out-of-scope requirements, please refer to the "Extensions" section (e.g. Bidirectional (Active-Active) Replication)

Alternative Designs Considered

This section highlights alternative designs that were considered, but rejected because of one reason or another. However, these alternative designs are helpful in deriving the final design.

Alternative Design 1: Periodic DistCp Jobs

DistCP is the current HDFS solution for cross-region replications. Internally we will run a distcp for each new Hive partition every day for Hive to Hive Cross-Region Replication.

We can use DistCp for our bucket replication. The idea is to set up a periodic distcp job between the source and the destination HDFS clusters.

Pros

- It is simpler: Setting up periodic jobs can be done quite easily (e.g. using cronjobs)
- Distcp implementation will setup a map reduce jobs that will parallelize the copy from the source and the cluster
- No additional Ozone components needed

Cons

- It is not “realtime”: The freshness of the data depends on how frequent and how fast the distcp runs
    - Distcp run interval is usually a few hours
- It incurs significant overhead to the source cluster: It requires scanning of all the files in the source bucket (possibly in the target bucket)
    - This might affect users accessing the paths that are undergoing replication
    - Might also affect OM service for a very large bucket
    - This is only if the distcp is without using snapshot

Alternative Design 2: WAL based replications

The idea is to replicate the WAL directly to the target cluster and let the target cluster replay the WALs directly. These are used by RDBMS (MySQL, PostgreSQL), HBase Cluster Replication (https://hbase.apache.org/book.html#_cluster_replication), and other similar systems.

Pros

- The changes in the source cluster are replicated asynchronously in real time

Cons

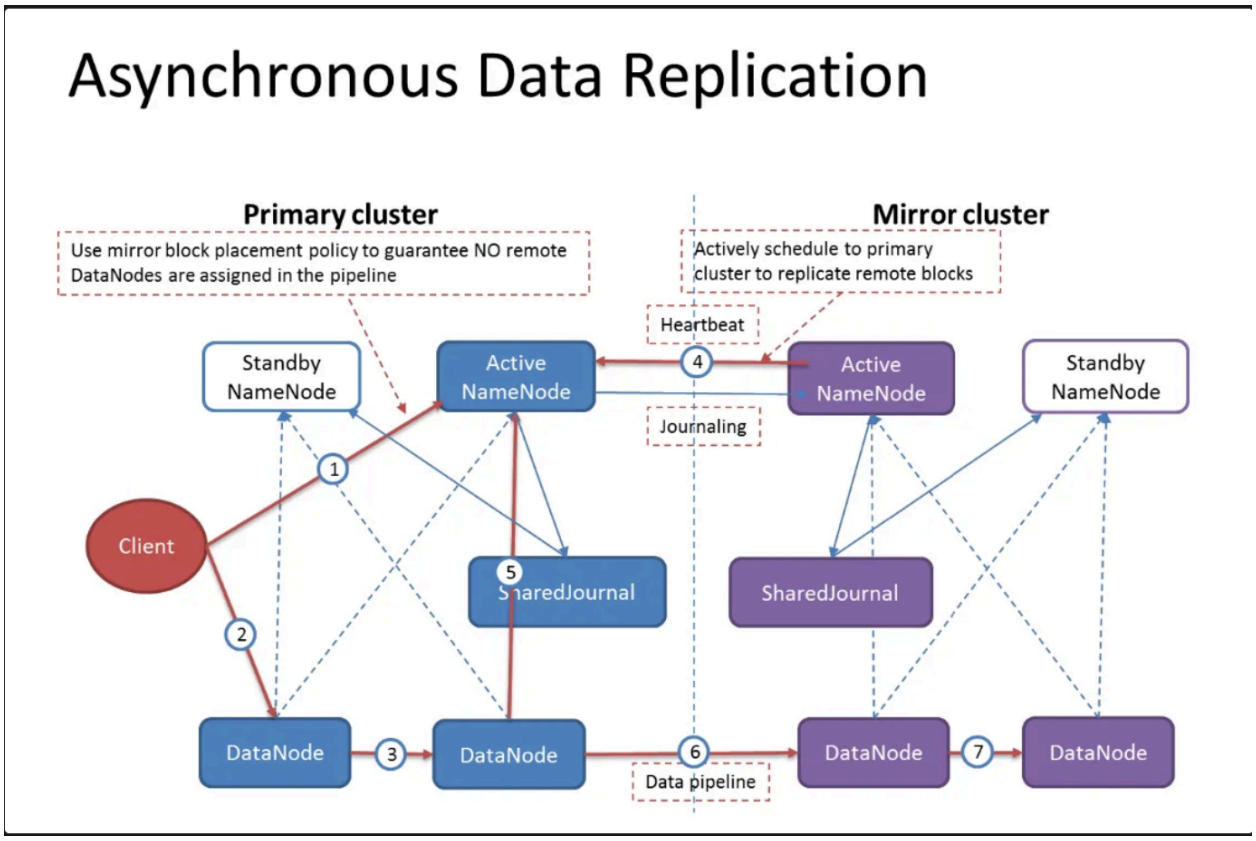
- Only deals with case where the data is already in WAL
    - The data for each WAL entry is quite small
- In Ozone, replicating the OM WAL (RocksDB WAL / Raft Log) is not enough since the data in the datanodes also need to be replicated
    - Also pure WAL based replications cannot handle scenarios when WAL entry needs to be modified
        - keyName needs to be modified
        - OmKeyLocationInfo should refer to blocks in the target cluster

Alternative Design 3: Zero loss HDFS data replication for multiple datacenters (HDFS-5442)

Ref: https://issues.apache.org/jira/browse/HDFS-5442

The HDFS community proposed a design of data replication between two HDFS clusters. There are some overlaps with the existing Ozone DR replication, such as synchronous writes.

We are only going through the asynchronous data replication part



Flow

1. When a client is writing a HDFS file, after the file is created, it starts to request a new block. And the primary cluster Active NameNode will allocate a new block and select a list of DataNodes for the client to write to. For the file which needs only asynchronous data replication, no remote DataNode from mirror cluster is selected for the pipeline at Active NameNode.
2. As usual, upon a successful block allocation, the client will write the block data to the first DataNode in the pipeline and also giving the remaining DataNodes.
3. As usual, the first DataNode will continue to write to the following DataNode in the pipeline until the last. But this time the pipeline doesn't span to the mirror cluster.

4. Asynchronously, the mirror cluster Active NameNode will actively schedule to replicate data blocks which are not on any of the local DataNodes. As part of heartbeats it will send MIRROR_REPLICATION_REQUEST which will contain batch of blocks to replicate with target DataNodes selected from mirror cluster. The mirror cluster doesn't need to aware of real block location in primary cluster.
5. As a result of handling the MIRROR_REPLICATION_REQUEST, the primary cluster Active NN takes care of selecting block location and schedules the replication command to corresponding source DN at primary cluster
6. A DN will be selected to replicate the data block from one of the Datanodes in primary cluster that hold that block
7. As a result of the replication pipeline, the local DN can replicate the block to other DNs of the mirror cluster

Pros:

- Easy DR failover
    - Using active-standby cluster pattern make DR failover more straightforward since the standby cluster is a mirror of the primary cluster
        - Clients only need to switch to the new cluster
- Possibly higher throughput
    - Since the actual data replications are only within the source cluster's and target cluster's DNs

Cons:

- The design seems to require major changes in HDFS implementations
    - A lot of the changes requires changes in the NameNode implementations (e.g. heartbeat logics
- Tight coupling between the Primary and Mirror cluster
    - Active NNs in both clusters need to communicate with each other
    - Hard to implement cases where the source and target keys have different storage policy
        - For example source bucket uses RATIS/THREE, but target is EC
- Active-standby clusters design is inflexible
    - The solution is currently designed for pure active-standby cluster where the standby cluster does not serve active user
        - This causes resource wastage since the standby cluster would be idle
    - The current design also need to replicate all the files to the standby cluster
        - The granularity is at cluster level, instead of directory or file level
- Possible inconsistencies between namespace and blockspace
    - Since the namespace and the blocks are asynchronously replicated separately, the mirror cluster may lags behind some portion of the data blocks
        - Namespace replications might be a lot faster than the block replications
    - Therefore, the mirror cluster may have the namespace, but some of the data blocks may be missing (corrupted files)

- It is hard to monitor the discrepancies between the namespace replication and the blockspace replication

Ozone Cross-Region Bucket Asynchronous Replication

Overview

Prerequisites

1. An identical bucket to source bucket needs to be created in the (the “target bucket”)
    a. These information needs to be the same
        i. Bucket layout (i.e. OBS / FSO)
        ii. Bucket ACLs (Might not be required)
        iii. Default replication configuration
        iv. Target bucket quota should be disabled or should be larger than the source bucket
    b. However, the target bucket name does not need to be the same
    c. Ease-of-use: Similar to HBase destination table creation, we should make it easy to export properties of a bucket, and for a destination to import the same bucket properties.
2. The target bucket should be empty
    a. If the bucket is not empty, the underlying keys might be overwritten
3. The target bucket should not be writable except by the Ozone admin
    a. This is to prevent other users writing to the target bucket while the replications are running, that might trigger edge case
4. Both source cluster OM service and target cluster OM service should have Snapshot enabled
    a. Snapshot is enabled by default in 1.4

These prerequisites will be checked by the replication subsystem before the bucket replication starts.

There are two major steps when setting up the bucket replication

1. Initial batch replication of source bucket (bootstrap)
    a. Backfill the newly created bucket with the existing objects from the source bucket
2. Asynchronous bucket live replication
    a. Tailing the subsequent changes (CDC) of the source bucket and replicating it to the destination bucket

Key Info Preservation

These are the information that will be preserved during the bucket replication

- Custom metadata (including ETag)
- Object tags
- ACLs (non-compulsory for Ranger-based authorization)
- Object owner

These information can be optionally preserved

- Modification time
- Creation time

These information will not be preserved

- ReplicationConfig: It depends on the replication config of the target bucket

Replication Subsystem

To provide a pluggable and independently scaled replication subsystem, we will use a separate daemon called **Syncer**. The responsibility of Syncer is to manage and carry out a single bucket replication task. There should be only one Syncer for a single bucket replication pair ([source bucket, target bucket]). This is enforced by source cluster OM service during the registration phase.

The main flow is as follow:

1. User starts a syncer specifying the source bucket and target bucket configuration
    a. Both source and target cluster should have snapshot enabled
    b. Target bucket should be empty
2. A Syncer process starts and generate a unique UUID (similar to datanode UUID)
3. The Syncer registers to the source OM service
4. The OM service checks whether another existing Syncer is running replications for the bucket
    a. If there is one, the Syncer will be rejected and it will not start
        i. The OM service persistently records the current active Syncer ID for the particular bucket replication

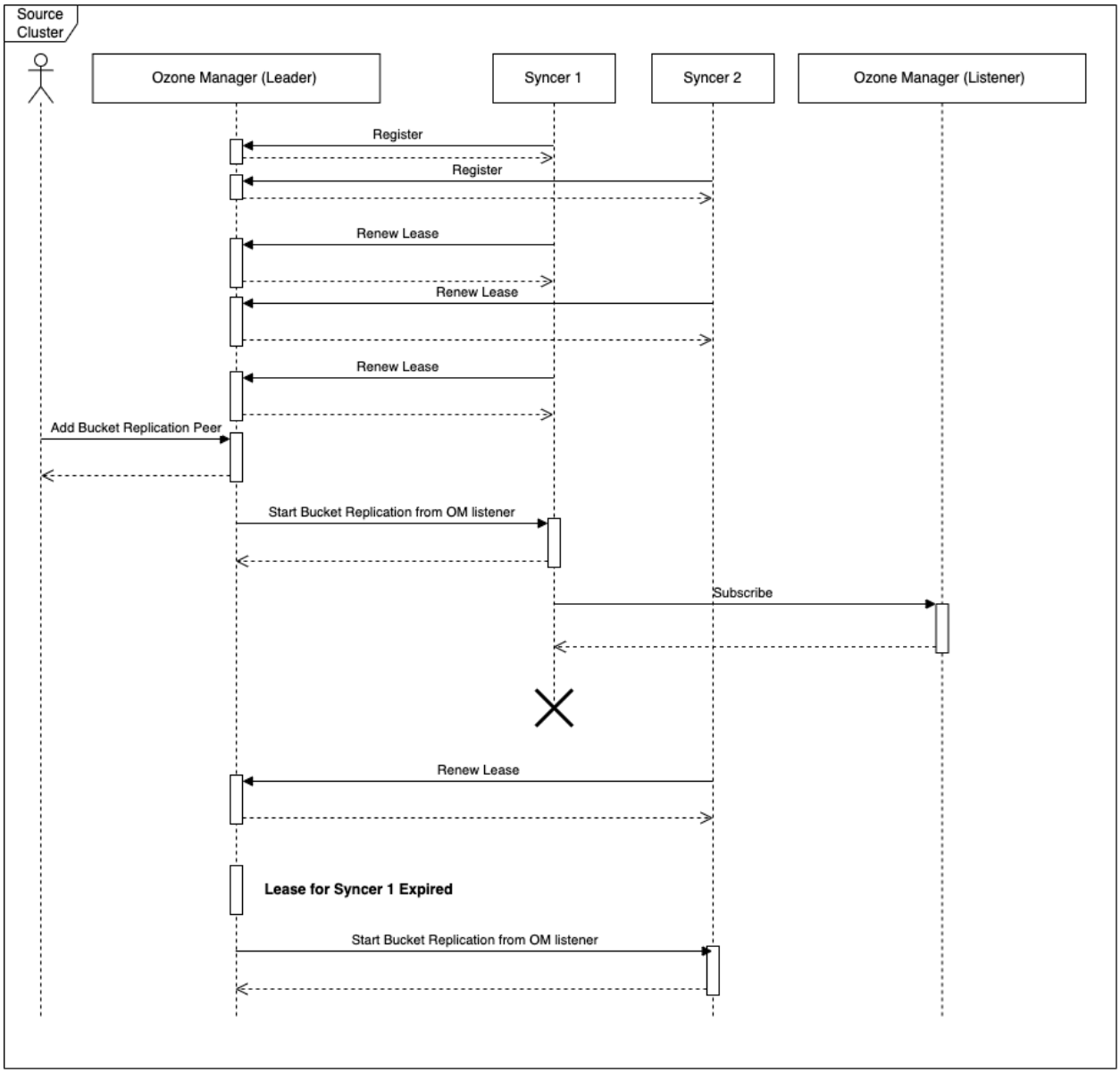
Design Requirements for Syncer

- Syncer should act like a “worker” that will carry out the underlying replication works, where the OM acts a “master” that contains centralized info regarding the replication process
- Syncer should be able to be replaced by another Syncer in another node if it fails
    - This means that Syncer should not hold information (e.g. last key seen, replication stage) that the next Syncer might need
    - It might store local information like the underlying persistent queue info

High Level Design

High Level Design

- Source OM service acts as the Master and responsible for
    - Managing bucket replication information
    - Assigning bucket replication tasks to the registered Syncers
- Syncer acts as the Worker which receives bucket replication tasks from the OM service and replicating it to the target buckets
    - For each machine, only one Syncer service will be created per OM service pairs, meaning one Syncer might be in charge to multiple bucket replications between a single OM service pairs



High Level Flow

1. Syncer registers to the source OM service
    1. After registration, the Syncer will be assigned a lease and is eligible to be assigned replication tasks
    2. The Syncer setup heartbeat mechanisms with the source OM service to periodically renew the lease (this can be implemented with gRPC bidirectional streaming API or simple RPCs)
2. User sends a create bucket replication request to the OM service
3. OM leader will assign the new bucket replication task to available Syncer and available OM listener
    1. The assignment strategies could be based by active bucket replications in the Syncer or other information (available bandwidth, disk space, etc)
4. Syncer starts requesting bucket replication data (snapshot, snapdiff, log entries) to the assigned OM listener
5. At every heartbeat, it will report its current state to the OM service and OM service will persist the information to its state
    1. See “Bucket Replication Policy Persistence” section
6. If the Syncer is disconnected / dead
    1. OM service will wait until the lease expires
    2. OM service will pick the other available Syncer and send replication tasks
    3. If the previous Syncer comes back and tries to ask for replication with the expired lease, it will be rejected
        1. The previous Syncer will clean up the specific bucket replication resources

Step 1: Initial Bucket Batch Replication

Before bucket replication starts, we need to backfill all the existing keys in the source bucket to the target bucket. This can be done using the Ozone snapshot feature.

General Flow

This the general flow of the initial bucket batch replication

1. Syncer takes a snapshot of the source bucket
    a. Using the CreateSnapshot API
        i. Ozone shell: ozone sh snapshot create
    b. This will create a RocksDB checkpoint in a separate snapshot directory containing all the data (SST files) under the bucket during the snapshot
    c. While the snapshot exists, the keys will not be deleted
2. Syncer will download keys from the Ozone snapshot
    a. Also take the latest applied index and the RocksDB sequence number of the RocksDB checkpoint for the snapshot

    b. This is necessary for the subsequent live replication(see section Phase 2:Near Real-time Live Replication) to decide from which log index / sequence number it should start from tailing from
3. Process all the keys in the snapshot
    a. Steps
        i. Download all the keys from the source cluster
        ii. Convert the snapshot key names to the original key names
        iii. Upload the keys in the snapshot to the target cluster's bucket
    b. This can be parallelized as much as possible since there are no overlapping keys in the snapshot (unlike in the subsequent bucket live replication)
    c. For large snapshots, we can support periodically persisting the last key uploaded to the OM service so that the next Syncer can pick up from the last Syncer left off in case of failure / restarts.

OM follower / listener support

When the CreateSnapshot request is sent to the OM service, all of the OM nodes will generate the same snapshot (RocksDB checkpoint). Hence, the Syncer can pick other OM followers / listeners to download the snapshot from. This should prevent additional overhead of the OM leader. However, bucket snapshot might not have been generated in the non-leader OM if the follower / listener is lagging from the OM leader. In that case, Syncer simply needs to retry the bucket snapshot download.

Step 2: Asynchronous Bucket Live Replication

The asynchronous bucket replication implementations will be split into two phases of implementations

- Phase 1: Incremental Ozone Snapshots
- Phase 2: Near Real-time Live Replication

Phase 1: Incremental Ozone Snapshots

The idea is first to take a snapshot of the bucket and replicate it to the target cluster. After the initial snapshot is finished, periodically take a subsequent snapshot and run snapdiff to check the difference between the two snapshots. Based on the snapdiff, the syncer will upload or delete keys accordingly. See https://ozone.apache.org/docs/edge/feature/snapshot.html for snapshot documentation.

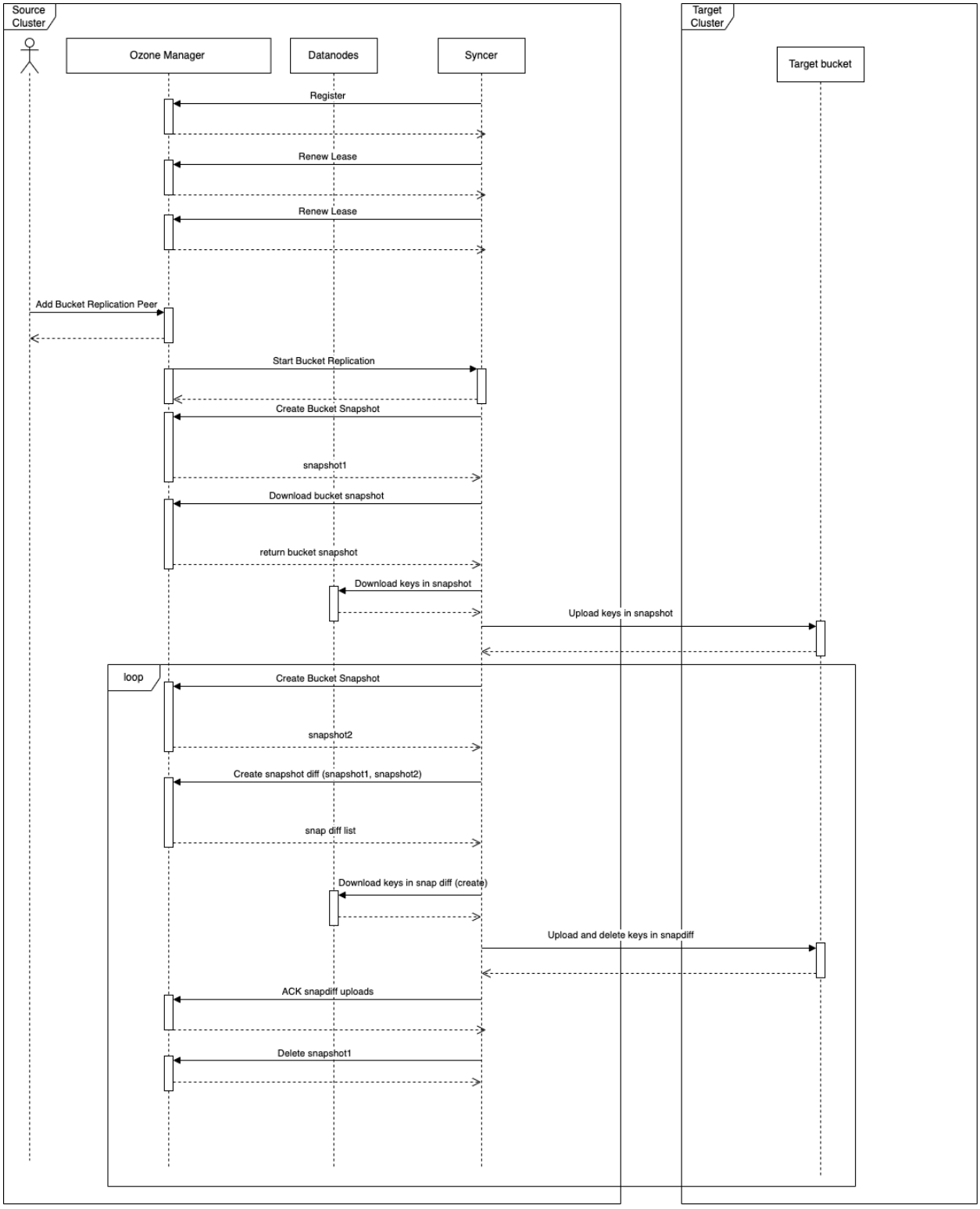
The implementation logic would be similar to "hadoop distcp -update -diff", but currently it does not need the target cluster to have a snapshot with the same name.

Some other systems already implemented the replication subsystem internally. For example

- Cloudera Replication Manager

    - https://docs.cloudera.com/replication-manager/1.5.4/replication-policies/topics/rm-pvce-understand-ozone-replication-policy.html
- CephFS mirroring
    - https://docs.ceph.com/en/reef/dev/cephfs-mirroring/

System Diagram



General Flow

1. A Syncer process is started that will register with the OM service
    a. OM service will check if there are active replication, and will reject the Syncer registration if it already exists
2. User will add a bucket replication peer to the OM service, specifying the target bucket configuration (e.g. bucket name, service ID, etc)
    a. This can also be done during the initial Syncer daemon
3. OM service will assign the new bucket replication task to the registered Syncer and send the bucket replication information to the Syncer
    a. This can include information regarding the previous replication (e.g. previous Syncer failed) or the current snapshot
4. The Syncer will createSnapshot request to OM for the source bucket (let's call it snapshot1)
5. The Syncer will download the snapshot1 from the OM and store it in the local directory
6. The Syncer will scan through all the keys in the snapshot
    a. Change the key names
    b. Upload the keys to the target bucket
7. The Syncer will use createSnapshot to create another snapshot (let's call it snapshot2)
8. The Syncer will call the SnapshotDiff API between snapshot1 and snapshot2 to get the keys that are added and deleted between two snapshots
9. The Syncer will download the snap diff list from the OM
10. The Syncer will scan through all the key diffs in the SnapDiff
    a. Change the key names
    b. Upload new keys, and delete the removed keys
11. After the key diffs have been replicated to the target cluster, send an ACK to the OM that the keys in the snapdiff has been uploaded
    a. OM will update the last replicated index internally
12. Delete the old snapshot1
    a. This will make the keys in snapshot1, but not snapshot2 to be eligible for deletions
13. Repeat to Step 7 (incremental snapshot)

Pros & Cons

Pros

- Unlike distcp this does not seem to require scanning all the files in the bucket since snapdiff will take the difference in stt files
    - We can run this more frequently (every few minutes)
- There are no conflict of keys in the snapshot / snapshot diffs, therefore we can parallelize the replications for all the keys
    - In the Phase 2 (Near Real-time Live Replication), there might be delta changes that conflicts with the ongoing delta replication
        - For example, a key is created and deleted and created again within a short amount of time

        - These operations need to be applied sequentially to avoid synchronization issue
- Less implementation overhead, the mechanism is already implemented in Ozone
    - We only need to implement the replication subsystem that will carry out the replication
- Unlike the Near Real-time (Phase 2) the keys will not be deleted while there is a snapshot
    - There won't be a case where the underlying key blocks have been deleted

Cons:

- Although should be better than periodic distcp jobs (we can probably run this for every few minutes), it is still not "realtime"
    - Any change on the source bucket needs to wait for the subsequent snapdiff for it to be replicated
- Ozone Snapshot might not be very stable yet
    - There are some previous issues on Ozone snapshots documented in community

Handling failures

Failure Condition	Solution
Destination bucket is full / no permission	Stop the replication and throw an exception
Network issues between source and destination clusters	Keep retrying to send the replication to the target cluster until successful
Source Ozone Manager is down	Wait until the OM node is up
Syncer failure	Run another Syncer. The new Syncer will get the information from OM and pick up where the last Syncer left off.

Phase 2: Near Real-time Live Replication

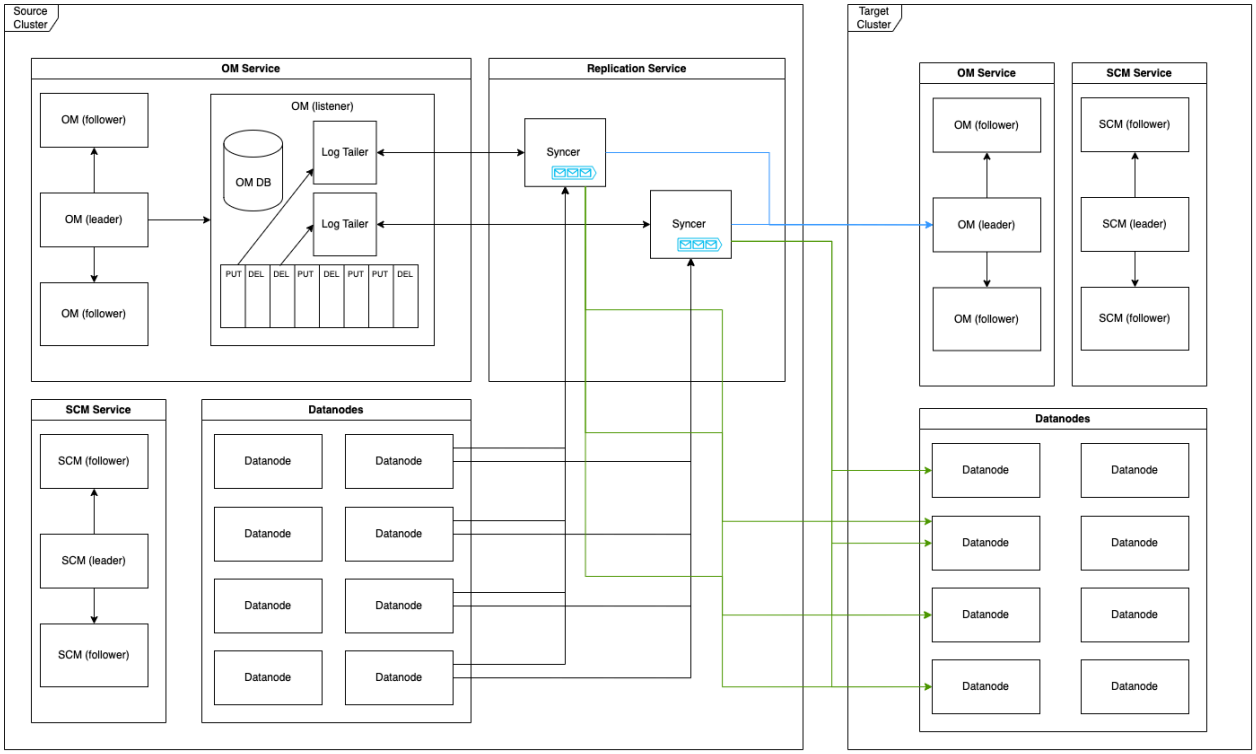
After Phase 1 has been implemented, we can implement a more real-time replication strategy. The main idea is that each Syncer will create a persistent gRPC bidirectional streaming channel (using HTTP/2) with one of the OM nodes that will send the log entries related to keys for the specific bucket to be created through a **Log Tailer**. The Syncer will then persist the log entries to its internal persistent queue which will be consumed by a work pool to replicate the data to the destination bucket.

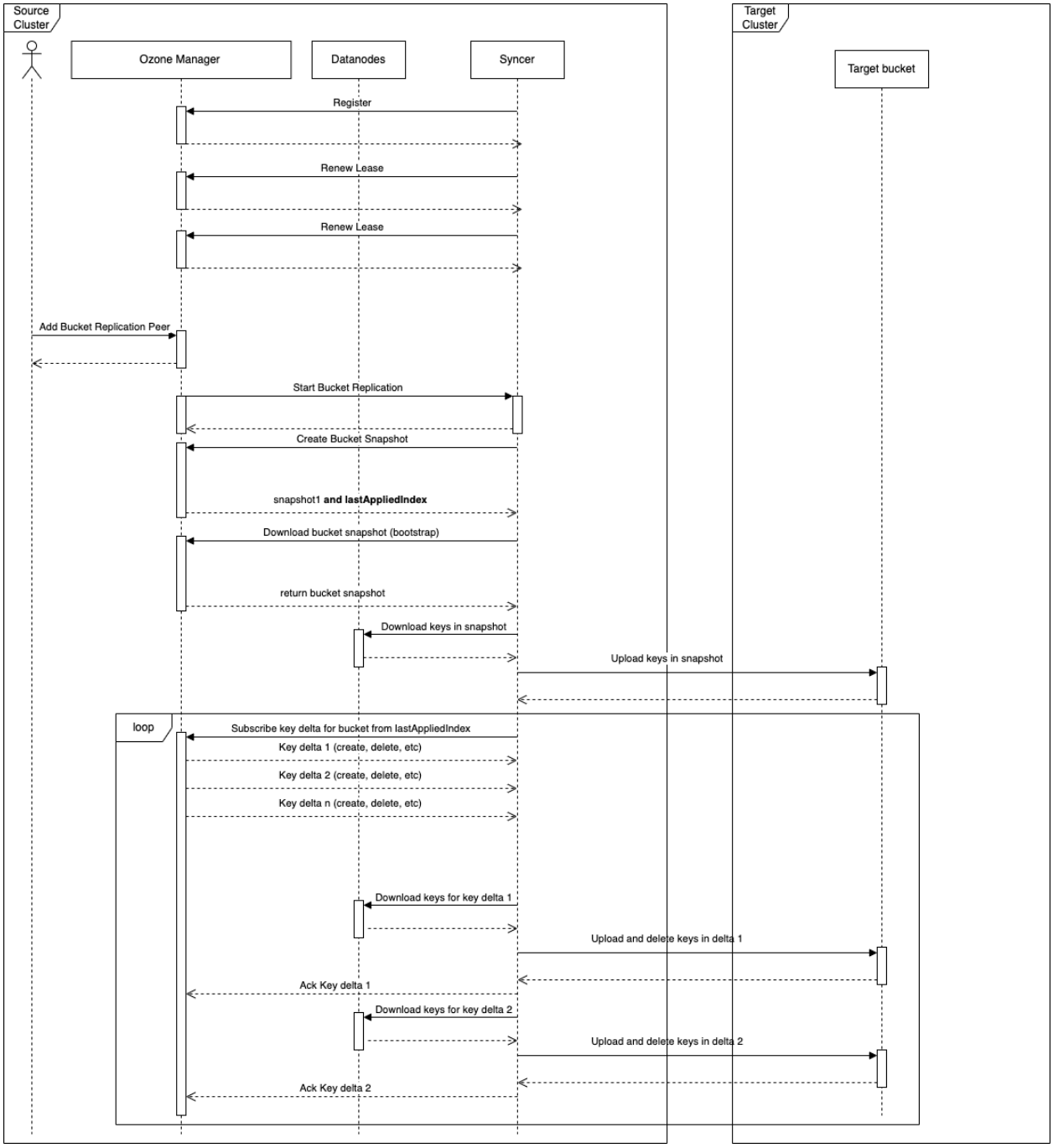
The design is inspired by the works in

- SeaweedFS
    - https://github.com/seaweedfs/seaweedfs/wiki/Filer-Active-Active-cross-cluster-continuous-synchronization)

- etcd mirror

System Diagram





General Flow

1. Syncer sends a subscribe API to one of the OM node (in this case, it is an OM listener)
    a. Syncer needs to specify the log index / sequence number to tail from, which was received from the incremental snapshot
2. Syncer and OM establish a gRPC bidirectional streaming connection
3. The OM will create a separate log tailer thread for the particular gRPC bidirectional streaming connection and send the log entries to this channel

    - This thread will continually tail the log entries (Raft log entries / RocksDB WAL) for the entries for the particular bucket and send them to the gRPC bidirectional channel
    - If there are two different bucket subscriptions, there will be two tailer threads (one for each bucket)
    - OM will keep sending the log entries until the maximum pending replications configuration is hit, afterwards, it will wait until Syncer has replicated it to the target bucket
4. Syncer will push the log entries to its own replication queue
    a. This is the same thread that communicating with OM
5. Another Syncer thread will process the log entries sequentially
    a. For key deletion log, Syncer directly sends it to the target bucket
    b. For key creation entry, Syncer first downloads the key data, and upload it to the target bucket
6. Once the Syncer sends a specific log entries successfully,
    a. Syncer will send a gRPC message containing the last log entries index replicated
        i. OM will record this information internally
        ii. The log entry is used by OM as backpressure mechanism to limit the number of pending log entries sent to the Syncer (e.g. 1,000,000 entries)
    b. Syncer will truncate the processed log entries from the persistent queue

Pros & Cons

- Pros
    - Real-time compared to the incremental snapshot
        - Objects can be replicated in a matter of seconds
- Cons
    - More implementation requirements
        - gRPC bidirectional streaming protocols
        - log tailers
        - replication queue
    - More edge cases
        - Handling purged logs
        - Unlike incremental snapshot (Phase 1) solution, when the Syncer process a log entries to replicate the object to the target bucket, the key might already been deleted since there are subsequent deletion log entries

Log Tailer (CDC)

Log Tailer is the main CDC (Change Data Capture) component to tail the OM WAL. There are a few choices of OM Logs to tail from:

- RocksDB WAL
    - Using RocksDB getUpdatesSince with the specified sequence number
    - Pros

        - Since RocksDB write operations only consist of PUTs and DELETEs, the log tailer implementation only needs look out for these operations in keyTable and fileTable
            - Therefore processing speed should be faster
        - When more new OM requests are added, there is no need to update the Log Tailer implementation
        - The replication solution can be reused for OM x Recon synchronization
    - Cons
        - RocksDB implementation has not been well-understood
        - Need to use getUpdatesSince that had a known bug to intermittently crash the OM
            - https://issues.apache.org/jira/browse/HDDS-8271
- Ratis log
    - Pros
        - Ratis implementation is more well understood and easier to change to fit out need
    - Cons
        - When a new OM request is added, tailer / syncer needs to add a filtering check for the OM request
            - In the future, once Ratis log only contains DB mutation (https://issues.apache.org/jira/browse/HDDS-11415), it will be similar to RocksDB WAL instead
- OM Audit log
    - Pros
        - Since our audit logs is ingested in a ElasticSearch-like platform, the logs will not be purged easily
            - We can even keep that longer (e.g. 1 week)
    - Cons
        - Depends on the log platform
        - Require pattern matching: This might be complex and failure-prone since OM audit logs do not have backward compatible schema (e.g. protobuf)

We will use Ratis log for this design since it is more understood. However, RocksDB can be more efficient and can be considered if it is more appropriate.

Replication Queue implementations

There are a few considerations of the replication queue to use

- Kafka (with MirrorMaker)
    - This will save the main implementation of the inter-region replication
        - Mirror-maker will implement the asynchronous replication logic
    - However not cost-effective since it requires at least two Kafka topics (one in the source cluster and one in the target cluster)
    - Need to decide how many Kafka topics created per bucket replication pair
- Kafka as a Queue

    - https://cwiki.apache.org/confluence/display/KAFKA/KIP-932%3A+Queues+for+Kafka
    - Kafka is currently working on changes to make Kafka more suited for a queue workloads
    - However, this has not been finished yet
- Distributed log implementation
    - Ratis LogService: Implementing our own Distributed Log Service
        - This has an overhead of a control plane (log manager) and at least 3 servers to store the logs
    - Apache BookKeeper
    - Distributed Log Service complexity might not be needed for now
- Local persistent queue
    - Chronicle Queue (https://github.com/OpenHFT/Chronicle-Queue)
        - Used in Apache Cassandra (https://github.com/apache/cassandra/blob/trunk/src/java/org/apache/cassandra/utils/binlog/BinLog.java)
    - Big Queue (https://github.com/bulldog2011/bigqueue)
    - It is simpler to implement locally, but it is not redundant (i.e. only one copy of the persistent queue exist)

The current decision is to use Local persistent queue (Chronicle Queue) per bucket replication for cost and simplicity sake.

OM follower / listener support

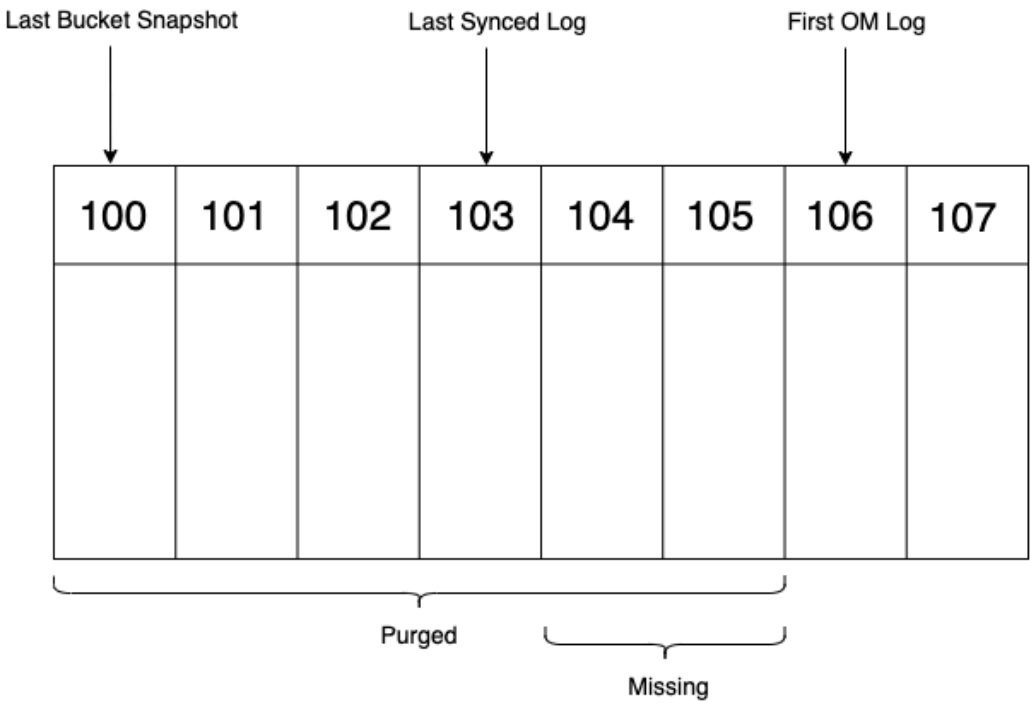
To prevent additional overhead on the OM leader, we need to support tailing a non-leader OM node, at the cost of slightly stale logs. We can pick a OM follower to tail, but there might be a chance that the OM follower can be promoted to leader any time. Therefore, to ensure that the bucket replication does not affect the OM leader and follower, the Syncer can get the replications from the OM listener. The tradeoff is that the OM listener might be slightly more stale than OM leader / follower.

Handling purged OM logs

Both RocksDB WAL and Raft logs can be purged. However, there might be situations where the logs needed for the bucket live replication are purged / missing

- Manual OM DB synchronization
    - This will most likely remove some Raft logs and RocksDB WAL
- OM cannot replicate the logs fast enough before the logs are purged
    - Slow syncer
    - Network issue

To see one such case, please refer to the following figure



In the figure there are three important log index information:

- Last bucket snapshot index: This is the log index of the bucket snapshot
    - The snapshot captures the state of the bucket at that particular index
- Last synced log index: This is the index of the last log that has been replicated to the Syncer
    - The Syncer should already persistently replicate this log or even replicate the underlying change
- First OM log index: This is the index of the OM logs
    - All the logs with lower index has been purged

The logs from up to index 105 has been purged, but the Syncer only have the logs up to index 103. This means that the logs 104 and 105 cannot be replicated to the Syncer anymore. This means that any bucket changes in log 104 and 105 will be lost.

We can have preventive measures used to ensure that OM has not purged the logs needed for replications

- Ratis needs to ensure that the Raft logs that might need to be replicated are not purged
    - This requires having a large enough purge preservation configuration or implementing a purge log index custom preservation logic
- We let the live replication start replicating to the replication queue while the initial batch replication is still running
    - After the batch replication finish, start the live replication from the the replication queue first
    - This ensures that if the batch replication takes a while, the purged logs would most likely have already been replicated to the replication queue to be processed later

However, if Syncer discovers that the log has been purged during live replication. There is no way for us to recover the purged logs changes. There are a few possible ways to handle this:

1. Delete all the data in the target bucket and restart the syncing process. This is the most straightforward solution.
    a. Pros
        i. At the end of the syncing process, the target bucket will be consistent with the source bucket
    b. Cons
        i. All the keys in the target bucket needs to be deleted and reuploaded again which can incur unacceptable space and network overhead
2. Take a periodic bucket snapshot (every a few minutes) and replicate the snapshot diff of the current snapshot and the last snapshot
    a. Pros
        i. Compared to full sync, this will incur less overhead since only the keys between the snapshot diffs are going to be handled
    b. Cons
        i. There will be duplicate key creations and deletions since some of the logs between the previous snapshot's index and the current snapshot index might already been replicated before

Handling duplication messages (deduplication)

There might be chances where OM resends the replication messages (network interruption) that have been seen by the Syncer. Syncer will check the first and last Raft log index / sequence number in the Persistent Queue, and will ignore the replication message if:

- The OM replicate log entries that have been replicated to the target bucket
- The OM replicate log entries that have not been replicated to the target bucket, but already in the Persistent Queue

To achieve this Syncer will have an internal information regarding the last replicated index and the first and last index in the persistent queue

Additionally, we can use the target bucket metadata which contains the last replicated index. The OM will ignore any replication that is lower than the last replicated index.

Handling failures

There are some failure conditions that need to be handled

Failure Condition	Solution
Destination bucket is full / no permission	Stop the realtime live replication and throw an exception
Network issues between source and destination clusters	Keep retrying to send the replication to the target cluster until successful

Source Ozone Manager is down	Wait until the OM node is up
Syncer failure	Run another Syncer. The new Syncer will get the information from OM and pick up where the last Syncer left off.

The Syncer and the OM will use the bidirectional streaming API as a way to check whether the other end is still online.

Supported operations

Supported operations

- Commit keys
- Complete MPU
- Single Delete
- Batch Delete
- ACL Add, Delete
- Key Renames
    - For cross-bucket key renames
        - If both rename source and destination bucket is in the source bucket, it can be a pure metadata operation that directly sent to the target OM service
        - If the rename source is not in the source bucket, but rename destination is in the source bucket, it will be regarded as a key creation
        - If the rename source is in the source bucket, but the rename destination is not in the source bucket, it will be regarded as a key deletion

Unsupported operations

- Snapshot operations
- Multi tenancy operation
- Secret operations
- …

Management and Configuring Cluster Replication

Ozone Shell

These are the Ozone commands used to manage the replication

- Cluster-wide
    - Enable bucket replication
    - Disable bucket replication
- Add bucket replication target / policy
- Delete bucket replication target / policy
- List bucket replication targets / policies
    - Support listing active bucket replications

- Cancel bucket replication
- Get bucket replication
- Pause bucket replication
- Resume bucket replication

Bucket Replication Stages

The bucket replication process can be divided into a few rough steps:

- BATCH_REPLICATION: Initial bucket batch replication
    - See the “Step 1: Initial Bucket Batch Replication” section
- INCREMENTAL_REPLICATION: Incremental Ozone Snapshots replication (Phase 1)
    - See the “Step 2: Asynchronous Live Replication Phase 1”
- LIVE_REPLICATION: Near Real-time replication (Phase 2)
    - See the “Step 2: Asynchronous Live Replication Phase 2”

This information is stored in the source OM service per bucket replication.

Bucket Replication Policy Persistence

The source OM service needs to persist the replication state information. These are:

- Replication Stage
    - INITIAL_BATCH_REPLICATION
    - INCREMENTAL_REPLICATION
    - LIVE_REPLICATION
- Replication Status
    - ACTIVE
    - ERROR
- Replication Status Message
- Target Bucket
    - Target OM Service
    - Target Volume Name
    - Target Bucket Name
- Replication info
    - Syncer UUID
    - Syncer host address
    - The last replicated index
        - This will be updated whenever the Syncer sends an acknowledgment that the index has been fully replicated to the target bucket
    - Exception
    - Time last sync
    - Last Seen Timestamp

The Syncer also needs to persists certain information locally

- The unique UUID of the Syncer
- Syncer directory
    - Parent directory: /hadoop/ozone/replication/

    - Directory: “<source_om_service>-<source_volume>-<source_bucket>-<target_om_service>-<target_volume>-<target_bucket>”
    - Contains
        - The persistent queue of the pending replicated log entries
        - META file
            - UUID of the Syncer
            - SourceBucket
            - TargetBucket
        - Checkpoint information
            - Replication Stage
            - OM Node ID
            - Initial Batch Replication
                - Current Snapshot
                - Last replicated key
            - Incremental Replication
                - Previous Snapshot
                - Current Snapshot
                - Last snapdiff
                - Last processed snapdiff key
            - Live Replication
                - The last index replicated (for)
        - LOCK file (in_use.lock)
            - To ensure only one Syncer runs per bucket replication pair

Configurations

Below are the relevant configurations

- OM
    - Maximum pending sync log entries: The maximum pending log entries that has not been to the Syncer
        - This is used as the backpressure mechanisms to prevent overloading both Syncer / Target bucket and the source OM

Resources Overhead

These are the extra resources needed to enable the bucket replication

- Machines to host Syncer processes
    - Memory and disk required to host the Syncer’s Persistent Queue
- Additional machine for OM listener if we decides to listen from the listener

Extensions

These are the possible extensions in the future.

Separate Replication Subsystem Master with HA

- In the future, we can make a master worker patterns for the replication subsystem to increase the data transfer throughput
    - This is addressed in https://docs.google.com/document/d/1EFJI6l76sIwwK7_f5Hs_tgt3mELXTIP0XJWpJKUkEy4/edit?tab=t.0#heading=h.byww9e6uslm9
        - The Syncer process will be run as a long-lived job worker
    - Master will act as the control plane on the existing bucket replications
        - We can use ZK (like in HBase) or it can be a new master process
        - Master can be a Raft quorum to ensure HA
    - Work assignment strategies
        - Option 1: Master will assign replication works to the registered worker nodes
        - Option 2: Workers request work from the master (master will assign work)
            - Based on some Token Pool
    - We can use lease to ensure that only one Syncer can run at the same time
- The master worker pattern can be adapted from the OSS distributed job management systems
    - Alluxio Job Service : https://docs.alluxio.io/os/user/stable/en/overview/Architecture.html#job-masters
    - Apache DolphinScheduler: https://dolphinscheduler.apache.org/en-us
    - Netflix Maestro: https://github.com/Netflix/maestro

Fast DR Failover

We can provide a DR failover mechanism so that clients can start writing to the target bucket in case the source cluster is down. For federation, this assumes that the ZK supports DR

This section highlights the SOP required to switch from source bucket to the target bucket

- If the source cluster fails
    - OFS client
        - Prerequisites
            - The client configuration should contain the OM service of the source and the target bucket
        - If federation is enabled, the Ozone administrator will update the mount table mapping in ZK to the target cluster
            - After the mount table mapping has been propagated to the client, the client can start working on the target bucket
        - If federation is disabled, the client needs to change the ofs service from the source cluster to the target cluster
    - S3 Client
        - If the S3G federation is configured with both the source and the target cluster

                - The client can switch to the target bucket name
            - If the S3G is not configured with both the source and the target cluster
                - The client needs to switch to the target cluster S3 endpoint
- If the source cluster recovers
    - OFS client
        - Prerequisites
            - The client configuration should contain the OM service of the source and the target bucket
        - If federation is enabled, the Ozone administrator will update the mount table mapping in ZK to the source cluster
            - After the mount table mapping has been propagated to the client, the client can start working on the source bucket
        - If federation is disabled, the client needs to change the ofs service from the target cluster to the source cluster
    - S3 Client
        - If the S3G is configured with both the source and the target cluster
            - The client can switch back to the source bucket name
        - If the S3G is not configured with both the source and the target cluster
            - The client needs to switch to the source cluster S3 endpoint
    - Backfilling
        - The data created in the target cluster can be backfilled using “hadoop distcp -update -diff” or basic “hadoop distcp”

Rate-limiting

- We can throttle the data replicated at one point of time by calculating the data size of keys to be replicated to ensure that the target cluster traffic is not spiked
    - We can use a DataTransferThrottler
    - This can be a part of the replication info stored in OM

Handling time difference between Regions

- Currently Ozone key uses the Time.now() / System.currentTimeMillis() instead of UTC
- Therefore, if we are migrating from SG -> US we need to normalize the modificationTime and creationTime to match the US timezone
- Tiimezone rollover: https://github.com/OpenHFT/Chronicle-Queue/blob/ea/docs/timezone_rollover.adoc

Syncer and OM co-location

We can add a mechanism to the Syncer and OM that are co-located in the same machines. Syncer just needs to get the snapshot / WAL path from the colocated OM. It will be similar to the

ongoing Short Circuit-Read (https://issues.apache.org/jira/browse/HDDS-10685) which uses domain sockets.

See https://stackoverflow.com/questions/54179843/how-to-create-a-grpc-service-over-a-local-socket-rather-then-inet-in-scala-java/54189910#54189910

Bidirectional Replications (Active-Active) & Complex Replication Topology

Handling Replication Loops

If the replication topologies forms a loop (e.g. a -> b -> c -> a), there will be infinite replications which we want to avoid

- Requires a logic to ensure that there are no infinite loop
    - For example, by tagging each change event by the cluster ID that it has encountered
        - Similar to HBase
        - The idea might be to add a region ID to each WAL entry and skip when there are some cycles detected
- This requires conflict resolution mechanisms (for objects with the same name), which can be handled by
    - Bucket versioning to be enabled (such as in AWS S3 / Minio bucket replications)
        - Since each object will have a unique version ID, replication simply create a new object version
        - However, this will add more overhead of storing multiple versions
        - Additionally, Ozone has not implemented object versioning
    - Custom conflict resolution
        - For example, Based on larger modification time

Conflict resolution for bidirectional replications

For Active-Active Replications, there needs to be conflict resolution mechanisms for objects with the same name. There are a few ways to handle this

- S3 versioning: If key versioning is enabled, conflict resolutions are not needed since each version has a unique version ID
    - In Minio and AWS S3, bucket replications requires both source and target buckets to enable versioning
    - Since each object will have a unique version ID, replication simply create a new object version
    - Eventually consistent: Meaning that eventually, both keys will have the same versions
    - However, this will add more overhead of storing and managing multiple versions
        - For example working with object version delete markers

    - Also HCFS (Hadoop-Compatible File System) has no concept of file versioning, which might cause some unpredictable behaviors
        - Which version should be shown to the user? Which version is the latest version?
    - Ozone has not supported key versioning yet and therefore requires some conflict resolution strategies
- Custom conflict solution strategies
    - Time-based conflict resolution
        - Similar to Couchbase XDCR
            - https://docs.couchbase.com/server/current/learn/clusters-and-availability/xdcr-conflict-resolution.html
        - It requires clocks between servers are synchronized (or at least the difference is not too large)
            - Need to confirm whether the NTP between different DCs are different
        - Since key modification time is recorded in local time, need to normalize the times during the bidirectional replication
    - Sequence ID based resolution
        - If there is a way to implement a perfectly ordered ID, we can pick the key with the higher ID
        - However, this requires a Globally Distributed Sequence ID Generator, which is out of the scope of the design
- CRDT
    - Need to do more research and see the feasibility over WAN
        - Garage (https://garagehq.deuxfleurs.fr/documentation/design/) uses CRDT, but haven't really delved into it yet

Support granular object replications

Support only replicating keys under certain prefix(es).

This requires some thought since for near-realtime tailing, the changes on other prefixes will be skipped. Therefore, if we add another replication config under a prefix, we might need to create a new bucket replication workflow for the prefix.

S3 Bucket Replication API Integration

We can try to add the following migrations

- Bucket replication API
    - PutBucketReplication
        - https://docs.aws.amazon.com/AmazonS3/latest/API/API_PutBucketReplication.html
    - DeleteBucketReplication

        - https://docs.aws.amazon.com/AmazonS3/latest/API/API_DeleteBucketReplication.html
    - GetBucketReplication
        - https://docs.aws.amazon.com/AmazonS3/latest/API/API_GetBucketReplication.html
- Get replication status information
    - Support x-amz-replication-status in GetObject and HeadObject
    - https://docs.aws.amazon.com/AmazonS3/latest/userguide/replication-status.html

Bucket Replication only needs to send one replica to the target bucket

To speed up the bucket replications throughput, we can allow Syncer to upload to one replica in the target cluster. The target cluster will use its ReplicationManager to replicate to the other two nodes at its own time.

Once StoragePolicy and StoragePolicySatisfier (SPS) is supported, this can be made possible. First the Syncer will upload the source key to the target bucket with a RATIS/ONE replication. And then StoragePolicySatisfier will asynchronously migrate these keys to RATIS/THREE.

Support reassignment of subscribed OM node in case of restarts / failure / slowness / migration

In case of OM listener / follower failure / slowness, the master might optionally pick another OM node (preferably another OM listener) to be the new preferred OM. This can be implemented by using the Ratis Event API (notifyConfigurationChanged, notifyLeaderChanged, etc).

The master might need to instruct the existing Syncers in workers to start listening from a new OM. Another use case would be if we only have followers and we want to migrate all the active subscriptions from an OM follower when it is promoted to a leader.

This is also useful for normal restarts or during OM bootstrap and decommission. However, we also need to balance not excessively migrate the existing worker to listen to the new

Limitations

The following are the current limitations and possible issues regarding the current solution

- Currently only LEGACY and OBS buckets are supported
    - FSO bucket requires extra key conversion logic (removing parent ID)
    - It is possible as long as we can derive the keyName of the FSO, after that it's uploaded as per normal
        - FileId and ParentId will be different from the source file

- For near-realtime replication, the resource is limited in on the machine running the Syncer that's subscribing the OM changes

Cross-cutting concerns

Security

- Currently, only Ozone administrator is allowed to start bucket replication
- In the future, we can allow bucket owner to start the bucket replication
- Prevent snapshots created by the Syncer from being deleted before the data has been replicated to the target bucket
    - Deleting them prematurely might cause unexpected issue during migration

Observability

- OM
    - Next log entry index to replicate
    - Replication lag: The elapsed time since the last log entry to replicate was read by Syncer and effectively replicated to target
    - Pending replications
- Syncer
    - Size of Replication Queue
        - The size of the replication queue for this Syncer
    - Timestamp started: Data of when this Syncer started
    - Number of Log Entries from queue
    - Number of Log Entries replicated
    - Number of Log Entries ignored
        - In case of duplicate sends from OM
    - Replication latency: Time taken to replicate the changes to the target bucket

Compatibility (Rolling Upgrade)

- Upgrade: Ensure that no active bucket replication is running during the upgrade
- Downgrade: Remove all the active bucket replication components
    - Stop all Syncers