Ozone Write Streaming Pipeline

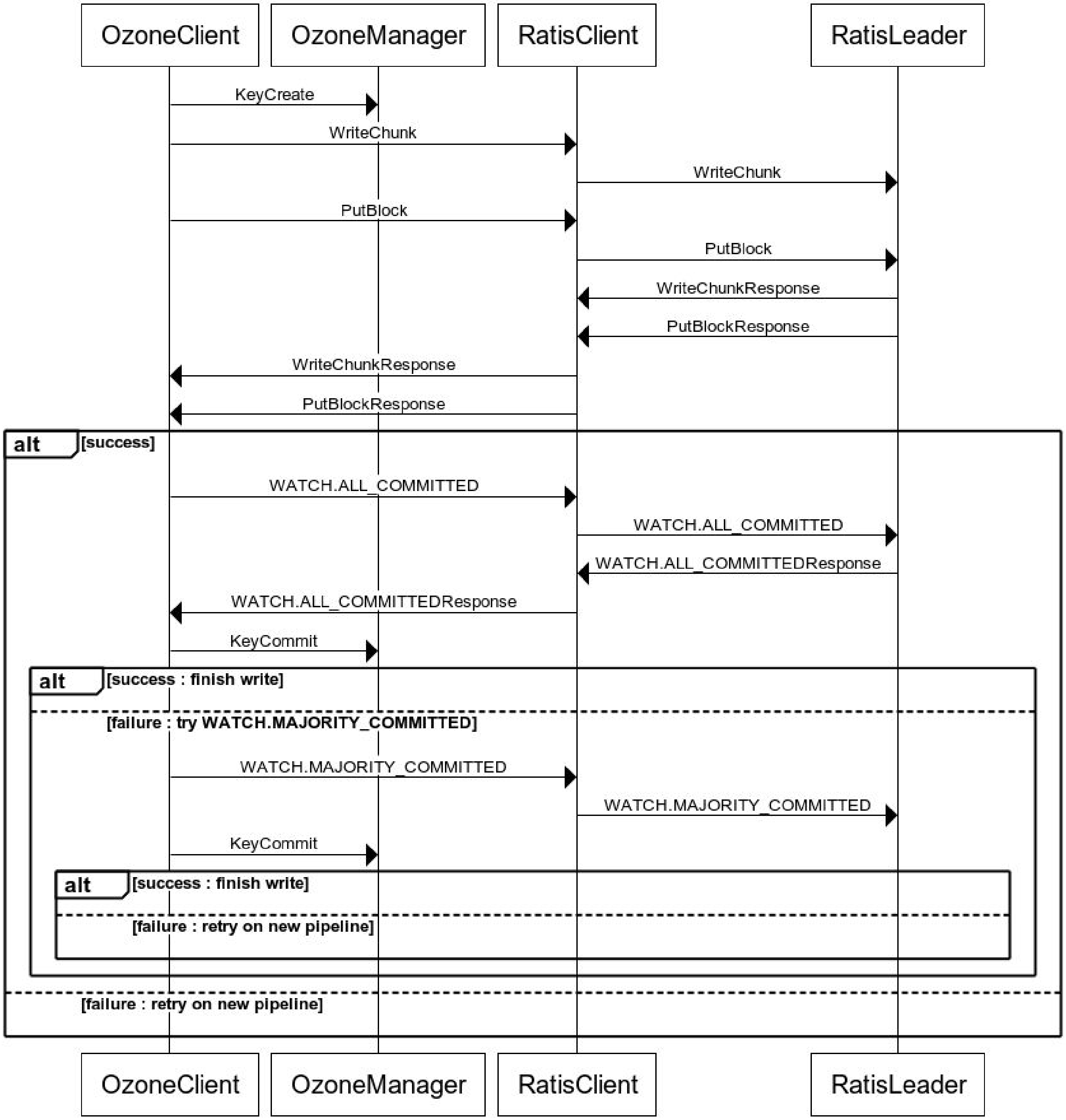
Current Write Path

Ozone current write pipeline involved a ratis 3 node raft group with one being the leader , others acting as followers. Currently, ozone client first initiates a call to Ozone Manager to create a file/key, the allocated blocks via SCM . Once a block is allocated and a write pipeline consisting of a ratis group is selected, ozone client starts writing to a client buffer in units called chunks. These chunks are backed by buffers equal to 4MB by default Once the buffers get full, a rpc call to initiate the write to data nodes hosting the ratis server instances is made by a ratis client.

The client also makes a putBlock call to update the block length to the datanode at every flush boundary/block close to commit length at every flush boundary(64 MB) by default.

Every write chunk/putBlock call is converted to a transaction log in ratis leader which is then appended to the followers. Ozone clients wait for data to be replicated to all 3 or at least 2 of before throwing away the client buffer(WatchForCommit).

Write sequence using Ratis



www.websequencediagrams.com

Issues with current Ozone pipeline

1) Currently, ozone writes to data nodes in a sort of bursty pattern. Ozone client waits for buffer size equals to chunk size(4MB) by default to even initiate a write chunk rpc call to Datanode. When the next buffer of 4MB gets written, the next write chunk rpc call will be initiated

2) Each rpc call to the ratis server running within the datanode process for writes and putBlock gets converted to a transactional log entry which needs to be synced to disk. Reducing the no of transactions significantly will reduce the time spent in raft log syncs.

3) With more no of transactions, more no of ratis snapshots will be taken as these are configured based on no of raft log transactions. Taking a snapshot involves another sync point in the write path as well.

4) In multiRaft setups, where a datanode becomes a leader for multiple groups its part of, tests show that the multileader node itself becomes the bottleneck.

5) There is a lot of buffer copy overhead during the transferring of data from ozone client buffers to datanodes.

Proposed Solution

The proposed solution here is to use Ratis Streaming in the Ozone write path(https://issues.apache.org/jira/browse/RATIS-979). The below proposals talk about using ratis streaming with zero additional buffer copying involved with data transfer from client to server either in data path only or both.

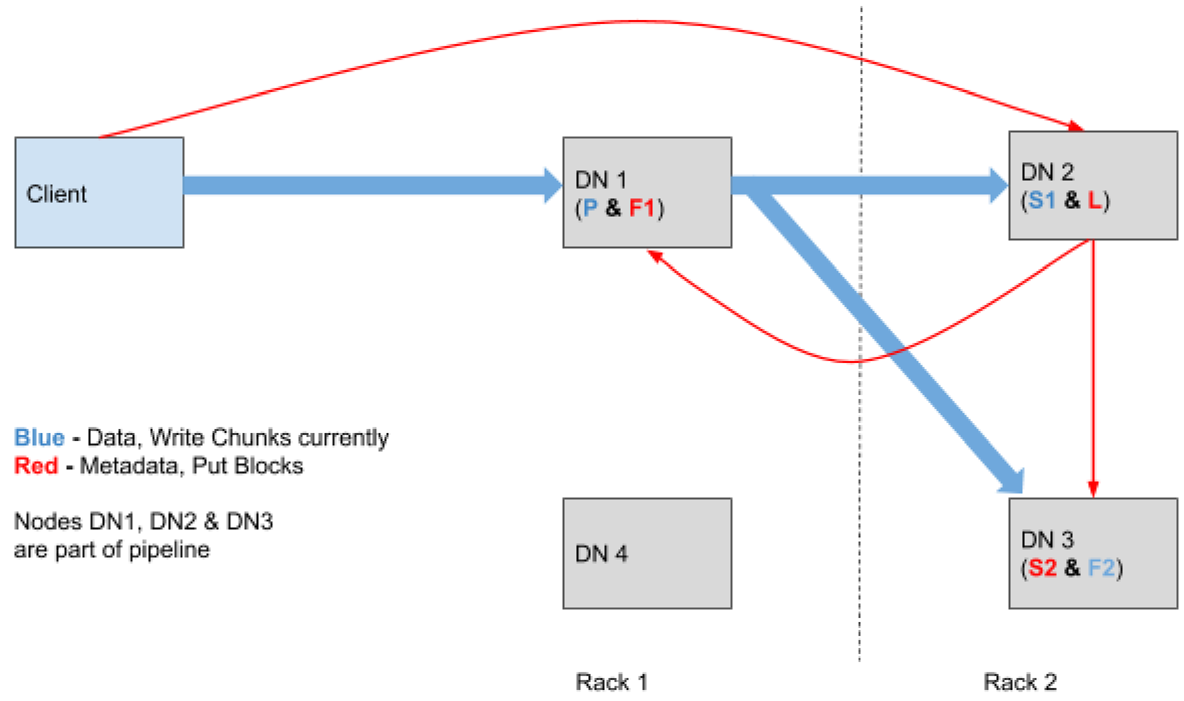
Advantages

- Data can be written in smaller units like 64KB than the current 1MB(currently).
- Reduces memory footprint on the leader because of smaller packets
- Clients can stream data through the closest node in the pipeline
- Caching on the Ratis Leader is eliminated because the data is streamed.
- Reduce no of raft log transactions created hence less no of raft log sync calls
- Reduce no of disk sync calls with lesser no of ratis snapshots.

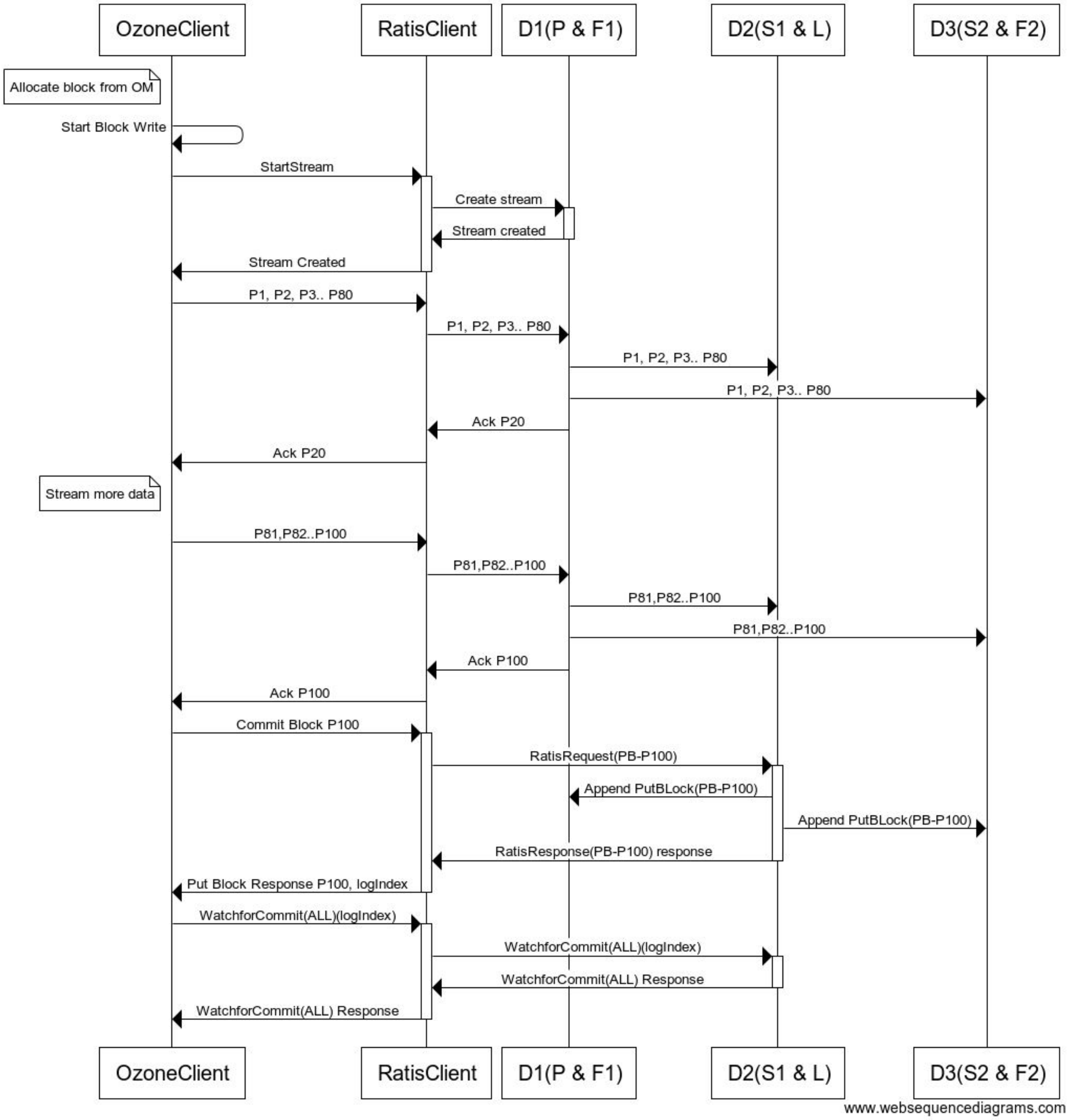
Approach 1: Separate Data and Metadata path

With the new data and metadata separation, the data will be written on the path in blue, where the Packets(P) in the pipeline are written to the closest node wrt the client in the pipeline.

The file data will be committed to the data node via PutBlock operation.



New Pipeline Write Data Path



a) Ozone clients will initiate connection with the primary datanode d1(as well as follower1) with d2 being the leader and other follower node being d3.

b) It will start sending packets to ratis client stream till packet 100(just for assumption) . The stream packet acknowledgement will be processed in a different thread.

c) The block length committed will keep on getting updated as per the stream msg ack

d) After packet 100, it will initiate close stream action and initiate a commit block request to leader D2.

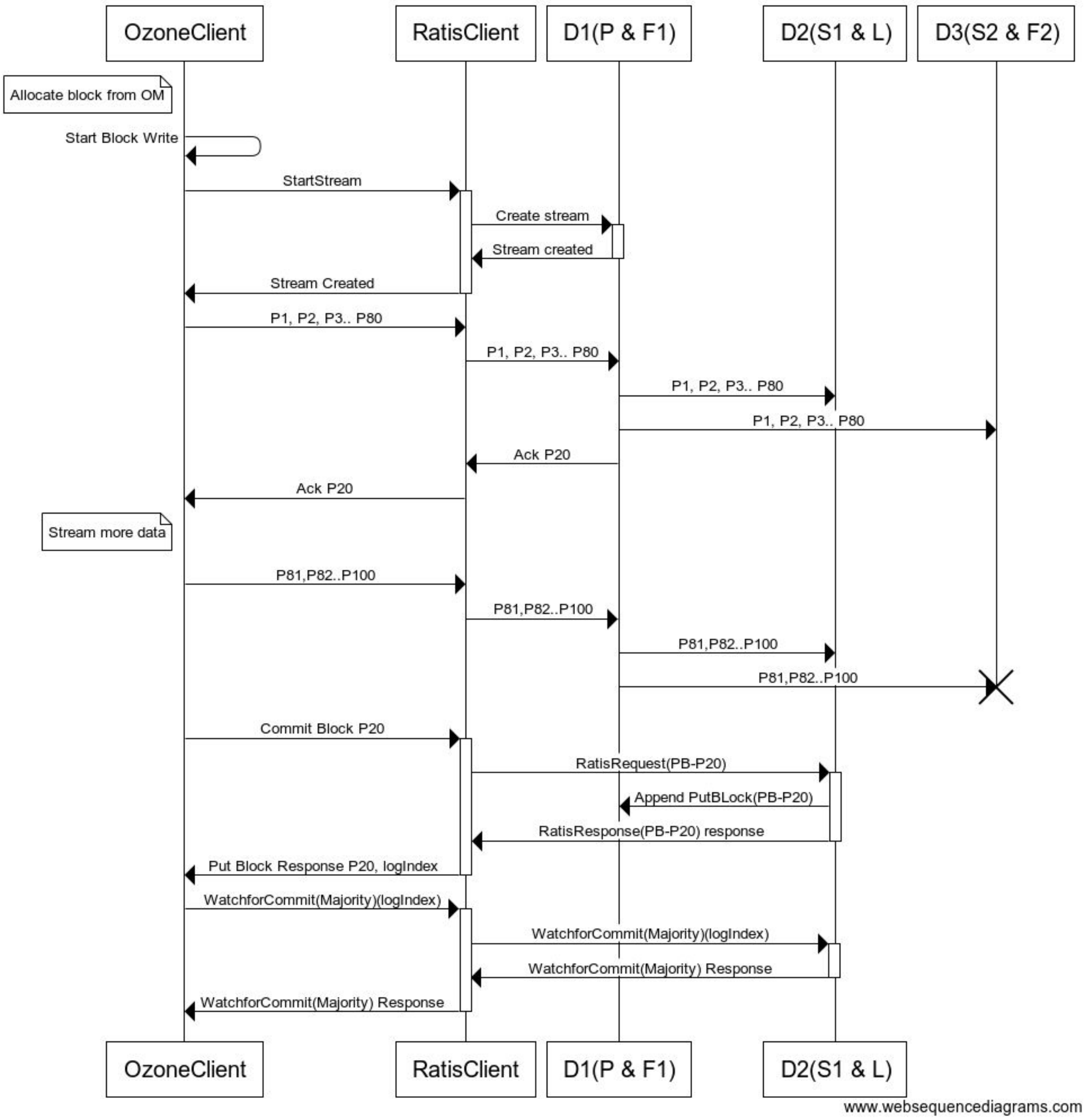
e) Once the commit block is received on the leader D2, it will create a raft log transaction and append to its own log and returns the response to primary datanode d1. It will wait for the append entries response from at least one of the followers.
f) D1 will acknowledge back to ratis client and thereby to ozone client as soon response from both D2 and D3 is received. The commit block response will have the log entry Index from the log transaction created on the leader.
g) Ozone client will then initiate a watch request on the leader for the commit block log index received in step e.
h) Watch on the leader will ensure implicit wait for the append to happen to both followers then complete the watch response from the leader
i) Once the watch is successful , client buffers are released.

Failure Handling

Stream Failure

Ozone client will keep track of the stream request responses with the length of the block getting updated per stream acknowledgement.
In case the stream close with the the putBlock request fails/timeout bcoz one of the nodes in the pipeline is dead, ozone client will try to commit the last committed block length by creating a putBlock request with the last ack'd length from the stream channel and send the request via ratis client to ratis leader by ratis log transaction path.
Since the stream has failed with one node not sending the ack back, it will just do a WATCH(MAJORITY_COMMITTED) for the put block log index.

New Pipeline Write Data Path on Error

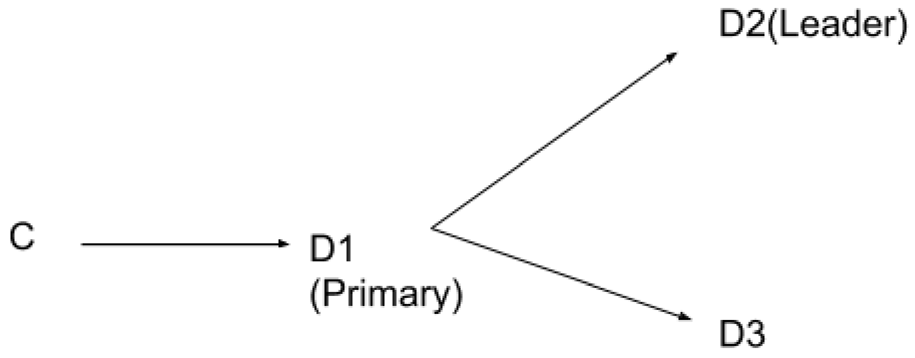


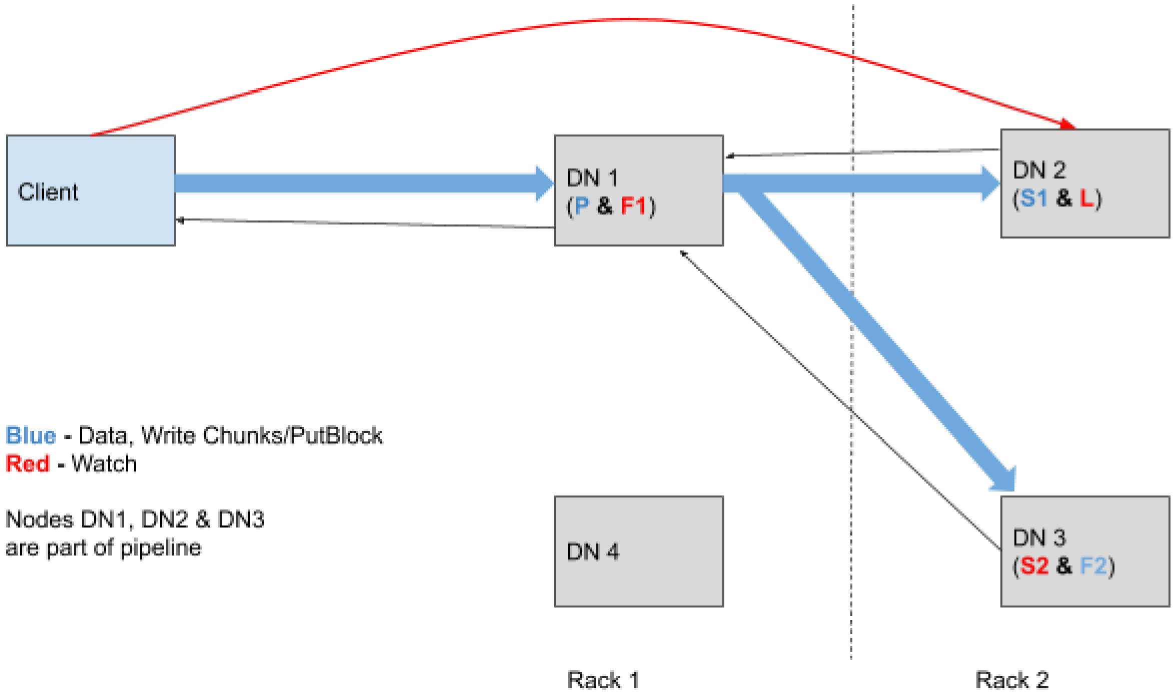
Primary Node Failure and 2 Node failure while streaming/WATCH

Get a new pipeline from SCM and start streaming the data again.

Approach 2 : Streaming Path for both Data and MetaData

The idea here is to use streaming for both Data and Metadata.





Pros

1. In case of using a streaming path for both data and metadata, Ozone client starts writing packets and does a putblock at the flush boundary , waits for the acknowledgment for PutBlock once the client buffer limit is hit and same process continues. While if we use separate metadata and data path as proposed, client

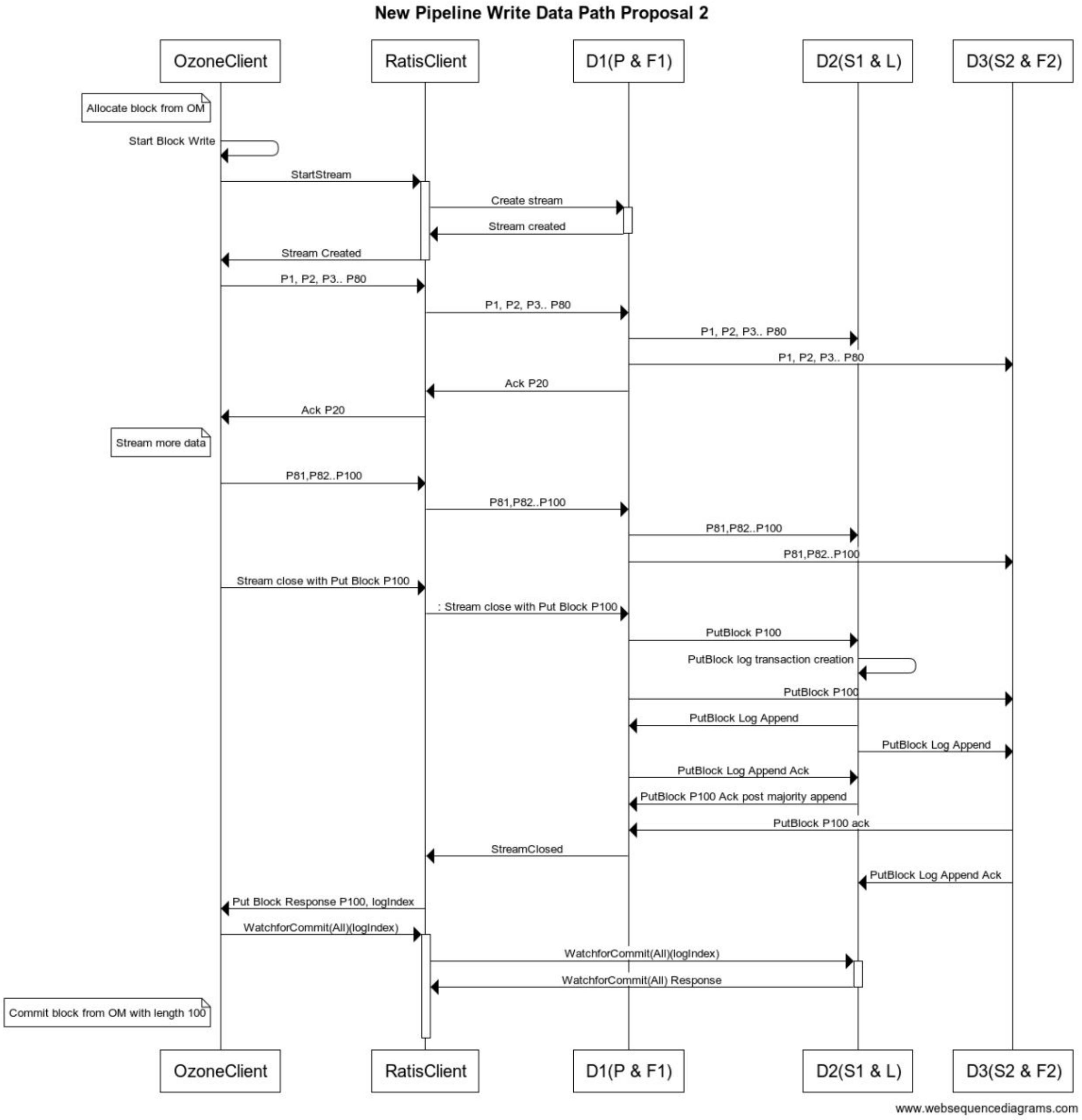
has to wait for the data write to finish by the streaming path , get ack back and then initiate the putBlock and wait for the ack back. So from Ozone client perspective, there are two blocking points when we use different data and metadata paths whereas there is one blocking point if we use the same streaming path for both data and metadata.

2. In a case, where streaming data packets also fail bcoz the one of the nodes in the pipeline (node being the leader is short on resources) doesn't respond, ozone client has to wait for first the write to fail after a timeout and then try to commit the block by sending the request to the leader which will again run into the same problem. It will double the total wait time here as well. Especially with multiRaft, Leader getting bottleneck is more likely.

3. Two different paths for data and metadata means failure handling cases will add up in general.

Cons

1. In case the commit block response is not received after the last data packet say at length x on the streaming channel, the commit needs to happen for the last acknowledged data length say (where x > y) on the leader by the usual raft protocol. In cases such as this, it may lead to a situation where transactions for the commit block for both lengths x and y can be created where log index corresponding to length x say tx < transaction log index for length y say ty. IN this case, a higher log index meaning higher BCSID corresponds to a lower length of a block leading to truncate semantic in container rocks db in datanodes. This needs special handling and is not handled with the current implementation.



j) Ozone clients will initiate connection with the primary datanode d1(as well as follower1) with d2 being the leader and other follower node being d3.
k) It will start sending packets to ratis client stream till packet 100 . The stream packet acknowledgement will be processed in a different thread.
l) The block length committed will keep on getting updated as per the stream msg ack
m) After packet 100, it will try to commit the block marking it as end of msg for the stream thereby initiating a close stream action.

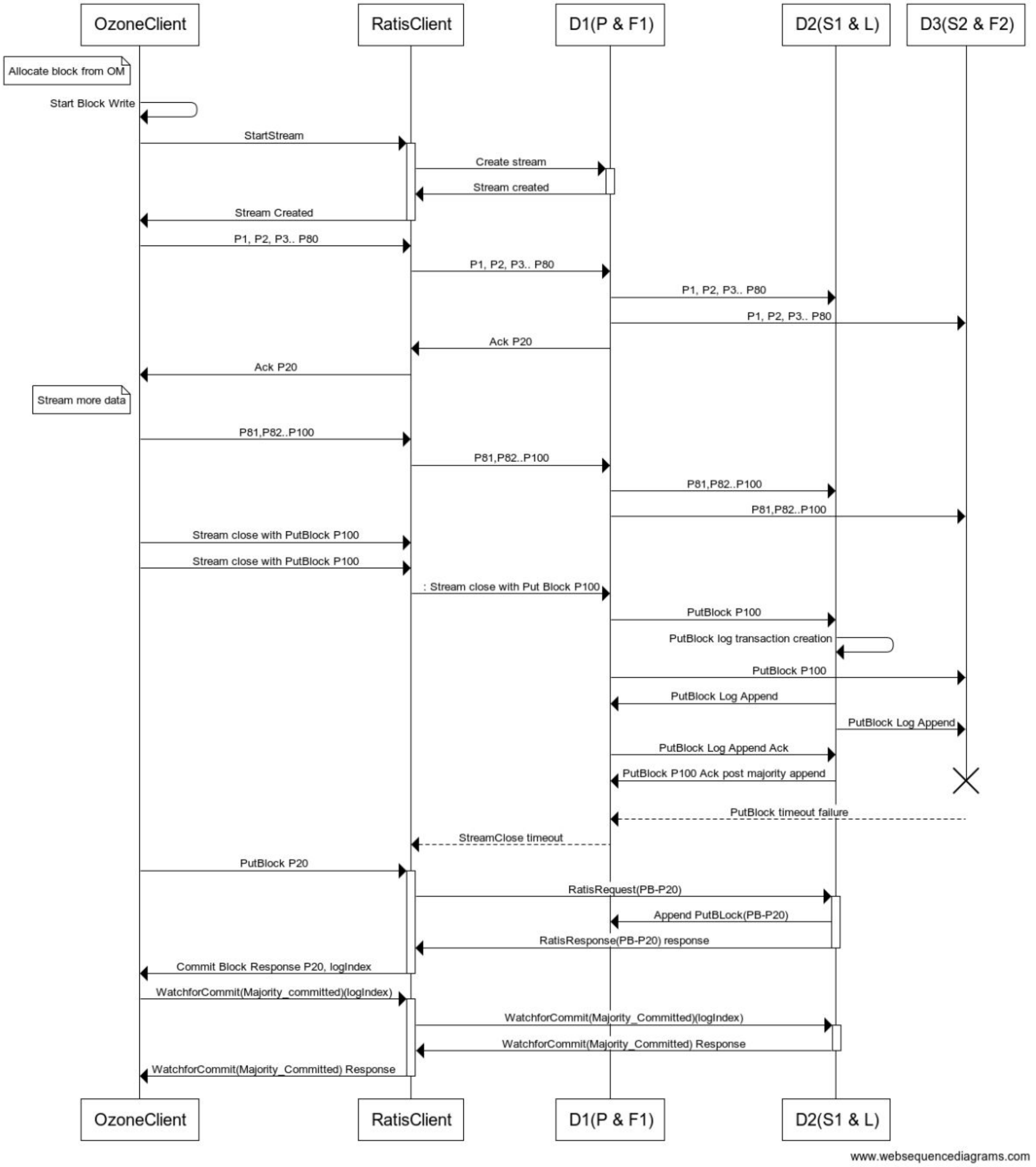
n) Once the commit block is received on the leader D2, it will create a raft log transaction and append to its own log and returns the response to primary datanode d1. It will wait for the append entries response from at least one of the followers.
o) D1 will acknowledge back to ratis client and thereby to ozone client as soon response from both D2 and D3 is received. The commit block response will have the log entry Index from the log transaction created on the leader.
p) Ozone client will then initiate a watch request on the leader for the commit block log index received in step e.
q) Watch on the leader will ensure implicit wait for the append to happen to both followers then complete the watch response from the leader
r) Once the watch is successful , client buffers are released.

Failure Handling

Stream Failure

Ozone client will keep track of the stream request responses with the length of the block getting updated per stream acknowledgement.
In case the stream close with the the putBlock request fails/timeout bcoz one of the nodes in the pipeline is dead, ozone client will try to commit the last committed block length by creating a new putBlock request with the last ack'd length from the stream channel and send the request via ratis client to ratis leader by ratis log transaction path. Since the stream has failed with one node not sending the ack back, it will just do a WATCH(MAJORITY_COMMITTED) for the put block log index.

New Pipeline Write Data Path Proposal 2 Stream Failure Handling



Watch Failure

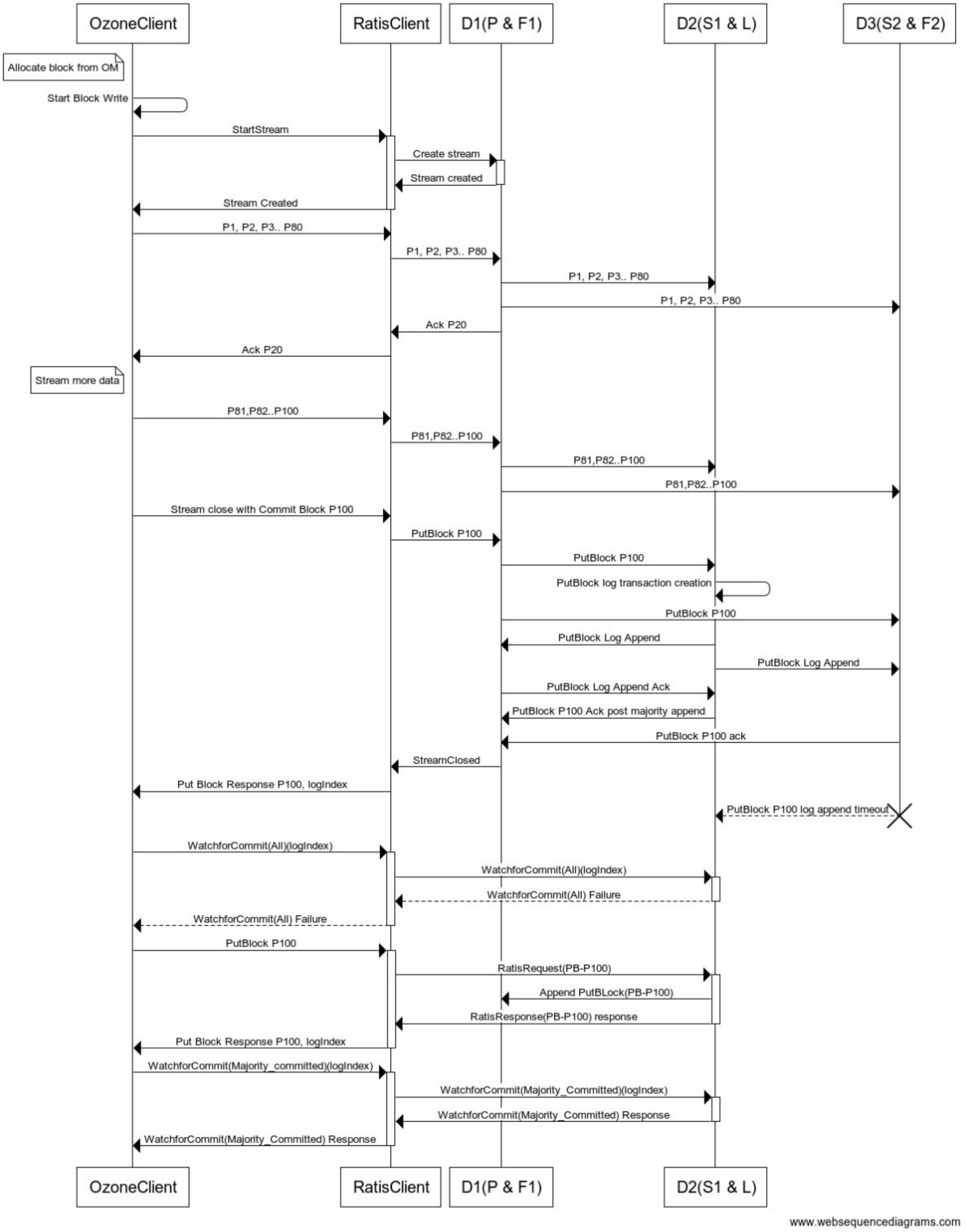
In this case, the stream close is successful but the next WATCH(ALL_COMMITTED) can potentially timeout as one of the nodes may fail after acknowledging back to the primary node but fail to append the last putBlock log from the leader. In such cases, WATCH will be retried with MAJORITY_COMMITTED for the log index of the putBlock for the last ack'd length (the length updated with stream close response)

NOTE:
With PutBlock happening over both the streaming channel as well as ratis transaction path in case of failure, these can potentially go out of order with respect to the block lengths in ratis log thereby creating 2 possibilities:

a) PutBlock for length l1 log index i1 > PutBlock for length l2 with index i2 where, l1 < l2 Where l1 is the length committed in OM, will result in generation of garbage.

b) **PutBlock for length l1 log index i1 < PutBlock for length l2 with index i2 where, l1 < l2 , where l1 is the length committed in OM, will result in truncation of length in Datanode for higher index. This case needs to handled in datanode as currently , this is not allowed**.

New Pipeline Write Data Path Proposal Watch Failure Handling


OzoneClient
RatisClient
D1(P & F1)
D2(S1 & L)
D3(S2 & F2)
Allocate block from OM
Start Block Write
StartStream
Create stream
Stream created
Stream Created
P1, P2, P3.. P80
P1, P2, P3.. P80
P1, P2, P3.. P80
P1, P2, P3.. P80
Ack P20
Ack P20
Stream more data
P81,P82..P100
P81,P82..P100
P81,P82..P100
P81,P82..P100
Stream close with Commit Block P100
PutBlock P100
PutBlock P100
PutBlock log transaction creation
PutBlock P100
PutBlock Log Append
PutBlock Log Append
PutBlock Log Append Ack
PutBlock P100 Ack post majority append
PutBlock P100 ack
StreamClosed
Put Block Response P100, logIndex
PutBlock P100 log append timeout
WatchforCommit(All)(logIndex)
WatchforCommit(All)(logIndex)
WatchforCommit(All) Failure
WatchforCommit(All) Failure
PutBlock P100
RatisRequest(PB-P100)
Append PutBLock(PB-P100)
RatisResponse(PB-P100) response
Put Block Response P100, logIndex
WatchforCommit(Majority_committed)(logIndex)
WatchforCommit(Majority_Committed)(logIndex)
WatchforCommit(Majority_Committed) Response
WatchforCommit(Majority_Committed) Response
OzoneClient
RatisClient
D1(P & F1)
D2(S1 & L)
www.websequencediagrams.com

Primary Node Failure and 2 Node failure while streaming/WATCH

Get a new pipeline from SCM and start streaming the data again.

Approach 3

Same as proposal 2 but in this case, the stream close with putBlock will return as soon as the log transaction is created on the leader. It won't wait for the append log response from any of the followers to ack last PutBlock on the stream channel close. A subsequent watch request will implicitly wait for the append to both the followers.

But in this case, in case a node failure where the leader goes down without being able to send the putBlock append log to any of the followers after ack of the last put block on the stream channel will run into a situation where the last putBlock log entry can be truncated/not exist/or a different log entry on the same index, on the new leader on the subsequent watch. In such case, Watch request needs to be validated inside the container stateMachine first to verify if log entry corresponds to the putBlock request for the desired length. This needs some additional metadata (block length) to be passed in WATCH request from client which will be used to validate the log entry inside ContainerStateMachine.

Upgrades

No specific challenges as the rocks db update for block length or write chunk layout is not going to change. On disk layout is supposed to remain the same.

Pre Upgrade should ensure all open containers are closed and all and the pipelines are destroyed.