Ratis Streaming

2020.9.29

Problem Statement

In RAFT, clients send requests to the Leader. The Leader then forwards the requests as appendEntries to all the followers with a star topology. Such approach has the drawbacks below:

1. The Leader requires more memory to cache:
    a. outstanding client requests, and
    b. outstanding log entries and data for each follower.

   Also, for retrying, the Leader may cache the same request/log entry multiple times.
2. The concurrent requests are ordered, therefore if a client data is written to a slower disk, it can slow down other clients that are writing to other faster disks.
3. Streaming gives better performance than writing chunks at a time since, when a node receives incoming streaming data, the node can immediately stream the data to another node(s) without waiting for the incoming stream to close.
4. The Leader has more network traffic since
    a. the Leader directly receives all the requests from all the clients, and has multiple connections to manage.
    b. the Leader uses twice or more network bandwidth than followers for appendEntries.
5. It is not optimal for network traffic.
    a. Even if a client is near to a follower and far from the Leader, the client has to send requests to the Leader and then the Leader sends appendEntries back to the near follower.
    b. Two followers reside in the same rack but the Leader resides in a different rack.

   The worst case is when a and b happen together.

Proposed Solution: Ratis Streaming

We focus on group size of 3 in the discussion. The solution can be generalized to any group sizes.

The appendEntries is used to broadcast the client requests to all the machines. Instead of a star topology, use pipelines (or spanning trees). All the machines, the Leader and the followers, can directly receive requests from clients and then stream the requests to the other machines using a pipeline. For example, the Leader and Follower 1 reside in the same rack and Follower 2 resides in a different rack.

```
                      Rack 1                       Rack 2
Case 1:
        Client  -->  Leader  -->  Follower 1  -------------->  Follower 2

Case 2:
                  Leader  <--  Follower 1  <--------------  Follower 2  <--  client

Case 3:
                  Leader  <--  Follower 1  -------------->  Follower 2
                                    ^
                                    |
                                  Client
```

When the Leader receives a forwarded request, it processes the request as usual. When a follower receives the request, it sends an acknowledgement to the Leader and the Leader replies the acknowledgement with the log index.

In Case 3 above, we may use a different pipeline to avoid Follower 1 to stream requests to two machines.

```
Case 3’:
        Client  -->  Follower 1  -->  Leader  -------------->  Follower 2
```

An Alternate Solution “Ozone Pipelines” is discussed in the Appendix.

Implementation Plan

In this section, we describe a plan to implement the proposed solution - Ratis Streaming. The streaming work and the pipeline work are separated as Step 1 and Step 2 below.

Step 1: Streaming

In Step 1, we focus on streaming implementation without changing the Ratis star topology to pipelines (i.e. keep using star topologies) and without considering the network topology (i.e. no rack awareness). We solve Problems #1, #2 and #3 in this step and leave Problem #4 and #5 to the next step.

Plan 1: Stream to the Leader, and then immediately stream the data to the followers without creating Ratis transactions.

In this plan, all the clients stream requests to the Leader, and then the Leader streams the data to all the followers in parallel. When a client starts a stream request to the Leader, the Leader will immediately stream the data to each follower. Note that a Ratis transaction is not yet created during the client streaming the request. The Leader creates a corresponding Ratis transaction once the client has closed the stream.

```
Client  ------->  Leader  ------->  Follower 1
                     |
                     \------------->  Follower 2
```

Packet Acknowledgements

When a client streams packets to the Leader, the Leader will acknowledge each packet back to the client. The client is safe to discard the acknowledged packets from its buffer. Similarly, the Leader streams packets to all the followers and the followers will send acknowledgements back to the Leader. Note that the Leader will send an acknowledgement of a packet to the client only if it has received all the acknowledgements of that packet from all the followers.

As a result, a slow follower will slow down the entire process, and also slow down the client. This is a desired behavior to throttle the traffic.

Multiple Clients

Until a transaction is created i.e. at the close of the stream, all clients go in parallel so that they do not interfere with each other. Even if a client is slow or a client is writing to a slow disk (in a follower), it does not slow down other clients.

Leader Re-election

Since all the clients stream to the Leader, what happens if there is a leader re-election? How could the clients failover to the new Leader? We will discuss this in Plan 1.1 below.

Plan 1.1: Stream to a primary node, and then immediately stream the data to the remaining nodes without creating Ratis transactions.

In Plan 1, the Leader does not create transactions until the client streams are closed. It is unnecessary to choose the Leader to receive client streams. Based on this observation, we modify Plan 1 as Plan 1.1 below.

In Plan 1.1, a client may stream to any one of the nodes, called a Primary node. Different clients may choose a different Primary node. The Primary node immediately streams client data to all the remaining nodes. When the Leader (may or may not be the Primary node) receives all the stream data (i.e. the stream is closed), it creates a transaction. In this plan, the client can continue streaming to the Primary node even if this is a leader re-election. If the Primary node fails, the client can failover to any one of the remaining nodes since the client has all the unacknowledged data in its buffer.

We discuss Plan 0.5 as an Alternate Implementation Plan for Step 1 in the Appendix.

Step 2: Pipeline

With a star topology, it is possible that the primary node may forward data to the remaining with additional cross traffic. Consider the following example, the client and Node 0 are collocated in Rack 1. Node 1 and Node 2 are collocated in Rack 2. Since Node 0 is the closest to the client, the client chooses it as the primary node to start with. Then, Node 0 will forward the data to Node 1 and Node 2 individually as shown below.

```
            Rack 1                                  Rack 2

   Client  ------->  Node 0 ----------------------------->  Node 1
                        |
                        \----------------------------------->  Node 2
```

Obviously, one of the cross traffic can be eliminated by using a pipeline such that Node 0 only forwards the data to Node 1 and then Node 1 forwards the data to Node 2.

```
             Rack 1                        Rack 2

     Client  ------->  Node 0 ------------------------------>  Node 1
                                                                 |
                                                                 V
                                                               Node 2
```

Step 2 is to replace the star topology by pipelines. Since clients may start with any one of the nodes in a Raft group, there are different pipelines in a Raft group depending on the starting node. Continuing with the example, we have Pipeline 0, 1 and 2 starting with Node 0, 1 and 2, respectively.

```
             Rack 1                                  Rack 2

Pipeline 0:  Node 0 ---------------------------------------->  Node 1
                                                                 |
                                                                 V
                                                               Node 2


Pipeline 1:                                                    Node 1
                                                                 |
                                                                 V
             Node 0 <----------------------------------------  Node 2


Pipeline 2:  Node 0 <----------------------------------------  Node 1
                                                                 ^
                                                                 |
                                                               Node 2
```

It only requires one cross rack traffic for all the pipelines above.

Appendix

Alternate Solution: Ozone Pipelines

Instead of steaming in Ratis, Ozone clients stream data to a tmp block location in each node.

```
Ozone Client  -->  Leader  -->  Follower 1  ------------->  Follower 2
```

Similar to HDFS, every Ozone client writes to a pipeline except that they write to the same set of nodes. Then, create a transaction into Ratis to commit the block.

When there are pipeline failures, the Ozone client will

- Take the last acked offset (Ack means it has been replicated to all 3)
- Commit that much length
- Destroy the pipeline

It needs a mechanism to replicate blocks between nodes since a node may not be available during client write. That node may join the group later on. Moreover, a block may be committed in Ratis even if some of the nodes do not have the block. (How should this case be handled?)

Is the Alternate Solution same as the Proposed Solution?

The main difference between two solutions is that the Proposed Solution is aware of Ratis but the Alternate Solution is not. The Alternate Solution has to handle the two cases below

1. The data may be missing during commit.
2. Replicate the data when some of the nodes do not have it.

Which transport mechanism should streaming use? gRPC? Netty?

In RATIS-997, we have benchmarked the performance of (1) gRPC with Protobuf, (2) gRPC with Flatbuffers and (3) Netty. We found that Netty has the best performance -- it has ~5x better performance than the others, where gRPC with Protobuf and gRPC with Flatbuffers have roughly the same performance. The reason is that Netty supports zero buffer copying but the others do not. Therefore, we will use Netty for the first Streaming implementation.

Can we reuse the DataTransferProtocol implementation from HDFS?

Yes, it may be a good idea since the code from HDFS is already tested extensively. However, we may as well improve it, noticeably the threading model (In DataTransferProtocol). Both the Proposed Solution and the Alternative Solution can reuse the HDFS code. Unfortunately, HDFS code supports neither zero buffer copying nor asynchronous event driven. It can be anticipated that streaming with Netty will outperform it.

Alternate Implementation Plan for Step 1

Plan 0.5: Stream to the Leader, create a transaction and then stream appendEntries to the followers.

In this plan, the clients stream requests to the Leader. The Leader collects all the data and waits until the stream is closed. Then, the Leader creates a Ratis transaction and stream appendEntries to the followers instead of sending a log entry (data + metadata) in a big chunk.

Unfortunately, Plan 0.5 only solves Problem #3 but not Problems #4 and #5.

We only state Plan 0.5 for discussion but not considering implementing it.