ClusterMembership

Cluster Membership: ZOOKEEPER-107

1. Motivation

ZooKeeper clusters are currently static: the set of servers participating in a cluster are statically defined in a configuration file. In many instances, however, it is desirable to have the ability of adding and removing servers from an ensemble. The difficulty of implementing such a feature is making sure that a change in the configuration does not cause inconsistencies in a ZooKeeper cluster. A related issue is the one of enabling a client to learn the current ensemble of servers dynamically.

2. Requirements

1. The ensemble of servers must agree upon the current configuration, and to reach agreement, Zab sounds like the obvious choice;
2. We need new client calls to add and remove servers. It is unclear whether we want one call for each modification or one call to propose a whole new configuration;
3. It must work with both majority and flexible quorums;
4. We need a mechanism, perhaps based on URIs, to enable a client to learn the current configuration.

3. Some pre-design random thoughts

When moving from one configuration to another, we need to make sure that a quorum of the old configuration and a quorum of the new configuration commit to the new configuration. A quorum of the old configuration needs to agree to avoid a split-brain problem, for example, when adding more servers. A quorum of the new configuration needs to agree for progress. We also need to make sure that a quorum of the old configuration confirms first, otherwise a partition could cause a split-brain problem.

If the current leader is part of the old and new configurations, then it can keep being the leader once the new leader is installed. Otherwise an epoch change becomes necessary.

It is critical to make sure that every operation committed once a configuration is installed is acknowledged by a quorum from the new configuration. Otherwise a leader crash can cause committed operations to be lost. It might be simpler to stall the pipeline of request processors when a reconfiguration goes through PrepRequestProcessor. By stalling I mean holding operations until the reconfiguration operation is committed.

4. Proposed Algorithm

This section contains a proposed algorithm and API for reconfiguring zookeeper cluster membership.
Any comments and suggestions are welcome (Alex Shraer, shralex@yahoo-inc.com).

Let M be the current configuration of Zookeeper. We define a configuration to be the set of participating servers, members(M), and a quorum system over these members. In order to reconfigure the system to a new configuration M’ an administrative client submits a reconfig(M’) operation through any member s in M. This causes s to:

1. Send a message to all members(M’) instructing them to connect to leader(M) as followers.
    - Followers learn about operations as they are proposed (not upon commit) and do not participate in leader election.
2. Wait for <connected> from a quorum of M’ (optional, see line 1 below)
3. Submit a reconfig(M’) operation to leader(M)

The algorithm for leader(M) is as follows:

- operation reconfig(M’)
    1. If no quorum of M’ is connected or the connected quorum is too far behind M return failure
    2. If leader(M) is in M’, designate leader(M) to be leader(M’)
    3. Otherwise, designate the most up-to-date server in connected quorum of M' to be leader(M’)
    4. Schedule the reconfiguration last in the queue of operations
       Phase 1:
    5. New operations (received by leader(M) during phase-1) will be sent to both M and M', s.t.:
        1. No commits are sent by leader(M) to members(M')
        2. Client submitting the operation can be acked as soon as one of the following happens:
            - a quorum of members(M) acks the operation
            - phase-2 completes
    6. send <reconfig-start, version(M), M’> to members(M)
    7. wait for <reconfig-start-ack> from quorum of members(M)
       Phase 2:
    8. Stop processing new operations, return failure for any further ops received
    9. send <activate-config, M’, leader(M’) > to members (M’)
    10. wait for <activate-config-ack> from quorum of members (M’)
       Phase 3:
    11. send <retire M> to members(M)

All servers do the following:

- upon receipt of <reconfig-start, version(M), M’>

    - store next(version(M)) = M’
    - send <reconfig-start-ack> to leader(M)

- upon receipt of <activate-config, M’, leader(M’)> in configuration M’
    - forward the message to members(M')
    - send <activate-config-ack> to leader(M) and to leader(M')
    - A non-leader server can start processing client operations in M’

- upon receipt of <activate-config, M’, leader(M’)> from leader(M) and <activate-config-ack> from quorum of members(M'), leader(M') does the following:
    - commit all operations sent by leader(M) to members(M') during phase-1.
    - start processing client operations in M’

- upon receipt of <retire, version(M)>
    - garbage-collect M

4.1. Notes:

- State transfer starts when members(M’) connect to leader(M), before reconfig(M’) is invoked. However, some suffix of operations might not have been transferred to M’ before the reconfiguration began. Since no new operations can be committed in M’ before this state-transfer completes, we’ d like to minimize this tail. A possible way to do that is to require that at most w operations scheduled before the reconfiguration are unknown to the connected quorum of M’ as a prerequisite (line 1).
- To prevent a violation of Global Primary Order, we can't have leader(M) committing operations in M', but it can propose the operations and leader (M') can then commit these operations.
- Operations arriving to leader(M) during phase-1 are sent to both M and M'. Because leader(M) can fail during phase-1, we must ensure that either a quorum of M got the operation or (a quorum of M' got it AND M' was activated). Completion of phase-2 indicates the latter. Having said that, acking the client is problematic after phase-2: currently, ZOOKEEPER only allows a server to be connected to one leader and those servers in M that also belong to M' will no longer be listening to messages from leader(M) after receiving the phase-2 message. ZOOKEEPER-22 currently prevents clients from finding out the status of their operations submitted in M by connecting to M'. We should discuss how to resolve this point.
- If phase 1 is done using a normal ZAB proposal, explicitly making sure that there are no incomplete reconfiguration requests that remain in the system after a recovery might not be necessary.
- phase-3 is not needed for correctness, however, if executed, it is important to execute it after phase-2 completes, since we want to make sure that M' is active before M goes away (otherwise the system can get stuck).

4.2. Recovery from leader failure

During state discovery in M, if some server responds next(M)!=null, let M’ be such returned non-null configuration. The algorithm (or ZAB) will make sure that at-most a single non-null value is returned.
The leader executes the reconfiguration alg. with the following changes:

- Instead of line 1, try to connect to M’ and transfer state, so that a quorum of M’ is connected and up-to-date. If unsuccessful, skip the reconfiguration.
- If a quorum of M’ indicate that they’ve already activated M’ then skip to phase 3
- Otherwise, if next(M) = M’ was returned by a quorum of M, skip to phase 2

The server s that submitted the reconfig operation to the leader should check what's the current configuration after a new leader is established. If it is still the old one it can re-submit the reconfig. If s fails, the client can re-submit the reconfig to another server. Recall that version(M) is included by the client in the request so a server receiving it should first check that the configuration is still M (using local state) before submitting it to the leader, and the leader should also make this check (since it has the latest state).

4.3. Reconfiguration API

A choice that has to be made is what kind of operations to support – incremental changes like “add server A” or “remove server B” (e.g., as in survey on reconfiguration of R/W objects , #DynaStore) or full membership specification, as in “reconfigure so that the membership becomes is A, B, C” (e.g., survey on reconfiguration with virtual synchrony , #Rambo).

One notable disadvantage of non-incremental requests is when multiple reconfigurations are proposed concurrently. Suppose that the initial configuration is A, B, C and one process proposes to add a server D whereas another proposes to remove B. If each process has to specify the full new configuration then the first process would propose A, B, C, D whereas the second would propose A, C. One of these would succeed first, suppose A, C. Then the second proposal should be aborted otherwise the resulting configuration would be A, B, C, D, where B appears even though it was already removed.

With the incremental approach this would not happen. Here, a configuration can be viewed as a set of changes. The initial configuration is (+A, +B, +C) then in the scenario above the next configuration is (+A, +B, +C, -B) and last one is (+A, +B, +C, -B, +D). This allows all reconfiguration requests to complete without aborting (a wait-free algorithm), as each change can be applied to the current configuration, whether this configuration remained the same or changed.

The non-incremental approach might be preferable for two main reasons:

- It is easy to completely change the quorum system of the new configuration (with the incremental approach It is perhaps possible to support that as a separate command, or automatically deduce the quorum system from a given set of members).
- One can argue that, in the scenario above, adding D was intended in the context of the original configuration A, B, C, and it should be left to the client whether the reconfiguration makes sense given that the system configuration has changed.

To support the non-incremental approach, we propose that the administrative client intending to execute a reconfig, first fetches the current config from the system, and then submits a reconfig request where he fully specifies the requested configuration, its quorum system, and includes the version of the current configuration (version(M)). If the server that gets this request already has some other proposal to reconfigure (next(version(M))!=null) the reconfiguration is aborted and the administrative client should decide whether to retry.

At first stage we propose to use the non-incrememtal API for reconfigurations. In the future we intend to use this non-incremental interface only for changing the quorum system and to add an incremental API for wait-free reconfigurations.

4.4. Old reconfiguration requests

Suppose that a reconfig request was issued and the leader started sending phase-1 messages to the current configuration M, but failed after sending to only one other server A. Then, when the recovery happened, the new leader did not see a message from A. Should we allow the reconfiguration request to surface at a later time ? If not, a possible solution might be to have a command “next(M) = null” as the first one issued by any elected leader. If ZAB is used for sending the message in Phase 1, explicitly making sure that there are no incomplete reconfigurations that can surface later might be unnecessary.

4.5. Online vs. offline reconfiguration

The idea of an “off-line” strategy for reconfiguration (survey on reconfiguration with with virtual synchrony , survey on reconfiguring state-machine replication) is to stop operations in the old configuration, transfer the state to the new configuration and then enable operations – in the new configuration. In contrast, an online reconfiguration approach (#RAMBO, #DynaStore) never stops the service while reconfiguring.
One of the complexities arising in the online approach is that a normal operation can be executing concurrently with a reconfiguration, however the state still must be transferred correctly to the next configuration. The easy case is when the operation occurs in the old configuration and the reconfiguration transfers the state. It is possible, however, that the reconfiguration misses the operation when it transfers the state and completes. In this case, existing online reconfiguration solutions (#RAMBO, #DynaStore) continue the operation and execute it in the new configuration.
In Zookeeper, we must be careful not to violate the global primary order property - a situation where a new primary commits operations and only then the old primary's commit arrives is not allowed in ZAB. The algorithm avoids that by having leader(M') commit operations proposed by leader(M) during the reconfiguration,
and it does so before committing any other operations that it itself proposed in M'.

4.7. Other issues

4.7.1. Bootstrapping the cluster

We should be able to bootstrap ZOOKEEPER using reconfig API instead of configuration files as it is currently done.

4.7.2. Formal Liveness Specification

Should be similar to the one in survey on reconfiguration with with virtual synchrony and #DynaStore.

4.7.3. Informing clients and servers about new configuration

We should probably have a DNS-like solution as a fall-back for clients that were not around during the reconfiguration (suppose that all of the servers that the client knew are already down). For normal operation we should also have a push-base solution to notify clients about the configuration change. Some preliminary thoughts:

1. It is preferable to move clients from the old configuration to the new one gradually so that we don't need to set up hundreds of new connections simultaneously.
2. one possibility could be transferring client list as part of state to M' and having leader(M') inform them
3. Its probably good to include the version of the current config in all client operation requests. This way if the server knows about a later config (which is active) it can let the client know.

For servers in M', should the server periodically broadcast the new configuration (or its id), to try to makes sure that all servers in M' know about M' ?

4.6. Bibliography

Surveys:
1. Ken Birman, Dahlia Malkhi, and Robbert Van Renesse, Virtually Synchronous Methodology for Dynamic Service Replication, no. MSR-TR-2010-151, November 2010 paper
2. Leslie Lamport, Dahlia Malkhi and Lidong Zhou, Reconfiguring a State Machine. In SIGACT News 41(1), SIGACT News 41(1): 63-73 (2010) paper
3. Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, Jean-Philippe Martin, Alexander Shraer:
Reconfiguring Replicated Atomic Storage: A Tutorial. In the Bulletin of the European Association for Theoretical Computer Science 102, pages 84-108, Distributed Computing Column, October 2010. paper

Rambo (R/W objects, fully specifies configurations, online, uses external consensus on configurations order):
4. Nancy Lynch and Alex Shvartsman. RAMBO: A recongurable atomic memory service for dynamic networks. In 5th International Symposium on Distributed Computing (DISC), 2002. paper

DynaStore (R/W objects, incremental reconfiguration, online, doesn’t require consensus):
5. Marcos Aguilera, Idit Keidar, Dahlia Malkhi, and Alex Shraer, Dynamic Atomic Storage Without Consensus, in Journal of the ACM, ACM, 2011 paper