HBase Consensus

Mikhail Antonov [~mantonov]
April 17, 2014
See https://issues.apache.org/jira/browse/HBASE-10909 for reference and discussion.

Motivation & Overview

In https://issues.apache.org/jira/browse/HBASE-10296 the discussion has started about how to allow the replacement of ZooKeeper (ZK) by a general 'consensus library' that can handle HMaster failures by replicating the state contained in zookeeper. The state is kept consistent by coordinating any updates to that state.

Whatever algorithm / implementation may be chosen (let's assume we have a library implementing such an algorithm), some refactoring of the current HBase codebase is needed to integrate it. As of now low-level ZK-related APIs (ZkUtil, ZooKeeperWatcher classes for example) are used throughout the codebase. So one of the first practical steps seems to be to abstract out ZK API by identifying logical constructs implemented now via ZK and providing pluggable API for that.

It appears that master failover isn't the only place in HBase which would benefit from a "consensus" service (term "consensus" was a bit loosely defined in this context).

This document covers following topics:

- Transforming the current ZooKeeper implementation to a more general model of consensus-based coordination of global/distributed state:
    - Stage 1. Abstracting implementation behind interfaces. Code pointers.
    - Stage 2. Revising interfaces & modifying backing implementation towards pluggable coordination engine approach
- A possible approach to HBase global/distributed state coordination (a thinking of what the 'ideal' design might look like), with high-level design points
    - A distributed coordination engine for HBase
    - The inherent benefits thereof and add-on improvements

HBase state coordination

Let's consider all distributed state in HBase cluster requiring coordination.
https://blog.cloudera.com/blog/2013/10/what-are-hbase-znodes/ - I will use this as the basic overview. Simply put, it's divided into:
- operational state (active/backup HMs, registered/draining RSs, unassigned regions, table locks etc)
- ACLs
- Replication- and snapshot-related state.

Before describing a "fresh" design for HBase distributed state management, we define a few terms:

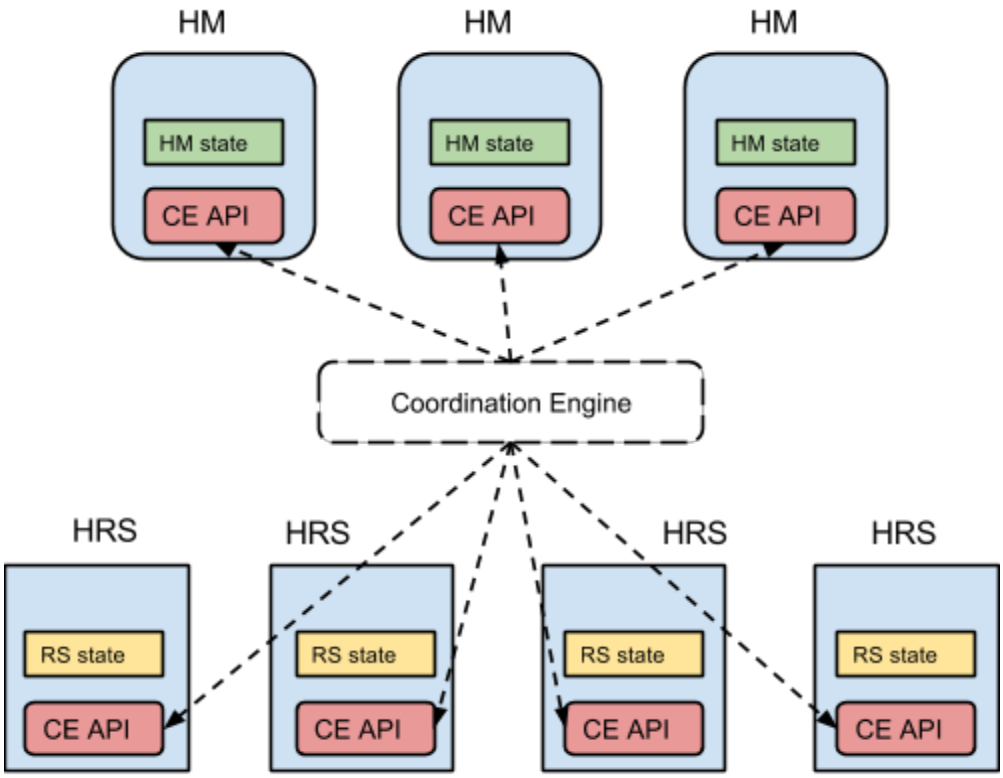
- A ***Coordination Engine (CE)*** is defined as a distributed service, which allows to agree on the order of events submitted to the engine by multiple proposers. Coordination Engine allows a distributed application composed of multiple nodes to executes the same commands (triggered by the observed events) in the same order, producing identical updates to the states of the nodes. A Coordination Engine essentially sequences events into a global sequence of events and let's the application nodes learn them in that order.
- ***Consistency Model***. Processing of the commands may take place at different rates on different nodes. Thus, at any given moment of "clock" time two nodes N1 and N2 may have different states, but all replicas which are in the same position in the global sequence have the same state. This is known in databases as *one-copy-equivalence.*
- The ***application state*** is a set of in-memory and persistent data structures on each node of the cluster. The structures are kept in-sync across the nodes using coordination engine.
- The ***role*** is the natural role of node in the cluster (Master or RegionServer). The set of commands the node may issue and the way it manages its application state is determined by the role of the node.
- A coordination engine allows HBase to update its state via a generic ***Coordination engine API***, which allows to:
    - submit a command as a ***proposal*** (optionally waiting for the agreement to be reached upon)
    - get notified when the ***agreement*** is reached

The agreement carries the information required to perform the actual command. The agreement is essentially a callback from the coordination engine to HBase, which updates the respective node's state.
Implementation of the coordination engine is intended to be pluggable. Depending on the application requirements it can implement different types of consensus algorithms, such as two phase commit, Paxos, ZAB. It can rely on a Zookeeper client to access the underlying ZK ensemble, or one of a Raft libraries, or such.

The whole distributed state is small enough to be comfortably kept as in-memory data structure on each node in the cluster (both masters and region servers).

Sketch of a replicated HBase Architecture

The diagram below depicts at high-level this schema applied to an HBase cluster. The state of a node depends on the node's role. HBase masters are fully replicated, that is their states are identical. Region Servers contain replicated regions, that is multiple replicas of a region can be maintained by multiple Region Servers. Although the sets of regions maintained by different RSs are different.



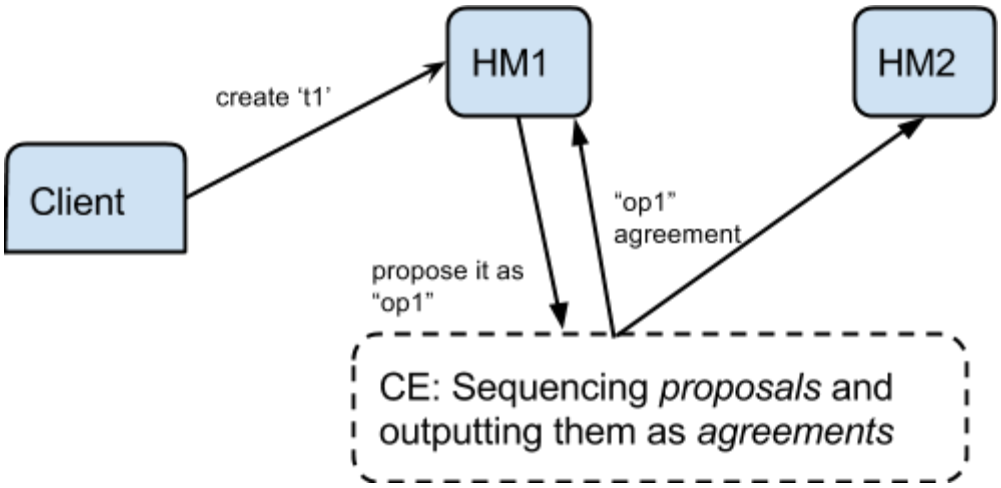
Let's take a closer look.

Each replica of HBase master or a reqion executes same sequence of commands which may look like this (the natural "place to hook in" appears to be the RPC calls; client -> HM, HM -> RS, client -> RS):

- ….

- region r1 gets assigned to HRS1
- region r2 gets assigned to HRS4
- table t1 created
- table t2 has 1 more column family
- …

Each node has set of registered handlers depending on it's role (HM or HRS). If node has handler registered for specific command type, it executes it, modifying the application state. Let's now look at how the particular HBase API call gets transformed to a command and executed.



Create table request from client gets intercepted on the node before changing the application state and runs through the coordination engine. The interception can be potentially done at RPC level or implemented with a use of co-processors. The request then is submitted to CE as a proposal for this operation. CE accepts the proposal, assigns it a global sequence number (GSN). It now becomes the agreement, which is delivered to all nodes affiliated with that type of operation. As the result of processing the agreement the application changes its state.

This approach can be used for all distributed state management commands. Let's look at opening a region as another example.

- "open region R1 at RS2" proposal is issued
- CE reaches consensus on the GSN for the operation and produces the agreement
- The agreement is delivered to the Region Servers
- For all nodes except RS2 the command to be executed will be no-op. RS2 will start opening R1 without worrying about possibility for it to be hijacked in the middle of operation by another region server.

Important note: The total number of nodes on the cluster is not directly related to the number of agents composing a quorum of the coordination engine. The quorum is determined by the implementation of the coordination engine and may be a set of several ZooKeeper nodes or set of Paxos acceptors, providing consensus for a larger number of application nodes.

Benefits (inherent and possible add-ons)

- **Improvement in current distributed state orchestration**
  Since all requests to update the state are coordinated, current code flow like opening region would not have to deal with znodes juggling and use "propose action, get consensus on that, once it's reached, execute" schema, rather than "set znode, start action, check znode, do some work, check znode again, finish the work and update znode checking version, aborting server if versions mismatch". That would also help to reduce number of points where we call server.abort() now if distributed state appears corrupted.

- **Multiple active masters**
  The client who needs to access the cluster retrieves the list of existing masters using some type of discovery mechanism (client-side config, DNS, etc.). The list of active masters may contain additional metadata for load-balancing purposes (e.g., masters are given weights based on the "proximity" to the client).
  Requests from all clients connected to any of the masters are agreed upon and sequenced before they are executed. The agreement and sequencing is provided by a coordination engine and the sequencing ensures the state of each master is consistent. When any of the masters fails, no cluster-wide interruption of service occurs as new clients can simply switch (using round-robin or randomly) to another active master. Clients connected to failed master can switch to any of existing masters using client connection timeout (the same mechanism may be used to switch away from a live, but slow or overloaded masters, or ones experiencing GC).

- **Multiple consistent read-write replicas of the same region may be hosted by different region servers, providing zero MTTR (for both read and write operations) in case of RS crash**
  Coordination Engine can be also used to allow hosting of multiple active replicas (read-write) of the same region on different RSs. This may be particularly useful for META table region(s), but is equally valuable for all tables, especially those experience high volume load. Clients requesting the location of any replicated region R1 receive list of replicas R1-r1, R1-r2, R1-r3, hosted by RS1, RS2, RS3 (may be with additional metadata for load balancing). If RS1 crashes, clients connected to it can switch to another RS hosting a replica of the region. The WAL splitting may actually not be required

at all, as the additional replicas of the regions hosted by failed RS may be spawned, or the WAL splitting may run in the background without regions downtime.

- **Avoiding responsiveness issues related to GC and major compactions on RS**
  If a region has many replicas, the clients can switch away from a RS performing a major JVM garbage collections or major compactions to another RS.

Abstracting ZooKeeper

Conceptually

ZK is used in HBase for:

- ***Access to transient shared state***
  Information is kept in ZK and all nodes read / write to it when they need to. For example, this information may contain a list of region servers currently registered with master or the IP of the current active master.
- ***Sending notifications to other nodes***[1]
  When znodes are created or modified the attached ZK watchers are notified. However, sometimes these notifications are missed by the watcher due to race conditions.
- ***Receiving notifications generated by other nodes***
  The ZooKeeperWatcher is used as a broker which receives watch-events from the ZK ensemble and pushes them to all registered ZooKeeperListeners.

These use cases could be abstracted from ZK as follows. In the following, we give a generalized model and how it can be implemented by ZK and Paxos, as possible examples of a consensus provider.

Shared state
Whenever we read / write shared state, we make a call to an abstract *Consensus* class. Individual Roles in HBase may extend this interface, e.g., the region server may use a *RegionServerConsensus* interface:

- The ZK-based implementation would call `ZkUtil.listChildrenNoWatch()`, or `ZkUtil.getDataNoWatch()`, which is what it does now.
- Paxos-based implementation would access locally-kept data (part of the application state, as defined by replicated state machine).

[1] 'Other nodes' may mean different part of the cluster which may be running on the same host or within the same JVM.

Sending notifications
Whenever we send notification we make a call to appropriate *Consensus* class, e.g., *SplitLogWorkerConsensus*:

- The ZK-based implementation would modify certain znodes in ZooKeeper namespace to trigger subscribed watchers.
- A Paxos-based implementation would generate a proposal to be agreed upon by a quorum of acceptors[2].

Receiving notifications
Whenever certain logic is to be executed in response to certain event, we let it to be called from the Consensus class.

- ZK-based implementation would make Consensus class a subclass of ZooKeeperListener, registered with ZooKeeperWatcher, with appropriate logic for events like nodeCreated(), nodeDeleted(), nodeDataChanded() etc.
- Paxos-based impl may be a learner, invoking appropriate handlers when the certain proposal gets agreed upon by acceptors and learned by the learner.

[2] http://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf

Abstracting ZK-level API

In the first instance we will follow the current ZK-based approach t®†foro distributed state orchestration closely, this way we can use existing tests and ensure nothing is broken when abstracting the ZK APIs into the interfaces described above. After this is done, the reconciliation of interfaces will replace the multiple checks of the global state with more “atomic” coordinated operations, as described above in “Benefits” section.

The list below describes the currently identified steps required to perform this abstraction. These are not prioritized and there may be unidentified work:

1. **Configuration:** We need to introduce the basis for consensus - configuration properties for consensus provider in `hbase-site.xml`, with a corresponding factory and classes which startup consensus provider as part of `HRegionServer` and `HMaster` startup steps.

2. Abstract **Region Assignment** from ZK:
    a. RS handlers.
        i. OpenRegionHandler
        ii. CloseRegionHandler
        iii. accompanying classes for Meta* handling
    b. HM handlers.
        i. OpenedRegionHandler
        ii. ClosedRegionHandler
    c. Region Transactions.
        i. SplitTransaction
        ii. RegionMergeTransaction
    d. Region Assignment.
        i. AssignmentManager

3. Abstract **HM admin operations**:
    a. Table operations:
        i. CreateTableHandler,
        ii. DeleteTableHandler
        iii. EnableTableHandler
        iv. DisableTableHandler

4. Abstract **WAL splitting**
    a. RS
        i. SplitLogWorker
        ii. HLogSplitter
    b. HM
        i. SplitLogManager

5. Abstract **distributed table locks**
    a. TableLockManager and related handlers (TODO: do we need a separate Jira ticket for that?)

6. Redesign ZooKeeperListeners to abstract them from ZooKeeper API.

Current design:

- The *ZooKeeperListener* interface is implemented by a number of classes and has 4 low-level API methods
    - *nodeCreated()*, *nodeDeleted(), nodeDataChanged()*, *nodeChildrenChanged()*
- *ZooKeeperWatcher* maintains list of listeners and upon receiving ZK event calls all registered listeners and passes them affected znode's path.

Proposed design:

- Create new interface *ConsensusHandler* that contains a single method, *handle(Agreement agreement).*
- Modify the classes currently implementing the *ZooKeeperListener* interface so they implement the new *ConsensusHandler* interface. In their *handle(...)* method they will typecast the *agreement* instance to specific type (like *AssignmentAgreement*), extract the information encoded in it, and do their work
- Create an interface called *AgreementLearner* which will maintain a list of all *ConsensusHandlers* as a map of znode prefixes to handler instance. The ZK implementation, *ZkAgreementLearner,* will be used to dispatch events to the appropriate namespaces in the *ZooKeeperWatcher* by parsing the znodes-level event, constructing Agreement objects of proper type and dispatching them to appropriate *ConsensusHandlers.*
- Modify ZooKeeperWatcher:
    - remove the internal collection of *ZooKeeperListeners*
    - move znode prefixes to *ZkAgreementLearner* class
    - in the handle(...) method pass the event to ZkAgreementLearner for further dispatching and execution.

  The following is a list of listeners to take care of (some are described in the items above already):

    - ZKLeaderManager
    - ZooKeeperNodeTracker
    - TableHFileArchiveTracker
    - ActiveMasterManager
    - AssignmentManager

    - SplitLogManager
    - ZKProcedureUtil
    - SplitLogWorker
    - ZKPermissionWatcher
    - ZKSecretWatcher
    - ZKNamespaceManager
    - DeletionListener
    - DrainingServerTracker
    - RecoveringRegionWatcher
    - RegionServerTracker
    - ReplicationPeer
    - MasterAddressTracker
    - MetaRegionTracker
    - ClusterStatusTracker
    - LoadBalancerTracker

7. Some other areas require additional design, including replication, visibility labels and distributed barriered procedures. The details of what work is required in this area will be updated in later revisions of this document.

Transforming APIs towards the coordinated model

- Once the abstraction of ZooKeeper is complete, the consensus API can be reconciled, and may start before the abstraction is completed.
- The path to reconciliation may be to take the state kept in ZooKeeper (like list of RS alive, list of unassigned regions, list of files to split) and transform it to:
    - State kept in ZK, which would be commands to be coordinated
    - In-memory state kept on each node

  This is where additional consensus libraries (Raft etc) integration will be very timely.

References

Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial.

Butler W. Lampson. How to Build a Highly Available System Using Consensus.