Procedure v2/Notification-Bus

Problem

1. Master operations (e.g. create/delete table, assignment, …) do not handle well failures. Support relies on hbck and manual cleanup of zookeeper nodes to help out customers get out from a situation where the cluster is locked out (not able to create/delete table or assign regions).
2. Coordinated operations like Snapshots or Cache Updates (e.g. ACLs, Visibility Tags) are implemented as adhoc code and are not fault tolerant leaving in an unknown state the operation. e.g. was the revoke permission applied on every machine?
3. The Assignment Manager it is kinda like a state machine, but it is not fault-tolerant unless META (which contains the state) is colocated with the Master, which among other reasons (compaction algo, java mem usage, …) limit the scalability of the number of regions. (see scaling up to 1M region jira)
4. HBaseAdmin is considered synchronous, but it is not. Half of the operation are asynchronous and the others seems to be synchronous but requires the code on the server side to remain unchanged. Also in case of failure (e.g. master failover/restart of bad network) the result of the operation is unknown. e.g. was the table deleted/created? was the split performed?

Solution

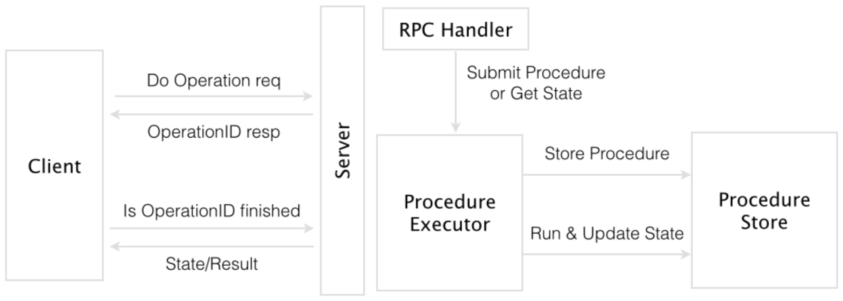
The Procedure v2/Notification-Bus aims to provide a unified way to build:

- multi-steps procedure with a rollback/rollforward ability in case of failure (e.g. create/delete table)
- notifications across multiple machines (e.g. ACLs/Labels/Quota cache updates)
- coordination of long-running/heavy procedures (e.g. compactions, splits, ...)
- procedures across multiple machines (e.g. Snapshots, Assignment)
- Synchronous calls, with the ability to see the state/result in case of failure.

Technical Approach

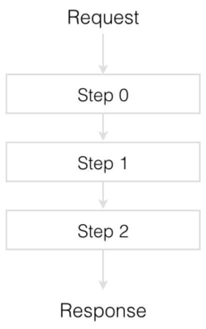
Core State-Machine (Procedure Engine)

The Procedure v2 has two main components, a “Procedure Executor” and a “Procedure Store”. The executor is responsible to run the procedures while the store keeps track of the state of each procedure. In case machine failure, the Procedure Executor is able to rerun the procedures that were not completed.



From the client point of view, a request will be sent to the server and an ID will be returned. The ID will be useful to poll the server until the request is completed. In case of failure, the client can log the operation ID and check it on restart or it can just list the operation in progress and grab the IDs on the one it is interested to wait on.

Procedures



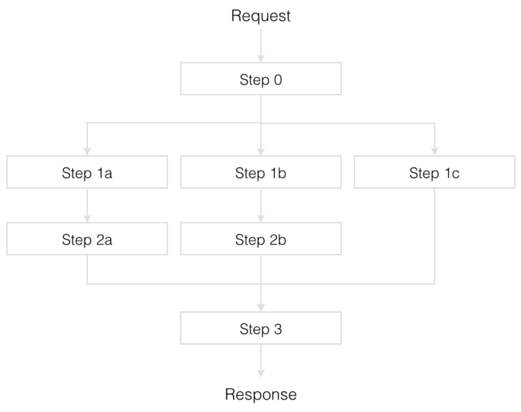
A procedure is an operation or a sequence of operations. A procedure changes the state of the system from State-A to State-B (e.g. adding/removing a table, taking a snapshot, …)
In case of failure at mid operation we should be able to rollback or rollforward, to keep the system in a consistent state.

Example: Create Table
from a code point of view the Create Table can be implemented as a sequence of 3 Procedures.

1. *Create FS Layout (Region Dirs, Region Infos)*
2. *Add Table Regions to META*
3. *Update Table Descriptor Cache*

Each Procedure will have an execute() method and a rollback() counterpart which will execute the operations to undo what done by the execute() method.

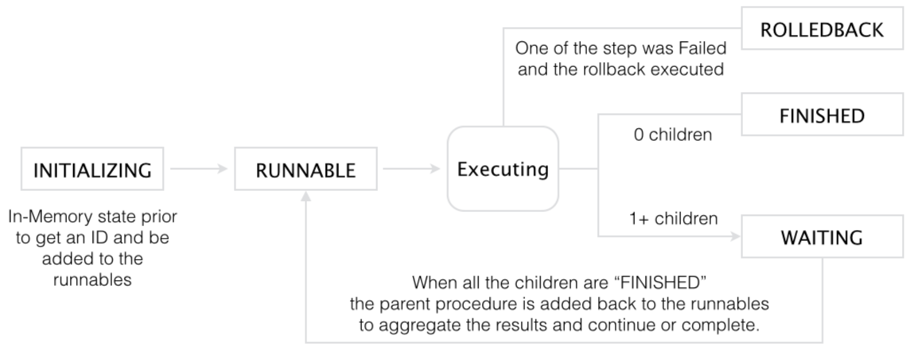
Implementation Details



Procedures can be simple as a linear sequence of operations but they can also be more complex.
Each procedure can return a set of subprocedure that may be executed in parallel. Once the subprocedures will are completed the parent is notified and it can go on by creating another set of subprocedures or declare the procedure as completed.

Each step of the procedure is pushed to the store, to allow to complete the unfinished procedures after a server failure.

Each Procedure must be written in a way that each step must be able to be executed multiple times (generating the same result) and must have a rollback step to undo what was done during the operation.



ROLLEDBACK
One of the step was Failed and the rollback executed
FINISHED
0 children
INITIALIZING
RUNNABLE
Executing
In-Memory state prior to get an ID and be added to the runnables
1+ children
WAITING
When all the children are "FINISHED" the parent procedure is added back to the runnables to aggregate the results and continue or complete.

State Store

The store is pluggable, and the interface is a simple load/insert/update/delete.
The separation of insert/update is useful for two reason. The first one being that insert should to be a transaction that includes the updated parent procedure state and the new subprocedures (it can be implemented as a sequence of updated, but the load will be a bit more complex in case of failure during the writes). The second advantage is that if the store is implemented as a log once you find an Insert you know that in the old logs that procedure is not present, so you can start it early if you want this kind of optimization.

Multi-Machine Procedures and Timeouts

Operations like Snapshots or ACLs cache updates requires a bit of coordination across multiple machine. To do that the procedure will send a message (may be done as poll via heartbeat) to each machine required by the procedure and will wait until each one respond. The procedure can have a timeout that will trigger a failure of the procedure causing the rollback.

Result/State and Queries

In case of synchronous operations the result must be keep around until the client ask for it, once we receive a "get" of the result we can schedule the delete of the record. For some operations the result may be "unnecessary" especially in case of failure (e.g. if the create table fail, we can query the operation result or we can just do a list table to see if it was created) so in some cases we can schedule the delete after a timeout.

On the client side the operation will return a "Procedure ID", this ID can be used to wait until the procedure is completed and get the result/exception.

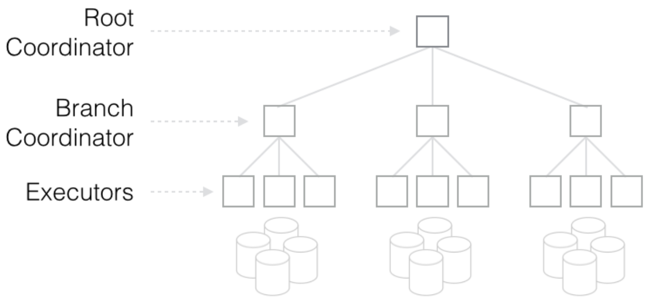
```
Admin.doOperation() {
   long procId = master.doOperation();
   master.waitCompletion(procId);
}
```

If the master goes down while performing the operation the backup master will pickup the half in-progress operation and complete it. The client will not notice the failure.

Notification-Bus

The notification-bus allows to send notifications/procedures to the Region Servers.

Assuming no other coordinator between the Master and the Region Server, the operation is sent down to the executor (Region Server) and the Master will be responsible to retry/resend the operation in case no response will be received in the expected period of time. If the Master goes down during the operation, on restart it will recover the previous state, and wait for responses or it will retry/resend the operation in case no response will be received in the expected time period. *(The Region Server will not persist any state, retry or re-execute any previously pending operation on restart, everything is coordinated by the master)*



OnePhaseProcedure

The OnePhaseProcedure describes the simplest request/response operation.
From the master point of view the operations are:

- Select a set of Machines to set the “procedure” to (e.g. all machines hosting the regions of the specified table)
- Send the procedure to the Region Server
- Wait a response from all the components involved.
    - If the master does not receve a response within a timeout, or the region was reassigned, it will resend the execution request.
    - If any of the procedure will fail, the master will send an abort to machines involved in the operations.

TwoPhaseProcedure

The TwoPhaseProcedure describes an execution with a “staging” state. Similar to the OnePhaseProcedure, the procedure is sent to the set of machines involved in the operation, then each machine will perform the operation in a “staging” area, and the operation will be committed only if every component involved in the operation was able to perform the “staging” operation, otherwise a rollback is sent to every machine.

Milestones 1 - State Machine

1. Implement the core state-machine (matteo)
2. Replace the create/delete table handlers with a procedure implementation
3. Testing

Create and Delete tables are the most exercised function from the test suite. so, once we have those up and running for a while we can say that the core state-machine is quite stable. We don’t have any test covering the failure conditions (the ones that at the moment requires manual intervention). We should spend time on covering those cases.

After that anyone can rewrite the other handlers (e.g. enable/disable table) as procedure. (The sync-client implementation can be done for the 2.0 branch, but we can’t backport that to keep the compatibility. New client methods can be added using the procedure)

Roliing upgrades are not a problem since at this level, code is on the master side only. The client side remains unchanged (due to compatibility reason, as explained above). On restart of the new master the state machine will not find any log to replay, so everything starts up as no operation where executed. Half states from a previous run (e.g. interrupted create/delete table) are not handled by the old master, and the only workaround is runing hbck or fixing it by hand, so nothing is changed.

Milestones 2 - Notification Bus

The core state-machine will provide the foundation to build the notification bus, which is just an exchange of messages between the multiple machines (e.g. master and region-servers).

1. Implement the core of notification bus (Matteo)
    a. Implement OnePhaseProcedure (single message-response between a coordinator and the executors)
2. Port “Online” Snapshots to Procedure v2/Notification-bus
3. Testing

Since snapshots are not a core part of the system and it is ok to have them fail and retry on-failure, snapshots are the best feature to stabilize the notification-bus.

Once we stabilized snapshots, anyone can implement the TwoPhaseProcedure for ACLs/Visibility-Labels updates (not too much different from OnePhaseProcedure).

Milestones 3 - Assignment Manager

By on the Notification-Bus we can fix (speedup and cleanup) the Assignment Manager. Most of the code will go away, since the core-state machine and the notification-bus will do most of the job. The Assignment Manager will just select the machine and send out the assign/unassign request.

At this point we can improve the Notification-Bus and the Procedure Executor Queues to let them know about the state of the regions. e.g. delay balancing/move during multi-machine procedures (e.g. snapshot, cache updates, ...)

Milestones 4 - Distributed Log Replay

Since the Notification-Bus is generic enough, we can replace the adhoc distributed log replay with a OnePhaseProcedure based version.

Milestones 5+ - New Features (e.g. coordinated compactions)

Same as the Distributed Log Replay, all the new features that requires a bit of coordination between multiple machines can be implemented in a fault tolerant way by using the notification bus.

By Release

- 1.1.x Fault tolerant Master Operations and Snapshots
- 1.2.x Cache Updates, WebUI
- 1.3.x Assignment Manager, Distributed Log Replay
- 2.x Sync-Client, coordinated compactions, ...