Apache HBase Master Redesign Proposal. The hbck-master DRAFT (HBASE-5487) (Long Version)

Jonathan Hsieh (jmhsieh@apache.org)
10/19/13 v2b, typo fixes and elaboration on core design
10/17/13 v2, public distribution

Abstract

This is a design proposal for an “hbck all the time” master/assignment manager. It is a pessimistic design that is essentially always listening for events and state updates and then constantly tries to “recover” to a valid configuration.

- Clean slate design that eventually contains all current master functionality and can be cleanly extended for new functionality.
- It assumes an active master that coordinates all regions and has exclusive atomic write access to durable storage.
- Uses Finite State Machines (FSM) model for tracking progress, managing state. This allows us to create arbitrary states to test recovery, and which can be fuzzed or tested exhaustively. Error states are explicitly defined.
- Decouples the state hbase is currently in (current state), where it should be in (intended state), and the soft state that describes where hbase currently is (actual state). This could be thought of as hbck all the time -- it is always trying to “recover” by making actual state match intended state.

Table of contents.

- Abstract
- Motivation
- Core Design
  - FSMs
  - Region FSM
- Failure handling
  - Master Failures
  - Region server failures
    - Meta hang, no correctness impact:
    - Normal client read / write region assignment transition process safe (assume no hang situation):
    - Attempt 1: Hang on close (hazard):
    - Attempt 2: Hang on close (with lease):
- Concurrency / performance limits

Evolving new features.
Implementation Options
    Testing
Open questions, sketches for follow on work.
Appendix: Detailed State transitions.
Appendix: Graphviz code for state machines.

Motivation

The current master is a complicated beast responsible for coordinating actions and changes in regions, tables, and regionservers. It has ad-hoc communication patterns between and has shared state between the client, master, zk, meta and regionservers. For example, an assignment operation (0.92/0.94 era) (see http://people.apache.org/~jmhsieh/hbase/120905-hbase-assignment.pdf) has approximately 18 network communications which requires complicated recovery logic. (we test a lot of single failure situations but few where there are multiple.) Since this incarnation of the master took over in the 0.90 versions, it has become more robust at the cost of complexity. Debugging the root causes of inconsistencies that occur in real-world operation can only be done a fraction of the HBase committers. Most admins and users instead use the hbck tool for repairs inconsistent state.

A by-no-means comprehensive list of concerns in the current system include:

- "Slow" assignment means we cannot assign a large number of regions quickly has moved us towards larger regions (which leads us fundamentally to compaction pains and away from predictable low-latency usecases).
- Can't recover multistep operations such as merges, splits, schema changes if the masters fail and can often end up in bad states requiring hbck in these situations. In the case of multistep operations, in most cases we revert work and either lose the operation or do it over from the start, affecting MTTR.
- Custom code paths / frameworks for each operation with ad-hoc recovery mechanisms (distributed log splitting, recovery, snapshots, merge, split).
- Conflicting actions can be concurrently initiated by the master, regionserver, hbck, hbase shell and "expert admin" zk hacks.
    - Requires interprocess coordination aka interprocess locks (which require expiry and thus can increase recovery time)
    - This makes the system hard to test, and expensive to test exhaustively.

A new design will have the following properties:

- **Rigorous.** We must consider all classes of error conditions (including nacks/no response/hangs/juliet pauses), multiple failure scenarios in the design and not as second class afterthoughts. Ideally we are able to prove the design is correct and use it for all error condition recovery cases.
- **Performant.** The design should improve the performance of operations critical for

recovery operations such as assignment and log replay coordination.

- **Recoverable**. We need something that can recover from arbitrary failures and continue to make progress and guarantee acked admin requests are completed. Compound admin operations should be recoverable in the event of a master or region it server failure.
- **Scalable.** We should allow for scaling the master out via sharding or replication.
- **Consistent.** Currently we need hbck to fix inconsistencies in an operational hbase. Ideally we have a design that obviates the need for such a tool.
- **Unified**. Having multiple frameworks for different operations (distributed log splitting, snapshots, assignment manager, draining servers, splitting) is cumbersome and in general having fewer of them means will mean fewer edge cases. Ideally eventually ALL operations (RS, Region, table, etc) fit a single pattern. This simplification aids our ability to test and to safely extend it. It should be easy to update design-level docs and allow more folks to quickly understand.
- **Admin API compatible.** The hbase admin api should remain API compatible.
- **Migration from existing implementations.** Ideally we could upgrade from existing versions without downtime. Ideally we could rolling upgrade from this design to an extended version.

Core Design

In this HBase design, the **master** manages **meta state** which contains the **current state** (or last valid state) and the **intended state** (transition target state). Meta state is a set of durable records specified by the user or by system initiated changes. Region servers are now dumb and only do what they are told by the master and report their **actual state** to the master. Admin clients issues commands which the master checks and conditionally update intended state or rejects. Clients do not directly modify current state. Data clients only read the current state to get redirected to RS's for data operations (get/put/scan).

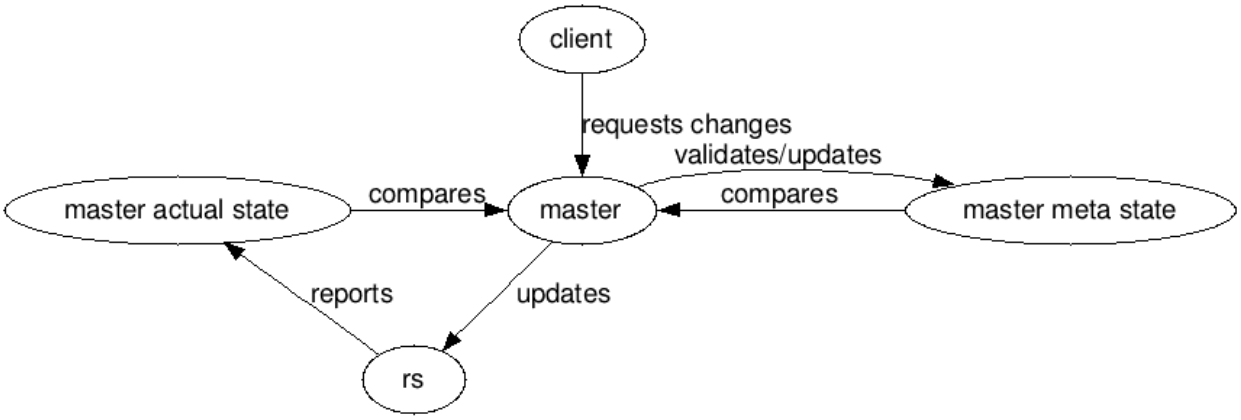
The brains of the master is reactive to changes in actual or meta state (whether user-initiated or system initiated). It decides on how to act based on the delta between intended state and actual state. This master is in effect always trying to recover, and designed such that the recovery execution paths are the same as the normal execution paths. When the master attempts an operation that returns with an ambiguous response (timeout) it transitions to a state that captures this ambiguity as something that it needs to recovered from.

Upon actual state updates, the master validates the meta's integrity (assignments on live region servers, no region holes or overlaps in tables, etc). If the meta state is invalid (hbase data restarted on different cluster), or actual state invalidates meta state (machine failure) the master updates meta's current state and initiates fixes by writing intended state. For example, RS A goes down and current state says region R was open on it. When the master detects A is down via a change in actual state, it invalidates R's current state (doesn't make sense for a

region to be open on down or nonexistent RS), and then determines and logs R's new intended state (assigned to another RS). This is similar to how hbck repairs holes or overlaps in regions -- but now we are taking RS's assignments into account when checking validity.

Updates to meta state can be triggered by user-initiated commands sent to the master or triggered by the master in reaction to actual state updates. These updates to meta state **must only** be done via the active master. RSs do not directly modify perform meta state changes.

Actual state is soft-state that can be wiped out without consequences and is not durably stored. The master's actual state can be updated by the return values (ack, nack, hang/timeout) from actual state changing operations (open on RS, write to HDFS) or the detection of external events (rs fail clean, rs fail dirty, rs hang).



Commands such region splits, region merges, and region moves cannot atomically modify multiple pieces of actual state to match intended state (we can not close on one rs and open on another atomically). These require **multi-step processes** that introduce the potential to fail between actual-state changing operations (e.g. region move requires: set intended state, region close, region open). To ensure these operations can be recovered upon master failure, the process's constituent operations must logged as progress is made and in some places are given explicit states. When a master fails, a new master will be elected (via ZK), restores the previous master's state simply by reading the meta state, and then continues.

FSMs

Each class of intended state entities (such as regions, tables, namespaces) are managed using a finite state machine. User operations simply durably update the intended state about the table/region/regionserver. The master compares the new intended and current state with the actual state uses the fsm to determine the set of actions required to reach the intended state.

Each state in the FSM has specific invariants and transitions require certain preconditions before

they can complete. When an entity is in a particular state, we can assume the invariants hold and operate. State transitions cannot be skipped -- they must be traversed to reach subsequent states. All states have an implicit transition to the unknown state which has must re-establish invariants to re-enter the state machine.

The state machines for tables and regions are actually tied together. Certain operations on a table (snapshot, restore, enable, disable) affect its constituent regions. A table being in a particular state essentially acts like a lock and may invalidate the state of regions. Similarly a region in a particular state will block certain table operations. Having this be a state means the locks state are preserved in the case of master failures. This obviates the need for the ZK based interprocess locks.

Region FSM

A region is the most granular abstraction that the master has to reliably manage state about. The region's state machine needs to cover all the states a region could be in from the active master's point of view. Region operations cause a FSM transition by logging an intendedState to meta. Operations include Create, Flush, Compact, Assign, Move, Snapshot region, restore snapshot region, Split, and Merge.

Region intended state is a map from regionname to region data:

Region Name / Row Key:

- tableName
- endKey/startKey (we can do HBASE-2600 while we do this change, or keep the endkey)
- regionTs (similar to current).

Region Data:

- **currentState** current FSM state in region state machine along with:
  - **server/servercode assignment** - RS identifier that region is assigned to.
  - **cfSchema** (column families and their schema, used to be in regioninfo)
  - **version/timestamp** (required to differentiate write/write hazards and to invalidate cached state on RSs in error handling cases).
  - Other region data like security settings, favored nodes etc would be stored here but are but are essentially treated like cfSchema.
- **intendedState** FSM state the region is in the process of transitioning to; not present if currentState == actualState. This includes a new version stamp and the delta with respect to the currentState's server/servercode assignment, cfSchema, etc.
- **operationLog** durable instructions to update while transitioning from sourceState to targetState. This is a stack of operations operated on while in this state (formally this makes this a push-down automata instead of an finite state machine)

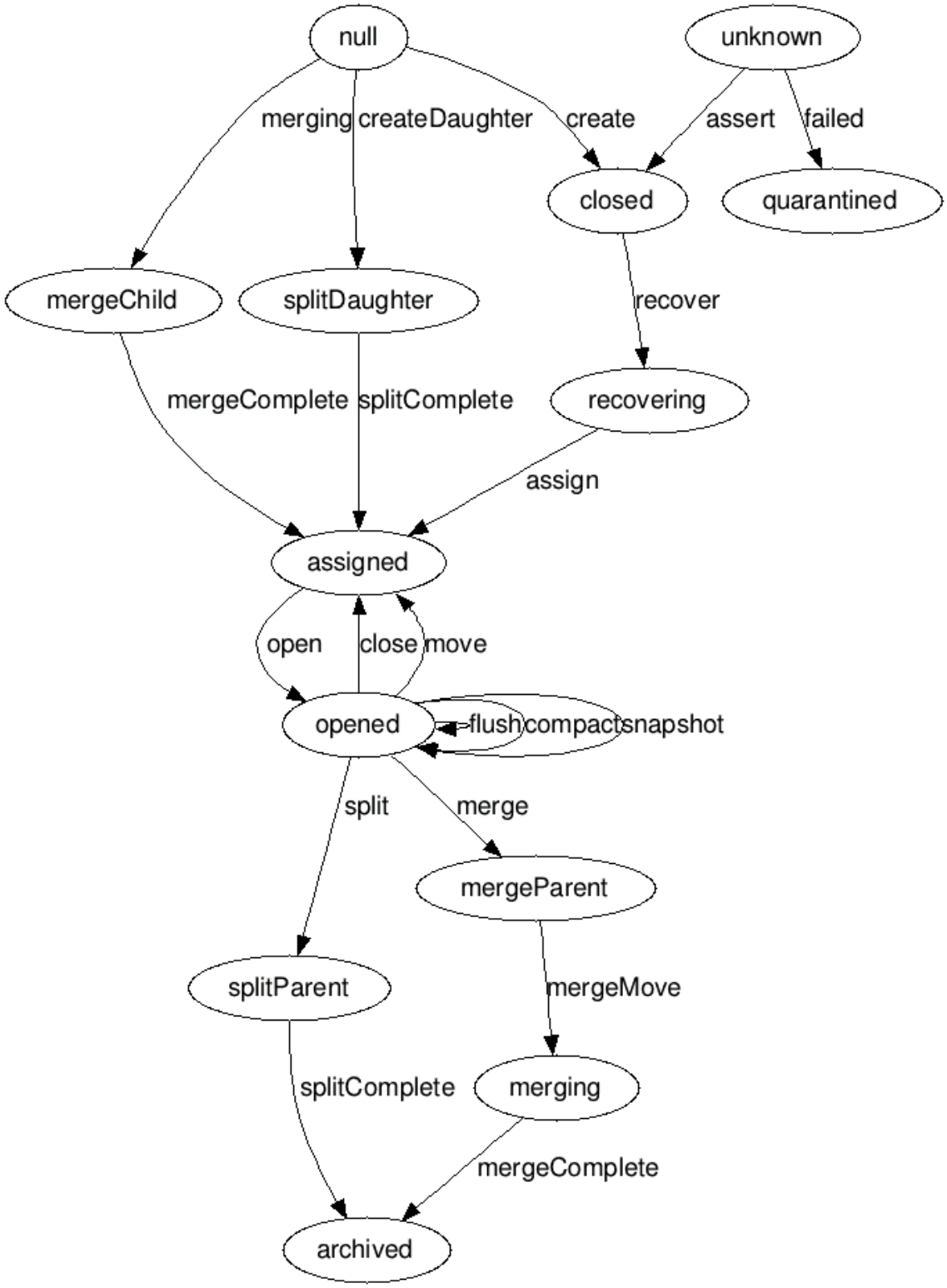
- **trace** for span/trace information to thread through, and for clients to get results in case of master failover between client op submission and op completion. (clients could submit to one master and get result from new).

A currentState-to-an-intendedState **transition** is in-progress when an intendedState is a present on the region row. A transition is complete when all of the atomic actual state changing operations are completed. After a completed transition, the intendedState is copied into the currentState and then the intendedState is removed. Now the master's planner may decide to trigger new transitions by deciding and writing a new intendedState.

Example actual state changing actions that require logging

- Master performs an atomic file system operation. This could include the master performs any # of file system writes to temp dir with single atomic dir rename. Ex: snapshot commit (atomic filesystem dir rename).
- Master performs multiple fs operations that would ideally be atomic to "live" hdfs dirs. Since not atomic, it must completely finish (or be completely reverted). Thus the intermediate operation progress of these processes must logged to the region's operationLog. Ex: snapshot restore - must be offline and should record progress of restore operation for new master to continue restore process.
- Master assigns an region to an RS.
- Master opens a region on its assigned RS.
- Master creates new regions (split, merge) or adds columns to region (alter)

View this table in the appendix for in depth state machine transition details, and the durable state invariants associated with the states. (it is not complete, some transitions descriptiong are missing)



Failure handling

Every network operation can essentially have 4 outcomes: **success ack**, **failure ack** (nack), **timeout with unknowingly that succeeds**, or **timeout with unknowingly failed**. For the model to be complete and rigorous we must account for all of these situations. In the past we've had problems where timeout falsely assumed failure -- see HBASE-4562 (splitting).

Master Failures

If a master fails an attempts to write meta state and receives a nack, the master should block until it succeeds. Since the master only reads meta when starting up (in memory state is coherent with durable meta), so receiving nacks on reads essentially force the master block as well. It could be doing log recovery for meta data and should wait to read cache metadata until after meta recovery is complete. If writes timeout the master should kill itself because it will not know the outcome of the meta write operation.

We face these situations:

- master fails clean (decommission) -- zk race to determine new active master. new master opens and reads intended state meta data tables into memory. This restores all of the FSM data from durable up to date meta state. Master gets up-to-date actual state from RS's, then continues as normal.
- master fails dirty -- previously active master znode expires. Other potential masters zk race to determine new active master. new master does log recovery of intended state meta data tables and reads them into memory. This restores all of the intended state with FSM data. Master's tracker gets up to date actual state from RS's, continue as normal.

Region server failures

A **region server down** is a normal FSM transition initiator - it is a change to actual state that merely causes the master to transition to a new intended state in meta.

These scenarios are roughly handled like this:

- rs fails clean -- RS notifies master of actual state with regions that will no longer hosted by RS. The master's controller receives the changes and realizes the intended state is invalid. Controller calculates a new valid intended state for regions. Controller logs operations to get back to expected state.
- rs fails dirty -- master sees timeout and updates actual state. same as above.

If a master is fails while attempting to change an RS's actual state (ex: opening/closing region on a RS) we still use the FSM to handle these case. For any nondeterministic cases (timeouts with unknown results), we can default to transitioning to the UNKNOWN state where our invariants are unknown. From the UNKNOWN state, the master must reassert all assumptions to re-enter the "normal" part of the state machine. For example the transition to UNKNOWN would force the current state to UNKNOWN, clean incomplete temp data in the filesystem, and then close

any RS's the actual state reports the region is open on. The master would then determine a new intended state for the region.

Hangs are situations where the master attempts operation, the operation completes (success or failure) but from the master POV the op timed out and the master has made another decision to rectify the situation. It should assume the operation had unknown state but we've commonly assumed failure which has lead in the past lead to double assignment, failed splits on recovery, and other region inconsistencies. In the current proposed solution, recovery time will suffer because we will depend upon a lease timeout to guarantee correctness on recovery. This situation is tricky so I'm going to go to a bit more formal notation to prove a solution. We present an example proof sketch for master failover, and then then the subtle case where closing times out during a region move operation. The latter has a failed proof sketch and then a successful proof sketch.

Definitions:
Let r_n be a region. Let s_m be region server. Let M_A be the active master and M be any arbitrary standby master.
Let $r_n \rightarrow S_m$ denote region r_n being assigned to server s_m.
Let $r_n \Rightarrow s_n$ denote region r_n being opened on server s_m. (though not necessarily assigned)

Correctness condition:
For all client read operations we only read rs with the latest current state version.
For all client write operations we only write rs with the latest current state version.
As long as we have a master and an rs and a sufficiently long period of non-failure time, we will make progress.

Assumptions.
$\forall r_n \rightarrow s_m$ there is a version number v_i and stored in M_A's current state
$\forall r_n \Rightarrow s_m$ there is a version number v_i stored in s_n's actual state
$\forall s_i$ reports $\forall r_n \Rightarrow s_i$ with v_x periodically to master's actual state.
Version number strictly increases when M_A's current state of r_n is updated.
A region is **cleanly open** iff $r_n \rightarrow s_m$ and $r_n \Rightarrow s_m$ and both are at v_i

Meta hang, no correctness impact:

M_A hangs and zk node expires. M_B wins zk race, updates zk, steps up, restores meta state and gathers actual state. Old clients cached that master is at M_A, new clients talk to new M_B.
Cases:

- Client catches that M_A zk node expiry event. It knows it must read new zk node for new master. Success.
- Client misses that M_A zk node expiry event. Client catches M_B new master event. It knows it must read new zk node for new master. Success.

- Client misses that M_A zk node expiry event. Client misses M_B new master event. Client thinks M_A is the master needs to read new current state for region location from meta. M_A recovers from hang. M_A **must know that it had hung.** M_A **rejects client's requests (causing region to reread zk).** M_A **must proactively recheck zk to see if it is still the master. (need to verify zk does this)**
    - **If it still is the master, future requests from client can succeed.**
    - **If it is not, the old master reject clients and abdicates.**

Since meta state is durable and does not change during failover, we can just continue with normal RS failure analysis.

Normal client read / write region assignment transition process safe (assume no hang situation):

A client will first contact M_A to find out $r_n \rightarrow s_m$ and v_i. It will cache this information. Client it assumes $r_n \Rightarrow s_m$ so it goes to s_m to do data operations. Cases:

- $r_n \Rightarrow s_m$ and versions match. All is well, read and write ops are safe.
- $r_n \Rightarrow s_m$ but the rs has newer version number. Client is out of date and must revisit M_A to get a new $r_n \rightarrow s_x$.
- $r_n \Rightarrow s_m$ but the rs has older version number. Client cannot read from region because it is out of date and must wait until RS's catch up.
- $r_n \not\Rightarrow s_m$. Start over by invalidating cached assignment and retry read M_A. cases:
    - we are early and $r_n \Rightarrow s_m$ will be true. Subsequent read of M_A will provide same result and client will eventually succeed. If still not retry.
    - r_n will not be open on s_m. Subsequent read from if M_A will provide $r_n \rightarrow s_{m+1}$ (a different assignment). We start over.

Attempt 1: Hang on close (hazard):

Master has $r_n \Rightarrow s_m$ with v0 and it was cleanly opened.
Master updates intended meta to reflect $r_n \rightarrow s_{m+1}$ with v1 in an attempt to go to ASSIGNED. It gets a timeout on close. Master no longer knows if $r_n \Rightarrow s_m$ with v0 so transitions to UNKNOWN state. Cases:

- Master eventually gets to ASSIGNED state with $r_n \rightarrow s_{m+1}$ with v1, and then successfully open such that $r_n \Rightarrow s_{m+1}$ with v1. In this case, the close on $r_n \Rightarrow s_m$ with v0 was **actually succeed**. Cases:
    - Close succeeds before region is OPENED with new assignment. Clients that cached the old assignment would eventually fall into the normal region assignment transition process.
    - Close succeeds after a pause but after new assignment $r_n \Rightarrow s_{m+1}$ with v1 and is in OPENED state. Problem: There is a period where new client Clients would read/write from $r_n \Rightarrow s_{m+1}$ with v1 while old clients would continue to read/write

$r_n \Rightarrow s_m$ with v0 on. **We have a small window of double assignment and failure of transactional isolation.**

- Master eventually gets to ASSIGNED state with $r_n \rightarrow s_{m+1}$ with v1, and then successfully OPENED such that $r_n \Rightarrow s_{m+1}$ with v1. In this case, the close on $r_n \Rightarrow s_m$ with v0 was **actually failed**. It remains open with the old assignments and clients that cached it continue to use it. Cases:
  - s_m reports actual state with including $r_n \Rightarrow s_m$ with v0. Master sees this and notices that it conflicts with intended meta state. Subsequent close attempt succeeds. Client seems new version for region, invalidates cache and falls into normal region assignment transition process.
  - s_m fails to reports actual state with including $r_n \Rightarrow s_m$ with v0. Master never sees the conflict. Old clients continue to talk to old rs assignment while new clients talk to newer rs assignment. **We have an indefinite window of double assignment and failure of transactional isolation.**

Attempt 2: Hang on close (with lease):

Master has $r_n \Rightarrow s_m$ with v0 and it was cleanly opened.
Master updates intended meta to reflect $r_n \rightarrow s_{m+1}$ with v1 in an attempt to go to ASSIGNED. It gets a timeout on close. Master no longer knows if $r_n \Rightarrow s_m$ with v0 so transitions to UNKNOWN state and waits for the lease period expire before allowing transitions away from UNKNOWN to complete. Cases:

- Master eventually gets to ASSIGNED state with $r_n \rightarrow s_{m+1}$ with v1, and then successfully open such that $r_n \Rightarrow s_{m+1}$ with v1. In this case, the close on $r_n \Rightarrow s_m$ with v0 was **actually succeed**. Cases:
  - Close succeeds before region is OPENED with new assignment. Clients that cached the old assignment would eventually fall into the normal region assignment transition process.
  - Close succeeds after a pause but after new assignment $r_n \Rightarrow s_{m+1}$ with v1 and is in OPENED state. Since we know assignment $r_n \Rightarrow s_{m+1}$ with v1 went through UNKNOWN state's wait condition, we know the lease period must has already elapsed. Thus r_n must update its lease and validate $r_n \Rightarrow s_m$ with v0 before serving the client. When it checks with M_a, it will see the r_n as at v1 in meta current state. r_n fails the subsequent client requests that assume $r_n \Rightarrow s_m$ with v0 and notifies client to invalidate its cached $r_n \Rightarrow s_m$ with v0 information.
- Master eventually gets to ASSIGNED state with $r_n \rightarrow s_{m+1}$ with v1, and then successfully OPENED such that $r_n \Rightarrow s_{m+1}$ with v1. In this case, the close on $r_n \Rightarrow s_m$ with v0 was **actually failed**. ~~It remains open with the old assignments and clients that cached it continue to use it.~~ Since we know assignment $r_n \Rightarrow s_{m+1}$ with v1 went through UNKNOWN state's wait condition, we know the lease period must has already elapsed. Thus r_n must update its lease and validate $r_n \Rightarrow s_m$ with v0 before serving the client.

When it checks with M_a, it will see the r_n as at v1 in meta current state. r_n fails the subsequent client requests that assume $r_n \Rightarrow s_m$ with v0 and notifies client to invalidate its cached $r_n \Rightarrow s_m$ with v0 information.

Finally there are cases we've used where the integrity of meta's tables are bad. (overlapping regions, region holes). We'll call these problems in meta state **validity problems.** Like hbck, meta must be valid before any other operations can occur. When we have invalid meta state, the master must analyzes the meta state and make decisions to repair make it valid again. Here we may change meta state to fabricate new regions to fill holes or merge regions to reconcile overlaps or quarantine regions to later bulk load them in. Any regions involved with the validity problems will be set to an INVALID current state during repairs and will likely need to observe the lease timeout before repairing the situation. This condition could happen with a "perfectly" timed failure if an operation was not operation but transitioned as if it was (doing multiple single row atomic updates but not actually being atomic).

Concurrency / performance limits

- for many operations like assignment and opening, each region is independent and can be parallelized and batched to region servers.
- some operations like splits, merges require coordination but can happen in parallel as long as the involved regions do not conflict. A first cut would require the table be in a particular state that blocks table operations like alter, merge, online snapshots.
- some operations (compact, flush) only work when a region is opened -- the FSM reflects this.
- some table operations should block region state transitions and vice versa. e.g. No disable while online snapshotting, no splits while repairing table region integrity.
- some admin operations do not affect the contents of meta state and thus can be managed by the RS. Ex: flush, compact.
- Performance bottleneck is the number of steps required to complete a process. Having fewer states and more types of transitions can improve performance.

Evolving new features.

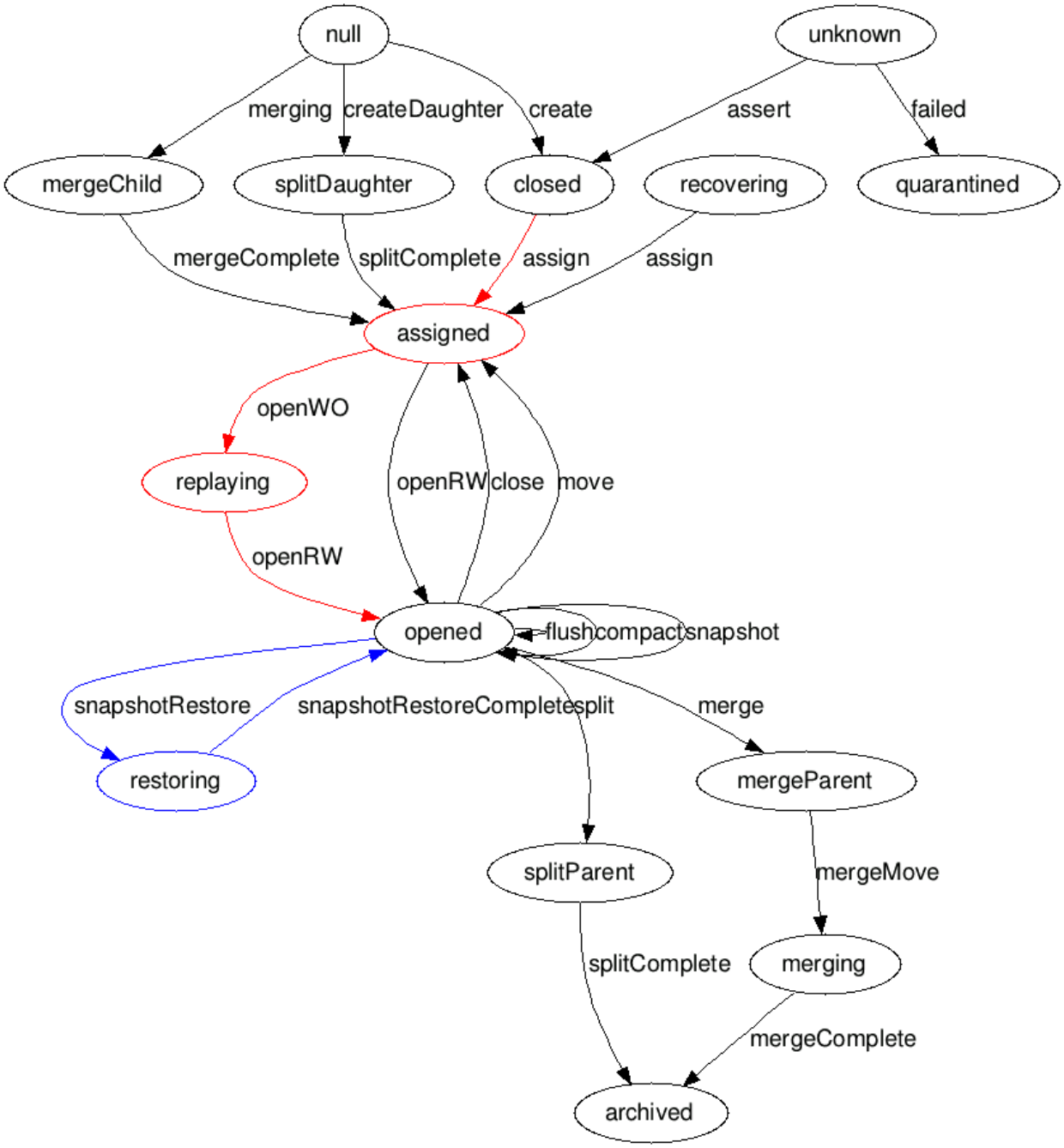
The finite state machine approach should be applied to any extensions or new functionality we add dealing with regions. This should be accomplished without having to be intimately familiar with the internals of other processes. As an example, we'll describe FSM modifications to handle fast write recovery process (ala HBASE-7006 Distributed Log Replay), online snapshot restore, FSM based read-only mode (could be also be done via schema), and a sketch for multiple read replicas.

Strawman: fast write recovery aka Distributed log replay. This feature opens a region write only to accept log replays and new fresh writes but blocks reads until replay has completed. We've updated the FSM with new red states ands and modified red transitions. We change the order from recover -> assign -> opened (RW) to assign -> replay (open write only) -> and then open (RW).

Strawman: Online restore. Today we must have a table disabled to do a snapshot restore operation; this would make it possible to do an online restore. The FSM is updated with new blue states and transitions. Currently we don't have online restore because it modifies multiple fs elements in place. The Table FSM would have one more arc from locked to locked, and would log a list region restore operations in the table's operationLog. While this was "locked" each involved region involved in would list hfiles/hfilelinks to create, remove, alter and populate.

Strawman: Multiple read replicas. Intended state would now have multiple assignments. In one possible design, we only write at one RS and but could read from other replicas. We would add new fields to current state (and clients would have to know how to use this). We'd now have a few new states for OPENED (one open for RW), and maybe SEMIOPEN states.

Strawman: FSM-based read-only mode. We'd add a new FSM state -- OPENRO, which would have the region open RO as an invariant. It would have all the same transitions leaving the state as the normal opened (RW) state.



Implementation Options

This doc purposely written describing a design without being specific to durable storage implementations.

Meta state could be stored in:

- durable ZK (accumulo does this I believe)
- **a set of HBase Table in a meta or system namespace (HBase discussions prefers**

here)

Actual state could be reported via:

- ZK
- **heartbeats from regions (offline conversations seem to prefered here).**

Where is meta and the master? Model doesn't have requirements but performance may.

- **colocated (HBase discussion prefers here, likely for first implementation since simplest).**
- separate entities.

Testing

Today it is difficult to claim the current state machine are correct because there accurate and succinct explanation of it. it is too complex and requires full mini-clusters to be testable.

We can first decouple testing the correctness of the state machines and its transitions from actual IO. We must however, modify all so that all transitions (IO operations) cover all 4 response cases (ack,nack, timeout fail, timeout succeed). If these four response cases are sufficient I claim we can prove the FSM design point to be correct.

Instead we 1) built the state machines in particular states and test them and 2) create a job that can essentially exhaustively test the state machine (todd did this with hdfs ha -- today's ha bugs are not in the core of the design) and 3) we run the blackbox system tests against it to verify.

Open questions, sketches for follow on work.

- tables and region state being tied. Elaborate on this and why.
- Why have the schema data in region in this design instead of only at the table level? Short answer: Because that is the way it is now. Longer answer: Could use table schema version number or consider other mechanisms.
- performance analysis: Moves/assigns were notoriously slow, is this any better?
    - would want to have meta co-located with the master so that it does not require network hops
- coprocessors and plugin extensions.
    - we could simply hook each of the individual state transitions.
- Regionservers ported over to FSM mechanisms. Would have states like starting, up down, restarting, draining, upgrading, quarantined, unknown.
- Namespaces ported over to FSM mechanisms. (not sure what for yet)
- Snapshots ported over to FSM mechanisms. (probably overkill)
- Scalability: multiple sharded masters/metas federated by namespace.
- Pluggable meta durable state interface to enable warm standby masters or eventually multiple active masters.

- Integration with Benoit's/Arista's fast server connection shoot down switch feature: just a command that issues an intended state transition.
- Use formal tools/rules engine to prove correctness and/or generated code.

Appendix: Detailed State transitions.

Start State	Transition	End State	End state invariants	Assigned	State on FS	State on RS	Transition Actions
Full split before available recovery (0.94 recovery)							
		null	entry in region intended state	none	none	none	
null	create	closed	region is on FS and not open on any RS but may have a suggested RS assignment	none or suggested	present	none	Create hdfs meta in region (atomic dir rename)
closed	recover	recovering	region is not assigned or opened and may have logs split ready for replay	none or suggested	present (could be dirty)	none	blocks until region's logs have been split, may have no recovered logs
recovering	assign	assigned	region is not opened on any rs; entry in region intended state now as target region server	yes	present	none	region is assigned to rs in intended state.
assigned	open	opened	region is open for RW on specified RS	yes	present	open RW	region is opened (replays recovered edits and made ready for RW) on assigned rs.
opened	close	closed	region is on FS and not open on any RS but may	none or suggested	present	none	reigon is closed on

			have a suggested RS assignment				assigned rs. RS is still may have old assignment
opened	flush	opened	no real intended region state change	yes	present	open RW	region is not allowed to transition until flush completes or aborts
opened	compact	opened	no real intended region state change	yes	present	open RW	region is not allowed to transition until compaction completes or aborts
opened	snapshot	opened	no real intended region state change	yes	present	open RW	snapshotregion - block writes, flush, capture snapshot metadata in temp, atomic tmp dir move unblock writes
opened	move	assigned	region is not opened on any rs; entry in region intended state now as target region server	yes	present	none	update to region's RS assignment intended state. open/close intented state is not changed. Worker closes on old rs and opens on new rs.
opened	split	splitParent	region is open for RW on an rs; it is the process of creating daughter regions	yes, also marked split parent	present	open RW	change intended state of

							region to splitting. tell rs to split region but not commit (create daughters on FS)
splitParent	splitComplete	archived	region is "deleted" and in the archive. It is not open or assigned to an RS	no	archived	none	created new closed daughters regions in intended state, update parent regions intended state to archived (close parent, open daughters)
opened	merge	mergeParent	region is open for RW and has intended state updated with regionserver assigned the merge	yes, marked merge parent, assigned to merge RS	present	open RW on original RS, not on merge RS	updated indended state
mergeParent	mergeMove	merging	region is open for RW on merging RS	yes	present	open RW on merge RS, not open on original RS	move region to merging RS
merging	mergeComplete	archived	region is "deleted" and in the archive. It is not open or assigned to an RS	no	archived	none	move to merge target RS, merges move both regions to

							same RS) create new child, in intended state
null	merge	mergeChild		yes	present	none	
mergeChild	mergeComplete	assigned	region is not opened on any rs; entry in region intended state now as target region server	yes	present	none	FS meriging operations completed
null	createDaughter	splitDaughter	daughter region fs metadata written, not open for reads/writes, assigned on parent region	yes	present	none	create daughter fs metadata, assign to parent region
splitDaughter	splitComplete	assigned	region is not opened on any rs; entry in region intended state now as target region server	yes	present	none	

Appendix: Graphviz code for state machines.

http://sandbox.kidstrythisathome.com/erdos/

```
digraph architecture {
  client ->  master [label="requests changes"]
  master ->  "master meta state"  [label="validates/updates"]
  "master meta state"  -> "master" [label="compares"]
  master ->  rs [label=updates]
  rs -> "master actual state" [label=reports]
  "master actual state" -> master  [label="compares"]

  { rank=same; master "master meta state" "master actual state" }
}

digraph table {
  start -> enabled [label="create"]
  start -> enabled [label="clone"]
  enabled -> locked [label="lock"]
  locked -> enabled [label="unlock"]
  locked -> locked [label="alter"]
```

```
  locked -> locked [label="snapshot"]

  enabled -> disabled [label="disable"]
  disabled -> disabled [label="alter"]
  disabled -> disabled [label="snapshot"]
  disabled -> disabled [label="restore"]

  disabled -> deleted [label="drop"]
  disabled -> enabled [label="enable"]

  rankdir = LR
}


digraph region {
  null -> closed [label="create"]
  assigned -> opened [label="open"]
  opened -> assigned [label="close"]
  opened -> opened [label="flush"]
  opened -> opened [label="compact"]
  opened -> opened [label="snapshot"]
  opened -> assigned
[label="move"]

  opened -> splitParent [label="split"]
  splitParent -> archived [label="splitComplete"]
  opened -> mergeParent [label="merge"]
  splitDaughter -> assigned [label="splitComplete"]
  null -> splitDaughter [label="createDaughter"]

  mergeParent -> merging [label="mergeMove"]
  merging -> archived [label="mergeComplete"]
  null -> mergeChild [label="merging"]
  mergeChild -> assigned [label="mergeComplete"]

  closed -> recovering [label="recover"]
  recovering -> assigned [label="assign"]

  unknown -> quarantined [label=failed]
  unknown -> closed [label=assert]
 }
```

```
digraph extendedRegion {
  null -> closed [label=create]
  assigned -> replaying [label=openWO color=red]
  replaying [color=red];
  assigned [color=red];
  assigned -> opened [label=openRW]
  replaying  -> opened [label=openRW color=red]
  opened -> assigned [label=close]
  opened -> opened [label=flush]
  opened -> opened [label=compact]
  opened -> opened [label=snapshot]
  opened -> assigned [label=move]
  recovering -> assigned [label=assign]
  closed -> assigned  [label=assign color=red]

  restoring [color=blue]
  opened -> restoring [label=snapshotRestore color=blue]
  restoring -> opened [label=snapshotRestoreComplete color=blue]


  opened -> splitParent [label=split]
  splitParent -> archived [label=splitComplete]
  splitDaughter -> assigned [label=splitComplete]
  null -> splitDaughter [label=createDaughter]

  opened -> mergeParent [label=merge]
  mergeParent -> merging [label=mergeMove]
  merging -> archived [label=mergeComplete]
  null -> mergeChild [label=merging]
  mergeChild -> assigned [label=mergeComplete]

  unknown -> quarantined [label=failed]
  unknown -> closed [label=assert]
 }
```