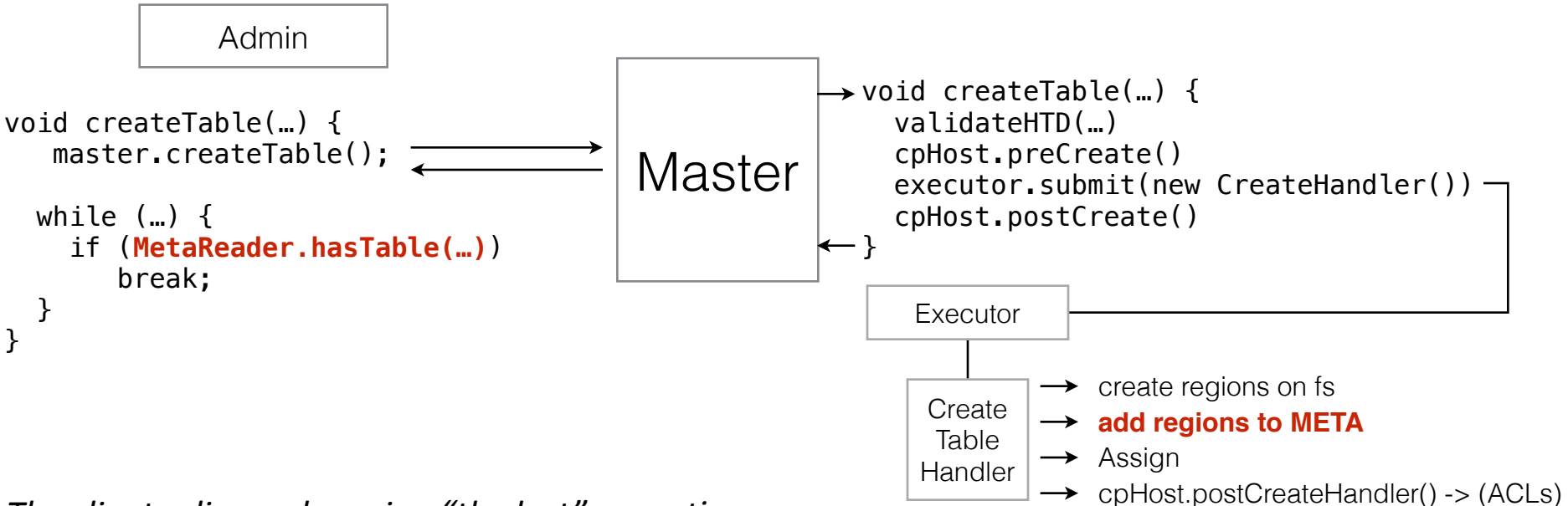
Procedure v2 - Overview

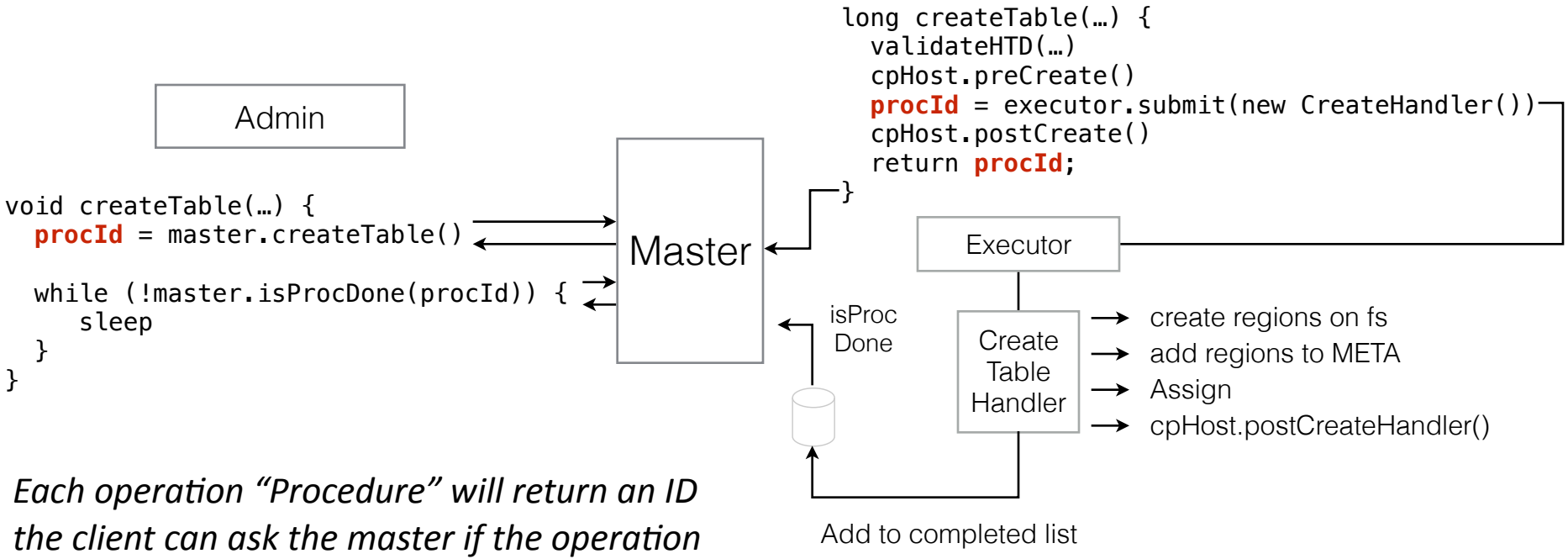
March 2015

Problem 1 - Sync Client



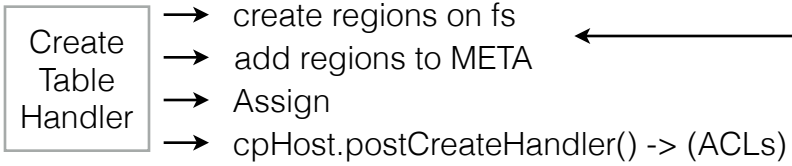
The client relies on knowing "the last" operation performed by the server to create a fake sync behavior.
result: if the crete handler is slow ACLs & co tests will be failing

Solution 1 - Sync Client



Each operation "Procedure" will return an ID the client can ask the master if the operation with the given ID is completed.
(We can freely change the server side code order)

Problem 2 - Multi-Steps proc & Failures



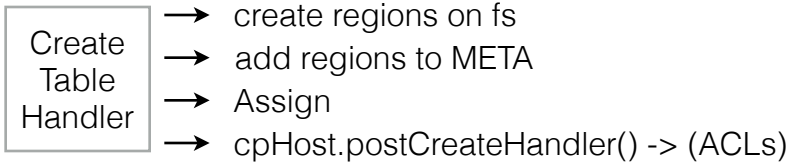
if we crash in between steps. we end up with half state File-System present, META not present
***hbck** MAY be able to repair it*

if we crash in the middle of a single step (e.g. create N regions on fs)
***hbck** has not enough information to rebuild a correct state.*

Requires manual intervention to repair the state

Solution 2 - Multi-Steps proc & Failures

Rewrite each operation to use a State-Machine



…each executed step is written to a store

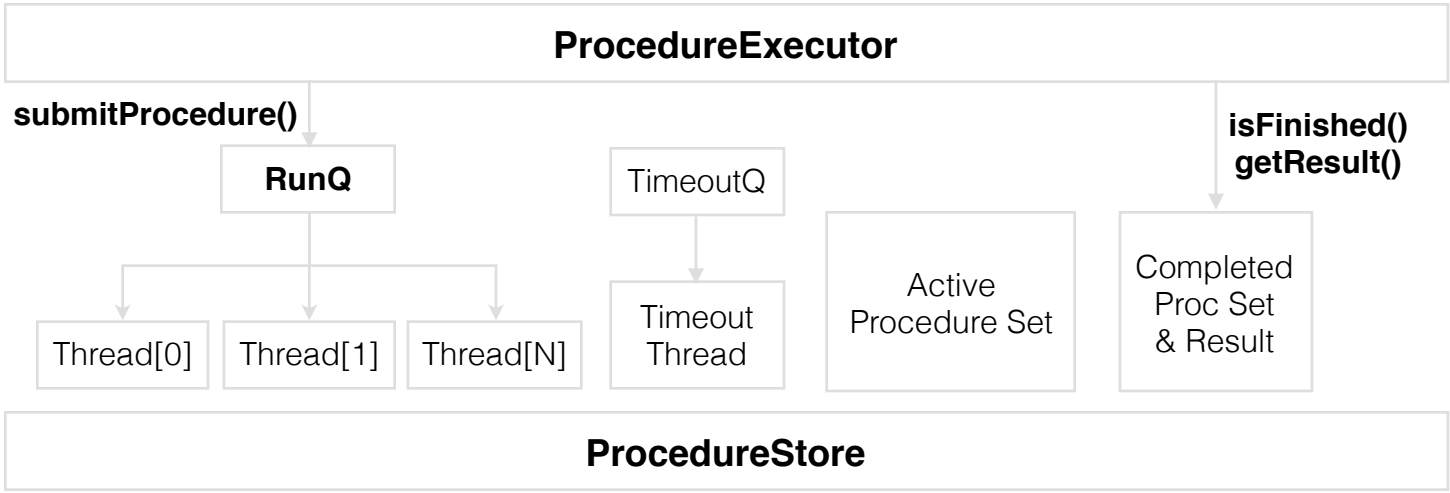
if the machine goes down
we know what was pending
and what should be rolledback
or how to continue to complete the operation

Procedure v2 - Implementation Details

- New “hbase-procedure” package
  - depends on: hbase-common
  - used by: hbase-server *(Master only at the moment)*

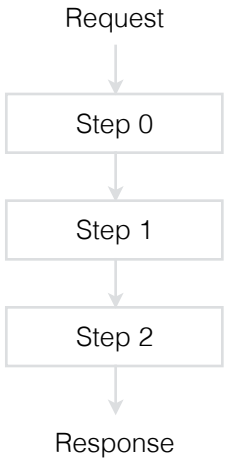
- *Procedure: execute(), rollback()*
- *ProcedureExecutor: sumitProcedure(Procedure), isFinished(procId), getResult(procId)*
- *ProcedureStore: load(Proc), insert(Proc), update(Proc), delete(Proc)*

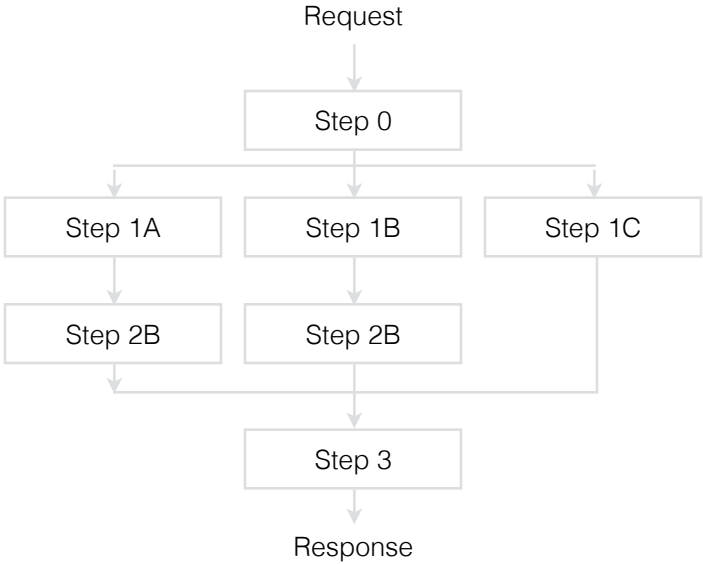
Procedure v2 - Core Components



Procedure v2 - What a procedure can do?

- A Procedure describes an “operation”
- A Procedure may be divided in sub-Procedures (“steps of the operation”)



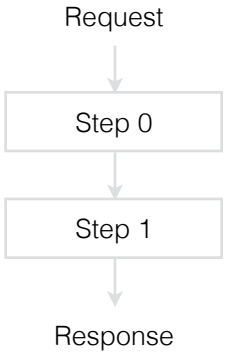


Procedure v2 - Code Example

- “Procedure” is the base object that allows you to do crazy stuff
- there are helpers that allows you to simplify the code e.g. SequentialProcedure

```
class Step0 extends SequentialProcedure {
  protected Procedure[] execute() {
    return new Procedure[] { new Step1(); };
  }
  protected void rollback() { ... }
}
```

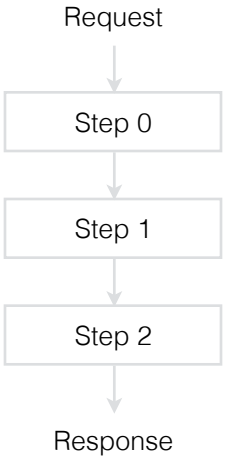
```
class Step1 extends SequentialProcedure {
  protected Procedure[] execute() {
    return null; // no other sub-procs
  }
  protected void rollback() { ... }
}
```



Procedure v2 - Code Example

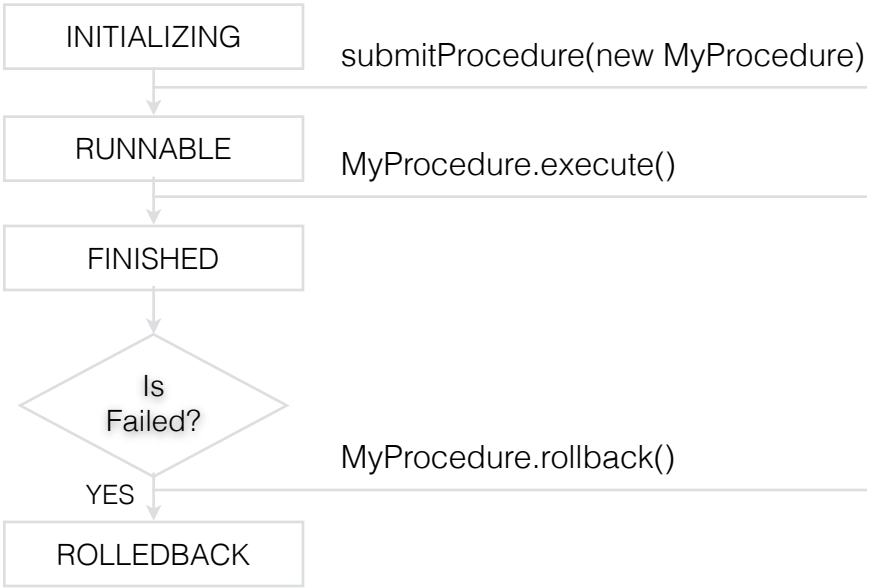
- A variant of a “SequentialProcedure” is “StateMachineProcedure”
- Less Procedure classes around, and a bit more readable for simple procs.

```
class MyProcedure extends StateMachineProcedure<MyState> {
  enum MyState { STEP_0, STEP_1, STEP_2 }
  protected Flow executeFromState(MyState state) {
    switch (state) {
      case STEP_0: setNextState(STEP_1); return Flow.HAS_MORE_STATE;
      case STEP_1: setNextState(STEP_2); return Flow.HAS_MORE_STATE;
      case STEP_2: return Flow.NO_MORE_STATE;
    }
  }
  protected void rollbackState(MyState state) {
    switch (state) {
      case STEP_0: break;
      case STEP_1: break;
      case STEP_2: break;
    }
```



Procedure v2 - Procedure “Framework” States

- The procedure (simplified) lifecycle is:



At each “state change” the Procedure State is written to a store.

The server may go down during the execution of the Procedure on restart the procedure.execute() will run again, since the state is not changed.

The server may go down during the execution of the Procedure on restart the procedure.rollback() will run again, since the state is not changed.

All the Steps you write in execute()/rollback() must be idempotent

Procedure v2 - Procedure Store / WAL

- ProcedureStore is an interface with load(), insert(), update, delete()
- Current Implementation is a simple WAL, which allows:
  - if no proc running/pending, throw away all the WALs
  - if all the procs were updated and added to the new WAL, throw away the old ones

state-001.log

INIT	Procedure 0	
UPDATE	Procedure 0	
INSERT	Procedure 0	Procedure 1
…		

state-002.log

UPDATE	Procedure 1
UPDATE	Procedure 0
INIT	Procedure 2
…	

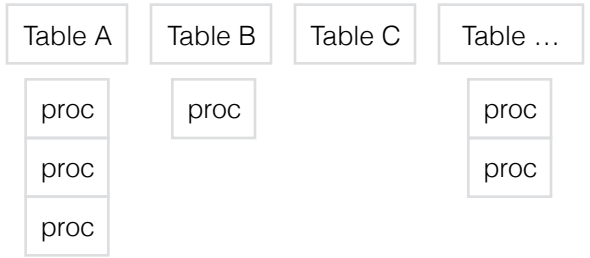
- Replay is from **new** to **old**
- A procedure can start only when the INIT is found (INIT = user submit) or we read all logs.

Procedure v2 - HMaster Implementation Details

- Master.proto
  - A new “proc_id” field was added to the Response of each operation
  - A new getProcedureResult(procId) rpc was added, to allow the client to get the result of the specified procedure (sync-client).
- HMaster
  - has a startProcedureExecutor()/stopProcedureExecutor()
  - MasterProcedureEnv passed to the procedures to have access to HMaster instance.
  - *Handler replaced by *Procedure

Procedure v2 - HMaster Implementation Details

- The master has a custom run-queue, that knows about tables.
  - The Procedure have to implement a “TableProcedure Interface”
- Operations on different tables can be executed concurrently.
  - (If not the case you have to take a lock)
- Write Operations on a single table must be serialized
- Read Operations on a single table may be executed concurrent
  - (we don’t have any)



Procedure v2 - Client with sync support

- Binary Compatibility: **void** operationAsync(…)
  - HBase 1.1: Future<T> operationFuture(…)
  - HBase 2.0: Future<T> operationAsync(…)

```
Future<T> operationAsync(..) {
  OperationResponse response = master.operation(..);
  return new ProcedureFuture<Void>(this, response.getProcId()) {
    protected T convertResult(GetProcedureResultResponse resp) {
      // Operation.convert(resp.getData())
    }
    protected T waitOperationResult(..) {
      // If there is no proc support,
      // sleep until condition (e.g. is entry in META)
    }
  }
}
```

- We now have the ability, to know if we have a “client timeout”, an error on the server or more.
- We can also abort the operation with future.cancel()

```
T operation(..) throw TimeOutEx, IOEx {
   Future<T> f = operationAsync(..);
   return f.get();
}
```

Procedure v2/NotificationBus

- The Procedure v2/NotificationBus aims to provide a unified way to build:
  - Synchronous calls, with the ability to see the state/result in case of failure.
  - multisteps 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)
  - Replication for Admin operations

Procedure v2/NotificationBus - Roadmap

- Apache HBase 1.1
  - Fault tollerant Master Operations (e.g. create/delete/…)
  - Sync Client (We are still wire compatible, both ways)
- Apache HBase 1.2
  - Master WebUI
  - Notification BUS, and at least Snapshot using it.
- Apache HBase 1.3 or 2.0 (depending on how hard is to keep Master/RSs compatibility)
  - Replace Cache Updates, Assignment Manager, Distributed Log Replay,…
  - New Features: Coordinated compactions, Master ops Replication (e.g. grant/revoke)