# Coprocessor Design Improvements

HBASE-17732

Author: Apekshit Sharma
Date: 9/27/2017

## Introduction

This doc explains current design of Coprocessor feature in brief, few issues I noticed, and suggestions on how to fix them & further improve overall design.
(This doc is not yet well-formatted as a design doc, but once the corresponding coproc changes are committed  (HBASE-17732) , i'll evolve this into engineering documentation for Coprocessor feature.)

**TL;DR**
We are moving from

- Observer **is** Coprocessor
- FooService **is** CoprocessorService

To

- Coprocessor **has** Observer
- Coprocessor **has** Service

See code example in Main Design Change suggestions.

## Terminology

hooks = functions in observers. Named because third-party use these functions to "hook up" custom logic to internal code paths.

## Background

Coprocessors are well documented in the refguide.

Here we give a little background information on involved classes, their responsibilities, and relationship to each other.

- Main classes
    - Coprocessor (interface)
        - All Observer interfaces derive from Coprocessor interface.

        - Coprocessor Interface is a *Marker* Interface. It just has start/stop methods and enums for stages in the Coprocessor Lifecycle.
    - Observers (interface)
        - Contain hooks which third-party programs can override to inject functionality in various internal code paths. For e.g preCreateTable(...) will be called just before any table is created.
        - Current set of observers: *MasterObserver, RegionObserver, RegionServerObserver, WALObserver, EndpointObserver, BulkLoadObserver.*
    - CoprocessorEnvironment (interface)
        - Encapsulates a coprocessor instance and other information like versions, priority, etc.
        - Coprocessor implementations use it to get access to tables.
        - Four main implementations: *MasterEnvironment, RegionEnvironment, RegionServerEnvironment, WALEnvironment.*
    - CoprocessorHost (abstract class)
        - Responsible for loading coprocessors
        - Four concrete sub-classes: MasterCoprocessorHost, RegionCoprocessorHost, RegionServerCoprocessorHost, WALCoprocessorHost
        - Each host is tied to corresponding environment type using template argument 'E'.

## Problems

- CoprocessorEnvironment has `'Coprocessor getInstance()'`. Since Observer types which can be handled by an environment are not statically tied to it, coprocessor hosts (which are statically tied to Environment) don't know which kind of coprocessors are relevant to them, i.e. MasterCoprocessorHost is tied to MasterEnvironment, but it doesn't know that it can only handle MasterObserver(s). As a result:
    - Problem 1: All hosts load all observers i.e. MasterCoprocessorHost will also load RegionObserver and other observers.
    - Problem 2: Hosts use runtime checks likes `'observer instanceOf ExpectedObserver'` in execOperation and other functions to filter out incompatible observers.
    - Problem 3: Many redundant functions in every implementation of coprocessor host.
- Observer **extends** Coprocessor (inheritance)
    - Problem 4: Any third-party coprocessor which wants to use many observers will have to extend all of them in same class. For eg.
      `class AccessController implements MasterObserver,`

```
    RegionObserver, RegionServerObserver,
EndpointObserver,
    BulkLoadObserver, AccessControlService.Interface,
    CoprocessorService
```
That results in big classes with 100+ functions.

## Proposed Solutions

- There are 6 types of observers (listed in ‘Background’ section above), but just 4 types of CoprocessorEnvironment. So some XEnvironment has to be handling multiple Observers (RegionEnvironment serves RegionObserver, EndpointObserver and BulkLoadObservers). Our aim is to statically tie environment to types of observers it can serve. There are two alternative choices here:
    - Option 1: Limit to 4 types of Observers. That fits nicely in our pattern-of-4 (4 hosts, 4 environments, 4 observers) and will make the overall design simpler. Although it may look simple at surface, it’ll actually lead to a single large observer interface which will only grow and may contain 100s of hooks in future (master already has 100+)
    - Option 2: Use multiple observers to group together similar kinds of hooks. Like we have RegionObserver, EndpointObserver and BulkLoadObserver; we can have ScannerObserver, AMObserver, etc instead of single MasterObserver. Benefits being
        - Proper encapsulation of related hooks and separation from unrelated hooks
        - We can give different Stability guarantees for different set of hooks. Something which’ll make our CP compat management much easier.

I believe Option 2 to be better than Option 1, and the design changes suggested later are based on Option 2.

- For problem 4, we should replace inheritance with composition, so developers have choice to break out observer implementations into separate classes.

## Main Design Change suggestions

- Extend pattern-of-4 up to coprocessor.
  CoprocessorHost → Env → Coprocessor
- Tie each CoprocessorEnvironment to corresponding Coprocessor
- Use composition instead of inheritance between Coprocessor and Observers.

### Current design

Only changing parts are mentioned here. Anything not changing is represented by “...”

```
interface Coprocessor {
  ...
}

interface CoprocessorEnvironment {
  Coprocessor getInstance();
  ...
}

interface CoprocessorService {
  Service getService();
}

abstract class CoprocessorHost<E extends
CoprocessorEnvironment> {
  ...
}

interface RegionObserver extends Coprocessor {
  ...
}

interface BulkLoadObserver extends Coprocessor {
  ...
}

interface EndpointObserver extends Coprocessor {
  ...
}
```

## New design

```
interface Coprocessor {
  ...
}

// Extend pattern-of-4 to coprocessors.
interface RegionCoprocessor extends Coprocessor {
  RegionObserver getRegionObserver();
  BulkLoadObserver getBulkLoadObserver();
  EndpointObserver getEndpointObserver();
  Service getService();
}
```

```
// Tie coprocessor to environment
interface CoprocessorEnvironment<C extends Coprocessor> {
  C getInstance();
  ...
}

abstract class CoprocessorHost<C extends Coprocessor, E extends
CoprocessorEnvironment<C>> {
  ...
}

// Doesn’t extend coprocessor
interface RegionObserver ~~extends Coprocessor~~ {
  …
}

// Doesn’t extend coprocessor
interface BulkLoadObserver ~~extends Coprocessor~~ {
  …
}
```

## How does it fix our issues?

- Fix #1: CoprocessorHost is now tied to types of coprocessors it can serve by template argument C. During load time, it can ignore any coprocessors which don’t match.
- Fix #2 and #3: Pull the duplicate functions into CoprocessorHost class. Individual host subclasses can use these directly. One interesting part here is ObserverGetter<C, O>. For any specific hook, say in observer O, subclasses specify ObserverGetter<C, O> so that the shared methods can extract observer O from coprocessor C.
- Fix #4: Choosing composition over inheritance, by adding getter functions in coprocessors (e.g. getRegionObserver()), implementations can now break up observer implementations into separate classes. For e.g. our AccessController will now just be:
  ```
  class AccessController implements
  AccessControlService.Interface,
      CoprocessorService
  ```

## Migrating existing CPs to new design

There’s a simple and small fix that can migrate existing coprocessors to the new design.

If  we had the following observer in the old design:

```
class FooObserver implements RegionObserver {
...
...
}
```

It can be “made to work” with the new design like this:

```
class FooObserver implements RegionCoprocessor, RegionObserver {
  public RegionObserver getRegionObserver() { return this; }
  ...
  ...
}
```

However, note that this is only a workaround to quickly migrate ~100 CPs in our code base to new design without creating new classes and files. **New CPs should NOT follow this pattern.**

## Additional Benefit

- Cleaner solution to HBASE-17106.
- We can have multiple observer interfaces for each environment now. For e.g We can break our single monolithic MasterObsever (~150 functions) to multiple observer interfaces - ScannerObserver, AMObserver, etc.
- These observers can be assigned different compatibility guarantees. For instances, new hooks by Backup feature could have been put into separate Observer and marked IS.Unstable, while features which have hardened over years can be marked IS.Stable.
- Only the coprocessors corresponding to hosts which support endpoint service will have “getService()” method. So WALCoprocessor which cannot support service doesn’t have one. That may look minor thing. But if our design can cleanly convey what is and isn’t supported, that’s beautiful and powerful and helpful for downstream developers.