**Multi-HBase Cluster Client**

**Design Document**

# Use Case

The use case is to use HBase as the backend storage solution of a system that has SLAs requirements that requires more reliable response times then any one HBase cluster can provide.

The reason a single HBase cluster can not provide reliable response times is because a single rowkey in HBase is pinned to a single Region Server and that region server can fail or experience a full GCs. Both a GC and Region Server Failure would result is a disruption of service and a period of time.

With the addition of HBase Master-Master and Master-Slave we now have the option to use two or more HBase clusters to reduce the risk of down time. However up until now to take advantage of a Master-Master or Master-Slave solution a custom client would have to be implemented by the development team of the application.

# Goal

This design document is to offer a solution for a common implementation of this multi-HBase client that will allow for the following benefits:

* Be able to switch between single HBase clusters to Multi-HBase Client with limited or no code changes. This means using the HConnectionManager, Connection, and Table, (Connection and Table hich are the new classes for HConnection, and HTableInterface in HBase version 0.99).
* Offer thresholds to allow developers to decide between degrees of strongly consistent and eventually consistent.
* Support N number of linked HBase Clusters

# Technical Approach

In the following sections we go over additional configurations and code changes to allow us to reach the goals above. But before going in to that, lets define some terms that we will use through out the document.

|  |  |
| --- | --- |
| **Term** | **Definition** |
| Primary HBase Cluster | We will go into why a primary cluster later in this document. But for now consider the primary HBase Cluster as the Cluster you first try. |
| Failover HBase Clusters | These are N number of HBase Cluster that we can go to when the Primary Cluster is down |
| Strongly Consistent | Means that when you request a value it will be the most up to date as possible |
| Eventual Consistence | Means when you request a value it way me right or it may be out of date, but at some point it will be right. |

## Configuration

The key to this solution avoiding coding change to the end developer is through additional configurations parameters. There are only a couple but they are very powerful so we will take time to drill down into each one of them.

**failout.clusters**

This is a common separated list of cluster names that will represent the clusters we will fail over too if our primary cluster is not responsive. The design of this parameter is very much like how Flume defines Sinks, Sources, or Channels for a flume agent.

Now you will note that there is no primary cluster definition, that is because we will leave the normal configurations as pointing the client to the primary HBase Cluster.

**hbase.failover.cluster.{failover.cluster.name}.{overrided client parameter}**

These parameters will allow us the option to override client configurations from the primary HBase Cluster configurations for the one that differ in the failover clusters.

Here is an example that will allow us to point to different zookeeper quorums for the different HBase clusters

|  |
| --- |
| hbase.failover.clusters=foo,bar  hbase.zookeeper.quorum=1.0.0.1  hbase.failover.cluster.foo.hbase.zookeeper.quorum=1.0.0.2  hbase.failover.cluster.bar.hbase.zookeeper.quorum=1.0.0.3 |

**hbase.failover.mode**

This parameter is to identify that we are running in Master-Master or Master-Slave. The two options would be “Master-Master” or “Master-Slave”. This will affect if puts/deletes will be sent to a failover cluster.

**hbase.wait.time.before.accepting.failover.result**

This parameter will decide how many milliseconds the client will wait for the primary request to return before we accept a response from a fail over cluster.

There are several options for the developer here. If this value is set to 0 then the client will accept the first response that returns no matter if it comes from the primary cluster or if it comes from the failover cluster. The advantage of setting this to 0 is the user will get an answer as fast as possible, but the down side is we may turn down a primary response for a failover response because of a couple millisecond difference. You may ask, “why is this a big deal?” The answer is the responses from a failover cluster have eventual consistent values so the value may not be up to date, where as the primary cluster will have strongly consistent values.

An alternative approach would be to set this value to 30 milliseconds, which will give the primary cluster more then enough time to get a response on a common get that even has to hit disk. With a delayed acceptance of a failover response we get a much higher percentage of strongly consistent values. In the optimal experience this will allow the user to get strongly consistent values in most cases until there is a major GC or Region Server failure, and only then will the user get eventual consistent values.

**hbase.wait.time.before.request.failover**

This parameter can be used in combination with “*hbase.wait.time.before.accepting.failover.result*” but is very different. This parameter will define a delay before the addition request is sent out.

If this value is set to 0 then in all cases a get/scan command will be sent out to the Primary HBase cluster and all Failover Clusters, which will give us our highest possible chance of our lowest response times.

However there are reasons for setting this to a higher value like 30 milliseconds.

1. Reduce the number of out going response from the client. Which consume CPU and network resources.
2. Better leverage the block caches of the clusters. One can imagine a design where depending on the rowkey a different primary HBase Cluster is defined. This would allow the block cache to focus on caching different information across the two clusters. If a get always goes to both clusters then the same data is cached in both cluster.

|  |
| --- |
| *Now note, most of the cases where reads are slowed down because of GCs and Region Server failures will be addressed from the HBASE-10070. This documents design for a multi-cluster client solution will be independent from HBASE-10070 solution and in fact be improved through it. However, it is important to note HBASE-10070 because it will provide faster fail over with in the primary cluster.* |

**hbase.wait.time.before.mutating.failover**

This parameter defines how long the client waits for a put/delete submission to complete from Primary HBase Cluster before sending it to the failover clusters.

One thing to note is we will be setting the timestamp in the client so even if the put or delete make it to all clusters they will resolve correctly.

So if this parameter is set to 0 then in all cases a put or delete will be sent to all clusters. The down side to this is time and bandwidth and more stress on all the WALs on all the Clusters.

If we sent this parameter to something high like 10 milliseconds we rarely have to send it to anything but the Primary HBase Cluster.

|  |
| --- |
| *Also not increment command is not included in this list, this is because increment is a little more complex to get correctly with multiple clusters. For now increment commands will only be sent to the primary cluster.* |

## HBase Core Changes

**Connection / HConnectionMultiCluster**

This will be a new class that will extends Connection/*HConnection*. It will also have Connections/*HConnections* for all the HBase clusters configured for primary and failover. Then it returns a Table/*HTableInterface* it will return a *HTableMultiCluster*. The developer should never directly use this class, unless they want to do specific multi-cluster functions like may be list cluster names. But the hope is they will just have in their code Connection/*HConnection* so they can switch between single or multi-cluster mode without code changes.

**Table / HTableMultiCluster**

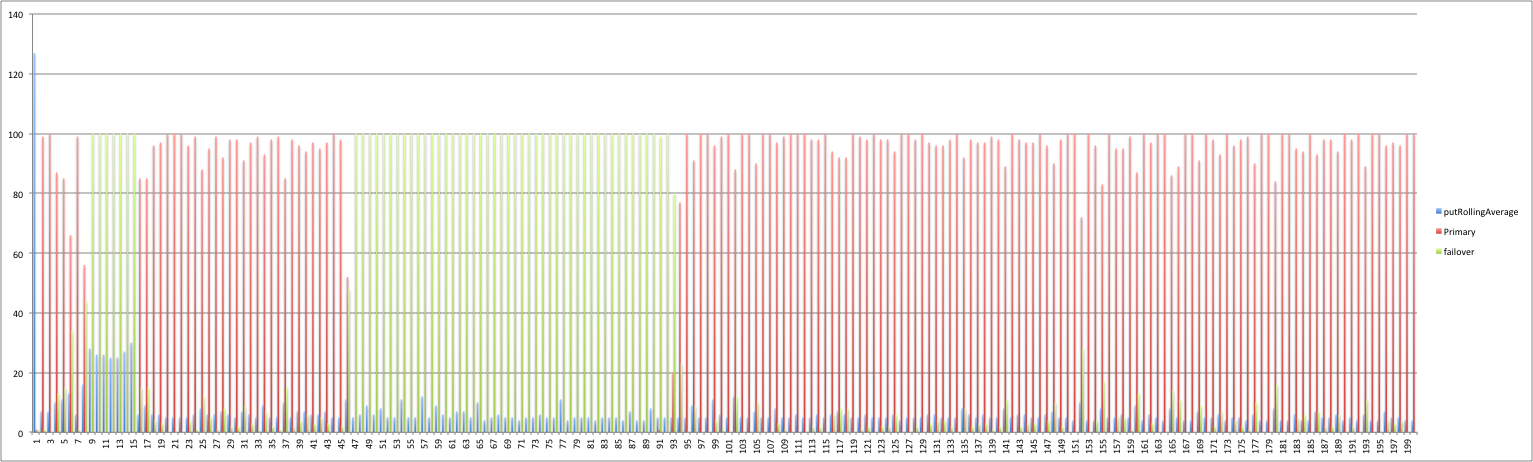
This will override *HTable*, which implements Table/*HTableInterface*. It will override functions like put, get, delete, checkPut, … and add logic for failover. Also the goal is the normal developer not use this class directory but use it through Table/*HTableInterface* so that they can switch between single and multi-cluster mode with out code changes.

**HConnectionManager**

The only change here is to check if the incoming config is one that is request to run in multi-cluster mode and if so construct and return an instance of *HConnectionMultiCluster.*

# Initial Results

The initial results from the beta implementation are positive. There is a lot of work still to be done, but these times are a lot better then any single cluster can give you.

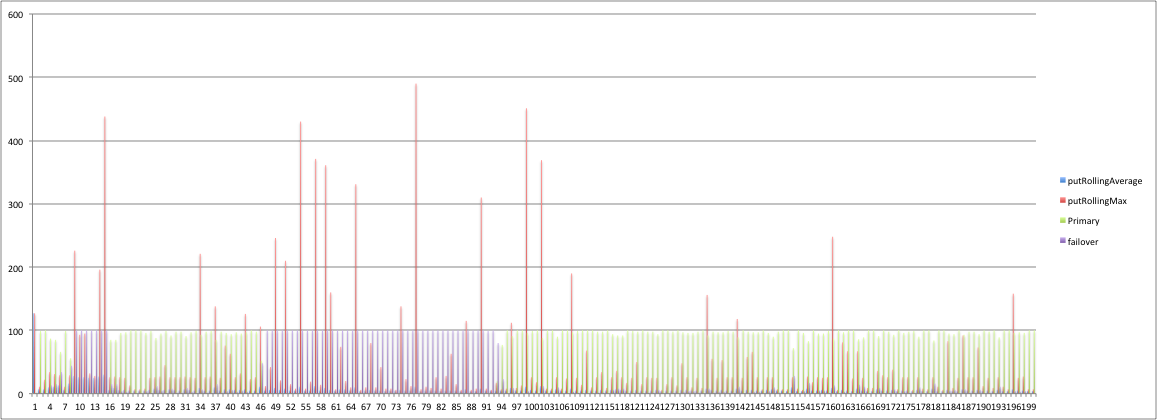


The image above is the initial results of a 20k put test with two clusters. The blue lines are the average time for the puts to commits to a Cluster.

Now the red lines are puts to the primary cluster and green lines are puts that are send to the failover cluster. When you see green lines they are the results of the primary cluster being shut down or bounced.

The cool thing here is we are keeping a pretty good average even in the case of extreme failure. There is an increase in the time to put to the failover cluster but that is because I set the parameters to only go to the failover cluster after waiting 20 milliseconds.

Now the next graph is a little more interesting because it show local maxes.



Note that there are cases where puts are taking many 100s of milliseconds; we are going to look at this in more detail in future versions. We would like to see this go down much more.