KIP-46: Self Healing Kafka

Status

Current state: *Under Discussion. This KIP currently has mostly architectural details. Finer details will be added once we agree on the core design.*

Discussion thread:

JIRA:

Motivation

Currently if the Kafka cluster loses a broker, there is no mechanism to transfer replicas from the failed node to others within the cluster other than manually triggering ReassignPartitionCommand. This is especially a problem when operating clusters with replication factor = 2. Loss of a single broker means that there is effectively no more redundancy which is not desirable. Automatic rebalancing upon failure is important to preserve redundancy within the cluster. This KIP effectively automates commands that operators typically perform upon failure.

Public Interfaces

Proposed Changes

This problem is broken down into 2 parts. Discovery of failed brokers and mitigation. The design for each is documented separately.

Part 1: Failure Discovery

How do we discover that a broker has failed? It is particularly important to avoid false alarms i.e. if a broker is being bounced it should not count as a failure and prematurely trigger a rebalance.

We feel that the following approach is better than a periodic scan (documented in the rejected alternatives section).

Explicit notification

The KafkaController receives an onBrokerFailure callback everytime a broker dies. This is an explicit signal that we should leverage because this is also used to effect leadership changes. The proposal is to introduce a new path in ZooKeeper that is managed by only the controller. onBrokerFailure callback can generate a znode in /admin/failed_brokers. This can have a timestamp corresponding to when this callback was first received by a controller.

```
/admin/failed_brokers/<broker_id>
{ "failure_timestamp" : xxx }
```

The current controller can schedule a task to which executes at "failure_timestamp + failure_interval". At this time if the znode still exists, that node is considered dead and we can go to the mitigation phase described below. If the failed broker re-enters the cluster before the failure_internal (30 mins, 1hr?), the corresponding znode is deleted and the scheduled rebalance task is cancelled.

The list of dead nodes needs to be persisted to survive controller handoffs and any newly elected controller can simply pick up the persisted list in ZK and reuse the initial failure time.

Part 2: Mitigation

Once the failure detection piece is solved we need to decide how best to solve the problem of moving replicas to hosts. If a broker has been deemed to be failed, the controller should generate a list of topic-partitions that it owned. In large brokers such as the ones we run within LinkedIn, each failed broker likely has several hundred partitions. We will saturate the network if we bootstrap all these at the same time. So some throttling is needed.

There are 2 approaches here:

1. **Move all partitions simultaneously:** This is simpler to implement but has the obvious downside of potentially saturating the network. We can limit the impact on network by throttling the bootstrap traffic from replicas. However, this ensures that the average catch up time is high for all partitions.
2. **Move partitions in chunks:** The controller should be able to breakdown tasks in granular chunks (i.e. 10 partitions at 1 time, 2 partitions per broker etc..). We can keep it simple and say that the controller will move x partitions at a single time.Once one task is done, start processing the next task. This is a bit trickier to implement but is the better solution because it generates fewer moving parts at one point of time. Retry/failure handling becomes easier because we operate on smaller chunks.

We recommend the second approach.

Reassign Partitions

Reassign partitions is an already available feature used in Kafka to move partition replicas between different hosts. Unless there is a strong reason not to, we should leverage this approach rather than reinvent.

This command creates znodes under /admin/reassign_partitions for the topic-partitions that need to be moved. The controller parses each of these partitions and makes sure that the partition replica changes are propogated within the cluster. More documentation is available here. The controller can use it's own internal APIs to trigger this.

Since the volume of partitions that can be moved is quite large we should consider making changes to reassign partitions to breakdown a large reassignment task into smaller chunks to avoid creating very large znodes. Once the assignments are written in ZK, the controller can parse each one sequentially like independent partition assignment tasks.

This has a nice side effect of improving the reassign partitions tool. Currently it breaks when it generates znodes larger than 1M for large brokers. This requires operators to manually breakdown the task into separate iterations that are long running.

Here's the how the ZK path can look:

```
/admin/reassign_partitions/seq_1
   { topic-partitions to move, isAutoGenerated : true/false }
/admin/reassign_partitions/seq_2
   { topic-partitions to move, isAutoGenerated : true/false }
/admin/reassign_partitions/seq_1
   { topic-partitions to move, isAutoGenerated : true/false }
```

The controller parses each of the chunks sequentially and deletes them when the chunk is processed.

Workflow

Let's walk through a sample sequence of events.

1. Broker X has a hard crash. Controller receives onBrokerFailure notification.
2. Controller persists failed_broker node to ZK under /admin/failed_brokers
3. Controller schedules a TimerTask that will generate partition reassignment /admin/reassign_partitions.
4. If Broker X is back online before the TimerTask fires, the task is cancelled and the znode under /failed_brokers is deleted.
5. When the task is executed, it computes a list of assignments for all partitions hosted by that broker. The list is broken down into chunks (configurable size) and persisted as sequential znodes under /admin/reassign_partitions
6. The controller processes each reassign task sequentially. When a reassign is completed, it moves on to the next task.

Completeness definition for a task

After the task is started, we need to wait on it to complete. Complete means that the replicas have caught up in ISR (once).

Broker restart after some assignments have been processed

To address this case, we should check if the partitions in a task all have the desired number of replicas before starting it. i.e. if Broker X is back online and catching up, there is no point in executing any subsequent jobs that have been queued up. However if the task is already running it is simpler to let it complete. Since tasks are not very large, they should not take very long to complete. The controller should only cancel tasks with "isAutoGenerated":true.

Deletion of failed_nodes znode

How do we GC the failed_nodes entries. They are deleted in 2 cases:

- If the failed broker comes back online.
- After the reassign tasks get scheduled, the znode can be deleted since the tasks themselves are persisted in ZK.

Controller failures

If a controller fails, the newly elected controller should rebuild it's internal state based on the nodes in /failed_brokers and the auto scheduled reassignment tasks.

There is a case where the controller fails after generating the reassignment tasks but before deleting the corresponding entry in /failed_brokers. There are a couple of ways to solve this:

1. The new controller can simply compare the assignments with the failed brokers. If the assignment has a reference to the failed broker, the controller knows which failed node created it. If it finds all tasks for a failed node, then it can simply assume the assignments are sufficient and delete the znode for that broker.
2. It should also be fine to let the controller simply reschedule the assignment tasks. Since the tasks are executed sequentially the previously executed tasks complete before these. Before scheduling any task we check to see if assignments are already satisfied. This will be true and hence the task can be skipped.

Option 2 is simpler.

Quotas

Should we throttle replica bootstrap traffic? This is tangentially related but we can actually throttle inter broker replication traffic to say "50%" of available network. Regardless of the number of failed replicas we have, we can impose a hard limitation on the total amount of network traffic caused by auto-rebalancing. Given that we plan to move only a set of partitions at one time this shouldn't really be a problem. Rather a nice to have.

Auto-Expansion

This doesn't really solve the problem of load balancing existing assignments within a cluster. It should be possible to ensure even partition distribution in the cluster whenever new nodes are added i.e. if I expand a cluster it would be nice to not move partitions onto them manually. This KIP does not attempt to solve that problem rather it makes it possible to tackle auto-expansion in the future since we could reuse the same pattern.

Configs

The following configs can be added.

- kafka.heal.chunk.size - Maximum number of partitions that will be moved in a chunk.
- kafka.heal.failure.interval - Time after which a node is declared to have failed.

In addition, we can also add a quota config for replica fetchers if we choose to tackle that in this KIP.

Compatibility, Deprecation, and Migration Plan

This should not impact existing users at all. Old behavior can be preserved by simply setting the failure interval to -1.

Behavior of reassign partitions:

- Older versions of reassign partitions may no longer work. New jars corresponding to this release will have to be used.
- While upgrading the cluster, we should make sure not to manually generate any reassign tasks. The new controller may not parse those tasks so they need to be rescheduled.

Rejected Alternatives

Periodic Scans

In this approach, the controller can identify dead replicas by iterating through all topics periodically and identify all the failed replicas based on the information under /topics/<topic_name>. If any replica is not online, write that replica to /admin/failed_brokers. The rest is similar to the approach described above. This does not rely on receiving an onBrokerFailure callback. However, since a significant amount of controller behavior depends on getting these callbacks right, it is perhaps more apt to use that as a direct signal. There are a few other challenges to this approach:

- Since the failure detection thread only sees a snapshot It is possible for it to think certain replicas are dead when they are in fact not. For example: a node bounced twice can be recognized as a failed node. This is possible during normal operation for e.g. rollback a broker soon after deployment.
- There is also a delay to this approach bounded the frequency of the scan. Frequent scans can cause a high volume of ZooKeeper activity since the scan approach scales with the number of topics and not the number of replicas. For clusters with a large number of partitions (40k+ topics at LinkedIn) the controller will read several hundred znodes.

Fast detection

After some discussion we decided against this proposal because of the added complexity. This is still worth discussing though.

Each of the proposals rely on a replica being down for a configurable period of time. Is this the right approach in all circumstances? In this section, we discuss some ideas to speed up the failure detection time and still not trigger rebalancing when not needed.

The following are perhaps the most common causes of hard broker failures:

- Disk failures - Upon receiving IO exceptions from Log.scala, brokers call System.exit. It should be possible to proactively report a failure i.e. this is not a rolling bounce, reassign my partitions as soon as possible.
- kill -9/power failure - In this case, it is not possible to report anything before going down. Hence reporting an failure proactively doesn't really work here (Captain Obvious statement).

What if we modify the above to report clean shutdowns instead of failures? In this case, each broker writes an entry to (say) /admin/rolling_bounce prior to shutting down cleanly. Most cases of clean shutdown are because of rolling bounces. Upon receiving an onBrokerFailure() notification, the controller can check to see if the corresponding rolling bounce znode exists. If not, the node did not exit cleanly and the controller can begin the rebalance immediately. If it does exist, the broker can wait for a period of time for the node to return. If the node does not come back in 30 minutes (configurable), the partitions can be reassigned and the corresponding entry can be removed from /admin/rolling_bounce.

The upside of this solution is that we detect real failures almost immediately while also handling clean shutdowns in which the node does not rejoin the cluster.

```
/admin/rolling_bounce/<broker_id>
{"timestamp" : xxx}
```