KIP-708: Rack aware StandbyTask assignment for Kafka Streams

Status

Current state: *Accepted*

Discussion thread: here

Voting thread: here

JIRA: ⚠ Unable to render Jira issues macro, execution error.

PRs*:* https://github.com/apache/kafka/pull/10851 https://github.com/apache/kafka/pull/10802 https://github.com/apache/kafka/pull/11837

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

Motivation

Similar to many distributed systems, Kafka Streams instances can also be grouped in different racks. When Kafka Stream's standby task is properly distributed in different rack compared to the corresponding active task, it provides fault tolerance and faster recovery time if the rack of the active task goes down.

Below we will explore how other distributed systems implement rack awareness and what kind of guarantees they aim to provide.

Elasticsearch

Rack awareness in Elasticsearch works by defining a list of tags/attributes, called *awareness attributes* to each node in the cluster. When Elasticsearch knows the nodes' rack specification, it distributes the primary shard and its replica shards to minimize the risk of losing all shard copies in the event of a failure. Besides defining an arbitrary list of tags/attributes for each node, Elasticsearch provides a means of setting which tags/attributes it must consider when balancing the shards across the racks.

Example:

```
node.attr.rack_id: rack_one
node.attr.cluster_id: cluster_one
cluster.routing.allocation.awareness.attributes: rack_id,cluster_id
```

Besides, Elasticsearch provides "Forced awareness" configuration, a safeguard to prevent racks from being overloaded in case of a failure. By default, if one location fails, Elasticsearch assigns all of the missing replica shards to the remaining locations. In the case of limited resources, a single rack might be unable to host all of the shards. **`cluster.routing.allocation.awareness.attributes`** configuration can be used to prevent Elasticsearch from allocating replicas until nodes are available in another location.

Example:

```
cluster.routing.allocation.awareness.attributes: rack_id
cluster.routing.allocation.awareness.force.zone.values: zone1,zone2
```

In the example above, if we start two nodes with **`node.attr.zone`** set to **`zone1`** and create an index with five shards and one replica, Elasticsearch creates the index and allocates the five primary shards but no replicas. Replicas are only allocated once nodes with **`node.attr.zone`** set to **`zone2`** is available.

Hadoop

In the case of Hadoop, rack is a physical collection of nodes in the cluster, and it's the mean of fault tolerance, as well as optimization. The idea in Hadoop is that read/write operation in the same rack is cheaper compared to when the process spans across multiple racks. With the rack information, Namenode chooses the closest Datanode while performing the read/write operation, which reduces network traffic.

A rack can have multiple data nodes storing the file blocks and replicas. Hadoop cluster with a replication factor of 3 will automatically write a particular file block in 2 different Datanodes in the same rack, plus in a different rack for redundancy.

Rack awareness in the Hadoop cluster has to comply with the following policies:

- There should not be more than 1 replica on the same Datanode.
- More than 2 replica's of a single block is not allowed on the same rack.
- The number of racks used inside a Hadoop cluster must be smaller than the number of replicas.

Redis

Rack "awareness" in Redis is called "Rack-zone awareness" and it's very similar to Kafka Broker's rack awareness. Rack-zone awareness only works in a clustered Redis deployment, and it's an enterprise feature.

Rack-zone awareness works by assigning a rack-zone ID to each node. This ID is used to map the node to a physical rack or logical zone (AWS availability zone, for instance). When appropriate IDs are set, cluster ensures that leader shards, corresponding replica shards, and associated endpoints are placed on nodes in different racks/zones.

In the event of a rack failure, the remaining racks' replicas and endpoints will be promoted. This approach ensures high availability when a rack or zone fails.

Proposed Changes

This KIP proposes to implement similar semantics in Kafka Streams as in Elasticsearch. Rack awareness semantics in Elasticsearch seems the most flexible and can cover more complex use-cases, such as multi-dimensional rack awareness. To achieve this, KIP proposes to introduce a new config prefix in StreamsConfig that will be used to retrieve user-defined instance tags of the Kafka Streams

```
/**
 * Prefix used to add arbitrary tags to a Kafka Stream's instance as key-value pairs.
 * Example:
 * client.tag.zone=zone1
 * client.tag.cluster=cluster1
 */
@SuppressWarnings("WeakerAccess")
public static final String CLIENT_TAG_PREFIX = "client.tag.";

/**
 * Prefix a client tag key with {@link #CLIENT_TAG_PREFIX}.
 *
 * @param the client tag key
 * @return {@link #CLIENT_TAG_PREFIX} + {@code clientTagKey}
 */
pubic static String clientTagPrefix(final String clientTagKey) {
        return CLIENT_TAG_PREFIX + clientTagKey
}
```

We will also add a new configuration option in `StreamsConfig`, which will be the means of setting which tags Kafka Streams must take into account when balancing the standby tasks across the racks.

```
public static final String RACK_AWARE_ASSIGNMENT_TAGS_CONFIG = "rack.aware.assignment.tags";
public static final String RACK_AWARE_ASSIGNMENT_TAGS_DOC = "List of client tag keys used to distribute standby
replicas across Kafka Streams instances." +
                                                            " When configured, Kafka Streams will make a best-
effort to distribute" +
                                                                                                    " the
standby tasks over each client tag dimension.";
```

Example configuration:

```
# Kafka Streams Client 1
client.tag.zone: eu-central-1a
client.tag.cluster: k8s-cluster1
rack.aware.assignment.tags: zone,cluster

# Kafka Streams Client 2
client.tag.zone: eu-central-1b
client.tag.cluster: k8s-cluster1
rack.aware.assignment.tags: zone,cluster

# Kafka Streams Client 3
client.tag.zone: eu-central-1a
client.tag.cluster: k8s-cluster2
rack.aware.assignment.tags: zone,cluster

# Kafka Streams Client 4
client.tag.zone: eu-central-1b
client.tag.cluster: k8s-cluster2
rack.aware.assignment.tags: zone,cluster
```

When **`client.tag.*`** dimensions are configured, Kafka Streams will read this information from the configuration and encode it into `SubscriptionInfoData` as key-value pairs. `SubscriptionInfoData` will be bumped to version 10

```
SubscriptionInfoData => Version LatestSupportedVersion ProcessId PrevTasks StandbyTasks UserEndPoint
TaskOffsetSums UniqueField ErrorCode ClientTags

   Version                => Int32
   LatestSupportedVersion  => Int32
   ProcessId              => UUID
   PrevTasks              => List<TaskId>
   StandbyTasks           => List<TaskId>
   UserEndPoint           => Bytes
   TaskOffsetSums         => List<TaskOffsetSum>
   UniqueField            => Int8
   ErrorCode              => Int32
   ClientTags             => List<ClientTag> // new change
```

Where is the struct with the following signature

```
ClientTag => Key Value
   Key     => Bytes
   Value   => Bytes
```

Kafka Streams's Task Assignor will make a decision on how to distribute standby tasks over the available clients based on encoded **`clientTags`** within the subscription info and configured **`rack.aware.assignment.tags`**

Standby task distribution algorithm is not specified in this KIP, but is left as an implementation detail. However, every distribution algorithm must handle gracefully when ideal standby task distribution is not possible; In that case, Kafka Streams must not fail the assignment but try to distribute the standby tasks on best-effort bases.

With an ideal task distribution, each client of the set of clients that host a given active task and the corresponding standby replicas has a unique value for each tag with regard to the other clients in the set.

Example of the ideal task distribution

Suppose we have the following infrastructure setup: Three Kubernetes Clusters, let us call them **K8s_Cluster1, K8s_Cluster2,** and **K8s_Cluster3.** Each Kubernetes cluster is spanned across three availability zones: **eu-central-1a**, **eu-central-1b**, **eu-central-1c**.

Our use-case is to have a distribution of the standby tasks across different Kubernetes clusters and AZs so we can be Kubernetes cluster and AZ failure tolerant.

With the new configuration options presented in this KIP, we will have the following:

```
Node-1:
client.tag.cluster: K8s_Cluster1
client.tag.zone: eu-central-1a
rack.aware.assignment.tags: zone,cluster
num.standby.replicas: 2

Node-2:
client.tag.cluster: K8s_Cluster1
client.tag.zone: eu-central-1b
rack.aware.assignment.tags: zone,cluster
num.standby.replicas: 2

Node-3:
client.tag.cluster: K8s_Cluster1
client.tag.zone: eu-central-1c
rack.aware.assignment.tags: zone,cluster
num.standby.replicas: 2

Node-4:
client.tag.cluster: K8s_Cluster2
client.tag.zone: eu-central-1a
rack.aware.assignment.tags: zone,cluster
num.standby.replicas: 2

Node-5:
client.tag.cluster: K8s_Cluster2
client.tag.zone: eu-central-1b
rack.aware.assignment.tags: zone,cluster
num.standby.replicas: 2

Node-6:
client.tag.cluster: K8s_Cluster2
client.tag.zone: eu-central-1c
rack.aware.assignment.tags: zone,cluster
num.standby.replicas: 2

Node-7:
client.tag.cluster: K8s_Cluster3
client.tag.zone: eu-central-1a
rack.aware.assignment.tags: zone,cluster
num.standby.replicas: 2

Node-8:
client.tag.cluster: K8s_Cluster3
client.tag.zone: eu-central-1b
rack.aware.assignment.tags: zone,cluster
num.standby.replicas: 2

Node-9:
client.tag.cluster: K8s_Cluster3
client.tag.zone: eu-central-1c
rack.aware.assignment.tags: zone,cluster
num.standby.replicas: 2
```

With the infrastructure topology and configuration presented above, we can easily achieve *The Ideal* standby task distribution. *The Ideal* standby task distribution is achievable because we have to allocate three tasks for any given stateful task (1 active task + 2 standby task), and it corresponds to unique values for each tag.

Assuming active stateful task 0_0 is in **Node-1**, *The Ideal* standby task distribution might look like this:

1. **Node-5** (different cluster, different zone), **Node-9** (different cluster, different zone)
2. **Node-6** (different cluster, different zone), **Node-8** (different cluster, different zone)

Algorithm will chose either 1st or 2nd option, but not both.

Compatibility, Deprecation, and Migration Plan

The changes proposed by this KIP shouldn't affect previously setup applications. Since we introduce new configuration options, existing ones shouldn't be affected by this change.

Changes in Task Assignment logic

Implementation of this KIP will not affect task assignor behaviour specified in the KIP-441. Proposal mentioned in this KIP will merely extend the behaviour of the distribution of standby tasks. Behaviour will be extended only if required configurations mentioned in this KIP specified by the Kafka Stream's user.

Rejected Alternatives

- The initial idea was to introduce two configurations in StreamsConfig, **`rack.id`**, which defines the rack of the Kafka Streams instance and **`standby.task.assignor`** - class that implements **`RackAwareStandbyTaskAssignor`** interface.

  The signature of RackAwareStandbyTaskAssignor was the following:

```
public interface RackAwareStandbyTaskAssignor {

    /**
     * Computes desired standby task distribution for a different {@link StreamsConfig#RACK_ID_CONFIG}s.
     * @param sourceTasks - Source {@link TaskId}s with a corresponding rack IDs that are eligible for
standby task creation.
     * @param clientRackIds - Client rack IDs that were received during assignment.
     * @return - Map of the rack IDs to set of {@link TaskId}s. The return value can be used by {@link
TaskAssignor}
     *           implementation to decide if the {@link TaskId} can be assigned to a client that is
located in a given rack.
     */
    Map<String, Set<TaskId>> computeStandbyTaskDistribution(final Map<TaskId, String> sourceTasks,
                                                            final Set<String> clientRackIds);
}
```

  By injecting custom implementation of RackAwareStandbyTaskAssignor interface, users could *hint* Kafka Streams where to allocate certain standby tasks when more complex processing logic was required — for example, parsing rack.id, which can be a combination of multiple identifiers (as seen in the previous examples where we have cluster and zone tags).

  The above mentioned idea was abandoned because it's easier and more user-friendly to let users control standby task allocation with just configuration options instead of forcing them to implement a custom interface.

  Defining multiple **`client.tag`** with combination of **`rack.aware.assignment.tags`** gives more flexibility, which, as already mentioned above, could have been only possible with pluggable custom logic Kafka Streams's user must provide.

  For instance, if we append multiple tags to form a single rack, it may not give desired distribution to the user if the infrastructure topology is more complex. Let us consider the following example with appending multiple tags to form the single rack.

```
Node-1:
rack.id: K8s_Cluster1-eu-central-1a
num.standby.replicas: 1

Node-2:
rack.id: K8s_Cluster1-eu-central-1b
num.standby.replicas: 1

Node-3:
rack.id: K8s_Cluster1-eu-central-1c
num.standby.replicas: 1

Node-4:
rack.id: K8s_Cluster2-eu-central-1a
num.standby.replicas: 1

Node-5:
rack.id: K8s_Cluster2-eu-central-1b
num.standby.replicas: 1

Node-6:
rack.id: K8s_Cluster2-eu-central-1c
num.standby.replicas: 1
```

In the example mentioned above, we have three AZs and two Kubernetes clusters. Our use-case is to distribute standby task in the different Kubernetes cluster and different availability zone. For instance, if the active task is in Node-1 (K8s_Cluster1-eu-central-1a), the corresponding standby task should be in either on Node-5 (K8s_Cluster2-eu-central-1b) or on Node-6 (K8s_Cluster2-eu-central-1c).

Unfortunately, without custom logic provided by the user, this would be very hard to achieve with a single **`rack.id`** configuration. Because without any input from the user, Kafka Streams might as well allocate standby task for the active task either:

- In the same Kubernetes cluster and different AZ (Node-2, Node-3)
- In the different Kubernetes cluster but the same AZ (Node-4)

On the other hand, with the combination of the new "client.tag.*" and "rack.aware.assignment.tags" configurations, standby task distribution algorithm will be able to figure out what will be the most optimal distribution by balancing the standby tasks over each client.tag dimension individually. And it can be achieved by simply providing necessary configurations to Kafka Streams.

- The second approach was to refactor **`TaskAssignor`** interface to be more user-friendly and expose it as a public interface. Users then could implement custom **`TaskAssignor`** logic and set it via **`StreamsConfig`**. With this, Kafka Streams users would effectively be in control of Active and Standby task allocation.
  Similarly to the point above, this approach also was rejected because it's more complex.
  Even though it's more-or-less agreed on the pluggable TaskAssignor interface's usefulness, it was decided to cut it out of this KIP's scope and prepare a separate one for that feature.