KIP-879: Multi-level Rack Awareness

This page is meant as a template for writing a KIP. To create a KIP choose Tools->Copy on this page and modify with your content and replace the heading with the next KIP number and a description of your issue. Replace anything in italics with your own description.

Status

Current state: *Under Discussion*

Discussion thread: *here*

JIRA: *KAFKA-14281*

Released: <Kafka Version>

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

Motivation

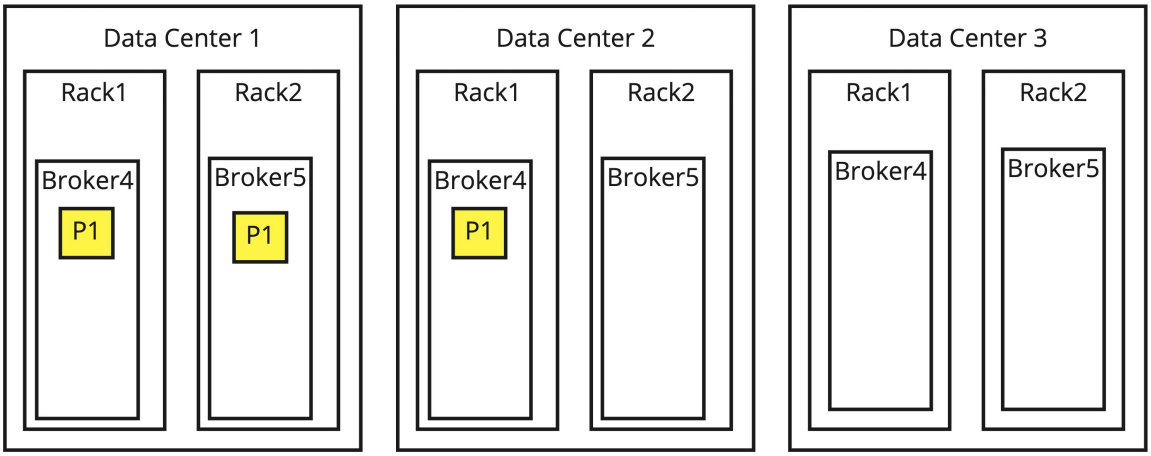
With replication services data can be replicated across independent Kafka clusters in multiple data centers. In addition, many customers need "stretch clusters" - a single Kafka cluster that spans across multiple data centers. This architecture has the following useful characteristics:

- Data is natively replicated into all data centers by Kafka topic replication.
- No data is lost when 1 DC is lost and no configuration change is required - design is implicitly relying on native Kafka replication.
- From an operational point of view, it is much easier to configure and operate such a topology than a replication scenario via MM2.
- Currently Kafka has a single partition assignment strategy and if users want to override that, they can only do it via manually assigning brokers to replicas on topic creation.

Multi-level rack awareness

Additionally, stretch clusters are implemented using the rack awareness feature, where each DC is represented as a rack. This ensures that replicas are spread across DCs evenly. Unfortunately, there are cases where this is too limiting - in case there are actual racks inside the DCs, we cannot specify those. Consider having 3 DCs with 2 racks each.

If we were to use racks as /DC1/R1, /DC1/R2, etc, then when using RF=3, it is possible that 2 replicas end up in the same DC, e.g. /DC1/R1, /DC1/R2, /DC2/R1 and DC3 will be unused.



We can see that in this case we could do a more optimal, more distributed solution that is now provided by Kafka if we were to distinguish levels in rack assignment.

Therefore Kafka should support "multi-level" racks, which means that rack IDs should be able to describe some kind of a hierarchy. To make the solution more robust, we would also make the current replica assignment configurable where users could implement their own logic and we would provide the current assignment as default and a new multi-level rack aware described above as it is becoming more and more popular to use stretch-clusters.

With this feature, brokers should be able to:

1. spread replicas evenly based on the top level of the hierarchy (i.e. first, between DCs)
2. then inside a top-level unit (DC), if there are multiple replicas, they should be spread evenly among lower-level units (i.e. between racks, then between physical hosts, and so on)
    a. repeat for all levels

In a very simple case the result would look like this:

Public Interfaces

We propose a number of changes to various interfaces of Kafka. The core change is to transform the rack assignment into a configurable algorithm. This would involve a number of public interface changes:

- Introduce new config to control the underlying implementation (much like the Authorizer interface).
- Create a Java interface in the clients module that sits behind the configuration.
- Refactor the current rack assignment strategy under the new interface
- Add a new implementation, called the multi-level rack assignment strategy, discussed above
- Add an Admin API so clients like the reassign partitions command would rely on the brokers instead of calling an internal utility to assign replicas.
- Add a new protocol to transfer requests and responses of the new Admin API.
- A multi-level rack aware replica selector that is responsible for selecting a replica to fetch from when consumers are using follower fetching. This makes it easier for replicas to fetch from the nearest follower in a multi-level rack environment.

These will be detailed under proposed changes.

Proposed Changes

Pluggable Partition Assignment Strategy

We will introduce a new broker config called `broker.replica.placer` that would make it possible for users to specify their custom implementation of replica placers. Its default value will be the current rack aware replica assignment strategy, we will just make that compatible with this new interface that we'll detail below. We will also allow any plugins to fall back to the original algorithm if needed or return an error if they can't come up with a suitable solution.

Interfaces

Equivalents of these proposed interfaces currently exist in the metadata component under `org.apache.kafka.placement`. We'd like to improve these interfaces slightly with some naming improvements and added/changed fields to better fit a general algorithm. We would also like to provide a new, multi-level rack aware implementation discussed in this KIP. The current algorithm would stay as it is.

BrokerReplicaAssignor

```
/**
 * This interface provides an API for implementing replica assignment strategies. Both rack aware and unaware
 * strategies are supported by this interface. If needed it can defer assignment to Kafka's default rack unaware
 * replicas assignment strategy.
 */
public interface ReplicaPlacer {

    /**
     * Creates an assignment of a topic given the racks, number of partitions and the replication factor.
     * @param clusterView is the necessary information about the cluster (such as current assignment, rack
information)
     *                    that is needed to be able to decide replica assignments.
     * @param placement the assignment parameters, such as the replication factor and number of partitions.
     * @param brokerIds is an optional list of brokerIDs that is used to generate an assignment. If null is
specified,
     *                  then all brokers are used.
     * @return
     * <ul>
     *     <li><code>null</code> to defer the assignment to Kafka, that uses its rack unaware assignment
strategy.</li>
     *     <li>the assignment in the form of zero based list of lists where mapping where the outer list
specifies the
     *         partition index and the inner list specifies the assigned broker ids.</li>
     * </ul>
     */
    List<List<Integer>> place(PlacementSpec placement, ClusterView clusterView, List<Integer> brokerIds);
}
```

The ClusterView class would be the improved version of ClusterDescriber. We would like to add the current list of partition assignments.

ClusterView

```
/**
 * This class defines a view of a cluster that the partition placer can see, such as the current assignment
 * or broker -> rack mappings.
 */
public class ClusterView {

    private final Map<TopicPartition, List<Integer>> partitionAssignments;
    private final Iterator<UsableBroker> usableBrokers;

    public ClusterView(Map<TopicPartition, List<Integer>> partitionAssignments, List<BrokerInfo> brokerInfos) {
        this.partitionAssignments = partitionAssignments;
        this.brokerInfos = brokerInfos;
    }

    public Map<TopicPartition, List<Integer>> topicAssignments() {
        return Collections.unmodifiableMap(partitionAssignments);
    }

    public List<BrokerInfo> brokerInfos() {
        return Collections.unmodifiableList(brokerInfos);
    }
}
```

Admin API

We would like to add a new Admin API method that enables commands like the reassignment command to generate an assignment that is suggested by the broker based on what's configured there. This would eliminate the need to pass down the corresponding assignor configuration to the client which may or may not be aware of what's configured on the brokers and also allows the active controller to be the single source to assign replicas.

```
    /**
     * Create a partitions assignment according to the partition assignment strategy set in the cluster. It
will use the
     * brokers and racks available to the cluster to create an assignment given the number of partitions and the
     * replication factor.
     * @param numPartitions is the number of partitions of the topic to act on
     * @param replicationFactor is the replication factor of the topic in action
     * @return a result object containing the assignment.
     */
    CreateReplicaPlacementResult createReplicaPlacement(int numPartitions, short replicationFactor);

    /**
     * Create a partitions assignment according to the partition placement strategy set in the cluster. It will
use the
     * brokers and racks available to the cluster to create an assignment given the number of partitions and the
     * replication factor. In an extra options parameter we can define a starting partition and replication
factor.
     * @param numPartitions is the number of partitions of the topic to act on
     * @param replicationFactor is the replication factor of the topic in action
     * @param options is the parameter where extra options can be set to further control replica placement.
     * @return a result object containing the partition -> brokerId[] mapping.
     */
    CreateReplicaPlacementResult createReplicaPlacement(int numPartitions, short replicationFactor,
                                                        CreateReplicaPlacementOptions options);
```

CreateReplicaPlacementResult

```
public class CreateReplicaPlacementResult {

    private KafkaFuture<List<List<Integer>>> future;

    CreateReplicaPlacementResult(KafkaFuture<List<List<Integer>>> future) {
        this.future = future;
    }

    public KafkaFuture<List<List<Integer>>> placements() {
        return future;
    }
}
```

CreateReplicaPlacementOptions

```
public class CreateReplicaPlacementOptions extends AbstractOptions<CreateReplicaPlacementOptions> {

    private boolean forceSimplePlacement;
    private List<Integer> brokerIds;

    public boolean forceSimplePlacement() {
        return forceSimplePlacement;
    }

    public CreateReplicaPlacementOptions forceSimplePlacement(boolean forceSimplePlacement) {
        this.forceSimplePlacement = forceSimplePlacement;
        return this;
    }

    public List<Integer> brokerIds() {
        return Collections.unmodifiableList(brokerIds);
    }

    public CreateReplicaPlacementOptions brokerIds(List<Integer> brokerIds) {
        this.brokerIds = brokerIds;
        return this;
    }
}
```

By specifying the `forceSimplePlacement` flag we instruct the brokers to use the default rack unaware algorithm of Kafka. This is needed to maintain compatibility with the `–disable-rack-awareness` flag in `kafka-partition-reassignment.sh`. This option will effectively be triggered by specifying that flag and will behave the same way, it will instruct the broker to skip the specified partition assignor and use the default rack unaware algorithm. Note that a rack aware assignor might support a scenario where just part of the brokers have assigned racks.

Some algorithms might work with only a specific subset of brokers as an input to the assignment. In the Admin API we could specify this with the `brokerIds` list.

Protocol

For the Admin API described above, we'll need a new protocol as well. This is defined as follows.

CreatePartitionPlacementRequest

```
{
  "apiKey": 68,
  "type": "request",
  "listeners": ["zkBroker", "broker", "controller"],
  "name": "CreateReplicaPlacementRequest",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
    { "name": "NumPartitions", "type": "int32", "versions": "0+",
      "about": "The number of partitions of the topic in the assignment." },
    { "name": "ReplicationFactor", "type": "int16", "versions": "0+",
      "about": "The number of replicas each of the topic's partitions in the assignment." },
    { "name": "ForceSimplePlacement", "type": "bool", "versions": "0+", "default": "false",
      "about": "Forces the default rack unaware partition assignment algorithm of Kafka." },
    { "name": "NodeId", "type": "[]int32", "versions": "0+", "entityType": "brokerId",
      "about": "The broker ID." },
    { "name": "timeoutMs", "type": "int32", "versions": "0+", "default": "60000",
      "about": "How long to wait in milliseconds before timing out the request." }
  ]
}
```

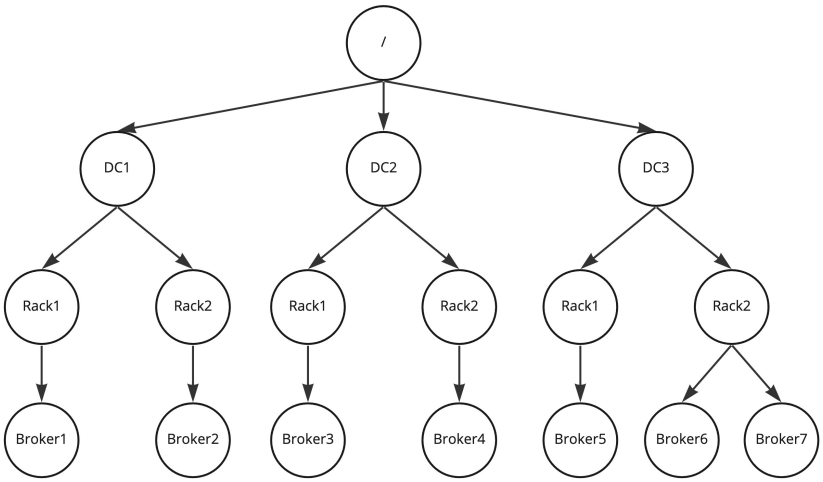
CreatePartitionPlacementResponse

```
{
  "apiKey": 68,
  "type": "response",
  "name": "CreateReplicaPlacementResponse",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
    { "name": "ReplicaPlacement", "type": "[]ReplicaPlacement", "versions": "0+",
      "about": "A partition to list of broker ID mapping.",
      "fields": [
        { "name": "PartitionId", "type": "int32", "versions": "0+", "mapKey": true },
        { "name": "Replicas", "type": "[]int32", "versions": "0+" }
      ]
    }
  ]
}
```

Multi-level rack assignment

In this section we'll detail the behavior of the multi-level replica assignment algorithm. Currently the replica assignment happens on the controller which continues to be the case. The controller has the cluster metadata, therefore we know the full state of the cluster.

The very first step is that we build a tree of the rack assignments as shown below.



The assignor algorithm can be formalized as follows:

1. Every rack definition must start with "/". This is the root node.
2. In every node we define a ring that'll iterate through the children (and it'll continue at the beginning if it reaches the last children). It by default points to a random leaf to ensure the even distribution of replicas of different partitions on the cluster.
3. On adding a new replica on every level we select the child that is pointed by the iterator, pass down the replica to that and increment the pointer.
4. Repeat the previous step for the child we passed down this replica until the leaf is reached.

To increase the replication factor, we basically pick up where the algorithm left the last partition and continue with the above algorithm from the next node that'd be given by the algorithm.

Compatibility, Deprecation, and Migration Plan

The change is backward compatible with previous versions as it is an extra addition to Kafka. The current strategy will remain the same but there will be a new one that users can optionally configure if they want to.

Test Plan

No extra tests needed other than the usual: extensive unit testing to make sure that the code works as expected.

Alternatives

Restricted Replica Assignment

I call this restricted because in the current design we don't only call the API when all of the brokers are rack aware but in any case, since someone can implement a rack assignment that places replicas not based on racks but for instance hardware utilization. This would make sense as for instance we may use Cruise Control that can periodically reassign partitions based on the load but we may make its job easier by placing replicas at creation to the right node (one less reassignment needed).

Now at this point all this above might not be needed and we could restrict the scope only to scenarios where all (or at least one) brokers have racks assigned as we simplify the broker side behavior as passing havin no racks would always result in calling the current rack unaware assignment strategy.

Using Cruise Control

Although Cruise Control is mainly used for broker load based partition reassignment, we can easily extend that with another Goal that reorganizes replicas based on the described multi-level algorithm in this KIP. However this still has an initial headwind as replicas are not immediately placed on the right brokers and users would need to force out a rebalance right after topic creation. Secondly, not all Kafka users use Cruise Control, they may use other software for load balancing or none at all.