KIP-36 Rack aware replica assignment

Status

Current state: *Accepted*

Discussion thread: *here*

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

Github Pull Request: https://github.com/apache/kafka/pull/132

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

Motivation

Machines in data center are sometimes grouped in racks. Racks provide isolation as each rack may be in a different physical location and has its own power source. When resources are properly replicated across racks, it provides fault tolerance in that if a rack goes down, the remaining racks can continue to serve traffic.

In Kafka, if there are more than one replica for a partition, it would be nice to have replicas placed in as many different racks as possible so that the partition can continue to function if a rack goes down. In addition, it makes maintenance of Kafka cluster easier as you can take down the whole rack at a time.

In AWS, racks are usually mapped to the concept of availability zones.

Public Interfaces

Changes to Broker property

An optional broker property will be added

```
case class Broker(id: Int, endPoints: Map[SecurityProtocol, EndPoint], rack: Option[String])
```

rack will be an optional field of version 3 JSON schema for a broker. The API ZkUtils.registerBrokerInZk will be updated to increment the JSON version.

For example:

```
{"version":3,
  "host","localhost",
  "port",9092
  "jmx_port":9999,
  "timestamp":"2233345666",
  "endpoints": ["PLAINTEXT://host1:9092",
                "SSL://host1:9093"],
  "rack": "dc1"
}
```

If no rack information is specified, the field will not be included in JSON.

Rack can be specified in the broker property file as

```
broker.rack=<rack ID as string>
```

For example,

`broker.rack=dc1`

`broker.rack=us-east-1c`

Consequently, Broker.writeTo will append rack at the end of ByteBuffer and Broker.readFrom will read it:

```
Broker => ID NumberOfEndPoints [EndPoint] Rack
ID => int32
NumberOfEndPoints => int32
EndPoint => Port Host ProtocolID
Port => int32
Host => string
ProtocolID => int16
Rack => string
```

Same kind of string serialization (as how host is serialized) will be applied to rack, which means it will first write the size of the string as a short, followed by the actual string content. If rack is not available, it will write size -1 only without any actual string content. When reading from ByteBuffer, -1 will be interpreted as "null".

Changes to UpdateMetadataRequest

The version will be incremented to 2 from 1 and rack will be included. For version 0, only broker ID, host and port will be serialized. For version 1 and 2, the complete Broker object will be serialized. This is done by calling Broker.writeTo and Broker.readFrom. Therefore, for version 2 the rack information will be automatically handled and the the serialization format is the same as the above.

The complete new format of UpdateMetadataRequest is the following:

```
VersionID CorrelationID ClientID Controller_ID Controller_Epoch Partition_States Brokers
Brokers => NumberOfBroker [Broker]
NumberOfBroker => Int32
Broker => ID NumberOfEndPoints [EndPoint] Rack // Rack is added in version 2
ID => int32
NumberOfEndPoints => int32
EndPoint => Port Host ProtocolID
Port => int32
Host => string
ProtocolID => int16
Rack => string
```

Changes to TopicMetadataRequest and TopicMetadataResponse

TopicMetadataRequest will increment its version from 0 to 1. If Kafka receives TopicMetadataRequest with version 1, it will create TopicMetadataResponse with rack as part of BrokerEndPoint. Here is the new format for TopicMetadataResponse:

```
CorrelationID BrokerEndPoints TopicsMetaData
BrokerEndPoints => NumberOfBrokers [BrokerEndPoint]
BrokerEndPoint => BrokerID Host Port Rack // Rack is added
BrokerID => int32
Host => string
Port => int32
Rack => string
```

Proposed Changes

- *`AdminUtils.assignReplicasToBrokers` will be updated to create broker-rack mapping from ZooKeeper data before doing replica assignment.*
- *When making the rack aware assignment, it has the following properties:*
  - *When the number of partition is N (where N is a positive integer) times number of brokers*
    - *if each rack has the same broker count, each broker will have the same leader count and replica count.*
    - *if each rack has different broker count, each broker will have the same leader count, but may have different replica count*
  - Assign to as many racks as possible. That means if the number of racks are more than or equal to the number of replicas, each rack will have at most one replica. On the other hand, if the number of racks is less than the the number of replicas (which should happen very infrequently), each rack should have at least one replica and no other guarantees are made on how the replicas will be distributed among racks. For example, if there are 2 racks and 4 replicas, one rack can have 3 replicas, 2 replicas or 1 replica. This is to keep the algorithm simple while still keeping other replica distribution properties and fault tolerance from the racks.
- Implementation detail of the rack aware assignment (copied from pull request https://github.com/apache/kafka/pull/132 with slight modification):

```
To create rack aware assignment, this API will first create an rack alternated broker list. For example,
from this brokerID -> rack mapping:

0 -> "rack1", 1 -> "rack3", 2 -> "rack3", 3 -> "rack2", 4 -> "rack2", 5 -> "rack1"

The rack alternated list will be

0 (rack1), 3 (rack2), 1 (rack3), 5 (rack1), 4 (rack2), 2 (rack3)

The leader of the partitions are chosen from rack alternated list round-robin. The follower
starts from the broker next to the leader. For example, assume we are going to create
assignment of for 6 partitions with replication factor 3. Here are the assignment created:

0 -> (0,3,1)
1 -> (3,1,5)
2 -> (1,5,4)
3 -> (5,4,2)
4 -> (4,2,0)
5 -> (2,0,3)

Once it has completed the first round-robin, if there are more partitions to assign, the algorithm will
start to
have a shift for the first follower. The shift is N times number of racks, which N increasing for each
round.
This is to ensure we will not always get the same set of sequence. In this case, for partitions number 6
to 11,
the assignment will be

6 -> (0,4,2) (instead of repeating 0,3,1 as partition 0, as shift of 3 is added to the first follower)
7 -> (3,2,0)
8 -> (1,0,3)
9 -> (5,3,1)
10-> (4,1,5)
11-> (2,5,4)

When there are uneven number of brokers in racks, the algorithm may have to skip a broker if the
its rack is already used for the partition.  For example, assume the following broker -> rack mapping:

0 -> rack1, 1 -> rack2, 2 -> rack2

The rack alternated list is (0,1,2)

To assign 3 partitions with replication factor of 2, the first partition will be assigned as 0 -> (0,1).
The second partition will be 1 -> (1,2) if we follow the first example. However, since broker 2 is on
rack2 and rack2 is already used (for broker 1), we have to skip broker 2 and go to broker 0 instead.
Therefore, the final assignment for partition 1 is 1 -> (1,0). The complete assignments for
three partitions are

0 -> (0,1)
1 -> (1,0)
2 -> (2,0)

As you can see, in this example, the leaders for 3 partitions are evenly distributed, but
replica distribution is not even. Broker 0 has three replicas, broker 1 has two, and
broker 2 only has one. Broker 0 is "penalized" to have the most replicas because it is the
only broker available in rack1.

However, we do not always have to "skip" a broker because its rack is already used. If every
rack already has one replica of a given partition, skipping no longer helps. This happens if
number of replicas is greater than the number of racks for a given partition.
As the result, if the number of replicas is equal to or greater than the number of racks,
it will ensure that each rack will get at least one replica. Otherwise, each rack will get
at most one replica. In the perfect situation where number of replica is the same as number of
racks and each rack has the same number of brokers, it guarantees that both leader and replica
distribution
is even across brokers and racks.
```

- *If one or more brokers does NOT have rack information*
    - *For auto topic creation, `AdminUtils.assignReplicasToBrokers` will create the same assignment as the current implementation (as if no broker has the rack information) and continue with topic creation. This allows auto topic creation to work when doing rolling upgrade.*
    - *For command line tools (TopicCommand and ReassignPartitionsCommand), an exception will be thrown. This will alert the user that a broker may be misconfigured. An additional command line argument --ignore-racks can be supplied to suppress such error and continue with topic creation ignoring all rack information.*

Compatibility, Deprecation, and Migration Plan

- Rack will be included in inter-broker protocol. Therefore, rolling upgrade requires these steps:
    1. Update server.properties file on all brokers and add the following property: inter.broker.protocol.version=0.9.0.X
    2. Upgrade the brokers. This can be done a broker at a time by simply bringing it down, updating the code, and restarting it.
    3. Once the entire cluster is upgraded, bump the protocol version by editing inter.broker.protocol.version and setting it to new version, for example, 0.10.0.
    4. Restart the brokers one by one for the new protocol version to take effect
- Rack property will be included in version 3 broker JSON schema and version 2 of UpdateMetadataRequest. Controller will include version specific broker information in its wire format so that broker with old version can still interoperate with broker with new version and rack.
- Due to a bug introduced in 0.9.0.0 in ZkUtils.getBrokerInfo(), old clients will throw an exception when it sees the broker JSON version is not 1 or 2. Therefore, **a minor release 0.9.0.1 is required** to fix the problem first so that old clients can parse future version of broker JSON in ZooKeeper. That means 0.9.0.0 clients must be upgraded to 0.9.0.1 before 0.10.0 upgrade can start. In addition, since ZkUtils.getBrokerInfo() is also used by broker, version specific code has to be used when registering broker with ZooKeeper. This is outlined as follows:

    - When inter.broker.protocol.version is set to 0.9.0.X, the broker will write the broker info in ZK using version 2, ignoring the rack info.
    - When inter.broker.protocol.version is set to 0.10.0, the broker will write the broker info in ZK using version 3, including the rack info.
    - If one follows the upgrade process, after the 2nd round of rolling bounces, every broker is capable of parsing version 3 of broker info in ZK. This is when the rack-aware feature will be used.