KIP-509: Rebalance and restart Producers

- Status
- Motivation
- Public Interfaces
- Proposed Changes
- Compatibility, Deprecation, and Migration Plan
- Rejected Alternatives

Status

Current state: Under Discussion

Discussion thread: here

JIRA: here

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

Motivation

At the moment producers on multiple servers either need to be coordinating which source partition is sent by whom or Kafka Connect is used as foundation. Then the servers need to be part of a cluster. Neither is convenient in an auto-load-balance Kubernetes environment.

For consumers, the Kafka server takes care of that. For KStreams as well. Just for producer it does not.

I would suggest that a producer instance has the option to specify how many partitions the source has. Example: The source database table has 10 partitions named p1 to p10, hence the producer specifies partitioncount=10 at connect.

The Kafka Server then responds with a list of partition numbers this instance should handle. If this instance is the first of the producer group, it gets a list with all 10. If it is the second producer of the same group, the first producer gets a rebalance signal to use just partitions 1-5 and the new producer is informed to handle partitions 6-10. So in essence this API feels pretty much like a consumer group. It is the responsibility of the producer to map the individual partition numbers to the information what data to actually produce.

Whenever a rebalance does happen or a new instance is started or the producer is restarted after an error, the producer needs to know where to pickup reading the source. This as well is currently required to be dealt with externally. As we support commits now, Kafka should provide convenience functions for that as well.

Public Interfaces

New properties for the producer config:

```
props.put(ProducerConfig.GROUP_ID_CONFIG, "ProducerForDB1");
props.put(ProducerConfig.SOURCE_PARTITION_COUNT, 10);
props.put(ProducerConfig.SESSION_TIMEOUT_MS_CONFIG, 30000);
producer = new KafkaProducer<byte[], byte[]>(props);
```

Producer can register a rebalancelistener. The call returns the list of partition numbers this instance shall be responsible for and a String to identify which source record to start reading from, e.g. a database transaction number.

```
Map<Integer, String> partitionmap = producer.subscribe(rebalancelistener);
```

The producer starts producing data and when committing the data, tells what the source identifier had been, e.g. the database transaction number.

```
producer.commitTransaction(transactionid);
```

Occasionally the producer has to send an alive-packet in cases where no source data is changed for the TIMEOUT_MS_CONFIG time.

```
producer.alive();
```

As a result, the cluster-y thing of how many instances are active at the moment, which instance is responsible for what data so data is not produced twice and finding the restart point is all handled by the Kafka server. The producer is only responsible to implement a formula, how the partition number is translated to an actual partition name in the source.

Proposed Changes

To make things more clear, let's go through a concrete example. Producer is a JDBC producer reading from three tables loading three topics. All three tables have a timestamp column with the last_change_ts.

Case 1: A first producer instance is started for the first time.

```
props.put(ProducerConfig.GROUP_ID_CONFIG, "ProducerForDB1");
props.put(ProducerConfig.SOURCE_PARTITION_COUNT, 3);
props.put(ProducerConfig.SESSION_TIMEOUT_MS_CONFIG, 30000);
producer = new KafkaProducer<byte[], byte[]>(props);
Map<Integer, String> partitionmap = producer.subscribe(rebalancelistener);
```

The partitionmap will look like (0, null), (1,null), (2,null) because it is the only producer instance and no record was committed before ever.

The mapping from partitions to the source database is defined in the code as 0...TableA, 1...TableB, 2...TableC. Therefore the producer will start reading all three tables from the beginning and commit the data at the end with a transactionid which is the highest last_change_ts.

```
producer.commitTransaction(max_last_change_ts);
```

Case 2: Above producer died and got restarted. Same procedure will happen except that the partitionmap will return tuples where the string value reflects the last value provided by the commitTransaction().

Case 3: A second producer with same group name is started.

The producer #1's rebalance listener will be triggered, telling that it is responsible for partitionmap = (0, 12:34:00), (1, 12:34:00). Hence it will care about TableA and TableB only.

The producer #2, when subscribing to the same three topics will get partitionmap = (2, 12:34:00) and therefore know to process TableC only, starting with the provided timestamp.

A third producer can be started as well but its partitionmap will be empty. It remains in standby until a rebalance happens.

Case 4: Producer #1 failed to produce data and did not send an alive-packet in time.

A rebalance does happen, assigning new partitions to producer #2 and #3.

Note: Above timestamp based delta has a couple of loop holes like uncommitted data. But that is the essence of a timestamp based delta. Just wanted to make sure neither expects above to be a timestamp based delta implementation or that we talk about timestamps instead of the feature. I used the timestamp as example because it is easy to relate to.

Compatibility, Deprecation, and Migration Plan

- As it is a pure addition, no impact on existing users. It offers help by additional properties and methods but does not impose itself on the developer.
- Kafka Connect would benefit from this, though.

Rejected Alternatives