# KIP-837: Allow MultiCasting a Result Record.



## Status

**Current state**: *"Adopted"*

**Discussion thread**:

**JIRA**: KAFKA-13602

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

## Motivation

Lot of times, in Kafka Streams users want to send a record to more than one partition on the sink topic. Currently, if a user wants to replicate a message into N partitions, the only way of doing that is to replicate the message N times and then plug-in a new custom partitioner to write the message N times into N different partitions. To give a more concrete example, let's say there's a sink topic with 10 partitions and a user wants to send records only to even numbered partitions.

In today's world, this is how the user can do the same:

```
final int numPartitions = 10;

final KStream<String, String> inputStream = builder.stream("input-topic", Consumed.with(Serdes.String(), Serdes.
String()));

for (int i = 0; i < numPartitions; i += 2) {
        final int evenNumberedPartition = i;
    inputStream.to("output-topic", Produced.with(Serdes.String(), Serdes.String(), (topic, key, val,
numPartitions) -> evenNumberedPartition));
}
```

As can be seen, there's no implicit way of sending the records to multiple partitions. This KIP aims to make this process simpler in Kafka Streams. As a side note, this KIP also adds a provision to drop records using a custom partitioner.

## Public Interfaces

The StreamPartitioner method would have a new method added called `partitions()` and the current `partition()` method would be marked as deprecated. The `partitions()` method would be marked as default within which it would invoke `partition()` method and construct a singleton set out of it. Here's how the interface would look like now:

```
import java.util.Set;
import java.util.Collections;

  public interface StreamPartitioner<K, V> {

    /**
     * Determine the partition number for a record with the given key and value and the current number of
partitions.
     *
     * @param topic the topic name this record is sent to
     * @param key the key of the record
     * @param value the value of the record
     * @param numPartitions the total number of partitions
     * @return an integer between 0 and {@code numPartitions-1}, or {@code null} if the default partitioning
logic should be used
     */
    @Deprecated
    Integer partition(String topic, K key, V value, int numPartitions);


       /**
     * Determine the partition numbers to which a record, with the given key and value and the current number
     * of partitions, should be multi-casted to.
     * @param topic the topic name this record is sent to
     * @param key the key of the record
     * @param value the value of the record
     * @param numPartitions the total number of partitions
     * @return an Optional of Set of integers between 0 and {@code numPartitions-1},
     * Empty optional means use default partitioner
     * Optional of an empty set means the record won't be sent to any partitions i.e dropped.
     * Optional of Set of integers means the partitions to which the record should be sent to.
     * */
    default Optional<Set<Integer>> partitions(String topic, K key, V value, int numPartitions) {
        Integer partition = partition(topic, key, value, numPartitions);
        return partition == null ? Optional.empty() : Optional.of(Collections.singleton(partition(topic, key,
value, numPartitions)));
    }

}
```

The return type is a Set so that for cases of a faulty implementation of `partitions` method which might return same partition number multiple times, we would still record it only once.

# Proposed Changes

- RecordCollector and it's implementation RecordCollectorImpl has the method which accepts a StreamPartitioner object and pushes it to the partitions returned by the partition method. This is the core piece of logic which would allow multi/broadcasting records. Here are the high level changes. Note that when the `partitions()` method return an empty set meaning we won't send the record to any partition, we would also be updating the droppedRecordsSensor.

```
@Override
    public <K, V> void send(final String topic,
                            final K key,
                            final V value,
                            final Headers headers,
                            final Long timestamp,
                            final Serializer<K> keySerializer,
                            final Serializer<V> valueSerializer,
                            final StreamPartitioner<? super K, ? super V> partitioner) {

        if (partitioner != null) {

            // Existing logic to fetch partitions

                        // Changes to be made due to the KIP
                        if (partitions.size() > 0) {
                final Optional<Set<Integer>> maybeMulticastPartitions = partitioner.partitions(topic,
key, value, partitions.size());
                if (!maybeMulticastPartitions.isPresent()) {
                    // New change. Use default partitioner
                    send(topic, key, value, headers, null, timestamp, keySerializer, valueSerializer,
processorNodeId, context);
                } else {
                    Set<Integer> multicastPartitions = maybeMulticastPartitions.get();
                    if (multicastPartitions.isEmpty()) {
                        // If a record is not to be sent to any partition, mark it as dropped.
                        log.info("Not sending the record to any partition");
                        droppedRecordsSensor.record();
                    } else {
                        for (final int multicastPartition: multicastPartitions) {
                            send(topic, key, value, headers, multicastPartition, timestamp,
keySerializer, valueSerializer, processorNodeId, context);
                        }
                    }
                }
            } else {
                throw new StreamsException("Could not get partition information for topic " + topic + "
for task " + taskId +
                        ". This can happen if the topic does not exist.");
            }
                  } else {
            send(topic, key, value, headers, null, timestamp, keySerializer, valueSerializer,
processorNodeId, context);
        }
        }
```

- StreamPartitioner is also used in FK-join and Interactive queries. For FK-join and IQ, the invocation of `partition()` would be replaced by `partitions()` and a runtime check would be added to ensure that the returned set is singleton.

### Example Usage

Continuing the example from the motivation section, with the new interface, here's how users can send to even number partitions:

```
// Define a partitioner class which sends to even numbered partitions

public static class EvenPartitioner<K, V> implements StreamPartitioner<K, V> {

        @Override
        public Integer partition(String topic, K key, V value, int numPartitions) {
            return null;
        }

        @Override
        public Set<Integer> partitions(String topic, K key, V value, int numPartitions) {
            final Set<Integer> partitions = new HashSet<>();
            for (int i = 0; i < numPartitions; i += 2) {
                partitions.add(i);
            }
            return partitions;
        }
    }

// Broadcasting
public static class BroadcastingPartitioner<K, V> implements StreamPartitioner<K, V> {

        @Override
        public Integer partition(String topic, K key, V value, int numPartitions) {
            return null;
        }

        @Override
        public Set<Integer> partitions(String topic, K key, V value, int numPartitions) {
                                return IntStream.range(0, numPartitions).boxed().collect(Collectors.toSet());
         }
    }

// Don't send to any partitions
public static class DroppingPartitioner<K, V> implements StreamPartitioner<K, V> {

        @Override
        public Integer partition(String topic, K key, V value, int numPartitions) {
            return null;
        }

        @Override
        public Set<Integer> partitions(String topic, K key, V value, int numPartitions) {
            return Collections.emptySet();
        }
    }


// Build Stream.
final KStream<String, String> inputStream = builder.stream("input-topic", Consumed.with(Serdes.String(), Serdes.
String()));

// Send to even numbered partitions
inputStream.to("output-topic", Produced.with(Serdes.String(), Serdes.String(), (topic, key, val, numPartitions)
-> new EvenPartitioner()));
// Broadcast
inputStream.to("output-topic", Produced.with(Serdes.String(), Serdes.String(), (topic, key, val, numPartitions)
-> new BroadcastingPartitioner()));
// Send to even numbered partitions
inputStream.to("output-topic", Produced.with(Serdes.String(), Serdes.String(), (topic, key, val, numPartitions)
-> new DroppingPartitioner()));
}
```

This way users just need to define the partitioner and the internal routing mechanism would take care of sending the records to relevant or no partitions.

# Compatibility, Deprecation, and Migration Plan

This KIP deprecates `partition()` method in `StreamPartitioner` . Here is what we would be doing as part of the same:

- StreamPartitioner is also used in FK-join and IQ. The invocation of `partition()` would be replaced by `partitions()` and a runtime check would be added to ensure that the returned set is singleton.
- Adding sufficient logging

# Rejected Alternatives

The first 3 alternatives were rejected as they focussed only broadcasting to all partitions which seemed restrictive.

- extend to() / addSink() with a "broadcast" option/config in KafkaStreams.
- add toAllPartitions() / addBroadcastSink() methods in KafkaStreams.
- allow StreamPartitioner to return `-1` for "all partitions".
- Adding a separate class to handle multi casting. This was rejected in favour of enhancing the current `StreamPartitioner` interface.

# Test Plan

- Add new tests(unit/integration) similar to the `partition` tests for the newly introduced `partitions` method.
- Add tests to fail when a non singleton set is returned for FK joins and IQ when querying.