By default, the producer does not care what partition a specific message is written to and **will balance messages over all partitions of a topic evenly**.

In some cases, the producer will direct messages to specific partitions. This is typically done using the message key and a partitioner that will generate a hash of the key and map it to a specific partition. This assures that all messages produced with a given key will get written to the same partition.

There are three primary methods of sending messages:

* Fire-and-forget - in which producer send a message to the server and don’t really care if it arrived succesfully or not. Most of the time, it will arrive successfully, since Kafka is highly available and the producer will retry sending messages automatically. However, some messages will get lost using this method. **Good for use case like “Collecting twitter feed of celebrity”, “Counting hits of Video”. Here losing 2 to 3% of data is ok.**
* Synchronous Send - we send a message, the send() method returns a Future object and we use get() to wait on the future and see if the send() was successful or not.
* Asynchronous Send - we call the send() method with a callback function, which gets triggered when receive a response from the Kafka broker. **This will limit throughput, because of waiting for ack. Also Producer don’t do anything in case of success sending. Only in failure case, some retry logic will be added at Producer side. “max.inflight.message” decides max number of ack to wait.**

Also note that a single producer object can be used by multiple threads to send messages, or you can use multiple producers. You will probably want to start with one producer and one thread. If you need better throughput, you can add more threads that use the same producer

**SENDING A MESSAGE SYNCHRONOUSLY**

ProducerRecord<String, String> record =

new ProducerRecord<>("CustomerCountry", "Precision Products", "France");

producer.send(record).get(); // wait on the response

**Kafka Producer Exception and Errors:**

KafkaProducer has two types of errors.

1. Retriable errors Ex: Broker is not available. Try alternate broker
2. Non-Retriable errors Ex: Message size too large.

**SENDING MESSAGE ASYNCHRONOUSLY**

In most cases, producer really don’t need a reply - Kafka sends back the topic, partition and offset of the record after it was written and this information is usually not required by the sending app. In order to send messages asynchronously and still to handle error scenarios, the Producer supports adding a callback when sending a record.

|  |
| --- |
| private class DemoProducerCallback implements Callback { |
| @Override |
| public void onCompletion(RecordMetadata recordMetadata, Exception e) { |
| if (e != null) { |
| e.printStackTrace(); |
| } |
| } |
| } |
|  |
| ProducerRecord<String, String> record = |
| new ProducerRecord<>("CustomerCountry", "Biomedical Materials", "USA"); |
| producer.send(record, new DemoProducerCallback()); |

**How to connect to multiple brokers**

**kafkaProps.put("bootstrap.servers","broker1:9092,broker2:9092");**

**Producer Acks**

When using a producer, you can configure its acks (Acknowledgments) which default to “all”. The acks config setting is the write-acknowledgment received count required from partition leader before the producer write request is deemed complete. This setting controls the producer’s durability which can be very strong (all) or none. Durability is a tradeoff between throughput and consistency. The acks setting is set to “all” (-1), “none” (0), or “leader” (1).

**Acks 0 (NONE)**

The acks=0 is none meaning the Producer does not wait for any ack from Kafka broker at all. The records added to the socket buffer are considered sent. There are no guarantees of durability. The record offset returned from the send method is set to -1 (unknown). There could be record loss if the leader is down. There could be use cases that need to maximize throughput over durability, for example, log aggregation.

**Acks 1 (LEADER)**

The acks=1 is leader acknowledgment. The means that the Kafka broker acknowledges that the partition leader wrote the record to its local log but responds without the partition followers confirming the write. If leader fails right after sending ack, the record could be lost as the followers might not have replicated the record yet. Record loss is rare but possible, and you might only see this used if a rarely missed record is not statistically significant, log aggregation, a collection of data for machine learning or dashboards, etc.

**Acks -1 (ALL)**

The *acks=all* or *acks=-1* is all acknowledgment which means the leader gets write confirmation from the full set of ISRs before sending an ack back to the producer. This guarantees that a record is not lost as long as one ISR remains alive. This ack=all setting is the ***strongest*** available guarantee that Kafka provides for durability.  
This setting is even stronger with broker setting *min.insync.replicas* which specifies the minimum number of ISRs that must acknowledge a write. Most use cases will use ***acks=all*** and set a ***min.insync.replicas > 1***.