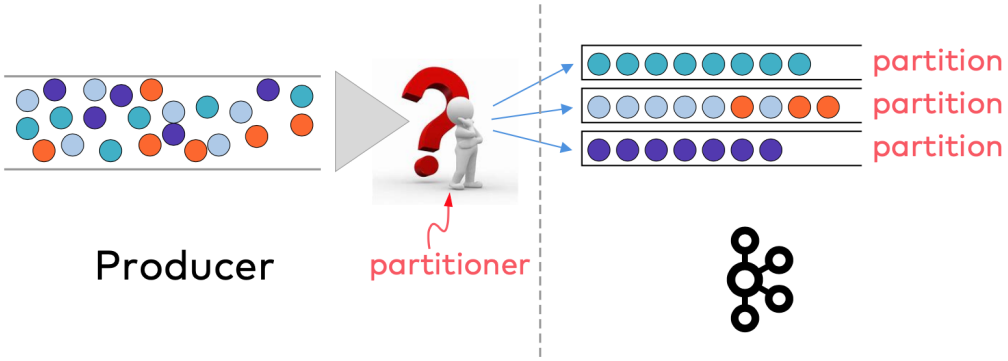
Producer Design

- A producer **sends** data to Kafka
- The producer (optionally) receives an **ACK** or **NACK** from Kafka
- If an **ACK** (acknowledged) is received then all is good
- If a **NACK** (not acknowledged) is received then the producer knows that Kafka was not
  able to accept the data for whatever reason. In this case the producer automatically retries to send the data.

- Can be written in any language
  - Native: Java, C/C++, Python, Go, .NET, JMS
  - More Languages by Community
  - REST Proxy for any unsupported Language

Producers use a Partitioning Strategy to assign each Message to a Partition



- Why partitioning?
  - Load Balancing
  - Concentrate data for **storage efficiency** and/or **indexing**
  - Consumers need **ordering guarantee**
  - Semantic Partitioning
  - Consumers need to **aggregate** or **join** by some key
- Partitioning Strategy specified by Producer
  - Default Strategy: hash(key) % number_of_partitions
  - No Key → Round-Robin
- Custom Partitioner possible

Basic Producer in Java

BasicProducer.java

```java
package clients;

import java.util.Properties;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerRecord;

public class BasicProducer {
    public static void main(String[] args) {
        System.out.println("*** Starting Basic Producer ***");

        Properties settings = new Properties();
        settings.put("client.id", "basic-producer-v0.1.0");
        settings.put("bootstrap.servers", "kafka-1:9092,kafka-2:9092");
        settings.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        settings.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

        final KafkaProducer<String, String> producer = new KafkaProducer<>(settings);

        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
            System.out.println("### Stopping Basic Producer ###");
            producer.close();
        }));

        final String topic = "hello_world_topic";
        for(int i=1; i<=5; i++){
            final String key = "key-" + i;
            final String value = "value-" + i;
            final ProducerRecord<String, String> record = new ProducerRecord<>(topic, key, value);
            producer.send(record);
        }
    }
}
```

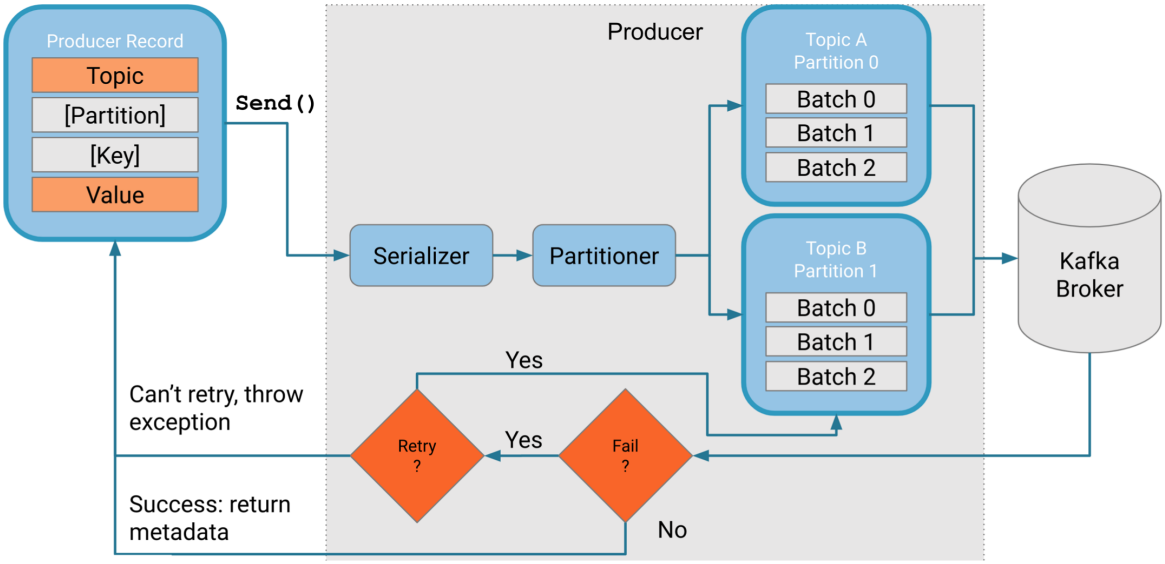
configuration

create producer

shutdown behaviour

sending data

Producer Design



▼ 👇

- The area with the **grey** underlay represents the code of the Kafka Client library - to be
  more precise, of it's producer part. Thus this is not the code that you write, but that is
  available to you with the Kafka client library
- On the left upper side you can see a single record. It contains elements such as the record
  key and value as well as other meta data such as topic name and partition number
- Your application code will send() the message to Kafka
- First your record is serialized into an array of bytes, using the pre-configured serializer
- Then the record is passing through a partitioner. By default the partitioner looks at the
  (serialized) key of the record and decides to which partition of the topic this

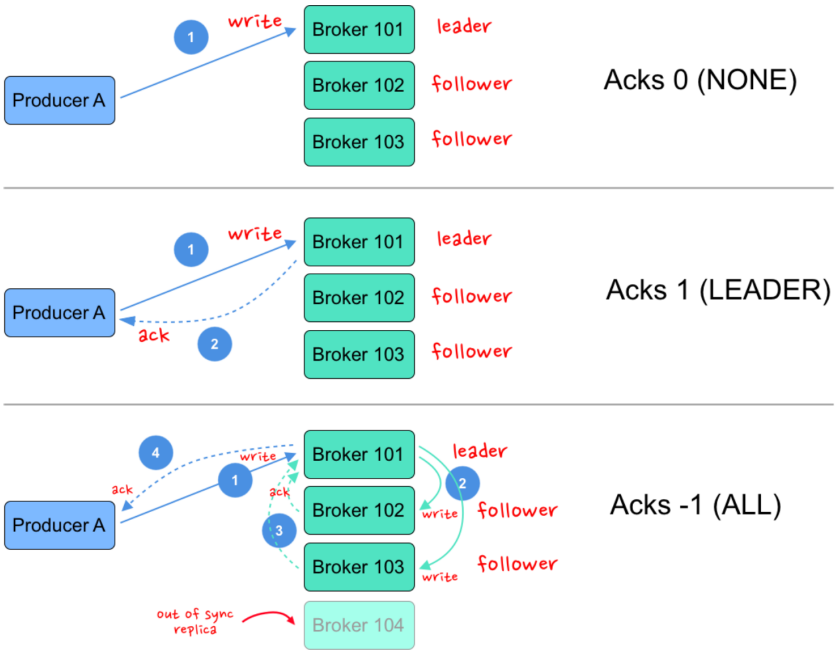
message will
be sent

- No often messages are not sent directly to the respective broker but first batched. This
  depends on some settings the developer can define in their code
- Once a batch is "full" the producer library code flushes the batch to the respective Kafka
  broker
- The broker tries to store the batch in its local commit log
- If the broker was successful it will answer with an ACK and some additional meta data. All

  is good in this case!
- If the broker encounters a failure while trying to save the batch, it will respond with a
  NACK. The producer then automatically tries to resend the same batch of messages
  again, until it succeeds
- If the sending is not successful and the number of retries have been exhausted, then the
  producer triggers an exception which needs to be handled by your code

---

Producer Guarantees



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.

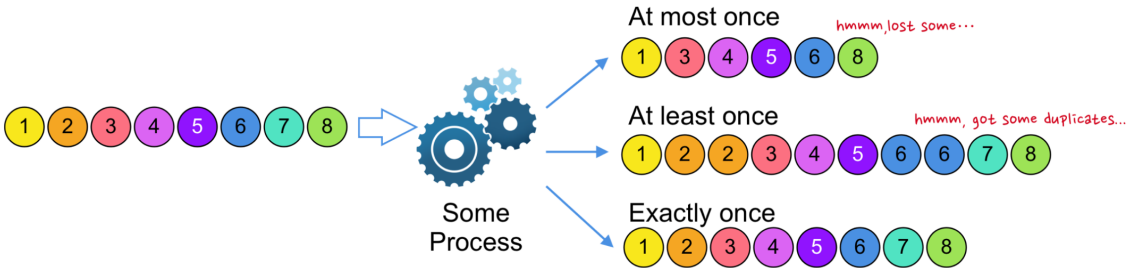
Most use cases will use acks=all and set a min.insync.replicas > 1.

In summary,

`acks` controls the acknowledgment behavior from the producer's perspective

while `min.insync.replicas` controls the minimum number of replicas that must be in sync to consider a write as successful from the broker's perspective.

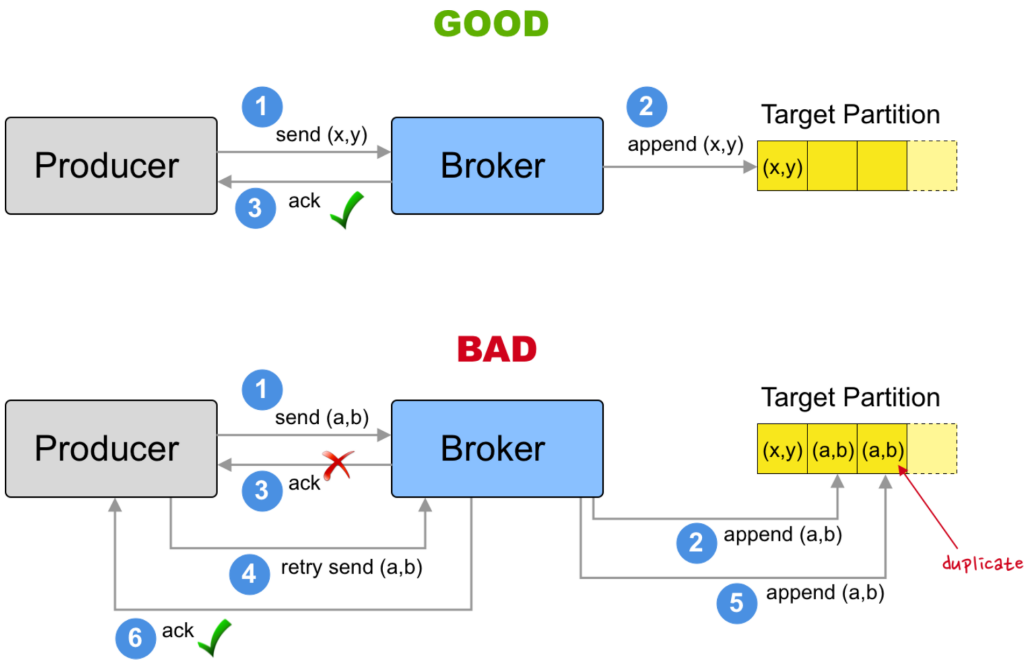
Delivery Guarantees



Kafka supports 3 delivery guarantees:

- **At most once:** From all records written to Kafka it is guaranteed that there will never be a
  duplicate. Under certain bad circumstances some record may be lost
- **At least once:** From all records written to Kafka none is ever lost. Under certain bad
  situations there could be duplicates in the log
- **Exactly once:** Every single record written to Kafka will be found in the Kafka logs exactly
  once. There is no situation where either a record is lost or where a record is duplicated

Idempotent Producers



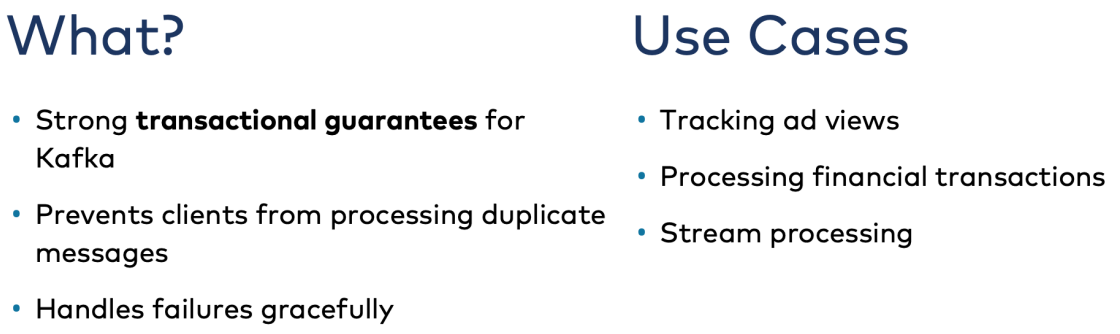
- To achieve exactly once semantics (EOS) the first step is to establish **idempotent** producers.
- Idempotence in producers is enabled by default (enable.idempotence = true) from CP
  7.0 / AK 3.0.
- If a producer is not idempotent. Upon failure one gets duplicates in the commit log.

But, an idempotent producer guarantees, in collaboration with the respective broker, that:

- all messages written to a specific partition are maintaining their relative order in the commit log of the broker
- each message is only written once. No duplicates ever are to be found in the commit log
- together with acks=all we also make sure that no message is ever lost

---

Exactly Once Semantics



For the longest time it seemed an impossibility to have transactional guarantees in a highly
distributed and asynchronous system such as Kafka.

But EOS gives us exactly this behavior.

Exactly Once Semantics (EOS) bring strong transactional guarantees to Kafka, preventing
duplicate messages from being processed by client applications, even in the event of client
retries and Broker failures

Use cases:

- tracking ad views
- processing financial transactions
- stream processing with e.g. aggregates only really makes sense with EOS

Process:

- A producer can start a transaction
- The producer then writes several records to multiple partitions on different brokers
- The producer can then commit the transaction
- If the TX succeeds, then the producer has the guarantee, that all records have been written exactly once and maintaining the local ordering to the Kafka brokers
- If the TX fails, then the producer knows that none of the record written will be showing up
  in the downstream consumers.
- That is, the aborted TX will leave no unwanted side effects
- NOTE: To have this downstream guarantee the consumers need to set their reading behavior to
  read committed