Apache Kafka for Java Developers

Producing Records

Boris Fresow, Markus Günther

JavaLand 2024, Nürburgring

The Apache Kafka Clients SDK

A Producer's Perspective

Clients for Apache Kafka come in different sizes and shapes.

- Built-in CLI
- Java-based clients
  - Apache Kafka Client library
  - Apache Kafka for Spring
  - SmallRye Reactive Messaging with Kafka
- `librdkafka`-based clients
  - C, Python, ...

The official Apache Kafka Client library is only a single dependency away.

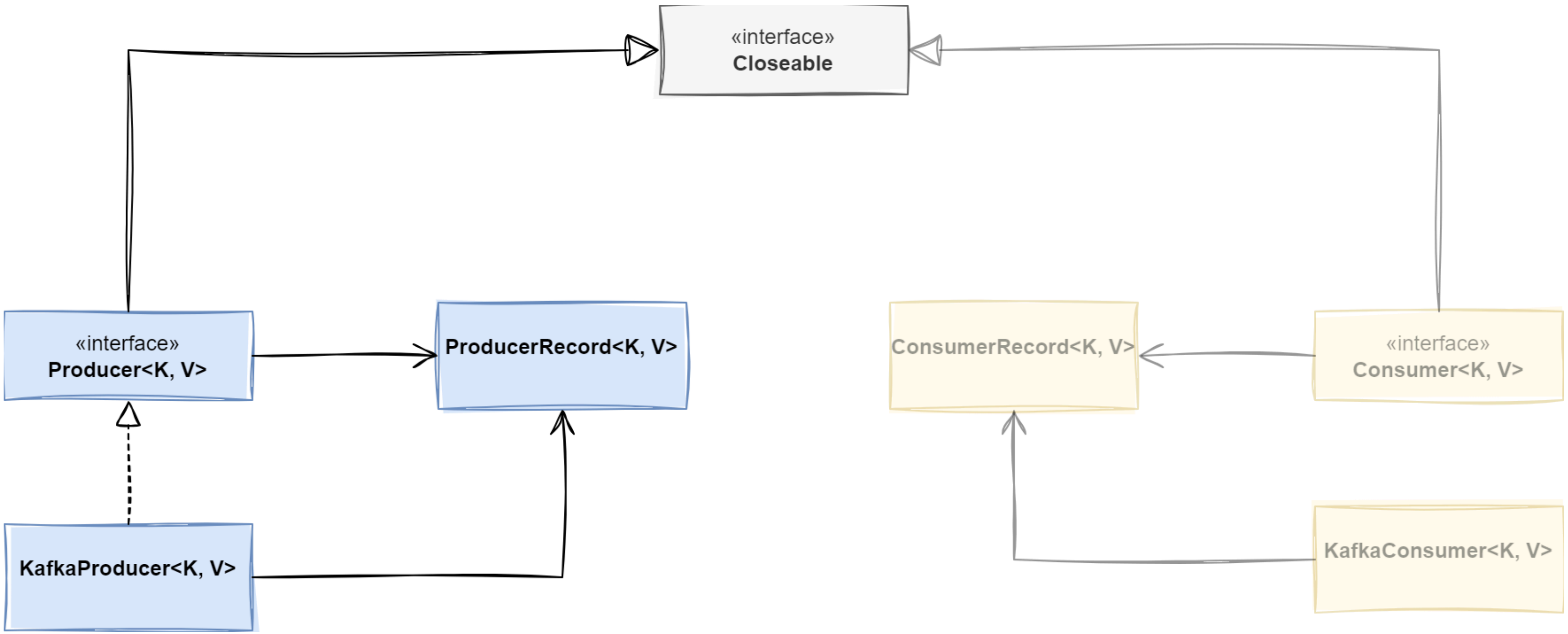
Maven

```xml
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-clients</groupId>
  <version>3.6.1</version>
</dependency>
```

Gradle

```groovy
dependencies {
  implementation "org.apache.kafka:kafka-clients:3.6.1"
}
```

We are able to write a basic producer by using just these few classes on the left.



- `Producer<K,V>`: This is the public interface of the producer client. It comprises the necessary methods for publishing records and using transactions.
- `KafkaProducer<K,V>`: This is the implementation of the `Producer` interface. Whereas the interface is barely documented, the implementation class offers rich documentation for the class itself as well as for every method of its public API.
- `ProducerRecord<K,V>`: This is a data structure that comprises all attributes of a Kafka record as perceived by a producer. To be more precise, this is the representation of a record **before** it has been published to a partition. It contains topic name, partition number, an optional set of headers, key, value, and a timestamp.

The parametric types `K` and `V` stand for the key resp. the value of of the record.

A Producer<K, V> has methods for publishing records and managing transactions.

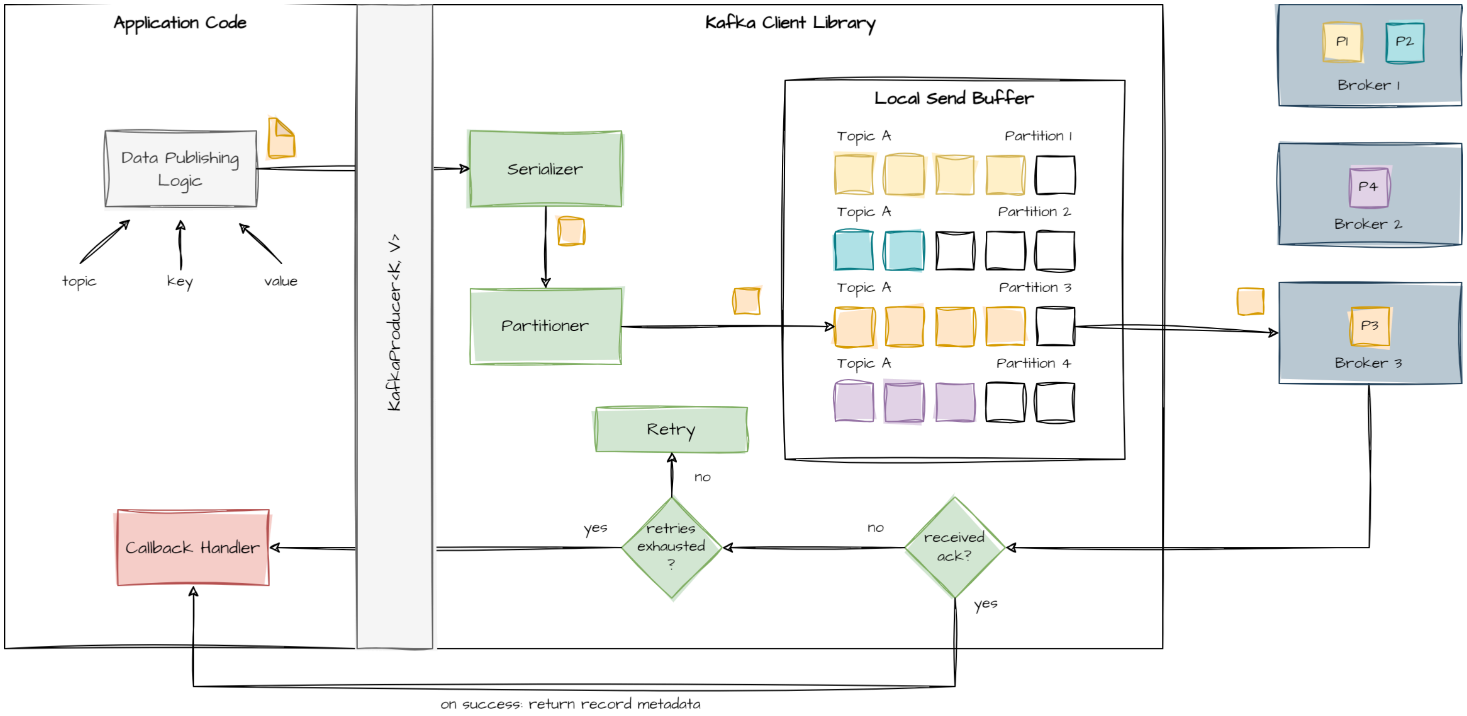
```java
public interface Producer<K, V> extends Closeable {

  // methods for publishing records
  Future<RecordMetadata> send(ProducerRecord<K, V> record);

  Future<RecordMetadata> send(ProducerRecord<K, V> record, Callback callback);

  // methods for managing transactional publishing
  void initTransactions();

  void beginTransaction() throws ProducerFencedException;

  void commitTransaction() throws ProducerFencedException;

  void abortTransaction() throws ProducerFencedException;

  // ... a couple of more methods omitted for brevity ...
}
```

The `send()` methods operate asynchronously, meaning they complete immediately after the record is serialized and placed into a local buffer. The process of actually transmitting the record occurs in the background, managed by a dedicated I/O thread. If the application needs to wait for the outcome, it should call the `get()` method on the `Future` object provided.

With this knowledge in mind, we are able to write a first, yet simple, producer!

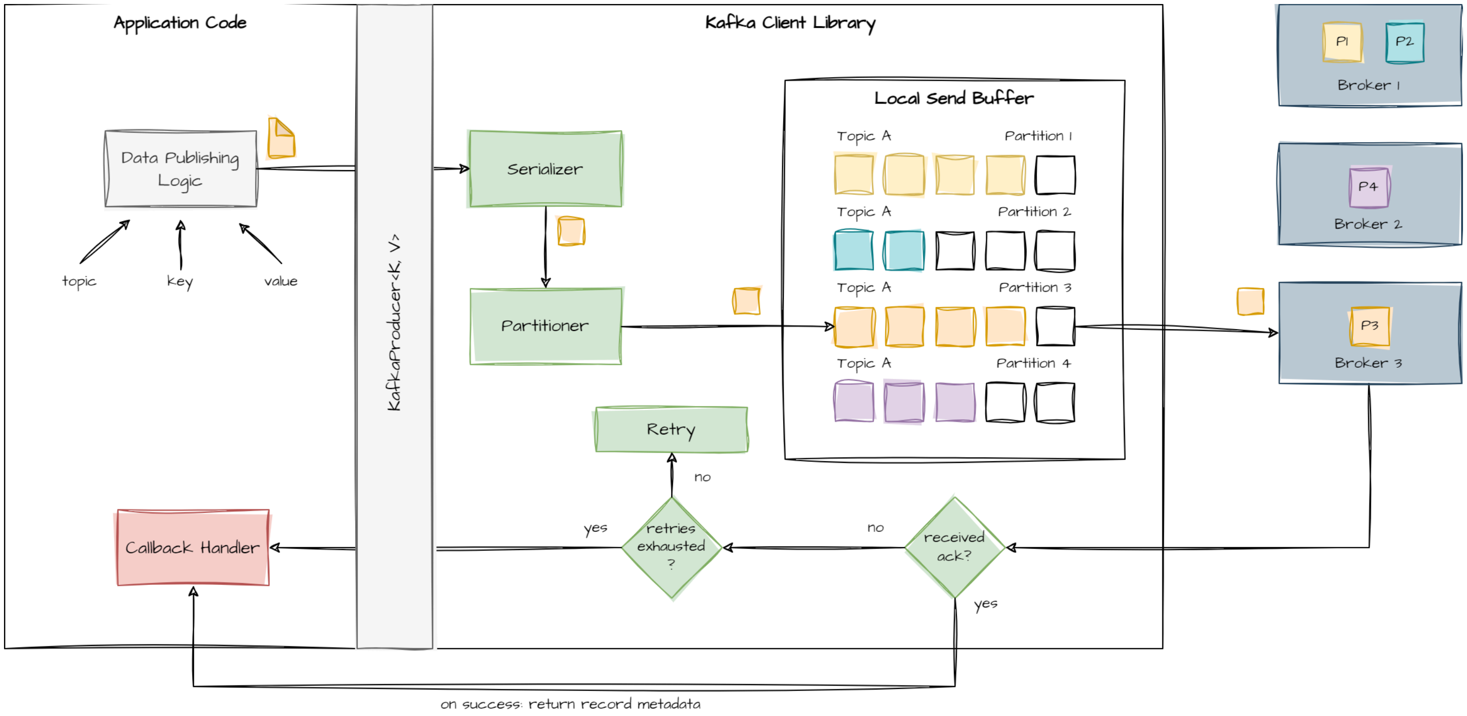
```java
public class BasicProducer {
  public static void main(String[] args) throws Exception {
    var topic = "getting-started";
    Map<String, Object> config = Map.of(
      ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092",
      ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName(),
      ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName()
    );
    try (var producer = new KafkaProducer<String, String>(config)) {
      var key = "my-key";
      var value = new Date().toString();
      Callback callback = (metadata, exception) -> {
        System.out.println("Published with metadata: %s, error: %s%n",
          metadata, exception);
      };
      producer.send(new ProducerRecord<>(topic, key, value), callback);
    }
  }
}
```

But what happens after we call `producer.send`?



Serialization

A `Serializer` encodes data of type T into `byte[]`.



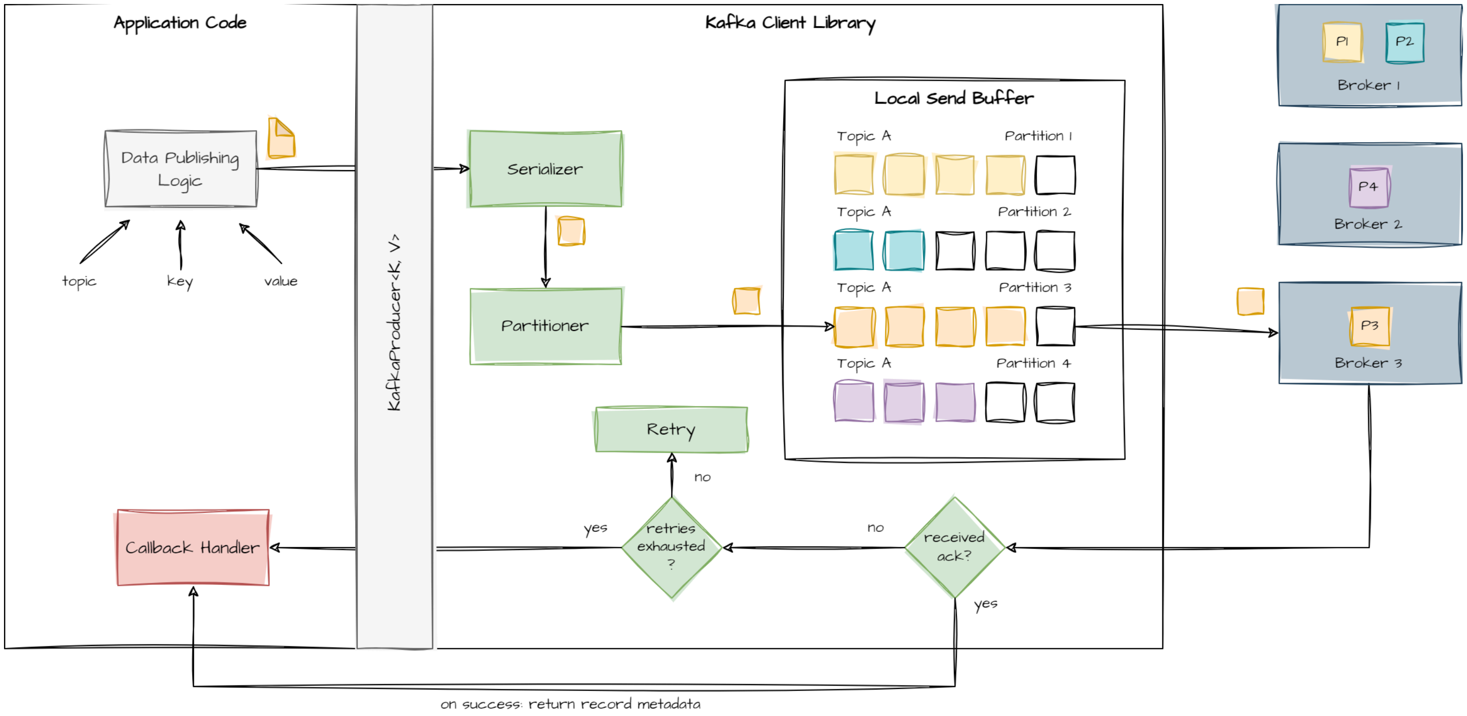
A Serializer encodes data of type T into byte[]. (cont.)

```java
public interface Serializer<T> extends Closeable {

  byte[] serialize(String topic, T data);

  default byte[] serialize(String topic, Headers headers, T data) {
    return serialize(topic, data);
  }

  default void configure(Map<String, ?> configs, boolean isKey) {
    // intentionally left blank
  }

  @Override
  default void close() {
    // intentionally left blank
  }
}
```

A `Serializer` encodes the given data into `byte[]`. This is the target format for Apache Kafka. Kafka does not interpret the payload of a record. It only deals with `byte[]`. Thus, serialization is a concern of the application.

Partition Assignment

After serialization, a partitioner sorts data into buffers for their resp. topic-partition.



Partition assignment factors in different strategies.

1. Use the partition of the `ProducerRecord<K,V>` (if assigned)
2. Use a custom `Partitioner` (if configured)
3. Try to calculate the partition based on record key (if not null)
    1. uses a hash function over the key
    2. default for Java-based clients is `murmur2`
    3. default for `librdkafka` is `crc32`
4. If there is no key or key should be ignored, use built-in partitioning logic

The `murmur2`-hash is calculated like this:

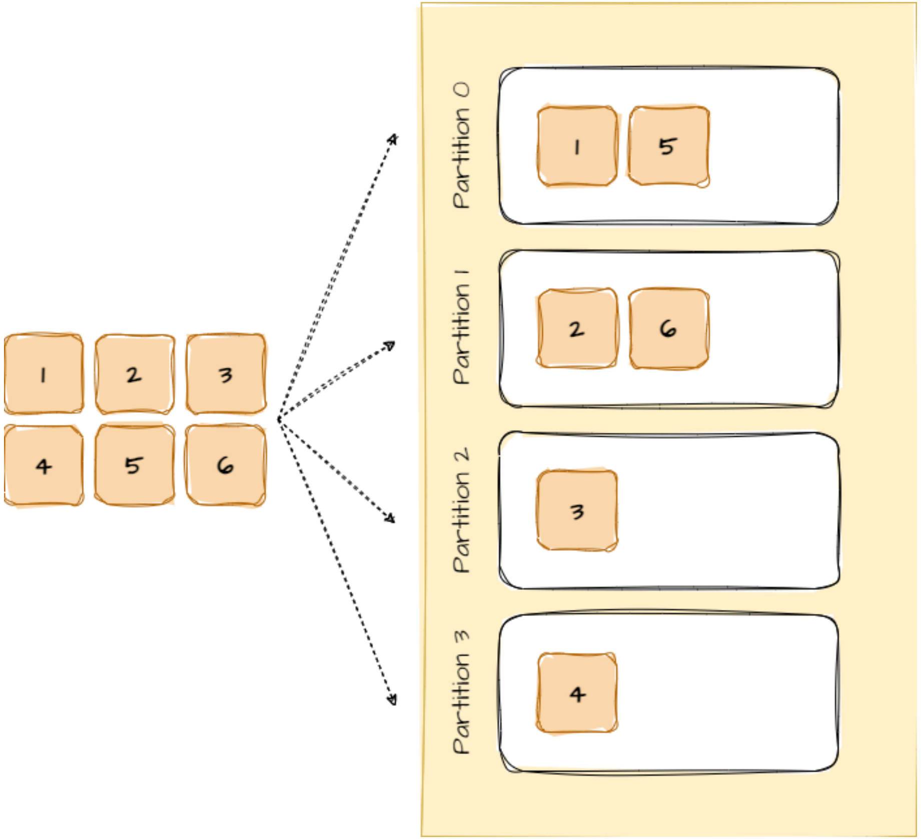
```
int targetPartition = Math.abs(Utils.murmur2(keyBytes)) % (numPartitions - 1);
```

Moving to the **Sticky Partitioner** is not just about the fact that the previously used round-robin strategy led to an imbalanced workload between topic-partitions (cf. KAFKA-9965), but also results in a significant increase in performance (cf. KIP-480: Sticky Partitioner).

For Kafka 2.3 and below, the default partitioner uses a round-robin strategy.

Round Robin Partitioner

- one batch per partition
  - more batches
  - smaller batches
- leads to
  - more requests
  - higher latency
- bug: leads to uneven distribution (2.4+)

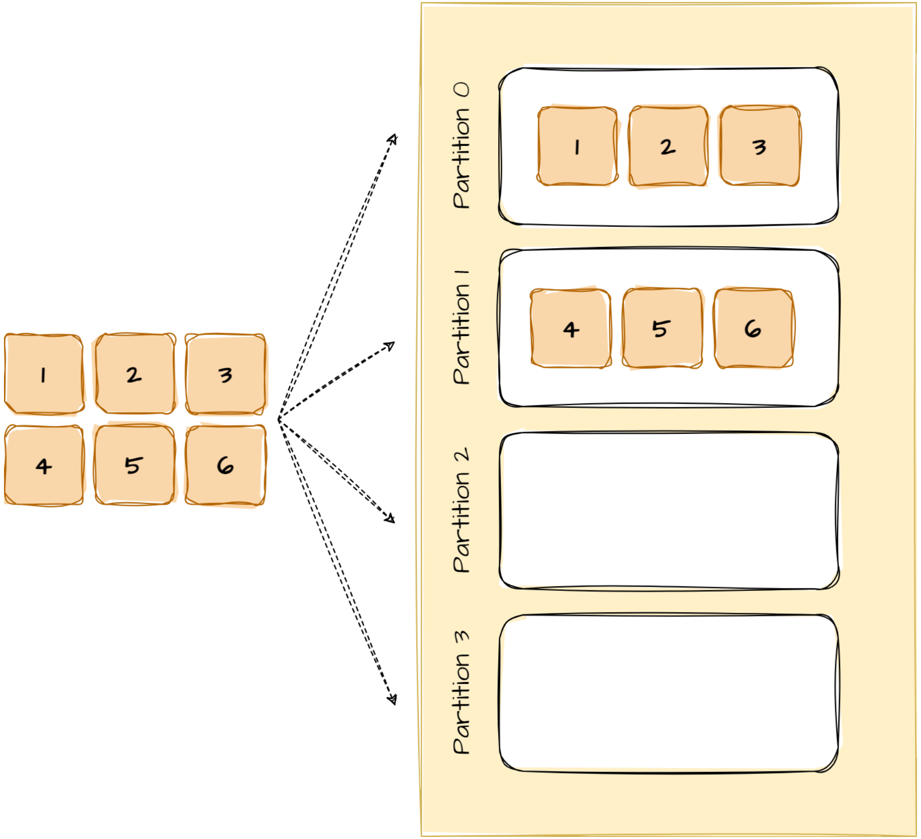


There is an bug that leads to an uneven distribution if the `RoundRobinPartitioner` is used in Kafka 2.4+. See https://issues.apache.org/jira/browse/KAFKA-9965 for details.

For Kafka 2.4. and up to Kafka 3.2, the default partitioner uses batch stickiness.

Sticky Partitioner

- stick to a partition
  - until batch is full
  - `linger.ms` has elapsed
- leads to
  - larger batches
  - reduced latency
- bug: uneven distribution (slow brokers)



This will lead to larger batches and reduced latency (due to larger requests, `batch.size` is more likely to be reached). Over time, the records are still spread *almost evenly* across partitions, so the balance of the cluster is not affected.

For Kafka 3.3 and higher, there are two new strategies available.

Uniform Sticky Batch Size

- don't switch unless `batch.size` bytes got produced to partition
- uniform throughput and data distribution
  - adapts well to higher latency brokers
- might slow down producer rate due to filling local buffer if brokers are constantly lagging behind

For Kafka 3.3 and higher, there are two new strategies available.

Uniform Sticky Batch Size

- don't switch unless `batch.size` bytes got produced to partition
- uniform throughput and data distribution
  - adapts well to higher latency brokers
- might slow down producer rate due to filling local buffer if brokers are constantly lagging behind

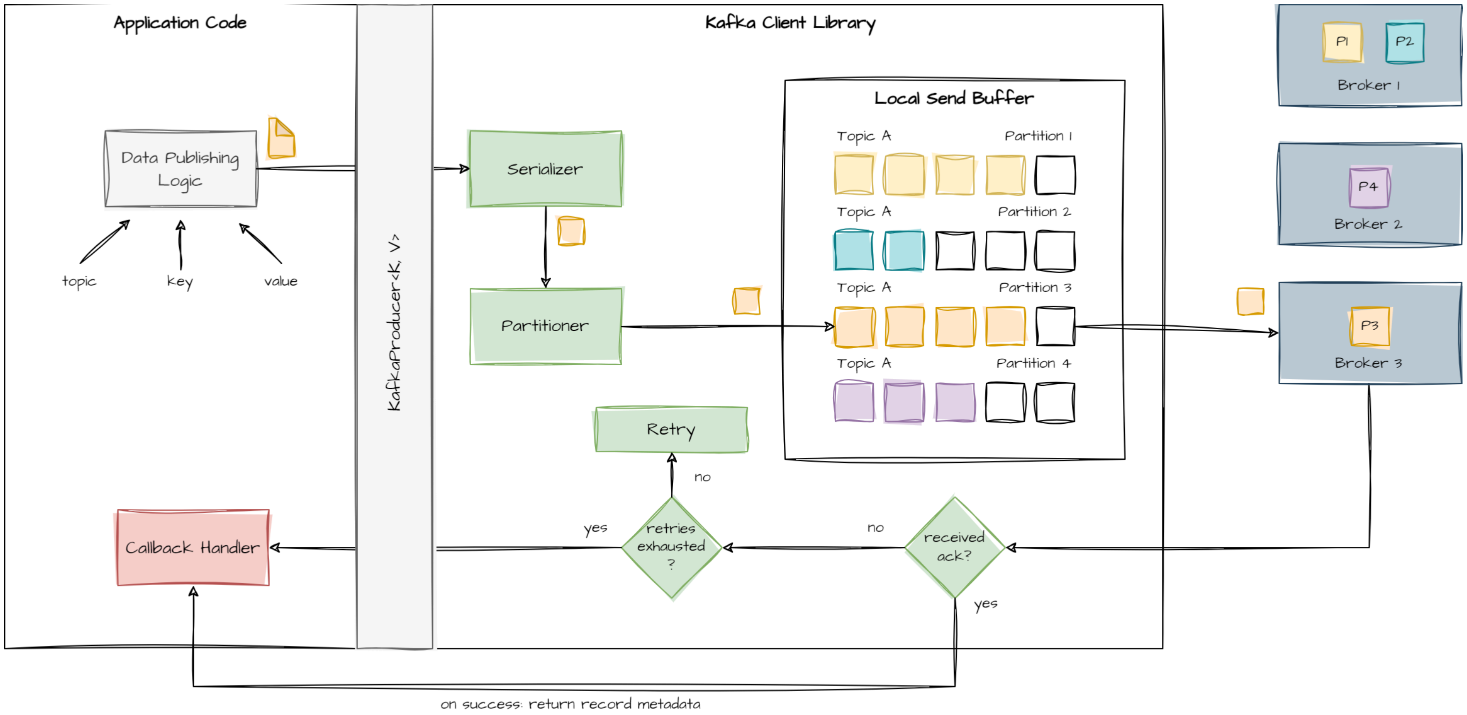
Adaptive Partition Switching (default)

- adapts to broker load
  - queue size of unsent batches is indicator
  - probability of choosing a partition is proportional to the inverse queue size
  - partitions with longer queues are less likely to be chosen
- `partitioner.availability.timeout.ms` > 0 to indicate a failed batch if the producer is unable to produce data within timeout

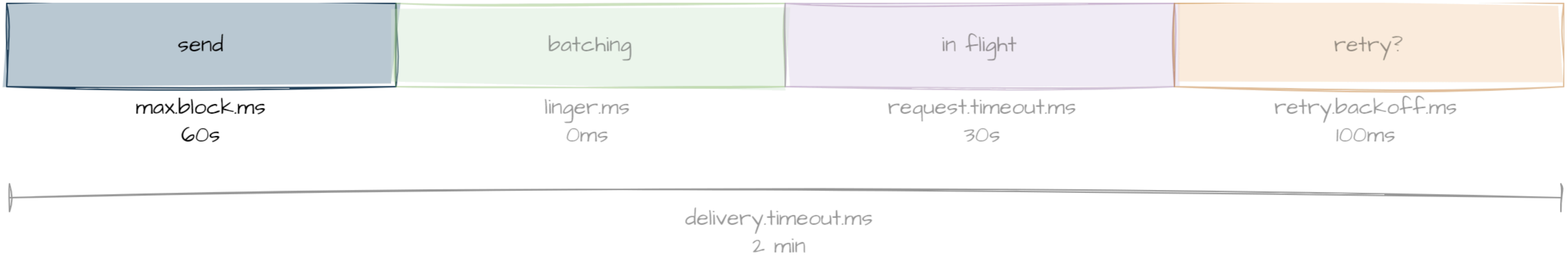
Set `partitioner.adaptive.partitioning.enable` to `false` to use **Uniform Sticky Batch Size**. If set to `true` (default), **Adaptive Partition Switching** is used.

Timeouts

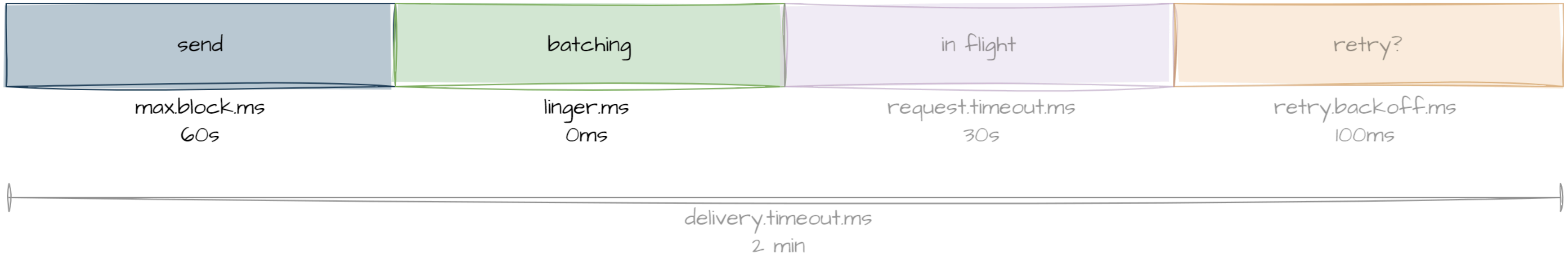
After partitioning, data is moved into a local buffer for the target topic-partition.



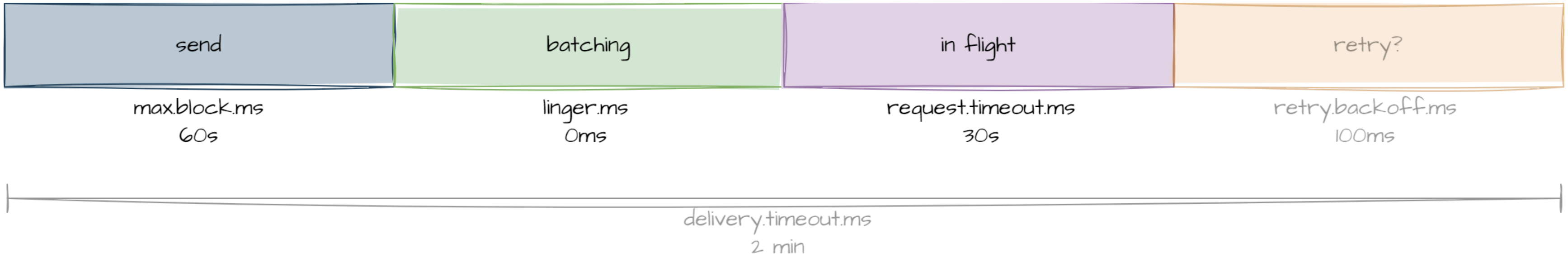
The `send` method of the producer waits a maximum of `max.block.ms`.



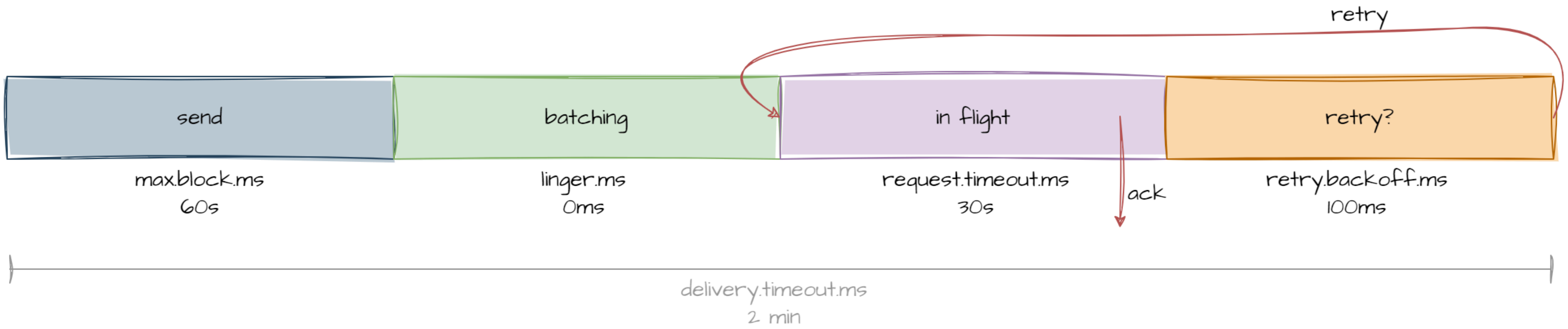
The producer waits a maximum of `linger.ms` for messages to include into a batch.



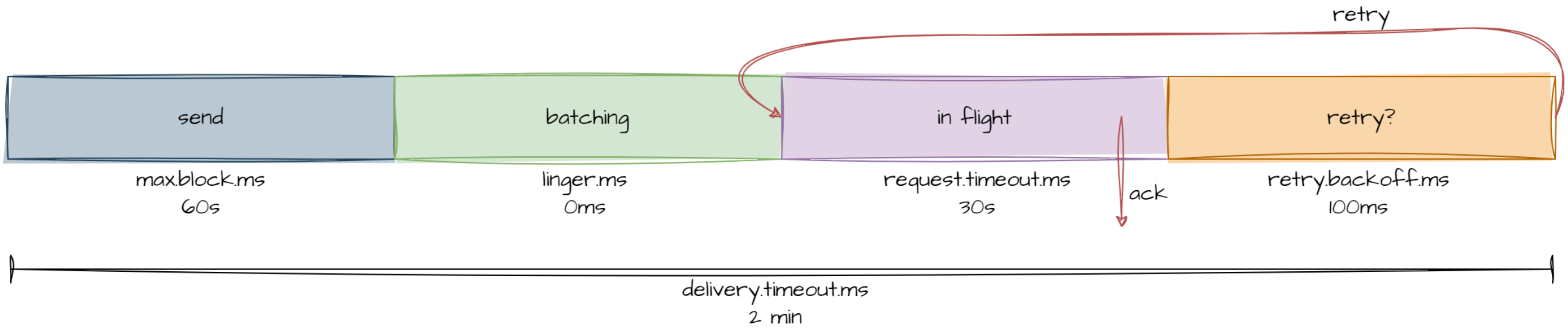
The producer waits for `request.timeout.ms` for an acknowledgement.



If the producer receives no ack within a certain time, its retry mechanism fires.



After a maximum of `delivery.timeout.ms` the attempt to publish will be aborted.



Acknowledgements

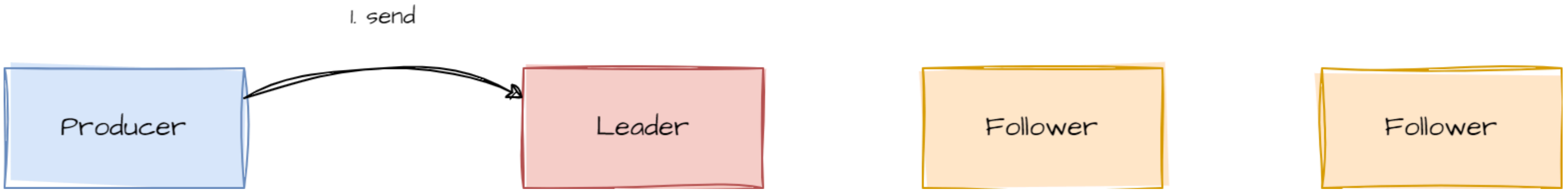
Depending on your acks setting, the producer waits for an acknowledgment (or not).

General

- Producer is able to submit data anew in case of missing ack
- If the error is persistent, the producer will generate an exception

> ***Acknowledgements*** *are not only used to signal that data has been received, but also play a part in Kafka's* ***replication strategy****.*

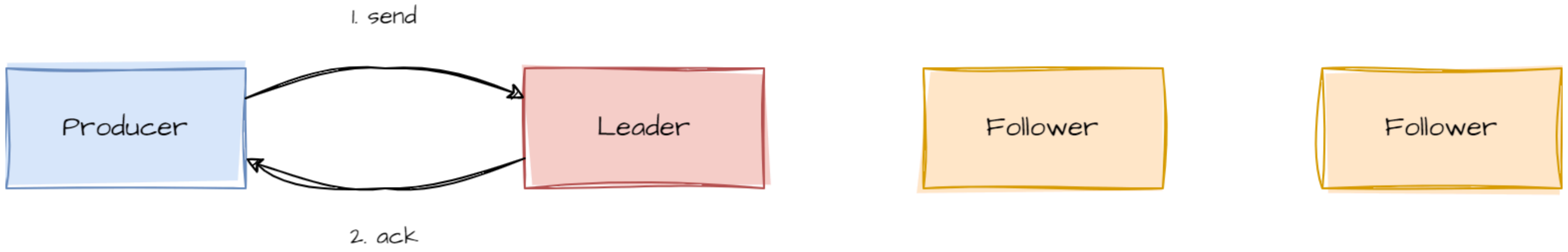
The producer does not wait for an acknowledgment if `acks=0`.



Traits

- Fire-and-forget
- Networking analogy: UDP
- Best performance, if data loss is tolerable

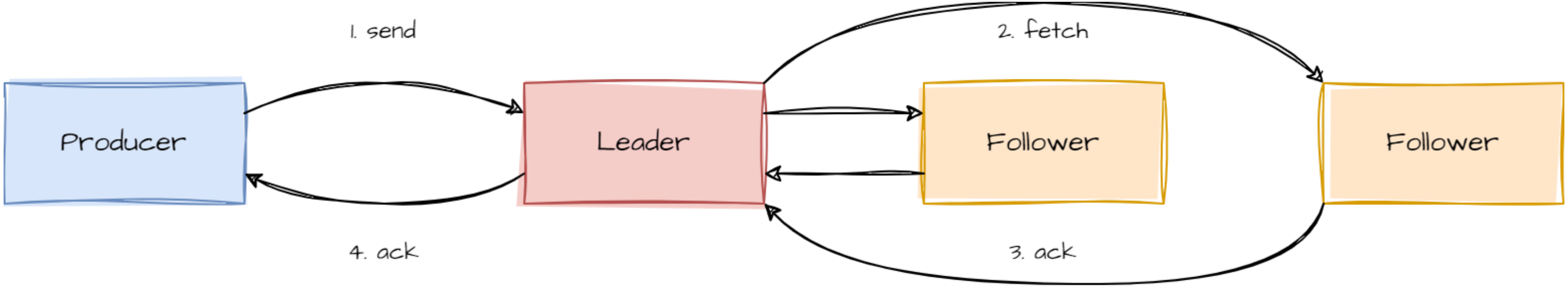
The leader acknowledges directly after receiving the record if `acks=1`.



Traits

- Does not wait for the replication result to followers
- Networking analogy: TCP
- Default configuration up until Apache Kafka 3.0

The leader replicates to all followers before it sends the acknowledgment if `acks=all`.



Traits

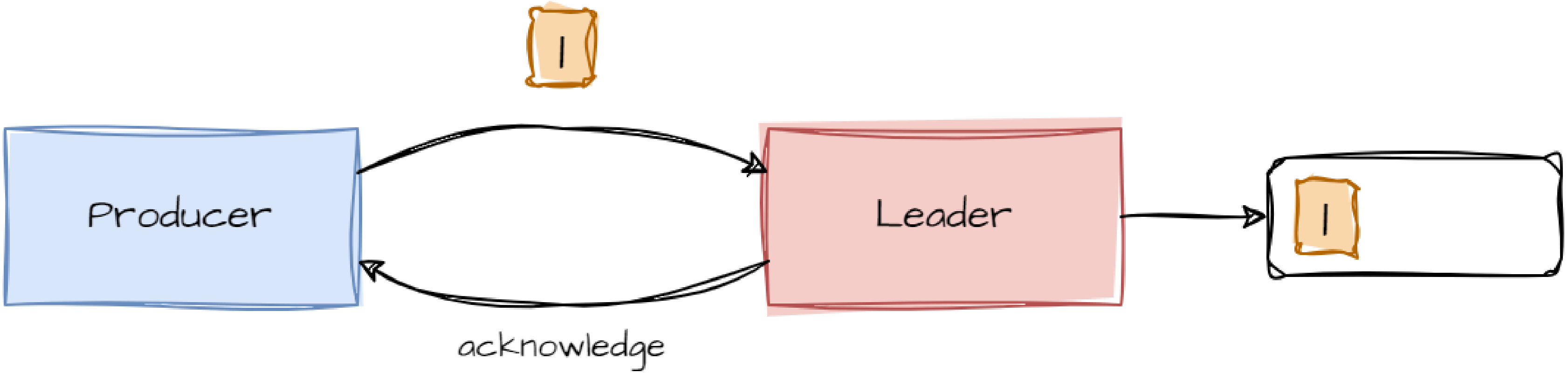
- Replicas are considered in-sync if they received latest data within 30 s
- Best consistency guarantees
- `min.insync.replicas` controls how many brokers must be in-sync

- Best consistency guarantees, but for the lack of a higher throughput

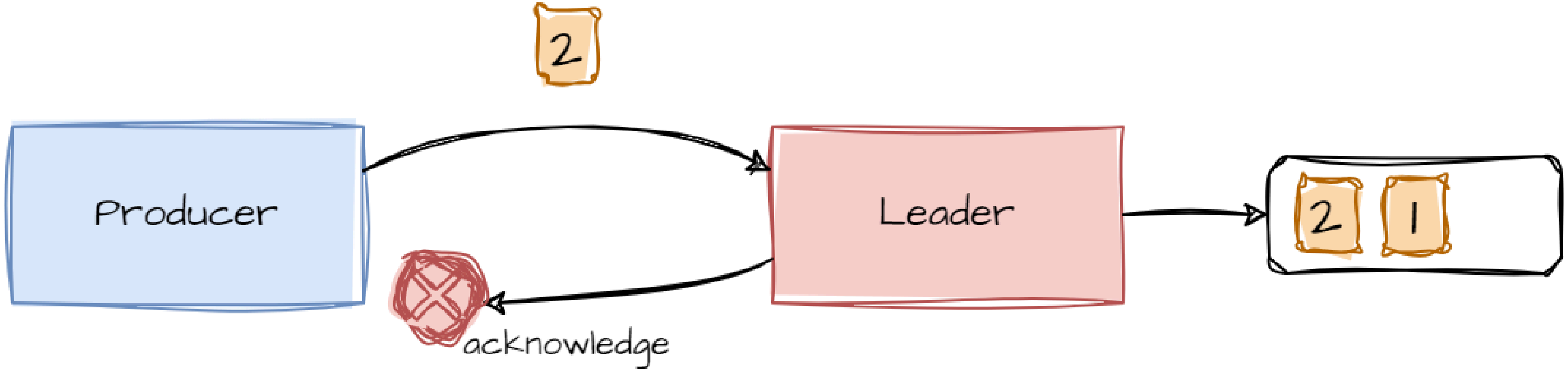
- If there are not enough `min.insync.replicas`, the broker sends an error back to the producer, which will raise a `NotEnoughReplicasException`

Delivery Guarantees

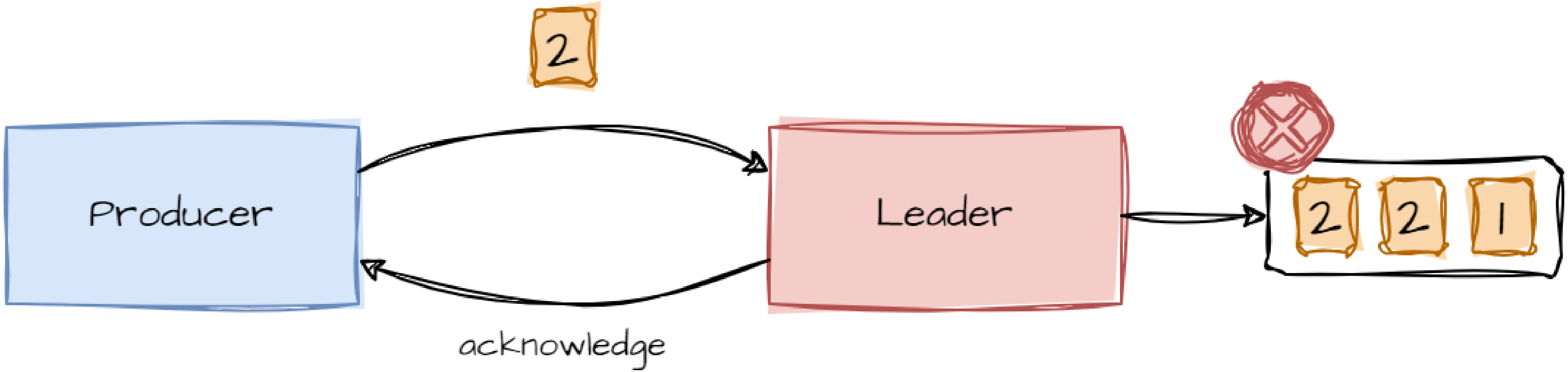
Lost acknowledgments may lead to duplicated records in the log.



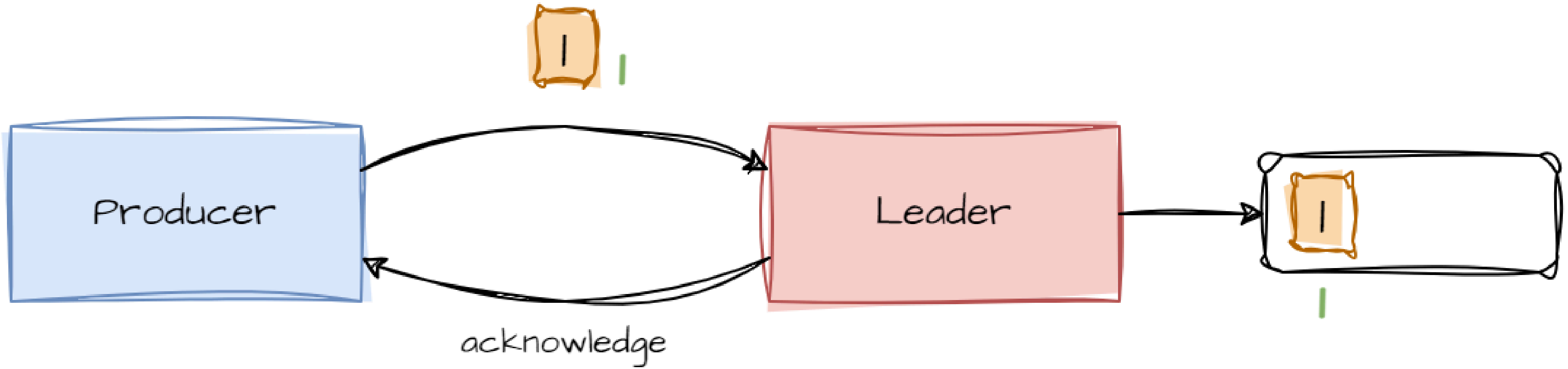
Lost acknowledgments may lead to duplicated records in the log. (cont.)



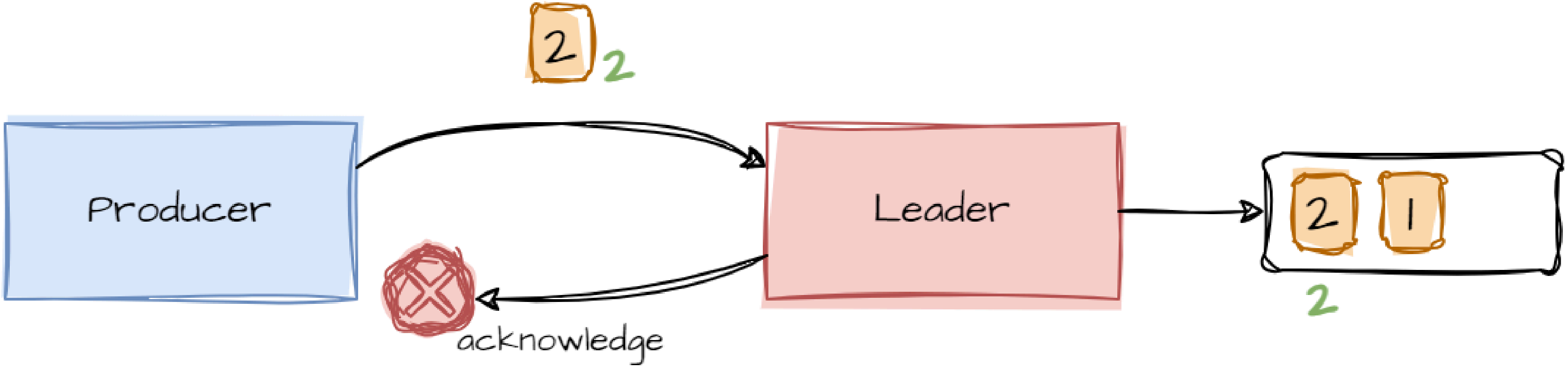
Lost acknowledgments may lead to duplicated records in the log. (cont.)



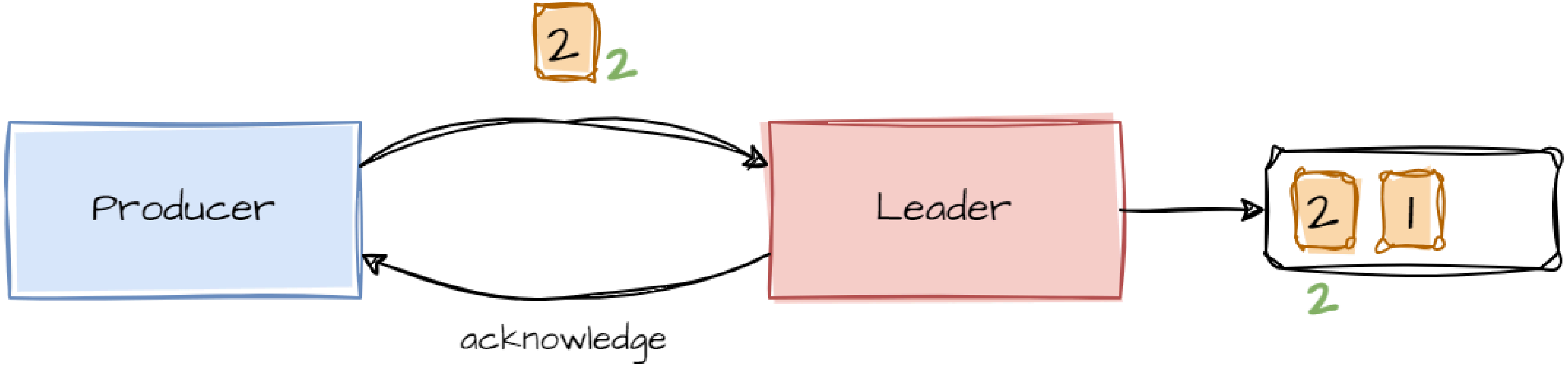
Setting `acks=all` and `enable.idempotence=true` prevents duplicates.



Setting `acks=all` and `enable.idempotence=true` prevents duplicates. (cont.)



Setting `acks=all` and `enable.idempotence=true` prevents duplicates. (cont.)



© 2024 Boris Fresow, Markus Günther

We distinguish between three different delivery guarantees.

At-most-once

> *Guarantees that a record will be delivered **at most one time**. There can be **no duplicates**. There is **no guarantee** that the record will be received from the broker.*

- This is the case for `acks=0`

We distinguish between three different delivery guarantees. (cont.)

At-least-once

> *Guarantees that a record* ***will be received*** *by the broker, but* ***possibly multiple times****. Hence, there* ***may be duplicates****.*

- This the case for
  - `acks=all`
  - `min.insync.replicas` to a sensible value

We distinguish between three different delivery guarantees. (cont.)

Exactly-once

> *Guarantees that a record **will be written to the log** exactly **one time** (idempotency).*

- This is the case for
  - `acks=all`
  - `enable.idempotence=true`
- Default setting since Apache Kafka 3.0

Summary

What did we learn?

- Client SDK Essentials
- Partition Assignment
- Timeouts
- Retries
- Error Handling
- Acknowledgements
- Delivery Guarantees

Summary

What did we learn?

- Client SDK Essentials
- Partition Assignment
- Timeouts
- Retries
- Error Handling
- Acknowledgements
- Delivery Guarantees

What's to follow?

- *Serialization*
- Interceptors
- Broker Internals
- Producer Designs
- Transactions

Questions?