

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.

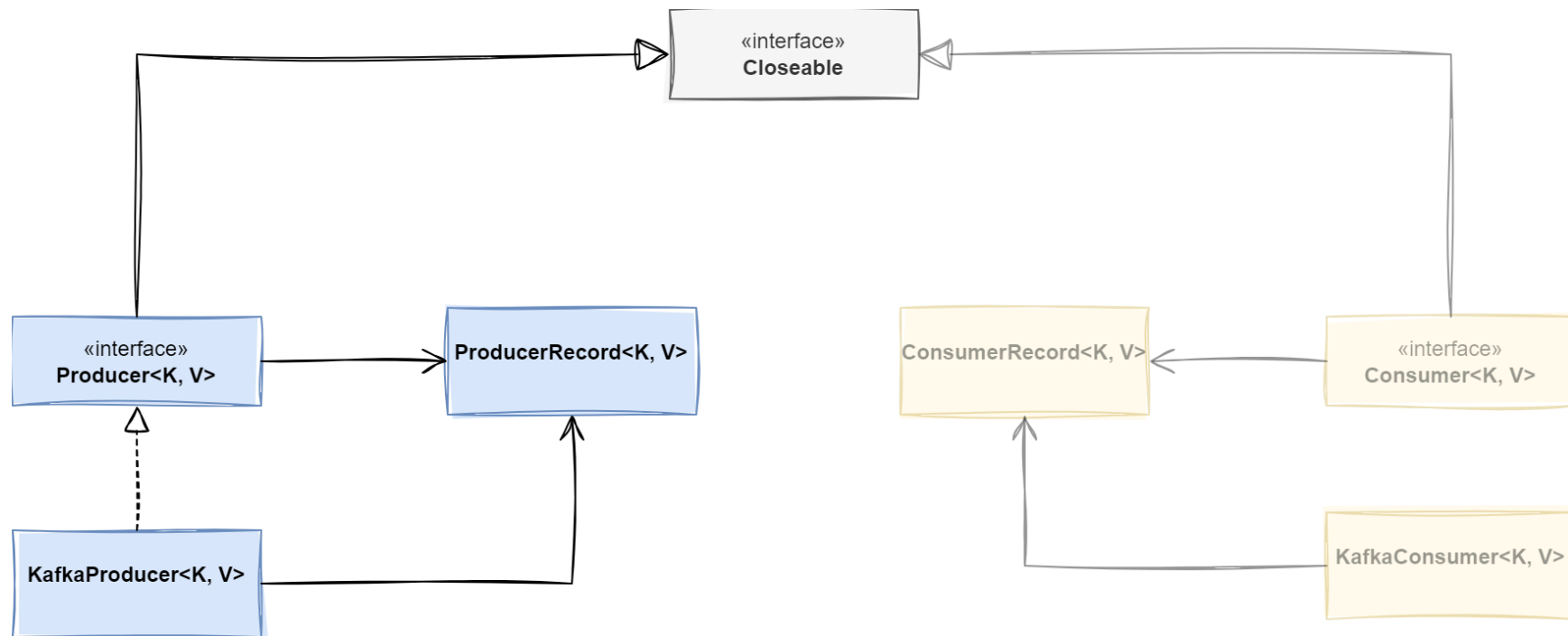
Maven

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

Gradle

```
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.



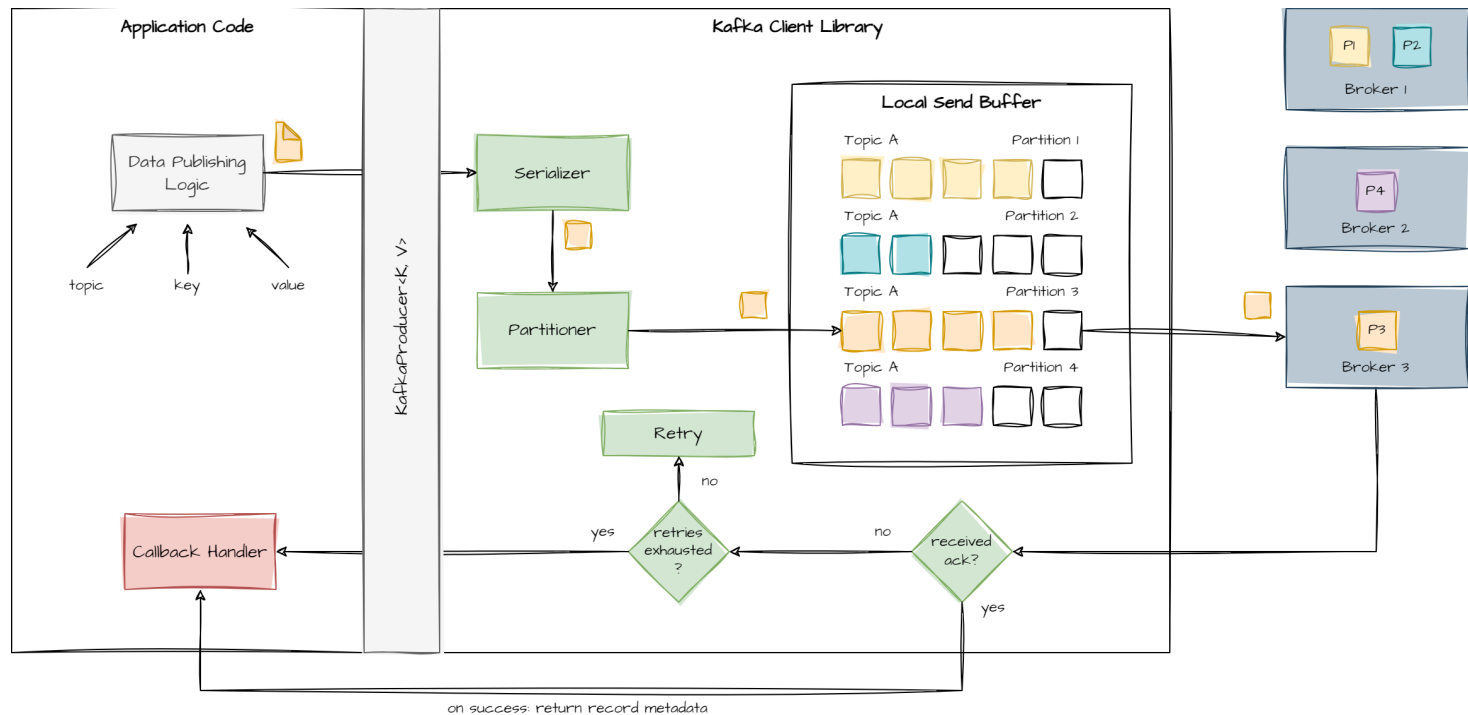
A Producer<K, V> has methods for publishing records and managing transactions.

```
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 ...  
}
```

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

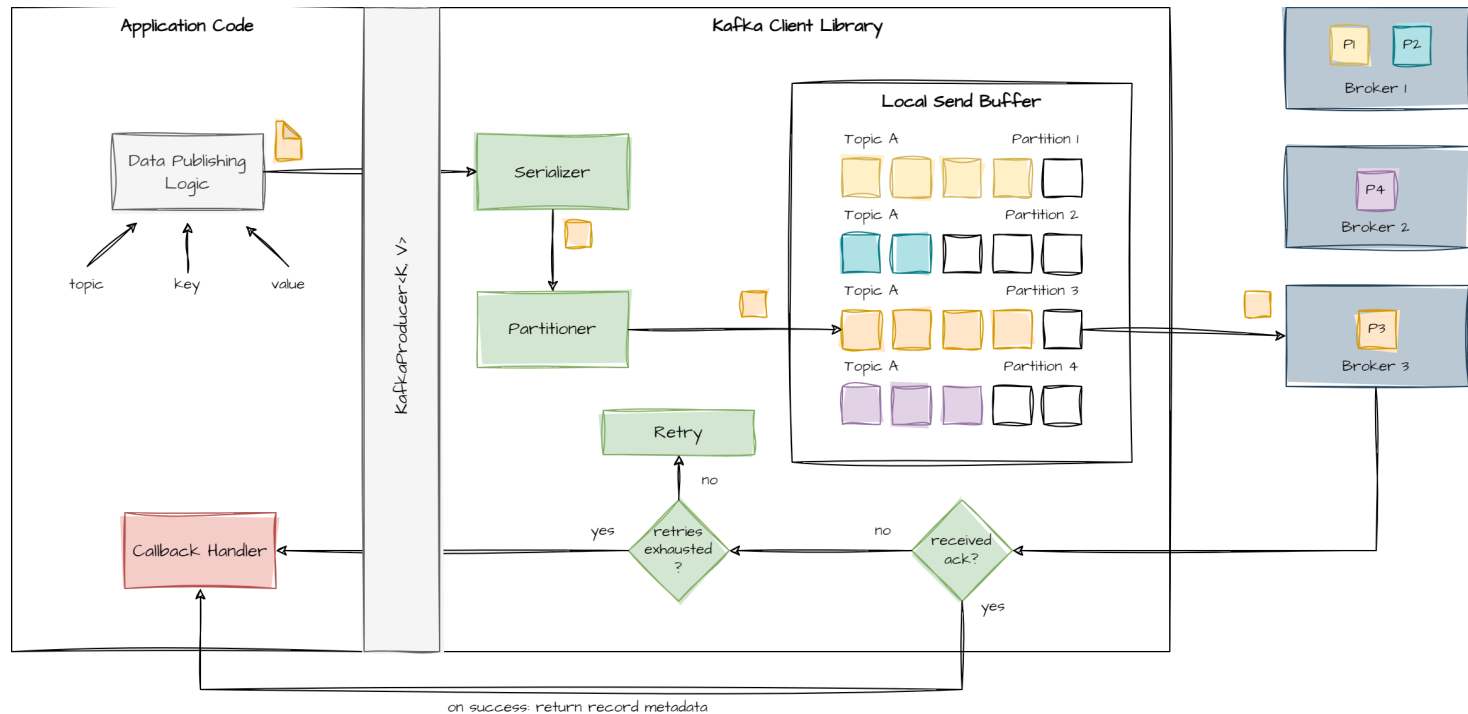
```
public class BasicProducer {  
    public static void main(String[] args) throws Exception {  
        var topic = "getting-started";  
        Map<String, Object> config = Map.of(  
            ProducerConfig.BootstrapServersConfig, "localhost:9092",  
            ProducerConfig.KeySerializerClassConfig, StringSerializer.class.getName(),  
            ProducerConfig.ValueSerializerClassConfig, 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.)

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

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

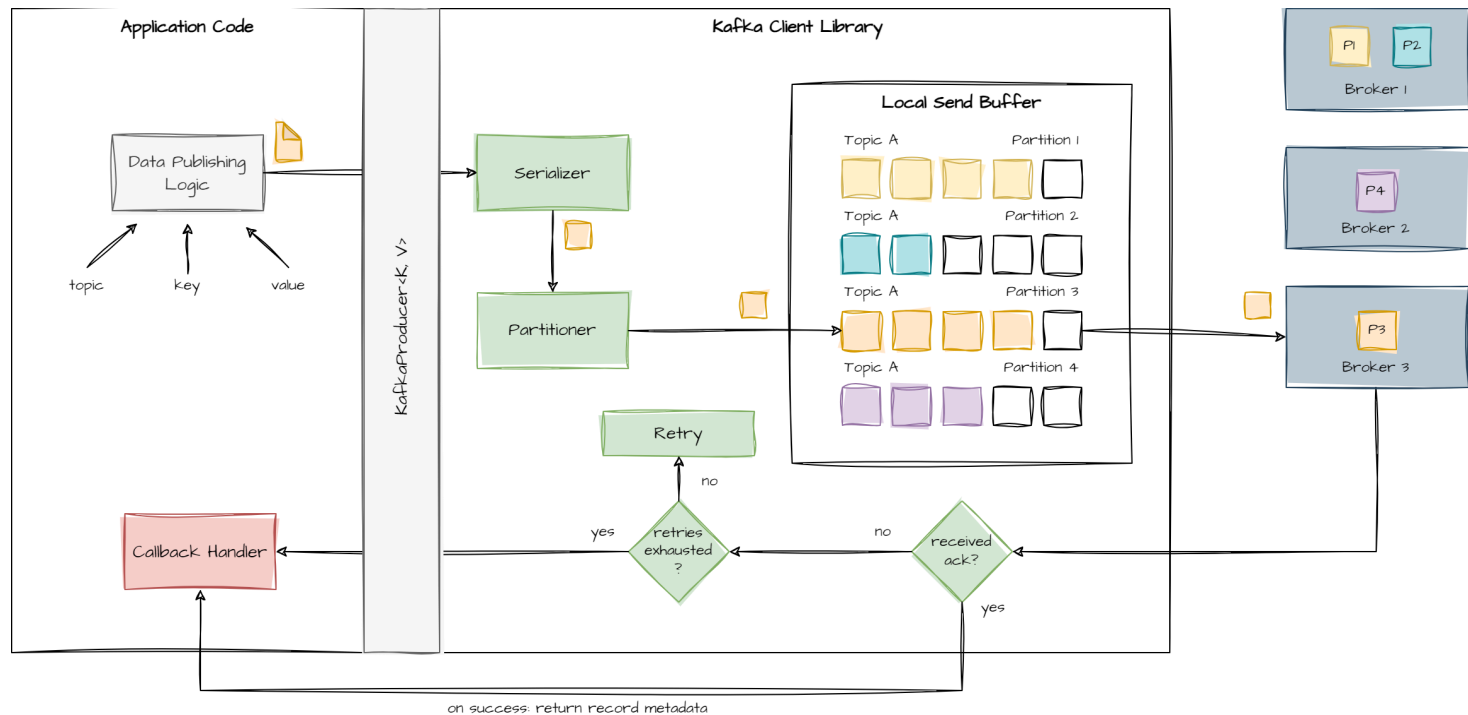
    default byte[] serialize(String topic, Headers headers, T data) {
        return serialize(topic, data);
    }

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

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

Partition Assignment

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



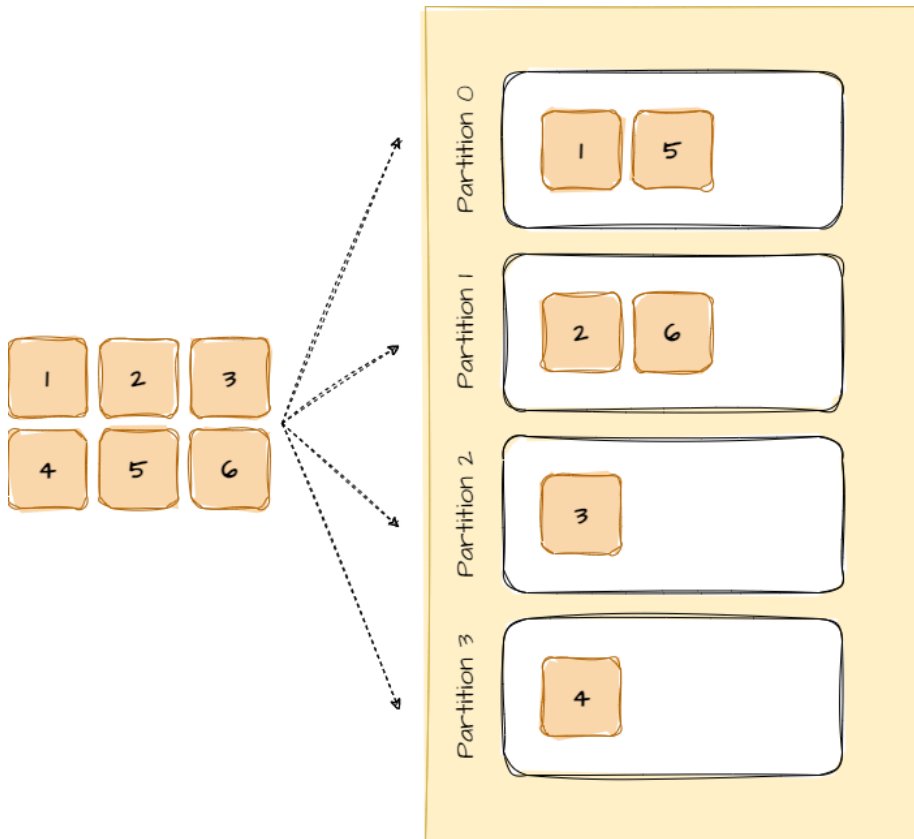
Partition assignment factors in different strategies.

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

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

Round Robin Partitioner

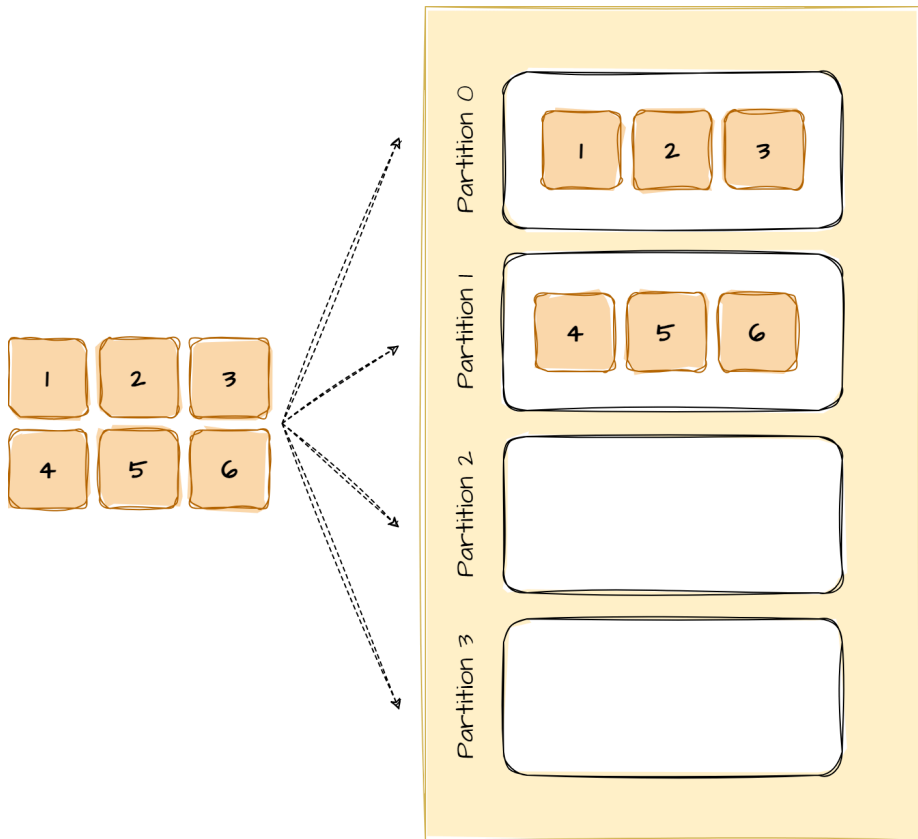
- one batch per partition
 - more batches
 - smaller batches
- leads to
 - more requests
 - higher latency
- bug: leads to uneven distribution (2.4+)



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)



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

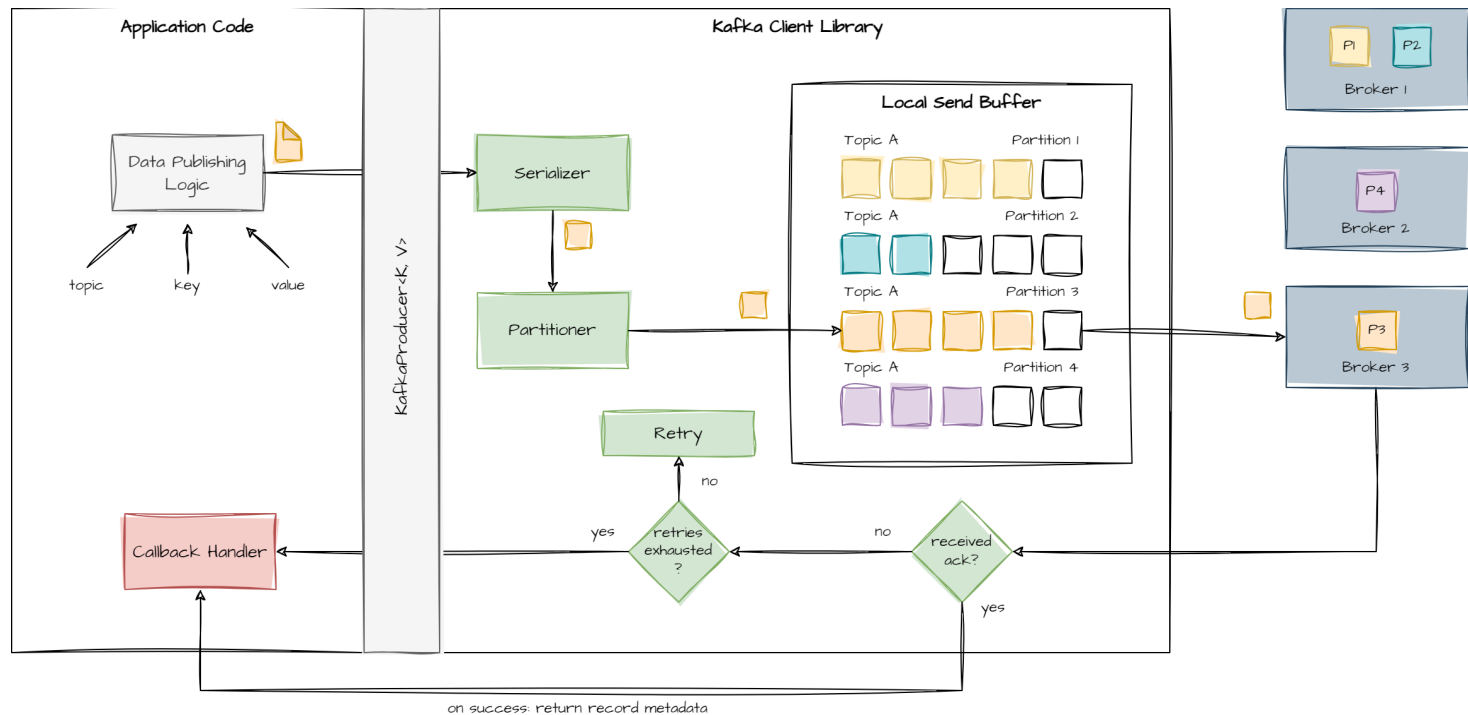
- don't switch unless `batch.size` bytes got produced to partition
- uniform throughput and data distribution
 - adapts well to higher latency brokers
- might slow down producer rate due to filling local buffer if brokers are constantly lagging behind

Adaptive Partition Switching (default)

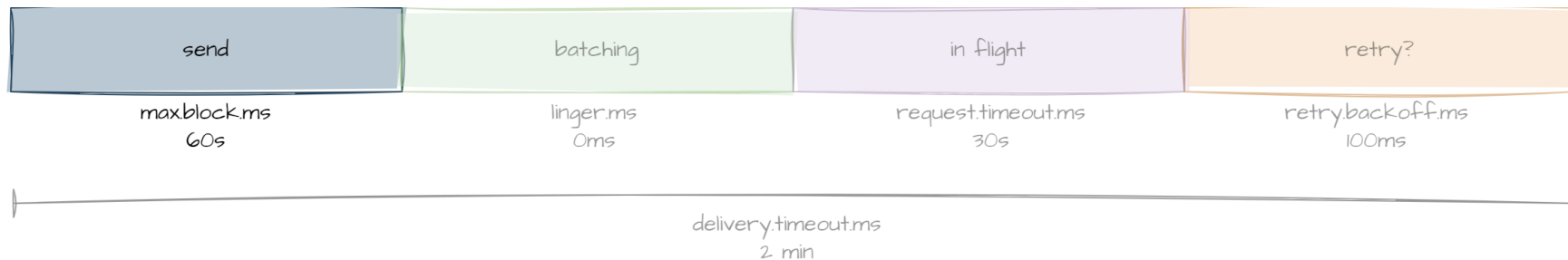
- adapts to broker load
 - queue size of unsent batches is indicator
 - probability of choosing a partition is proportional to the inverse queue size
 - partitions with longer queues are less likely to be chosen
- `partitioner.availability.timeout.ms` > 0 to indicate a failed batch if the producer is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

Timeouts

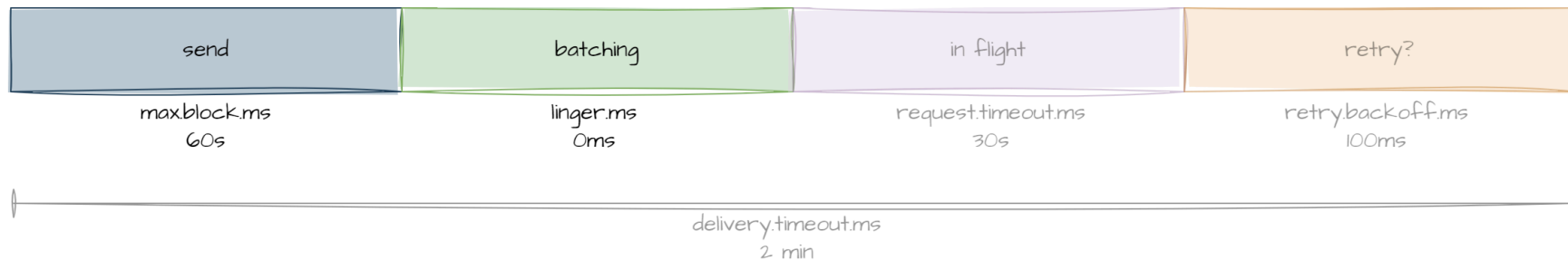
After partitioning, data is moved into a local buffer for the target topic-partition.



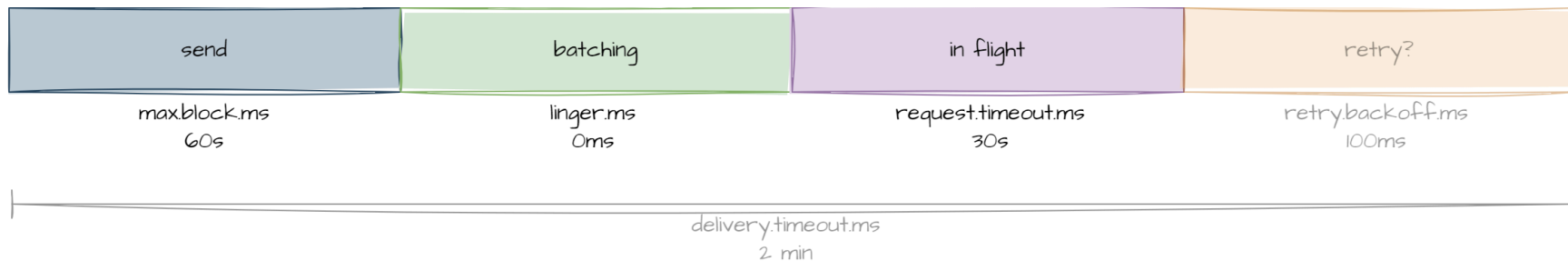
The send method of the producer waits a maximum of `max.block.ms`.



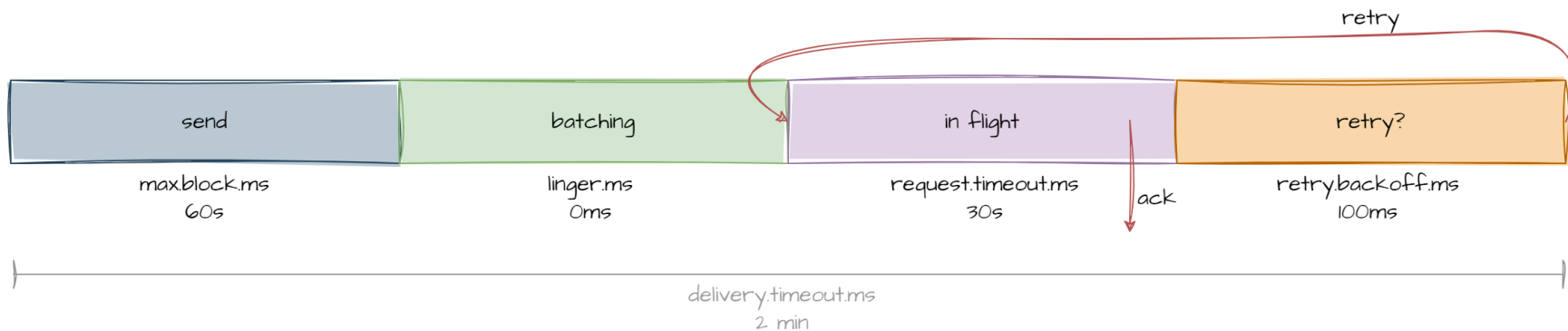
The producer waits a maximum of `linger.ms` for messages to include into a batch.



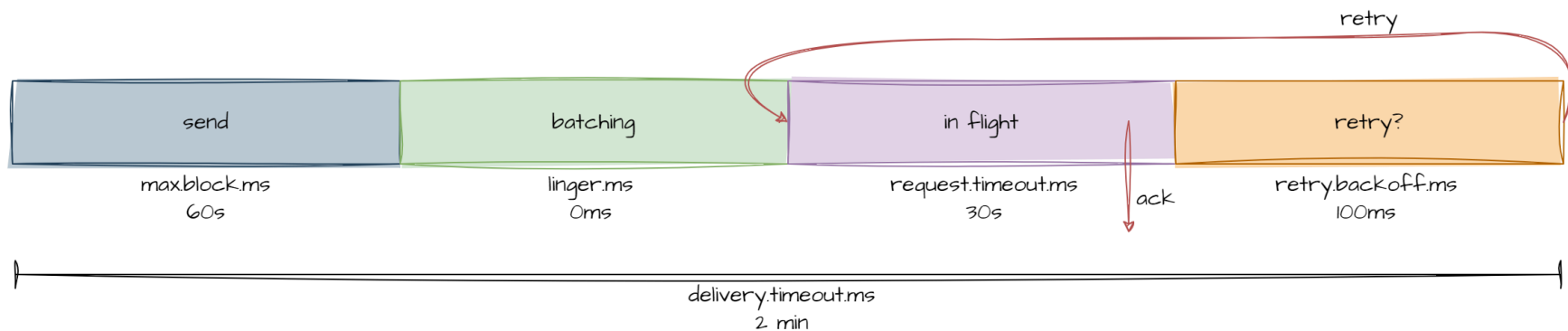
The producer waits for `request.timeout.ms` for an acknowledgement.



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**.*

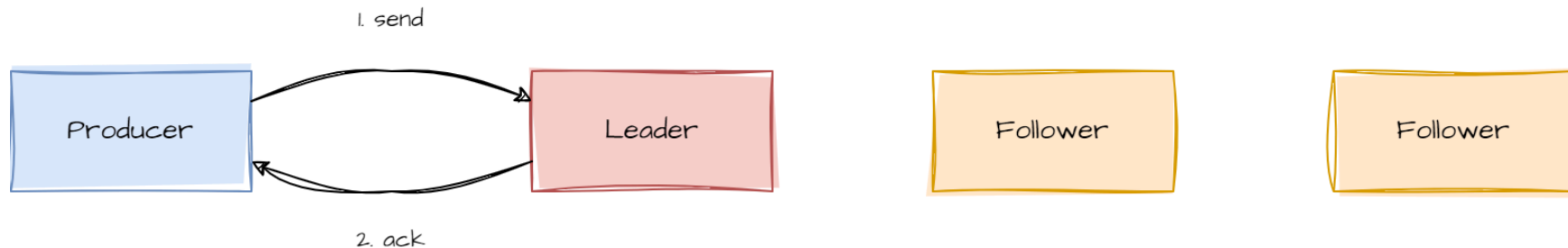
The producer does not wait for an acknowledgment if $acks=0$.



Traits

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

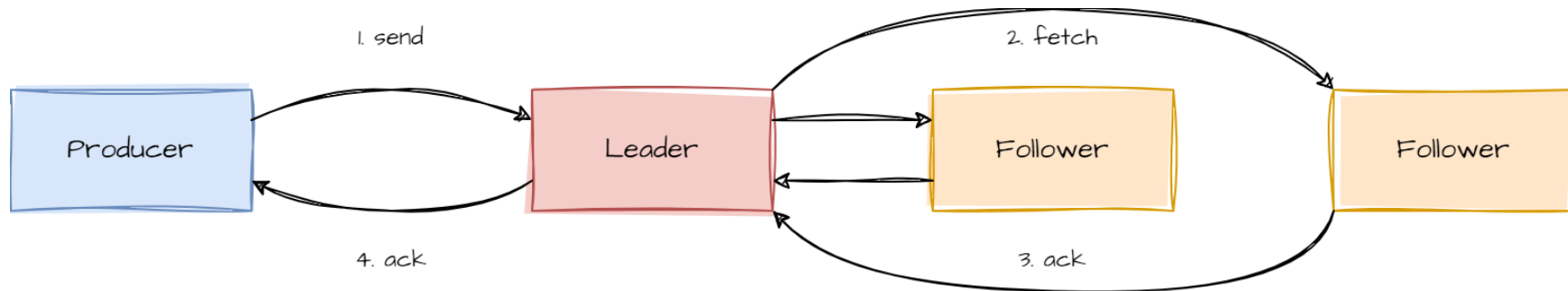
The leader acknowledges directly after receiving the record if $acks=1$.



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`.

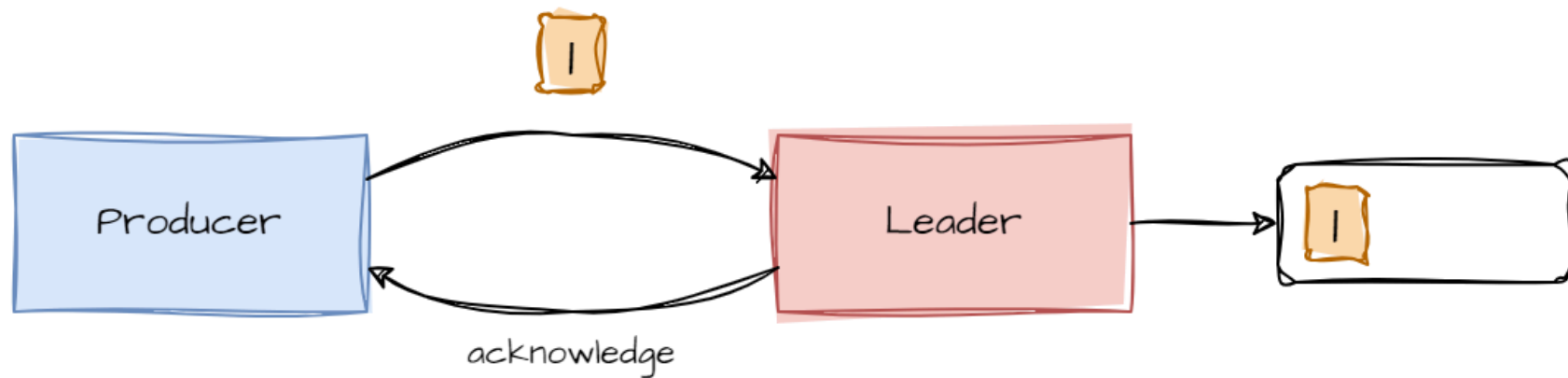


Traits

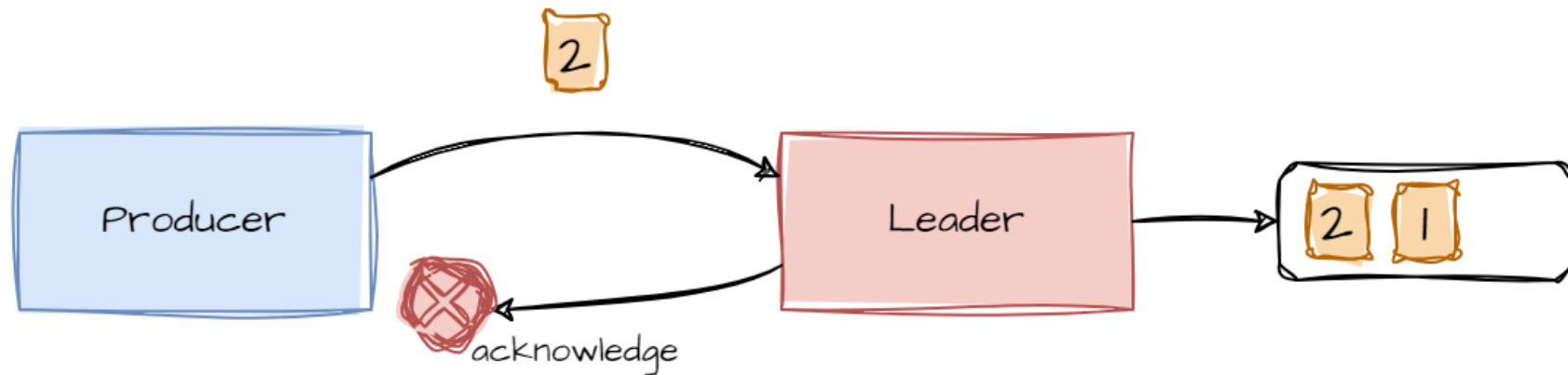
- Replicas are considered in-sync if they received latest data within 30 s
- Best consistency guarantees
- `min.insync.replicas` controls how many brokers must be in-sync

Delivery Guarantees

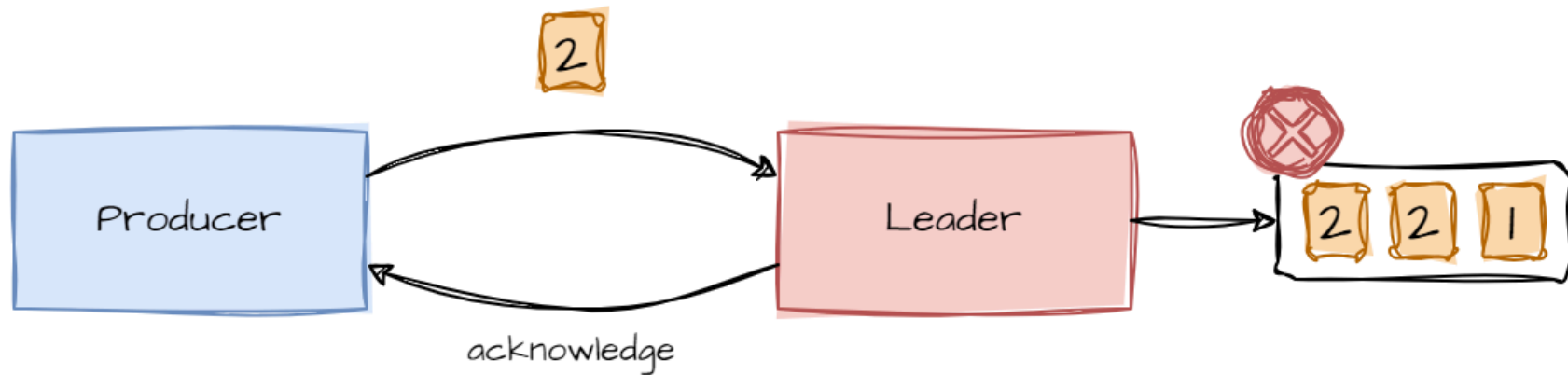
Lost acknowledgments may lead to duplicated records in the log.



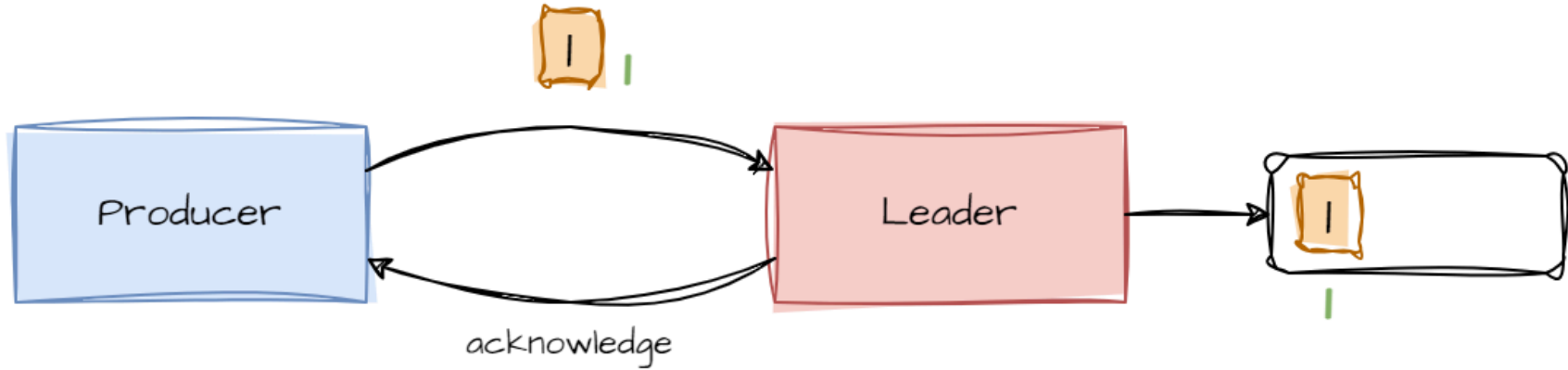
Lost acknowledgments may lead to duplicated records in the log. (cont.)



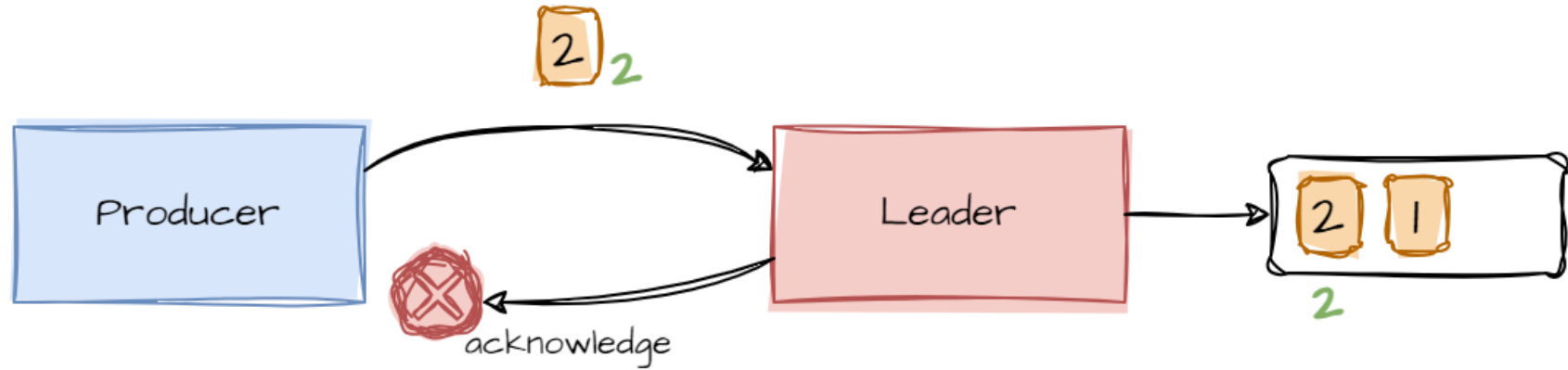
Lost acknowledgments may lead to duplicated records in the log. (cont.)



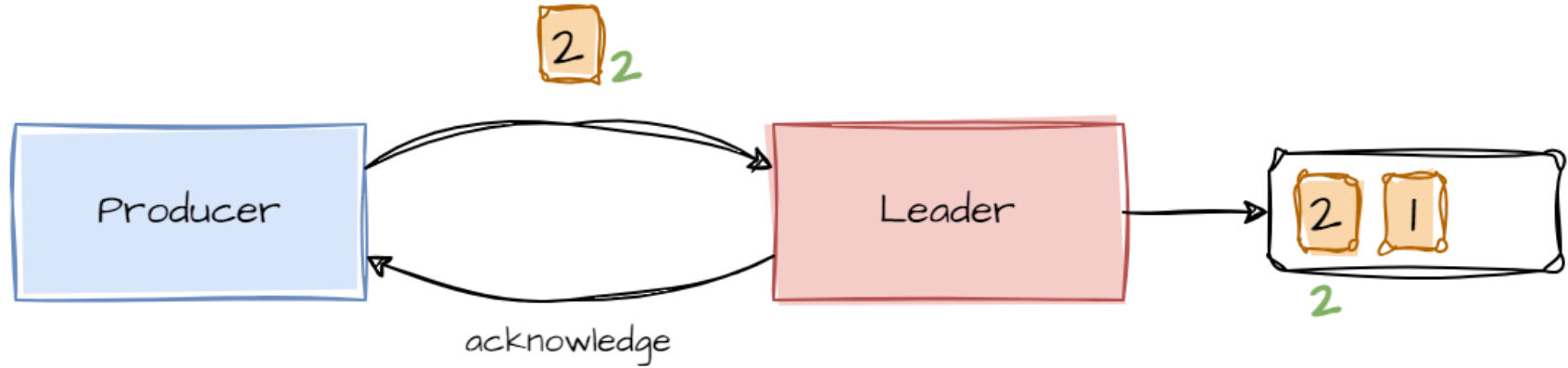
Setting `acks=all` and `enable.idempotence=true` prevents duplicates.



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



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



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

What did we learn?

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

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?