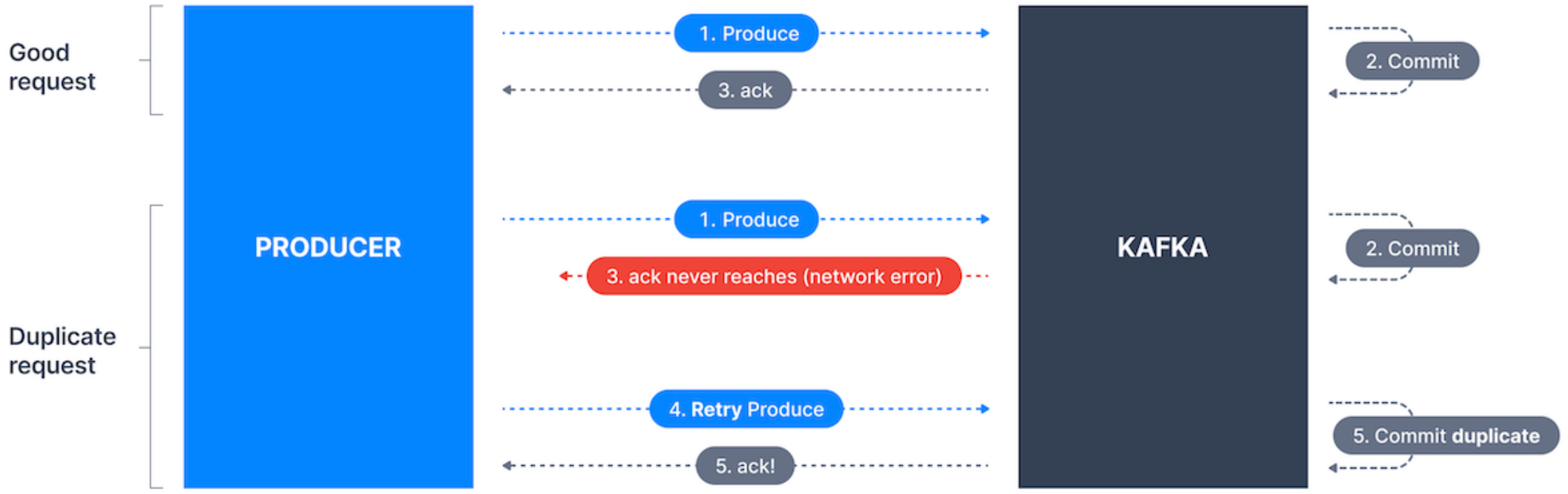
Kafka API Advance:

Idempotent Producer

& Message Delivery Semantics

Problem with Retries

Tos



Duplicate Messages

Problem with Retries

Tos

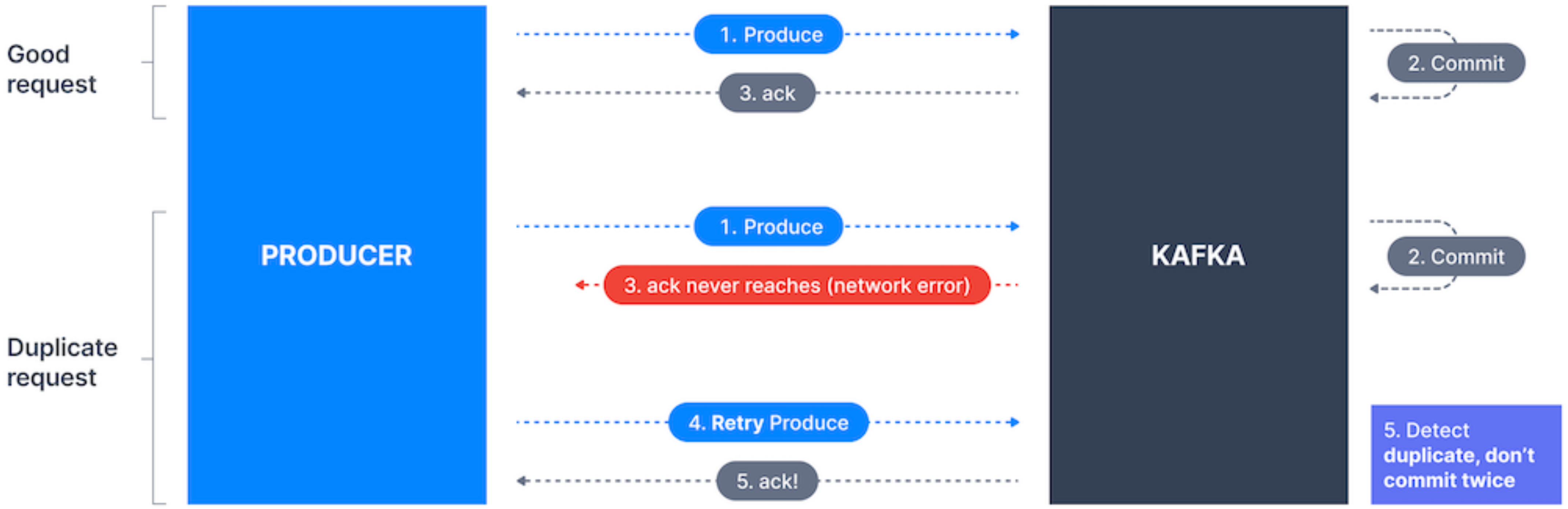
Retrying to send a failed message often includes a small risk that both messages were successfully written to the broker, leading to duplicates. This can happen as illustrated below.

- Kafka producer sends a message to Kafka
- The message was successfully written and replicated
- Network issues prevented the broker acknowledgment from reaching the producer
- The producer will treat the lack of acknowledgment as a temporary network issue and will retry sending the message (since it can’t know that it was received).
- In that case, the broker will end up having the same message twice.

idempotent producer

Tos

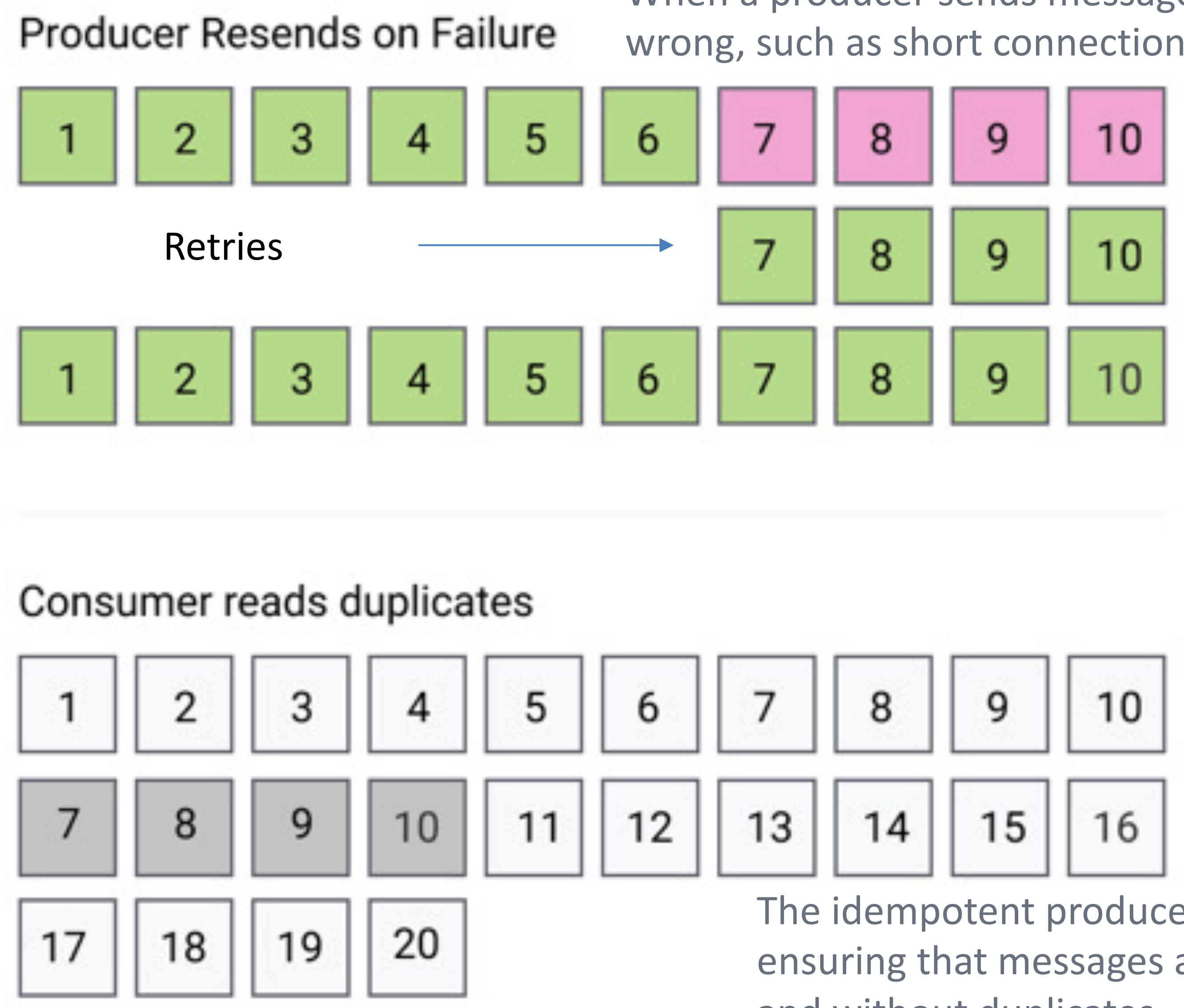
Producer idempotence ensures that duplicates are not introduced due to unexpected retries.



With Idempotent Enable

idempotent producer feature

When a producer sends messages to a topic, things can go wrong, such as short connection failures

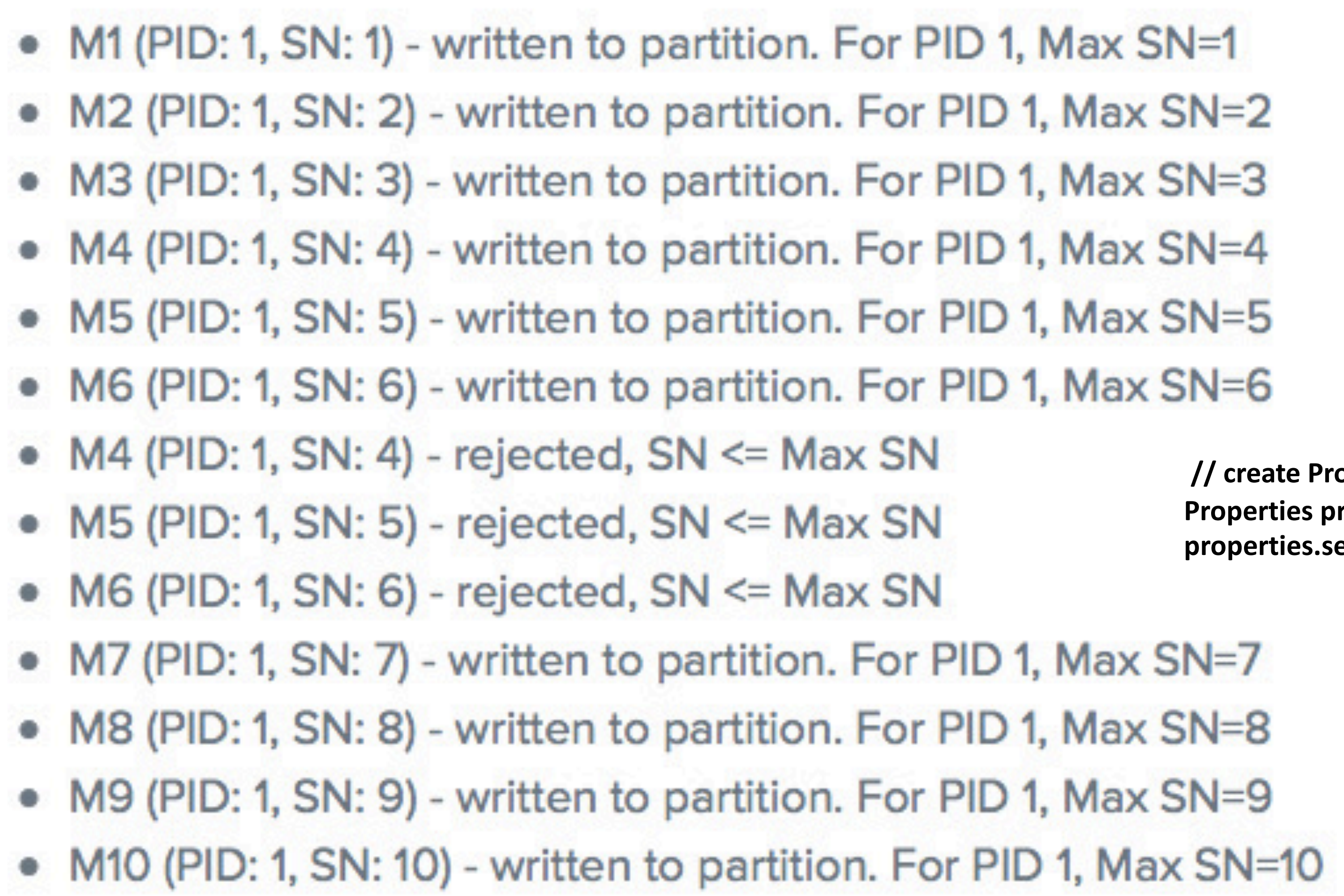


The idempotent producer feature addresses these issues ensuring that messages always get delivered, in the right order and without duplicates

The idempotent producer feature addresses these issues ensuring that messages always get delivered, in the right order and without duplicates.

How it Works

Tos



- M1 (PID: 1, SN: 1) - written to partition. For PID 1, Max SN=1
- M2 (PID: 1, SN: 2) - written to partition. For PID 1, Max SN=2
- M3 (PID: 1, SN: 3) - written to partition. For PID 1, Max SN=3
- M4 (PID: 1, SN: 4) - written to partition. For PID 1, Max SN=4
- M5 (PID: 1, SN: 5) - written to partition. For PID 1, Max SN=5
- M6 (PID: 1, SN: 6) - written to partition. For PID 1, Max SN=6
- M4 (PID: 1, SN: 4) - rejected, SN <= Max SN
- M5 (PID: 1, SN: 5) - rejected, SN <= Max SN
- M6 (PID: 1, SN: 6) - rejected, SN <= Max SN
- M7 (PID: 1, SN: 7) - written to partition. For PID 1, Max SN=7
- M8 (PID: 1, SN: 8) - written to partition. For PID 1, Max SN=8
- M9 (PID: 1, SN: 9) - written to partition. For PID 1, Max SN=9
- M10 (PID: 1, SN: 10) - written to partition. For PID 1, Max SN=10

```
 // create Producer properties
Properties properties = new Properties();
properties.setProperty(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, "true");
```

Limitation 1: Acks=All

Limitation 2: max.in.flight.requests.per.connection <= 5

How it Works

Tos

Each producer gets assigned a **Producer Id (PID)** and it includes its PID every time it sends messages to a broker.
Additionally, each message gets a monotonically increasing **sequence number.**
A separate sequence is maintained for each topic partition that a producer sends messages to.
On the broker side, on a per partition basis, it keeps track of the largest PID-Sequence Number combination is has successfully written.
When a lower sequence number is received, it is discarded

Exactly Once Delivery

Some applications require exactly-once semantics.
Each message is delivered exactly once.
This may be achieved in certain situations if Kafka and the consumer application cooperate to make exactly-once semantics happen.

Ways to Enable:

- This can only be achieved for Kafka topic to Kafka topic workflows using the transactions API. The Kafka Streams API simplifies the usage of that API and enables exactly once using the setting.
  processing.guarantee=exactly.once

- For Kafka topic to External System workflows, to *effectively* achieve exactly once, you must use an **idempotent** consumer.

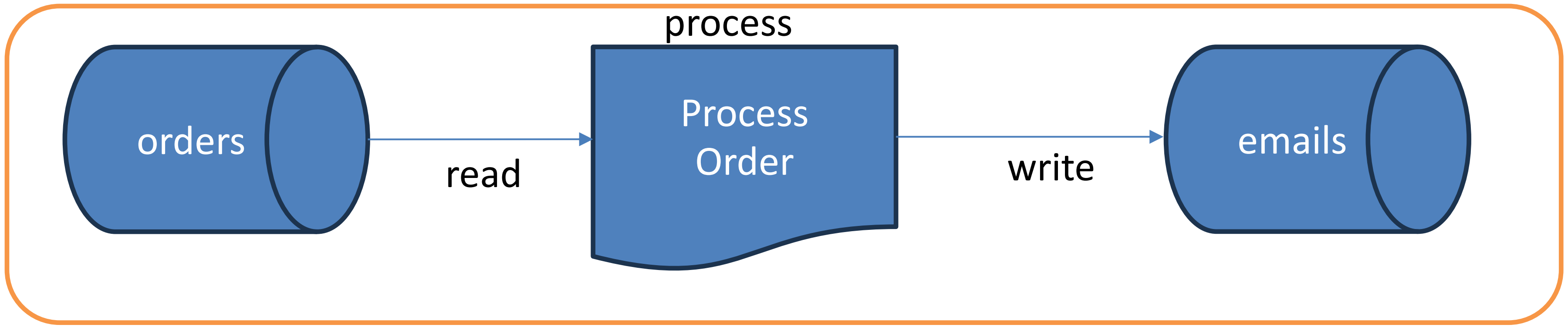
Transaction capabilities

Kafka offers an advanced mechanism for correlating the records consumed from input topics with the resulting records on output topics – by Transaction capabilities.

read-process-write cycle is atomic.

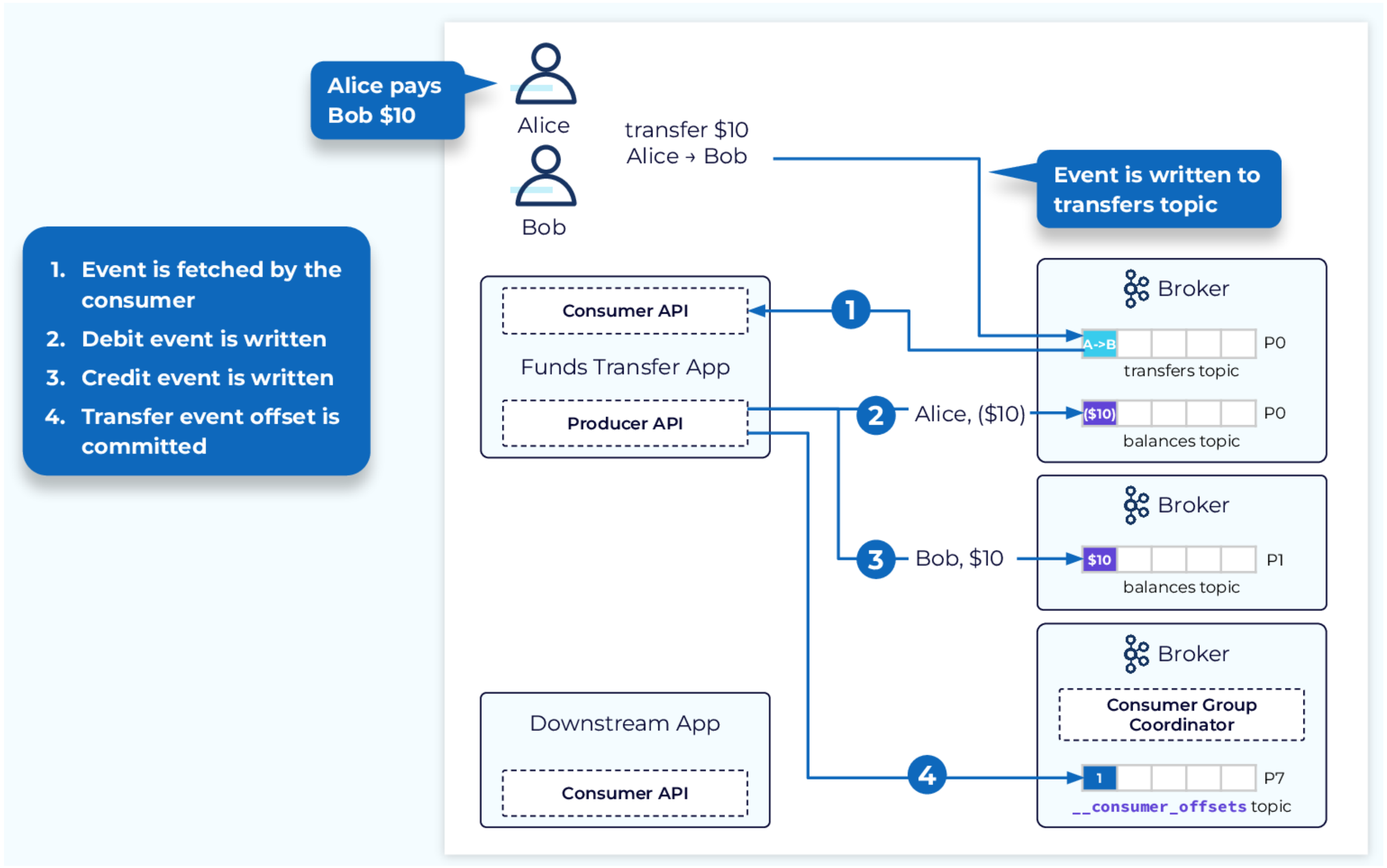
Transactions can create joint atomicity and isolation around the consumption and production of records, such that either all scoped actions appear to have occurred, or none.

Where the target endpoint is a (non-Kafka) message queue, the downstream receiver must be made idempotent.



Why Are Transactions Needed? (EOS)

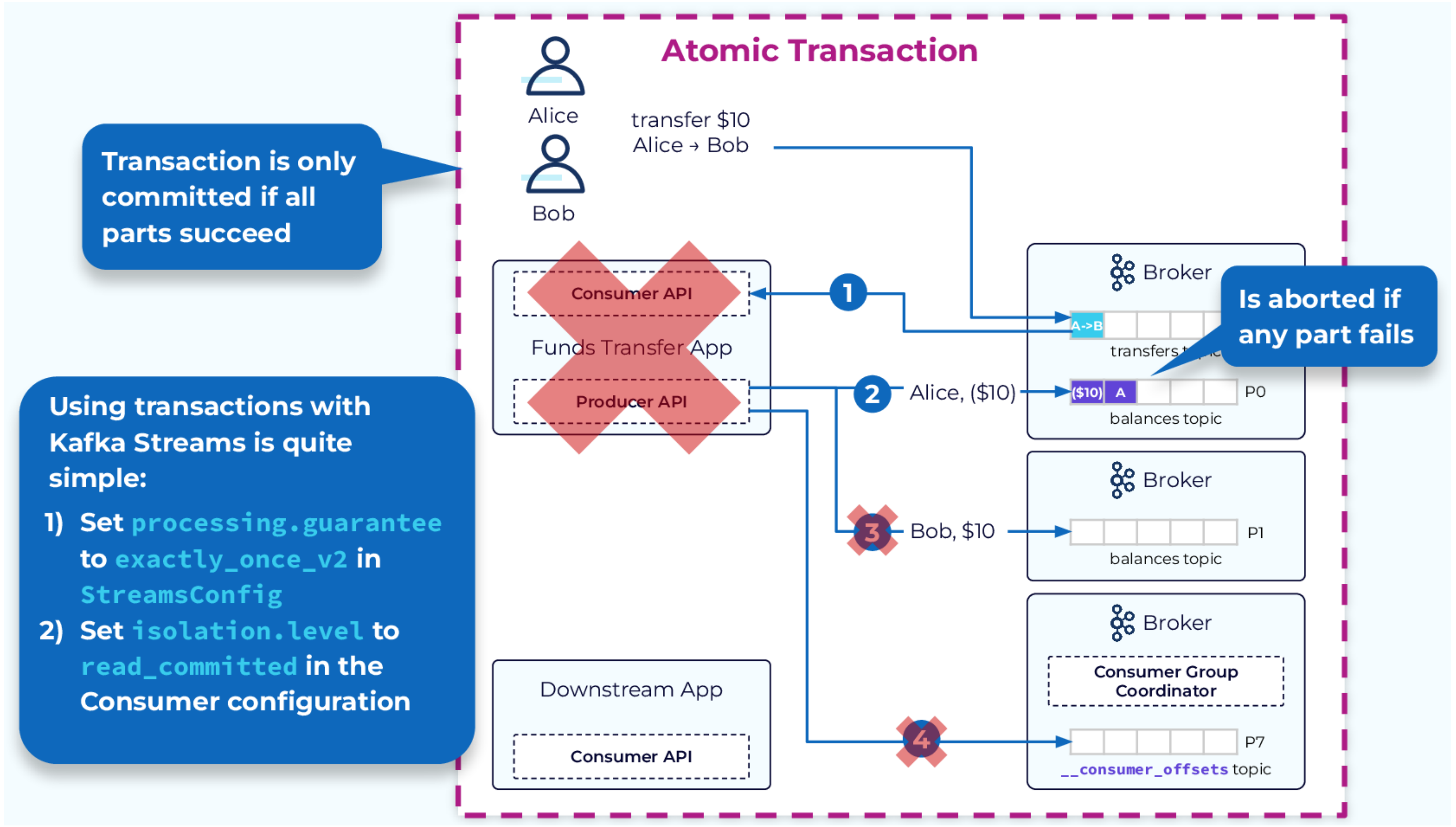
Tos



If a client application writes to multiple topics, or if multiple events are related, we may want them to all be written successfully or none.

Kafka Transactions Deliver Exactly Once

Tos



Transaction is only committed if all parts succeed

Is aborted if any part fails

Using transactions with Kafka Streams is quite simple:

1) Set `processing.guarantee` to `exactly_once_v2` in `StreamsConfig`
2) Set `isolation.level` to `read_committed` in the Consumer configuration

With transactions we can treat the entire **consume-transform-produc**e process topology as a single atomic transaction, which is only committed if all the steps in the topology succeed. If there is a failure at any point in the topology, the entire transaction is aborted. This will prevent duplicate records or more serious corruption of our data.

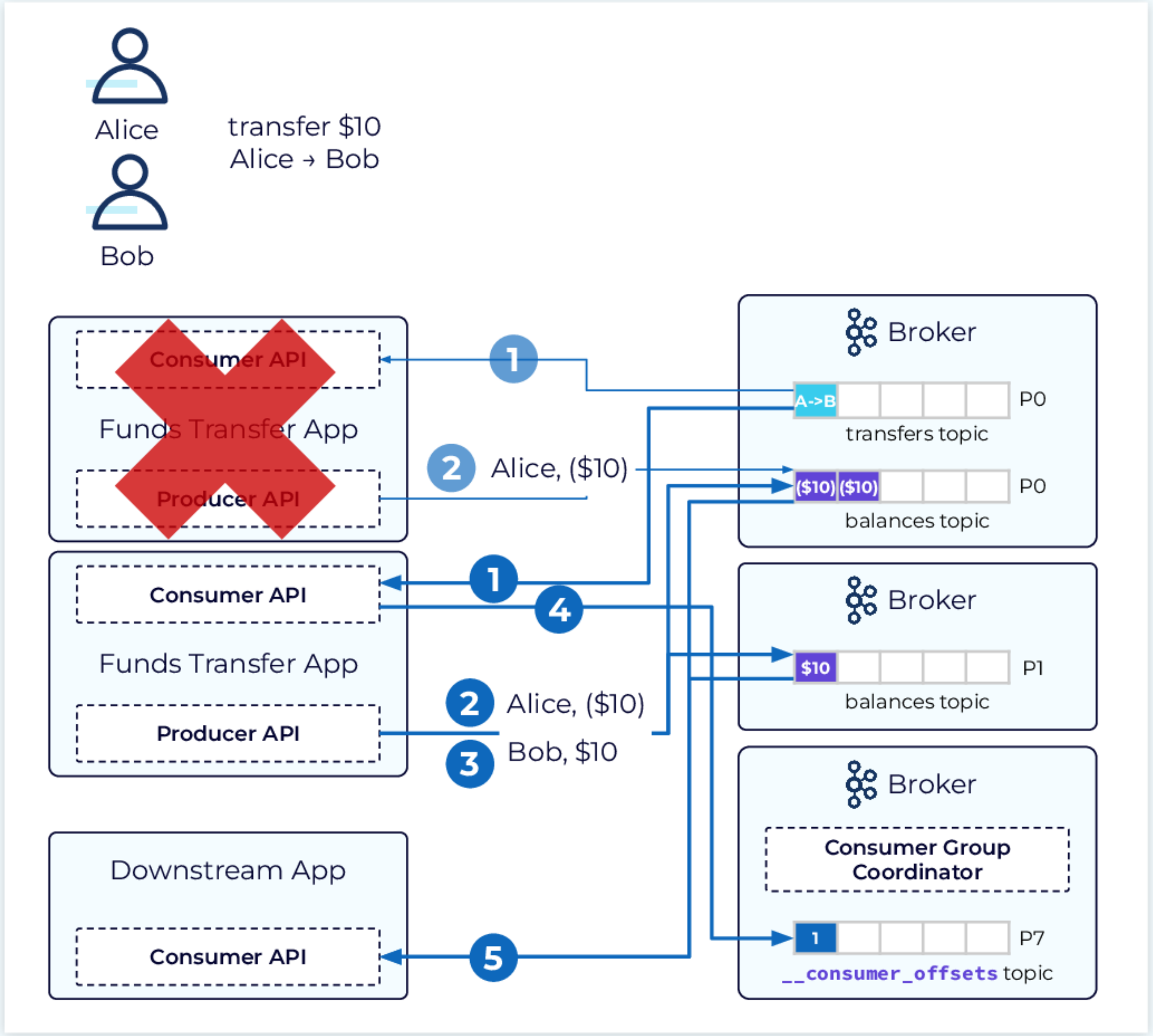
System Failure Without Transactions

Tos

1. Event fetched by consumer
2. Alice's account debited

Application instance fails without committing offset and new application instance starts

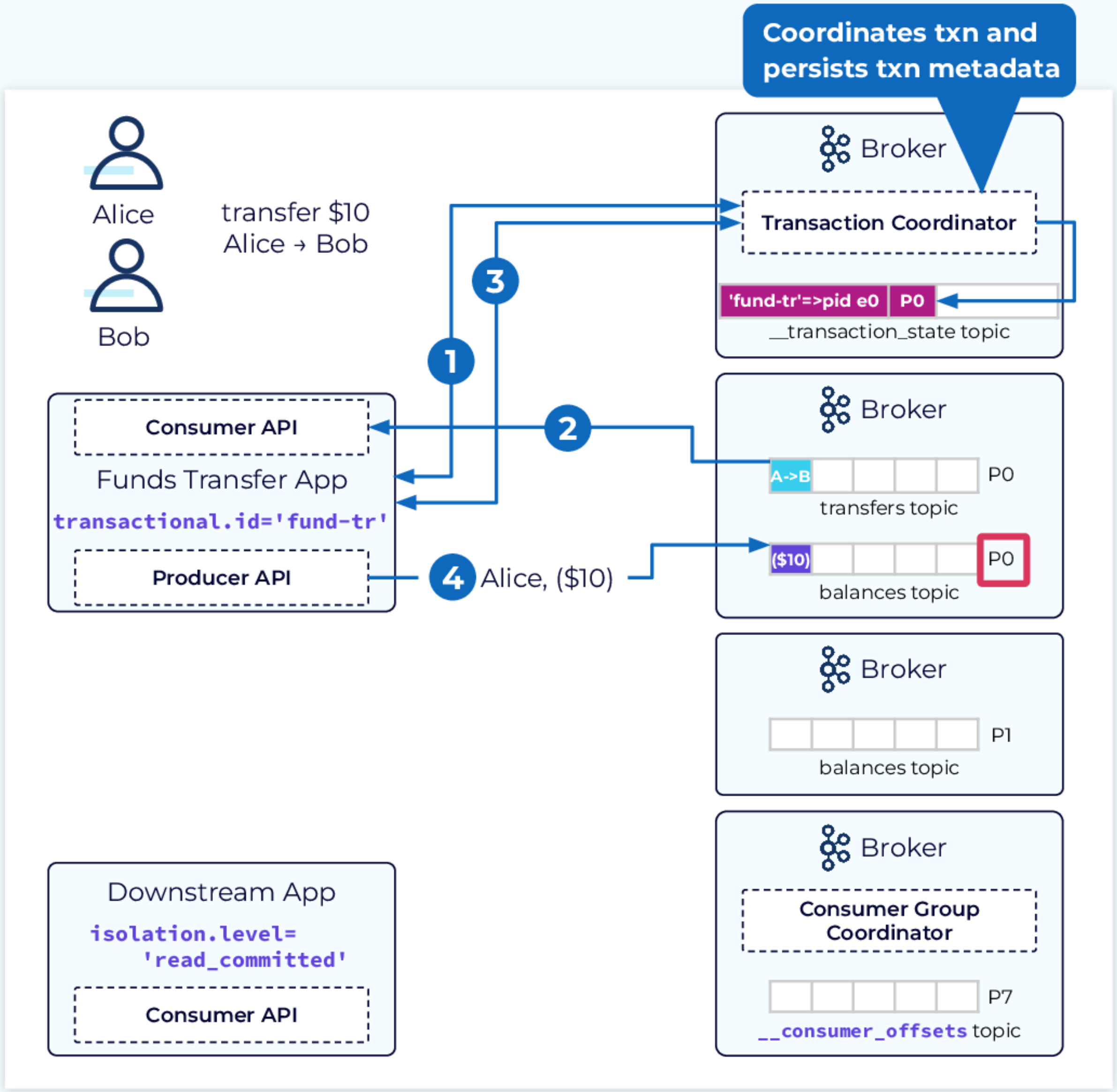
1. **Event fetched by consumer**
2. **Alice's account is debited a second time**
3. **Bob's account is credited**
4. **Consumer offset committed**
5. **Two debit events processed by downstream consumer**



System Failure with Transactions

Tos

1. **Requests txn ID, is returned PID and txn epoch**
2. **Event fetched by consumer**
3. **Notifies coordinator of partition being written to**
4. **Alice's account debited**



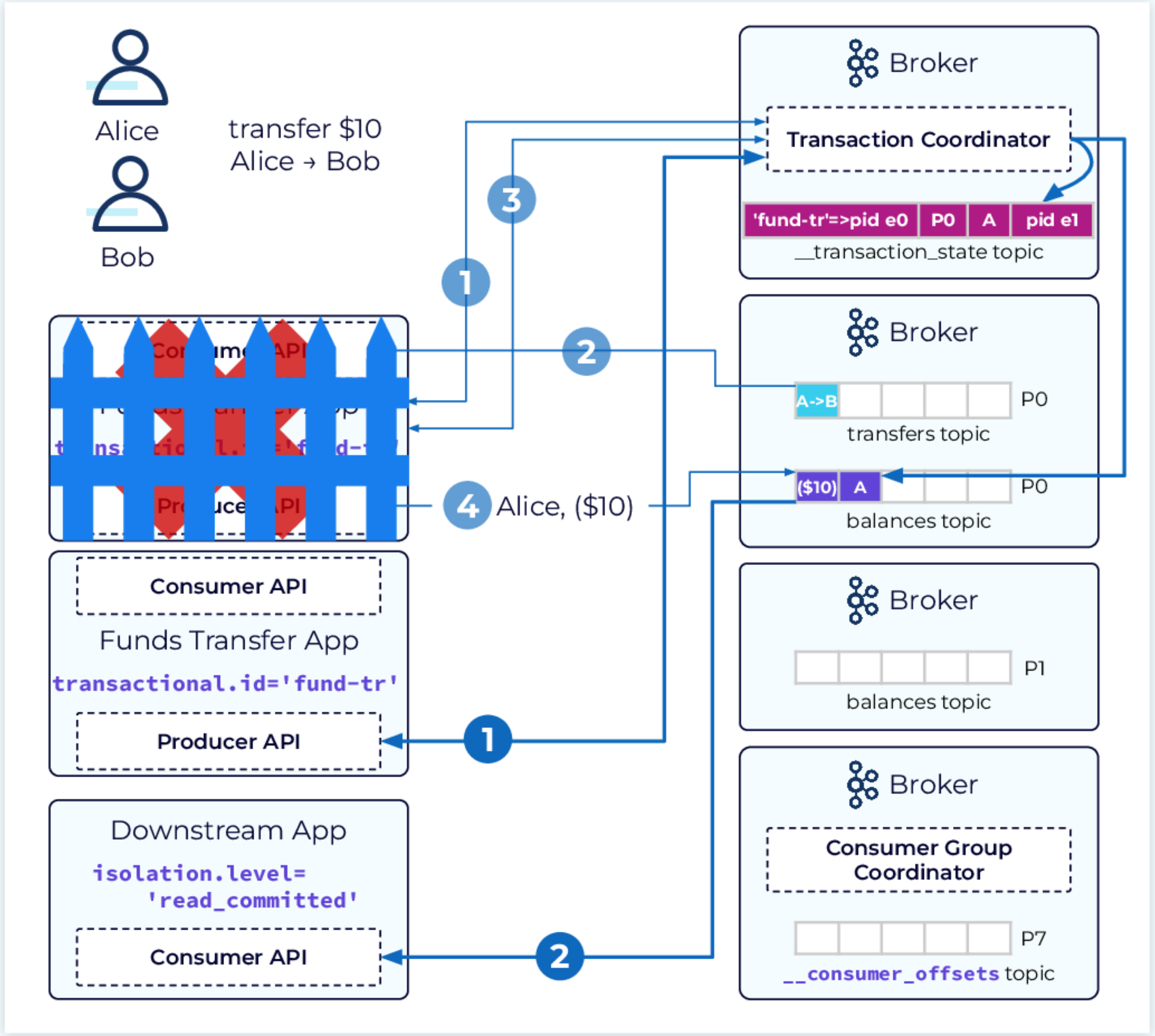
System Failure with Transactions …

Tos

1. Requests txn ID, is returned PID and txn epoch
2. Event fetched by consumer
3. Notifies coordinator of partition being written to
4. Alice's account debited

Application instance fails without committing offset and new application instance starts

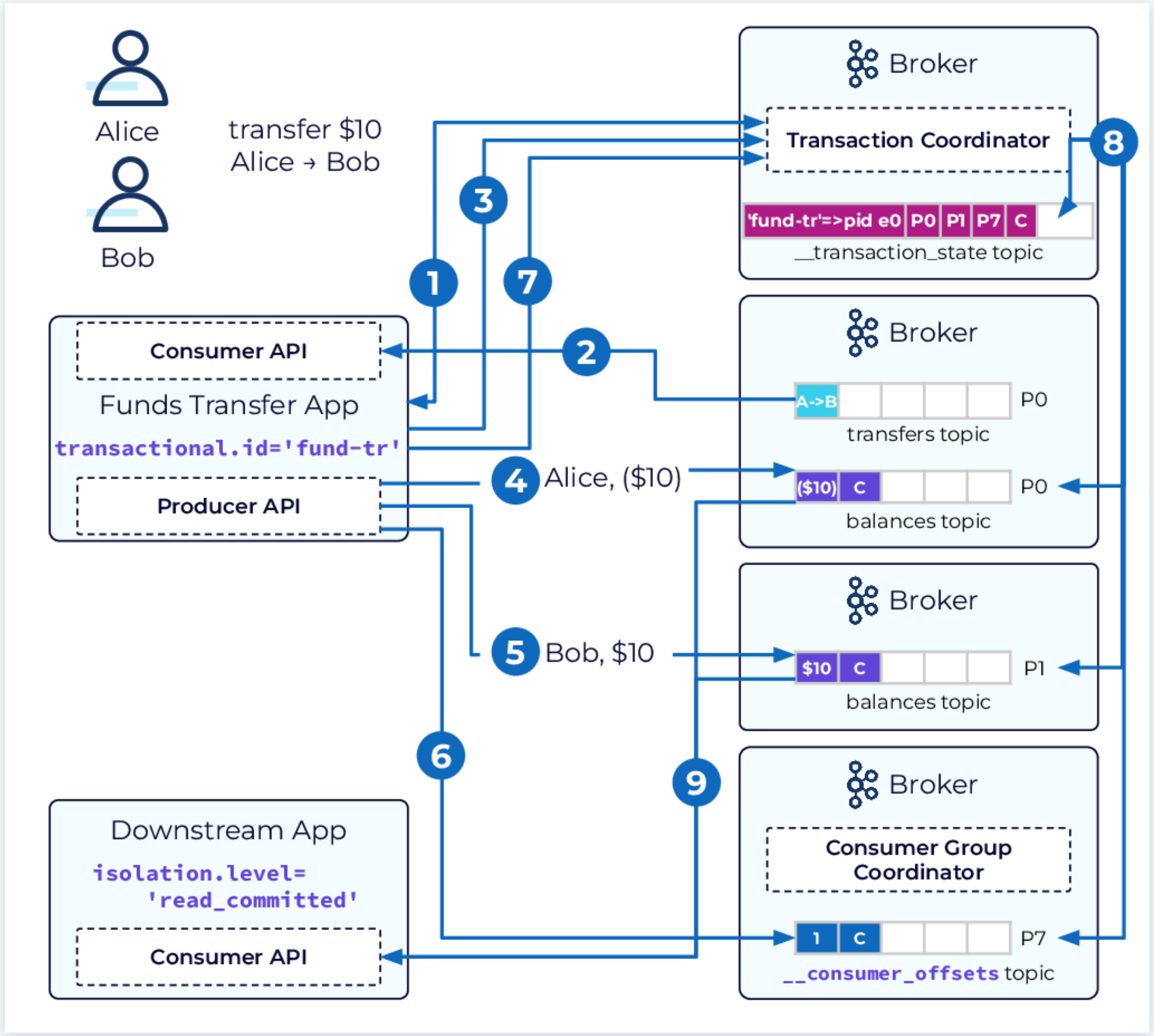
1. **New instance requests txn ID**
   a. **Coordinator fences previous instance by aborting pending txn and bumping up epoch**
2. **Downstream consumer with `read_committed` discards aborted events**



System with Successful Committed

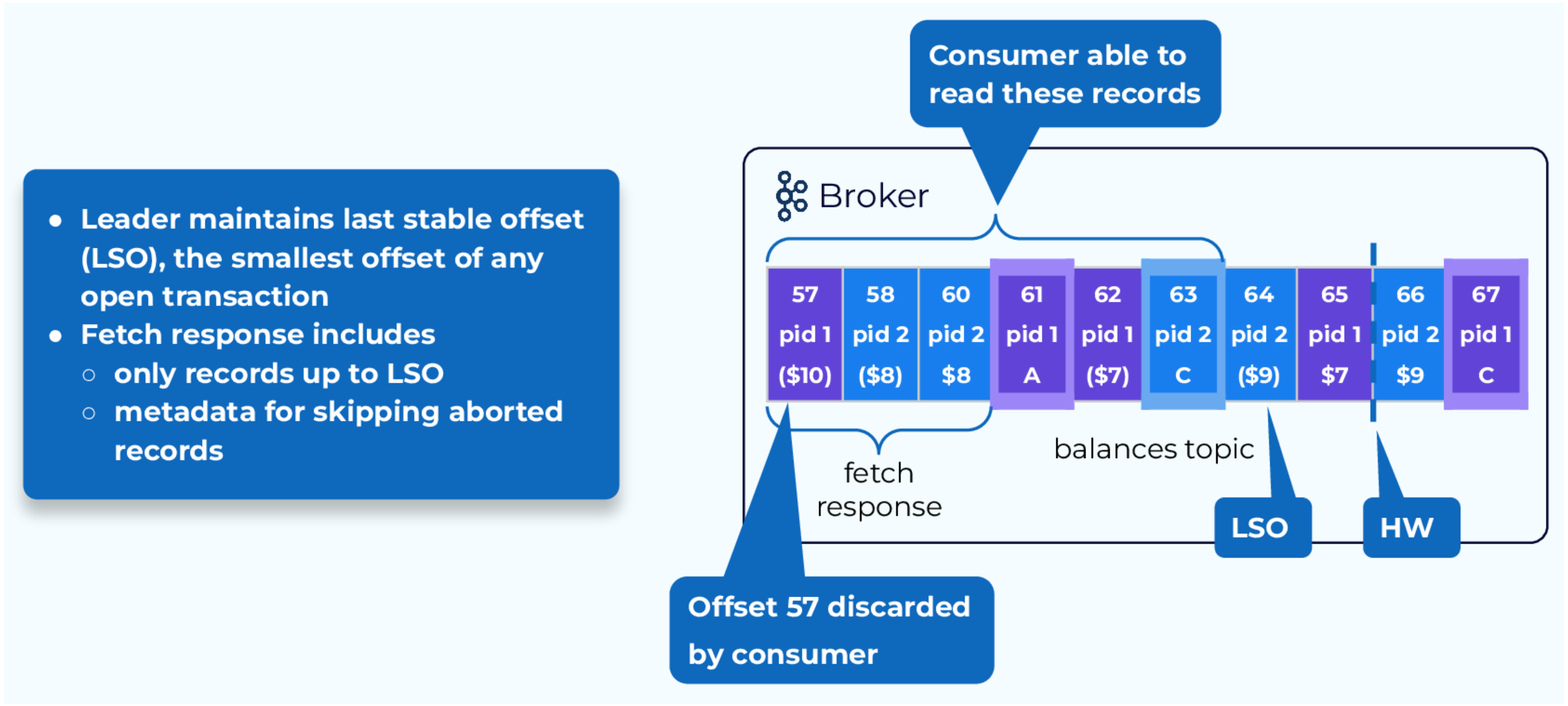
Tos

1. **Requests txn ID and assigned PID and epoch**
2. **Event fetched by consumer**
3. **Notifies coordinator of partition being written to**
4. **Alice's account debited**
5. **Bob's account credited**
6. **Consumer offset committed**
7. **Notify coordinator that transaction is complete**
8. **Coordinator writes commit markers to p0, p1, p7**
9. **Downstream consumer with `read_committed` processes committed events**



Consuming Transactions with read_committed

Tos



Consumer will receive events in offset order as usual, but it will only receive those events with an offset lower than the last stable offset (LSO). The LSO represents the lowest offset of any open pending transactions.

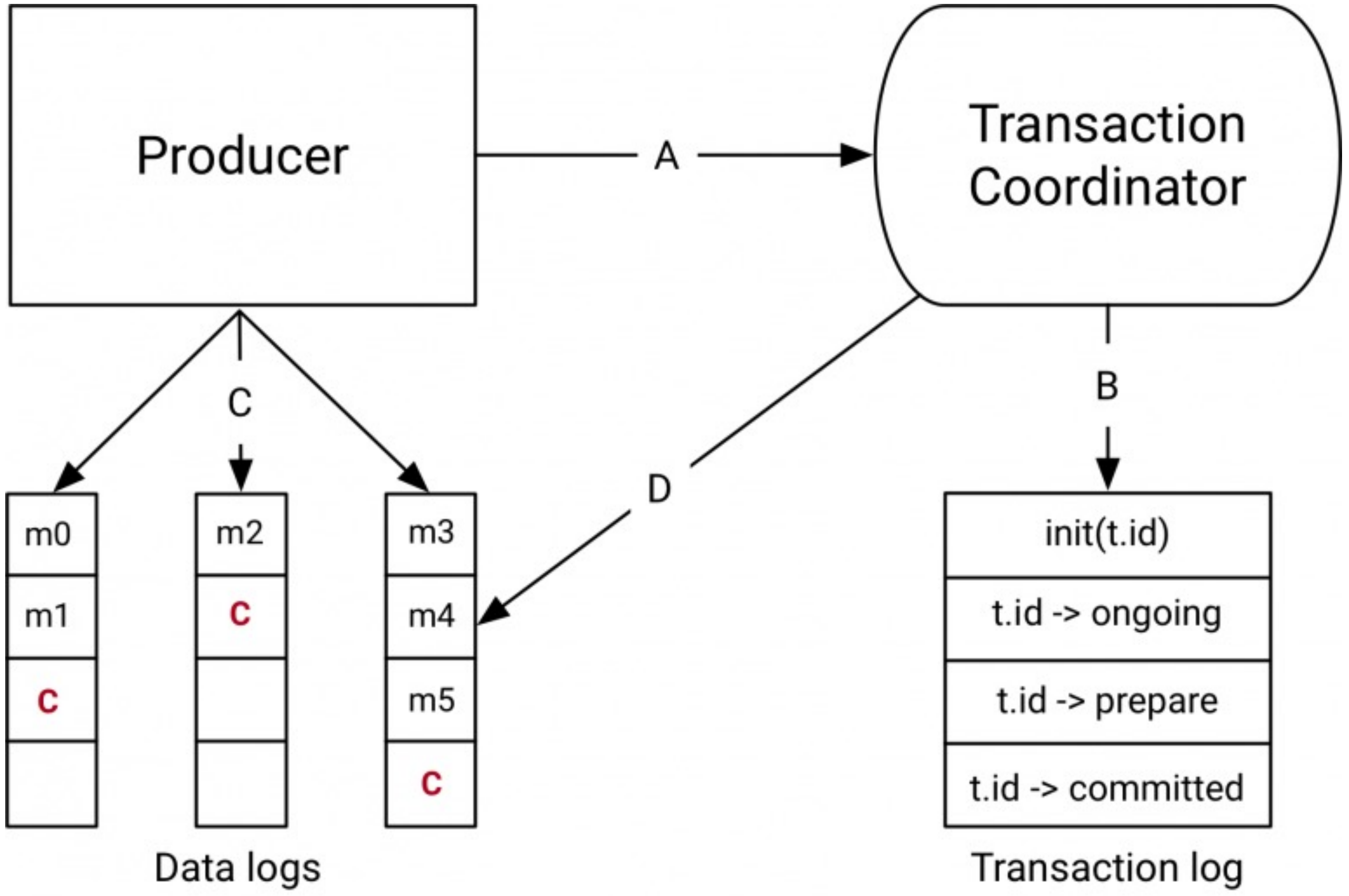
This means that only events from transactions that are either committed or aborted will be returned.
The fetch response will also include metadata to show the consumer which events have been aborted so that the consumer can discard them.

Transaction – On The Consumer Side

In order for this to work, consumers reading from transactional partitions should be configured to only read committed data.
This can be achieved by by setting the ***isolation.level=read_committed*** in the consumer's configuration.

- `read_committed`: In addition to reading messages that are not part of a transaction, you can also read ones that are, after the transaction is committed.
- `read_uncommitted`: Read all messages in offset order without waiting for transactions to be committed. This option is similar to the current semantics of a Kafka consumer.

Transaction simple design

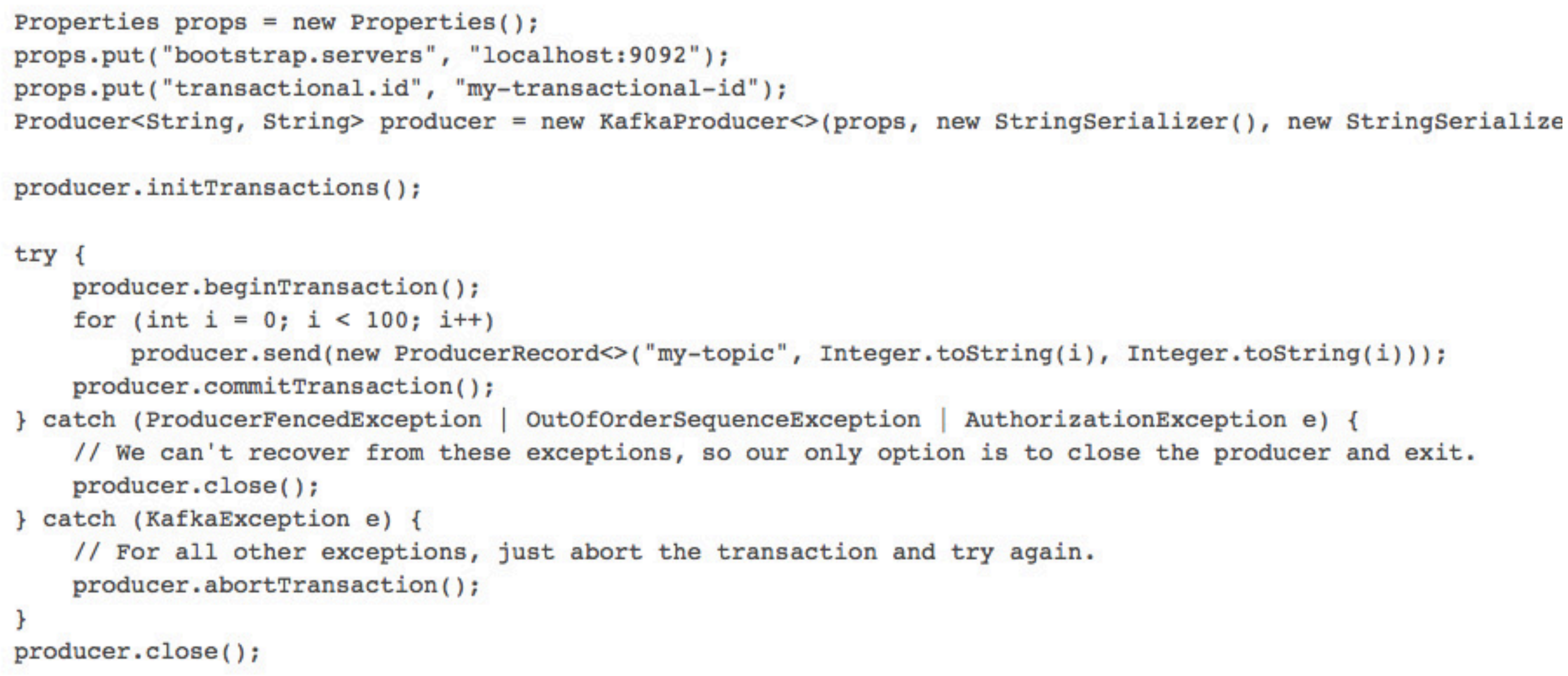


Transaction log is a Kafka topic and hence comes with the attendant durability guarantees.

Newly introduced *transaction coordinator* (which manages the transaction state per producer) runs within the broker and naturally leverages Kafka's leader election algorithm to handle failover.

For stream processing applications built using Kafka's Streams API, it leverages the fact that the source of truth for the state store and the input offsets are Kafka topics.

transactional producer

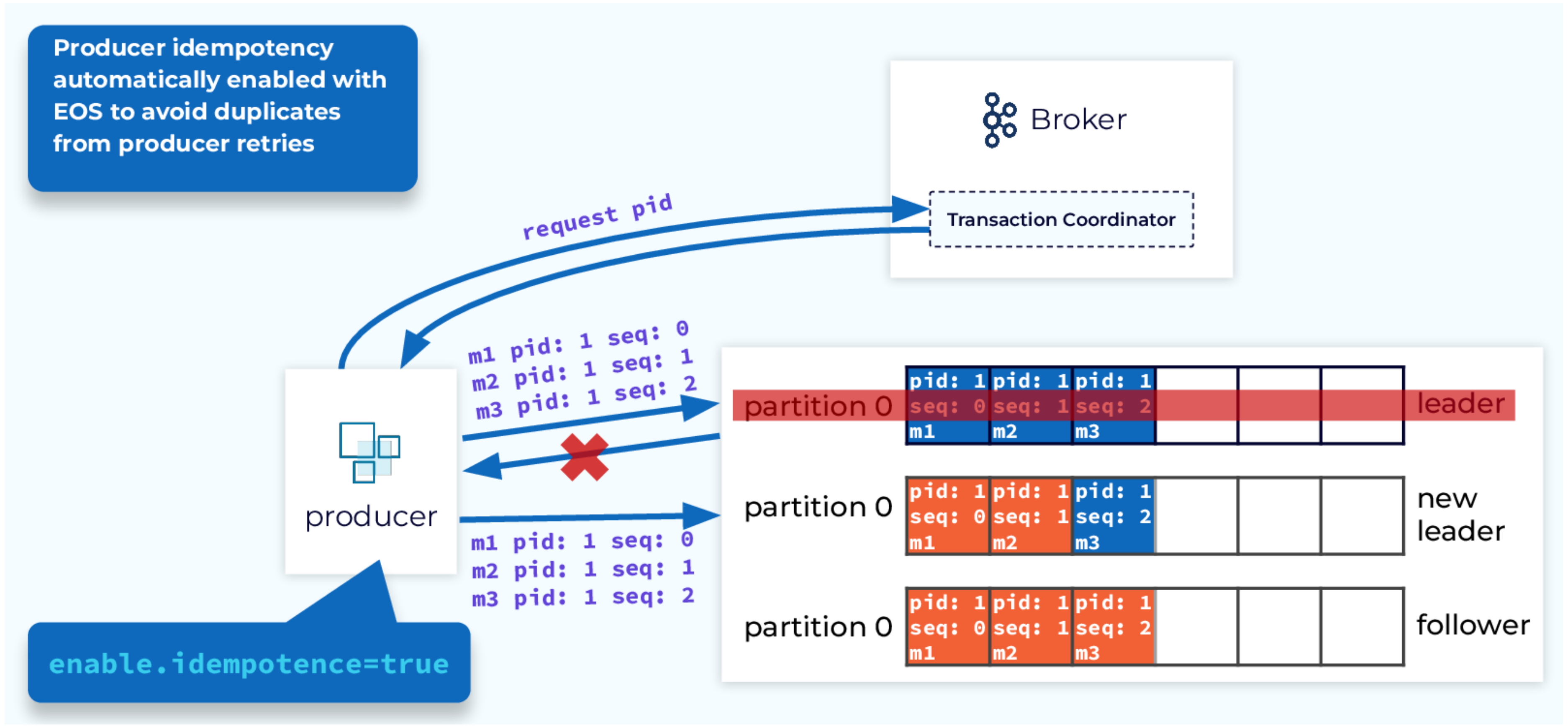


```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("transactional.id", "my-transactional-id");
Producer<String, String> producer = new KafkaProducer<>(props, new StringSerializer(), new StringSerialize

producer.initTransactions();

try {
    producer.beginTransaction();
    for (int i = 0; i < 100; i++)
        producer.send(new ProducerRecord<>("my-topic", Integer.toString(i), Integer.toString(i)));
    producer.commitTransaction();
} catch (ProducerFencedException | OutOfOrderSequenceException | AuthorizationException e) {
    // We can't recover from these exceptions, so our only option is to close the producer and exit.
    producer.close();
} catch (KafkaException e) {
    // For all other exceptions, just abort the transaction and try again.
    producer.abortTransaction();
}
producer.close();
```

Transactions: Producer Idempotency



Producer idempotency, which we talked about earlier, is critical to the success of transactions, so when transactions are enabled, idempotence is also enabled automatically.

Transactions: Balance Overhead with Latency

Tos

One thing to consider, specifically in Kafka Streams applications, is how to set the **commit.interval.msconfiguration**.
This will determine how frequently to commit, and hence the size of our transactions. There is a bit of overhead for each transaction so many smaller transactions could cause performance issues.

However, long-running transactions will delay the availability of output, resulting in increased latency. Different applications will have different needs, so this should be considered and adjusted accordingly.

Kafka Producer: Buffering and batching

Tos

- Kafka Producer buffers are available to send immediately as fast as broker can keep up (limited by inflight ***max.in.flight.requests.per.connection***)
- To reduce requests count, set ***linger.ms*** > 0
  - wait up to **linger.ms** before sending or until batch fills up whichever comes first
  - Under heavy load **linger.ms** not met, under light producer load used to increase broker IO throughput and increase compression
- ***buffer.memory*** controls total memory available to producer for buffering
  - period blocks (**max.block.ms**) after then Producer throws a **TimeoutException**

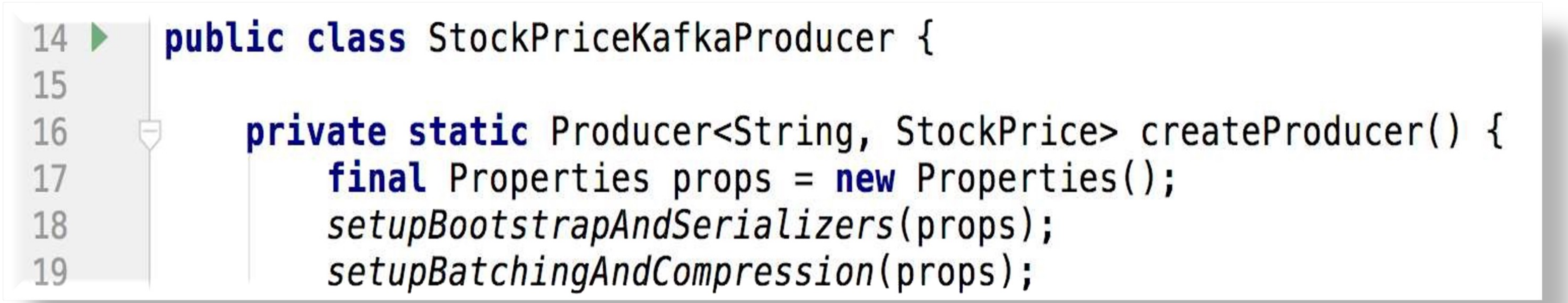
Producer Buffer Memory Size

Tos

- Producer config property: ***buffer.memory***
    - ***default 32MB***
- Total memory (bytes) producer can use to buffer records to be sent to broker
- Producer blocks up to ***max.block.ms*** if ***buffer.memory*** is exceeded
    - if it is sending faster than the broker can receive, exception is thrown

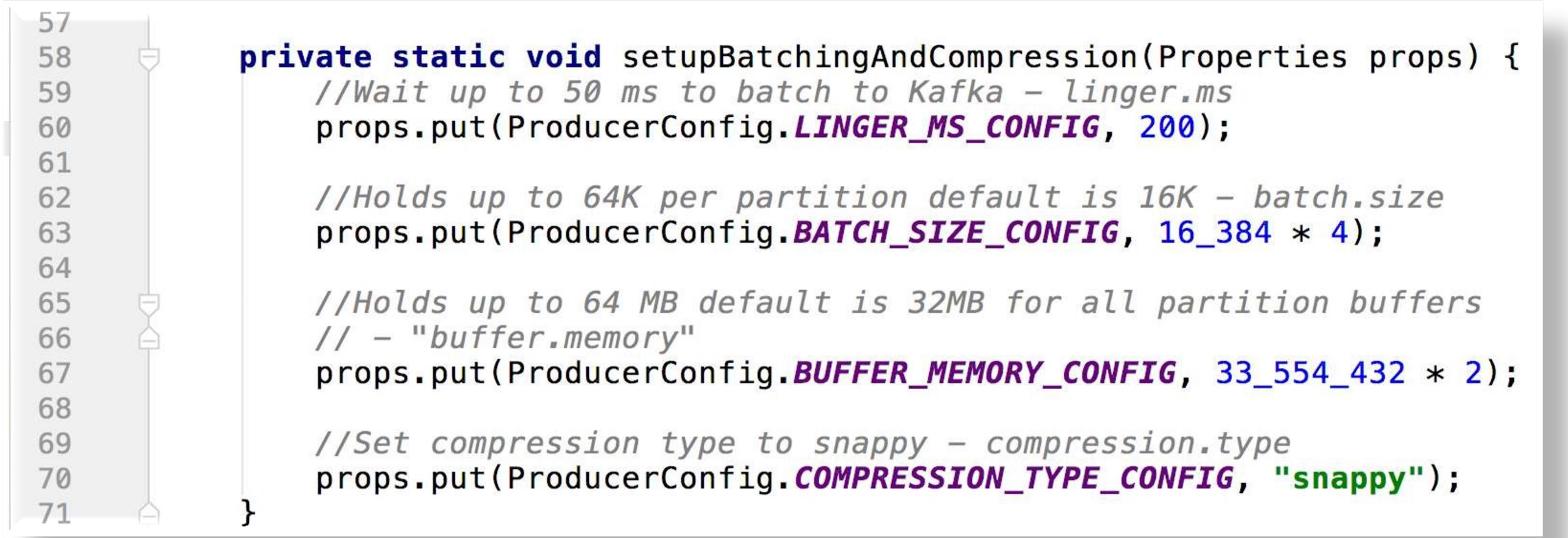
Batching and Compression

Tos



```java
public class StockPriceKafkaProducer {

    private static Producer<String, StockPrice> createProducer() {
        final Properties props = new Properties();
        setupBootstrapAndSerializers(props);
        setupBatchingAndCompression(props);
```



```java

    private static void setupBatchingAndCompression(Properties props) {
        //Wait up to 50 ms to batch to Kafka - linger.ms
        props.put(ProducerConfig.LINGER_MS_CONFIG, 200);

        //Holds up to 64K per partition default is 16K - batch.size
        props.put(ProducerConfig.BATCH_SIZE_CONFIG, 16_384 * 4);

        //Holds up to 64 MB default is 32MB for all partition buffers
        // - "buffer.memory"
        props.put(ProducerConfig.BUFFER_MEMORY_CONFIG, 33_554_432 * 2);

        //Set compression type to snappy - compression.type
        props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "snappy");
    }
```