

EXACTLY ONCE (доступно только в java)

15 декабря 2020 г. 15:06

strong consistency - кафка обеспечивает сильную согласованность

- at this time EOS is not supported in librdkafka based clients (теперь работает только для java-клиентов)
- для работы EOS продьюсер должен установить настройку `acks=all`

EOS - Motivation for Exactly Once Semantics

*Writing real-time, mission-critical streaming applications, requiring **exactly once** processing guarantees.*

Use Cases:

- Financial transactions
- Tracking ad views
- Kafka Streams & KSQL apps

Supported Clients

- Producer and Consumer (Java only)
- Kafka Streams API
- Confluent KSQL

- Write real-time, mission-critical streaming applications that require guarantees that data is processed “exactly once”
 - Complements the “at most once” or “at least once” semantics that exist today
- Exactly Once Semantics (EOS) bring **strong transactional guarantees** to Kafka
 - Prevents 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**



For EOS to work, the producer must be set with `acks=all`. Through the use of header fields and other improvements, EOS can take the "at least once" guarantee of this acks setting and dedupe messages as they pass through Kafka.



At this time EOS is **not** supported in `librdkafka` based clients. Only idempotent producers are supported.



This section is only a high-level overview of EOS. For a more complete discussion, refer to the online talk and the original KIP: <https://cncf.io/EOS-KIP-98>

(1) `enable.idempotence=true` на продьюсере включить идемпотентность

- и также `acks=all`

- нам больше не нужно беспокоиться о том, чтобы в отправляемых нами сообщениях были соответствующие уникальные ключи
- Идемпотентность в этом контексте - это просто **дедупликация**. Каждому производителю дается **идентификатор, и каждому сообщению дается порядковый номер**. Их сочетание однозначно определяет каждый пакет отправленных сообщений. Кафка-нода использует этот уникальный порядковый номер, чтобы определить, есть ли сообщение в журнале, и отбрасывает его, если оно есть. Это значительно более эффективный подход, чем хранение каждого ключа, который вы когда-либо видели в базе данных.

1

Use **idempotent** producer(s)

`enable.idempotence=true`

- Configure the Producer send operation to be **idempotent**
 - An idempotent operation is one which can be performed many times without causing a different effect than only being performed once
 - Removes the possibility of duplicate messages being delivered to a particular Topic Partition due to Producer or Broker errors
 - Set `enable.idempotence` to **true** (Default: **false**)

если использую транзакции то callback ненужен (тк об ошибке я и так узнаю)
успешный коммит и отсутствие ошибки будет означать успешную доставку сообщения

When using transactions, it's not necessary to register a callback to be notified of failed acks - there will be an exception thrown in that case

как альтернативный вариант: помнить все идентификаторы сообщений на обоих концах (и тогда пофиг на все промежуточные звенья)

- So it is actually the use case itself that is important. So long as deduplication happens at the start and end of each use case, it doesn't matter how many duplicate calls are made in between. This is an old idea, dating back to the early days of TCP. It's called the End-to-End Principle.
- в результате большинство систем, управляемых событиями, в конечном итоге дедуплицируют каждое полученное сообщение перед его обработкой, и каждое отправленное сообщение имеет тщательно выбранный идентификатор, чтобы его можно было дедуплицировать в исходящем направлении. В лучшем случае это немного хлопотно. В худшем случае это питательная среда для ошибок.
- Люди десятилетиями создавали системы, управляемые как событиями, так и запросами, просто делая свои процессы идемпотентными с помощью идентификаторов и баз данных.
- На самом деле в большинстве создаваемых нами систем мы решаем эти проблемы с дублированием автоматически, поскольку многие системы просто отправляют данные в базу данных, которая автоматически дедуплицируется на основе первичного ключа. Такие процессы естественно идемпотентны. Поэтому, если клиент обновляет свой адрес, и мы сохраняем эти данные в базе данных, нам все равно, создадим ли мы дубликаты, поскольку в худшем случае таблица базы данных, содержащая адреса клиентов, обновляется дважды, что не является проблемой. Это относится и к примеру оплаты, если у каждого из них есть уникальный идентификатор. Пока дедупликация происходит в конце каждого варианта использования, не имеет значения, сколько повторяющихся вызовов выполняется между ними. Это старая идея, восходящая к ранним дням TCP (протокол управления передачей). Это называется end-to-end principle .

(2) продьюсер должен использовать transaction API для отправки сообщений (методы initTransaction, beginTransaction, sendOffsetsToTransaction, commitTransaction)

- транзакции нужны на тот случай если сама нода кафки упадет, чтобы сообщения в итоге дошли в правильном порядке
- транзакции сразу же снимают проблему идемпотентности с сервисов, связанных с Kafka. Поэтому, когда мы создаем службы, которые следуют шаблону: чтение, обработка, (сохранение), отправка, нам не нужно беспокоиться о дедупликации входных данных или создании ключей для выходных данных.

2
Write **atomically** across multiple partitions

use Transaction API

```

1 producer.initTransactions();
2 try {
3     producer.beginTransaction();
4     producer.send(record1);
5     producer.send(record2);
6     producer.sendOffsetsToTransaction(...);
7     producer.commitTransaction();
8 } catch(ProducerFencedException e) {
9     producer.close();
10 } catch(KafkaException e) {
11     producer.abortTransaction();
12 }

```

2

Write **atomically** across multiple partitions

use Transaction API

```
1 producer.initTransactions();
2 try {
3     producer.beginTransaction();
4     producer.send(record1);
5     producer.send(record2);
6     producer.sendOffsetsToTransaction(...)
7     producer.commitTransaction();
8 } catch(ProducerFencedException e) {
9     producer.close();
10} catch(KafkaException e) {
11    producer.abortTransaction();
12}
```

- A Producer writes atomically across multiple Partitions using "transactional" messages
 - Either all or no messages in a batch are visible to Consumers
 - Use the new transactions API

(3) установить на продьюсере пропертю **transactional.id**

- должен быть уникальным для каждого инстанса продьюсера (иначе продьюсер-дубликат будет изолирован кафкой)
- если инстанс упал то у нового инстанса должен быть такой же transactional.id как и у старого (те как бы получается что он одинаковый через несколько продьюсерских сессий, независимо от того упал продьюсер или нет)
 - While there may be a 1-1 mapping between an Transactional ID and the internal Producer ID, the main difference is the the Transactional ID is provided by the Producer configurations, and is what enables idempotent guarantees across producers sessions.
 - Ensures any transactions initiated by previous instances of the producer with the same transactional.id are completed. If the previous instance had failed with a transaction in progress, it will be aborted. If the last transaction had begun completion, but not yet finished, this method awaits its completion

<https://www.confluent.io/blog/transactions-apache-kafka/>

3

Fence off open transactions

Use **transactional.id**

- Allow reliability semantics to span multiple producer sessions
 - Set **transactional.id** to guarantee that transactions using the same Transactional ID have completed prior to starting new transactions

Transactional guarantees enable applications to batch messages into a single atomic unit. In particular, a "batch" of messages in a transaction can be written to multiple partitions, and are "atomic" in the sense that writes will fail or succeed as a single unit.

Additionally, stateful applications will also be able to ensure continuity across multiple sessions of the application. In other words, Kafka can guarantee idempotent production and transaction recovery across application bounces.

To achieve this, Kafka requires that the application provide a unique id which is stable across all sessions of the application – the `transactional.id`. While there may be a 1-1 mapping between an Transactional ID and the internal Producer ID, the main difference is the the Transactional ID is provided by the Producer configurations, and is what enables idempotent guarantees across producers sessions.

The method `sendOffsetsToTransaction` sends the consumed offsets to the offsets topic as part of the transaction. With this guarantees, we know that the offsets and the output records will be committed as an atomic unit. This applies in **produce-consume-produce** scenarios.

```
KafkaProducer producer = createKafkaProducer(  
    "bootstrap.servers", "localhost:9092",  
    "transactional.id", "my-transactional-id");  
producer.initTransactions();  
KafkaConsumer consumer = createKafkaConsumer(  
    "bootstrap.servers", "localhost:9092",  
    "group.id", "my-group-id",  
    "isolation.level", "read_committed");  
consumer.subscribe singleton("inputTopic"));  
while (true) {  
    ConsumerRecords records = consumer.poll(Long.MAX_VALUE);  
    producer.beginTransaction();  
    for (ConsumerRecord record : records)  
        producer.send(producerRecord("outputTopic", record));  
    producer.sendOffsetsToTransaction(currentOffsets(consumer), group);  
    producer.commitTransaction();  
}
```

(4) установить на консьюмере `isolation.level=read_committed`

- нужно также выключить автокомит
- также НЕ использовать методы явного комита `commitSync/commityAsync` (эти методы консьюмера заменяются методом продюсера `sendOffsetsToTransaction`)

Transactional

- `isolation.level=read_committed`
- reads only committed transactional messages and all non-transactional messages

Non-Transactional (default)

- `isolation.level=read_uncommitted`
- reads all transactional messages and all non-transactional messages

If using EOS-enabled producers, consumers running older versions (i.e., 0.10.x or older) should be updated as soon as possible. When a consumer sends a request for an older version, the broker assumes the READ_UNCOMMITTED isolation level and converts the message set to the appropriate format before sending back the response, essentially disabling zero-copy for those fetch requests. This conversion can be costly when compression is enabled, so it is important to update the client as soon as possible.



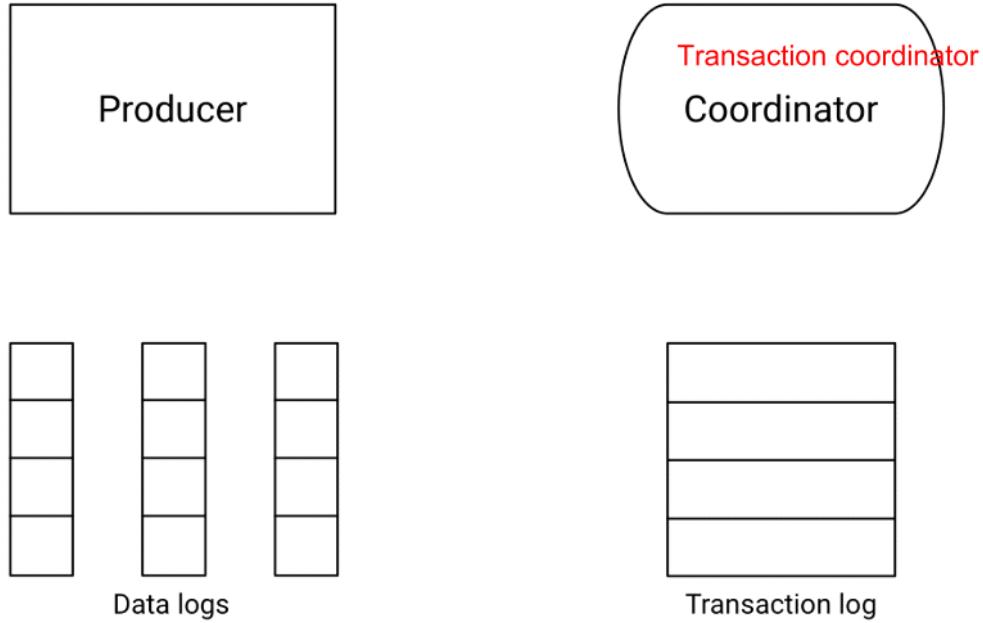
If you design an application as a consume-process-produce pipeline (e.g. read from one topic, write to another), you can combine the consumer offset commits with the producer writing to the output topic in a single atomic transaction, which means each message is effectively processed only once. However, if failure occurs, it's possible messages could be processed multiple times. But in that case, the output of the processing never got committed the first time around, so the semantics are exactly-once, even if it's not truly exactly once delivery and processing under the covers.

One exception is when rebalancing happens in a consumer group - that could result in moving processing of messages from a partition to a different process with a different transactional.id, and EOS is not guaranteed in that case.

Kafka Streams has some special functionality to make the rebalancing case work with EOS, but it's not built-in at this level. EOS was really designed with Kafka Streams in mind, where consume-process-produce with Kafka input and output is the standard execution model.

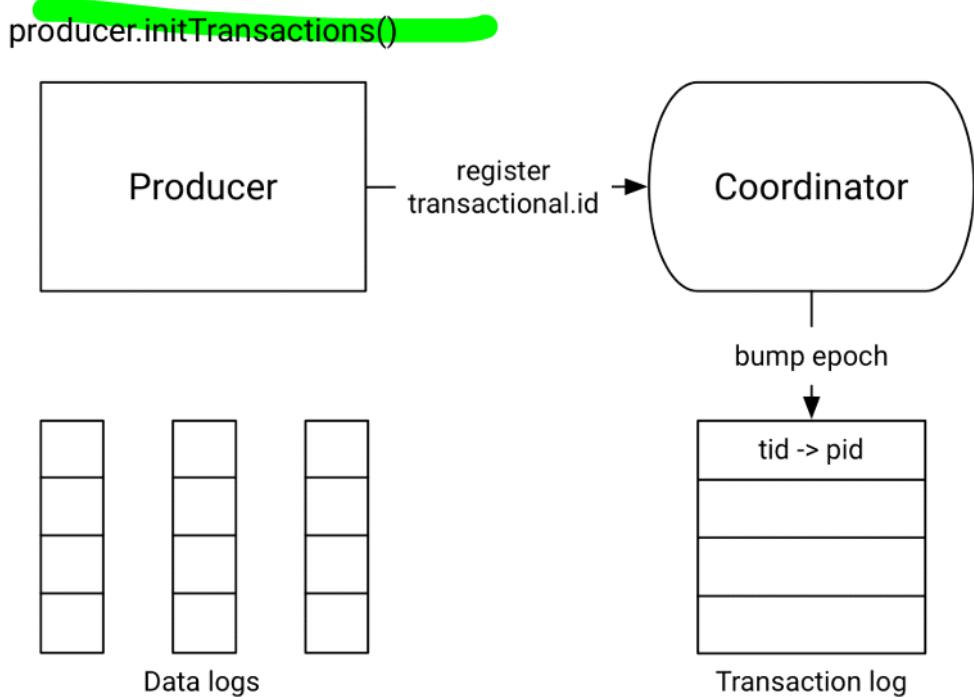
объяснение

- использование транзакций увеличивает нагрузку всего на 3% (те можно не беспокоится и всегда включать exactly once)
 - расчет выполнен для сообщения 1 кб с длиной транзакции 100миллисекунд
 - если сообщение сделать меньше то расходы на транзакцию возрастут
 -



Producers can send data atomically across multiple Partitions to guarantee the receipt of sets of messages. This is a transaction.

A Transaction Coordinator is a module that is available on any Broker. The Transaction Coordinator is responsible for managing the lifecycle of a transaction in the "Transaction Log" – an internal Kafka Topic partitioned by `transactional.id`. The Broker that acts as the Transaction Coordinator is not necessarily a Broker that the Producer is sending messages to. For a given **Producer** (identified by `transactional.id`), the Transaction Coordinator is the leader of the Partition of the Transaction Log where `transactional.id` resides. Because the Transaction Log is a Kafka Topic, it has durability guarantees.

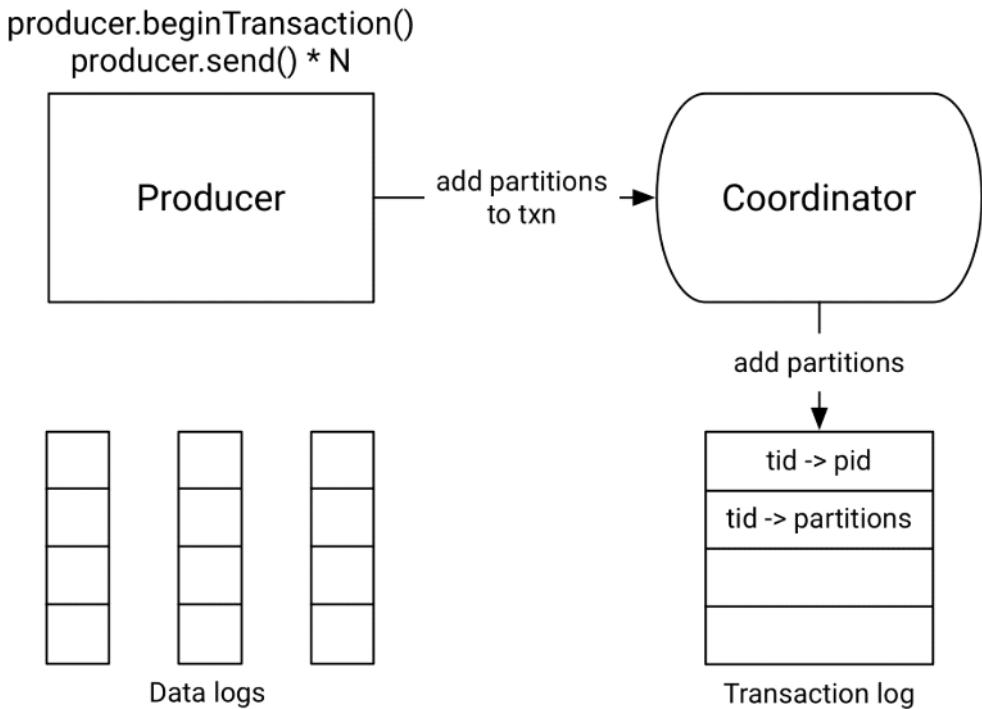


The Producer initiates a transaction. The Producer registers itself to the Transaction Coordinator with `transactional.id`. The Transaction Coordinator records a mapping {
`Transactional ID : Producer ID`}. If the Transactional ID already exists, it means two Producers are fighting for the same identity. The Transaction Coordinator will "fence off" Producers with old PIDs (so-called "zombie Producers") and only allow the Producer with the newest PID to participate in transactions identified by `transactional.id`.

The Transaction Coordinator also increments an **epoch** associated with the `transactional.id`. The epoch is an internal piece of metadata stored for every `transactional.id`.



Once the epoch is bumped, any producers with same `transactional.id` and an older epoch are considered zombies and are fenced off, ie. future transactional writes from those producers are rejected.

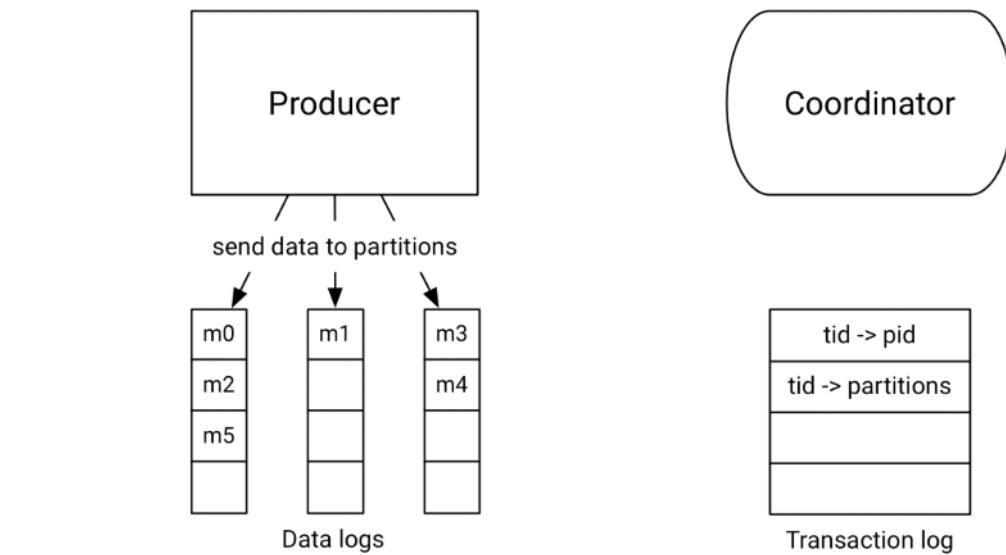


The Producer sends information to the Transaction Coordinator about the Partitions it will write to.



To be precise - "Register Partitions" would happen after producer starts sending data to various partitions (see next slide)
 The producer sends this request to the transaction coordinator the first time a new TopicPartition is written to as part of a transaction. The addition of this TopicPartition to the transaction is logged by the coordinator in step 4.1a. We need this information so that we can write the commit or abort markers to each TopicPartition (see section 5.2 for details). If this is the first partition added to the transaction, the coordinator will also start the transaction timer.

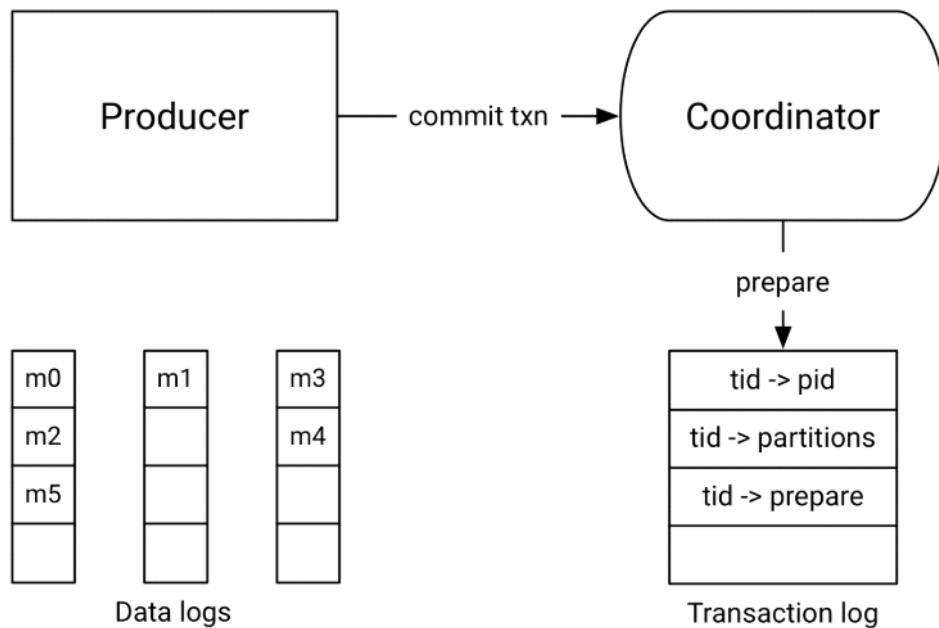
EOS - Transactions - (4/8)



The Producer sends transactional messages which may be on multiple Partitions on multiple Brokers.

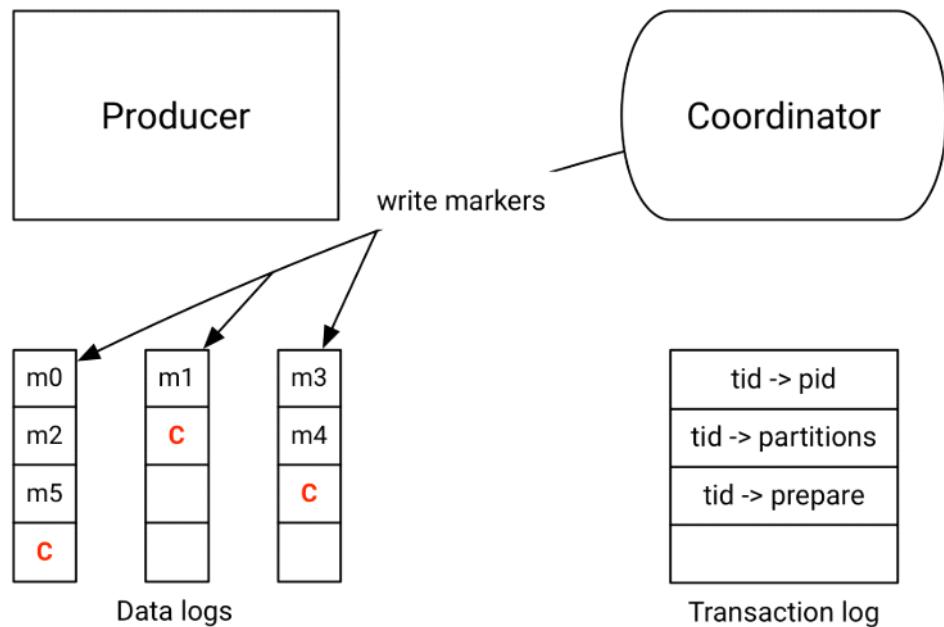
EOS - Transactions - (5/8)

producer.commitTransaction()



Producer commits the transaction. The transaction coordinator marks the transaction as in status of "preparing".

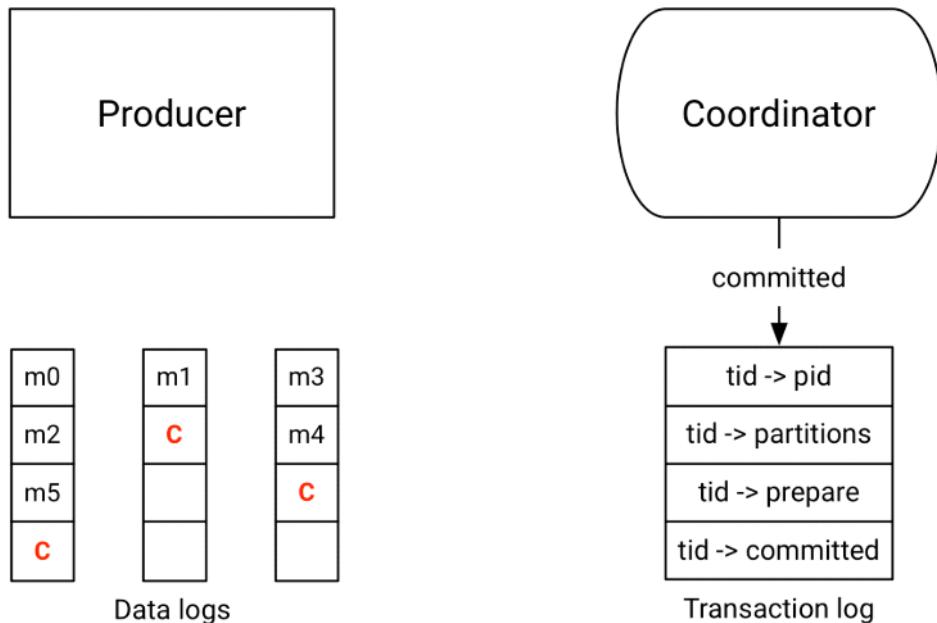
`producer.commitTransaction()`



The Transaction Coordinator writes commit markers to the Partitions the Producer writes to. Commit markers are special messages which log the producer id and the result of the transaction (committed or aborted). These messages are internal only and are not exposed by standard consumer operations.

EOS - Transactions - (7/8)

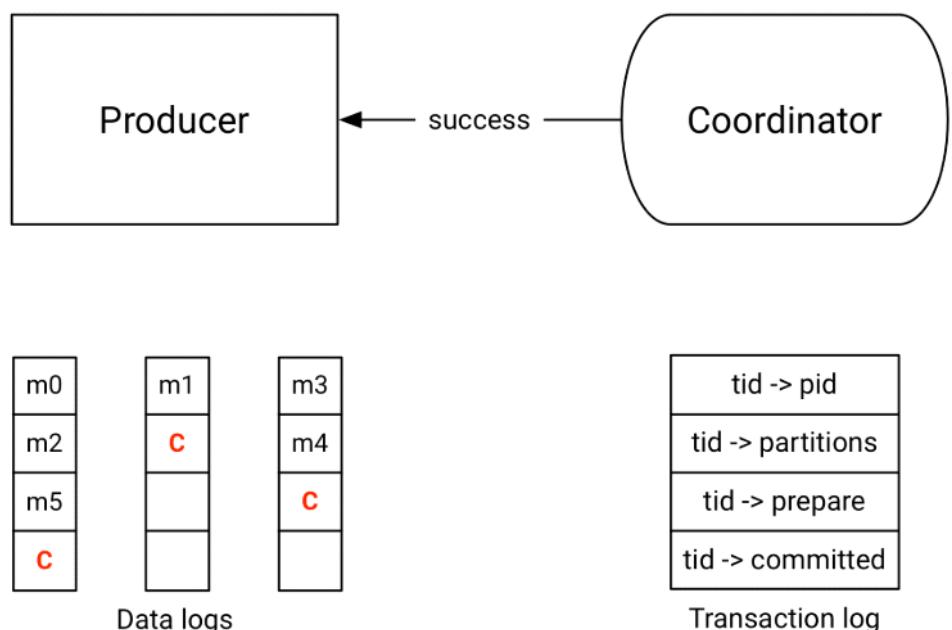
producer.commitTransaction()



The Transaction Coordinator marks the transaction as committed.

EOS - Transactions - (8/8)

producer.commitTransaction()



As a final step the transaction coordinator sends an acknowledgment to the producer.

```

1 producer.initTransactions();
2 try {
3     producer.beginTransaction();
4     producer.send(record1);
5     producer.send(record2);
6     producer.sendOffsetsToTransaction(...);
7     producer.commitTransaction();
8 } catch(ProducerFencedException e) {
9     producer.close();
10} catch(KafkaException e) {
11    producer.abortTransaction();
12}

```

What are these methods doing?

initTransactions

Needs to be called before any other methods when the `transactional.id` is set in the configuration. This method does the following:

1. Ensures any transactions initiated by previous instances of the producer with the same `transactional.id` are completed. If the previous instance had failed with a transaction in progress, it will be aborted. If the last transaction had begun completion, but not yet finished, this method awaits its completion.
2. Gets the internal producer id and epoch, used in all future transactional messages issued by the producer. Note that this method will raise `TimeoutException` if the transactional state cannot be initialized before expiration of `max.block.ms`. Additionally, it will raise `InterruptException` if interrupted. It is safe to retry in either case, but once the transactional state has been successfully initialized, this method should no longer be used.

<code>beginTransaction</code>	<p>Must be called to signal the start of a new transaction. The producer records local state indicating that the transaction has begun, but the transaction won't begin from the coordinator's perspective until the first record is sent.</p> <p>Throws a <code>ProducerFencedException</code> if another producer with the same <code>transactional.id</code> is active</p> <p>Note that prior to the first invocation of this method, you must invoke <code>initTransactions()</code> exactly one time.</p>
<code>commitTransaction</code>	<p>Commits the ongoing transaction. This method will flush any unsent records before actually committing the transaction. Further, if any of the <code>send(ProducerRecord)</code> calls which were part of the transaction hit irrecoverable errors, this method will throw the last received exception immediately and the transaction will not be committed. So all <code>send(ProducerRecord)</code> calls in a transaction must succeed in order for this method to succeed.</p>
<code>sendOffsetsToTransaction</code>	<p>Although not on the slide, this method is important in more complex scenarios (e.g. stream processing).</p> <p>Sends a list of specified offsets to the consumer group coordinator, and also marks those offsets as part of the current transaction.</p> <p>These offsets will be considered committed only if the transaction is committed successfully. The committed offset should be the next message your application will consume, i.e.</p> $\text{lastProcessedMessageOffset} + 1.$ <p>This method should be used when you need to batch consumed and produced messages together, typically in a consume-transform-produce pattern. Thus, the specified <code>consumerGroupId</code> should be the same as config parameter <code>group.id</code> of the used consumer. Note, that the consumer should have <code>enable.auto.commit=false</code> and should also not commit offsets manually (via sync or async commits).</p>

ребаланс партиций может сломать транзакции (на консьюмерах)

One exception is when rebalancing happens in a consumer group - that could result in moving processing of messages from a partition to a different process with a different `transactional.id`, and EOS is not guaranteed in that case.

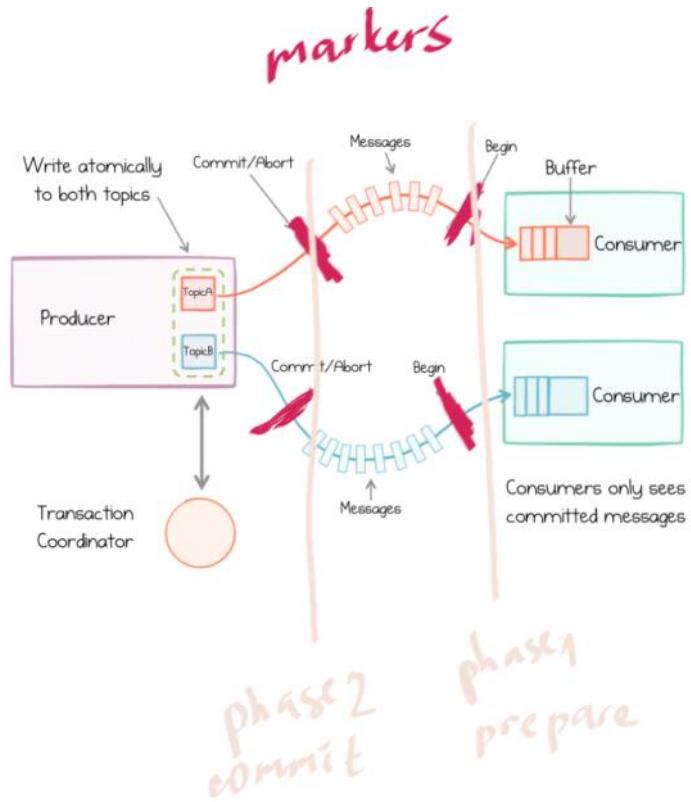
объяснение модели маркеров Лампорта 2FA

<https://www.confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it/>

Маркеры «Начало» отправляются вниз по обоим *. Затем мы отправляем наши сообщения. Наконец, когда мы закончили, мы добавляем к каждой теме маркер «Применить» (или «Прервать»), который завершает

транзакцию.

Теперь цель транзакции - гарантировать, что нижестоящие программы будут видеть только «зарегистрированные» данные. Чтобы это сработало, когда потребитель видит маркер «Начало», он начинает внутреннюю буферизацию. Сообщения задерживаются до тех пор, пока не появится маркер «Фиксация». Тогда и только тогда сообщения представляются программе-потребителю. Эта буферизация гарантирует, что потребители будут читать только зарегестрированные данные *.





When does the problem arise?



- As a Kafka Consumer (at least once):

- You receive twice the same message if the Kafka broker reboots or your Kafka Consumer restarts.
- This is because offsets are committed once in a while, but the data may have been processed already



- As a Kafka Producer:

- You send twice the same message to Kafka if you don't receive an **ack** back from Kafka (because of the retry logic).
- But not receiving an ack does not mean Kafka hasn't received your message. It may mean that the network just failed, instead of Kafka

What's the problem with At Least Once Semantics anyway?



- Cases when it's **not acceptable** to have at least once:
 - Getting the exact count by key for a stream
 - Summing up bank transactions to compute a person's bank balance
 - Any operation that is not idempotent
 - Any financial computation
- Cases when it's **acceptable** to have at least once:
 - Operations on time windows (because the time itself can be vague)
 - Approximate operations (counting the number of times an IP hits a webpage to detect attacks and web scraping).
 - Idempotent operations (such as max, min, etc...)

??

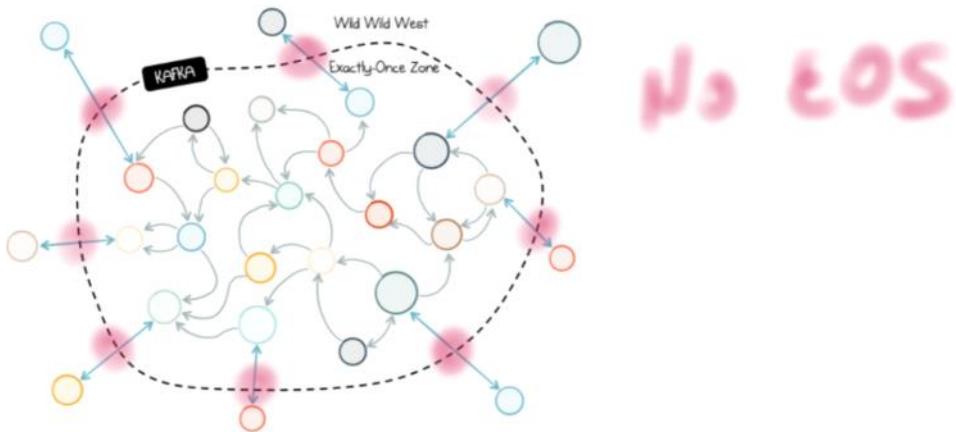
BankBalance Kafka Producer Solution



- **MUST WATCH** – Special setting for idempotent producer!!
- Let's go ahead and write our BankTransactionsProducer!

(1) EOS может не сработать, когда в цепочке стоит **вызов внешней службы** или сервиса

- во-первых, они работают только в ситуациях, когда и ввод, и вывод проходят через Kafka. Если вы вызываете внешнюю службу (например, через HTTP), обновляете базу данных, записываете в stdout или что-то другое, кроме записи в брокер Kafka и из него, транзакционные гарантии не применяются, и вызовы могут дублироваться. Так же, как и с использованием транзакционной базы данных, транзакции работают только тогда, когда вы используете Kafka.



Но LOS

(2) kafka-транзакцию невозможно откатить (нету rollback)

- тк сообщения в логе не удаляются
- только путем добавочных компенсирующих сообщений
- Также похоже на доступ к базе данных, транзакции фиксируются при отправке сообщений, поэтому после их фиксации их невозможно откатить, даже если последующая транзакция в исходящем потоке не удалась. Таким образом, если пользовательский интерфейс отправляет транзакционное сообщение в службу заказов, а служба заказов выходит из строя при отправке собственных сообщений, любые сообщения, отправленные службой заказов, будут отменены, но нет способа откатить транзакцию в пользовательском интерфейсе. Если вам нужны мультисервисные транзакции, подумайте о внедрении Sagas.

(3) нет никаких гарантий относительно того, когда произвольный потребитель прочитает эти закомиченные-сообщения (также как в БД, транзакция коммитится в кафку-БД)

- Transactions commit atomically in the Broker (just like a transaction would commit in a database) but there are no guarantees regarding when an arbitrary consumer will read those messages. This may appear obvious, but it is sometimes a point of confusion. Say we send a message to the Orders topic and a message to the Payments topic, inside a transaction, there is no way to know when a consumer will read one or the other, or that they might read them together. But again note, this is identical to the contract offered by a transactional database.

??? But while there is full support for individual producers and consumers, the use of transactions with consumer groups is subtly nuanced (essentially you need to ensure that a separate transactional producer is used for each consumed partition) so we recommend using the Streams API when chaining together producers and consumers where consumer groups are supported without any extra considerations on your part

* гарантии транзакций (atomic commit)

4 января 2021 г. 9:44

общее

- Kafka поставляется со встроенными транзакциями почти так же, как и большинство реляционных баз данных. Как мы увидим, реализация совершенно иная, но цель схожа: гарантировать, что наши программы будут давать предсказуемые и повторяемые результаты, даже когда что-то не получается.
- <https://www.confluent.io/blog/transactions-apache-kafka/>

(гарантия атмосферности транзакций 1) Messages sent to different topics, within a transaction, will either all be written or none at all

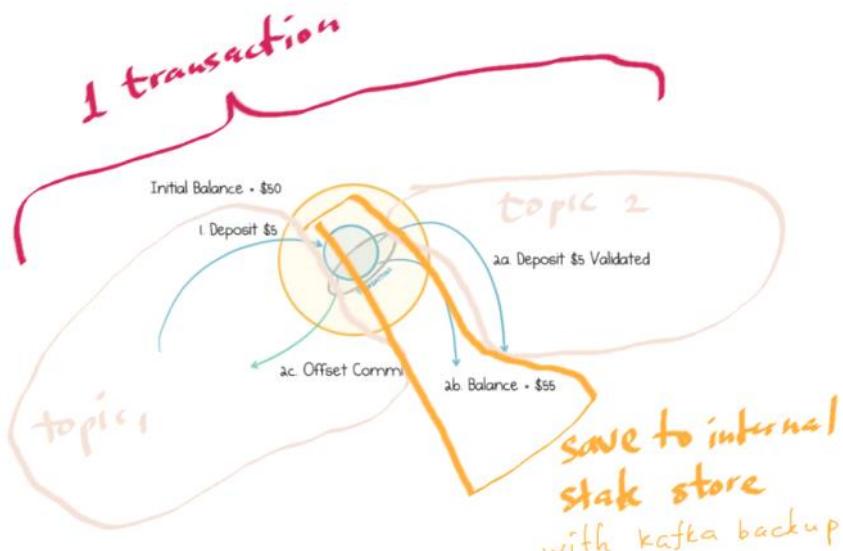
- Сообщения, отправленные в разные темы в рамках транзакции, будут записаны либо полностью, либо вообще не будут.
- Они позволяют атомарно отправлять группы сообщений по разным темам - например, «Заказ подтвержден» и «Уменьшить уровень запасов», что приведет к нарушению согласованности системы в случае успешного выполнения только одного из двух.

(гарантия атмосферности транзакций 2) Messages sent to a single topic, in a transaction, will never be subject to duplicates, even on failure.

- Сообщения, отправленные в одну тему в транзакции, никогда не будут дублироваться, даже в случае сбоя.
- Они удаляют дубликаты, из-за чего многие потоковые операции дают неверные результаты (даже такие простые, как подсчет).

в одной транзакции можно записать в свой state store из одного топика + отправить в следующий топик

- Поскольку Kafka Streams использует хранилища состояний, а хранилища состояний поддерживаются темой Kafka, когда мы сохраним данные в хранилище состояний, а затем отправляем сообщение в другую службу, мы можем обернуть все это в транзакцию. Это свойство оказывается особенно полезным.
- As a state store gets its durability from a Kafka topic, we can use transactions to tie writes to the state store and writes to other output topics together. This turns out to be an extremely powerful pattern because it mimics the tying of messaging and databases together atomically, something that traditionally required painfully slow protocols like XA.
- Если транзакции включены в Kafka Streams, все эти операции будут автоматически заключены в транзакцию, гарантируя, что баланс всегда будет атомарно синхронизирован с депозитами



Обновление базы данных и отправка сообщения должны происходить в пределах одной транзакции

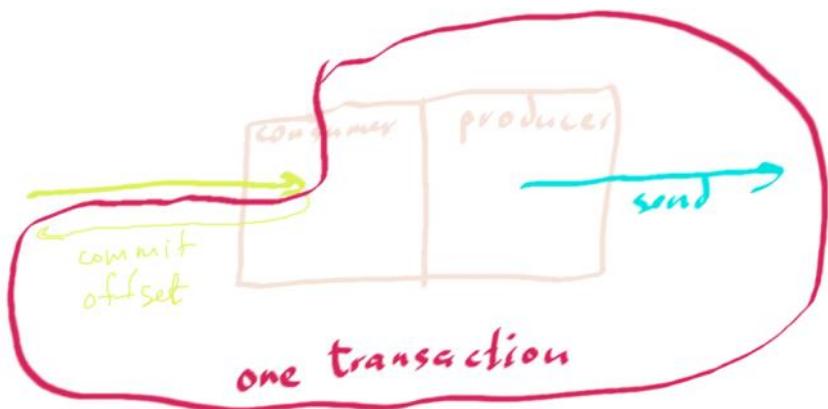
- иначе сервис может обновить БД и, например, отказать до того, как сообщение будет отправлено. Если не выполнять эти две операции атомарно, сбой может оставить систему в несогласованном состоянии.

Транзакции в Kafka позволяют создавать длинные цепочки сервисов, в которых обработка каждого шага в цепочке обернута EOS гарантией

- Transactions in Kafka allow the creation of long chains of services, where the processing of each step in the chain is wrapped in exactly-once guarantees. This reduces duplicates, which means services are easier to program and, as we'll see later in this chapter, transactions let us tie streams and state together when we implement storage either through Kafka Streams state stores or using the Event Sourcing design pattern. All this happens automatically if you are using the Kafka Streams API.
- Kafka records the progress that each consumer makes by storing an offset in a special topic, called `consumer_offsets`. So to validate each deposit exactly once, we need to perform the final two actions—
 - o (a) send the “Deposit Validated” message back to Kafka, and
 - o (b) commit the appropriate offset to the `consumer_offsets` topic—as a single atomic unit (Figure 12-3). The code for this would look something like the following

```
//Read and validate deposits
validatedDeposits = validate(consumer.poll(0))

//Send validated deposits & commit offsets atomically
producer.beginTransaction()
producer.send(validatedDeposits)
producer.sendOffsetsToTransaction(offsets(consumer))
producer.endTransaction()
```



в DDD нет распределенных транзакций, есть только транзакции в пределах одного микросервиса(ограниченного контекста или агрегата)

- Let's begin by making one thing clear: transactions are fine within individual services, where we can, and should, guarantee strong consistency. This means that it is fine to use transactional semantics within a single service (the bounded context)—which is something that can be achieved in many ways: using a traditional SQL database like Oracle, a modern distributed SQL database like CockroachDB, or using event sourcing through Akka Persistence. What is problematic is expanding them beyond the single service, as a way of trying to bridge data consistency across multiple services (i.e. bounded contexts).
- The problem with transactions is that their only purpose is to try to maintain the illusion that the world consists of a single globally strongly consistent present—a problem that is magnified exponentially in distributed transactions (XA, Two-phase Commit, and friends). We already have discussed this at length: it is simply not how the world works, and computer science is no different.

Use Distributed Sagas, Not Distributed Transactions

- <http://vasters.com/archive/Sagas.html>
- [Caitie McCaffery - Distributed Sagas: A Protocol for Coordinating Microservices - .NET Fringe 2017](#)
- Этот шаблон был определен Гектором Гарсиа-Молиной в 1987 году как способ сократить период времени, в течение которого база данных должна принимать блокировки. Он не создавался для распределенных систем, но оказалось, что он очень хорошо работает в распределенном контексте.
- Шаблон Saga 8 - это шаблон управления отказами, который обычно используется как

альтернатива распределенным транзакциям.

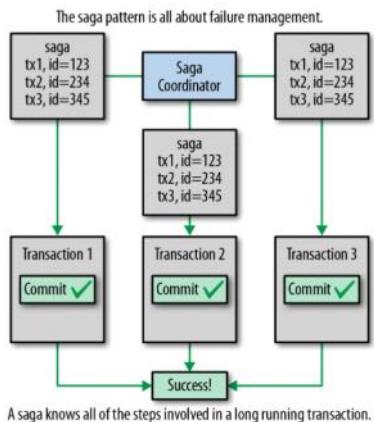
- Это помогает вам управлять длительными бизнес-транзакциями, которые используют компенсирующие действия для управления несоответствиями (сбои транзакций).

saga - Суть идеи состоит в том, что одну длительную распределенную транзакцию можно рассматривать как композицию нескольких быстрых локальных транзакционных шагов

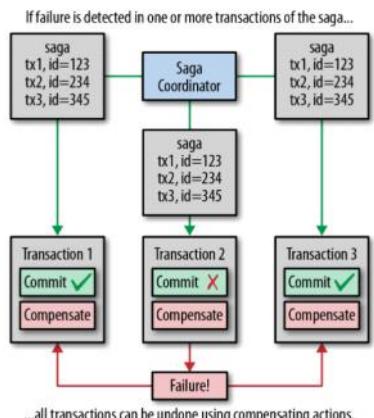
- Суть идеи состоит в том, что одну длительную распределенную транзакцию можно рассматривать как композицию нескольких быстрых локальных транзакционных шагов.
- Каждый шаг транзакции сопряжен с компенсирующим реверсивным действием (реверсирование с точки зрения бизнес-семантики, без обязательного сброса состояния компонента), так что вся распределенная транзакция может быть отменена в случае сбоя путем выполнения компенсирующего действия каждого шага.
- В идеале эти шаги должны быть коммутативными, чтобы их можно было выполнять параллельно. (коммутативный закон сложения: $m + n = n + m$)
 - o Saga - отличный инструмент для обеспечения атомарности длительных транзакций.
 - o Но важно понимать, что это не решение для изоляции .
 - o Одновременно выполняемые саги потенциально могут влиять друг на друга и вызывать ошибки.
 - Если это приемлемо, это зависит от варианта использования.
 - Если это неприемлемо, вам нужно использовать другую стратегию, например, убедиться, что сага не охватывает несколько границ согласованности, или просто использовать другой шаблон или инструмент для работы.
- The essence of the idea is that one long-running distributed transaction can be seen as the composition of multiple quick local transactional steps. Every transactional step is paired with a compensating reversing action (reversing in terms of business semantics, not necessarily resetting the state of the component), so that the entire distributed transaction can be reversed upon failure by running each step's compensating action. Ideally, these steps should be commutative so that they can be run in parallel.
- Одним из преимуществ этого метода (см. Рис. 6-3) является то, что он **eventually consistent** в конечном итоге согласован и хорошо работает с разделенными и асинхронно взаимодействующими компонентами, что делает его отличным подходом для архитектур, управляемых событиями и сообщениями.

saga pattern - is all about failure management

- long-running distributed workflows across multiple services



A saga knows all of the steps involved in a long running transaction.



...all transactions can be undone using compensating actions.

* optimistic concurrency control

16 октября 2020 г. 7:34

Optimistic concurrency control

From Wikipedia, the free encyclopedia

..each transaction verifies that no other transaction has modified the data it has read...

transaction has modified the data it has read. If the check reveals conflicting modifications, the committing transaction rolls back and can be restarted.^[1] Optimistic concurrency control was first proposed by H.T. Kung and John T. Robinson.^[2]

в сообщении должен быть идентификатор сообщение и номер версии (которая меняется после каждого события Created, Validated...)

```
"Customer"{
    "CustomerId": "1234"
    "Source": "ConfluentWebPortal"
    "Version": "1"
    ...
}
```

- The basic premise of identity is that it should correlate with the real world: an order has an OrderId, a payment has a PaymentId, and so on. If that entity is logically mutable (for example, an order that has several states, Created, Validated, etc., or a customer whose email address might be updated), then it should have a version identifier also:

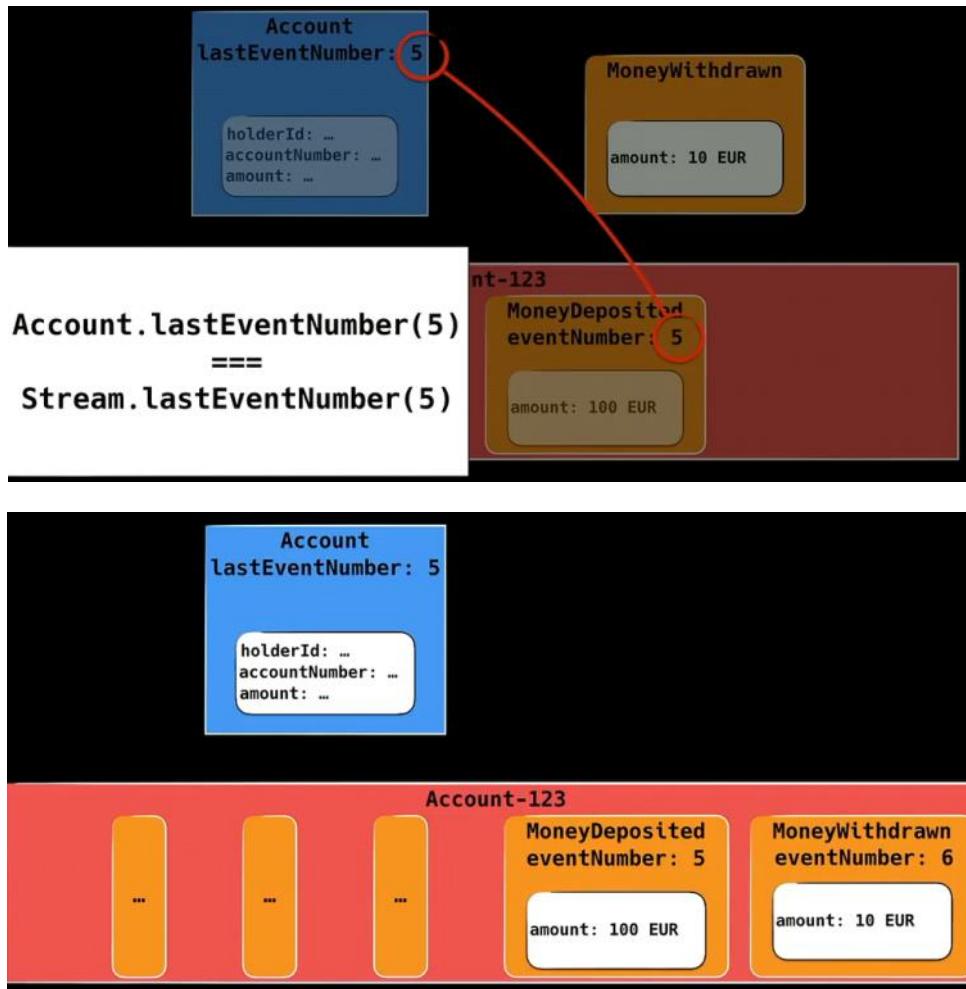
бизнесовый идентификатор сообщения не всегда легко придумать

- Хотя определение идентификатора заказа относительно очевидно, не все потоки событий имеют такое четкое понятие идентичности. Если бы у нас был поток событий, представляющий средний баланс учетной записи на регион в час, мы могли бы придумать подходящий ключ, но вы можете представить, что он был бы намного более хрупким и подверженным ошибкам.

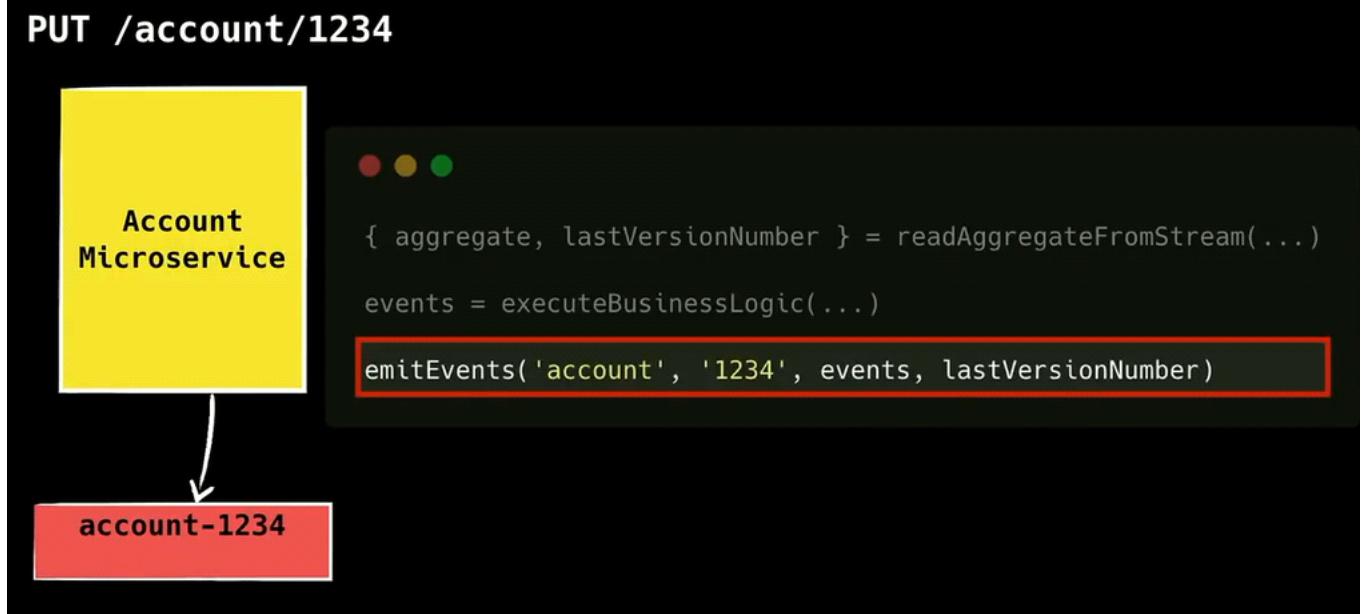
хороший кейс

- запоминается номер последнего ивента на запись
- и если придет ивент на чтение то проверяется все ли ивенты на запись обработаны
- <https://issues.apache.org/jira/browse/KAFKA-2260>

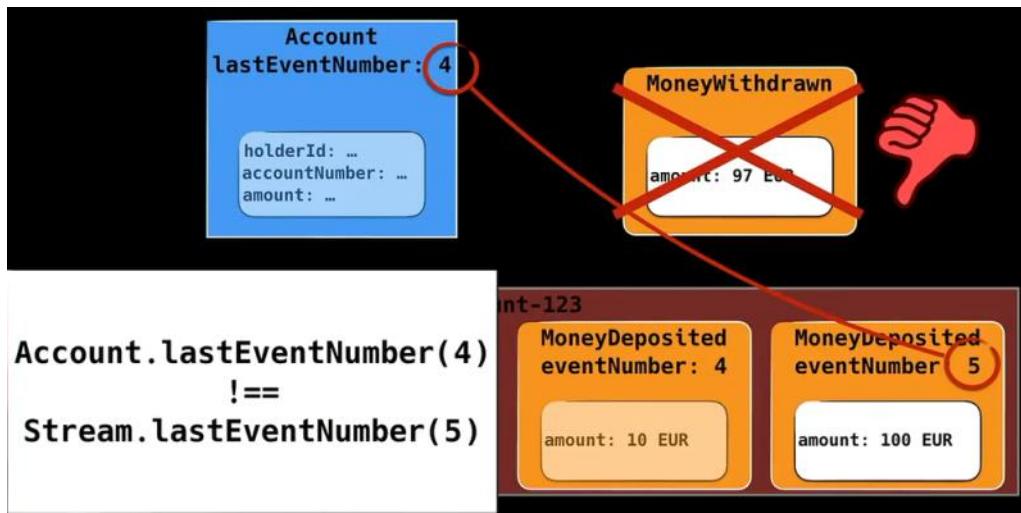
- <https://issues.apache.org/jira/browse/KAFKA-2260>



PUT /account/1234



плохой кейс



@ at least once / at most once

18 октября 2020 г. 15:32

(1) At-most-once delivery (это самый базовый тип гарантии доставки)

—Each request is sent once and never retried. If it is lost or the recipient fails to process it, there is no attempt to recover. Therefore, the desired function may be invoked once or not at all. **This is the most basic delivery guarantee.** It has the lowest implementation overhead, because neither sender nor receiver is required to maintain information on the state of their communication.

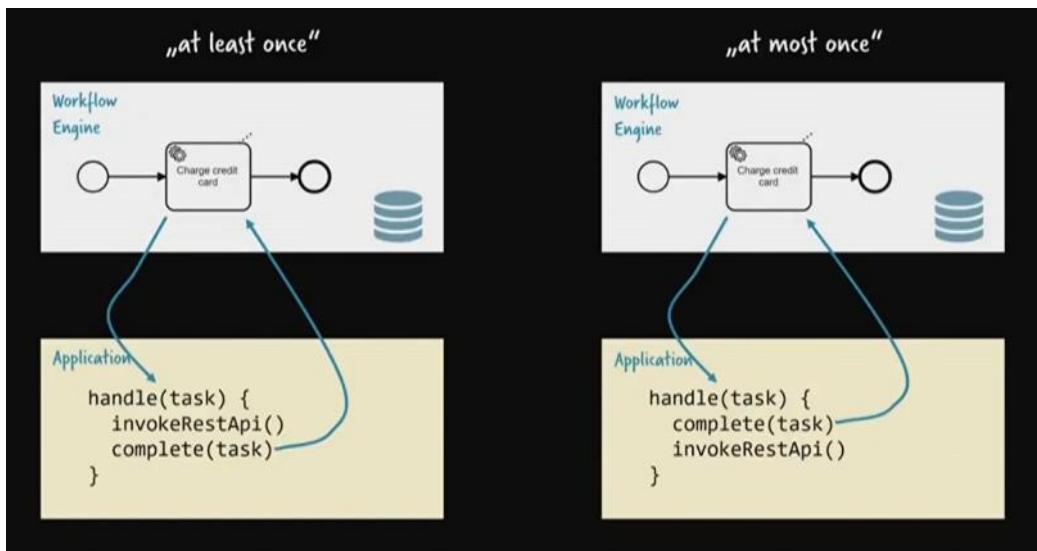
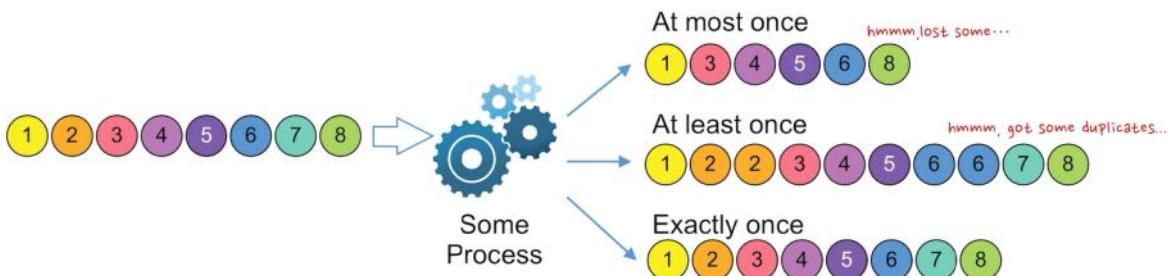
(2) At-least-once delivery—

Trying to guarantee that a request is processed requires two additions to at-most-once semantics. First, the recipient must acknowledge receipt or execution (depending on the requirements) by sending a confirmation message. Second, the sender must retain the request in order to resend it if the sender doesn't receive the confirmation. Because a resend can be the result of a lost confirmation, the recipient may receive the message more than once. Given enough time and the assumption that communication will eventually succeed, the message will be received at least once.

(3) Exactly once delivery

If a request must be processed but must not be processed twice, then, in addition to at-least-once semantics, the recipient must deduplicate messages: that is, they must keep track of which requests have already been processed. **This is the most costly scheme because it requires both sender and receiver to track communication state.** If, in addition, requests must be processed in the order they were sent, then throughput may be further limited by the need to complete confirmation round trips for each request before proceeding with the next—unless flow-control requirements are compatible with buffering on the receiver side.

Delivery Guarantees



Семантики обработки сообщений

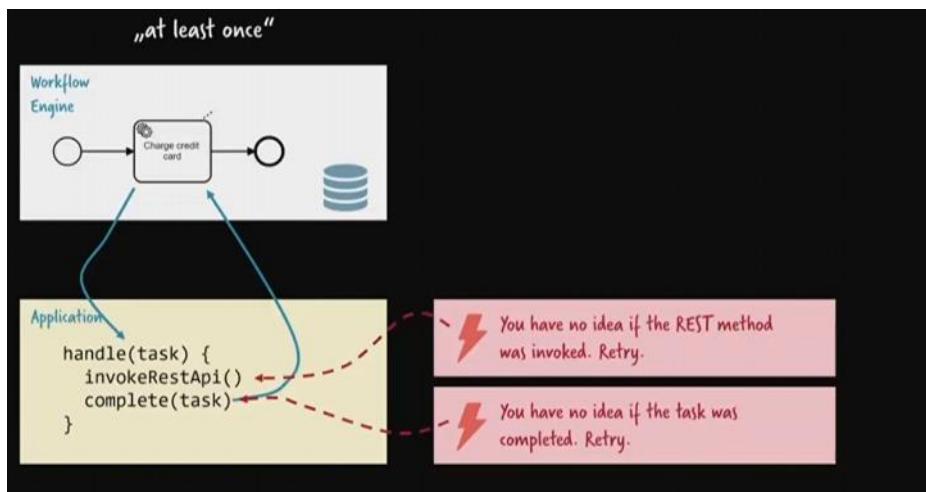
- Минимум один раз / At-least-once
- Максимум один раз / At-most-once: вообще фигня какая-то!
- Точно один раз / Exactly-once: То что вы на самом деле хотите

at least once ==значает больше чем раз (легко сделать идемпотентность)

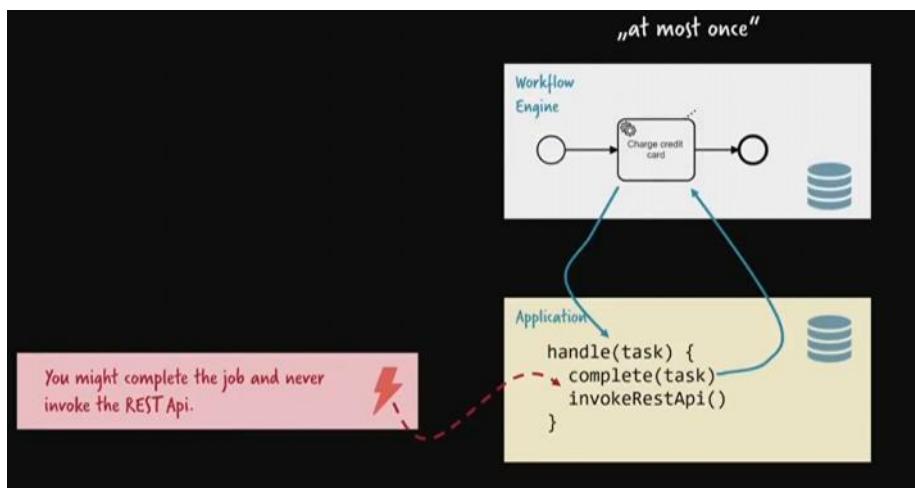
Сообщение потреблено, но еще не подтверждено. База данных обновляется, а затем сообщение подтверждается. В случае сбоя он перезапускается, и сообщение обрабатывается снова.

Дilemma проистекает из того факта, что доставка сообщения не связана напрямую с обновлением устойчивых данных, кроме как через действие приложения. Хотя можно связать потребление сообщений с обновлением надежных данных, это обычно не доступно. Отсутствие этой связи приводит к возникновению окон сбоев, когда сообщение доставляется более одного раза. Вместо того, чтобы терять сообщения, система передачи сообщений доставляет их хотя бы один раз.

Следствием такого поведения является то, что приложение должно допускать повторные попытки сообщения и доставку не по порядку.



не больше одной



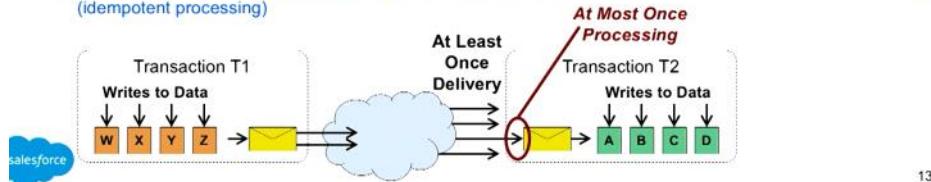
Messaging Semantics

- Transactional messaging is pretty cool

- A transaction may include the desire to send a message
 - Transactional updates happen atomically with desire to send
- A transaction may atomically consume an incoming message
 - Message consumption is atomic with the work of the message

- Exactly once semantics can be supported

- A committed desire to send, causing one or more sends (retry until acknowledged)
- The message must be processed at the receiver at-most-once (idempotent processing)



13

Challenges with At-Most-Once

Remember the Messages You've Processed

Don't Process the Message Twice

How Do You Remember Messages?

Gotta Detect Duplicates

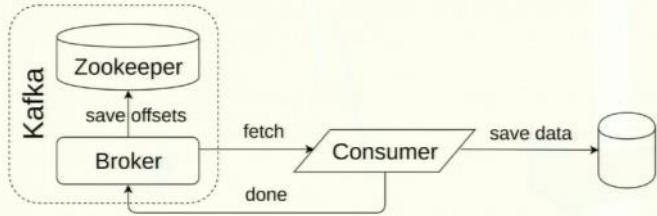
How Long Do You Remember?

Does the Destination Split? Move?

если я в в клиенте прочитал ивенты -> записал в свою БД -> не смог закомитить оффсет в кафку
то сообщения забираются повторно и у меня будут дубли в моей БД

Сохранение отступов

- Сохранение вручную (at least once)



Exactly Once Semantics

What?

- Strong **transactional guarantees** for Kafka
- Prevents clients from processing duplicate messages
- Handles failures gracefully

Use Cases

- Tracking ad views
- Processing financial transactions
- Stream processing

Слабости семантики

- Повторные попытки продюсера небезопасны
- Данные не пишутся автоматично с их смещениями (offsets)
- Нет защиты от зомби

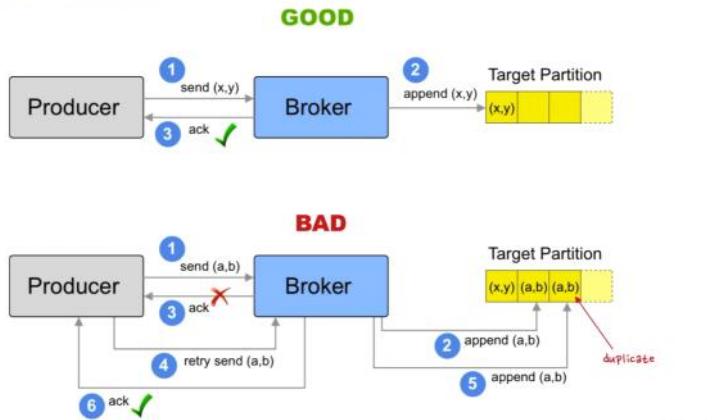
Семантика «Только один раз»

- Идемпотентный продюсер – доставка сообщений без дубликатов
- Транзакционный продюсер: возможность автоматной записи в несколько патриции (в том числе в Хранилище offsets)
- Read-committed консьюмер: только законченные сообщения будут возвращены

(1) идемпотентный продюсер (к сообщению добавляется producer_id)

для того чтобы побороть повторные сообщения продюсера при потере ack от брокера

Idempotent Producers



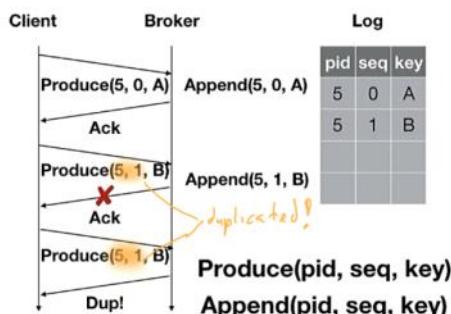
Идемпотентный продюсер

- ProducerId определяет экземпляр продюсера
- Sequence служит для дедупликации
- enable.idempotence=true

Log Message

Offset
Key
Value
Timestamp
Headers
ProducerId
Sequence

- ProducerId определяет экземпляр продюсера
- Sequence служит для дедупликации

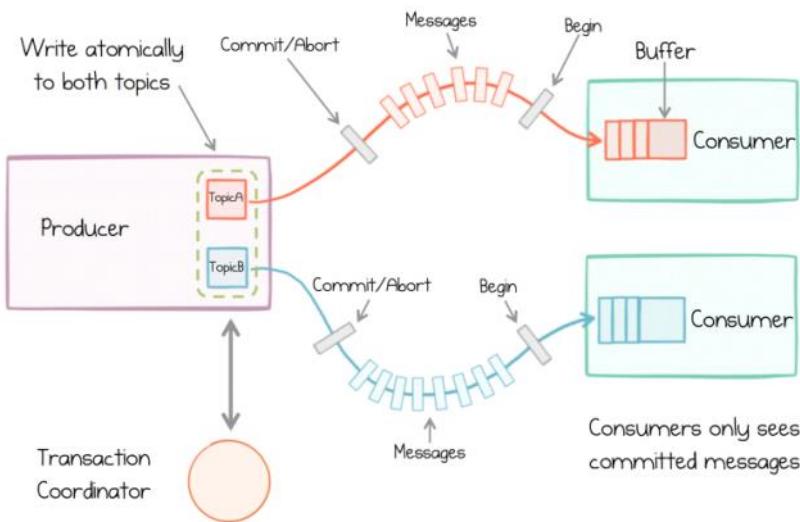


(2) снапшоты Чэнди-Лэмпорта (часы Лэмпорта, векторные часы)

- для того чтобы данные писались в брокер атомарно вместе с записью их смещений
 - Atomic writes across multiple partitions
- <https://blog.acolyer.org/2015/04/22/distributed-snapshots-determining-global-states-of-distributed-systems/>
- <https://www.confluent.io/blog/chain-services-exactly-guarantees/>

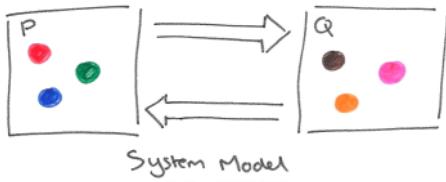
- специальные маркеры "S" которые определяют снапшоты
- Одним из ключевых отличий является использование сообщений-маркеров, которые проходят через различные потоки. Сообщения-маркеры - это идея, впервые представленная Ченди и Лэмпортом почти тридцать лет назад в методе, называемом «Модель маркеров моментальных снимков». Транзакции Канфа - это адаптация этой идеи, хотя и с несколько иной целью.
- Маркеры «Начало» отправляются вниз по обоим *. Затем мы отправляем наши сообщения. Наконец, когда мы

- закончили, мы добавляем к каждой теме маркер «Commit» (или «Прервать»), который завершает транзакцию.
- Теперь цель транзакции - гарантировать, что нижестоящие программы будут видеть только «зафиксированные» данные. Чтобы это сработало, когда потребитель видит маркер «Начало», он начинает внутреннюю буферизацию. Сообщения задерживаются до тех пор, пока не появится маркер «Commit». Тогда и только тогда сообщения представляются программе-потребителю. Эта буферизация гарантирует, что потребители когда-либо читают только зафиксированные данные

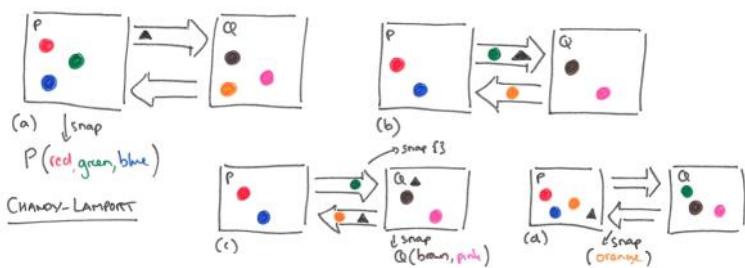
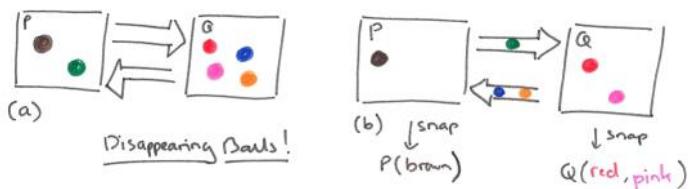


алгоритм Лампорта

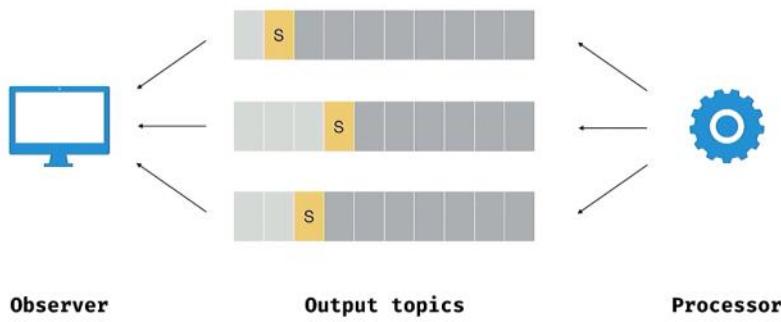
- Записанные состояния процесса и каналов должны быть собраны и скомпонованы для формирования записанного глобального состояния.
- В небольшом повороте примеров, приведенных в статье, представьте, что два процесса обмениваются набором цветных шаров между собой (отправляя сообщения, указывающие на передачу права собственности на цветной шар). Состояние каждого из двух процессов (назовем их P и Q) в любой момент времени - это просто набор шаров, которыми они владеют. Мы запустим систему в начальном состоянии с красным, зеленым и синим шариками, удерживаемыми процессом P, и коричневыми, розовыми и оранжевыми шариками, удерживаемыми процессом Q.
- P снимает свое состояние как P (красный, зеленый, синий), помещает маркер в канал PQ, а затем продолжает обработку, отправляя зеленый шар в Q по каналу PQ. Параллельно Q отправил оранжевый шар P, поэтому Q находится в состоянии Q (коричневый, розовый). Q получает маркер на канале PQ. Когда он получает маркер, Q снимает свое состояние - Q (коричневый, розовый) - и записывает состояние канала PQ как пустое. Q теперь отправляет маркер по каналу QP. P получает оранжевый шар на канале QP, а затем маркер. Поскольку P уже записал свое состояние, он просто записывает состояние канала QP как (оранжевый)



так как шары находятся в пути то моментальный снимок их учесть не может

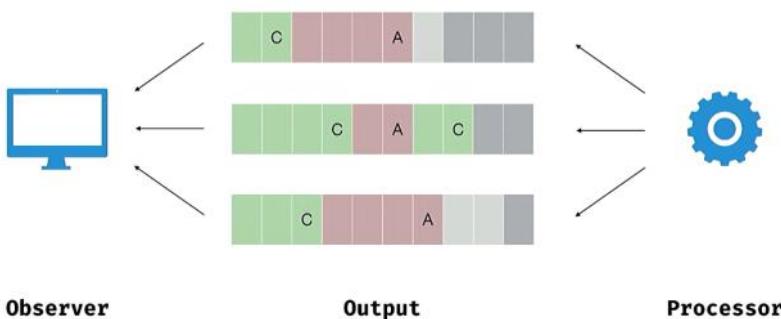


модификация для кафки



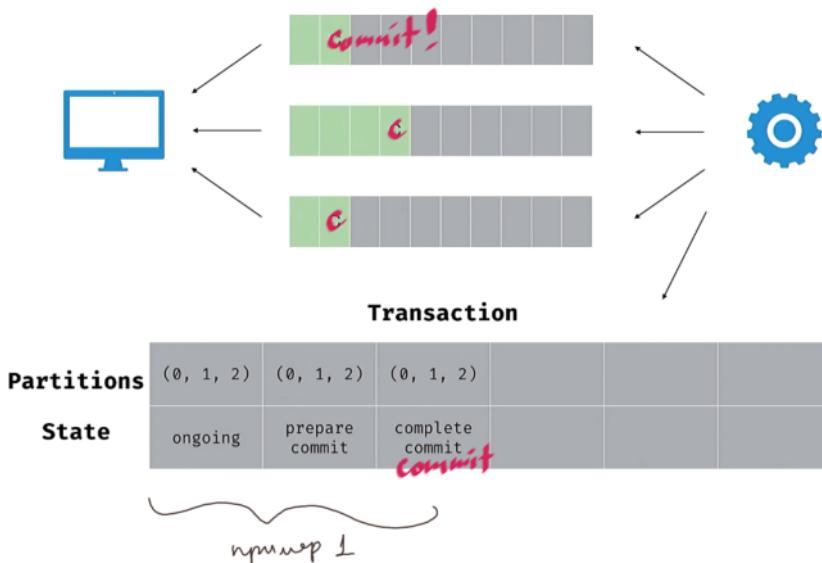
- Два типа маркеров COMMIT and ABORT

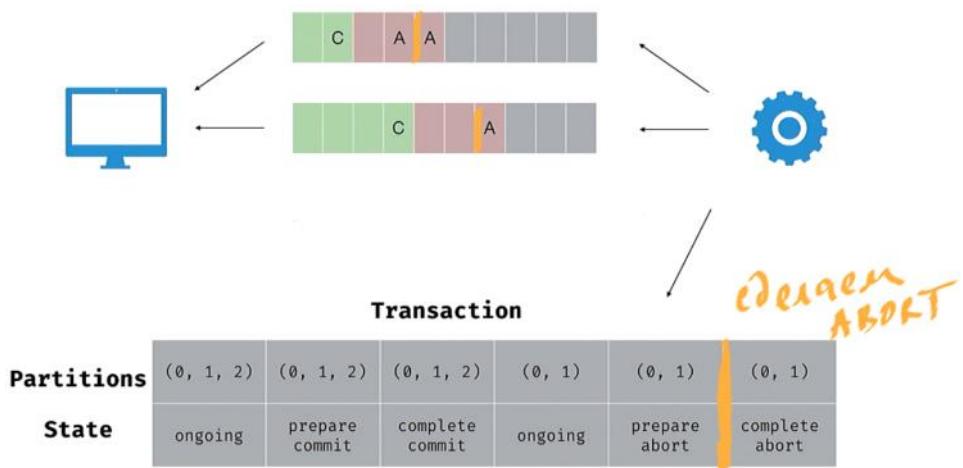
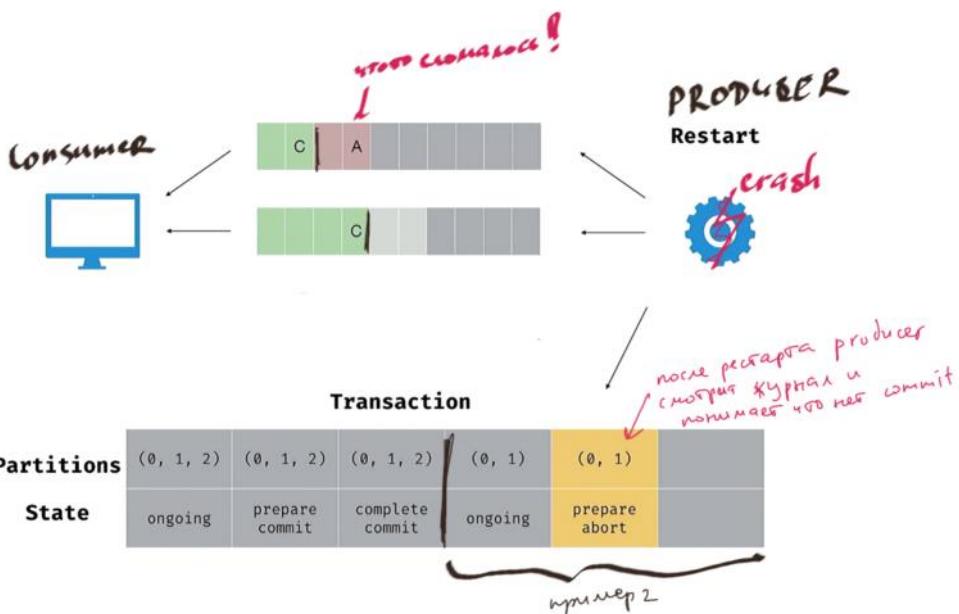
- Мягкие (Soft) типы сбоев обрабатываются записью ABORT маркера во все патриции



- 2-х фазный коммит и транзакционный лог

- Запись «желания коммита» в лог
- Запись маркеров в топики
- Запись маркера о завершении транзакции (commit / abort)

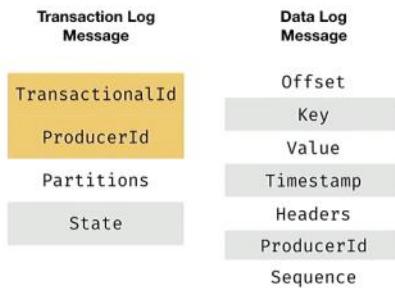




(3) Zombie Fencing (фехтование для защиты от зомби процессов которые то возникают то пропадают)

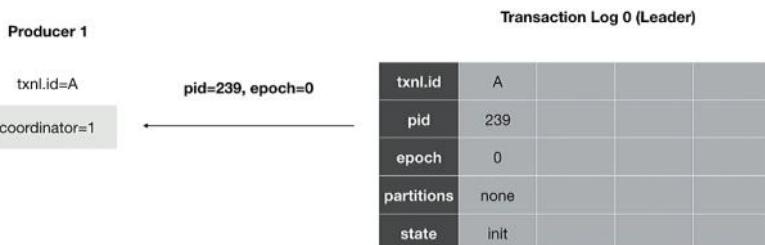
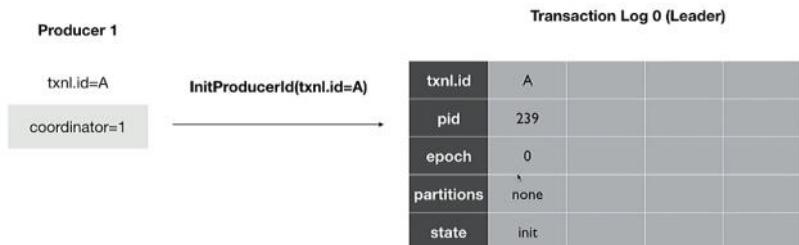
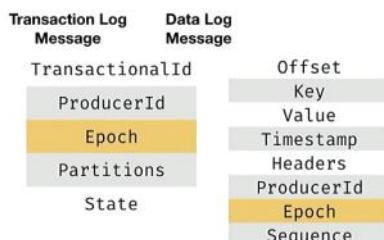
- Чтобы остановить зомби нам нужен «забор»
- Но для начала, надо каким-то образом идентифицировать экземпляр пробьюсера между перезапусками

- “transactionalId” связан с producerId
- Не надо сохранять producerId
- `transactional.id={id}`

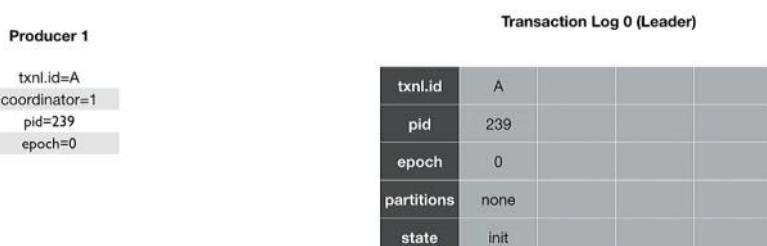


- Мы добавили эпоху продюсера epoch

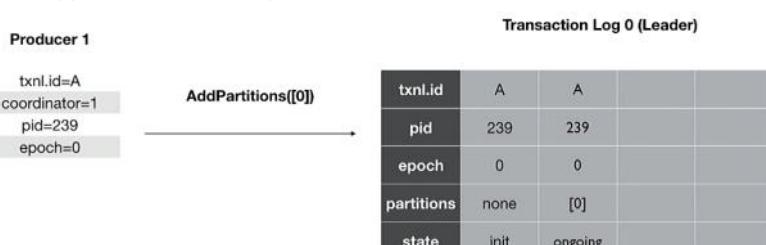
- Epoch инкрементируется при инициализации продюсера

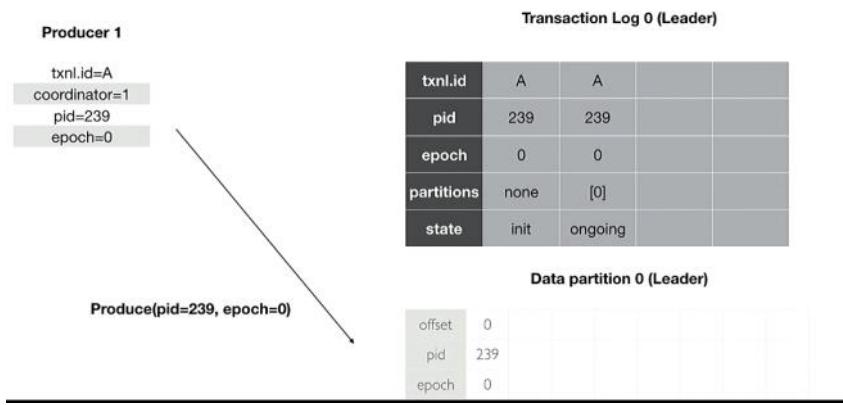


теперь продюсер знает свой номер и эпоху

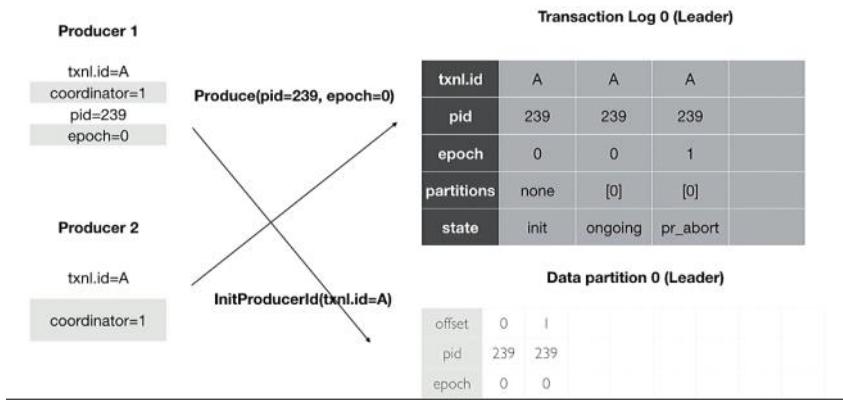


если продюсер уже пишет сообщение в очередь





эпохи двух продьюсеров не совпадают



Семантика «exactly-once»

```

p.initTxns()
c.subscribe(inputTopics)
while (true) {
    offsets, input = c.poll()
    output = process(input)
    p.beginTxn()          void beginTransaction();
    p.send(output)
    p.sendOffsets(offsets)
    p.commitTxn()
}

```

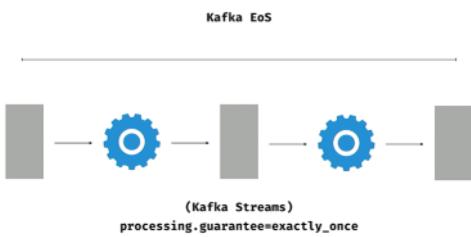
так как кафка не умеет вычищать грязные сообщения то эту настройку надо задать на консьюмере

Изоляция чтения у Consumer

- Consumer isolation.level:
 - **read_committed**: доступны только закоммиченные сообщения
 - **read_uncommitted**: все сообщения доступны

kafka streams тоже поддерживает exactly once

end-to-end eos



exactly once (retention policy)

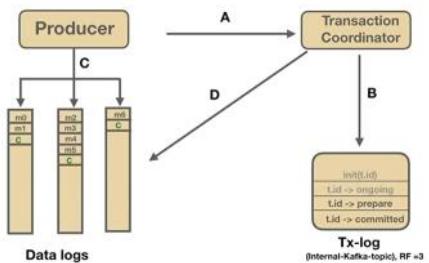
Exactly Once Processing, comes from Kafka Transaction



если вдруг упадет сеть на получении ответа отправки и ответа о приемке то сообщения задублируются
Duplication Source?



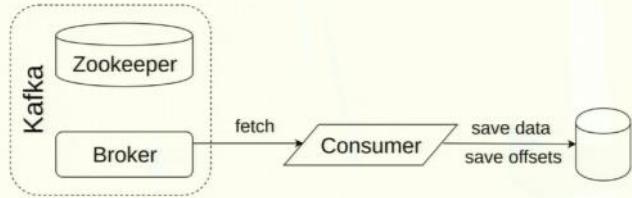
transaction coordinator - это просто внутренняя очередь кафки



Consumer:
isolation.level =
read_committed

как сделать exactly once при сохранении данные в нетранзакционную БД типа HDFS
если при поднятии консьюмера мы обнаружим необработанные файлы в папке то мы удалим уже перемещенные(очевидно что не все переместились) в рамках предыдущей транзакции откатить операцию - способ goblin от linkedin

- Сохранение вне Kafka (возможность exactly once)



Сохранение в HDFS (exactly once)

1. hdfs dfs -mv /tmp/file1 /logs/file
2. hdfs dfs -mv /tmp/file2 /logs/file
3. hdfs dfs -mv /tmp/offsets /runtime/offsets

(1) `PROCESSING_GUARANTEE_CONFIG=EXACTLY_ONCE` настройки stream-app

So...

How to do exactly once in Kafka Streams?



- One additional line of code, the rest is the same!

```
Properties props = new Properties();
...
props.put(StreamsConfig.PROCESSING_GUARANTEE_CONFIG, StreamsConfig.EXACTLY_ONCE);
...
KafkaStreams streams = new KafkaStreams(builder, props);
```

- Currently, **Kafka Streams is the only library that has implemented this feature, but it's possible that Spark, Flink and other frameworks implement it in the future too.**
- What's the trade-off?
 - Results are published in transactions, which might incur a small latency
 - You fine tune that setting using `commit.interval.ms`

```
BankBalanceExactlyOnceApp main()
-----
public static void main(String[] args) {
    Properties config = new Properties();

    config.put(StreamsConfig.APPLICATION_ID_CONFIG, "bank-balance-application");
    config.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "127.0.0.1:9092");
    config.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");

    // we disable the cache to demonstrate all the "steps" involved in the transformation - not recommended in prod
    config.put(StreamsConfig.CACHE_MAX_BYTES_BUFFERING_CONFIG, "0");

    // Exactly once processing!
    config.put(StreamsConfig.PROCESSING_GUARANTEE_CONFIG, StreamsConfig.EXACTLY_ONCE);

    final Serializer<JsonNode> jsonSerializer = new JsonSerializer();
    final Deserializer<JsonNode> jsonDeserializer = new JsonDeserializer();
    final Serde<JsonNode> jsonSerde = Serdes.serdeFrom(jsonSerializer, jsonDeserializer);
```

(2) `ENABLE_IDEMPOTENCE_CONFIG=true` настройки продьюсера

особенно если `RETRIES_CONFIG>1` то значит кафка будет отправлять сообщения повторно и они потенциально могут задублироваться

The screenshot shows the IntelliJ IDEA interface with the following details:

- Project Structure:** The left sidebar shows the project structure for "kafka-streams-course" with "bank-balance-exactly-once" selected. It includes files like .idea, src, main, java, com.github.simplesteph, simplesteph, udemy, kafka, streams, and BankTransactionsProducer.java.
- Code Editor:** The right pane displays the Java code for "BankTransactionsProducer". The code initializes a Kafka producer with specific configuration properties, sends transactions to a topic, and handles exceptions.
- Toolbars and Status Bar:** The bottom of the screen has toolbars for TODO, Version Control, and Terminal. The status bar at the bottom right shows "Event Log".

```
BankTransactionsProducer(main())
1 package com.github.simplesteph.udemy.kafka.streams;
2
3 import ...
4
5 public class BankTransactionsProducer {
6     public static void main(String[] args) {
7         Properties properties = new Properties();
8
9             // kafka bootstrap server
10            properties.setProperty(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "127.0.0.1:9092");
11            properties.setProperty(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class
12                .getName());
13            properties.setProperty(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class
14                .getName());
15            // producer acks
16            properties.setProperty(ProducerConfig.ACKS_CONFIG, "all"); // strongest producing guarantee
17            properties.setProperty(ProducerConfig.RETRIES_CONFIG, "3");
18            properties.setProperty(ProducerConfig.LINGER_MS_CONFIG, "1");
19            // leverage idempotent producer from Kafka 0.11 !
20            properties.setProperty(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, "true"); // ensure we don't
21            push duplicates
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44 }
```

Семантика «Максимум один раз»

```
c.subscribe(inputTopics)
while (true) {
    input = c.poll()
    output = process(input)
    p.send(output)
    c.commitOffsets()
}
```

Input	0	1	2	3	4	5	6	7
Output	0	1	1	2	2	3	4	
Committed Offsets	1	2	3	4				

```
while (true) {
    data = c.poll()
    p.send(data)
    c.commitOffsets()
}
```



```
while (true) {
    data = c.poll()
    p.send(data)
    c.commitOffsets()
}
```

после этого боя
продолжаю работу

Input

0	1	2	3	4	5	6	7
---	---	---	---	---	---	---	---

Output

0	1	1	2	2	3	4	3
---	---	---	---	---	---	---	---

**Committed
Offsets**

1	2	3	4				
---	---	---	---	--	--	--	--

```
while (true) {  
    data = c.poll()  
    p.send(data)  
    c.commitOffsets()  
}
```

```
while (true) {  
    data = c.poll()  
    p.send(data)  
    c.commitOffsets()  
}
```

- Дубликаты

- Повторы продюсера

- Сбой перед записью смещений (offset)

- Зомби



чтобы избежать проблем сначала комитим оффсет а затем записываем результат (зато сообщения не будут дублироваться)

Input

0	1	2	3	4	5	6	7
---	---	---	---	---	---	---	---

Output

0	3	5	4				
---	---	---	---	--	--	--	--

**Committed
Offsets**

1	2	3	4	5	6		
---	---	---	---	---	---	--	--

- Потери сообщений

- Повторы продюсера не разрешены

- Сбои перед тем как записываем данные

- Зомби могут повлиять на порядок сообщений

Слабости семантики

- Повторные попытки продюсера небезопасны
- Данные не пишутся автоматично с их смещениями (offsets)
- Нет защиты от зомби



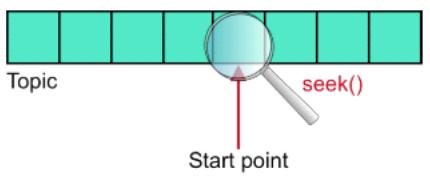
How does Kafka 0.11 solve the problem?

- Without getting in too much engineering details:
 - The producers are now **idempotent** (if the same message is sent twice or more due to retries, Kafka will make sure to only keep one copy of it).
 - You can write multiple messages to different Kafka topics as part of one transaction (either all are written, or none is written). This is a new advanced API
- To achieve this they had to change some logic and the internal message format in Kafka 0.11, therefore only this broker and client version can achieve it.
- You will only have to use libraries that implement the new API
- **Like Kafka Streams!**

вариант самодельного EOS на самодельном хранилище оффсетов

16 декабря 2020 г. 17:36

Storing Offsets Outside of Kafka

Default	Wanted	Consequence
Offset stored in 	Store offset externally  	Use <code>seek()</code> to position consumer 

Topic `--consumer_offsets`

- By default, Kafka stores offsets in a special Topic
 - Called `--consumer_offsets`
- In some cases, you may want to store offsets outside of Kafka
 - For example, in a database table
- If you do this, you can read the value and then use `seek()` to move to the correct position when your application launches

Storing offsets outside of Kafka can in some cases allow exactly-once processing to be implemented (e.g., when working with a transactional external resource like an RDBMS).