Chapter 18: Transactions

When discussing event stream processing application, one topic of conversation that invariably comes up is that of delivery guarantees.

To recount our journey thus far, Chapter 3: Architecture and Core Concepts has introduced the concept of delivery semantics — distinguishing between *at-most-once* and *at-least-once* models and how either can easily be achieved in a Kafka consumer by changing the relative order of processing a record and confirming its offsets.

We subsequently looked at the notions of *idempotence* and *exactly-once* delivery in Chapter 6: Design Considerations — namely, the role of the client application in ensuring that the effects of processing a record are not repeated if the same record were to be consumed multiple times.

Finally, Chapter 10: Client Configuration demonstrated the use of the `enable.idempotence` producer property to ensure that records are not persisted in duplicate or out-of-order in the face of intermittent errors.

This chapter looks at one last control made available by Kafka — *transactions*. Transactions fill certain gaps of idempotent consumers with respect to the side-effects of record processing, and *enable end-to-end, exactly-once transactional semantics across a series of loosely-coupled stream processing stages*.

Preamble

Before we get too involved into unravelling the joys of transactional event processing, it is worth taking a moment to contemplate the extent of the work that has gone into its design and construction, and acknowledge the colossal efforts of numerous contributors who have made this possible. The KIP comprised around 60 individual work items and was planned for over three years. The final design was subject to a nine-month period of public scrutiny and evolved substantially as a result of community feedback. The design was one of the most profound changes to Kafka since its public release, delivering in excess of 15,000 lines of unit tests alone. Crucially, the performance overhead of transactions was kept to a minimum — subtracting an average of three to five per cent from the throughput of an equivalent non-transactional producer.

Transactions arrived in release 0.11.0.0, as part of a much larger KIP-98[46]. The significance of this may not be immediately apparent, but the reader has already witnessed this KIP in action in Chapter 10:

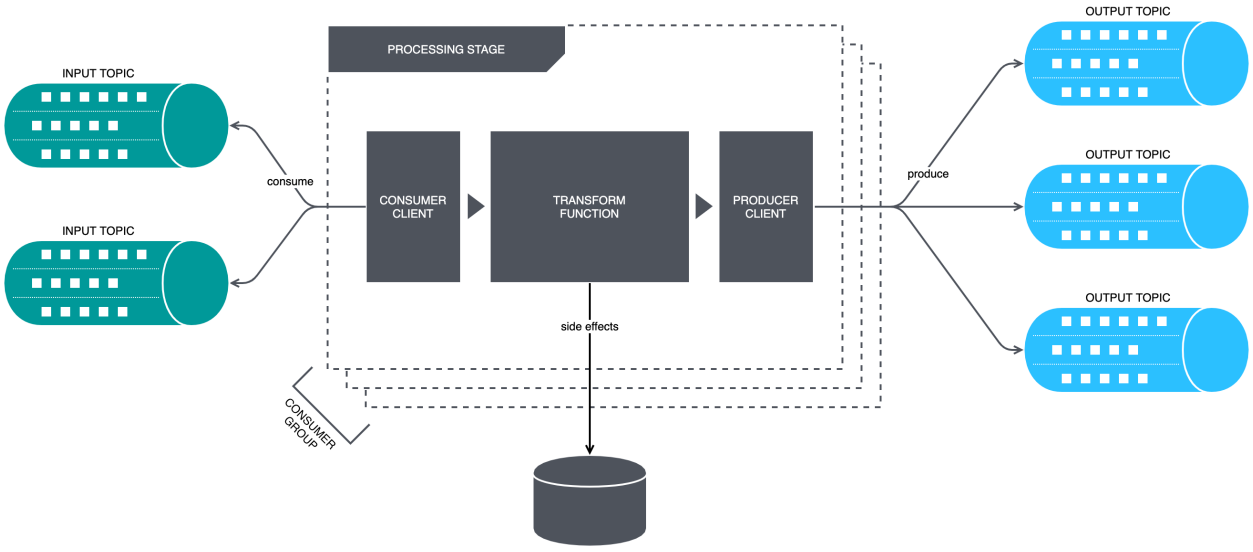
[46] https://cwiki.apache.org/confluence/x/ASD8Aw

Client Configuration, and most of our examples have, in fact, utilised the capabilities introduced by this KIP — namely, the `enable.idempotence` property. That's correct, both the *idempotent producer* and *transactional messaging* features are highly related and share a great deal in common. So in effect, the forthcoming study of transactional messaging might be seen as closing off the discussions started many chapters ago.

The rationale behind transactions

As it has been extensively discussed in the introductory chapters, the role of Kafka as an Event Streaming Platform is to facilitate the distribution and processing of events within a broader, Event-Driven Architecture.

Event streaming systems can be visualised as a set of loosely coupled actors that form a directed acyclic graph (DAG), where the nodes of the graph are processes that interact with Kafka topics, and the edges are the topics themselves. The processes, also called *stages*, may act either as a producer or a consumer, or a combination of both *vis à vis* one or more topics and client instances. A stage may also interact with other resources outside of Kafka, such as databases, APIs, and so forth. A reference model of a single stream processing stage and its neighbouring topics is depicted in the diagram below.



Stream processing stage

Recall the discussion on exactly-once delivery in Chapter 6: Design Considerations. To quote from the chapter:

> To achieve the coveted *exactly-once* semantics, consumers in event streaming applications must be *idempotent*. In other words, processing the same record repeatedly should have no net effect on the consumer ecosystem. If a record has no additive effects, the consumer is inherently idempotent. [...] Otherwise, the consumer must check whether a record

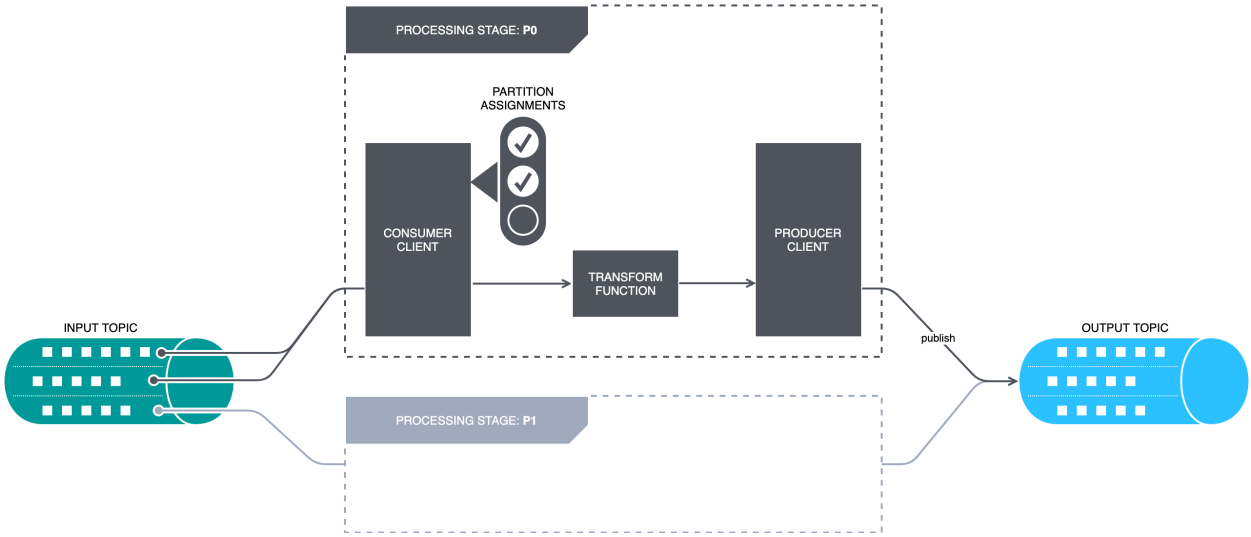
has already been processed, and to what extent, prior to processing the record. *The combination of at-least-once delivery and consumer idempotence collectively leads to exactly-once semantics.*

The problem — duplicate records

The ability to check and selectively carry out an action works well with databases, given that just about every database in existence supports the notion of a secondary index. (Put simply, a way of efficiently accessing records by means of some piece of information other than a primary key.) Kafka, although often likened to a database, does not support user-defined secondary indexes. The primary index of a record is the topic-partition-offset triple; in addition, Kafka allows us to index records by their timestamp for convenience. What Kafka lacks is the ability to nominate some attribute of a record that may be used to efficiently retrieve the record, or at least ascertain the record's presence.

Speaking of databases, Kafka was originally designed as a durable record store. Kafka's replication mechanism ensures that, with the appropriate configuration in place, once a record is acknowledged on the broker, that record is persisted in a manner that survives a broad range of failures. Aside from the *durability* guarantee, Kafka did not address any of the other attributes of *ACID* (Atomicity, Consistency, Isolation, Durability). We will touch on ACID again shortly.

Without a secondary index, how can a processing stage ensure that a record has not been published, for some input record that might be processed repeatedly as a result of at-least-once delivery? To illustrate the problem, consider a simple stage that transforms an input stream to an output stream using a straightforward mapping function:



Simple stream processing example

To further distil the problem, let's assume that a single 'input' record trivially maps to one 'output' record. Assume the input topic is a set of integers and the transform function simply squares each integer before forwarding it on. Contrived as it may be, this example serves to demonstrate the problem of attempting to apply idempotence to a storage medium that does not inherently support it. The topic topology and the transform function are not important; what is important is the fundamental property that there is exactly one output record corresponding to a processed input record, despite the prevalent at-least-once behaviour. And this property must be upheld in the presence of multiple instances of the transformation stage, operating within an encompassing consumer group.

Under a conventional *consume-transform-produce* model, the consumer side of a stage will read a record from the input topic, apply a transformation, and publish a corresponding record on the output topic, via a dedicated producer client. Once it receives an acknowledgement from the leader that the record has been durably persisted, it will commit the offsets of the input record. For simplicity, assume the producer is operating with idempotence enabled, being the recommended mode for most applications.

Consider the types of failures one might encounter with the model above. Putting aside buggy and misbehaving applications (and more complex Byzantine faults), at minimum, we would need to account for the failure of brokers, the abrupt termination of the transform process, and network connectivity issues. On the consumer side, network and broker failure can be dealt with by the retry mechanism embedded into `Consumer.poll()`. On the producer end, the use of idempotence will deal with broker and network failures, provided the failure does not last longer than the delivery timeout. This leaves us with the most taxing failure scenario: process failure. The outcome will depend on the exact point at which the process fails. Consider the options:

1. Failure before consumption.
2. Failure after consumption, but before the transform.
3. Failure following the transform, but before publishing.
4. Failure after publishing, but before confirming the offsets.
5. Failure after committing the offsets.

Case #1 is benign: nothing has yet happened and so the recovery is trivial. Case #2 is like #1, because Kafka (unlike its message queuing counterparts) does not update the state of the topic after a record has been read. Assuming the transform operation is a pure function that entails no state change of its own, case #3 is also a non-issue; the recovering process will replay the consumption and transform steps. Case #4 is where it gets interesting, being the first point where the failure occurs after a state change. The recovering process, having no awareness of the prior publish step will repeat all steps, resulting in two output records for the same input record. Finally, case #5 is again benign, as the committing of the offsets marks the completion of the process cycle.

Analysis of the above shows that, in the worst case, multiple output records may be written for a single input record; in the best case, one output record will be written. Assuming the input record is eventually processed, resulting in its offsets committed, at no point after the commit will there

be an observable absence of an output record. And that is the essence of at-least-once delivery. All prior material in this book has consistently guided the reader towards this direction.

We could have easily expanded upon the example above to include multiple input and output topics, and more complex transformation logic with possible side-effects. Fundamentally, this would not have changed the basic failure scenarios.

This basic formula for at-least-once delivery extends to any other stage that might be downstream of the one being considered. Provided an application-level identifier is included in every record that helps the consumers correlate duplicates, and every consumer can behave idempotently with respect to the processed records, the entire processing graph will exhibit exactly-once behaviour in a limited sense. Specifically, the aspects of record processing that relate to databases and idempotent API calls are covered. Conversely, message queues and Kafka topics that don't provide application-level deduplication are excluded from the exactly-once guarantee; duplicates will occur, and it is something we have come to accept.

Idempotence requires collaboration not only from the consuming process, but from all downstream resources that said process manipulates.

Presumably, message-oriented middleware is a transport mechanism that facilitates the distribution of messages between endpoints or the replication of events, whichever paradigm one chooses to accept; it is not the definitive source of truth, in that it does not attempt to replace a database. It would be unacceptable for a database to return multiple rows for the same entity. But Kafka, being part of the transport apparatus, somehow gets away with this — relying on the application to perform deduplication.

What if Kafka was used as primary storage — the proverbial 'source of truth'; for example, acting in the role of an event store in an event sourcing system? It would hardly be acceptable to have two records representing the same logical event. This is not to say that a consumer could not be designed to cope with duplicates, but it would be fair to assume that duplicates would be undesirable in most cases — irritating, to put it mildly.

The solution — transactions

The transactional messaging capability introduced in release 0.11.0.0 strengthens Kafka's delivery semantics. Namely, it introduces limited *atomicity*, *consistency* and *isolation* guarantees on top of the existing *durability* pledge, for a combined *ACID* experience that is characteristic of relational databases (and occasionally of NoSQL). Specifically —

- **Atomicity** — ensures that, for a group of records published within an encompassing transaction scope, either all records are *visibly* persisted to their respective logs, or none are persisted. In other words, the transaction either succeeds or fails as a whole. The logs, in this case, are not limited to a single partition; transactions may span multiple topics and partitions.

- **Consistency** — a logical corollary of atomicity, consistency ensures that the cluster transitions from one valid state to another, without violating any invariants in between. In our context, consistency speaks to the correlation between records on input and output topics of a processing stage; an input record must result in an output record once consumed, or it must not be consumed at all.
- **Isolation** — ensures that concurrent execution of transactions leaves the system in the same state that would have been obtained if the transactions were executed sequentially. In simple terms, the effects of a transaction cannot be externally visible until it commits. (In reality, the visibility of in-flight transactions depends on the configured isolation level of the consumer.)

Transactions under the hood

Role of the transaction coordinator

Chapter 10: Client Configuration had briefly touched on some of the inner workings of idempotent producers. At the heart of the implementation is a unique *producer ID* (PID) that is assigned by a *transaction coordinator* for the duration of the producer's session. A transaction coordinator is a module running within a broker process, servicing idempotent and transactional producers — akin to the group coordinator used by consumers to manage group membership. The transaction coordinator is responsible for issuing PIDs and managing transaction state. The producer starts by issuing an `InitProducerId` request to the appropriate transaction coordinator (there may be multiple coordinators); the latter replies with a unique PID that is valid for the duration of the producer's session.

Transactional messaging builds upon this infrastructure by increasing the lifetime of a producer's PID such that it survives a single producer session. This is achieved by specifying an optional `transactional.id` property on the producer. If the property is not set, the producer will be issued a new PID on each request; otherwise, the transaction coordinator will associate the issued PID with the transactional ID and maintain this association for the duration specified in the `transactional.id.expiration.ms` broker property, which defaults to `604800000` (one week). The *transactional ID* ⟶ *PID* mapping includes the epoch corresponding to the point when the association was last updated. The epoch acts as a fencing mechanism, blocking *zombie* processes that have been displaced by a newer PID assignment. (A zombie producer might attempt to use their PID to manipulate an in-flight transaction or initiate a new one; however, they will act using an older epoch number, which gives them away.) A `ProducerFencedException` is thrown when the producer attempts to manipulate a transaction that has been fenced off.

The load balancing of transaction coordinator duties is done by assigning the partitions of the internal `__transaction_state` topic to the brokers in the cluster, such that the assigned owner of any given partition becomes the notional coordinator for that partition's index. The identification of a suitable transaction coordinator is performed by hashing the transactional ID, modulo the partition index — arriving at the coordinator in charge of the partition. By piggybacking on an existing leadership election process, Kafka conveniently avoids yet another arbitration process to elect transaction coordinators. Also, the management of transactions is divided approximately equally across all the brokers in the cluster, which allows the transaction throughput to scale by increasing the number of brokers.

Producer API enhancements

Transactional messaging adds several methods to the `Producer` API, namely:

- `initTransactions()`: Initialises the transactional subsystem by obtaining the PID and epoch number, and finalising all prior transactions for the given transactional ID with the transaction coordinator. (And in doing so, fencing any zombies for the previous epoch.) This method should only be called once for a producer session, and must be called prior to any of the methods below. Once initialised, the producer will be transitioned to the `READY` state.
- `beginTransaction()`: Demarcates the start of transaction scope on the producer. Performs a series of local checks to ensure that the client is operating in transactional mode and transitions the producer to the `IN_TRANSACTION` state. This method must be invoked before any of the methods below.
- `sendOffsetsToTransaction()`: Incorporates the given set of consumer-side per-topic-partition offsets into the scope of the current transaction and forwards them to the group coordinator of the supplied consumer group. The offsets will come into effect when the transaction subsequently commits.
- `commitTransaction()`: Flushes any unsent records and commits the current transaction by instructing the transaction coordinator to do so. This places the producer into the `COMMITTING_TRANSACTION` state, after which it is not possible to invoke any other method. (It is possible to retry `commitTransaction()` in this state.)
- `abortTransaction()`: Discards any pending records for the current transaction and instructs the transaction coordinator to abort the transaction. This places the producer into the `ABORTING_TRANSACTION` state, from which point no other transactional methods may be invoked (except for retrying `abortTransaction()`).

Transactional messaging in Kafka is largely a producer-side concern. This makes sense, as *transactions control the atomicity of write operations*. The reader might recall from Chapter 3: Architecture and Core Concepts, Kafka employs a recursive approach to managing committed offsets, utilising the internal `__consumer_offsets` topic to persist and track offsets for consumer groups. Because the committing of an offset is modelled as a write, it is necessary for consumers to piggyback on an existing transaction scope managed by a producer. In other words, for a typical processing stage

that comprises both a consumer client and a producer client, the consumer must forward its offsets to the producer, rather than use its native `commitSync()` or `commitAsync()` API. On top of this, offset auto-commit must be disabled on the consumer; *the use of transactions demands manual offset committing*. There is one other aspect of transactional behaviour that must be accounted for — *isolation* — which will be covered shortly.

With transactions enabled, each iteration of a typical *consume-transform-produce* loop resembles the following:

```
producer.beginTransaction();
try {
  // process records ...
  producer.send(...);
  // ...
  producer.sendOffsetsToTransaction(...);
  producer.commitTransaction();
} catch (KafkaException e) {
  producer.abortTransaction();
  throw e;
}
```

Assigning a transactional ID

And now we arrive at the crux of the challenge — the choice of the transactional ID. This is widely considered as among the most confusing and perplexing aspects of transaction management in Kafka. Overwhelmingly, the questions in various forums that relate to transactions are posed around the transactional ID.

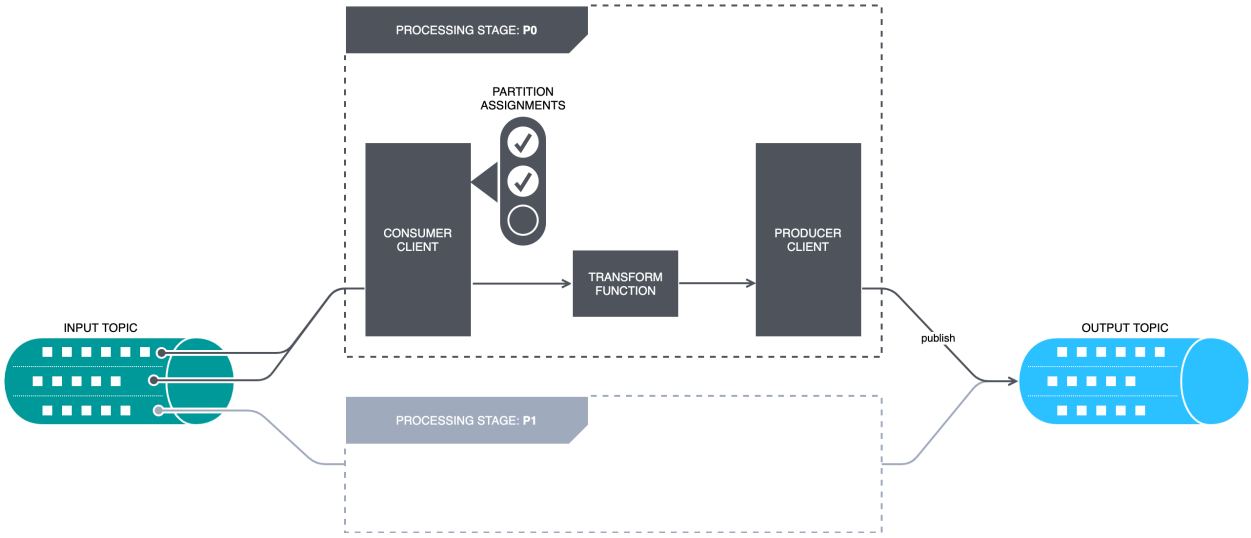
The reason this is so hard is that the transactional ID must survive producer sessions. It is also used as a fencing mechanism, which means that there must only be one (legitimate) producer using the transactional ID at any given time. This is further complicated by the fact that the number of processes in a stage is not a constant, and may scale in and out depending on load, as well as factors outside of our control, such as process restarts, network partitions, and so on.

Before revealing the recommended transactional ID assignment scheme, it is instructive to explore a handful of options to appreciate the complexity of the problem.

Starting with the simplest — a common transactional ID shared by all processes. We can dismiss this option hastily, as it creates an irreconcilable fencing issue for all but the most recent processes that present the shared transactional ID. As the assignment of PIDs is associated with an epoch number, the last producer to call `initTransactions()` will acquire the highest epoch number — turning its peers into zombies (from the perspective of the transaction coordinator).

Crossing to the opposite extreme — a random transactional ID with sufficient collision-resistance to guarantee its uniqueness. One might use UUIDs (versions 3, 4 and 5) or some other unique quantity.

This clearly avoids the issue of fencing legitimate peer processes, but in doing so it effectively disables zombie fencing altogether. Consider the implications by reverting to our earlier example of a simple processing stage, surrounded by a pair of input and output topics. Assume two processes *P0* and *P1* contend for the assignment of three partitions in the input topic *I0*, *I1* and *I2* — transforming the input and publishing records to some partitions in the output topic. The output partitions, as will shortly become apparent, are irrelevant — it is only the input partitions that matter for fencing.



Simple stream processing example

Initially, *P0* is assigned *I0* and *I1*, and *P1* is assigned *I2*. Both processes are operating with random transactional IDs. At some point, an issue on *P1* blocks its progress through *I2*, eventually resulting in failure detection on the group coordinator, followed by partition reassignment. Thereafter, *P0* becomes the assignee of all partitions in the input topic; *P1* is presumed dead.

What if *P1* had in-flight transactions at the time of reassignment? The transaction coordinator will provide for up to the value of `transactional.id.expiration.ms` to honour a pending transaction. This setting defaults to `3600000` (one hour). During this time, producers will be allowed to write to the affected partitions, but no record that appears after the partial transaction on the affected partitions will be delivered to consumers operating under the `read_committed` assurance level. That amount of downtime on the consumer is generally unacceptable in most stream processing applications. Of course, the expiration time could be wound down from its default value, but one must be careful not to interrupt the normal operation of transactional producers by setting an overly aggressive expiration time. At any rate, some appreciable amount of downtime cannot be avoided.

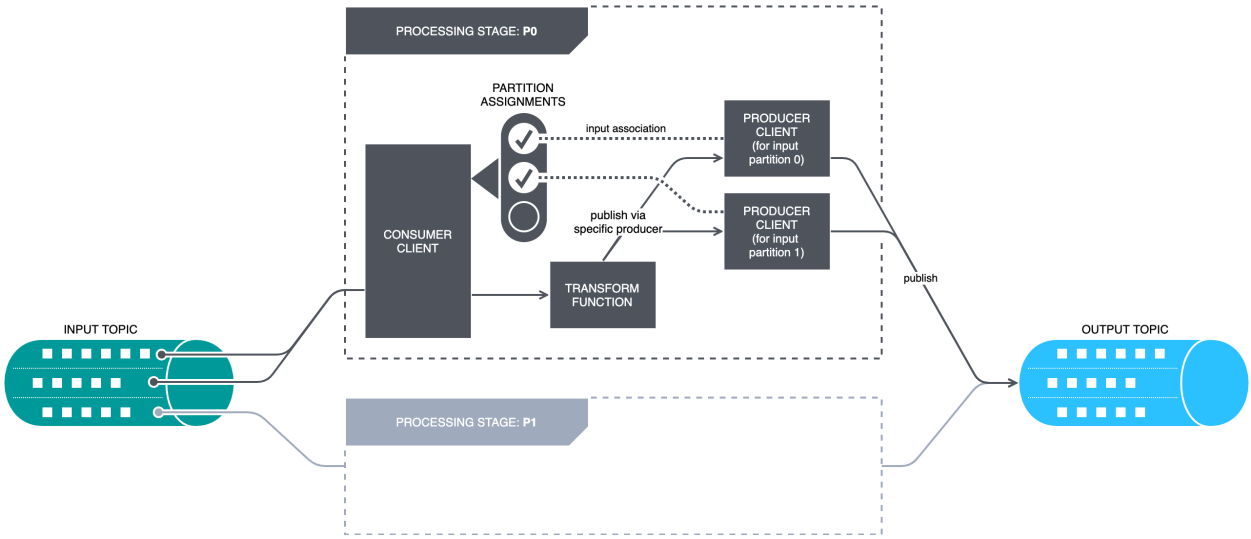
Another problem with this approach is what happens on *P1* if it suddenly resumes operating within the transaction expiry period. Being excluded from its consumer group, it will not be able to consume further records, but any in-flight records (returned from a previous poll) are still fair game from its perspective. It may attempt to publish more records on the output topic within the existing transaction scope — an operation that will be accepted by the transaction coordinator (which knows nothing about the consumer-side status of the process). In the worst case, the output topic will contain duplicates that have been emitted by the two processes, corresponding to the same input

record.

We can try our hand at various other transactional assignment schemes, but the result will invariably be the same — either undue fencing of legitimate producers or bitter defeat at the hands of zombies.

The very point of fencing is to eliminate these sorts of scenarios, but for fencing to function correctly, the producer's session must somehow relate back to the input source. This is where a confession must be made: the issue isn't just with the transactional ID assignment scheme, but with the internal architecture of the processing stage. In the original example, we used a pair of clients — a consumer and a producer — in a back-to-back arrangement. A process would read from any of the assigned partitions in the input set, via a single consumer instance. It would then publish a record (or multiple records) to an arbitrary output partition via single, long-lived producer instance.

Before continuing with the transactional ID conundrum, we need to rework the internal architecture of the processing stage. Let the singleton consumer be, but replace the singleton producer with a collection of producers — one for each assigned partition in the input set. The assignment of partitions is subject to change, which can be accommodated by registering a `ConsumerRebalanceListener`. When a partition is assigned, create and initialise a corresponding producer instance; when a partition is revoked, dispose of the producer instance. When publishing an output record, always use the producer instance that corresponds to the partition index of the input record. The diagram below illustrates this internal topological arrangement.



Pinned producers

The number of partitions in the input topic(s) might be in the hundreds or thousands; spawning a producer instance for each conceivable topic-partition pair is wasteful, entailing a potentially significant connection establishment overhead, in addition to utilising memory (for buffers), file handles and OS-level threads. As such, reducing the producer clients to the set of assigned partitions is the preferred approach. Furthermore, the producer instances may be lazily initialised — amortising the overhead from the point of partition assignment to the points of the first read from each input topic-partition.

Now for the reveal: the transactional ID for each producer instance is derived by concatenating the corresponding input topic and partition index pair.

To understand why this approach works, consider the earlier example with *P0* and *P1*, where *I2* was transferred from *P1* to *P0* as a result of a perceived failure. For simplicity, assume the input and output topics are named `tx-input` and `tx-output`, respectively. Under the revised architecture, *P0* starts with one consumer and two producers with transactional IDs `tx-input-0` and `tx-input-1`. *P1* has one transactional producer — `tx-input-2`. Upon reassignment, *P0* has `tx-input-0`, `tx-input-1` and `tx-input-2`. *P1 thinks* it has `tx-input-2`.

When *P0* acquires *I2* and instantiates a producer with the transactional ID `tx-input-2`, the call to `initTransactions()` will result in the finalisation of any pending transaction state with the transaction coordinator. The coordinator will either commit or roll back the transaction, depending on the state of its transaction log, recorded in the internal `__transaction_state` topic. If the outgoing producer was able to fire off a commit request prior to revocation, the transaction will be committed by writing `COMMITTED` control messages to the affected partitions in `tx-output`. Conversely, if the outgoing producer aborted the transaction or otherwise failed to explicitly end the transaction prior to revocation, the transaction will be forcibly rolled back by writing `ABORTED` control messages to the affected partitions. At any rate, any pending transaction will be finalised before *P0* is permitted to publish new records via its `tx-input-2` producer. As part of this ceremony, the epoch number is incremented.

When a transaction is rolled back, Kafka does not delete records from the affected partitions. Doing so would hardly sit well with the notion of an append-only ledger. Instead, by writing an `ABORTED` control message to each of the affected partitions, the coordinator signals to a transactional consumer that the records should not be delivered to the application.

In the meantime, *P1* might attempt to complete its transaction, acting under the `tx-input-2` transactional ID. It may add more records to the transaction scope, which is accompanied by an `AddPartitionsToTxnRequest` to the transaction coordinator. Alternatively, it may attempt to commit the transaction by sending an `EndTxnRequest`. Both requests include a mandatory epoch number. Because the initialisation of the producer on *P0* had incremented the epoch number on the transaction coordinator, all further actions under the lapsed epoch number will be disallowed by the coordinator.

To summarise: by pinning a producer client instance to the input topic-partition, we are capturing the causality among input and output records within the identity of the producer — its PID, by way of a derived transactional ID. As the partition assignment changes on the group coordinator, the causal relationship is carried forward to the new assignee; the outgoing assignee will fail if it attempts to publish a record under the same identity.

Transactional consumers

It was stated earlier that transactional messaging is largely a producer-side concern. Largely, but not wholly — consumers still have a role to play in implementing the *isolation* property of transactions.

Specifically, when records are published within transaction scope, these records are persisted to their target partitions even before the transaction commits (or aborts, for that matter). A transaction might span multiple records across disparate topics. For transactions to retain the isolation property, it is essential that the consumer ecosystem collaborates with the producer and the transaction coordinator. This is accomplished by terminating a sequence of records that form part of a transaction by a control marker — for every partition that is featured within the transaction scope.

All consumers above version 0.11.0.0 understand the notion of control markers; however, the way a consumer reacts to a control marker varies depending on the consumer's `isolation.level` setting. The default isolation level is `read_uncommitted` — meaning the consumer will disregard the control markers and deliver the records as-is to the polling application. This includes records that are part of in-flight transactions as well as those records that were part of an aborted transaction.

To enable transactional semantics on a consumer, the `isolation.level` must be set to `read_committed`. Within this mode, the behaviour of the producer changes with respect to the end offset. Normally, the end offsets are equivalent to the high-water mark — the offset immediately following the last successfully replicated record. A transactional consumer replaces its notion of end offsets with the *Last Stable Offset* (LSO) — the minimum of the high-water mark and the smallest offset of any open transaction. Under the constraint of the LSO, a producer will not be allowed to enter a region in the log that contains an open transaction — not until that transaction commits or aborts. In the former, the contents of the transaction will be delivered in the result of `Consumer.poll()`. In the latter, the records will be silently discarded.

Having acquired an understanding of how consumers are bound in their progression through the assigned partitions, it is easy to see why zombie fencing is an essential element of transactional messaging. Without fencing measures in place, a producer stuck in an open transaction will impede the advancement of the LSO for up to `transactional.id.expiration.ms` — impacting all downstream consumers running with `isolation.level=read_committed`.

Simple stream processing example

Having set the theoretical foundations for transactional messaging, it is time to implement the above scenario. To get started, we need a pair of topics — `tx-input` and `tx-output`:

```
$KAFKA_HOME/bin/kafka-topics.sh --bootstrap-server localhost:9092 \
    --create --topic tx-input --partitions 3 --replication-factor 1


$KAFKA_HOME/bin/kafka-topics.sh --bootstrap-server localhost:9092 \
    --create --topic tx-output --partitions 3 --replication-factor 1
```

The replication factor was kept to the allowed minimum to support our single-node test broker. Three partitions are used on either side to demonstrate the affinity between producer instances and

the input topic-partition. For simplicity, we are not using authentication or authorization in this example.

The complete source code listing for this example is available at github.com/ekoutanov/effectivekafka[47] in the `src/main/java/effectivekafka/transaction` directory.

The listing below depicts a simple transformation stage that takes an integer as input and squares it before publishing the resulting value in an output record.

```
import static java.lang.System.*;

import java.time.*;
import java.util.*;

import org.apache.kafka.clients.consumer.*;
import org.apache.kafka.clients.producer.*;
import org.apache.kafka.common.*;
import org.apache.kafka.common.serialization.*;

public final class TransformStage {
  public static void main(String[] args) {
    final var inputTopic = "tx-input";
    final var outputTopic = "tx-output";
    final var groupId = "transform-stage";

    final Map<String, Object> producerBaseConfig =
        Map.of(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
               "localhost:9092",
               ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
               StringSerializer.class.getName(),
               ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
               IntegerSerializer.class.getName(),
               ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG,
               true);

    final Map<String, Object> consumerConfig =
        Map.of(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
               "localhost:9092",
               ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
               StringDeserializer.class.getName(),
```

[47] https://github.com/ekoutanov/effectivekafka/tree/master/src/main/java/effectivekafka/transaction

```
            ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
            IntegerDeserializer.class.getName(),
            ConsumerConfig.GROUP_ID_CONFIG,
            groupId,
            ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
            "earliest",
            ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG,
            false);

final var producers = new PinnedProducers(producerBaseConfig);

try (var consumer =
     new KafkaConsumer<String, Integer>(consumerConfig)) {
  consumer.subscribe(Set.of(inputTopic),
                     producers.rebalanceListener());

  while (true) {
    final var inRecs = consumer.poll(Duration.ofMillis(100));

    // read the records, transforming their values
    for (var inRec : inRecs) {
      final var inKey = inRec.key();
      final var inValue = inRec.value();
      out.format("Got record with key %s, value %d%n",
                 inKey, inValue);

      // prepare the output record
      final var outValue = inValue * inValue;
      final var outRec =
          new ProducerRecord<>(outputTopic, inKey, outValue);
      final var topicPartition =
          new TopicPartition(inRec.topic(), inRec.partition());

      // acquire producer for the input topic-partition
      final var producer = producers.get(topicPartition);

      // transactionally publish record and commit input offsets
      producer.beginTransaction();
      try {
        producer.send(outRec);
        final var nextOffset =
            new OffsetAndMetadata(inRec.offset() + 1);
        final var offsets = Map.of(topicPartition, nextOffset);
```

```
            producer.sendOffsetsToTransaction(offsets, groupId);
            producer.commitTransaction();
          } catch (KafkaException e) {
            producer.abortTransaction();
            throw e;
          }
        }
      }
    }

    /**
     *  Mapping of producers to input topic-partitions.
     */
    private static class PinnedProducers {
      final Map<String, Object> baseConfig;

      final Map<String, Producer<String, Integer>>
          producers = new HashMap<>();

      PinnedProducers(Map<String, Object> baseConfig) {
        this.baseConfig = baseConfig;
      }

      ConsumerRebalanceListener rebalanceListener() {
        return new ConsumerRebalanceListener() {
          @Override
          public void onPartitionsRevoked
              (Collection<TopicPartition> partitions) {
            for (var topicPartition : partitions) {
              out.format("Revoked %s%n", topicPartition);
              disposeProducer(getTransactionalId(topicPartition));
            }
          }

          @Override
          public void onPartitionsAssigned
              (Collection<TopicPartition> partitions) {
            for (var topicPartition : partitions) {
              out.format("Assigned %s%n", topicPartition);
              createProducer(getTransactionalId(topicPartition));
            }
          }
```

```
      };
    }

    Producer<String, Integer> get(TopicPartition topicPartition) {
      final var transactionalId = getTransactionalId(topicPartition);
      final var producer = producers.get(transactionalId);
      Objects.requireNonNull(producer,
                             "No such producer: " + transactionalId);
      return producer;
    }

    String getTransactionalId(TopicPartition topicPartition) {
      return topicPartition.topic()
          + "-" + topicPartition.partition();
    }

    void createProducer(String transactionalId) {
      if (producers.containsKey(transactionalId))
        throw new IllegalStateException("Producer already exists: "
            + transactionalId);

      final var config = new HashMap<>(baseConfig);
      config.put(ProducerConfig.TRANSACTIONAL_ID_CONFIG,
                 transactionalId);
      final var producer =
          new KafkaProducer<String, Integer>(config);
      producers.put(transactionalId, producer);
      producer.initTransactions();
    }

    void disposeProducer(String transactionalId) {
      final var producer = producers.remove(transactionalId);
      Objects.requireNonNull(producer,
                             "No such producer: " + transactionalId);
      producer.close();
    }
  }
}
```

The example employs a single consumer client and multiple producers. The producer configuration is named `producerBaseConfig` because each producer's configuration will be slightly different — having a distinct `transactional.id` setting. The management of producers is conveniently delegated to the `PinnedProducers` class, which keeps a mapping of transactional IDs to producer instances. The

producer lifecycle is managed by exposing a `ConsumerRebalanceListener`, which spawns a producer upon partition assignment and disposes of the producer upon revocation. The `transactional.id` value is derived by concatenating the input topic name with the partition index, delimited by a hyphen character. For example, `tx-input-2` for partition 2 in the topic named `tx-input`.

Returning to the `main()` method, we have a classic *consume-process-publish* loop. Having consumed a batch of records, the application iterates over the batch and acquires a corresponding pinned producer instance for each record. A transaction is started for each input record, containing the output `send()` call as well as the committing of offsets via the producer. Note, all write operations *must* go via the producer API.

Before we can see this example in action, we need a way of generating input records and viewing the output. This is taken care of by a pair of classes — `InputStage` and `OutputStage`:

```
import static java.lang.System.*;

import java.util.*;

import org.apache.kafka.clients.producer.*;
import org.apache.kafka.common.serialization.*;

public final class InputStage {
  public static void main(String[] args)
      throws InterruptedException {
    final var topic = "tx-input";

    final 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,
               IntegerSerializer.class.getName(),
               ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG,
               true);

    try (var producer = new KafkaProducer<String, Integer>(config)) {
      while (true) {
        final var key = new Date().toString();
        final var value = (int) (Math.random() * 1000);

        out.format("Publishing record with key %s, value %d%n",
                   key, value);
        producer.send(new ProducerRecord<>(topic, key, value));
```

```
        Thread.sleep(500);
      }
    }
  }
}
```

```
import static java.lang.System.*;

import java.time.*;
import java.util.*;

import org.apache.kafka.clients.consumer.*;
import org.apache.kafka.common.serialization.*;

public final class OutputStage {
  public static void main(String[] args) {
    final var topic = "tx-output";
    final var groupId = "output-stage";

    final Map<String, Object> config =
        Map.of(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
               "localhost:9092",
               ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
               StringDeserializer.class.getName(),
               ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
               IntegerDeserializer.class.getName(),
               ConsumerConfig.GROUP_ID_CONFIG,
               groupId,
               ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
               "earliest",
               ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG,
               false,
               ConsumerConfig.ISOLATION_LEVEL_CONFIG,
               "read_committed");

    try (var consumer = new KafkaConsumer<String, Integer>(config)) {
      consumer.subscribe(Set.of(topic));

      while (true) {
        final var records = consumer.poll(Duration.ofMillis(100));
        for (var record : records) {
```

```
            out.format("Got record with key %s, value %d%n",
                       record.key(), record.value());
          }
          consumer.commitAsync();
        }
      }
    }
}
```

Run all three, starting them in any order. Each will do their thing; the `OutputStage` application will display the resulting records. The `TransformStage` app has additional logging in place to show the assignment and revocation of partitions. The logging isn't overly interesting when running a single transform instance, but when multiple processes are launched, it clearly highlights where the revocations of partitions on one consumer correspond to the assignment of partitions on the other.

Limitations

There are several limitations of transactional messaging in Kafka which the reader should be mindful of.

Bound to Kafka resources

Transactional messaging is implemented using a proprietary Kafka protocol that is not interoperable with other persistence systems and messaging middleware. Kafka does not support standard transaction APIs such as XA and JTA.

Cannot span producers

Transactions cannot be used to span multiple producer instances. Transactions are forwarded to specific transaction coordinators, which are assigned the role of managing transactional IDs by way of a hash function. Two producers with different transactional IDs may map to two different coordinators, making cross-producer transactions intractable within the constraints of the current design.

Cannot span clusters

Where the producer and consumer sides of a processing stage connect to different clusters, consumer-side offsets cannot be committed by relaying them through the transaction coordinator, as the latter resides in a different cluster.

May be partially observed by a consumer

As transactions can span multiple partitions, it is possible for any given consumer operating within a consumer group to only witness a subset of the records emitted within a broader transaction, being oblivious to the effects of the transaction that do not affect its assigned partitions. Because partition assignment is balanced among members of a consumer group, the records of one transaction may be subdivided among multiple consumers in a group.

Transactions may straddle multiple log segments in a partition. When the lapsed log segments are eventually deleted as a result of retention constraints, it may appear that parts of the transaction have vanished. In practice, this problem only impacts consumers who have accumulated significant lag, or are starting to read a topic from the beginning.

Records published within a transaction are not accorded any special treatment from the perspective of compaction. A background compaction process may void individual records, leaving a partial set. To be fair, the behaviour of transactional records with respect to retention and compaction is not a limitation of transactions as such, but more of a caveat that warrants awareness.

Incomplete exactly-once semantics

The biggest limitation of transactional messaging with respect to exactly-once processing is that it does nothing to prevent an input record from being handled twice; if the process suffers from a failure mid-stream, an alternate process will take over and may need to deal with partially processed records. This is straightforward if the processing stage embodies a pure function; in other words, it is both deterministic and has no state of its own. On the other hand, if the processing stage must write to a database or invoke a downstream service, prior side effects must be taken into account.

Are transactions over-hyped?

For all the proverbial 'tyre pumping' of the transactional messaging capability, one must question the added complexity of dealing with transactions and pinning producers, in terms of benefits that it provides. Assuming all persistent side effects are idempotent, a competently written *consume-transform-produce* loop provides the requisite exactly-once semantics end-to-end. Of course, this assumes we don't care about duplicates in Kafka, and one might go as far as arguing that the at-least-once tenets of event stream processing already imply some degree of robustness *vis à vis* duplicates, provided there is a reliable way of identifying them.

Transactional messaging eliminates duplicate records in event stream processing graphs. This may be useful when applications rely on Kafka as a primary event store and where dealing with duplicate records may be non-trivial at the application level. The complexity of managing pinned consumers may be encapsulated within a dedicated messaging layer, or a framework. For example, the *Kafka Streams* client library can transparently deal with transactions, relieving the application from having to manage consumers and producers, demarcate transactions and commit offsets. Kafka Streams is a

good fit for a broad range of map-reduce style and simple windowed operations, but understandably, it might not solve all your stream processing needs.

> When the fruits of KIP-98[a] were released to the general public, many an opportunity was exploited to publicise the implications of transparently facilitating exactly-once delivery guarantees at the middleware level. Suggestions were made that the new capability solves one of the most difficult problems in distributed computing, and one that was previously considered impossible.
>
> It was stated in Chapter 6: Design Considerations, that *exactly-once semantics are not possible at the middleware layer without tight-knit collaboration with the application.* This axiom applies to the general case, where record processing entails side effects. It is important to appreciate that this statement holds in spite of Kafka's achievements in the area of transactional messaging. The latter solves a limiting case of the exactly-once problem; specifically, it eliminates duplicate records in a graph of event stream processing stages, where each stage comprises exclusively of pure functions. Once again, there is no 'silver bullet'.
>
> It should be acknowledged that the *exactly-once impossibility* dictum does not take anything away from Kafka; the release of transactional messaging is nonetheless useful in a limited sense. Where the application domain does not fit entirely into the limiting case for which transactional messaging holds, the reader ought to take their own measures in ensuring idempotence across all affected resources.
>
> [a] https://cwiki.apache.org/confluence/x/ASD8Aw

For the majority of event-driven applications, the most useful and practical aspect of transactional messaging is the idempotence guarantee on the producer. This feature utilises the same underlying PID concept and transactional infrastructure, ensuring that records do no arrive out-of-order on the broker within the timeframe allowed by `delivery.timeout.ms`.

We have just emerged from one of the most taxing topics in all of Kafka. Transactional messaging was introduced in version 0.11.0.0, bringing about capabilities to support a combination of producer idempotence and end-to-end exactly-once delivery guarantees.

Transactional messaging improves upon Kafka *durability* guarantee, adding *atomicity*, *consistency* and *isolation* with respect to the production and consumption of records in a graph of stream processing applications. When operating within transaction scope, processing stages can avoid duplicate records in the output topics — ensuring a one-to-one correspondence between the set of input records and the output set.

The challenges of transactional messaging are largely concentrated in the mapping of stage inputs to outputs, by way of a stable transactional ID. The latter identifies a logical producer session spanning multiple physical producer instances. As partitions are reassigned among consumer processes, the transactional ID follows the partition assignment, resuming the processing of records and fencing

prior activity that might still be in effect. Implementing transactional stages requires the application to maintain multiple producer instances that are pinned to the origin of the record, managing the lifecycle of the producers in response to changes in partition assignments.

In its leaner guise, the underlying transactional infrastructure can be used to ensure that the publishing of any given record is idempotent insofar as the queued record will appear exactly-once on its target partition, and in the order prescribed by the producer. (A guarantee that holds within the extent of the delivery timeout, but could be contravened if the record is queued a second time.) This feature has an immediate and practical benefit, and is recommended to be enabled on all producers — not just stream processing graphs.