

# KIP-691: Enhance Transactional Producer Exception Handling

- [Status](#)
- [Motivation](#)
- [Proposed Changes](#)
  - [Unify Wrapped KafkaException](#)
  - [Callback Exception Improvement](#)
  - [Streams Side Change](#)
  - [Documentation change](#)
- [Compatibility, Deprecation, and Migration Plan](#)

## Status


Current state: *Accepted*


Discussion thread: <https://lists.apache.org/list?dev@kafka.apache.org:2020-12:KIP-691>


JIRA:


 Unable to render Jira issues macro, execution error.


Related JIRA:


 Unable to render Jira issues macro, execution error.

 Unable to render Jira issues macro, execution error.

 Unable to render Jira issues macro, execution error.

 Unable to render Jira issues macro, execution error.

 Unable to render Jira issues macro, execution error.

 Unable to render Jira issues macro, execution error.

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

## Motivation

Kafka Producer supports transactional semantics since 0.11, including the following APIs:

- **InitTransaction** for transactional producer identity initialization
- **beginTransaction** to start a new transaction

- **sendOffsetsToTransaction** to commit consumer offsets advanced within the current transaction
- **commitTransaction** commit the ongoing transaction
- **abortTransaction** abort the ongoing transaction

Within a transaction session, the internal state tracks the last fatal/abortable error. When the Producer hits a fatal/abortable exception, it will transit to the error state, and when next time someone uses a transactional API, it will throw the buffered exception. The caveat is that today we wrap many non-fatal exceptions as *KafkaException*, which does not make a clear boundary on whether the thrown exception is fatal – should fail fast, or just abortable – should catch and abort the ongoing transaction to resume. It affects the stability of upstream users as well such as Streams EOS. This KIP tries to address this gap by revamping Producer APIs to make it more robust and let exception handling coherent and consistent.

As of 08/16/2022, we ([guozhang Wang](#)) made a pass inside the *KafkaProducer* on how we categorizes those exceptions, and also on how we wrapped those exceptions.

First of all, the exceptions can be thrown in the following ways:

1) Thrown from **TransactionManager#maybeFailWithError**, which is thrown from **KafkaProducer#initTxn, #beginTxn, #commitTxn, #abortTxn, #sendOffsetsToTransaction, #send**. Except #send, all other functions would trigger maybeFailWithError right at the beginning of the function call. However, *KafkaProducer#send* would only call this check AFTER the record is appended to the accumulator, and hence it's possible that the sender has already sent this record out for idempotent producer (for transactional producer the record would not be sent until we successfully added the partition to the txn coordinator), this exception may be thrown directly from the #send call or maybe thrown from the #send callback of the future (KAFKA-14138). See below for more details. When they are thrown, they are mostly wrapped as *KafkaExceptions*

2) Thrown from **KafkaProducer#send's callback**, or the returned **future.get()**. Note that only produce request related exceptions may be thrown here, but these exceptions may also be thrown in 1) above since some of these exceptions would also be cause *TransactionManager* to transit to error states.

Now here's a full summary of all possible exceptions below:

Exception	Thrown Scenarios	Thrown places	Error Type	Suggestion
IllegalStateException	Checked for various call paths checking impossible situations, indicating a bug.	1), Wrapped as <i>KafkaException</i> (e)	Fatal	We should always throw <i>IllegalStateException</i> directly without wrapping
AuthenticationException	Only for txnal requests whose <i>txn.id</i> are not authenticated	1), Wrapped as <i>KE</i> (e)	Fatal	None, all good
InvalidPidMappingException	Only for txnal request with the encoded <i>txnID</i> are not recognized or its corresponding PID is incorrect	1), Wrapped as <i>KE</i> (e)	After KIP-360 (2.5+), Abortable as we bump epoch; otherwise Fatal	None
UnknownProducerIdException	Similar to <i>InvalidPidMappingException</i> but only for produce request (i.e. the PID is not recognized on the partition leader).  <i>NOTE this is removed as part of KIP-360, and hence would only be returned by old brokers. We keep this error code for now since we may re-use it in the future.</i>	1), Wrapped as <i>KE</i> (e)	After KIP-360 (2.5+), Abortable as we bump epoch; otherwise Fatal	None
TransactionAbortedException	Only for produce request batches, when the txn is already aborting we would simply abort all unsent batches with this error	2)	N/A since it is not an error case	None
ClusterAuthorizationException	Most of these errors are returned from produce responses (txnal response could also return <i>UnsupportedVersionException</i> ).	1), Wrapped as <i>KE</i> (e); and	Fatal	See below
TransactionalAuthorizationException	When these errors return, we would immediately mark the <i>txnManager</i> to error state as well. These are examples where the exceptions could be thrown in both <i>txnManager#maybeFailWithError</i> as well as from <i>send</i> callback/future.	2)		
UnsupportedVersionException				
UnsupportedFormatException				

InvalidRecordException	These are all errors returned from produce responses, that are non-fatal (timeout exception on expired batch).	1), Wrapped as KE(e); and 2)	Abortable	See below
InvalidRequiredAcksException				
NotEnoughReplicasAfterAppendException				
NotEnoughReplicasException				
RecordBatchTooLargeException				
InvalidTopicException				
CorruptRecordException				
UnknownTopicOrPartitionException				
NotLeaderOrFollowerException				
TimeoutException				
TopicAuthorizationException	TopicAuthorizationException could be thrown via addPartition.		Abortable	Should be fatal.
GroupAuthorizationException	GroupAuthorizationException could be thrown via sendOffsetToTxn / findCoordinator.  Today they are all categorized as abortable but I think this should be fatal.			
FencedInstanceIdException	Thrown from TxnOffsetCommit (CommitFailedException are translated from UNKNOWN_MEMBER and ILLEGAL_GENERATION).	1) Wrapped as KE(e)	Abortable	Should be fatal.
CommitFailedException	Today it's treated as abortable. BUT I think it should really be fatal since it's basically indicating a fenced situation.			
InvalidProducerEpochException	This error used to be returned from both txnal response and produce response, but as of KIP-588 (2.7+), we would not let txn coordinator to return InvalidProducerEpochException anymore, but only from partitions leaders on produce responses. Also since only older versioned coordinators still return InvalidProducerEpochException, clients would treat them as fatal ProducerFencedException at the client side.  HOWEVER, for TxnOffsetCommit (sent to the group coordinator) we did not do this conversion which is a bug — we should always convert to ProducerFenced.	1), BUT not wrapped	Fatal if from txnal response (translated to ProducerFenced);  Abortable if from produce response.	It's unclear why we wrap all other exceptions but leave these two un-wrapped; we should have a consistent wrapping mechanism.  Plus, we should fix the bug for TxnOffsetCommit error handling.
ProducerFencedException	This error used to be returned from both txnal response and produce response, but as of KIP-588 it should only be from txnal responses. It is a typical fatal error indicating that another producer with the same PID and newer epoch is in use.  With KIP-447, producers from Kafka Streams should not be fenced by txn.id any more since we would fence them based on the GroupCoordinator instead; the actual case this would be thrown is usually when a txn is timed out (pending KIP-588 to be completed)		Fatal	It's unclear why we wrap all other exceptions but leave these two un-wrapped; we should have a consistent wrapping mechanism.
OutOfOrderSequenceException	From produce response only, when the sequence does not match expected value	1), Wrapped as KE(e)	Abortable (for idempotent producer we would handle it internally by bumping epoch)	See below
InvalidTxnStateException	From txnal response, only, indicating the producer is issuing a request that it should not be.  NOTE that we are handling this exception inconsistently: in endTxn it's wrapped as KE(e), in addPartitions it's wrapped as KE(KE(e))	1), Wrapped as either KE(e) or KE(KE(e))...	Fatal	Should fix the inconsistent wrapping.
KafkaException	We definitely overloaded this one for various unrelated cases (which I think should be fixed):  1. when we failed to resolve those sequence-unresolved batches	1), Wrapped as KE(KE)	After KIP-360 (2.5+), Abortable as we bump epoch;  otherwise Fatal	Nested wrapping KafkaException (KafkaException (KafkaException...)) should be avoided.  For this case I suggest we wrap as KE (OutOfOrderSequenceException).
	2. when we are closing the producer, and hence need to garbage collect all pending txnal requests, we simply transit		Fatal	I don't think we should transit to error state at all for this case, and also shouldn't throw this exception either.
	3. When a txnal response does not contain the "response()" field.	1), Wrapped as KE(KE)	Fatal	Again, this should be an IllegalStateException since this should never happen.
	4. All unexpected errors from txnal response	1), Wrapped as KE(KE)	Fatal	Again, should not wrap it twice as KafkaException (KafkaException()).

	5. When addPartition response returns with partition-level errors	1), Wrapped as KE(KE (e))	Abortable	Again should not wrap it twice as as KafkaException (KafkaException()).
RuntimeException	For any txnal requests, when request / response correlation id does not match	1), Wrapped as KE(e)	Fatal	I think we should throw CorrelationIdMismatchException instead, which inherits from IllegalStateException, hence should not be wrapped either.

Besides the detailed suggestions on each of the above category lines, there are also a few meta-level proposals:

1. We should have a clear distinguishment between fatal and abortable errors. Today they are all wrapped as KafkaExceptions and hence users cannot really tell the difference, so they have to just treat them all as fatal. One idea is to wrap them differently: we use KafkaException to wrap fatal errors, while use AbortableException to wrap abortable errors.
2. We should have consistent mechanisms on wrapping errors. More specifically:
  - a. All exceptions should be wrapped at most once.
  - b. We should not have a nested wrapping like KE(KE(e)). Instead we should just have KE(e).
  - c. For IllegalStateException / Runtime errors which indicate a bug, we should not wrap but directly throw.
3. For errors that can be thrown from both scenario 1) and 2) above, we should have a clear guidance on how EOS users should handle them. More specifically:
  - a. Users should be safely ignore the returned future since all errors that should be set inside the TransactionManager as well and hence users should just capture the function itself.
  - b. The only error that's thrown only in 2) is TransactionAbortedException, and it could just be ignored since the user asked the producer to close.
4. From EOS-related exceptions thrown directly from the call, we would just check if they are abortable or fatal:
  - a. For abortable exception, re-throw them as TxnCorruptedException which will be handled by aborting the txns.
  - b. For fatal exception, re-throw them as TaskMigratedException which will cause us to lose all tasks and re-join the group.

## Proposed Changes

We are proposing a new transactional API usage template which makes EOS processing safer from handling a mix of fatal and non-fatal exceptions:

## Sample.java

```
KafkaConsumer consumer = new KafkaConsumer<>(consumerConfig);
producer.initTransactions();
volatile boolean isRunning = true;

try {
    while (isRunning) {
        ConsumerRecords<String, String> records = consumer.poll(CONSUMER_POLL_TIMEOUT);
        final boolean shouldCommit;
        try {
            producer.beginTransaction();

            // Do some processing and build the records we want to produce.
            List<ProducerRecord> processed = process(consumed);

            for (ProducerRecord record : processed)
                producer.send(record, (metadata, exception) -> {
                    // not required to capture the exception here.
                });
            producer.sendOffsetsToTransaction(consumedOffsets, consumer.groupMetadata());

            shouldCommit = true;
        } catch (Exception e) {
            // Catch any exception thrown from the data transmission phase.
            shouldCommit = false;
        }

        try {
            if (shouldCommit) {
                producer.commitTransaction();
            } else {
                resetToLastCommittedPositions(consumer);
                producer.abortTransaction();
            }
        } catch (CommitFailedException e) {
            // Transaction commit failed with abortable error, user could reset
            // the application state and resume with a new transaction. The root
            // cause was wrapped in the thrown exception.
            resetToLastCommittedPositions(consumer);
            producer.abortTransaction();
        }
    }
} catch (KafkaException e) { // Recommended closing producer/consumer for fatal exceptions
    producer.close();
    consumer.close();
    throw e;
}
```

In the above example, we separate the transactional processing into two phases: the data transmission phase, and the commit phase. In data transmission phase, any exception thrown would be an indication of the ongoing transaction failure, so that we got a clear signal for the next stage whether to commit or abort the ongoing transaction.

In the commit phase, we should decide whether to commit or abort transaction based on the previous stage result. In new Producer API, `commitTransaction()` will no longer throw non-fatal exceptions in their raw formats. Instead, it would try to wrap all non-fatal exceptions as `CommitFailedException`. This means any exception other than `CommitFailedException` caught during the commit phase will be definitely fatal, so user's error handling experience could be simplified by just doing a controlled shutdown.

The only debatable case is timeout exception within commit/abort transaction. It could be treated either fatal or not, as strictly speaking producer would have already done the retrying for *max.block.ms*, so a timeout here may be suggesting a fatal state to a basic user's perspective. Blindly call `abortTxn` upon timeout could result in illegal state as well when the previous commit already writes `prepare_commit` on the broker side. Usually caller level could have more sophisticated handling to do an application level retry if necessary, but we don't do any recommendations here. It is highly recommended to increase the request timeout here instead of relying on unreliable retries.

We also put another try-catch block outside of the whole processing loop to get a chance catching all fatal exceptions and close producer and consumer modules. It is a recommended way to handle fatal exceptions when the application still wants to proceed without any memory leak, but is optional to users.

## Unify Wrapped KafkaException

As discussed in the motivation section, in `KafkaProducer` we have a logic to wrap all thrown exceptions as `KafkaException`. To make the semantic clear and for advanced users such as Kafka Streams to better understand the root cause, we shall no longer wrap any fatal exceptions, but instead only wrap non-fatal ones as `KafkaException`. We also detect certain cases where we did a double-wrap of `KafkaException` internally, which will be addressed to ensure only one layer wrapping is supported.

## Callback Exception Improvement

As we have seen, there is a callback mechanism in the `producer#send` which carries the exception type. In EOS setup, it is not required to handle the exception, but for non-EOS cases, the current exception type mechanism is complicated as it throws raw exceptions. Although in the callback function comments we listed all the fatal and non-fatal exceptions, in users' perspective they still need to maintain an exhausting list for checking exception types and take proper actions. The application code is fragile when facing ever-changing underlying producer logic with new exception types, and we have seen the difficulty to classify populated exceptions in application level such as Streams.

On the other thread, there are [proposals](#) around making producer API adopt more modern return types than a mere Future. This is a potential good opportunity to merge these two efforts.

To make the handling easier and consistent, we suggest to add a new exception type called `ProduceFailedException` which wraps the thrown exception and contains an enum field indicating failure type. Right now there would be 3 types of failures:

1. message rejected: this error associates with this specific record being produced, such as `InvalidTopic` or `RecordTooLarge`
2. delivery failed: suggests a failure to produce last record, such as `NotEnoughReplicas` or `Timeout`
3. transaction error: an exception relating specifically to transactional semantic, such as `ProducerFenced` or `InvalidTransactionState`

With additional flagging, producer users would be in a much better position interpreting the callback and take proper actions with less effort to diagnose the failures by themselves.

This new exception type would be thrown back to the user only in the new producer API depicted in [KIP-706](#).

## Streams Side Change

For EOS Kafka Streams case, we would adopt these simplified exception throwing logic by catching all exceptions in the data transmission phase to decide for Streams commit. Furthermore, these changes leave to door open for us to analyze the non-fatal exceptions thrown as well by unwrapping `KafkaException`'s cause and reading failure type through callback.

More specifically:

- For known exceptions such as `ProducerFenced`, the handling shall be simplified as we no longer need to wrap them as `TaskMigratedException` in the send callback, since they should not crash the stream thread if thrown in raw format, once we adopt the new processing model in the send phase.
- When handling lost-all-partitions, which would trigger when 1) the rebalance listener's `onPartitionsLost` are called, indicating the consumer member has been kicked out of the group, 2) a task-migration exception is thrown, we should not need to reset the producer by closing the current one and re-creating a new producer any more. Instead, we should still be able to reuse the same producer after we've re-joined the consumer group. Instead we just need to re-`initTxn`` on the producer to make sure the previous handling txns have been aborted before new transactions are about to start.
- We should distinguish exceptions thrown from the `send()` callback v.s. from the `send()` / `commit()` / etc call directly. With EOS, the only exception that would ONLY be thrown in the callback would be `TransactionAbortedException`, which we can actually ignore; and hence we would only need to just capture all exceptions thrown from the calls directly. That means we would handle exceptions differently between ALOS and EOS:
  - ALOS: try to capture exceptions from the callback, handle them just as today.
  - EOS: ignore exceptions from the callback, instead handle directly from the function calls.

## Public Interfaces

As mentioned in the proposed changes section, we would be doing the following public API changes:

- The `commitTransaction()` API will throw `CommitFailedException` to wrap non-fatal exceptions
- All the non-fatal exceptions thrown from data transmission APIs will be wrapped as `KafkaException`, which we will be documented clearly. This includes:
  - **`beginTransaction`**
  - **`sendOffsetsToTransaction`**
  - **`send`**

We would also let `commitTransaction` API only throw `CommitFailedException` with wrapped cause when hitting non-fatal exceptions, to simply the exception try-catching.

We would add a new Producer error type called `ProduceFailedException` which tries to wrap send/produce exceptions with root cause and reasoning.

## Callback.java

```
/**
 * Exception indicating a produce failure for the given record, with root cause and reasoning embedded.
 */
public ProduceFailedException extends ApiException {

    private final FailureType type;

    public ProduceFailedException(Throwable cause, FailureType type) {
        super(cause);
        this.type = type;
    }

    public FailureType failureType() {
        return type;
    }

    enum FailureType {
        MESSAGE_REJECTED, // the specific record being produced was rejected, such as InvalidTopic or
        RecordTooLarge
        DELIVERY_FAILED, // a failure to produce last record, such as NotEnoughReplicas or Timeout
        TRANSACTION_FAILED // a transactional processing failure, such as ProducerFenced or
        InvalidTransactionState
    }
}
```

The failure type in the embed exception should be helpful, for example, they could choose to ignore transactional errors since other txn APIs are already taking care of them. We expect this error code to be implemented once KIP-706 is accepted, which would provide a more user-friendly send API with `CompletableFuture` or similar.

## Documentation change

We shall put the newly marked fatal exceptions on the public Producer API docs correspondingly, including

- ***beginTransaction***
- ***sendOffsetsToTransaction***
- ***commitTransaction***
- ***abortTransaction***
- ***send***

## Compatibility, Deprecation, and Migration Plan

This is a pure client side change which only affects the resiliency of new Producer client and Streams. For customized EOS use case, user needs to change their exception catching logic to take actions against their exception handling around `commitTransaction()`, since it no longer throws non-fatal exception, which means it does not indicate a success of commit when not throwing. However, all the thrown exceptions' base type would still be `KafkaException`, so the effect should be minimal.

## Rejected Alternatives

*We thought about exhausting all the possible exception types on the Streams level for resiliency, but abandoned the approach pretty soon as it would require a joint code change every time the underlying Producer client throws a new exception. The encapsulation should help reduce the amount of work on the caller side for exception handling.*

We also proposed to add a non-fatal exception wrapper type called **TransactionStateCorruptedException** to help users distinguish thrown exception types. This solution has compatibility issue and is not necessarily making the developer and user's life easier.

We proposed to add a return boolean in `commitTransaction`, so that even if the commit failed internally with non-fatal exception but wasn't throwing, we still got a clear returned signal from `commitTransaction` to know whether the last commit was successful, as certain EOS users rely on external data storage component and need to perform non-rollback commit operation as necessary. This approach was abandoned as it broke the compatibility since old users would not assume a `commitTxn` not to be indicating a success when it passes without throwing non-fatal exceptions.

We discussed about throwing `ProduceFailedException` within the send callback, however there are compatibility concerns. The other approach is to make failure reason as part of the callback function, which is less optimized than introducing a true async method like KIP-706 did.