

KIP-609: Use Pre-registration and Blocking Calls for Better Transaction Efficiency


- [Status](#)
- [Motivation](#)
- [Proposed Changes](#)
 - [Block Begin Transaction](#)
 - [Pre-register Output Partitions And Consumer Group](#)
- [Public Interfaces](#)
 - [Performance Testing](#)
- [Compatibility, Deprecation, and Migration Plan](#)
- [Rejected Alternatives](#)

Status

Current state: Under Discussion

Discussion thread: TBD

JIRA:

key	summary	type	created	updated	due	assignee	reporter	priority	status	resolution
 JQL and issue key arguments for this macro require at least one Jira application link to be configured										

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

Motivation

Kafka transaction uses a two-phase model, where the transaction state is stored by transaction coordinator. As of current, the way to make it work is through `AddPartitionToTxn` call each time we see a new output partition. In reality though, the set of output partitions for a producer doesn't change very often, and even deterministic at the transaction beginning, so it is a waste of network round trips to update that set unnecessarily based on witnessing of new output records.

The other issue is the infinite retry mechanism for `AddPartitionToTxn` when being served as the starting request for a new transaction. Right now the `EndTxn` call returns immediately when the txn commit message is materialized in transaction log. However broker needs to propagate this information to all the affected parties to complete the transaction, which means there is a black-out period that no more transaction could be started. During this time, all the new `AddPartitionToTxn` calls shall be rejected and asked to retry, which again creates many unnecessary network trips. One upside with the current model is that the first `AddPartitionToTxn` call could be deferred until the saturation of the first Producer batch either due to size (`batch.size`) or latency bound (`linger.ms`), to make the blocking behavior less influential.

A minor issue is that the current `AddOffsetsToTxn` API is only registering a consumer group id towards the transaction coordinator. Unless for certain advanced use cases where multiple consumer groups are behind one transactional producer, this shall also be deterministic and could be combined with `AddPartitionToTxn` to save a network trip as well.

More details on a transaction session workflow could be found [here](#).

Proposed Changes

Block Begin Transaction

We propose to make `AddPartitionToTxn` a blocking call when there is other pending transaction going on. This could be done by holding the response callback as part of the transaction marker completion in the purgatory. The goal is to reduce unnecessary bounces during the transaction marker completion.

Pre-register Output Partitions And Consumer Group

Look at the latest EOS example below:

```

KafkaConsumer consumer = new KafkaConsumer<>(consumerConfig);
// Recommend a smaller txn timeout, for example 10 seconds.
producerConfig.put(ProducerConfig.TRANSACTION_TIMEOUT_CONFIG, 10000);
KafkaProducer producer = new KafkaProducer(producerConfig);

producer.initTransactions();
while (true) {
    // Read some records from the consumer and collect the offsets to commit
    ConsumerRecords consumed = consumer.poll(Duration.ofMillis(5000)); // This will be the fencing point if
    there are pending offsets for the first time.
    Map<TopicPartition, OffsetAndMetadata> consumedOffsets = offsets(consumed);

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

    // Write the records and commit offsets under a single transaction
    producer.beginTransaction();
    for (ProducerRecord record : processed)
        producer.send(record);

    // Pass the entire consumer group metadata
    producer.sendOffsetsToTransaction(consumedOffsets, consumer.groupMetadata());

    producer.commitTransaction();
}

```

As one could see, with the set of output records, developer could easily infer the output partitions. The consumer group id is also known before the transaction begins.

Public Interfaces

We would bump the AddPartitionToTxn API version by one to add a nullable consumer group id field:

AddPartitionsToTxn.json

```

{
  "apiKey": 24,
  "type": "request",
  "name": "AddPartitionsToTxnRequest",
  // Version 1 is the same as version 0.
  //
  // Version 2 adds consumer group id.
  "validVersions": "0-2",
  "flexibleVersions": "none",
  "fields": [
    { "name": "TransactionalId", "type": "string", "versions": "0+", "entityType": "transactionalId",
      "about": "The transactional id corresponding to the transaction." },
    { "name": "ProducerId", "type": "int64", "versions": "0+", "entityType": "producerId",
      "about": "Current producer id in use by the transactional id." },
    { "name": "ProducerEpoch", "type": "int16", "versions": "0+",
      "about": "Current epoch associated with the producer id." },
    // ----- START NEW FIELD -----
    { "name": "GroupId", "type": "string", "versions": "2+", "entityType": "groupId", "nullableVersions": "2+",
      "default": "null",
      "about": "The unique group identifier." }
    // ----- END NEW FIELD -----
    { "name": "Topics", "type": "[]AddPartitionsToTxnTopic", "versions": "0+",
      "about": "The partitions to add to the transaction.", "fields": [
        { "name": "Name", "type": "string", "versions": "0+", "mapKey": true, "entityType": "topicName",
          "about": "The name of the topic." },
        { "name": "Partitions", "type": "[]int32", "versions": "0+",
          "about": "The partition indexes to add to the transaction" }
      ]
    }
  ]
}

```

We would change beginTransaction API on the KafkaProducer to give user the access for including the pre-registered partitions.

KafkaProducer.java

```
/**
 * Should be called before the start of each new transaction. Note that prior to the first invocation
 * of this method, you must invoke {@link #initTransactions()} exactly one time.
 *
 * @param outputPartitions the set of output partitions to produce to.
 * @param consumerGroupId the consumer group id to send offsets to.
 *
 * @throws IllegalStateException if no transactional.id has been configured or if {@link
 #initTransactions()}
 *         has not yet been invoked
 * @throws ProducerFencedException if another producer with the same transactional.id is active
 * @throws org.apache.kafka.common.errors.UnsupportedVersionException fatal error indicating the broker
 *         does not support transactions (i.e. if its version is lower than 0.11.0.0), or doesn't support
 pre-registration of
 *         partitions and consumer.
 * @throws org.apache.kafka.common.errors.AuthorizationException fatal error indicating that the configured
 *         transactional.id is not authorized. See the exception for more details
 * @throws KafkaException if the producer has encountered a previous fatal error or for any other
 unexpected error
 */
public void beginTransaction(Set<TopicPartition> outputPartitions, Optional<String> consumerGroupId) throws
ProducerFencedException;
```

Note that the consumerGroupId could be null so that it is not a strong requirement to put it as part of user code when consumer is not used.

This API shall be flexible for user to decide whether to call it before the record processing to pipeline the effort, as:

```
// Read some records from the consumer and collect the offsets to commit
ConsumerRecords consumed = consumer.poll(Duration.ofMillis(5000)); // This will be the fencing point if
there are pending offsets for the first time.
Map<TopicPartition, OffsetAndMetadata> consumedOffsets = offsets(consumed);

// ----- Call before process -----
// The application could infer output partition beforehand
Set<TopicPartition> outputPartitions = outputPartition(consumed);
producer.beginTransaction(outputPartitions, Optional.of(consumer.groupMetadata().groupId()));

// Do processing while sending out the built records asynchronously
for (ProducerRecord record : process(consumed))
    producer.send(record);

// Pass the entire consumer group metadata
producer.sendOffsetsToTransaction(consumedOffsets, consumer.groupMetadata());

producer.commitTransaction();
```

or user could do it after the processing for larger produce batch.

```

// Read some records from the consumer and collect the offsets to commit
ConsumerRecords consumed = consumer.poll(Duration.ofMillis(5000)); // This will be the fencing point if
there are pending offsets for the first time.
Map<TopicPartition, OffsetAndMetadata> consumedOffsets = offsets(consumed);

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

// ----- Call after process -----
Set<TopicPartition> outputPartitions = outputPartitions(processed);
producer.beginTransaction(outputPartitions, Optional.of(consumer.groupMetadata().groupId()));

// Write the records and commit offsets under a single transaction
for (ProducerRecord record : processed)
    producer.send(record);

// Pass the entire consumer group metadata
producer.sendOffsetsToTransaction(consumedOffsets, consumer.groupMetadata());

producer.commitTransaction();

```

To ensure that we don't encounter human mistake, when the application is using the new `beginTransaction` API, we shall not expect any produced record going to an unknown partition, or commit to some non-registered group. We shall let `producer.send()` throw `UNKNOWN_TOPIC_OR_PARTITION` and `producer.sendOffsets()` throw `INVALID_GROUP_ID`, while augmenting their corresponding definitions:

```

UNKNOWN_TOPIC_OR_PARTITION(3, "This server does not host this topic-partition, or this error is thrown from a
transactional producer who doesn't expect this topic partition to send to.", UnknownTopicOrPartitionException::
new),

INVALID_GROUP_ID(69, "The configured groupId is invalid, or this error is thrown from a transactional producer
who doesn't expect to commit offset to this group.", InvalidGroupIdException::new),

```

Updated JavaDoc for *send* and *sendOffsetsToTransaction*:

KafkaProducer.java

```
/**
 * Asynchronously send a record to a topic and invoke the provided callback when the send has been
 * acknowledged.
 *
 * @param record The record to send
 * @param callback A user-supplied callback to execute when the record has been acknowledged by the server
 * (null indicates no callback)
 *
 * @throws AuthenticationException if authentication fails. See the exception for more details
 * @throws AuthorizationException fatal error indicating that the producer is not allowed to write
 * @throws IllegalStateException if a transactional.id has been configured and no transaction has been
 * started, or
 *
 * @throws InterruptedException If the thread is interrupted while blocked
 * @throws SerializationException If the key or value are not valid objects given the configured serializers
 * @throws UnknownTopicOrPartitionException If the record topic partition is not included in the pre-
 * registered partitions
 * @throws KafkaException If a Kafka related error occurs that does not belong to the public API exceptions.
 */
@Override
public Future<RecordMetadata> send(ProducerRecord<K, V> record, Callback callback) {
    // intercept the record, which can be potentially modified; this method does not throw exceptions
    ProducerRecord<K, V> interceptedRecord = this.interceptors.onSend(record);
    return doSend(interceptedRecord, callback);
}

/**
 * Sends a list of specified offsets to the consumer group coordinator, and also marks
 * those offsets as part of the current transaction.
 *
 * @throws IllegalStateException if no transactional.id has been configured or no transaction has been
 * started.
 * @throws ProducerFencedException fatal error indicating another producer with the same transactional.id
 * is active
 * @throws org.apache.kafka.common.errors.UnsupportedVersionException fatal error indicating the broker
 * does not support transactions (i.e. if its version is lower than 0.11.0.0) or
 * the broker doesn't support latest version of transactional API with consumer group metadata (i.
 * e. if its version is
 * lower than 2.5.0).
 * @throws org.apache.kafka.common.errors.UnsupportedForMessageFormatException fatal error indicating the
 * message
 * format used for the offsets topic on the broker does not support transactions
 * @throws org.apache.kafka.common.errors.AuthorizationException fatal error indicating that the configured
 * transactional.id is not authorized, or the consumer group id is not authorized.
 * @throws org.apache.kafka.clients.consumer.CommitFailedException if the commit failed and cannot be
 * retried
 * (e.g. if the consumer has been kicked out of the group). Users should handle this by aborting
 * the transaction.
 * @throws org.apache.kafka.common.errors.FencedInstanceIdException if this producer instance gets fenced
 * by broker due to a
 *
 * mis-configured consumer instance id
 * within group metadata.
 * @throws org.apache.kafka.common.errors.InvalidGroupIdException if the given consumer group id
 * doesn't match the
 *
 * pre-registered value.
 * @throws KafkaException if the producer has encountered a previous fatal or abortable error, or for any
 * other unexpected error
 */
public void sendOffsetsToTransaction(Map<TopicPartition, OffsetAndMetadata> offsets,
    ConsumerGroupMetadata groupMetadata) throws ProducerFencedException
```

There is also an edge case where a user registers a lot of topic partitions, but didn't write to all of them. In this case, it is a waste of resource to write transaction markers to all of the registered partitions. To minimize such unexpected cost, we shall also bump the EndTxn to include the actual written partitions.

EndTxnRequest.json

```
{
  "apiKey": 26,
  "type": "request",
  "name": "EndTxnRequest",
  // Version 1 is the same as version 0.
  //
  // Version 2 adds the support for new PRODUCER_FENCED error code.
  //
  // Version 3 adds the partitions actually gets written during the current transaction, or null to indicate
  that
  // all the partitions registered so far gets written to.
  "validVersions": "0-3",
  "flexibleVersions": "none",
  "fields": [
    { "name": "TransactionalId", "type": "string", "versions": "0+", "entityType": "transactionalId",
      "about": "The ID of the transaction to end." },
    { "name": "ProducerId", "type": "int64", "versions": "0+", "entityType": "producerId",
      "about": "The producer ID." },
    { "name": "ProducerEpoch", "type": "int16", "versions": "0+",
      "about": "The current epoch associated with the producer." },
    { "name": "Committed", "type": "bool", "versions": "0+",
      "about": "True if the transaction was committed, false if it was aborted." },
    // ----- START NEW FIELD -----
    { "name": "Topics", "type": "[]EndTxnTopic", "versions": "3+",
      "about": "The partitions ending up writing in current transaction.", "fields": [
        { "name": "Name", "type": "string", "versions": "0+", "mapKey": true, "entityType": "topicName",
          "about": "The name of the topic." },
        { "name": "Partitions", "type": "[]int32", "versions": "3+",
          "about": "The partition indexes to add to the transaction" }
      ]
    }
  ]
  // ----- END NEW FIELD -----
}
```

This logic will be handled internally for the producer to bookkeep sent record partitions, no action needed for user.

Performance Testing

Since this KIP is about performance improvement, benchmarking the producer performance with/without pre-registration is necessary. There already exists a performance benchmark as *org.apache.kafka.tools.ProducerPerformance* which could be utilized to perform local benchmark tests. A new config parameter shall be added:

ProducerPerformance

```
parser.addArgument("--pre-registration")
    .action(store())
    .required(false)
    .type(Boolean.class)
    .metavar("PRE-REGISTRATION")
    .help("Whether to pre-register topic partitions");
```

Moreover, the *ProducerPerformance* module currently only deals with producer only, which doesn't take consumer into consideration. We would like to introduce a separate set of performance benchmark called *EOSPerformance* to verify the throughput under poll-process-produce scenario. The new benchmark would be having a similar setup as *ProducerPerformance* but adds a couple of extra configurations, and remove some unnecessary configs (record generation payload, for example):

EOSPerformance

```
// ----- New configurations beyond ProducerPerformance -----
parser.addArgument("--input-topic")
    .action(store())
    .required(true)
    .type(String.class)
    .metavar("INPUT-TOPIC")
    .dest("inputTopic")
    .help("consume messages from this topic");

parser.addArgument("--output-topic")
    .action(store())
    .required(true)
    .type(String.class)
    .metavar("OUTPUT-TOPIC")
    .dest("outputTopic")
    .help("produce messages to this topic");

parser.addArgument("--consumer-config")
    .action(store())
    .required(false)
    .type(String.class)
    .metavar("CONFIG-FILE")
    .dest("consumerConfigFile")
    .help("consumer config properties file.");

parser.addArgument("--consumer-props")
    .nargs("+")
    .required(false)
    .metavar("PROP-NAME=PROP-VALUE")
    .type(String.class)
    .dest("consumerConfig")
    .help("kafka consumer related configuration properties like bootstrap.servers,client.id etc. " +
        "These configs take precedence over those passed via --consumer-config.");

parser.addArgument("--consumer-group-id")
    .action(store())
    .required(false)
    .type(String.class)
    .metavar("CONSUMER-GROUP-ID")
    .dest("consumerGroupId")
    .setDefault("performance-consumer-default-group-id")
    .help("The consumerGroupId to use if we use consumer as the data source.");

parser.addArgument("--poll-timeout")
    .action(store())
    .required(false)
    .type(Long.class)
    .metavar("POLL-TIMEOUT")
    .dest("pollTimeout")
    .setDefault(100L)
    .help("consumer poll timeout");

parser.addArgument("--pre-registration")
    .action(store())
    .required(false)
    .type(Boolean.class)
    .metavar("PRE-REGISTRATION")
    .help("Whether to pre-register topic partitions");
```

We have built a [prototype](#) last year but it is incomplete and could break some external dependency on *ProducerPerformance*. In this KIP proposal, we would try to build the benchmark with the effort to maintain compatibility as necessary.

Compatibility, Deprecation, and Migration Plan

The blocking behavior for the first *AddPartitionsToTxn* shall be applied to both old and new producers, which are expected to be retried when the request gets timeout eventually.

The new *beginTransaction* API is only available for new producers, so there should not be a compatibility issue. To enable pre-registration, the broker has to be on the latest version, otherwise *UnsupportedVersionException* would be thrown.

Rejected Alternatives

*We have thought about using a context session on the broker side to remember the partitions being written to in the last transaction, and the session gets updated by the *EndTxn* call.*

This solution has less flexibility and especially makes the reinitialization harder, for example a re-initialized producer doesn't know what previous session looks like, so there needs to be special logic to make sure it doesn't rely on any pre-defined state which could be false. And it is harder to understand, as well.