KIP-42: Add Producer and Consumer Interceptors

Status

Current state: *Accepted*

Discussion thread: here and here

JIRA: **KAFKA-3162** - Getting issue details... STATUS **KAFKA-3196** - Getting issue details... STATUS **KAFKA-3303** - Getting issue details... STATUS

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

Motivation

Today, Kafka metrics are only collected for individual clients or brokers. This makes it difficult for users to trace the path of individual messages across the cluster, providing a complete end-to-end picture of system performance and behavior. Technically, it is possible to measure end-to-end performance today by modifying users applications to collect and track additional information, but that isn't always practical for critical infrastructure applications. The ability to quickly deploy tools to observe, measure, and monitor Kafka client behavior, down to the message level, is valuable in production environments. At the same time, metrics might need contextual metadata that may vary across applications. The ability to measure and monitor clients without writing new code or recompiling applications is essential. (In some cases, it might help to connect to running applications.)

To enable this functionality, we would like to add producer and consumer interceptors that can intercept messages at different points on producer and consumer. The mechanism that we are proposing is inspired by the interceptor interface in Apache Flume. While there are potentially many ways to use an interceptor interface (for example, detecting anomalies, encrypting data, filtering fields), each of them would require a careful evaluation of whether or not it should be done with interceptor or with another mechanism. It is better to add the related APIs when there is a clear motivation for those use cases. Thus, we are proposing minimal producer and consumer interceptor interfaces that are designed to support only measurement and monitoring.

While it is possible to add more metrics or improve monitoring in Kafka, we believe that creating a flexible, customizable interface is beneficial for the following reasons:

1. Common monitoring tools. In a large company, different teams collaborate on building systems. Often, different teams develop and deploy different components over time. In addition, organizations want to standardize on common metrics, formats, and data collection systems. We think it is valuable for an organization to develop and deploy common Kafka client monitoring tools and deploy these across all applications that use Kafka.
2. Monitoring can be expensive. Adding additional metrics to Kafka might compromise performance. (For example see this JIRA ticket for an example of a performance regression caused by just checking timestamps.) Unfortunately, there is sometimes a tradeoff between system performance and data collection. As an example, consider the problem of measuring message sizes. The cheapest, simplest, and most straightforward approach is to measure average values. Calculating percentiles on a distributed system is more expensive and complicated than calculating simple averages, but would be useful in many applications. We would like to give users the ability to adopt different algorithms for metric collection, or to choose not to collect metrics at all.
3. Different applications require different metrics. For example, a user might find it important to monitor the cardinality of different keys in Kafka messages. It would be impractical for Kafka to provide all possible metrics internally; a pluggable intercept system provides a simple way to develop customized metrics.
4. Kafka is often a part of a bigger infrastructure in an organization, and it would be very useful to enable end-to-end tracing in that infrastructure. Consider LinkedIn's use of Samza to trace frontend user calls across all services by tagging each call with a unique value, called TreeId, and propagating that value across all subsequent service calls. Interceptors will allow tracing of Kafka clients through the same infrastructure, tracing with the same TreeId stored in a message.

In this KIP, we propose adding two new interfaces: ProducerInterceptor on producer and ConsumerInterceptor on consumer. User will be able to implement and configure a chain of custom interceptors and listen to events that happen to a record at different points on producer and consumer. Interceptor API will allow mutate the records to support the ability to add metadata to a message for auditing/end-to-end monitoring.

Public Interfaces

We add two new interfaces: *ProducerInterceptor* interface that will allow plugging in classes that will be notified of events happening to the record during its lifetime on the producer; and *ConsumerInterceptor* interface that will allow plugging in classes that will be notified of record events on the consumer. ProducerInterceptor API will allow to modify keys and values pre-serialization. For symmetry, ConsumerInterceptor API will allow to modify keys and values post-deserialization.

Both ProducerInterceptor and ConsumerInterceptor inherit from Configurable. Properties passed to configure() method will be consumer/producer config properties (including clientId if it was not specified in the config and assigned by KafkaProducer/KafkaConsumer). We will document in the Producer /ConsumerInterceptor class description that they will be sharing producer/consumer config namespace possibly with many other interceptors and serializers. So, it could be useful to use a prefix to prevent conflicts.

All exceptions thrown by interceptor callbacks will be caught by the caller method and ignored. The alternative was to allow exceptions to propagate through the original calls (at least for some of the callbacks), which will enable an additional level of control. For example, interceptors can filter messages this way on consumer side or stop messages on producer because they do not have the right field. However, this will effectively change KafkaProducer and KafkaConsumer API, because now they can throw exceptions that are not documented in KafkaProducer and KafkaConsumer API. In this KIP, we propose to ignore all exceptions from interceptors, but this could be changed in the future if/when we have strong use-cases for this.

Add a new configuration setting *interceptor.classes* to the KafkaProducer API which sets a list of classes to use as producer interceptors. Each specified class must implement ProducerInterceptor interface. The default configuration will have an empty list.

Add a new configuration setting *interceptor.classes* to the KafkaConsumer API which sets a list of classes to use as consumer interceptors. Each specified class must implement ConsumerInterceptor interface. The default configuration will have an empty list.

Here is more detailed description of new interfaces:

ProducerInterceptor interface

ProducerInterceptor

```
/**
 * A plugin interface to allow things to intercept events happening to a producer record,
 * such as sending producer record or getting an acknowledgement when a record gets published
 */
public interface ProducerInterceptor<K, V> extends Configurable {
    /**
     * This is called when client sends record to KafkaProducer, before key and value gets serialized.
     * @param record the record from client
     * @return record that is either original record passed to this method or new record with modified key and
value.
     */
    public ProducerRecord<K, V> onSend(ProducerRecord<K, V> record);

    /**
     * This is called when the send has been acknowledged
     * @param metadata The metadata for the record that was sent (i.e. the partition and offset). The metadata
information may be only partially filled, if an error occurred. Topic will be always set, and if partition is
not -1, partition will be set partition set/assigned to this record.
     * @param exception The exception thrown during processing of this record. Null if no error occurred.
     */
    public void onAcknowledgement(RecordMetadata metadata, Exception exception);

    /**
     * This is called when interceptor is closed
     */
    public void close();
}
```

onSend() will be called in KafkaProducer.send(), before key and value gets serialized and before partition gets assigned. If the implementation modifies key and/or value, it must return modified key and value in a new ProducerRecord object. The implication of interceptors modifying a key in onSend() method is that partition will be assigned based on modified key, not the key from the client. If key/value transformation is not consistent (same key and value does not mutate to the same, but modified, key/value), then log compaction would not work. We will document this in ProducerInterceptor class. However, known use-cases, such as adding app name, host name to a message will do consistent transformation.

Another implication of onSend() returning ProducerRecord is that the interceptor can potentially modify topic/partition. It will be up to the interceptor that ProducerRecord returned from onSend() is correct (e.g. topic and partition, if given, are preserved or modified). KafkaProducer will use ProducerRecord returned from onSend() instead of record passed into KafkaProducer.send() method.

Since there may be multiple interceptors, the first interceptor will get a record from client passed as the 'record' parameter. The next interceptor in the list will get the record returned by the previous interceptor, and so on. Since interceptors are allowed to mutate records, interceptors may potentially get the record already modified by other interceptors. However, we will state in the javadoc that building a pipeline of mutable interceptors that depend on the output of the previous interceptors is discouraged, because of potential side-effects caused by interceptors potentially failing to mutate the record and throwing and exception. If one of the interceptors in the list throws an exception from onSend(), the exception is caught, logged,and the next interceptor is called with the record returned by the last successful interceptor in the list, or otherwise the client.

onAcknowledgement() will be called when the send is acknowledged. It has same API as Callback.onCompletion(), and is called just before Callback.onCompletion() is called. In addition, onAcknowledgement() will be called just before KafkaProducer.send() throws an exception (even when it does not call user callback). The difference in the behavior of ProducerInterceptor.onAcknowledgement() is that if an error occurred, metadata parameter will not be null. In this case, metadata will contain topic and possibly partition information (if available). If partition information is not available, then partition will be assigned -1.

ProducerInterceptor APIs will be called from multiple threads: onSend() will be called on submitting thread and onAcknowledgement() will be called on producer I/O thread. It is up to the interceptor implementation to ensure thread safety. Since onAcknowledgement() is called on producer I/O thread, onAcknowledgement() implementation should be reasonably fast, or otherwise sending of messages from other threads could be delayed.

ConsumerInterceptor **interface**

ConsumerInterceptor

```
/**
 * A plugin interface to allow things to intercept Consumer events such as receiving a record or record being
consumed
 * by a client.
 */
public interface ConsumerInterceptor<K, V> extends Configurable {
    /**
     * This is called when the records are about to be returned to the client.
     * @param records records to be consumed by the client. Null if record dropped/ignored/discarded (non
consumable)
     * @return records that is either original 'records' passed to this method or modified set of records
     */
    public ConsumerRecords<K, V> onConsume(ConsumerRecords<K, V> records);

    /**
     * This is called when offsets get committed
     * This method will be called when the commit request sent to the server has been acknowledged.
     * @param offsets A map of the offsets and associated metadata that this callback applies to
     */
    public void onCommit(Map<TopicPartition, OffsetAndMetadata> offsets);

    /**
     * This is called when interceptor is closed
     */
    public void close();
}
```

onConsume() will be called in KafkaConsumer.poll(), just before poll() returns ConsumerRecords. The implementation of onConsume() is allowed to modify key and values in ConsumerRecords, and if so, return them in new ConsumerRecords. This method is designed to be symmetric to ProducerInterceptor.onSend() and provides a way to undo a transformation done in an onSend producer interceptor. The records returned from onConsume will be returned to the user from KafkaConsumer.poll(). Thus, the implication of this callback is that the interceptors can potentially modify topic, partition, offset of the record. We will clearly document this is up to the interceptor to make sure that topic, partition, and offset returned in ConsumerRecords are valid.

Since there may be multiple interceptors, the first interceptor will get records consumed by the consumer. The next interceptor in the list will get the records returned by the previous interceptor, and so on. Since interceptors are allowed to mutate records, interceptors may potentially get the records already modified by other interceptors. However, we will state in the javadoc that building a pipeline of mutable interceptors that depend on the output of the previous interceptors is discouraged, because of potential side-effects caused by interceptors potentially failing to mutate the records and throwing and exception. If one of the interceptors in the list throws an exception from onConsume(), the exception is caught, logged,and the next interceptor is called with the records returned by the last successful interceptor in the list, or otherwise consumed from brokers.

onCommit() will be called when offsets get committed: just before OffsetCommitCallback.onCompletion() is called and in ConsumerCoordinator.commitOffsetsSync() on successful commit.

Since new consumer is single-threaded, ConsumerInterceptor API will be called from a single thread. Since interceptor callbacks are called for every record, the interceptor implementation should be careful about adding performance overhead to consumer.

Add more record metadata to RecordMetadata and ConsumerRecord

Currently, RecordMetadata contains topic/partition, offset, and timestamp (KIP-32). We propose to add remaining record's metadata in RecordMetadata: checksum and record size. Both checksum and record size are useful for monitoring and audit. Checksum provides an easy way to get a summary of the message and is also useful for validating a message end-to-end. For symmetry, we also propose to expose the same metadata on consumer side and make available to interceptors.

We will add checksum and record size fields to RecordMetadata and ConsumerRecord.

RecordMetadata

```
public final class RecordMetadata {
    private final long offset;
    private final TopicPartition topicPartition;
    private final long checksum;               <<== NEW: checksum of the record
    private final int size;                    <<== NEW: record size in bytes(before compression)
.......
```

ConsumerRecord

```
public final class ConsumerRecord<K, V> {
 .......
    private final long checksum;              <<== NEW: checksum of the record
    private final int size;                   <<== NEW: record size in bytes (after decompression)
```

We will make it clear in the documentation (of ConsumerRecord and onAknowledgement/onConsume) that checksum the consumer sees may not always be the one initially set on the producer. CRC may be overwritten by the broker during upgrade after message format change or in the case of topic config with timestamp type == LogAppendTime, which requires over-writing message timestamps in the message on the broker and as a result overwriting.

Proposed Changes

We propose to add two new interfaces listed and described in the Public Interfaces section: ProducerInterceptor and ConsumerInterceptor. We will allow a chain of interceptors. It is up to the user to correctly specify the order of interceptors in *producer.interceptor.classes* and in *consumer.interceptor.classes*.

Kafka Producer changes

- We will create a new class that will encapsulate a list of ProducerInterceptor instances: *ProducerInterceptors*

ProducerInterceptors

```
/**
 * This class wraps custom interceptors configured for this producer.
 */
public class ProducerInterceptors<K, V> implements Closeable {
    private final List<ProducerInterceptor<K,V>> interceptors;

    public ProducerInterceptors(List<ProducerInterceptor<K,V>> interceptors) {
        this.interceptors = interceptors;
    }

    public ProducerRecord<K, V> onSend(ProducerRecord<K, V> record) {
        ProducerRecord<K, V> interceptRecord = record;
        for (ProducerInterceptor interceptor: this.interceptors) {
                        try {
                            interceptRecord = interceptor.onSend(interceptRecord);
                        } catch (Throwable t) {
                            // do not propagate interceptor exception, ignore and continue calling other
interceptors
                            log.warn("Error executing interceptor onSend callback for topic: " + record.
topic() + ", partition: " + record.partition(), t);
                        }
        }
        return interceptRecord;
    }

    public void onAcknowledgement(RecordMetadata metadata, Exception e) {
                for (ProducerInterceptor<K, V> interceptor: this.interceptors) {
                    try {
                        interceptor.onAcknowledgement(metadata, exception);
                    } catch (Throwable t) {
                        // do not propagate interceptor exceptions, just ignore
                        log.warn("Error executing interceptor onAcknowledgement callback", t);
                    }
                }
    }

    @Override
    public void close() {
        for (ProducerInterceptor<K,V> interceptor: this.interceptors) {
                        try {
                            interceptor.close();
                        } catch (Throwable t) {
                            log.error("Failed to close producer interceptor ", t);
                        }
        }
    }
}
```

- KafkaProducer will have a new member:
    - `ProducerInterceptors<K, V> interceptors;`
- KafkaProducer constructor will load instances of interceptor classes specified in *interceptor.classes.* If *interceptor.classes* config does not list any interceptor classes, interceptors list will be empty. It will call configure() on each interceptor class, passing in ProducerConfig.originals(). KafkaProducerconstructor will instantiate 'interceptors' with a list of interceptor classes.
- To be able to call interceptor on producer callback, we wrap client callback passed to KafkaProducer.send() method inside *ProducerCallback* – a new class that inherits *Callback* and will have a reference to client callback and 'interceptors'. ProducerCallback.onCompletion() implementation will call client's callback onCompletion (if client's callback is not null) and will call 'interceptors' onAcknowledgement().

```
/**
 * This class is a callback called on every producer request complete.
 */
public class ProducerCallback<K, V> implements Callback {
    private final Callback clientCallback;
    private final ProducerInterceptors<K, V> interceptors;

    public ProducerCallback(Callback clientCallback, ProducerInterceptors<K, V> interceptors) {
        this.clientCallback = clientCallback;
        this.interceptors = interceptors;
    }

    public void onCompletion(RecordMetadata metadata, Exception e) {
        interceptors.onAcknowledgement(metadata, e);
        if (clientCallback != null)
            clientCallback.onCompletion(metadata, e);
    }
}
```

- KafkaProducer.send() will create ProducerCallback and call onSend() method.

```
producerCallback = new ProducerCallback(callback, this.interceptors);

ProducerRecord<K, V> sentRecord = interceptors.onSend(record);
```

- The rest of KafkaProducer.send() code will use sendRecord in place of 'record'.
- KafkaProducer.close() will close interceptors:

```
ClientUtils.closeQuietly(interceptors, "producer interceptors", firstException);
```

Kafka Consumer changes

- We will create a new class that will encapsulate a list of ConsumerInterceptor instances: *ConsumerInterceptors*

ConsumerInterceptors

```
/**
 * This class wraps custom interceptors configured for this consumer. On this callback, all consumer
interceptors
 * configured for the consumer are called.
 */
public class ConsumerInterceptors<K, V> implements Closeable {
    private final List<ConsumerInterceptor<K,V>> interceptors;

    public ConsumerInterceptors(List<ConsumerInterceptor<K,V>> interceptors) {
        this.interceptors = interceptors;
    }

    public void onConsume(ConsumerRecords<K, V> records) {
        ConsumerRecords<K, V> interceptRecords = records;
        for (ConsumerInterceptor<K,V> interceptor: this.interceptors) {
                        try {
                            interceptRecords = interceptor.onConsume(interceptRecords);
                        } catch (Throwable t) {
                            // do not propagate interceptor exception, ignore and continue calling other
interceptors
                            log.warn("Error executing interceptor onConsume callback", t);
                        }
                }
        return interceptRecords;
    }

    public void onCommit(Map<TopicPartition, OffsetAndMetadata> offsets) {
        for (ConsumerInterceptor<K,V> interceptor: this.interceptors) {
                        try {
                            interceptor.onCommit(offsets);
                        } catch (Throwable t) {
                            // do not propagate interceptor exception, just ignore
                            log.warn("Error executing interceptor onCommit callback", t);
                        }
        }
    }

    @Override
    public void close() {
        for (ConsumerInterceptor<K,V> interceptor: this.interceptors) {
                        try {
                            interceptor.close();
                        } catch (Throwable t) {
                            log.error("Failed to close consumer interceptor ", t);
                        }
        }
    }
}
```

- KafkaConsumer will have a new member

  ```
  ConsumerInterceptors<K, V> interceptors;
  ```

- KafkaConsumer constructor will load instances of interceptor classes specified in *interceptor.classes.* If *interceptor.classes* config does not list any interceptor classes, interceptors list will be empty. It will call configure() on each interceptor class, passing in ConsumerConfig.originals() and clientId. KafkaConsumer constructor will instantiate 'interceptors' with a list of interceptor classes.
- KafkaConsumer.close() will close 'interceptors':

  ```
  ClientUtils.closeQuietly(interceptors, "consumer interceptors", firstException);
  ```

- KafkaConsumer.poll will call
  `this.interceptors.onConsume(consumerRecords);`

  and return ConsumerRecords<K, V> returned from onConsume().
- ConsumerCoordinator.commitOffsetsAsync and commitOffsetsSync will call onCommit().

Compatibility, Deprecation, and Migration Plan

It will not impact any of existing clients. When clients upgrade to new version, they do not need to add *interceptor.classes* config.

Future compatibility. When/if new methods will be added to ProducerInterceptor and ConsumerInterceptor (as part of other KIP(s)), they will be added with an empty implementation to the Producer/ConsumerInterceptor interfaces. This is a new feature in Java 8.

Rejected Alternatives

Alternative 1 - Interceptor interfaces on the broker

This KIP proposes interceptors only on producers and consumers. Adding message interceptor on the broker makes a lot of sense, and will add more detail to monitoring. However, the proposal is to do it later in a separate KIP for the following reasons:

- Broker interceptors are more risky because brokers are more sensitive to overheads that could be added by interceptors. Added performance overhead on brokers would affect all clients.
- Producer and consumer interceptors are less risky, and give us good risk vs. reward tradeoff, since producer and consumer interceptors alone will enable end-to-end monitoring.
- As a result, it is better to start with producer and consumer interceptors and gains experience to see how usable they are.
- Once we see usability from experience with producer and consumer interceptors, we can create a broker interceptor KIP, which will allow us to have a more complete/detailed message monitoring.

Alternative 2 – Interceptor callbacks that expose internal implementation of producer/consumer

The producer and consumer interceptor callbacks proposed in this KIP are fundamental aspects of producer and consumer protocol, and they don't depend on implementation of producer and consumer. In addition to the proposed methods, it may be useful to add more hooks such as ProducerInterceptor.onEnqueue (called before adding serialized key and value to the accumulator) or producerInterceptor.onDequeue(). They can be useful, but have disadvantage of exposing internal implementation. This can be limiting as changing internal implementation in the future may require changing the interfaces.

We can add some of these methods later if we find concrete use-cases for them. For the use-cases raised so far, it was not clear whether they should be implemented by interceptors or by other means. Examples:

- Use onEnqueue() and onDequeue() methods to measure fine-grain latency, such as serialization latency or time records spend in the accumulator. However, the insights into these latencies could be provided by Kafka Metrics.
- Encryption. There are several design options here. One is per-record encryption which would require adding ProducerInterceptor.onEnqueued() and ConsumerInterceptor.onReceive(). One could argue that in that case encryption could be done by adding a custom serializer/deserializer. Another option is to do encryption after message gets compressed, but there are issues that arise regarding broker doing re-compression. Thus, it is not clear yet whether interceptors are the right approach for adding encryption.

Alternative 3 – Wrapper around KafkaProducer and KafkaConsumer.

Some monitoring can be done (such as using unique ID for end-to-end tracing) by using a wrapper around KafkaProducer and KafkaConsumer. he wrappers could catch the events at similar points as KafkaProducer.onSend() and onAcknowledgement() and KafkaConsumer.onConsume and onCommit:

1. Requires changes in clients to use the wrappers to KafkaConsumer and KafkaProducer
2. Will not be able to catch events at intermediate stages of a request lifetime in KafkaConsumer and KafkaProducer.