KIP-210 - Provide for custom error handling when Kafka Streams fails to produce

This KIP is aimed at improving the error-handling semantics in Kafka Streams when Kafka Steams fails to produce a message to the downstream sink by providing an interface that can provide custom massaging of the error (e.g. report to a custom metrics system) and indicate to Streams whether or not it should re-throw the Exception, thus causing the application to fall over.

Status

Current state: *Adopted (1.1.0)*

Discussion thread: Click here

JIRA: KAFKA-6086

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

Motivation

At MailChimp, we've run into occasional situations where a message that comes into streams just under the size limit on the inbound size (say for the sake of illustration, 950KB with a 1MB `max.request.size` on the Producer) and we change it to a different serialization format for producing to the destination topic. In these cases, it's possible that the serialization format we change to comes in as larger than the inbound message. (For example, if we were going from a binary format to JSON we might get something much larger on the outbound side.)

These cases are rare, but when they occur they cause our entire application to fall over and someone gets woken up in the middle of the night to figure out how to deal with it. Further, solutions that address this issue by hacking around it (increasing the max.request.size or trying to manually commit to the offsets topic to skip the large messages) each have their own problems. It would be preferable for us to be able to optionally provide code to ignore an `ApiException` returned from the producer. Such an interface would permit us to provide code that will log an error and instruct Streams to not re-throw the error.

Public Interfaces

We are proposing the addition of:

- A public enumeration, `ProductionExceptionHandlerResponse`, with two possible values: `CONTINUE` and `FAIL`
- A public interface named `ProductionExceptionHandler` with a single method, `handle`, that has the following signature:
  - `ProductionExceptionHandlerResponse handle(ProducerRecord<byte[], byte[]> record, Exception exception)`
- One default implementation of ProductionExceptionHandler
  - The `DefaultProductionExceptionHandler`, the default implementation that maintains the current behavior of always failing when production exceptions occur.
- A new configuration parameter for Streams named `default.production.exception.handler` that accepts the fully qualified class name of the `ProductionExceptionHandler` to use.

Proposed Changes

This implementation will modify the `KafkaStreams` constructor to create a `ProductionExceptionHandler` from the aforementioned config value, defaulting to a default implementation that always re-throws the error (the `DefaultProductionExceptionHandler` mentioned above). We'll pipe this processor through the `StreamThread`/`StreamTask` into `RecordCollectorImpl`.

We'll implement the following error handling logic to the onCompletion handler in RecordCollectorImpl:

1. If the Exception that is thrown is a `ProducerFencedException`, behave as we do today and **do not** invoke the `ProductionExceptionHandler` as these exceptions are self-healing.
2. If the Exception that is thrown is fatal will affect *all* records and should cause Streams to *always* fail. If so, then **do not** invoke the `ProductionExceptionHandler` because its result will have to be ignored. We should log that we're ignoring these exceptions at DEBUG level.
   a. The exceptions that meet this classification are:
      i. `AuthenticationException`
      ii. `AuthorizationException`
      iii. `SecurityDisabledException`
      iv. `InvalidTopicException`

        v. `UnknownServerException`
        vi. `IllegalStateException`
        vii. `OffsetMetadataTooLarge`
        viii. `SerializationException`
        ix. `TimeoutException` when it occurs immediately on send due to a full buffer
3. If the `Exception` that is thrown meets neither of the above conditions, determine if `sendException` is already set. If so, **do not** invoke the `ProductionExceptionHandler` because this would mean that we've already invoked it and decided to `FAIL`. Invoking it again would just result in an ignored result.
4. If none of the conditions above is met, invoke the `handle` method in the `ProductionExceptionHandler` and check the result.
    a. If the result is `CONTINUE`, log a note at `DEBUG` that we received that result and are not failing Streams as a result. This ensures that it's not possible for a client developer to ship code that totally swallows errors without presenting any kind of activity in the log.
    b. If the result is `FAIL`, log a message at `ERROR` that we received that result and set `sendException` so Streams will fail.

The error handler will *only* be invoked for exceptions that are returned via the producer callback, and **will not** be invoked for Exceptions thrown directly from send as all of those exceptions should be seen by Streams immediately.

These changes will facilitate a number of error handling scenarios. For example, one could choose to write an interface that always fails, but does some additional logging in the process:

```
class ExtraLoggingProductionExceptionHandler extends ProductionExceptionHandler {
  ProductionExceptionHandlerResponse handle(ProducerRecord <byte[], byte[]> record, Exception exception) {
    val keyString = new String(record.key(), "UTF-8");
    logger.error("Got an error! Key: " + keyString, exception);
    return ProductionExceptionHandlerResponse.FAIL;
  }
}
```

You could also create a similar interface that just continues processing and logs a warning:

```
class ExtraLoggingProductionExceptionHandler extends ProductionExceptionHandler {
  ProductionExceptionHandlerResponse handle(ProducerRecord <byte[], byte[]> record, Exception exception) {
    val keyString = new String(record.key(), "UTF-8");
    logger.warn("Got an error! Key: " + keyString, exception);
    return ProductionExceptionHandlerResponse.CONTINUE;
  }
}
```

Compatibility, Deprecation, and Migration Plan

The default behavior here will be consistent with existing behavior. Changing that behavior will be opt-in by providing the new config setting and an implementation of the interface. Constructors of `RecordCollectorImpl`, `StreamThread`, and `StreamTask` will need to change, but as those aren't (to my knowledge) part of the public interface, so that should be fine. We could even provide overloaded constructors with the old signatures if we're concerned about binary compatibility of this change.

Rejected Alternatives

We also considered:

- A very targeted config setting that would ignore record too large errors, but feel that this solution is better because it could also be used to do granular reporting to other services on any kind of exception that could come from the completion handler.