KIP-905: Broker interceptors

Status

Current state: *Under Discussion*

Discussion thread: *here*

JIRA: KAFKA-14700

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

Motivation

Most large enterprises with sizable Kafka deployments support a large number of polyglot clients. Today, enforcing platform-wide conventions and best practices in these decentralized environments - e.g. schema validation, serialization, privacy enforcement, etc. -, either force platform teams to distribute custom logic in client library extensions *and/or* maintain special-purpose applications that enforce standards across the platform. Neither of these approaches are without their challenges. I will illustrate this with some examples below.

Library extension

Managing a set of library extensions in multiple languages is a non-trivial task that poses a variety of challenges:

1. Having to reimplement the same feature across multiple languages can create substantial development overhead for platform teams.
2. Duplicate implementations can also increase the risks of faulty releases, inconsistencies in implementation and regressions.
3. Moreover, any change in platform standards or best practices is very likely to impose migration costs on client teams, who will need to bump their apps to new library versions.
4. Having to rely on clients to migrate can significantly delay the rollout of important new features. This is particularly true in the case of changes that break existing producer-consumer contracts.
5. Even if a platform team succeeds in building robust library extensions, it is very hard to ensure that clients only ever use Kafka through these libs.

Special-purpose platform apps

Another popular pattern for enforcing platform standards and distributing features to all clients is through the deployment of what I refer to as (special-purpose) platform apps. Think of the stream processing application that performs some sort of stateless transformation on *every* message, and writes the result back to Kafka. In many instances, this architecture might be warranted (e.g. if the computations are expensive or the logic is complex). But in a lot of cases, these systems emerged as a way to work around the library extension pattern's limitations and allow platform owners to deploy features at scale without managing the lifecycle of library extensions. Unfortunately, using platform apps to enforce standards is an expensive strategy with rather poor scalability features. For instance, if a company decides to solve privacy enforcement through a platform app that redacts every Kafka message in-flight, the system will double disk space, the number of partitions and require a large amount of additional compute.

A third way: broker interceptors

This KIP wants to provide a third way for platform owners: the ability to extend Kafka's server-side behavior through broker interceptors. **At their core, broker interceptors can be thought of as very lightweight, stateless stream processors that can intercept, mutate and filter messages either at produce or consume time.**

The interceptor (or module) pattern is very common in the OSS community. Nginx offers modules for extending the proxy functionality with custom logic. Kubernetes uses custom resources. In the streaming world, Redpanda supports server-side processing through WASM.

To date, Kafka hasn't released a comparable feature to OSS users, and this KIP wants to to change that through the addition of broker interceptors to the stack.

Broker interceptors would allow platform owners to either fully move the enforcement of certain messaging standards to the brokers, or to perform two-stage rollouts where new features are first deployed server-side, until the corresponding library extensions are ready and rolled out.

Of course, this design isn't without its challenges and pitfalls (primarily concerning performance). When defining interceptors, operators would need to exercise good judgement to avoid causing cluster-wide performance degradation.

Prior art

- Server-side schema validation is supported by the Confluent server, which presupposes a similar feature
- https://cwiki.apache.org/confluence/display/KAFKA/KIP-686%3A+API+to+ensure+Records+policy+on+the+broker
- https://cwiki.apache.org/confluence/display/KAFKA/KIP-729%3A+Custom+validation+of+records+on+the+broker+prior+to+log+append

Public Interfaces

ProduceRequestInterceptor interface

```
/**
 * ProduceRequestInterceptors can be defined to perform custom, light-weight processing on every record
received by a
 * broker.
 *
 * Broker-side interceptors should be used with caution:
 *  1. Processing messages that were sent in compressed format by the producer will need to be decompressed and
then
 *    re-compressed to perform broker-side processing
 *  2. Performing unduly long or complex computations can negatively impact overall cluster health and
performance
 *
 * Potential use cases:
 *   - Schema validation
 *   - Privacy enforcement
 *   - Decoupling server-side and client-side serialization
 */
public abstract class ProduceRequestInterceptor {
    // Custom function for mutating the original message. If the method returns a
ProduceRequestInterceptorSkipRecordException,
    // the record will be removed from the batch and won't be persisted in the target log. All other exceptions
are
    // considered "fatal" and will result in a request error
    public abstract ProduceRequestInterceptorResult processRecord(byte[] key, byte[] value, String topic, int
partition, Header[] headers) throws Exception;

    // Define the topic name pattern that will determine whether this interceptor runs on a given batch of
records
    public abstract Pattern interceptorTopicPattern();

    // Method that gets called during the interceptor's initialization to configure itself
    public abstract void configure();
}
```

The main new interface required for produce-time interceptors is the abstract `ProduceRequestInterceptor` class. User-defined implementations of this class would serve as a container for lightweight server-side record pre-processing. The class' `processRecord` method enables three basic types of operations:

1. Map: Mutate the original payload, and return the updated key and value in a `ProduceRequestInterceptorResult` object
2. Filter: Remove a record from the batch by throwing a `ProduceRequestInterceptorSkipRecordException` from `processRecord`
3. Side-effect: Call an external system and keep the original record batch intact

Once a user-defined implementation of `ProduceRequestInterceptor` is compiled and added to the classpath of the Kafka runtime, it could be registered as an interceptor with the help of three new broker-side config options:

- **produce.request.interceptors:** A list of class names that implement the `ProduceRequestInterceptor` interface and should be loaded from the classpath during startup.
- **produce.request.interceptors.timeout.ms:** The total amount of time in ms that produce request interceptors have to finish processing a request.
- **produce.request.interceptors.max.timeout.retries:** The number of times that interceptor processing can be retried in the face of timeouts. Once the retries have been exhausted, a timeout will result in a `REQUEST_TIMEOUT` error

New errors

I'm proposing the definition of a new error response that is returned by the server if the produce request interceptor encounters an unexpected error.

```
PRODUCE_REQUEST_INTERCEPTOR_FAILED(110, "The registered produce request interceptors were unable to
successfully process the request", ProduceRequestInterceptorUnhandledException::new)
```

Proposed Changes

See this draft PR for a good overview.

Compatibility, Deprecation, and Migration Plan

As a completely new opt-in feature, adding support for broker interceptors should be fully backward compatible with previous versions.

Test Plan

This PR already contains a good number of integration tests, though a few important ones are still missing (e.g. a produce request that contains data for multiple partitions). More tests could also be added to cover the behavior of `ProduceRequestInterceptorManager` 's methods.

Rejected Alternatives

N/A