KIP-739: Block Less on KafkaProducer#send

Status

Current state: "Under Discussion"

Discussion thread: here

JIRA: here

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

Motivation

KafkaProducer#send returns a java.util.concurrent.Future, but it also blocks when it can't get partition metadata, or when the queue is full. This is a surprising user experience, since typically methods that return Future are asynchronous, and do not block. This can clash with other asynchronous systems, like Netty, Finagle, or gRPC, where the assumption is that you do not block the IO loop, and where blocking can cause deadlocks for other requests.

Public Interfaces

This section is a WIP while we try to iterate on a good solution

KafkaProducer#send (add an overload)

Producer#send (add an overload)

Proposed Changes

There are two things that make this really tricky. First, KafkaProducer#send has an existing contract that we don't want to break, which is to preserve the order of the records that it sends. Second, we need to figure out which thread is going to be responsible for waiting for metadata, or for waiting for space to be free in the queue.

There are many plausible approaches here for each solution. We'll break them down by problem.

I've added an asterisk * next to the solutions that I think are the most pragmatic.

Preserving the contract:

*Add another queue

When we encounter a situation where we would need to block, we can instead add a continuation to a single additional intermediate queue that must be processed in order.

Pros:

1. Solves the problem completely, substantially reduces complexity of user handling
2. Preserves semantics exactly

Cons:

1. Makes it harder to both reason about and measure queueing in KafkaProducer
2. Adds complexity to the implementation
3. May block unaffected topics unnecessarily

Block on metadata in the constructor

For the blocking problem related to metadata, it's something that needs to be done a single time, and then can be cached, so it would be nice to be able to do it on construction. However, we don't have the full information we need to fetch the metadata at construction yet

Pros:

1. Nice API

Cons:

1. Risks blocking on construction
2. Doesn't solve the problem wrt the send queue filling up

Add per-topic queues

When we encounter a situation where we would need to block, we can instead add a continuation to a queue that's appropriate for that specific topic that must be processed in order.

Pros:

1. Solves the problem completely, substantially reduces complexity of user handling
2. Preserves semantics exactly

Cons:

1. Makes it harder to both reason about and measure queueing in KafkaProducer
2. Adds a lot of complexity to the implementation

Update RecordAccumulator to keep track of asynchronous information

When we need to write to a queue but we don't have the relevant metadata information yet, we can pass a Future<TopicPartition>, instead of passing the concrete TopicPartition. Then we can block sending until we know the TopicPartition.

Pros:

1. Solves the problem completely, substantially reduces complexity of user handling
2. Preserves semantics exactly

Cons:

1. Adds complexity to the implementation, potentially quite invasive
2. Doesn't handle the "queue is too full" case
3. Needs to update the RecordAccumulator API and implementation

Add a per-send timeout

This simplifies doing retries on the client-side. It makes it possible to try a send with an immediate retry, and then to trigger a retry in a different thread with a longer timeout without having to maintain multiple clients. The downside of this approach is that it forces the customer to change their API to Future<Future<RecordMetadata>> to do this safely, so it still creates work for the customer, but it is a much simpler retrying story than the current "safe" one, which requires that the customer implement reasonable backoffs on 0-duration sends.

Pros:

1. Minimally invasive
2. Preserves semantics exactly

Cons:

1. Pushes most of the complexity to the user

Change the API to return Future<Future<RecordMetadata>>

This allows the API to better-represent what's going on, which is that there are two potentially blocking things that we want to represent, and we want to signal when each of them is done.

Pros:

1. Easy to implement
2. Solves the problem completely

Cons:

1. Massive breaking change

Do nothing

Keep the status quo.

Pros:

1. Trivial to implement
2. Preserves semantics exactly

Cons:

1. The user keeps all of the complexity
2. Confusing API that's difficult to use correctly

Which thread should be responsible for waiting?

*Default to Common Pool, Allow User Overrides

By default, we will use a common threadpool (either one managed by Kafka, or by the JDK (like FJP.commonpool)). However, it will be possible for a user to override the pool.

Pros:

1. Easy to implement
2. Users have fine-grained control if they want, or they can relax and trust us

Cons:

1. Breaking API change, if only for the user override.

User-provided Threadpool

Update the API so that the user provides a threadpool, in which we do the blocking.

Pros:

1. Easy to implement

Cons:

1. Breaking API change
2. User needs to think about the thread pool

Kafka Threadpool

Change it so that it's truly asynchronous, and that the Kafka IO loop does the next chunk of work when it's ready.

Pros:

1. Doesn't require managing additional threads
2. No change in API
3. User doesn't need to think about the thread pool

Cons:

1. Extremely invasive, and challenging to implement and review

Common Threadpool (eg ForkJoin common pool)

Use a common threadpool that's provided by the JDK, like the ForkJoin common pool to take care of the blocking.

Pros:

1. No change in API
2. User doesn't need to think about the thread pool

Cons:

1. Reduces user control over thread pools
2. No real backpressure

User Thread

Keep the status quo.

Pros:

1. Trivial to implement
2. No change in API

Cons:

1. The user keeps all of the complexity
2. Confusing API that's difficult to use correctly

Compatibility, Deprecation, and Migration Plan

TBD. It's too early to discuss now.

- *What impact (if any) will there be on existing users?*
- *If we are changing behavior how will we phase out the older behavior?*
- *If we need special migration tools, describe them here.*
- *When will we remove the existing behavior?*

Rejected Alternatives

Prior Art: KIP-286

Everything is on the table! We can add things here as we reject them.