KIP-91 Provide Intuitive User Timeouts in The Producer

Status

Current state: *Adopted*

Discussion thread: [DISCUSS] KIP-91

Vote thread: [VOTE] KIP-91

JIRA: KAFKA-5886

Release: 2.1.0

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

Motivation

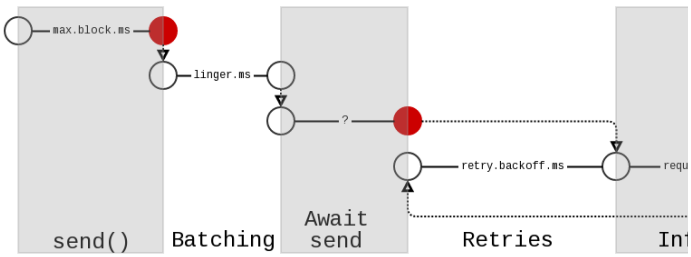
In KIP-19, we added a request timeout to the network client. This change was necessary **primarily** to bound the time to detection of broker failures. In the absence of such a timeout, the producer would learn of the failure only much later (typically several minutes depending on the TCP timeout) during which the accumulator could fill up and cause requests to either block or get dropped depending on the `block.on.buffer.full` configuration. One additional goal of KIP-19 was to make timeouts intuitive. It is important for users to be provided with a guarantee on the maximum duration from when the call to `send` returns and when the callback fires (or future is ready). Notwithstanding the fact that intuition is a subjective thing, we will see shortly that this goal has not been met.

In order to clarify the motivation, it will be helpful to review the lifecycle of records and record-batches in the producer, where the timeouts apply, and changes that have been made since KIP-19.

Behavior in KIP-19

- The initial call to `send` can block up to `max.block.ms` either waiting on metadata or for available space in the producer's accumulator. After this the record is placed in a (possibly new) batch of records.
- The batch is eligible to be considered for sending when either `linger.ms` or `batch.size` bytes has been reached, whichever comes first. Although the batch is *ready*, it does not necessarily mean it can be sent out to the broker.
- The batch has to wait for a transmission opportunity to the broker. A ready batch can only be sent out if the leader broker is in a *sendable* state (i.e., if a connection exists, current inflight requests are less than `max.inflight.requests`, etc.). In KIP-19, we use the `request.timeout.ms` configuration to expire requests in the accumulator as well. This was done in order to avoid an additional timeout, especially one that exposes the producer's internals to the user. The clock starts ticking when the batch is *ready*. However, we added a condition that if the metadata for a partition is known (i.e., it is possible to make progress on the partition) then we do not expire its batches even if they are ready. *In other words, it is difficult to precisely determine the duration spent in the accumulator.* Note that KIP-19 claims that *"The per message timeout is easy to compute - linger.ms + (retries + 1) * request.timeout.ms".* This is false.
- When the batch gets sent out on the wire, we reset the clock for the actual wire timeout `request.timeout.ms`.
- If the request fails for some reason before the timeout and we have retries remaining, we reset the clock again. (i.e., each retry gets a full `request.timeout.ms`.)

The following figure illustrates the above phases. The red circles are the potential points of timeout.



Change in KAFKA-2805 (to handle cluster outages)

One problem with the implementation of KIP-19 was that it did not check if metadata is stale or not. So for example, if the cluster suddenly becomes unavailable, the producer would never expire batches if it already has metadata available. So in KAFKA-2805 we completely removed the check on availability of metadata and indiscriminately expire batches that are ready and have remained in the accumulator for at least `request.timeout.ms` even if the leader broker is available.

Further change in KAFKA-3388 (to handle pessimistic timeouts and out-of-order callbacks when `max.inflight.requests == 1`)

The complete removal of the metadata availability check in KAFKA-2805 was problematic in that it:

1. Leads to unfair/unnecessary timeouts especially when preceding batches that are inflight encounter retries. (Unfair because those batches are given another `request.timeout.ms` to expire.)
2. Can cause callbacks to fire out of order when strict ordering is required. i.e., when accumulator batches expire, their callbacks fire before callbacks for inflight batches (that are actually preceding batches to the batches in the accumulator). Note that this is really an issue only for `max.inflight.requests == 1` since we don't attempt to make any strict ordering guarantees for other inflight settings.
3. Pessimistically expires batches even though it may be possible to make progress. (KIP-19 takes an optimistic view on the other hand - i.e., do not expire batches if metadata is available since we may be able to make progress.)

So KAFKA-3388 added a check (for the `max.inflight.requests == 1` scenario only) on the inflight request queue and only expires batches if there is currently no inflight request.

There are still pessimistic timeouts (KAFKA-4089)

One problem with the above incremental change is the way in which we check whether there is any inflight request. Since it only applies to the scenario where `max.inflight.requests == 1` we check if the partition is *muted* or not. (We mute partitions when a batch is inflight for that partition in order to ensure ordering even during leader movements - see KAFKA-3197 for more details on that.) The issue though is that if a metadata request is inflight (say, due to a normal metadata refresh) the partitions on that broker will not be in a muted state (since it is not a batch that is inflight) and can expire if they have been sitting in the accumulator for at least `request.timeout.ms`. This is an unintuitive side-effect given that they would otherwise have been sent out (had the metadata refresh not occurred).

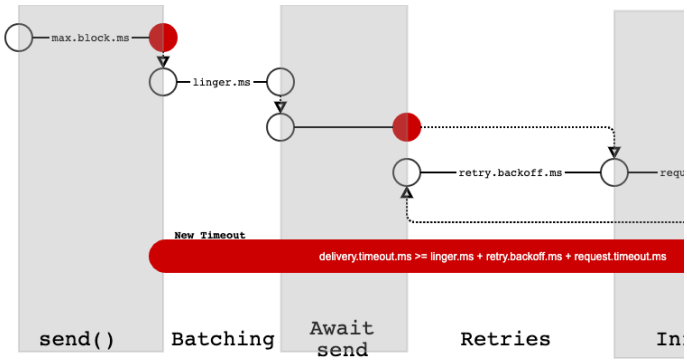
This is actually a highly probable scenario in the case of a high-volume producer that sets `max.inflight.requests` to one such as the mirror maker and leads to an unnecessary shutdown of the mirror maker.

It is possible to bump up the `request.timeout.ms` but that is undesirable as explained in the rejected alternatives section.

Note also that it is insufficient to tweak the above check to see if it is a metadata request that is inflight because the problem is more fundamental than that: we would like to keep the `request.timeout.ms` to be relatively small (at most a minute or so). If we continue to reuse `request.timeout.ms` for timing out batches in the accumulator it is highly likely for a high volume producer to expire several batches even in routine scenarios such as bouncing the cluster that the producer is sending to. E.g., if a broker is disconnected due to a bounce then metadata will still be available, but there will be no inflight request to that broker and so several batches that have been in the accumulator for more than `request.timeout.ms` will get expired. This would be fine if the accumulator timeout is large enough to account for the expected time batches will sit in the accumulator, but this could be high for a high-volume producer. In the absence of an explicit accumulator timeout the only option here is to artificially bump up `request.timeout.ms`.

Proposed Changes

We propose adding a new timeout `delivery.timeout.ms`. The window of enforcement includes batching in the accumulator, retries, and the inflight segments of the batch. With this config, the user has a guaranteed upper bound on when a record will either get sent, fail or expire from the point when send returns. In other words we no longer overload `request.timeout.ms` to act as a weak proxy for accumulator timeout and instead introduce an explicit timeout that users can rely on without exposing any internals of the producer such as the accumulator.



This config enables applications to delegate error handling to Kafka to the maximum possible extent (by setting retries=MAX_INT and delivery.timeout.ms=MAX_LONG). And it enables MirrorMaker to bound the effect of unavailable partitions by setting delivery.timeout.ms to be sufficiently low, presumably some function of the expected throughput in the steady state. Specifically, setting delivery.timeout.ms to a minimum of request.timeout.ms + retry.backoff.ms + linger.ms, would allow at least one attempt to send the message when the producer isn't backed up.

The "timer" for each batch starts "ticking" at the creation of the batch. Batches expire in order when max.in.flight.request.per.connection==1. An in-flight batch expire when delivery.timeout.ms has passed since the batch creation irrespective of whether the batch is in flight or not. However, the producer waits the full request.timeout.ms for the in-flight request. This implies that user might be notified of batch expiry while a batch is still in-flight.

Public Interfaces

- Add a new producer configuration `delivery.timeout.ms` with default value 120 seconds.
- Change the default value of retries to MAX_INT.
- request.timeout.ms–no changes in the meaning, but messages are not expired after this time. I.e., request.timeout.ms is no longer relevant for batch expiry.

Compatibility, Deprecation, and Migration Plan

Setting an explicit value of retries should be done with caution. If all the retries are exhausted, the request will fail and all the contained batches within the request (even if delivery.timeout.ms has not fully elapsed). In essence, batch expires when either delivery.timeout.ms has elapsed or the request containing the batch has failed, whichever happens first. (Note: Due to change in the default value of retries from 0 to MAX_INT and the existing default value of max.in.flight.request.per.connection==5, reordering becomes a possibility by default. To prevent reordering, set max.in.flight.request.per.connection==1).

Validation

This configuration is backwards compatible. Throw ConfigException for timeouts that don't make sense. (E.g., delivery.timeout.ms `< linger.ms + request.timeout.ms + retry.backoff.ms`).

Test Plan

- Additional unit tests in o.a.k.c.p.i.RecordAccumulatorTest.
- TBD

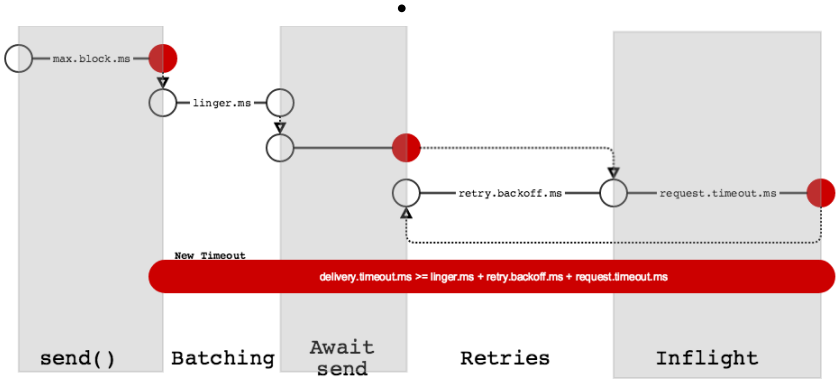
Rejected Alternatives

- Bumping up request timeout does not work well because that is an artificial way of dealing with the lack of an accumulator timeout. Setting it high will increase the time to detect broker failures.
- In KAFKA-4089 we also considered looking at whether metadata is stale or not to determine whether to expire. This may work to address the problem raised in KAFKA-4089 but is still hard for users to understand without understanding internals of the producer and makes it difficult to put an upper bound on the overall timeout.
- We cannot repurpose `max.block.ms` since there are use-cases for non-blocking calls to send.
- We also discussed the ideal of providing precise per-record timeouts or at least per-batch timeouts. These are very difficult to implement correctly and we believe it is sufficient to provide users with the ability to determine an upper bound on delivery time (and not specify it on a per-record level). Supporting per-record timeouts precisely is problematic because we would then need the ability to extract records from compressed batches which is horribly inefficient. The difficulty is even more pronounced when requests are inflight since batches need to be extracted out of inflight requests. If we are to honor the retry backoff settings then this would mean that we have to split an inflight request with an expiring record or batch into smaller requests which is again horribly inefficient. Given the enormous complexity of implementing such semantics correctly and efficiently, and the limited value to users we have decided against pursuing this path. The addition of an explicit timeout as summarized in this proposal will at least give the users the ability to come up with a tight bound on the maximum delay before a record is actually sent out.
- Allow `batch.expiry.ms` to span the inflight phase as well. This won't work because a request would contain batches from multiple partitions. One expiring batch should not cause the other batches to expire, and it is too inefficient to surgically remove the expired batch for the subsequent retry.
- An end-to-end timeout Model: A single end-to-end timeout for the entire send operation is a very easy to use and therefore, very compelling alternative. It's, however, not without pitfalls. The model of end-to-end timeout we considered is exclusive to segment-wise timeouts. In other words, you specify either of them but not both.

    1. An end-to-end timeout does not subsume max.block.ms because the latter is a bound on how long the application threads may block. An end-to-end timeout may only subsume time spent in accumulator (including linger.ms) and on the wire.
    2. In applications such as mirror-maker where consumed records are immediately produced on the other side, "catch-up mode" is a frequent scenario that creates additional challenges to end-to-end timeout model. A record may spent 99.95% of it's end-to-end budget in the accumulator and may not leave much budget at all for retrying over the wire. The producer may expire batches without any retries at all. An end-to-end delay of MAX_INT may be sufficient for mirror-makers but the same could not be said about a general application that has a few seconds of end-to-end timeout. It's unclear if a send failure due to timeout is due to an unavailable partition or just acccumulator wait. Exhaustion of nRetries combined with retry.backoff.ms pretty much guarantees that failure's due to service unavailability.
    3. An end-to-end timeout may be partially emulated using the future.get(timeout). The timeout must be greater than (batch.expiry.ms + nRetries * (request.timeout.ms + retry.backoff.ms)). Note that when future times out, Sender may continue to send the records in the background. To avoid that implementing a cancellable future is a possibility.
- An additional configuration called "partition.availability.budget.ms" for producing applications that don't care about end-to-end bound on message delivery (and hence don't want to configure batch.expiry.ms) but do care about partitions that never make progress. A notional partition-

unavailability-budget is useful for kafla-mirror-maker-like apps. It could be a function of retries, backoff period, and request timeout. Deemed too complicated for the benefit received. Specifically, a new mechanism is needed at the producer side that keeps track of unavailability of partitions over time. Partition unknown and partition unavailable have different nuances.

- The original proposal in this KIP from LinkedIn was to add a new timeout called `batch.expiry.ms`. The window of enforcement would be from the time `send` returns until the produce request is sent on the wire. With this change, the user has a guaranteed upper bound on when a record will either get sent, fail or expire: `max.block.ms + batch.expiry.ms + nRetries * (request.timeout.ms + retry.backoff.ms)`. In other words, `request.timeout.ms` is no longer overloaded to act as a weak proxy for accumulator timeout. This proposal introduced an explicit timeout that users can rely on without exposing any internals of the producer such as the accumulator. In the following figure, the possible timeout points are colored red. The new global timeout can occur at any point after the batch is ready.



- The delivery.timeout.ms proposal is preferred over batch.expiry.ms for the following reasons.
  - It's clearer for users to ocnfigure one number that encompases batching, await-send, and inflight segments together as opposed to having to configure multiple segments via separate configs.
  - In the delivery.timeout.ms approach, clock for a batch starts when a batch is created. Starting the clock at the beginning of a batch avoids the pitfalls of starting the clock at close. As of now, closing of batch may be arbitrarily delayed because a batch is closed only when the
    batch is sendable (i.e., broker is available, inflight request limit is not exceeded, etc). The possibility of unbounded delay in closing a batch is incompatible with the goal of this kip.
  -