KIP-62: Allow consumer to send heartbeats from a background thread

Status

Current state: *Adopted*

Discussion thread: http://markmail.org/message/oeg63goh3ed3qdap

JIRA: **KAFKA-3888** - Getting issue details... STATUS

Released: 0.10.1.0

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

Motivation

A common issue the new consumer is the mix of its single-threaded design and the need to maintain liveness by sending heartbeats to the coordinator. We recommend users do message processing and partition initialization/clean from the same thread as the consumer's poll loop, but if this takes longer than the configured session timeout, the consumer is removed from the group and its partitions are assigned to other members. In the best case, this causes unneeded rebalancing when the consumer has to rejoin. In the worst case, the consumer will not be able to commit offsets for any messages which were handled after the consumer had fallen out of the group, which means that these messages will have to be processed again after the rebalance. If a message or set of messages *always* takes longer than the session timeout, then the consumer will not be able to make progress until the user notices the problem and makes some adjustment.

The options for the user at the moment to handle this problem are the following:

1. Increase the session timeout to give more time for record processing.
2. Reduce the number of records handled on each iteration with `max.poll.records`.

The addition of `max.poll.records` in KIP-41 addressed one of the major sources of unexpected session timeouts. Before it, there was no bound on the number of messages that could be returned in a single call to `poll()`, which meant any tuning of the processing time probably had to be based on `max.partition.fetch.bytes`. With KIP-41, the idea was that users could tune `max.poll.records` to a fairly low value while also keeping `session.timeout.ms` low so that failure detection time didn't need to be sacrificed. This works when the variance in per-message handling time is relatively small, but consumers often use a batch processing consumption model in which records are collected in memory before flushing them to another system. Kafka Connect, for example, encourages this approach for sink connectors since it usually has better performance. When the user controls the batching, it can be tuned, but sometimes it is hidden in another library without a direct way to control it. This is the case in Kafka Streams where writes to RocksDB are cached in memory prior to being pushed to disk. In this case, `max.poll.records` doesn't help and users are again left trying to adjust `session.timeout.ms`, which typically means sacrificing some time to detect failures. It can be difficult in these situations to even estimate worst-case processing time without significant testing, which leads to a painful readjustment loop.

The batch processing model has particularly bad interplay with group rebalances. When a rebalance begins, the consumer must finish whatever processing it is currently doing, commit offsets, and rejoin the group before the session timeout expires. In other words, the session timeout is also used as a rebalance timeout. Even if a consumer continues sending heartbeats to the coordinator, it will be removed from the group if it cannot rejoin fast enough when a rebalance begins. Since rebalances are usually not predicted by consumers, they will typically have a non-empty batch of records which must be flushed as soon as they realize that the group has begun rebalancing. This is a general problem which all Kafka clients have to handle even if they do not take the single-threaded approach that the Java client takes. Typically the only thing that can be done is let the processing finish and raise an exception if the commit fails.

It's helpful to take a step back and reconsider the motivation for the design of the poll loop. The idea was to provide a built-in mechanism to ensure that the consumer was alive and making progress. As long as the consumer is sending heartbeats, it basically holds a lock on the partitions it was assigned. If the process becomes defunct in such a way that it cannot make progress but is nevertheless continuing to send heartbeats, then no other member in the group will be able to take over the partitions, which causes increasing lag. The fact that heartbeating and processing is all done in the same thread, however, guarantees that consumers must make progress to keep their assignment. Any stall which affects processing also affects heartbeats. This protects consumer applications not only from process failures, but also from "livelock" situations in which the process is alive, but is not making progress.

The problem of ensuring progress is fundamental to consumer applications. There is no way around it. If you take this mechanism out of the consumer, then you push it to the application monitoring infrastructure, where it is much more difficult to respond to failures. However, the current approach puts all classes of consumer failures into the same bucket by trying to govern them all with the same timeout value. While we may have an intuitive notion of what a reasonable timeout should be to detect a crashed process, it's much more difficult trying to define a timeout for positive progress without environment-dependent experimentation and tuning. And if the user finds after tuning that a session timeout of 10 minutes is needed in the worst case, then this will also be the minimum amount of time it will take to detect any kind of hard failure. This is a tough tradeoff to accept, especially since users are typically more concerned about process crashes which are outside of their direct control than livelock situations. Some users have instead chosen to implement their own asynchronous processing pattern (typically using pause/resume). This problem is also complicated by the fact that the broker enforces a maximum session timeout which may be difficult to change in some environments.

Proposed Changes

We believe the root of the problem described above is the conflation of the session timeout as

1. an indication of a failed or unreachable application;
2. the maximum time for a consumer to process a batch of records; and
3. the maximum time for the group to complete a rebalance.

Our proposal is basically to decouple these timeouts.

Decoupling the processing timeout: We propose to introduce a separate locally enforced timeout for record processing and a background thread to keep the session active until this timeout expires. We call this new timeout as the *"process timeout"* and expose it in the consumer's configuration as **`max.poll.interval.ms`**. This config sets the maximum delay between client calls to `poll()`. When the timeout expires, the consumer will stop sending heartbeats and send an explicit LeaveGroup request. As soon as the consumer resumes processing with another call to `poll()`, the consumer will rejoin the group. This is equivalent to the current processing model except that it allows the user to set a higher timeout when processing while also using a lower session timeout for faster crash detection.

As in the current implementation, rebalancing and fetching will continue to be executed in the calling thread. The background thread will be enabled automatically when the user calls `subscribe()`, which means that users depending on manual assignment will not have the additional overhead. As an optimization, we could also stop the background thread when `unsubscribe()` is called, but we doubt this is necessary since it's unlikely that users need to switch between automatic and manual assignment. The background thread is only allowed to send heartbeats in order to keep the member alive and to send the explicit LeaveGroup if the processing timeout expires. This could be achieved, for example, by synchronizing access to `ConsumerNetworkClient`.

In practice, we expect the process timeout to always be at least as large as the session timeout. If it were smaller, then the user may as well reduce the session timeout to the same value since the effect of reaching the process timeout is the same if as if the session timeout is reached (i.e. the consumer is removed from the group). It would be odd otherwise for detection of a crashed process via the session timeout to actually take more time than detection of a hung process via the process timeout. For compatibility with the existing client, we choose to use the maximum of the session timeout and the process timeout as the effective process timeout. This ensures that users who have tuned the session timeout for 0.9 and 0.10 consumers will not suddenly see a lower timeout enforced. One minor point worth mentioning is that the background thread is technically not necessary when the effective process timeout matches the session timeout. We leave this to the implementation as a possible optimization.

Decoupling the rebalance timeout: Additionally, we propose to decouple the session timeout from the time to complete a rebalance. Currently, when the group begins rebalancing, we suspend each consumer's individual heartbeat timer and start a rebalance timer which is set to expire according to the session timeout. We propose to add a separate rebalance timeout, which is passed to the coordinator in the JoinGroup request. This requires a bump of the JoinGroup version which is shown in the section below. As with the session timeout, the broker will use the maximum rebalance timeout across all members in the group (typically it is only during rolling upgrades that they will differ).

The question then is where the value for this timeout should come from. Since we give the client as much as `max.poll.interval.ms` to handle a batch of records, this is also the maximum time before a consumer can be expected to rejoin the group in the worst case. We therefore propose to set the rebalance timeout in the Java client to the same value configured with `max.poll.interval.ms`. When a rebalance begins, the background thread will continue sending heartbeats. The consumer will not rejoin the group until processing completes and the user calls `poll()`. From the coordinator's perspective, the consumer will not be removed from the group until either 1) their session timeout expires without receiving a heartbeat, or 2) the rebalance timeout expires.

There are a couple points worth discussing a bit more on the use of the rebalance timeout:

- A side effect of having a rebalance timeout which is not coupled with the session timeout is that we can handle hard crashes more gracefully. Currently, when a consumer suffers a hard crash and cannot send a LeaveGroup request, the coordinator has to depend on the expiration of the session timeout to detect the failure. If a rebalance is in progress when this happens, the coordinator is forced to wait the full session timeout before the rebalance completes which stalls the entire group. However, this change allows the user to define a much smaller session timeout so that failed process can be detected faster which allows the rebalance to also complete faster.
- By using the processing timeout as the rebalance timeout, users can use it to tune the impact of rebalances on the group. For example, to ensure that rebalances never take longer than 5 seconds, the user should ensure that each round of message processing can always complete within that time. This was also possible previously by tuning the session timeout, but users had to tradeoff failure detection. However, there is also a risk for users who are forced by uncertainty to use a larger timeout since it will cause longer rebalances in the worst case. Alternatively, if they set a lower process timeout, rebalances will complete faster, but the risk of commit failures will increase since the consumer can fall out of the group before a round of processing completes.

Public Interfaces

This KIP adds the `max.poll.interval.ms` configuration to the consumer configuration as described above. With the decoupled processing timeout, users will be able to set the session timeout significantly lower to detect process crashes faster (the only reason we've set it to 30 seconds up to now is to give users some initial leeway for processing overhead). To avoid the need for most users to tune these settings manually, we suggest the following default values for the three relevant configurations which affect the poll loop behavior:

- `session.timeout.ms`: 10s
- `max.poll.interval.ms`: 5min
- `max.poll.records`: 500

We've reduced the default session timeout, but actually increased the amount of time given to consumers for message processing to 5 minutes. We've also set a fairly conservative `max.poll.records` to give users a more reasonable default batch size to avoid the need for many users to tune it in the first place (the current default is Integer.MAX_VALUE).

This KIP also bumps the protocol version for the JoinGroup request to 1. The new version introduces a new field for the rebalance timeout:

```
JoinGroupRequest => GroupId SessionTimeout RebalanceTimeout MemberId ProtocolType GroupProtocols
  GroupId => string
  SessionTimeout => int32
  RebalanceTimeout => int32  ;;; this is new
  MemberId => string
  ProtocolType => string
  GroupProtocols => [ProtocolName ProtocolMetadata]
    ProtocolName => string
    ProtocolMetadata => bytes
```

Compatibility, Deprecation, and Migration Plan

Technically, these changes are not compatible with the current client. The main problem is the semantic difference in the session timeout, which now only controls the time to detect crashed consumer processes. We see the following cases:

1. For users who have not made any changes to the configuration, the changes will probably not be noticed, but detection of crashed processes should be faster and there will be more time allowed for processing. Both of these seem fine.
2. Some users may have already increased the session timeout in order to give more time for record processing. Since we use the maximum of the session timeout and the configured process timeout as the *effective* process timeout, the observable behavior should not be different.
3. For users who have tuned the session timeout lower, the addition of the process timeout will mean that it will take longer to detect livelock situations. We think it is unlikely that users are depending on a tight bound for this detection, so the effect of allowing more processing time should be harmless, but we can't rule out the possibility that some users depend on this timeout behavior for other purposes.

In short, although this KIP may be incompatible, the impact does not seem significant. Additionally, this change will have no impact on 0.9 and 0.10 consumers ability to work with future versions of Kafka. When receiving version 0 of the JoinGroup request, the coordinator will use the session timeout as the rebalance timeout which preserves the old behavior.

Test Plan

This KIP will be tested primarily through unit and integration testing. On the client, we need to verify max.poll.interval.ms is enforced correctly, including during rebalances. On the server, we need to verify that the rebalance timeout passed in the JoinGroup is enforced, including the case when two members use conflicting values. Since this KIP bumps the JoinGroup API version, it may also make sense to add a system test which verifies compatibility in groups with consumers using the new version and the old version.

Rejected Alternatives

1. **Add a separate API the user can call to indicate liveness:** We considered adding a `heartbeat()` API which the user could use from their own thread in order to keep the consumer alive. This also solves the problem, but it puts the burden of managing that thread (including shutdown coordination) on the user. Although there is some advantage to having a separate API since it allows users to develop their own notion of liveness, we feel must users would simply spawn a thread and call `heartbeat()` in a loop. We leave this as a possible extension for the future if users find they need it.
2. **Maybe no need for a rebalance timeout in the group protocol?** If we only introduce the background thread for heartbeating, then the session timeout could continue to be used as both the processing timeout and the rebalance timeout. This still addresses the most significant problem that users are seeing, which is the consumer falling out of the group because of long processing times. The background thread will keep the consumer in the group as long as the group is stable. However, if a rebalance begins while the consumer is processing data, then there is still the possibility of the consumer falling out of the group since it may not be able to finish processing and join the group fast enough. This scenario is actually common in practice since users often use a processing model where records are collected in memory prior to being flushed to a remote system in a single batch. In this case, once a rebalance begins, the user must flush the existing batch and then commit offsets.
3. **Perhaps we don't need max.poll.interval.ms?** We could enable the background thread through an explicit configuration and let it keep the consumer in the group indefinitely. This feels a bit like a step backwards since consumer liveness is actually an important problem which users

must face. Additionally, users can get virtually the same behavior by setting the timeout to a very large value as long as they are willing to accept longer rebalances in the worst case. Users who require both short rebalances and indefinite processing

4. **Move rebalancing to the background thread instead of heartbeats only?** In this proposal, we have intentionally left rebalances in the foreground because it greatly simplifies the implementation, and also for compatibility, since users currently expect the rebalance listener to execute from the same thread as the consumer. Alternatively, we could move *all* coordinator communication to the background thread, even allowing rebalances to complete asynchronously. The apparent advantage of doing so is that it would allow the consumer to finish a rebalance while messages are still being processed, but we're not sure this is desirable since offsets for messages which arrived before the rebalance cannot generally be committed safely after it completes (which usually necessitates reprocessing). The current proposal gives users direct control over this tradeoff. To rebalance faster, users must tune their processing loop to work with smaller chunks of data. To give more time for record processing, users must accept a longer worst-case rebalance time. Finally, this change would basically require a rewrite of a huge piece of the consumer, so we've opted for something more incremental.
5. **The rebalance timeout could be configured separately from the process timeout**: It may make sense to expose the rebalance timeout to the user directly instead of using the process timeout as we've suggested above. This might make sense if users were willing to accept some message reprocessing in order to ensure that rebalances always complete quickly. Unfortunately, the single-threaded model of the consumer means that we would have to move the rebalance completion to the background thread, which we already rejected above (see above). Also, there is no obvious reason why a user would ever want to set a rebalance timeout higher than the process timeout.