KIP-983: Full speed async processing during rebalance

Status

Current state: *Under Discussion*

Discussion thread: *here*

JIRA: No jira.

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

Design Goal

This KIP has the goal to improve the java client consumer API such that:

- it allows full speed consuming and asynchronous processing from partitions that are *not* revoked while a (cooperative) rebalance is on-going,
- it becomes easier to use the API in a way that does not cause duplicate messages,
- it becomes easier to process data asynchronously with receiving more records,
- it changes minimally such that existing users need no changes.

Motivation

Duplicate record handling happens when a partition is reassigned to another consumer, but the first consumer did not yet commit all offsets of processed records. When the second consumer starts, it starts reading at the last committed offset. However, since the first consumer did not yet commit all offsets, the second consumer will re-read some records that were already processed by the first consumer.
Hence record duplication.

Currently, there are 2 approaches to use the Kafka java client correctly, that is, in a way that does not cause record duplication during normal operation. (Duplicates are still possible due to failures.)

Approach 1 - process all records immediately (synchronous)

This approach is useful when all processing can happen between the calls to poll.

Clients that follow this approach have 1 component:

1. a poll loop which:
    a. calls consumer.poll
    b. process the received records
    c. commits offsets for the just processed records
    d. repeat

The big advantage of this approach is that it is the simplest. This is also the only approach that can be used with auto-commit.

However, this approach has the downside that asynchronous processing can not be used. Therefore, we do not consider it further in this KIP.

Approach 2 - asynchronous processing

In this approach, processing can be done asynchronously while receiving more records. As soon as processing completes, the offset can be committed.

Clients that follow this approach have the following components:

1. a poll loop which:
    a. calls consumer.poll (note: rebalance listener might be called during poll)
    b. start processing received records

    c. commits offsets for records that completed processing
    d. repeat
2. a rebalance listener. When partitions are revoked it:
    a. waits (blocking) until processing of all consumed records (from revoked partitions) has completed
    b. when the poll-loop commits (step 1.c.) were async, wait for these commits to complete
    c. synchronously commits offsets for records that completed processing

This approach looks very flexible but has some problems:

1. While the partition-revoked callback method is waiting, the poll loop is waiting for the poll to complete. **As a consequence *all* partitions can no longer make progress and total throughput plummets.** This is especially relevant for cooperative rebalancing where only a few partitions are revoked at a time.
2. Keeping track of which records are being processed can be non-trivial. This challenge is compounded by having to share this information with the rebalance listener.
3. When an async runtime (for example: Kotlin coroutines, Pekko, ZIO, Cats Effect) is used, the single-thread consumer lock makes it very hard to call the consumer from the callback. (See KIP-944.)
4. Even though processing is asynchronous, offsets must be committed in order.
5. For better throughput, offsets should be aggregated before committing.

Note that problems 4 and 5 are hard because both the poll-loop and the rebalance listener perform commits. This KIP does not solve problems 4 and 5, but it does allow for a simpler implementation.

A new approach

This KIP proposes an alternative API for handling revoked partitions.

In this KIP we recognize that a 'revoke' operation consists out of two steps:

- revoke requested — the consumer learns that the partitions must be revoked
- revoke completed — the consumer has released control of the partition, another consumer may start consuming

In addition, we recognize that a partition is lost when a consumer takes too long to release control of the partition.

Currently, when a revoke is requested (while a poll is in progress), the callback listener is immediately invoked, and the revoke is completed during the same poll.

In the new approach, when a revoke is requested in one poll, the revoke will be completed in the *next* poll.

When a poll invocation completes and a partition revoke is requested, the user has a choice:

    a. delay calling poll until the records of these partitions have been processed and their offsets committed,
    b. request the revoking to be delayed one more poll to the future, and then call poll again.

Option b. is the interesting one and the core of this KIP; by quickly calling poll again, it is possible to continue processing records from partitions that are not revoked.

Clients that use the new approach (option b) have again only 1 component:

1. a poll loop which:
    a. calls consumer.poll
    b. starts processing received records
    c. commits offsets for records that completed processing
    d. gets a list of to be revoked partitions, if any:
        i. validate that all records for those partitions have been processed,
        ii. if some records are not completely processed yet, request revoking to be delayed
    e. gets a list of lost partitions, if any:
        i. try to abort processing of records from these partitions
        ii. make sure to no longer commit anything from these partitions
    f. repeat

It is not possible to delay partition revocation indefinitely. The deadline is when the partition is lost. Once the partition is lost, the delay request is denied.

To allow a program to stop committing offsets for a lost partition, a new consumer method is introduced that gives the partitions that were lost in the previous poll. This method is used in step 1.e. Note, ideally, this does not happen because the user configures the partition assigner such that a partition is declared lost only after the maximum processing time has passed (this is outside the scope of this KIP).

Some smaller details:

- Consumer.poll could return early when a partition revoke is requested.
- Consumer.poll should never return any records from revoked partitions.

The problems listed with approach 2 now disappear:

1. The client continues to call poll in a tight loop, and keeps processing events at full speed for partitions that do not participate in the rebalance.
2. The program is simple again; only a single loop, no complex coordination with a rebalance listener.
3. Asynchronous runtimes are no longer hindered by the same-thread lock.

In addition, the API change is very small, existing clients do not need to be changed.

Public interfaces

Three new methods need to be added to the consumer:

- `Set<TopicPartition> getToBeRevokedPartitions()` gives partitions that will be revoked in the next poll.
- `Set<TopicPartition> getLostPartitions()` gives partitions that were lost during the last call to poll.
- `bool delayRevoke(tps: Set<TopicPartition>)` request revoke for given partitions to be delayed to the poll following the next poll, returns true on success, false when some of these partitions were lost.

Proposed changes

This section shows how the extension *can* be implemented. This is not necessary exactly how this KIP will be implemented.

The consumer keeps:

- a list of partitions that will be revoked in the next poll (revoke-next list)
- a list of partitions that will be revoked in the poll thereafter (revoke-thereafter list)
- a list of partitions that were lost during the last poll

During a consumer.poll:

- At the start of poll:
    1. complete the revoke of all partitions on the revoke-next list, then empty the list,
    2. empty the list of lost partitions,
    3. move partitions from the revoke-thereafter list to the revoke-next list.
- When a revoke is requested, the revoked partitions are added to the revoke-next list.
- When a partitions is lost, it is removed from the revoke-next and revoke-thereafter lists, and added to the list of lost partitions.

Method `getToBeRevokedPartitions` returns the revoke-next list.
Method `getLostPartitions` returns the list of partitions that were lost during the last poll.
Method `delayRevoke` moves the given partitions from the revoke-next list to the revoke-thereafter list.

Compatibility, deprecation, migration plan

There is a small chance we break clients that depend on a revoke to be completed in the same poll as it was requested. If this is a problem, a configuration could be introduced that restores the current behavior. Another option to mitigate this is described in variant 1 below.

When this KIP is implemented, the rebalance listener is no longer needed. However, it is assumed that it will continue to be supported for backward compatibility. Therefore, no deprecation or migration is needed.

When at some point in the future the rebalance listener has been deprecated and then removed, the same-thread lock in the consumer can be replaced by a simpler non-reentrant semaphore.

Test plan

TBD.

Variants

Variant 1

Instead of introducing methods `getToBeRevokedPartitions` and `getLostPartitions`, it is also possible to introduce a new poll method that returns all of the following: the new records, the partitions that will be revoked in the next call to poll, the partitions that were just lost. Because this is a separate poll method, it is possible to keep partition revoke behavior of the current poll method exactly the same (which is to complete partition revoke in the same poll). This prevents any backward compatibility problems at the cost of potential user confusion due there being 2 poll methods.

Rejected alternatives

None.