KIP-568: Explicit rebalance triggering on the Consumer

Status

Current state: *Accepted*

Discussion thread: *here*

JIRA: Unable to render Jira issues macro, execution error.

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

Motivation

Consumer group rebalances are generally intended to be abstracted away from the user as much as possible, but more advanced users of the Consumer client may actually want to explicitly trigger a rebalance at times. For example, Streams may do this if it has detected through some other means that one of its members has dropped out of the group and needs to rejoin right away. As of KIP-268 , it also relies on triggering a rebalance to provide a smoother upgrade path through its version probing protocol. And lastly, in the recently-approved KIP-441 we intend to improve the availability of stateful tasks by letting new instances warm up standbys. To probe for sufficiently caught-up standbys and transfer the active task assignment to them, Streams will once again leverage the ability to trigger a rebalance.

For this design to be as effective as possible, Streams makes three assumptions about an enforced rebalance:

1. it happens *immediately*
2. it happens *always*
3. it is *quick, cheap, & low overhead*

Currently the only way to enforce a rebalance in the absence of metadata/membership changes are by unsubscribing (and resubscribing) the consumer. Unfortunately this *can* violate assumptions #1 or #2, and *always* violates assumption #3.

If static membership is used, unsubscribing will not trigger a rebalance as the member may not actually leave the group. If after some timeout the member has not rejoined, a rebalance will be triggered to redistribute its partitions (violating assumption #1). If it does rejoin within the timeout, it will be handed back its old assignment without requiring the entire group to rebalance (violating assumption #2). This makes static membership incompatible with version probing and the upcoming KIP-441. All three are crucial improvements to the operation and usability of Streams, and users should not be forced to make a choice between them.

If a rebalance *is* reliably triggered by calling #unsubscribe, it will cause all the owned partitions to be revoked (or lost). Prior to the improvements in KIP-429 this was always the case for any rebalance and assumption #3 was never true, regardless of who/what caused the rebalance. But incremental cooperative rebalancing in 2.4 now allows members to hold on to their owned partitions during a rebalance, and Streams can continue processing active tasks during a rebalance starting with 2.5. KIP-441 was specifically designed with this in mind, and builds off of this by trading additional rebalances for high availability. This is a good tradeoff if the rebalances are lightweight, but not so good if they actually contribute to overall unavailability as would be the case if calling unsubscribe.

Rebalancing delays and unavailability has long been a major pain point to Streams applications. We hope to finally mitigate this with the combination of KIP-345, KIP-429, and KIP-441 – but without the ability to explicitly trigger a rebalance we won't be able to fully realize the high availability potential of Streams.

Public Interfaces

```
Consumer {
   /**
     * Alert the consumer to trigger a new rebalance by rejoining the group. This is a nonblocking call that
forces
     * the consumer to trigger a new rebalance on the next {@link #poll(Duration)} call. Note that this API
does not
     * itself initiate the rebalance, so you must still call {@link #poll(Duration)}. If a rebalance is already
in
     * progress this call will be a no-op. If you wish to force an additional rebalance you must complete the
current
     * one by calling poll before retrying this API.
     * <p>
     * You do not need to call this during normal processing, as the consumer group will manage itself
     * automatically and rebalance when necessary. However there may be situations where the application wishes
to
     * trigger a rebalance that would otherwise not occur. For example, if some condition external and
invisible to
     * the Consumer and its group changes in a way that would affect the userdata encoded in the
     * {@link org.apache.kafka.clients.consumer.ConsumerPartitionAssignor.Subscription Subscription}, the
Consumer
     * will not be notified and no rebalance will occur. This API can be used to force the group to rebalance
so that
     * the assignor can perform a partition reassignment based on the latest userdata. If your assignor does
not use
     * this userdata, or you do not use a custom
     * {@link org.apache.kafka.clients.consumer.ConsumerPartitionAssignor ConsumerPartitionAssignor}, you
should not
     * use this API.
     *
     * @throws java.lang.IllegalStateException if the consumer does not use group subscription
     */
    void enforceRebalance() {
}
```

Proposed Changes

This KIP proposes to add an API that will request a rebalance *without* revoking all currently owned partitions. Both static and dynamic members will attempt to rejoin the group. It is a nonblocking call that does not itself initiate the rebalance (as in *consumer.unsubscribe()*) but instead just marks the consumer as needing to rejoin. A rebalance will then be triggered on the next poll() call.

The behavior is fairly straightforward overall, but there are some edge cases to consider:

A rebalance is already in progress

There is always the possibility that a rebalance is already in progress and the consumer is just awaiting its new assignment, or waiting for the remaining members to rejoin. It's also possible that an application may try to trigger a rebalance from multiple consumers in the group at once. The appropriate handling of this case depends on the application-level logic and reason for forcing a rebalance to begin with: for example, the current plan for KIP-441 is for the leader to periodically trigger a rebalance to check for ready standbys and give them an active assignment. If the rebalance is triggered based on an arbitrary time interval we should just allow the current rebalance to complete and move on, since the ongoing rebalance will collect the latest metadata just as the enforced rebalance would have.

But what if the rebalance was triggered not at some arbitrary point in time, but based on some just now satisfied condition? One alternative we considered for KIP-441 was to have each member trigger a rebalance once they finished preparing their standbys. If a rebalance was already in progress the subscription metadata would have been sent before this condition was met, and the resulting assignment would not take into account the readiness of the standby. If we just allowed the current rebalance to complete and did not trigger another one, the ready standby would not get converted to an active task.

Clearly, the behavior of this API in the case on an ongoing rebalance is application-dependent. However it seems best to leave it up to the user to determine whether or not to retry based on the results of the completed rebalance by checking the assignment received.

The consumer is not part of an active group

If the consumer is not part of an active group, either because it dropped out or it has not yet joined the group, we should already be attempting to rejoin. But if we are not yet in the *REBALANCING* state (above case), then we have not yet sent out the metadata and so the next rebalance should include whatever update caused the app to want to force a rebalance. In this case we just return "true".

example usage:

Let's say your app has some heavy initialization to do before it is ready to start processing partitions from certain topics (call it topic A), while other topics (topic B) can be processed right away. You want to avoid assigning any of the topic A partitions to a new member until it's ready to work on them, so your assignor will need to include whether the member is initialized or not in the userdata. This way the assignor can make sure a member that has just joined will only receive partitions from topic B, allowing other members of the group to continue making progress on the partitions of topic A until the newer member is ready for them. In the example processing loop below, each member will check some system condition to determine whether it is ready to receive partitions from topic A and trigger a rebalance if so. By using this new API, in combination with a cooperative assignor, the app can actually continue to poll and process records while the rebalance goes on in the background.

MyApp.java

```
mainProcessingLoop() {
        if (justCompletedInitialization) {
                needsRebalance = true;
                justCompletedInitialization = false;
        }

        if (needsRebalance) {
                consumer.enforceRebalance()
        }

        records = consumer.poll();

        // check the assignment in case you need to retry, eg if a rebalance was already in progress was
enforceRebalance was called
        if (receivedFinalAssignment()) {
                needsRebalance = false;
        }
        ...
        // do something
}
```

Compatibility, Deprecation, and Migration Plan

We will be adding a new method to the Consumer interface, and as such any new implementations should override this method. No default implementation will be provided so existing Consumer implementations that wish to upgrade should be extended and recompiled.

Rejected Alternatives

Instead of adding this directly to the consumer client, we could have added this to the admin client as in

```
Admin {
        forceRebalance(String groupId);
}
```

On the one hand, triggering a rebalance seems like an administrative action and thus more appropriate for the admin client than the consumer. However this API is not intended to be an operational one, but instead a way for consumers to proactively communicate with the group and efficiently channel updates relevant to their assignment. I'm also not sure whether this would even be possible to implement without broker-side changes.

Another idea that was debated was whether to make this a blocking call, where the consumer will actually initiate the rebalance (ie send out a JoinGroup) and then potentially wait up to some provided timeout for it to complete. On the surface this seems to offer the simpler approach: the rebalance will be triggered by this single API, and users can choose to wait on the rebalance to complete or fall back to a nonblocking call by setting the timeout to zero. However, the same thing can be achieved by just calling poll afterwards (with whatever timeout). This makes the enforceRebalance API much cleaner and easier to reason about, as all the edge case (eg coordinator unavailable/unknown) and error handling (for example if the rebalance callback throws an error) can remain part of poll, and the user need only worry about whether they want to retry in the case a rebalance was already ongoing. This also keeps the implementation clean, by keeping all actual rebalancing within the scope of poll and just setting a flag to rejoin.

Furthermore, some use cases (such as the current design plan of KIP-441) will involve calling this new API at some regular, periodic basis. For those cases a more straightforward solution might be to simply add a "rebalance.interval" config that allows users to specify some interval at which to automatically trigger periodic rebalances. However this does not fit all use cases: some, such as version probing or a more advanced, heuristic-driven KIP-441 design, require triggering of a rebalance in response to specific system changes. The chosen design is intended to allow users to trigger a reassignment based on conditions external/unknown to the consumer (and not for example as a way to resolve temporary imbalances due to membership changes).