Incremental Cooperative Rebalancing: Support and Policies

References:

- Kafka Client-side Assignment Proposal
- A Guide To The Kafka Protocol#GroupMembershipAPI
- KafkaConsumer Javadoc
- KIP-26 - Add Kafka Connect framework for data import/export
- KIP-62: Allow consumer to send heartbeats from a background thread
- KIP-134: Delay initial consumer group rebalance

Introduction

Rebalancing between distributed application processes in Apache Kafka was enhanced considerably when it was decoupled as logic from Kafka brokers and was moved as responsibility to the clients and specifically to Kafka Consumer. This pattern has been working robustly for quite a while now and has been used successfully, besides Kafka Consumer itself, in Kafka Streams, Kafka Connect, and other applications outside the Apache Kafka code repository. It is based on a simple principle: when a distributed process leaves or joins the group, all the members of this group stop processing and coordinate in order to redistribute the resources they share, such as their assignments to Kafka topic partitions. At the start of rebalancing, the processes will have any assigned resources to them revoked. At the end of rebalancing they will receive a new assignment of resources. At any time, one process in the group is the designated leader. The group leader is responsible for computing the assignment of resources within the group during a rebalance.

In this proposal, we introduce necessary improvements to the existing rebalancing procedure and propose extensions to the client side protocols - also known as embedded protocols - that will allow Kafka applications to perform rebalancing incrementally and in cooperation. This will allow Kafka applications to scale even further under circumstances where the existing rebalancing mechanism is reaching its limits or incurs unnecessary costs. Before we discuss the general approach and each design in detail, we list definitions of common terms that are used in this proposal and have been used in Kafka proposals in the past.

Definitions

Group: In this context the term *group* is used to describe a group of distributed processes that use Kafka to run in cooperation as a common group.

Resource: A type of resource to be shared and distributed among the members of a **group**. Common types of resources include Kafka topic-partitions that are assigned to Kafka applications, as well as connectors and tasks that are distributed among Kafka Connect workers. Resources can be application-defined and Kafka coordinator is agnostic of their definition.

Rebalance/Rebalancing: the procedure that is followed by a number of distributed processes that use Kafka clients and/or the Kafka coordinator to form a common **group** and distribute a set of **resources** among the members of the group.

Group Membership Protocol: A protocol that is used between Kafka applications and the Kafka coordinator to define groups of distributed processes within a specific Kafka application.

Request/Response Types used by the Group Membership Protocol: The procedure that defines membership within a **group** during **rebalancing** is implemented by using messages between processes of a **group** and the Kafka coordinator of the following types:

- *JoinGroupRequest*
- *JoinGroupResponse*
- *SyncGroupRequest*
- *SyncGroupRespose*
- *HeartbeatRequest*
- *HeartbeatRespose*

Embedded Protocol: A protocol that is used by Kafka applications and consists of the definition of types of **resources** and an algorithm that uses **rebalancing** as a process in order to distributed these resources among a **group** in a Kafka application.

Class Types used by the Embedded Protocol: So far, all embedded protocols have been using two classes to describe and distribute resources among the group:

- *Assignment*
- Subscription

Motivation

Currently, rebalancing is based on two protocols. First, a core, lower-level protocol that ensures group membership. Kafka coordinator and any Kafka component that forms groups (clients, Connect or others) are aware of the core protocol that is based on the definition of the rebalance request/response types mentioned above. Then, on top of the core protocol, a second protocol is defined that is responsible to describe resources which need to be distributed among the members of a group. This type of protocol is called embedded protocol and its logic is piggybagged within the core protocol. Because of this layering, the Kafka coordinator remains agnostic of an embedded protocol and only the members of the group, including the group leader, are using it. Since the resource aware protocols can be embedded to the core protocol without the latter being changed, it makes sense for several embedded protocols to exist on top of a single core group membership protocol. Every embedded protocol defines and distributes a different set of resources during rebalancing. However, up to now, across all embedded protocols the resources are exchanged (and therefore balanced) in an all-or-nothing fashion: in every rebalancing round, each member is releasing its resources, stopping at the same time any processing related to these resources and requests to re-join the group.

This situation, known also as stop-the-world effect, has proven inflexible and expensive at large scale and under specific circumstances. Specifically, there are two dimensions in which a more selective management of resources and the ability to cope with temporary imbalances could allow Kafka applications to scale even better:

1. A resource that does not have to be given up because, after rebalancing, its reassignment will happen to the same process. In this case, the overhead of releasing and re-acquiring the same resources can be avoided. Furthermore, the application does not have to stall completely. Resources that do not have to be redistributed can keep being utilized. Normally, only a few resources will need to change hands and because of that the application will notice only a partial, and therefore potentially shorter, interruption due to rebalancing.
2. Isolated support of heterogeneous resources. It is already the case that members of a group do not have to share resources that belong to the same application. For instance, Kafka Connect shares across a cluster of Connect workers multiple connectors and tasks of different types and different users. A worker leaving the group does not have to interrupt connectors that don't even run on this worker, or even stop connectors that have only a few tasks running on this worker. The connectors can keep running in the remaining workers until the full load is redistributed across the Connect cluster or another worker joins the group.

Based on these two general observations, here we will examine a set of specific use cases that can benefit from incremental cooperative rebalancing that is more flexible and can handle temporary imbalances without enforcing *stop-the-world* across the board. These use cases are:

- **Kubernetes process death**. A node hosting a specific Kafka client process dies, but the underlying orchestrator framework is able to replace this node quickly and restore the process. A good example is a Kafka Streams application, that has state that is persisted and can be restored quickly when the process comes back up to a selected node by the orchestrator. In this case, it is often better for the application to endure a short imbalance and temporarily decrease its running tasks, instead of performing two rebalances only to return to the state that the group was operating before the node failure. Here, Kubernetes is used as a convenient placeholder for this use case's name, but the situation is equivalent across different orchestrators and cluster managers.
- **Rolling bounce**. Similar to what might happen unexpectedly with a node failure, can occur with an intentional rolling upgrade of a cluster running Kafka applications. In this case, we have control over how and when a nodes comes back up from maintenance and same as in the case of failure, we would prefer to tolerate a temporary imbalance rather than perform an unnecessary and disruptive redistribution of resources several times.
- **Scale up/down**. In cases where a cluster of Kafka applications scales up or down, the applications that are not affected by this scaling should not be interrupted. Even more, it would be desirable to control the pace to which scaling takes effect.

General Approach and Common Features

The key idea behind all the designs proposed here is to change the assumption we've always made with groups and their resources until now: when you go into a JoinGroup, it is expected that you give up control of all resources that you control and get them into a clean state for handoff. Among various components that use rebalancing, here are a few examples of what resources mean:

- **Kafka Consumer**: Topic Partitions (clean up and handoff means offset commits)
- **Kafka Streams**: Topic Partitions (clean up and handoff means offset commits)
- **Kafka Connect**: Connectors and Tasks (clean up and handoff means flush and offset commits)
- For other known use cases: leadership (no clean up necessary)

We'd want to change the semantics of rebalancing in a way that allows processes to hold onto resources and allow things to be in an imbalanced state for a while. This leads to an enhanced embedded protocol as you need to:

- Be able to express what resources you're holding on to.
    - This is by definition part of the embedded protocol as the broker is not aware of what resources are being managed.
    - This could be done either by having each member include the resource list in their metadata, could default to holding onto everything, or something more customized to the system if it makes sense (since this is also implemented in the embedded protocol).
- Be able to pass information from the leader back to the members about how things are imbalanced or what resources they need to give up in the near future.
    - Again, fundamentally part of embedded protocol since the broker is not aware of what resources are being managed.
    - This can be added to the SyncGroupResponse as another metadata field.
- Be able to trigger another rebalance either immediately or at some time in the future based on feedback from the leader about imbalance/what resources they should give up.
    - Requires being able to trigger a rebalance in `AbstractCoordinator` either immediately or later.
    - For the consumer and Connect this is straightforward. In fact Connect already uses this process with multiple rounds of rebalancing when it detects that the leader is behind in reading the config topic (which would result in bad assignments).

With respect to the implementation aspects of this enhancement, two key characteristics are: a) that ideally we'd want to apply changes to the embedded protocol only, keeping the Kafka brokers and coordinator still agnostic of the changes, and b) that the common components should be shared and usable across Kafka applications and that it will be easy for each application (e.g. Consumer, Streams, Connect) to implement and provide its own policies for rebalancing in a modular way.

Benefits & Tradeoffs

- Good: Only modifies the embedded protocol. This has significant compatibility, and therefore adoption, implications, especially for shops where upgrading brokers is considered is a big deal compared to upgrading apps.
- Good: Decouples implementation across various Kafka components such as Consumer, Streams and Connect.
- Good: If there are different needs with respect to how imbalance is handled, we have easy flexibility in doing things differently across components. For example, Connect could start considering imbalance in Connector threads differently than imbalance in task threads.
- Bad: Need to implement N times, where N == # of `AbstractCoordinator` implementations
- Good: But, by aligning on requirements for the embedded rebalance protocol a good amount of code could be shared and reused.
- Bad: If Streams can't just leverage consumer changes, then they need to add an implementation of `AbstractCoordinator` to make use of this. Currently it only relies on the `PartitionAssignor`.
- Good: Nothing we do using this approach has to affect non-Java clients unless they want to interoperate in the same group as Java consumers.

Concrete Designs

Here we'll explore a few concrete designs for the proposed protocol of incremental cooperative rebalancing. We will also evaluate which problems each solves and how various consumer group events are handled (e.g. the case where a new leader is elected).

There's one uncommon event that we already know we want to consider for each design:

- Leader exits and new leader is elected in this rebalance

Further, we want to characterize the behavior of each in terms of the following scenarios defined above in the Motivation section:

- Kubernetes process death
- Rolling bounce
- Scale up/down

Below, the proposed designs will focus on the Kafka Consumer use case. For Connect, although the resources managed differ, the rebalance process is similar enough that what works for consumer groups would work for Connect as well. For Streams we propose to keep leveraging the consumer group implementation as we do today and apply any changes required to adapt to the new partition management model. Uses cases that use the rebalance mechanism just for leadership assignment are not affected by the proposed changes and will continue to work in the same way as today.

Design I: Simple Cooperative Rebalancing

This is the simplest version that provides incremental cooperative rebalancing in multiple rounds. The format of the `Subscription` and `Assignment` clas ses respectively are defined as follows:

Subscription (Member Leader):

```
Subscription => Version SubscribeTopics PartitionAssignorData AssignedTopicPartitions
  Version                 => Int16
  SubscribeTopics         => [String]
  PartitionAssignorData   => Bytes
  AssignedTopicPartitions => [Topic Partitions]
    Topic         => String
    Partitions    => [int32]
```

Assignment (Leader Member):

```
Assignment => Version AssignedTopicPartitions RevokeTopicPartitions
  Version                                     => int16
  AssignedTopicPartitions         => [Topic Partitions]
    Topic         => String
    Partitions    => [int32]
  RevokeTopicPartitions         => [Topic Partitions]
    Topic         => String
    Partitions    => [int32]
```

Member process:

- Include subscription topics as usual in `JoinGroup`. Also include your previous assignment (null or empty for new members)
- When you need to join group, by default hold on to all partitions and continue processing.
- Assignment response includes usual assignment information. Start processing any new partitions. (Since we expect sticky assignment, we could also optimize this and omit the assignment when it is just repeating a previous assignment)
- If assignment response includes anything in RevokePartitions, stop processing the revoked partitions (continuing processing of still assigned partitions), commit, and then initiate another join group immediately.

Leader process:

- Use partition assignor on subscription requests as usual for each round of join group. Compute diff from previous data (which we have from `Assi gnedTopicPartitions` in the subscription) and include necessary partitions in `RevokePartitions`. The returned `AssignedTopicPartitio ns` should be the computed assignment minus anything included in `RevokePartitions` since we need to wait for them to give up.

- The leader must also account for "lost" or unaccounted topic partitions when a member is no longer in the group. Such are topic partitions that are included in the metadata for the subscribed topics but are not present in `AssignedTopicPartitions`. This case should be handled just by ensuring all topic partitions in the subscriptions are assigned.

Events:

- *Member joins* - On the first join group round, the leader will compute partition assignments for the new member but none will be returned in `AssignedPartitions` while we wait for other members to do revocation. The new member will get its assignment after everyone else revokes and rejoins.

Example: Member joins

```
Initial group and assignment: A(T1,T4), B(T2), C(T3)
D(T) joins
Leader computes new assignment as: A(T1), B(T2), C(T3), D(T4)
but sends assignment with revocation request: A(assigned:,revoked:T4), B(assigned:,revoked:), C(assigned:,
revoked:), D(assigned:,revoked:)
Members join with subscriptions: A(T,assigned:T1), B(T,assigned:T2), C(T,assigned:T3), D(T,assigned:)
Leader computes new assignment as: A(assigned:,revoked:), B(assigned:,revoked:), C(assigned:,revoked:), D
(assigned:T4,revoked:)
```

- *Member leaves* - Rebalance will be triggered by leave group (or session timeout). When the leader computes new assignments, the partitions previously assigned to the former member will not be in any `AssignedTopicPartitions` of the Subscription messages sent by the currently participating members. Therefore, these partitions are considered revoked already and they can be immediately assigned in this iteration. No further rebalances will be required.

Example: Member leaves

```
Initial group and assignment: A(T1), B(T2), C(T3), D(T4)
D(T4) leaves
Rebalance is triggered. Remaining member rejoin with subscriptions:
A(T,assigned:T1), B(T,assigned:T2), C(T,assigned:T3)
Leader computes new assignment as: A(assigned:T1,T4,revoked:), B(assigned:T2,revoked:), C(assigned:T3,revoked:)
```

- *Member bounces* - This case is just a *member leaves* event, followed by a *member joins* event. There will be one rebalance when the member leaves and partitions are reassigned, then two rebalances: the first one when the member rejoins and revocations are returned and then the second one that will deliver the new assignment to all members.
- *Leader exits and new leader is elected in this rebalance* - Including AssignedTopicPartitions in the member metadata requires re-sending that information, but allows a new leader, even if it was new to the group, to properly maintain stickiness.

Characteristics:

- **Kubernetes process death** - IMPROVED if restoring the process takes longer than it takes for other members to rejoin the group, this version doesn't help for this situation. However, it is better than before as only the partitions on the node are affected.
- **Rolling bounce** - IMPROVED rebalances are still triggered immediately so rolling bounces would see multiple rebalances. However, the impact of this is still greatly reduced as only the bounced node's partitions jump around.
- **Scale up/down** - YES everyone is involved in the rebalance, but only affected topic partitions actually stop processing temporarily

Discussion:

In general this policy results in more rebalances. However, it has the benefit that in all cases only the partitions being moved are affected (assuming join group moved to background thread, see note in decisions below). Further, some of the rebalances are expected to be fast if we get all members to know to rejoin quickly. So having more rebalances isn't really an issue unless it has substantial resource overhead.

One implicit change here is that `ConsumerRebalanceListener` now may be invoked with partial sets, or we need to introduce another callback interface that handles that case. Possibly this change and a config to switch between protocol versions are the only public API changes that would be needed.

The main overhead here is the addition of `AssignedTopicPartitions`, which could be substantial in cases like MirrorMaker. However, this is no worse than the assignment data being sent from the leader to members.

Also, note that we need to take care in computing assignments. The `AssignedTopicPartitions` could be a subset of the total set of topic partitions; combining this with a new leader, it is important to generate assignments from the `SubscribeTopics`, only using `AssignedTopicPartitions` to a) keep things sticky and b) figure out what needs to be included in `RevokeTopicPartitions`.

As with all of these proposals, the assumption is that members are cooperative, so there is no explicit indication that topic partitions were actually given up.

Finally, one potential concern is that not all members have an indicator that a rebalance is going to be required since they base this on the presence of any `RevokeTopicPartitions`. One possible improvement would be to rely on an empty, non-null list (or an extra boolean indicator field) to notify them immediately. This could potentially speed up the second rebalance since they wouldn't have to wait to get the indicator via heartbeat.

Design II: Deferred Resolution of Imbalance

This variant builds on the previous one and adds control over when we should schedule another rebalance instead of always trying to resolve imbalance immediately. `Subscription` and `Assignment` classes are defined as follows:

Subscription (Member Leader):

```
Subscription => Version SubscribeTopics PartitionAssignorData AssignedTopicPartitions
  Version                => Int16
  SubscribeTopics        => [String]
  PartitionAssignorData  => Bytes
  AssignedTopicPartitions => [Topic Partitions]
    Topic         => String
    Partitions    => [int32]
```

Assignment (Leader Member):

```
Assignment => Version AssignedTopicPartitions RevokeTopicPartitions ScheduledRebalanceTimeout
  Version                                     => int16
  AssignedTopicPartitions                 => [Topic Partitions]
    Topic         => String
    Partitions    => [int32]
  RevokeTopicPartitions                => [Topic Partitions]
    Topic         => String
    Partitions    => [int32]
  ScheduledRebalanceTimeout        => int32
```

Note that the only difference in the format is `ScheduledRebalanceTimeout` and we might also choose to make this a fixed constant set via configuration instead of dynamic in the protocol.

Member process:

- *(Same as above)* Include subscription topics as usual in join group. Also include your previous assignment (null or empty for new members)
- *(Same as above)* When you need to join group, by default hold on to all partitions and continue processing
- *(Same as above)* Assignment response includes usual assignment information. Start processing any new partitions. (Since we expect sticky assignment, we could also optimize this and omit the assignment when it is just repeating a previous assignment)
- *(Same as above)* If assignment response includes anything in RevokePartitions, stop processing, commit, and then initiate another join group immediately.
- If `ScheduledRebalanceTimeout` > 0, plan to rejoin as soon as possible after that timeout. (This should only be set if the `RevokePartitions` is empty.)

Leader process:

- *(Same as above)* Use partition assignor on subscription requests as usual for each round of join group. Compute diff from previous data (which we have from `AssignedTopicPartitions` in the subscription) and include necessary partitions in `RevokePartitions`. The returned `AssignedTopicPartitions` should be the computed assignment minus anything included in `RevokePartitions` since we need to wait for them to give up.
- "Lost" or unaccounted topic partitions that belong to topics subscribed by at least one member but are not listed in any `AssignedTopicPartitions` from members indicate either:
    - a) new subscriptions
        - This case can be detected based on the subscriptions and assignments. If a topic is completely missing from previous assignments, assume it is this case.
        - These should be resolved immediately, so just continue with immediate assignment.
    - b) a member left the group
        - If we didn't detect a new subscription, assume it is due to this case.
        - However, if we detect a new member, we want to utilize it immediately to resolve the imbalance. This could happen if the process manages to get restarted within the time it takes to do the rebalance. This process rejoins the group but its list of previously `AssignedTopicPartitions` is empty due to the restart. For a stable leader, they can just compare the set of member IDs to the last generation. For a new leader we can use a heuristic such as looking for a member that has no previous assignments (assuming there's enough members that they *should* have had an assignment).
        - If not reassigning immediately, set `ScheduledRebalanceTimeout` appropriately to defer the actual movement in the hopes the member will reappear. In this case there should be no `RevokeTopicPartitions`.

Events:

- *Member joins* -
    - First member: The behavior here is customizable. Either the leader can immediately assign topic partitions or use the deferment process to allow other members to join in that period.
    - Additional members: Reassignment happens immediately via two step revoke + assign rebalances. Deferring only applies to "lost" partitions.
- *Member leaves* - The policy detects a member left and uses `ScheduledRebalanceTimeout` to get the group to wait some time before resolving any imbalance.

Example: Member leaves

```
Initial group and assignment: A(T1), B(T2), C(T3), D(T4)
D(T4) leaves
Rebalance is triggered. Remaining member rejoin with subscriptions:
A(T,assigned:T1), B(T,assigned:T2), C(T,assigned:T3)
Leader computes detects "lost" partition T4. Sends empty assignments, without revocations and a scheduled
rebalance timeout of t1:
A(assigned:,revoked:,t1), B(assigned:,revoked:,t1), C(assigned:,revoked:,t1)
After t1, remaining members join again:
A(T,assigned:T1), B(T,assigned:T2), C(T,assigned:T3)
Leader sends updated assignment:
A(assigned:T1,T4,revoked:,-), B(assigned:,revoked:,-), C(assigned:,revoked:,-)
```

- *Member bounces* - This case is same as a *member leaves* event, followed by a *member joins* event. The new member is detected as a probably returning member and gets the same assignment back (assuming subscriptions haven't changed and the partition assignor is sticky)

Example: Member bounces

```
Initial group and assignment: A(T1), B(T2), C(T3), D(T4)
D(T4) bounces. First leaves the group.
Rebalance is triggered. Remaining member rejoin with subscriptions:
A(T,assigned:T1), B(T,assigned:T2), C(T,assigned:T3)
Leader computes detects "lost" partition T4. Sends empty assignments, without revocations and a scheduled
rebalance timeout of t1:
A(assigned:,revoked:,t1), B(assigned:,revoked:,t1), C(assigned:,revoked:,t1)
Before t1, member D joins again as D'
Rebalance is triggered. All members join with subscriptions:
A(T,assigned:T1), B(T,assigned:T2), C(T,assigned:T3), D'(T,assigned:)
Leader sends updated assignment:
A(assigned:,revoked:,-), B(assigned:,revoked:,-), C(assigned:,revoked:,-), D'(assigned:T4,revoked:,-)
```

- *Leader exits and new leader is elected in this rebalance* - This case is treated as loss of member. Enough info is available to at least heuristically detect whether the leader bounced back and can immediately be reassigned or if we put the missing partitions into purgatory during the timeout.
  - What if the previous leader was in the middle of waiting for a scheduled timeout?
    - If the new leader was already in the group, they can just use the timeout they should already know about and not override it.
    - If the new leader is new to the group, they should fall back to assuming there wasn't a wait period in effect.

Characteristics:

- **Kubernetes process death** - YES One of the key cases this extension is designed for. This policy will wait for a process to have a chance to reappear before reassigning topic partitions. If the process appears earlier than the `ScheduledRebalanceTimeout` timeout, we just assign immediately and get back to work, so you are limited by how fast kubernetes detects the failure and brings back your process.
- **Rolling bounce** - YES Effectively the same as above. We still have multiple rebalances for each process bounce, but they don't have the same impact. As in the case above, we only need to wait as long as it takes for the process to come back up.
- **Scale up/down** - YES Scale up is detected as a new member and reassignment happens immediately. Scale down takes the `ScheduledRebalanceTimeout` to resolve since we need to be confident the member has really left the group.

Discussion:

As noted, in the cases where a process comes back, the outage is basically only limited to the time it takes the process to come back (plus the heartbeat interval since other members need to find out about the need to rejoin). This means users in environments like that could reasonably set the scheduled rebalance timeout relatively high as long as they don't expect to regularly have scale down events.

Having the flexibility to control delaying assignment in the protocol covers the need for delayed rebalancing when a new consumer group is formed, which was the subject of KIP-134 (https://cwiki.apache.org/confluence/display/KAFKA/KIP-134%3A+Delay+initial+consumer+group+rebalance). Therefore, implementation of this deferred resolution of imbalance policy could allow us to provide a more holistic solution and deprecate the specific configuration (group.initial.rebalance.delay.ms), that is addressing consecutive rebalances only at the initial stage of a consumer group.

The loss of state when the leader changes to a member new to the group can result in a liveness situation under which the time before actually performing an assignment keeps getting extended indefinitely. This can only happen if the leader changes to a new member of the group, within the scheduled rebalance timeout indefinitely. Arguably, this is unlikely enough that it is safe to ignore (bouncing members so fast probably means you're doing something wrong and you shouldn't expect stable groups anyway).

Design III: Incremental Resolution of Imbalance

This variant is just an extension of the behavior of the deferred resolution. The idea is simple so a full description is skipped here. The only change is that the leader could choose to only reassign *some* partitions and handle the rest in the same iterative fashion incrementally.

In this case you probably want the `ScheduledRebalanceTimeout` to be explicit in the message because you might use one interval for detecting members leaving and another for how long to wait between moving partitions.

This would be less for the detection of nodes that go missing/reappear since in that case you really just want to assign everything that was "lost" to the new node, or evenly distribute. The case where this is useful would be something like Connect where you might scale down # of tasks for a connector such that you would be in an imbalanced state when just those tasks were deleted, or you're scaling up your cluster and add a new node (the key characteristic being that the cluster is imbalanced, but everything is still assigned and humming along). If you move everything at once, all affected tasks have to stop + commit before anyone can proceed. If you have an heavy connector with a lot of buffered data and a light connector with little, the light connector tasks get stuck doing nothing while waiting for the heavy connector tasks to finish. By moving each individually, we decouple them and minimize the outage for each task.

Characteristics:

- **Kubernetes process death** - YES
- **Rolling bounce** - YES
- **Scale up/down** - YES Here, scale up has the added benefit that you get a smoother, incremental move back to balance and no coupling of flush/commit behavior across partitions.

Discussion:

This policy aims to be helpful in very large deployments of groups with diverse resources, such as large Connect clusters with a diverse set of connectors and tasks.

Compatibility

As discussed above, the proposed changes here focus on the embedded protocol. In this case, a round of rolling bounces will be sufficient to upgrade all the members of a group to the latest version of the embedded protocol. Another round might be useful if it's desired to disable the old version completely.

Conclusions

Here are some conclusions that derive from the above:

- We will probably need to make it possible for the consumer to join group in the background thread. This is necessary because even if we don't give up partitions, the join group process itself blocks processing since the consumer works in single threaded mode. This means that if node A doesn't have any revoked partitions but node B does and takes 1ms to flush + commit + join and node C also has some revoked but takes minutes, when node A joins it will still be blocked from processing until node C finishes up.
- Out of the available options regarding any potential enhancement to the rebalancing mechanisms and the respective protocols, an incremental approach where we keep the changes primarily to the embedded protocol is a good first step. We can evaluate the need for other changes that might involve changes in the Kafka coordinator's protocol after implementing the proposed enhancements here. This will allow for the implementation of various policies in the different Kafka components to move at their own pace.

Acknowledgements

This document was composed based on original contributions by Ewen Cheslack-Postava , Jason Gustafson , guozhang Wang and Konstantine Karantasis