KIP-415: Incremental Cooperative Rebalancing in Kafka Connect

Status

Current state: *Adopted.*

Discussion thread: here

Vote thread: here

JIRA: KAFKA-5505

Released: *AK 2.3.0*

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

Motivation

Since its initial release, Kafka Connect has used ingeniously the group membership API in Kafka to distribute connectors and tasks among Workers that consist a Connect cluster. The simplicity of this concept has allowed Connect Workers to form robust and highly available clusters. However, contrast to the Kafka Consumer that uses the group membership API to disseminate resources (topic-partitions) in a homogeneous way among the members of the same Kafka application, Kafka Connect distributes heterogeneous resources instead - connectors and tasks - among its Workers. This difference comes with a side-effect: Every connector that submits a new or updated configuration interrupts the execution of existing connectors, even though such connectors are unrelated to the connector scheduled for execution. From the perspective of a user that does not own a whole Connect cluster this is unintuitive. Furthermore, at large scale, this side-effect might lead to long startup times, following a complete rebalance of connectors and tasks in the Connect cluster.

This stop-the-world effect is known for a while in the context of Kafka client applications. In a recent design document (Incremental Cooperative Rebalancing) a set of use cases has been summarized that could benefit from alleviating the stop-the-world effect. 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. 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. In this use case, 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 node comes back up from maintenance and same as in the case of a 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.

Offering a first improvement to Connect's behavior under the above scenarios is the primary motivation of this KIP.

Proposed Changes

Out of the policies that are listed in Incremental Cooperative Rebalancing, this KIP is proposing an implementation that applies deferred resolution of imbalance (Design II in Incremental Cooperative Rebalancing). This policy appears to strike a good balance for a first implementation of Incremental Cooperative Connect protocol, since it aims to help all the aforementioned use cases and keeps imbalance resolution simple across consecutive rebalance rounds. It's worth mentioning here that, compared to other recent approaches that aim to alleviate the symptoms of an expensive rebalancing process basically by avoiding rebalancing altogether (for example see KIP-345), this KIP attempts to make rebalancing significantly lighter, even if that means that, in some cases, the number of actual rebalances increases. Connect provides a good platform for a first implementation of Incremental Cooperative Rebalancing and its reassignment heuristics, because connectors and tasks traditionally do not depend on local state that has to be restored upon restart. The changes necessary to implement an incremental and cooperative approach to Connect's rebalance protocol are described in the rest of the section.

Changes to Connect's Rebalancing Process

In Connect, processing of rebalancing procedures happens independently and asynchronously of the Worker threads that run tasks and connectors. In order to drive group membership and coordinate rebalancing, the threads of the `DistributedHerder` are used instead. This decoupling of responsibilities facilitates the main proposition of this KIP, which is to change when revocation of resources (connectors and tasks) is happening in a Connect cluster. Following the general approach described in Incremental Cooperative Rebalancing, Connect Workers should now be allowed to keep running connectors and tasks until these resources are explicitly revoked by the Leader of the group. Briefly this leads to the following changes:

- When a Worker is prepared to join the group, it does not revoke (stop) any connectors or tasks that it currently runs (previously a Worker would stop all connectors and tasks in preparation of joining its group).
- When a Worker sends its metadata as part of the join request, it includes its current assignment (previously the metadata included in a join request did not include any assignment information)
- When a Worker is elected as Leader, it computes a new assignment, describing both assigned and revoked connectors and tasks (previously the Leader computed an assignment from scratch without defining revoked resources).
- When a Worker receives its assignment, if this assignment includes any revoked connectors or tasks, it stops (releases) these resources and then immediately rejoins the group with an assignment that excludes revoked resources (previously, upon receipt of assignment, the Worker started the connectors and tasks and operated in the new generation of the group protocol until the next rebalance some time in the future).
- Normally in the next assignment round, the Leader will assign resources according to its policy and there will be no revoked resources in any of the Workers. If that's not the case, the previous steps will be repeated until the group converges into an assignment without revocations.
- When a Worker receives assignments without any revocations, it starts the assigned connector and tasks, and defers rejoining the group for time equal to the amount of time included in the `ScheduledRebalanceDelay` field of the assignment included in the sync response.

Overall, as far as the rebalancing workflow is concerned, implementation of Incremental Cooperative Rebalancing does not change the definition of events that might trigger a rebalance. Rebalance can still be triggered by configuration updates, a Worker joining the group, or a Worker leaving the group - intentionally or due to a failure.

Fencing based on Rebalancing

Until now the process of rebalancing has also implied a global synchronization barrier among the members of the group. With respect to the shared resources among the group, this barrier imposed a happens-before relationship between the release and the acquisition of a resource. In Connect this simply means that a task had to stop execution on a specific worker and save its progress, before starting execution on another worker after a rebalance. This is a necessary property for resources and is still guaranteed under this proposal. However, instead of the rebalance being a flat global synchronization point for all the connectors and tasks, the happens-before relationships are enforced only on tasks that are revoked and subsequently reassigned, whenever this is required. This optimization, that allows resolution of the stop-the-world effect (as it's also known in the current form of rebalancing), is allowed because the barrier imposed by rebalancing is implicit, it concerns the Connect workers and is not exposed in Connect's API. Connect tasks run independently and when a task stops and resumes progress it is not required that all tasks of a connector do the same. Of course, incremental cooperative rebalancing can be applied not only on individual tasks but also in a group of tasks, such as the tasks of a specific connector. However, in the changes proposed in this KIP, each task remains independent and is scheduled (assigned/revoked/reassigned) independently of other tasks.

Embedded Protocol Serialization

The new Connect protocol that will introduce incremental cooperative rebalancing will extend the existing protocol by appending its new fields as optional fields on the existing format. This is a backwards compatible and easy to integrate way to have both protocols handled and live in the same code base for as much as it is necessary. Introducing a different serialization format was also considered initially (read below under Rejected Alternatives a short summary). To read how the existing protocol is extended to support incremental cooperative rebalancing read below the section on Public Interfaces.

Deferred Resolution of Imbalance under Different Rebalancing Scenarios

The behavior of the suggested Incremental Cooperative Connect protocol is described below in different scenarios with examples. Regarding the encoding of the examples, first letter of the recourse is a Connector instance (e.g. Connector A, Connector B, etc). Second letter is type: C for Connector, T for task. Number is regular task numbering. 0 for Connectors, greater or equal to 1 for Tasks. W represents a Worker, with W1 and W2, etc being different Workers joining the same group. Primes are used to represent a Worker that was member of the group and rejoins soon after a short period of being offline.

In every case, the heuristics that will decide assignment computation are based on the fact that Leader election is sticky between successive rebalances. The Leader of a group will remain mostly the same, until it gets removed from the group metadata stored on Kafka.

Worker joins the group

This scenario has two subcategories: a Worker is the first member to join or the Worker is joining a group with an existing number of members.

Given that joining the group of Worker in the Connect cluster is a relatively quick operation, compared to reading the internal config topic or starting a large number of connectors and tasks from scratch, in this KIP the suggestion is to start rebalancing immediately. Thus:

First new member joins

```
Initial group and assignment: -
Config topic contains: AC0, AT1, AT2, BC0, BT1
W1 joins with assignment: []
Rebalance is triggered
W1 becomes leader
W1 computes and sends assignments:
W1(delay: 0, assigned: [AC0, AT1, AT2, BC0, BT1], revoked: [])
```

Non-first new member joins

```
Initial group and assignment: W1([AC0, AT1, AT2, BC0, BT1])
Config topic contains: AC0, AT1, AT2, BC0, BT1
W1 is current leader
W2 joins with assignment: []
Rebalance is triggered
W3 joins while rebalance is still active with assignment: []
W1 joins with assignment: [AC0, AT1, AT2, BC0, BT1]
W1 becomes leader
W1 computes and sends assignments:
W1(delay: 0, assigned: [AC0, AT1], revoked: [AT2, BC0, BT1])
W2(delay: 0, assigned: [], revoked: [])
W3(delay: 0, assigned: [], revoked: [])
W1 stops revoked resources
W1 rejoins with assignment: [AC0, AT1]
Rebalance is triggered
W2 joins with assignment: []
W3 joins with assignment: []
W1 becomes leader
W1 computes and sends assignments:
W1(delay: 0, assigned: [AC0, AT1], revoked: [])
W2(delay: 0, assigned: [AT2, BC0], revoked: [])
W3(delay: 0, assigned: [BT1], revoked: [])
```

Worker leaves the group

In this scenario, the Worker that leaves the groups, leaves permanently or without returning within the predefined delay

Worker leaves

```
Initial group and assignment: W1([AC0, AT1]), W2([AT2, BC0]), W3([BT1])
Config topic contains: AC0, AT1, AT2, BC0, BT1
W1 is current leader
W2 leaves
Rebalance is triggered
W1 joins with assignment: [AC0, AT1]
W3 joins with assignment: [BT1]
W1 becomes leader
W1 computes and sends assignments:
W1(delay: d, assigned: [AC0, AT1], revoked: [])
W3(delay: d, assigned: [BT1], revoked: [])
After delay d:
W1 joins with assignment: [AC0, AT1]
W3 joins with assignment: [BT1]
Rebalance is triggered
W1 becomes leader
W1 computes and sends assignments:
W1(delay: 0, assigned: [AC0, AT1, BC0], revoked: [])
W3(delay: 0, assigned: [BT1, AT2], revoked: [])
```

Worker bounces

In this scenario, the Worker that leaves the groups, leaves temporarily and returns within the predefined delay

Worker bounces

```
Initial group and assignment: W1([AC0, AT1]), W2([AT2, BC0]), W3([BT1])
Config topic contains: AC0, AT1, AT2, BC0, BT1
W1 is current leader
W2 leaves
Rebalance is triggered
W1 joins with assignment: [AC0, AT1]
W3 joins with assignment: [BT1]
W1 becomes leader
W1 computes and sends assignments:
W1(delay: d, assigned: [AC0, AT1], revoked: [])
W3(delay: d, assigned: [BT1], revoked: [])
Before delay d expires:
W2 joins with assignment: []
Rebalance is triggered
W1 becomes leader
W1 computes and sends assignments:
W1(delay: d', assigned: [AC0, AT1], revoked: [])
W2(delay: d', assigned: [], revoked: [])
W3(delay: d', assigned: [BT1], revoked: [])
d' is the remaining delay
After delay d':
W1 joins with assignment: [AC0, AT1]
W2 joins with assignment: []
W3 joins with assignment: [BT1]
Rebalance is triggered
W1 becomes leader
W1 computes and sends assignments:
W1(delay: 0, assigned: [AC0, AT1], revoked: [])
W2(delay: 0, assigned: [AT2, BC0], revoked: [])
W3(delay: 0, assigned: [BT1], revoked: [])
```

Leader leaves the group

In terms of the process being followed when a member leaves the group, this scenario is the same as a regular Worker leaving the group. However, because this proposal uses the fact that the leader assignment is sticky, meaning that, once a Worker is elected as a Leader it will keep being a Leader in subsequent rebalances until it fails, a Leader leaving the group affects the heuristics that determine whether the unaccounted resources are new or they are resources that were lost due to a Worker's failure. In this KIP the suggestion is for the new Leader to:

- Compute and assign resources without a delay, if the departure of the previous Leader was the event that triggered the rebalance and there was no active delay in progress. In this case, the new Leader has no way of deciding whether unaccounted tasks are new or lost.
- Compute and assign resources, abiding also by a delay, if such a delay is in progress at the time that the previous Leader was detected as absent. In this case, unaccounted resources are not immediately assigned to Workers and the delay is honored. If before the delay expires, a Worker joins the group, the heuristic is to assign it the unaccounted resources, favoring the case that this could be the previous Leader that bounced back to the group as a Worker.

Leader leaves

```
Initial group and assignment: W1([AC0, AT1]), W2([AT2, BC0]), W3([BT1])
Config topic contains: AC0, AT1, AT2, BC0, BT1
W1 is current leader
W1, which is the leader, leaves
Rebalance is triggered
W2 joins with assignment: [AT2, BC0]
W3 joins with assignment: [BT1]
W3 becomes leader.
No delay is in progress.
W3 computes and sends assignments:
W2(delay: 0, assigned: [AT2, BC0, AC0], revoked: [])
W3(delay: 0, assigned: [BT1, AT1], revoked: [])
```

Leader bounces

```
Initial group and assignment: W1([AC0, AT1]), W2([AT2, BC0]), W3([BT1])
Config topic contains: AC0, AT1, AT2, BC0, BT1
W1 is current leader
W2 leaves
Rebalance is triggered
W1 joins with assignment: [AC0, AT1]
W3 joins with assignment: [BT1]
W1 becomes leader
W1 computes and sends assignments:
W1(delay: d, assigned: [AC0, AT1], revoked: [])
W3(delay: d, assigned: [BT1], revoked: [])
Before delay d expires:
W2 joins with assignment: []
Rebalance is triggered
W1 becomes leader
W1 computes and sends assignments:
W1(delay: d', assigned: [AC0, AT1], revoked: [])
W2(delay: d', assigned: [], revoked: [])
W3(delay: d', assigned: [BT1], revoked: [])
d' is the remaining delay
W1, which is the leader, leaves
Rebalance is triggered
W2 joins with assignment: []
W3 joins with assignment: [BT1]
W3 becomes leader.
There's an active delay in progress.
W3 computes and sends assignments:
W2(delay: d'', assigned: [], revoked: [])
W3(delay: d'', assigned: [BT1], revoked: [])
d'' is the remaining delay
Before delay d'' expires:
W1 joins with assignment: []
Rebalance is triggered
W3 becomes leader
W3 computes and sends assignments:
W1(delay: d''', assigned: [], revoked: [])
W2(delay: d''', assigned: [], revoked: [])
W3(delay: d''', assigned: [BT1], revoked: [])
After delay d''':
W1 joins with assignment: []
W2 joins with assignment: []
W3 joins with assignment: [BT1]
Rebalance is triggered
W3 becomes leader
W3 computes and sends assignments:
W1(delay: 0, assigned: [BC0, AT2], revoked: [])
W2(delay: 0, assigned: [AC0, AT1], revoked: [])
W3(delay: 0, assigned: [BT1], revoked: [])
```

The last scenario, highlights that if a Leader bounces, and there's a delay in progress, there's no guarantee that the assignments will match the initial assignments before they delay was triggered. If a Leader leaves first, rebalancing is triggered with no delay set. If the Leader leaves with an active delay, the delay is honored but this also means that Leader was not the first member to leave the group. Thus, the heuristic of the assignment is not guaranteed to repeat the initial assignment.

Public Interfaces

To implement Incremental Cooperative rebalancing with deferred resolution of imbalance, the following changes in Connect's embedded protocol format are suggested:

Connect protocol's format

Subscription (Member Leader):

```
 Subscription => Version ConfigOffset Allocation
   Version                 => Int16
   Url                           => [String]
   ConfigOffset            => Int64
   Allocation              => [Byte] (optional field, default null)



Allocation contains a serialized Assignment type, representing the current assignment of the worker.
```

Assignment (Leader Member):

```
 Assignment => Version Error Leader LeaderUrl Assignment Revoked ScheduledDelay
   Version              => Int16
   Error                => Int16
   Leader               => [String]
   LeaderUrl            => [String]
   Assignment               => [Byte]
     Connector              => [String]
     Tasks                  => [int32]
   Revoked                                 => [Byte] (optional field, default null)
     Connector              => [String]
     Tasks                  => [Int32]

   ScheduledDelay               => Int32 (optional field, default 0)
```

Configuration Properties

Along with the changes in Connect's protocol format, the addition of the following configuration properties is proposed:

- `scheduled.rebalance.max.delay.ms`
  Type: Int32
  Default: 300000 (5min)
  This is a delay that the leader may set to tolerate departures of workers from the group by allowing a transient imbalance connector and task assignments. During this delay a worker has the opportunity to return to the group and get reassigned the same or similar amount of work as before. This property corresponds to the maximum delay that the leader may set in a single assignment. The actual delay used by the leader to hold off redistribution of connectors and tasks and maintain imbalance may be less or equal to this value.

- `connect.protocol`
  Type: Enum
  Values: `eager`, `compatible`
  Default: `compatible`
  This property defines which Connect protocol is enabled.
  - `eager` corresponds to the initial non-cooperative protocol that resolves imbalance with an immediate redistribution of connectors and tasks (version 0).
  - `compatible` corresponds to both eager (protocol version 0) and incremental cooperative (protocol version 1 or higher) protocols being enabled with the incremental cooperative protocol being preferred if both are supported (version 1 or version 0).

Compatibility, Deprecation, and Migration Plan

The new configuration property `connect.protocol` controls which version of the Connect protocol is enabled. Setting this property to:

```
connect.protocol = eager
```

runs the Connect protocol version 0 unchanged, without the improvements introduced by Incremental Cooperative Rebalancing.

Connect protocol version 0 will be marked deprecated in the next major release of Apache Kafka (currently 3.0.0). After adding a deprecation notice on this release, support of version 0 of the Connect protocol will be removed on the subsequent major release of Apache Kafka (currently 4.0.0).

Migration of Connect Workers to the new version of the Connect protocol is supported without down time. In order to perform live migration a rolling bounce process is preferred as follows:

- Bounce each Worker one-by-one after setting:

```
connect.protocol = compatible
```

To downgrade your cluster to use protocol version 0 from version 1 or higher with `eager` rebalancing policy what is required is to switch one of the workers back to `eager` mode.

```
connect.protocol = eager
```

Once this worker joins, the group will downgrade to protocol version 0 and `eager` rebalancing policy, with immediately release of resources upon joining the group. This process will require a one-time double rebalancing, with the leader detecting the downgrade and first sending a downgraded assignment with empty assigned connectors and tasks and from then on just regular downgraded assignments.

Test Plan

- Parameterize existing unit tests to test all Connect protocols and compatibility modes.
- Add additional unit tests covering the new Connect protocol for Incremental Cooperative Rebalancing.
- Write the first integration tests for Connect protocols using the integration test framework for Kafka Connect: https://issues.apache.org/jira/browse/KAFKA-7503.
- Write system tests that exercise different scenarios.

Rejected Alternatives

Keep the current implementation and deploy smaller and more focused Connect clusters

Sizing a Connect cluster as well as selecting the number and type of connectors that run on it should be an architectural decision driven by business logic and not by system constraints as much as possible. Connect is a lightweight framework with a mission to be scalable and relieve operators from the burden of running individual services for every integration of Kafka with third-party systems. Connect's scalability has been demonstrated on numerous production deployments and this KIP aims to alleviate a pain point in Connect's scalability that might occur more frequently under the situations described in the motivation section. Therefore, seems preferable to address this gap rather than depend on ad-hoc workarounds and an exclusive dependence on external tooling.

Use a more versatile serialization format for the new and subsequent Connect protocols.

Initially, this KIP included a suggestion to express the new protocol format for Connect on the serialization format defined by flatbuffers. This suggestion was made to set up the protocol for more flexible upgrades in the future. However, after discussion on the Kafka developer's mailing list, it became apparent that introducing such a dependency with this KIP did not seem absolutely necessary at this point. Since this suggestion was orthogonal to the - already significant - changes brought up by this proposal, consideration of a different serialization format for the Connect protocol was postponed for future versions, and only if that's deemed necessary then. Admittedly, the changes in the protocol fields are not very frequent, with version 1 aiming to succeed version 0 after more than 3 years since its initial release.

Reduce number of rebalances and differentiate between initial and successive rebalances

The current proposal aims to offer a scalable and easy to configure and use solution to Connect's Worker rebalancing pain points under a wide range of use cases. Without ruling out the existence of use cases that would appreciate a more fine grain control over specific phases of the Worker rebalance process (e.g. avoid a redundant rebalance in some cases or use a different delay for initial vs successive group joins), this KIP errs on the side of retaining Connect's simplicity in terms of implementation and configuration demands, while addressing the majority of the use cases under which the current rebalance protocol is showing its limitations.