

# KIP-429: Kafka Consumer Incremental Rebalance Protocol


- [Status](#)
- [Motivation](#)
- [Background](#)
  - [Consumer Rebalance Protocol: Stop-The-World Effect](#)
  - [Example: Streams Assignor Rebalance Metadata](#)
  - [Example: Consumer Sticky Assignor](#)
- [Proposed Changes: Incremental Consumer Rebalance Protocol](#)
  - [Consumer Protocol](#)
  - [Consumer Coordinator Algorithm](#)
  - [Rebalance Callback Error Handling](#)
  - [Consumer Metrics](#)
  - [CooperativeStickyAssignor and custom COOPERATIVE Assignors](#)
  - [ConsumerRebalanceListener and ConsumerPartitionAssignor Semantics](#)
  - [Compatibility and Upgrade Path](#)
    - [Consumer](#)
    - [Streams](#)
  - [Allow Consumer to Return Records in Rebalance](#)
  - [Looking into the Future: Heartbeat Communicated Protocol](#)
  - [Looking into the Future: Assignor Version](#)
  - [Edge Cases Discussion](#)
    - [Non-active partition assignor](#)
    - [Downgrading and Old-Versioned New Member](#)
- [Public Interface](#)
- [Compatibility, Deprecation, and Migration Plan](#)
  - [Minimum Version Requirement](#)
  - [Recommended Upgrade Procedure](#)
- [Rejected Alternatives](#)

## Status

**Current state:** *Accepted (2.4.0)*

**Discussion thread:** [link](#)

**JIRA:**

key	summary	type	created	updated	due	assignee	reporter	priority	status	resolution
 JQL and issue key arguments for this macro require at least one Jira application link to be configured										

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

## Motivation

Recently Kafka community is promoting [cooperative rebalancing](#) to mitigate the pain points in the stop-the-world rebalancing protocol and an initiation for Kafka Connect already started as [KIP-415](#).

This KIP is trying to customize the incremental rebalancing approach for Kafka consumer client, which will be beneficial for heavy-stateful consumers such as Kafka Streams applications.

In short, the goals of this KIP are:

- Reduce unnecessary downtime due to unnecessary partition migration: i.e. partitions being revoked and re-assigned.
- Better rebalance behavior for falling out members.

## Background

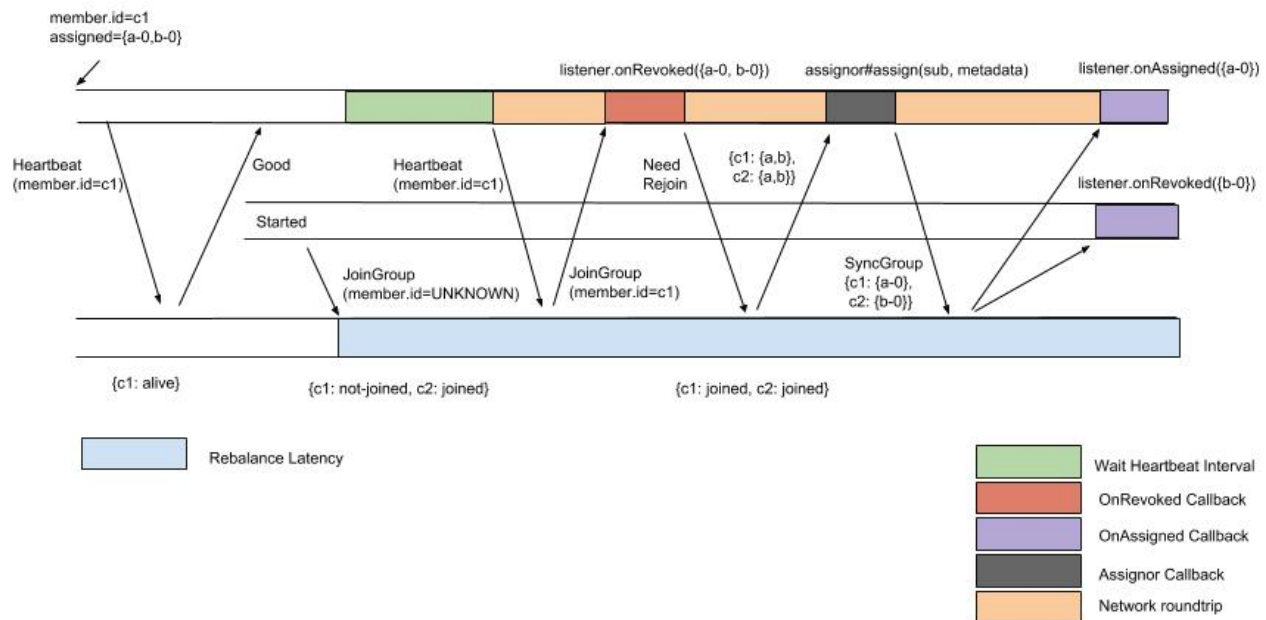
## Consumer Rebalance Protocol: Stop-The-World Effect

As mentioned in motivation, we also want to mitigate the stop-the-world effect of current global rebalance protocol. A quick recap of current rebalance semantics on KStream: when rebalance starts, all stream threads would

1. Join group with all currently assigned tasks revoked.
2. Wait until group assignment finish to get assigned tasks and resume working.
3. Replay the assigned tasks state.
4. Once all replay jobs finish, stream thread transits to running mode.

The reason for revoking all ongoing tasks is because we need to guarantee each topic partition is assigned with exactly one consumer at any time. In this way, any topic partition **could not be re-assigned before it is revoked**.

Below shows the process of the rebalance protocol with a new member joining the group.



## Example: Streams Assignor Rebalance Metadata

Today Streams embed a full fledged `Consumer` client, which hard-code a `ConsumerCoordinator` inside. Streams then injects a `StreamsPartitionAssignor` to its pluggable `PartitionAssignor` interface and inside the `StreamsPartitionAssignor` we also have a `TaskAssignor` interface whose default implementation is `StickyPartitionAssignor`. Streams partition assignor logic today sites in the latter two classes. Hence the hierarchy today is:

```
KafkaConsumer -> ConsumerCoordinator -> StreamsPartitionAssignor -> StickyTaskAssignor.
```

`StreamsPartitionAssignor` uses the subscription / assignment metadata byte array field to encode additional information for sticky partitions. More specifically on subscription:

```
KafkaConsumer:
```

```
Subscription => TopicList SubscriptionInfo
  TopicList      => List<String>
  UserData       => Bytes
```

```
-----
```

```
StreamsPartitionAssignor:
```

```
UserData (encoded in version 4) => VersionId LatestSupportVersionId ClientUUID PrevTasks StandbyTasks EndPoint
  VersionId      => Int32
  LatestSupportVersionId => Int32
  ClientUUID     => 128bit
  PrevTasks      => Set<TaskId>
  StandbyTasks   => Set<TaskId>
  EndPoint       => HostInfo
```

And on assignment:

```
KafkaConsumer:
```

```
Assignment = AssignedPartitions AssignmentInfo
  AssignedPartitions      => List<String, List<Int32>>
  UserData               => Bytes
```

```
-----
```

```
StreamsPartitionAssignor:
```

```
UserData (encoded in version 4) => VersionId, LatestSupportedVersionId, ActiveTasks, StandbyTasks,
PartitionsByHost, ErrorCode
  VersionId      => Int32
  LatestSupportVersionId => Int32
  ActiveTasks     => List<TaskId>
  StandbyTasks    => Map<TaskId, Set<TopicPartition>>
  PartitionsByHost => Map<HostInfo, Set<TopicPartition>>
  ErrorCode       => Int32
```

## Example: Consumer Sticky Assignor

We also have a StickyAssignor provided out of the box trying to mitigate the cost of unnecessary partition migrations. This assignor only relies on subscription metadata but not modifying assignment metadata, as follows:

```
KafkaConsumer:
```

```
Subscription => TopicList SubscriptionInfo
  TopicList      => List<String>
  UserData       => Bytes
```

```
-----
```

```
StickyAssignor:
```

```
UserData (encoded in version 1) => AssignedPartitions
  AssignedPartitions      => List<String, List<Int32>>
```

The goal of this incremental protocol, is to fully leverage on the sticky assignors which will try to reassign partitions to its previous owners in best effort, such that we will revoke less partitions as possible since the revocation process is costly.

## Proposed Changes: Incremental Consumer Rebalance Protocol

We will augment the consumer's rebalance protocol as proposed in [Incremental Cooperative Rebalancing: Support and Policies](#) with some tweaks compared to [KIP-415](#). The key idea is that, instead of relying on the single rebalance's synchronization barrier to rebalance the group and hence enforce everyone to give up all the assigned partitions before joining the group as the new generation, we use consecutive rebalances where the end of the first rebalance will actually be used as the synchronization barrier.

### Consumer Protocol

More specifically, we would first inject more metadata at the consumer-layer, as:

```
KafkaConsumer:

Subscription => TopicList UserData AssignedPartitions
  TopicList          => List<String>
  UserData           => Bytes
  AssignedPartitions => List<String, List<Int32>> // new field

Assignment = AssignedPartitions UserData
  AssignedPartitions => List<String, List<Int32>>
  UserData           => Bytes
```

Note that it is compatible to inject additional fields after the assignor-specific SubscriptionInfo / AssignmentInfo bytes, since on serialization we would first call assignor to encode the info bytes, and then re-allocate larger buffer to append consumer-specific bytes; with the new protocol, we just need to append some fields before, and some fields (a.k.a. those new fields) after the assignor-specific info bytes, and vice-versa on deserialization. So adding fields after the assignor-bytes is still naturally compatible with the plug-in assignor. However there are indeed some compatibility challenges for the consumer protocol upgrade itself, which we will tackle [below](#).

In addition, we want to resolve a long-lasting issue that when consumer's being kicked out of the group, since it no longer owns the partitions the `commit` call would doom to fail. To distinguish this case with the normal case that consumers are likely still within the group but just try to re-join, we introduce a new API into the consumer rebalance listener:

```
public interface ConsumerRebalanceListener {

    void onPartitionsRevoked(Collection<TopicPartition> partitions);


    void onPartitionsAssigned(Collection<TopicPartition> partitions);

    // new API
    default void onPartitionsLost(Collection<TopicPartition> partitions) {
        onPartitionsRevoked(partitions);
    }
}
```

For users implementing this rebalance listener, they would not need to make code changes necessarily if they do not need to instantiate different logic; but they'd still need to **recompile their implementation class**. The semantics of these callbacks do differ in the new cooperative protocol however, so you should review your implementation to make sure there are no logical changes needed. For details, see **ConsumerRebalanceListener and ConsumerPartitionAssignor Semantics** below.

Note that adding new fields would increase the size of the request, especially in cases like Streams where user metadata has been heavily encoded with

assignor-specific metadata. We are working on

 Unable to render Jira issues macro, execution error.

with compression / reformation

to reduce the user metadata footprint, and with that we believe adding this new field would not be pushing the size exceeding the message size limit.

## Consumer Coordinator Algorithm

Rebalance behavior of the consumer (captured in the consumer coordinator layer) would be changed as follows.

1. For every consumer: before sending the join-group request, change the behavior as follows based on the join-group triggering event:
  - a. If subscription has changed: revoke all partitions who are not of subscription interest by calling **onPartitionsRevoked**, send join-group request with whatever left in the owned partitions in Subscription.
  - b. If topic metadata has changed: call **onPartitionsLost** on those owned-but-no-longer-exist partitions; and if the consumer is the leader, send join-group request.
  - c. If received REBALANCE\_IN\_PROGRESS from heartbeat response / commit response: re-join group with all the currently owned partitions as assigned partitions.
  - d. If received UNKNOWN\_MEMBER\_ID or ILLEGAL\_GENERATION from join-group / sync-group / commit / heartbeat response: reset generation / clear member-id correspondingly, call rebalance listener's **onPartitionsLost** for all the partition and then re-join group with empty assigned partition.
  - e. If received MEMBER\_ID\_REQUIRED from join-group request: set the member id, and then re-send join-group (at this moment the owned partitions should be empty).
2. For the leader: after getting the received subscription topics, as well as the assigned-partitions, do the following:
  - a. Collect the partitions that are claimed as currently owned from the subscriptions; let's call it **owned-partitions**.
  - b. Call the registered assignor of the selected protocol, passing in the cluster metadata and get the returned assignment; let's call the returned assignment **assigned-partitions**. Note the this set could be different from owned-partitions.
  - c. Compare the owned-partitions with assigned-partitions and generate three exclusive sub-sets:
    - i. Intersection(owned-partitions, assigned-partitions). These are partitions that are still owned by some members, and some of them may be now allocated for new members. Let's call it **maybe-revoking-partitions**.
    - ii. Minus(assigned-partitions, owned-partitions). These are partitions that are not previously owned by any one. This set is non-empty when its previous owner is on older version and hence revoked them already before joining, or a partition is revoked in previous rebalance by the new versioned member and hence not in any assigned partitions, or it is a newly created partition due to add-partitions. Let's call it **ready-to-migrate-partitions**.
    - iii. Minus(owned-partitions, assigned-partitions). These are partitions that does not exist in assigned partitions, but are claimed to be owned by the members. It is non-empty if some topics are deleted, or if the leader's metadata is stale (and hence the generated assignment does not have those topics), or if the previous leader has created some topics in its assignor that are not in the cluster yet (consider the Streams case). Let's call it **unknown-but-owned-partitions**.
  - d. For maybe-revoking-partitions, check if the owner has changed. If yes, exclude them from the assigned-partitions list to the new owner. The old owner will realize it does not own it any more, revoke it and then trigger another rebalance for these partitions to finally be reassigned.
  - e. For ready-to-migrate-partitions, it is safe to move them to the new member immediately since we know no one owns it before, and hence we can encode the owner from the newly-assigned-partitions directly.
  - f. For unknown-but-owned-partitions, it is also safe to just give them back to whoever claimed to be their owners by encoding them directly as well. If this is due to topic metadata update, then a later rebalance will be triggered anyways.
3. For every consumer: after received the sync-group response, do the following:
  - a. Calculate the **newly-added-partitions** as  $\text{Minus}(\text{assigned-partitions}, \text{owned-partitions})$  and the **revoked-partitions** as  $\text{Minus}(\text{owned-partitions}, \text{assigned-partitions})$ .
  - b. Update the assigned-partitions list.
  - c. If the set of revoked-partitions is non-empty, call the rebalance listener's **onPartitionsRevoked** and rejoin to trigger another rebalance.
  - d. For those newly-added-partitions, call the rebalance listener's **onPartitionsAssigned** (even if empty).

Below is a more illustrative of the different set of partitions and their assignment logic:



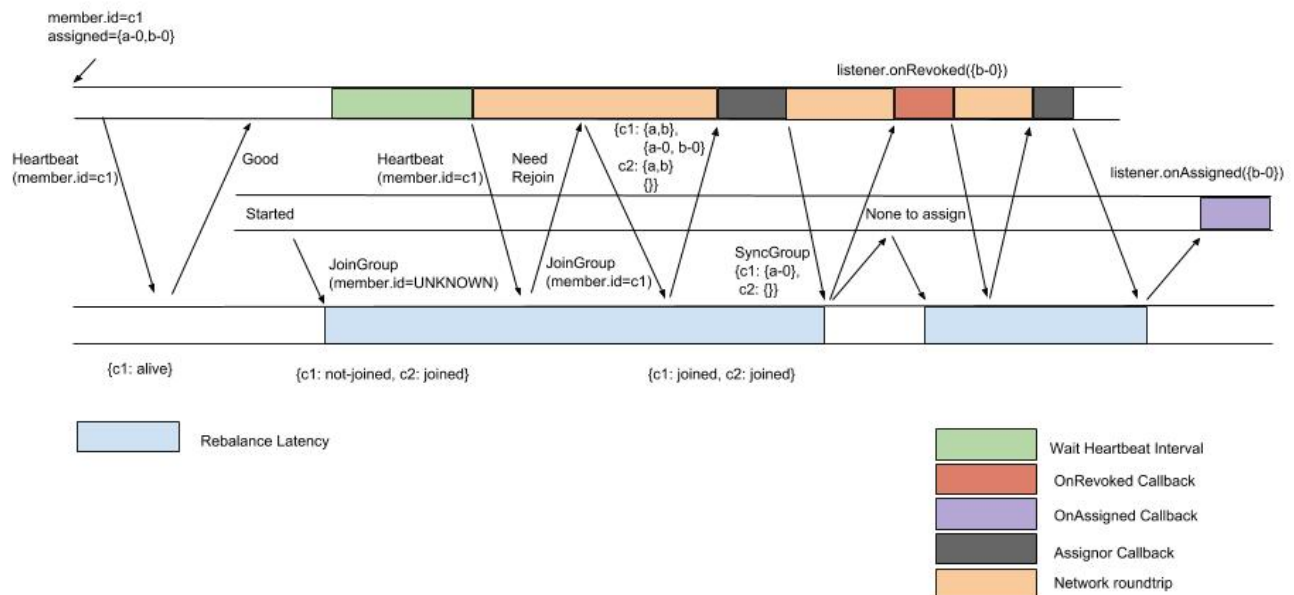
No changes required from the broker side, since this logic change is completely wrapped inside the consumer protocol / coordinator implementation itself, and to brokers it is just the same as previous version's rebalances.

Note that one minor difference compared with [KIP-415](#) is that we do not introduce the `scheduledDelay` in the protocol, but instead the consumer will trigger rebalance immediately. This is because the consumer protocol would applies to all consumers (including streams) and hence should be kept simple, and also because [KIP-345](#) is being developed in parallel which is aimed for tackling the scaling out / rolling bounce scenarios already.

We would omit the common scenarios description here since it is already covered in KIP-415, which is very similar to this KIP with the difference of the `scheduledDelay` above.

**NOTE** that for this new algorithm to be effective in reducing rebalance costs, it is really expecting the plug-in assignor to be "sticky" in some way, such that the diff of the newly-assigned-partitions and the existing-assigned-partitions can be small, and hence only a few subset of the total number of partitions need to be revoked / migrated at each rebalance in practice – otherwise, we are just paying more rebalance for little benefits. We will talk about how sticky `StreamsPartitionAssignor` would be updated accordingly in Part II.

Now the new process for the same new-member-joining example becomes the following under the new protocol (note that on the existing member we will now only call `revoke` on b-0 once, and no longer revoking both a-0 and b-0 and then later assigning a-0 again:



## Rebalance Callback Error Handling

Today, when a user customized rebalance listener callback throws an exception, as long as it is not `WakeupException` / `InterruptedException` it will be swallowed and logged with an error. This has been complained by our users that it was not an efficient way for notifying them when it happened (

A screenshot of a Jira error message. It features a yellow warning triangle icon on the left. To the right of the icon, the text reads "Unable to render Jira issues macro, execution error." in a blue, monospaced font. The entire message is enclosed in a thin orange rectangular border.

In this KIP, we propose to slightly modify the semantics of the listener, such that the callback is used more like a "notification" to users about what gets assigned and what gets revoked, etc, and even exceptions gets thrown, it would not cause any of these assignment / revocation results to be changed. Also, if the `ConsumerCoordinator` found out that there's no need to notify since no "new" partitions have been assigned or revoked, the corresponding callback would not be triggered either.

More specifically, we will change this behavior as well:

1. `ConsumerCoordinator` will check if the newly assigned / revoked / lost partitions set is empty or not; and if not, we will not trigger the corresponding listener.
2. When the listener callback throws an exception, `ConsumerCoordinator` will log an error, and then re-throws this exception all the way up to `KafkaConsumer.poll()`.
  - a. Note that with the new rebalance protocol, `onPartitionsAssigned` / `onPartitionsRevoked` may be called sequentially. If one throws an exception, we would still proceed to complete the rest of the callbacks while remembering the "first-thrown exception", and then at the end throws the remembered exception to the user.

Upon capturing the error, users can do the following depend on their exception handling logic:

- Shutdown the consumer gracefully if the exception is considered a fatal error.
- Retry `consumer.poll()` if they believe the exception is re-tryable, be aware that **it will be treated as if no side-effects are taken at all from the exception-thrown callback**.

To give a concrete example, suppose your current assigned partitions are {1,2}, and the newly assigned partitions are {2,3}. The consumer will call `onPartitionsRevoked(1)` and then `onPartitionsAssigned(3)`. Suppose the former failed with an exception, `ConsumerCoordinator` would still proceed to complete the latter callback (and assume the latter callback succeeds). If users decide to retry, it is still considered as having successfully changed to {2, 3} – i.e. we consider all of the effects user indicated in the callback have taken place.

## Consumer Metrics

As part of this KIP we will also add some metrics on the consumer side related to rebalancing. These include:

1. listener callback latency
  - a. `partitions-revoked-latency-avg`
  - b. `partitions-revoked-latency-max`
  - c. `partitions-assigned-latency-avg`
  - d. `partitions-assigned-latency-max`
  - e. `partitions-lost-latency-avg`
  - f. `partitions-lost-latency-max`
2. rebalance rate and latency (# rebalances per day, and latency including the callback time as well)
  - a. `rebalance-rate-per-hour`
  - b. `rebalance-total`
  - c. `rebalance-latency-avg`
  - d. `rebalance-latency-max`
  - e. `rebalance-latency-total`
  - f. `failed-rebalance-rate-per-hour`
  - g. `failed-rebalance-total`
3. `last-rebalance-seconds-ago` (dynamic gauge)

## CooperativeStickyAssignor and custom COOPERATIVE Assignors

Since we've already encoded the assigned partitions at the consumer protocol layer, for consumer's sticky partitioner we are effectively duplicating this data at both consumer protocol and assignor's user data. Similarly we have a `StreamsPartitionAssignor` which is sticky as well but relying on its own user data to do it. We have added a new out-of-the-box assignor for users that leverages the `Subscription`'s built-in `ownedPartitions`. Consumer groups plugging in the new "cooperative-sticky" assignor will follow the incremental cooperative rebalancing protocol. A specific upgrade path is required for users wishing to do a rolling upgrade to the new cooperative assignor, as described in the compatibility section below.



Users may also wish to implement their own custom assignor, or are already doing so, and want to use the new cooperative protocol. Any assignor that returns COOPERATIVE among the list in `#supportedProtocols` indicates to the `ConsumerCoordinator` that it should use the cooperative protocol, and must follow specific assignment logic. First, the assignor should try and be as "sticky" as possible, meaning it should assign partitions back to their previous owner as much as possible. The assignor can leverage the new `ownedPartitions` field that the `Subscription` has been augmented with in order to determine the previous assignment. Note that "stickiness" is important for the cooperative protocol to be effective, as in the limit that the new assignment is totally different than the previous one then the cooperative protocol just reduces to the old eager protocol as each member will have to completely revoke all partitions and get a whole new assignment. In addition, any time a partition has to be revoked it will trigger a follow up rebalance, so the assignor should seek to minimize partition movement. Second, in order to ensure safe resource management and clear ownership, the assignor must make sure a partition is revoked by its previous owner before it can be assigned to a new one. Practically speaking, this means that the assignor should generate its "intended" assignment and then check against the previous assignment to see if any partitions are being revoked (that is, in the `ownedPartitions` but not in the new assignment for a given consumer). If that is the case, that partition should be removed from the new assignment for that round, and wait until it has been revoked so that it can be assigned to its final owner in the second rebalance. See the `CooperativeStickyAssignor` implementation for an example.

Note that the `CooperativeStickyAssignor` is for use by plain consumer clients – the existing `StreamsPartitionAssignor` has simply been modified to support cooperative so users should not try to plug in the `CooperativeStickyAssignor` (or any other). The upgrade path for `Streams` differs slightly from that of the clients `CooperativeStickyAssignor` as well. See the section on `Streams` below for details.

## ConsumerRebalanceListener and ConsumerPartitionAssignor Semantics

If you do choose to plug in a cooperative assignor and have also implemented a custom `ConsumerRebalanceListener`, you should be aware of how the semantics and ordering of these callbacks has changed. In the eager protocol, the timeline of a rebalance is always exactly as follows:

1. **`Listener#onPartitionsLost`**: if the member has missed a rebalance and fallen out of the group, this new callback will be invoked on the set of all owned partitions (unless empty). The member will then rejoin the group.
2. **`Listener#onPartitionsRevoked`**: called on the full set of assigned partitions
3. **`Assignor#subscriptionUserData`**: called when sending the `JoinGroup` request
4. **`Assignor#assign`**: called only for group leader
5. **`Assignor#onAssignment`**: invoked after receiving the new assignment
6. **`Listener#onPartitionsAssigned`**: called on the full set of assigned partitions (may have overlap with the partitions passed to `#onPartitionsRevoked`)

In the cooperative protocol, the timeline is less exact as some methods may or may not be called, at different times, and on different sets of partitions. This will instead look something like the following:

1. **`Listener#onPartitionsLost`**: if the member has missed a rebalance and fallen out of the group, this new callback will be invoked on the set of all owned partitions (unless empty). The member will then rejoin the group.
2. **`Listener#onPartitionsRevoked`**: if the topic metadata has changed such that some owned partitions are no longer in our subscription or don't exist, this callback will be invoked on that subset. If there are no partitions to revoke for those reasons, this callback will not be invoked at this point (note that this will likely be the case in a typical rebalance due to membership changes, eg scaling in/out, member crashes/restarts, etc)
3. **`Assignor#subscriptionUserData`**: called when sending the `JoinGroup` request
4. **`Assignor#assign`**: called only for group leader. Note that the `#assign` method will now have access to the `ownedPartitions` for each group member (minus any partitions lost/revoked in step 0. or 1.)
5. **`Listener#onPartitionsRevoked`**: this will be called on the subset of previously owned partitions that are intended to be reassigned to another consumer. *If this subset is empty, this will not be invoked at all.* If this *is* invoked, it means that a followup rebalance will be triggered so that the revoked partitions can be given to their final intended owner.
6. **`Assignor#onAssignment`**: invoked after receiving the new assignment (will always be *after* any `#onPartitionsRevoked` calls, and *before* `#onPartitionsAssigned`).
7. **`Listener#onPartitionsAssigned`**: called on the subset of assigned partitions that were not previously owned before this rebalance. There should be no overlap with the revoked partitions (if any). This will *always* be called, even if there are no new partitions being assigned to a given member.

The *italics* indicate a callback that may not be called at all during a rebalance. Take note in particular that it is possible for `#onPartitionsRevoked` to never be invoked at all during a rebalance, and should not be relied on to signal that a rebalance has started. The `#onPartitionsAssigned` callback will however always be called, and can therefore be used to signal to your app that a rebalance has just completed.

## Compatibility and Upgrade Path

Since we are modifying the [consumer protocol](#) as above, we need to design the upgrade path to enable consumers upgrade to the new rebalance protocol in an online manner. In fact, the most tricky thing for this KIP is actually how to support safe online upgrade path, such that even if users made mistakes and not strictly following the instructions, we can pause the consumer from proceeding and hence eventually users will realized it by seeing e.g. consumer lags and investigating logs, rather than letting them to fall into an undefined behavior or even worse, having some partitions to be owned by more than one member.

Note that since we are injecting additional fields at the end of the consumer protocol, the new protocol would still be compatible with the old version. That means, an old-versioned consumer would still be able to deserialize a newer-versioned protocol data (as long as we only append new fields at the end, this would be the case).

However, when consumers with V1 is joining the group, there's a key behavioral difference that they would NOT revoke their partitions, and hence it is not safe to re-assign any of their partitions as we did in the current (V0) assignment logic. That means, **the leader can only proceed the assignment when it knew that all the members are either on V0, or V1 versions.**

Another thing to keep in mind that, if the leader itself is still on older version, it would still be able to deserialize the V1 subscription protocol as V0, by ignoring the additional fields, and hence it may "think" everyone is still on V0, while some of them may actually be on the newer version.



The key idea to resolve this, is to let Assignor implementors themselves to indicate the ConsumerCoordinator whether they are compatible with the protocol, and then relying on multi-assignor protocols for a safe upgrade path: users need to keep two assignors when switching the rebalance protocol, and after that they can use another rolling bounce to remove the old versioned protocol.

More specifically, we will introduce the new public API ConsumerPartitionAssignor class and its #Subscription / #Assignment (the old classes are actually in 'internal' package mistakenly, so we use this KIP to deprecate that class with this new one, along with augmented methods) as follows:

```
ConsumerGroupMetadata {
    public String groupId();

    public int generationId();

    public String memberId();

    public Optional<String> groupInstanceId();
}

public interface ConsumerPartitionAssignor {

    /**
     * Return serialized data that will be included in the {@link Subscription} sent to the leader
     * and can be leveraged in {@link #assign(Cluster, GroupSubscription)} ((e.g. local host/rack information)
     *
     * @param topics Topics subscribed to through {@link org.apache.kafka.clients.consumer.
KafkaConsumer#subscribe(java.util.Collection)}
     * and variants
     * @return nullable subscription user data
     */
    default ByteBuffer subscriptionUserData(Set<String> topics) {
        return null;
    }

    /**
     * Perform the group assignment given the member subscriptions and current cluster metadata.
     * @param metadata Current topic/broker metadata known by consumer
     * @param groupSubscription Subscriptions from all members including metadata provided through {@link
#subscriptionUserData(Set)}
     * @return A map from the members to their respective assignments. This should have one entry
     *         for each member in the input subscription map.
     */
    GroupAssignment assign(Cluster metadata, GroupSubscription groupSubscription);

    /**
     * Callback which is invoked when a group member receives its assignment from the leader.
     * @param assignment The local member's assignment as provided by the leader in {@link #assign(Cluster,
GroupSubscription)}
     * @param metadata Additional metadata on the consumer (optional)
     */
    default void onAssignment(Assignment assignment, ConsumerGroupMetadata metadata) {
    }

    /**
     * Indicate which rebalance protocol this assignor works with;
     * By default it should always work with {@link RebalanceProtocol#EAGER}.
     */
    default List<RebalanceProtocol> supportedProtocols() {
        return Collections.singletonList(RebalanceProtocol.EAGER);
    }

    /**
     * Return the version of the assignor which indicates how the user metadata encodings
     * and the assignment algorithm gets evolved.
     */
    default short version() {
        return (short) 0;
    }

    /**
     * Unique name for this assignor (e.g. "range" or "roundrobin" or "sticky"). Note, this is not required

```

```

    * to be the same as the class name specified in {@link ConsumerConfig#PARTITION_ASSIGNMENT_STRATEGY_CONFIG}
    * @return non-null unique name
    */
    String name();

    final class Subscription {
        public List<String> topics();

        public ByteBuffer userData();

        public List<TopicPartition> ownedPartitions();

        public void setGroupInstanceId(Optional<String> groupInstanceId);

        public Optional<String> groupInstanceId();
    }

    final class Assignment {
        public List<TopicPartition> partitions();

        public ByteBuffer userData();
    }

    final class GroupSubscription {
        public GroupSubscription(Map<String, Subscription> subscriptions);

        public Map<String, Subscription> groupSubscription();
    }

    final class GroupAssignment {
        public GroupAssignment(Map<String, Assignment> assignments);

        public Map<String, Assignment> groupAssignment();
    }

    /**
     * The rebalance protocol defines partition assignment and revocation semantics. The purpose is to
     establish a
     * consistent set of rules that all consumers in a group follow in order to transfer ownership of a
     partition.
     * {@link ConsumerPartitionAssignor} implementors can claim supporting one or more rebalance protocols via
     the
     * {@link ConsumerPartitionAssignor#supportedProtocols()}, and it is their responsibility to respect the
     rules
     * of those protocols in their {@link ConsumerPartitionAssignor#assign(Cluster, GroupSubscription)}
     implementations.
     * Failures to follow the rules of the supported protocols would lead to runtime error or undefined
     behavior.
     *
     * The {@link RebalanceProtocol#EAGER} rebalance protocol requires a consumer to always revoke all its owned
     * partitions before participating in a rebalance event. It therefore allows a complete reshuffling of the
     assignment.
     *
     * {@link RebalanceProtocol#COOPERATIVE} rebalance protocol allows a consumer to retain its currently owned
     * partitions before participating in a rebalance event. The assignor should not reassign any owned
     partitions
     * immediately, but instead may indicate consumers the need for partition revocation so that the revoked
     * partitions can be reassigned to other consumers in the next rebalance event. This is designed for sticky
     assignment
     * logic which attempts to minimize partition reassignment with cooperative adjustments.
     */
    enum RebalanceProtocol {
        EAGER((byte) 0), COOPERATIVE((byte) 1);

        private final byte id;

        RebalanceProtocol(byte id) {
            this.id = id;
        }

        public byte id() {

```

```

        return id;
    }

    public static RebalanceProtocol forId(byte id) {
        switch (id) {
            case 0:
                return EAGER;
            case 1:
                return COOPERATIVE;
            default:
                throw new IllegalArgumentException("Unknown rebalance protocol id: " + id);
        }
    }
}

```

Note the semantical difference between the above added fields:

1. The **assignor version** indicate that for the same assignor series, when its encoded metadata and algorithm changed. It is assumed the newer versioned assignor is compatible with older versions, i.e. it is able to deserialize the metadata and adjust its assignment logic to cope with other older versioned members. It will be used in the JoinGroup request so that broker-side coordinator can select the one with highest version to be the leader (details see below). As for the upcoming release, it is not necessary to be used but in the future it can be useful if brokers have also been upgraded to support the augmented JoinGroup request.
2. The **assignor preferred protocol** indicate the rebalance protocol it would work with. Note that the same assignor cannot change this preferred protocol value across in higher versions. ConsumerCoordinate will get this information and with that value it will decide which rebalance logic (e.g. the old one, or the newly proposed process in this KIP) to be used.
3. The **subscription / assignment version** will be aligned with the assignor version, and it will not be exposed via public APIs to users since they are only used for Consumer Coordinator internally. Upon deserialization / serialization, the version of the subscription / assignment will be de / encoded first and the follow-up serde logic can then be selected correspondingly.

With the existing built-in Assignor implementations, they will be updated accordingly:

	Highest Version	Supported Strategy	Notes
RangeAssignor	0	Eager	Current default value.
RoundRobinAssignor	0	Eager	
StickyAssignor	0	Eager	
CooperativeStickyAssignor	0	Eager, Cooperative	To be default value in 3.0
StreamsAssignor	4	Eager, Cooperative	

The reason we make "range" and "round-robin" to not support cooperative rebalance is that, this protocol implicitly relies on the assignor to be somewhat sticky to make benefits by trading an extra rebalance. However, for these two assignors, they would not be sticky (although sometimes range may luckily reassign partitions back to old owners, it is not best-effort) and hence we've decided to not make them be selected for cooperative protocol. The existing StickyAssignor was not made to support Cooperative to ensure users follow the smooth upgrade path outlined below, and avoid running into trouble if they already use the StickyAssignor and blindly upgrade.

The ConsumerCoordinator layer, on the other hand, will select which protocol to use based on the first assignor specified in its configs, as the following:

- Only consider protocols that are supported by all the assignors. If there is no common protocols supported across all the assignors, throw an exception during starting up time.
- If there are multiple protocols that are commonly supported, select the one with the highest id (i.e. the id number indicates how advanced is the protocol, and we would always want to select the most advanced one).

The specific upgrade path is described below. Note that this will be different depending on whether you have a plain consumer app or a Streams app, so make sure to follow the appropriate one.

## Consumer

From the user's perspective, the upgrade path of leveraging new protocols is similar to switching to a new assignor. For example, assuming the current version of Kafka consumer is 2.2 and "range" assignor is specified in the config (or no assignor is configured, which is identical as the RangeAssignor is the default below 3.0). The upgrade path would be:

1. The first rolling bounce is to replace the byte code (i.e. swap the jars) and introduce the cooperative assignor: set the assignors to "cooperative-sticky, range" (or round-robin/sticky/etc if you are using a different assignor already). At this stage, the new versioned byte code sends both

assignors in their join-group request, but will still choose EAGER as the protocol since it's still configured with the "range" assignor, and the selected rebalancing protocol must be supported by *all* assignors. in the list. The "range" assignor will be selected to assign partitions while everyone is following the EAGER protocol. This rolling bounce is safe.

2. The second rolling bounce is to remove the "range" (or round-robin/sticky/etc) assignor, i.e. only leave the "cooperative-sticky" assignor in the config. At this stage, whoever has been bounced will then choose the COOPERATIVE protocol and not revoke partitions while others not-yet-bounced will still go with EAGER and revoke everything. However the "cooperative-sticky" assignor will be chosen since at least one member who's already bounced will not have "range" any more. The "cooperative-sticky" assignor works even when there are some members in EAGER and some members in COOPERATIVE: it is fine as long as the leader can recognize them and make assignment choice accordingly, and for EAGER members, they've revoked everything and hence did not have any pre-assigned-partitions anymore in the subscription information, hence it is safe just to move those partitions to other members immediately based on the assignor's output.

The key point behind this two rolling bounce is that, we want to avoid the situation where leader is on old byte-code and only recognize "eager", but due to compatibility would still be able to deserialize the new protocol data from newer versioned members, and hence just go ahead and do the assignment while new versioned members did not revoke their partitions before joining the group. Note the difference with KIP-415 here: since on consumer we do not have the luxury to leverage on list of built-in assignors since it is user-customizable and hence would be black box to the consumer coordinator, we'd need two rolling bounces instead of one rolling bounce to complete the upgrade, whereas Connect only need one rolling bounce.

## Streams

Streams embeds its own assignor which will determine the supported protocol. In 2.4 it will enable cooperative rebalancing by default, so a specific upgrade path must be followed in order to safely upgrade to 2.4+ since assignors on the earlier versions do not know how to handle a cooperative rebalance safely (or even what it is). To do so you must perform two rolling bounces as follows:

1. The first rolling bounce is to replace the byte code (i.e. swap the jars): set the UPGRADE\_FROM config to 2.3 (or whatever version you are upgrading from) and then bounce each instance to upgrade it to 2.4. The UPGRADE\_FROM config will turn off cooperative rebalancing in the cluster until everyone is on the new byte code, and we can be sure that the leader will be able to safely complete a rebalance.
2. The second rolling bounce is to remove the UPGRADE\_FROM config: simply remove this and bounce each instance for it to begin using the cooperative protocol. Note that unlike plain consumer apps, this means you will have some members on COOPERATIVE while others may still be on EAGER – as long as everyone is on version 2.4 or later, this is safe as the Streams assignor knows how to handle the assignment with either protocol in use.

Note that as long as the UPGRADE\_FROM parameter is set to 2.3 or lower, Streams will stay on EAGER. If for some reason you decide you would like to stay on eager, or return to it after switching to cooperative, you can simply set/leave the UPGRADE\_FROM parameter in place. If you intend to use this config to stay on eager even after upgrading, it is recommended that you set it to a version in the range 2.0 - 2.3 as setting it to earlier than 2.0 will force Streams to remain stuck on an earlier metadata version.

These changes should all be fairly transparent to Streams apps, as there are no semantics only improved rebalancing performance. However, users using Interactive Queries (IQ) or implementing a StateListener will notice that Streams spends less time in the REBALANCING state, as we will not transition to that until the end of the rebalance. This means all owned stores will remain open for IQ while the rebalance is in progress, and Streams will continue to restore active tasks if there are any that are not yet running, and will process standbys if there aren't.

## Allow Consumer to Return Records in Rebalance

As summarized in



Unable to render Jira issues macro, execution error.

, a further optimization would be to allow consumers to still

return messages belong to its owned partitions even when it is within a rebalance.

In order to do this, we'd need to allow the consumer#commit API to throw RebalanceInProgressException if it is committing offset while a rebalance is undergoing.

```

/**
 * ...
 *
 * @throws org.apache.kafka.common.errors.RebalanceInProgressException if the consumer instance is in the
middle of a rebalance
 *
 * so it is not yet determined which partitions would be assigned to the consumer. In such cases
you can first
 *
 * complete the rebalance by calling {@link #poll(Duration)} and commit can be reconsidered
afterwards.
 *
 * NOTE when you reconsider committing after the rebalance, the assigned partitions may have
changed,
 *
 * and also for those partitions that are still assigned their fetch positions may have changed
too
 *
 * if more records are returned from the {@link #poll(Duration)} call.
 * ...
 */
@Override
public void commitSync() {
    commitSync(Duration.ofMillis(defaultApiTimeoutMs));
}

```

With this optimization (implemented in 2.5.0) consumer groups can continue to process some records even while a rebalance is in progress. This means that in addition to processing standby and restoring tasks during a rebalance, Streams apps will be able to make progress on running active tasks.

## Looking into the Future: Heartbeat Communicated Protocol

Note that the above upgrade path relies on the fact that COOPERATIVE and EAGER members can work together within the same generation as long as the leader recognize both; this however may not be true moving forward if we add a third rebalance protocol. One idea to resolve this in the future is that, instead of letting the members to decide which protocol to use "locally" before sending the join-group request, we will use Heartbeat request / response to piggy-back the communication of the group's supported protocols and let members to rely on that "global" information to make decisions. More specifically:

- On Heartbeat Request, we will add additional field as a list of protocols that this member supports.
- On Heartbeat Response, we will add additional field as a single protocol indicating which to use if the error code suggests re-joining the group.

The broker, upon receiving the heartbeat request, if the indicated supported protocols does not contain the one it has decided to use for the up-coming rebalance, then reply with an fatal error.

## Looking into the Future: Assignor Version

One semi-orthogonal issue with the current assignor implementation is that, the same assignor may evolve its encoded user metadata byte format; and when that happens, if the leader does not recognize the new format it will cause it to crash. For example today StreamsAssignor has evolved to version 4 of its user metadata bytes, and to cope with old versioned leader we have introduced the "version probing" feature which will require additional rebalances.

To resolve this issue, I'd propose to modify the JoinGroup protocol in this KIP as well to take the read `protocol version` from the PartitionAssignor.

```

JoinGroupRequest (v5) => groupId SessionTimeout RebalanceTimeout memberId ProtocolType Protocols ProtocolVersion
  GroupId                => String
  SessionTimeout          => Int32
  RebalanceTimeout        => Int32
  MemberId                => String
  ProtocolType            => String
  Protocols                => List<Protocol>

Protocol (v0) => ProtocolName ProtocolMetadata
  ProtocolName            => String
  ProtocolMetadata        => Bytes
  ProtocolVersion          => Int32                // new field

```

And then on the broker side, when choosing the leader it will pick the one with the highest protocol version instead of picking it "first come first serve".

Although this change will not benefit the upgrade path at this time, in the future if we need to upgrade the assignor again, as long as they would not change the rebalance semantics (e.g. like we did in this KIP, from "eager" to "cooperative") we can actually use one rolling bounce instead since as long as there's one member on the newer version, that consumer will be picked.

For example, this can also help saving "version probing" cost on Streams as well: suppose we augment the join group schema with `protocol version` in Kafka version 2.3, and then with both brokers and clients being in version 2.3+, on the first rolling bounce where subscription and assignment schema and / or user metadata has changed, this protocol version will be bumped. On the broker side, when receiving all member's join-group request, it will choose the one that has the highest protocol version (also it assumes higher versioned protocol is always backward compatible, i.e. the coordinator can recognize lower versioned protocol as well) and select it as the leader. Then the leader can decide, based on its received and deserialized subscription information, how to assign partitions and how to encode the assignment accordingly so that everyone can understand it. With this, in Streams for example, no version probing would be needed since we are guaranteed the leader knows everyone's version -- again it is assuming that higher versioned protocol is always backward compatible -- and hence can successfully do the assignment at that round.

## Edge Cases Discussion

This proposal depends on user's correct behavior that when upgrading to the new version everyone is still using "eager" protocol. But if user makes mistakes, it is still not going to fall into undefined behavior, as the assignor mechanism will guarantee that we must form a consensus on the protocol names, and whoever does not support the chosen one will be kicked out of the group and hence users would be notified about the mis-configure.

There's a few edge cases to illustrate this:

### Non-active partition assignor

In some assignors like StreamsPartitionAssignor, there are secondary type of resources (e.g. standby-tasks) being assigned in addition to the active partition assignment which may not obey the cooperative manner since the assignor may not know the "currently owned" list of such resources from the subscription, and therefore it could immediately reassign such resources from one owner to another even if they are not revoked at all from the current owner yet.

In this case, as long as the secondary resources does not require any synchronization between its owners like the active partition (e.g. reading previously committed offsets) it should be fine. Take StreamsPartitionAssignor for example, its standby-task are owned separately and independently between instances and are only depend on local state directories (including the data as well as the checkpoint file corresponding to the offset). When a standby task is assigned to a host:

- If the host does not contain any state directory for this task, it would bootstrap the standby-task from scratch, this is okay;
- If the host does contain a state directory for this task, AND it also has this standby-task in previous generation (and it is not revoked at all), in this case we should just proceed normally and it is okay;
- If the host does contain a state directory for this task, AND it does not have this standby-task in previous generation, then we should find its checkpoint offset from the current directory gracefully committed when this task migrates out of this instance more than one generation ago. It would resume from that checkpointed offset and is okay.

So more generally, an assignor supporting COOPERATIVE protocol and is assigning other types of resources than the partition as well would need to make sure that these resource assignment is also respecting the COOPERATIVE rules, or their assignment does not require any synchronization between instances as well.

### Downgrading and Old-Versioned New Member

If a consumer is downgraded incorrectly after the above upgrade path is complete: i.e. it just replaced with the old jar without changing any configs, it is treated as first leaving the group, and then rejoining the group as a new member with "eager". This situation can also be reflected when a new member with "eager" is joining a group (probably mistakenly) whereas everyone else have been switched to "cooperative".

At the moment, no consensus protocol can be chosen with this member joining, and hence this member or everyone else will be kicked out of the group with a fatal error.

The right way to downgrade is to perform a rolling bounce to first add back the RangeAssignor (or whichever assignor you wish to use), and then perform a second rolling bounce in which you remove the CooperativeStickyAssignor and also downgrade the consumers to the old byte code. It's essentially the same as the upgrade path, but in reverse.

## Public Interface

This is to quickly summarize what we would change on the public interface exposed to the user.

1. Introducing the new {ConsumerPartitionAssignor} interface and deprecate the old {PartitionAssignor} interface (details see [ConsumerProtocol](#)).
2. Augmenting the existing {ConsumerRebalanceListener} interface with the new {onPartitionsLost} function (details see [ConsumerProtocol](#)).
3. Adding a list of new metrics to the consumer instance reflecting rebalance events (details see [ConsumerMetrics](#)).
4. Augmenting the JoinGroupRequest protocol with the protocol version (details see [LookingintotheFuture:AssignorVersion](#)).

## Compatibility, Deprecation, and Migration Plan

### Minimum Version Requirement

This change requires Kafka broker version  $\geq 0.9$ , where broker will react with a rebalance when a normal consumer rejoin the encoded metadata. Client application needs to update to the earliest version which includes KIP-429 version 1.0 change.

## Recommended Upgrade Procedure

As we have mentioned above, a new protocol type shall be created. To ensure smooth upgrade, we need to make sure the existing job won't fail. The procedure is described above.

## Rejected Alternatives

Another solution that we have discussed about, is to make each assignor only supports one protocol, i.e.:

```
interface PartitionAssignor {  
  
    // existing interfaces  
  
    short version();           // new API, the version of the assignor which indicate the user metadata /  
    algorithmic difference.  
  
    String name();  
  
    PartitionAssignorProtocol supportedProtocol(); // new API, indicate the ConsumerCoordinator the  
    rebalance strategy it would work with.  
}
```

The existing built-in Assignor implementations will then be updated to:

	Highest Version	Supported Strategy	Notes
RangeAssignor	0	Eager	Current default.
RoundRobinAssignor	0	Eager	
StickAssignor (old)	0	Eager	
StickAssignor (new)	0	Cooperative	Will be new default in 3.0
StreamsAssignor (old)	4	Eager	
StreamsAssignor (new)	4	Cooperative	

Although it makes the upgrade path simpler since we would no longer need the "rebalance.protocol" config on consumer anymore, while just encoding multiple assignors during the first rolling bounce of the upgrade path, it requires duplicated assignor class (of course, the new class could just be extending from the old one and there's not much LOC duplicated) which is a bit cumbersome.