KIP-848: The Next Generation of the Consumer Rebalance Protocol

Status

Current state: *Accepted*

Discussion thread: *Thread 1* and *Thread 2*

JIRA: *here*

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

Note this is a joint work proposed by David Jacot, guozhang Wang and Jason Gustafson.

Motivation

It has been about 8 years since we introduced the so-called new consumer which does group membership and rebalancing through Kafka. Although it was a huge improvement over the old Zookeeper based consumer, it has still been a major pain point from an operation perspective. There are multiple reasons for this, let's dive into them:

- The protocol relies on thick clients. Thick clients are annoying in many ways:
    - We have had many bugs in the rebalance protocol over the last years and the majority of them required client side bug fixes. In the cloud area, this is really annoying because we effectively depend on the adoption of the clients in order to fix the issues in production. Unfortunately, the adoption is usually rather slow in the Kafka community;
    - It is almost impossible to debug issues in the protocol without having access to the client logs. In the cloud area, it is a bit annoying to have to request client logs to troubleshoot your system;
    - The clients specify the so-called embedded protocol in the rebalance protocol. While this allows the core protocol to be reused for different purposes, for instance it is used by both the consumer and connect in Apache Kafka, this makes inspecting the state on the broker side hard because the brokers get a bunch of raw bytes. The compatibility of the embedded protocols has also been a challenge; and
    - The clients are responsible for monitoring the metadata and for triggering rebalances. This has caused all sorts of issues in the past because clients of a given group might have a different view of the metadata at a given point in time.
- The protocol relies on a group-wide synchronization barrier. This means that a single misbehaving consumer can take down or disturb the whole group because a rebalance of the whole group is required whenever a consumer joins, leaves or fails. This also limits its scalability as the cost of a rebalance increases with the number of members in the group. Even the cooperative rebalancing protocols depend on the barrier. Specifically, one of the deficiencies of the cooperative protocol is that offsets cannot be committed while the consumer is waiting on the rebalance to complete. So even though a consumer can keep fetching while the rebalance is in progress, it still tends to get stuck behind the barrier.

- The protocol has gotten too complex over the years. We started with a rather simple protocol and we have extended it a few times over the years. For instance, we introduced KIP-429: Kafka Consumer Incremental Rebalance Protocol, KIP-345: Introduce static membership protocol to reduce consumer rebalances, and a few others. All the incremental changes that we have made have increased the complexity of the protocol.
- The group protocol has been used for general state propagation between members. This is especially the case for power users such as Kafka Streams. While the state propagation is not an issue in itself, the protocol only propagates the state during a rebalance so we have introduced fake or dummy rebalance with the sole goal to propagate some new state to the leader of the group or to all the members of the group. This has been very confusing for our users both on the client side and the broker side. This also makes the interpretation of rebalances through metrics or logs more difficult.

Design Goals

We propose to introduce a new group membership and rebalance protocol for the Kafka Consumer and, by extensions, Kafka Streams. The proposed protocol is built on top of the following design goals.

- The protocol should be truly incremental and cooperative and should not rely on a global synchronization barrier anymore. Ideally, a consumer should not be impacted at all by a rebalance if its assignment is not changed.
- The complexity should move away from the consumer to the group coordinator. We want to be able to troubleshoot issues without requiring client logs and we want to fix issues without having to wait on consumer adoption.
- The protocol should still allow power users such as Kafka Streams to run assignment logic on the client. This is important for Kafka Streams to remain independent from the broker. However, we want this process to be driven and controlled by the group coordinator.
- The protocol should provide the same guarantee as the current protocol that is at-least-once in the worst case scenario and exactly-once when the hand off between members is clean.
- The protocol should support upgrading the consumers without downtime.

Note that Kafka Connect is not supported by this new protocol. We discuss how Kafka Connect could evolve by using a similar protocol in the future in the future work section.

Proposed Changes

Rebalance Protocol in a Nutshell

The proposed rebalance protocol is based on the concept of a declarative assignment for the group and the use of reconciliation loops to drive members toward their desired assignment. Members can independently converge and the group coordinator takes care of resolving the dependencies - e.g. revoking a partition before it can be assigned - between the members if any.

The desired (or target) assignment is either directly computed by the group coordinator using a server side assignor or computed by one of the group members if a client side assignor is specified. The former is the new default for consumers while the latter allows power users such as Kafka Streams to continue using purpose-built assignors. It is important to note that the entire rebalance process is driven by the group coordinator with this new protocol.

Unlike the current protocol which keeps the heartbeat mechanism as lightweight as possible, the new protocol piggybacks on it to let the group coordinator assign/revoke partitions to/from group members while allowing group members to propagate their current state to the group coordinator. The ConsumerGroupHeartbeat API is introduced for this purpose.

When a client side assignor is used, the group coordinator requests the assignment from one group member by notifying it via the heartbeat protocol. The chosen member uses the ConsumerGroupPrepareAssignment API and the ConsumerGroupInstallAssignment API to respectively get the current state of the group and to install the computed assignment. Thanks to this, the input of the client side assignor is entirely driven by the group coordinator. The consumer is no longer responsible for maintaining any state besides its assigned partitions.

The new protocol's RPCs are specified in the details in the public interfaces section of this document while the details of the rebalance logic is described in the next chapter.

Group Coordinator

This KIP proposes to evolve the group coordinator to rely on an event loop. The rational of using an event loop is that 1) it simplifies the concurrency and 2) enables simulation testing. The group coordinator will have a replicated state machine per *__consumer_offsets* partitions, where each replicated state machine is modelled as an event loop. Those replicated state machines will be executed in *group.coordinator.threads* threads.

Consumer Groups

The group coordinator already supports the so called consumer groups. Those groups are groups which implement the *consumer* embedded protocol type. With the introduction of the new consumer rebalance protocol, we need a way to differentiate the existing groups from the new consumer groups. This is important because the existing group relies on a specific set of APIs whereas the new consumer group will use a different set of APIs.

Therefore, we propose to introduce the notion of types within the group coordinator. This will allow us to support different types of groups in the future. We propose to call the current group *generic* as it represents a generic implementation of the membership protocol which is specialized by a protocol type and to call the new consumer group *consumer*. Effectively, we would have consumer groups and generic groups using the consumer embedded protocol, the old one and the new one proposed in this document.

The ListGroups API will be extended to support both filtering on the group types and returning the group types of the queried groups.

Data Model

Before diving into the details of the new rebalance process, let's define the data model of the group as the group coordinator will bookkeep it. Note that this data model is a logical one. The detailed records are described in the Public Interfaces section of this document.

Consumer Group & Member

The group and the members represents the current state of a consumer group.

Consumer Group		
Name	**Type**	**Description**
Group ID	string	The group ID as configured by the consumer. The ID uniquely identifies the group.
Group Epoch	int32	The current epoch of the group. The epoch is incremented by the group coordinator when a new assignment is required for the group.
Members	[]Member	The set of members in the group.
Partitions Metadata	[] PartitionMetadata	The metadata of the partitions that the group is subscribed to. This is used to detect partition metadata changes.
Member		
Name	**Type**	**Description**
Member ID	string	The unique identifier of the member. It is generated by the coordinator upon the first heartbeat request and must be used during the lifetime of the member. The ID is similar to an incarnation ID.
Instance ID	string	The instance ID configured by the consumer.
Rack ID	string	The rack ID configured by the consumer.
Client ID	string	The client ID configured by the consumer.
Client Host	string	The client host configured by the consumer.
Subscribed Topic Names	[]string	The current set of subscribed topic names configured by the consumer.
Subscribed Topic Regex	string	The current subscription regular expression configured by the consumer.
Server Assignor	string	The server side assignor used by the group.
Client Assignors	[]Assignor	The list of client-side assignors supported by the member. The order of this list defined the priority.
Assignor		
Name	**Type**	**Description**
Name	string	The unique name of the assignor.
Reason	int8	The reason why the metadata was updated.
Minimum Version	int16	The minimum version of the metadata schema supported by this assignor.
Maximum Version	int16	The maximum version of the metadata schema supported by this assignor.
Version	int16	The version used to encode the metadata.
Metadata	bytes	The metadata provided by the consumer for this assignor.

Target Assignment

The target (or desired) assignment of the group. This represents the assignment that all the members will eventually converge to. It is a declarative assignment which is generated by the assignor based on the group state.

Target Assignment		
Name	**Type**	**Description**
Group ID	string	The group ID as configured by the consumer. The ID uniquely identifies the group.

Assignment Epoch	int32	The epoch of the assignment. It represents the epoch of the group used to generate the assignment. It will eventually match the group epoch.
Assignment Error	int8	The error reported by the assignor.
Members	[]Member	The assignment for each member.

Member		
Name	**Type**	**Description**
Member ID	string	The unique identifier of the member.
Partitions	[] TopicIdPartition	The set of partitions assigned to this member.
Metadata	bytes	The metadata assigned to this member.

Current Assignment

The Current Assignment represents the current epoch and assignment of a member. Note that members of a given group could be at a different epoch but they will all eventually converge to the target assignment.

Current Assignment		
Name	**Type**	**Description**
Group ID	string	The group ID as configured by the consumer. The ID uniquely identifies the group.
Member ID	string	The member ID of this member.
Member Epoch	int32	The current epoch of this member. The epoch is the assignment epoch of the assignment currently used by this member. This epoch is the one used to fence the member (e.g. offsets commit).
Error	int8	The error reported by the assignor.
Partitions	[] TopicIdPartition	The current partitions used by the member.
Version	int16	The version used to encode the metadata.
Metadata	bytes	The current metadata used by the member.

Rebalance Process

The rebalance process is entirely driven by the group coordinator and revolves around three kinds of epoch: the group epoch, the assignment epoch and the member epoch. The process and the epochs are explained in the following chapters.

Group Epoch - Trigger a rebalance

The group coordinator is responsible for triggering a rebalance of the group when the metadata of the group changes. The metadata of the group is used as the input of the assignment function. For tracking this, we introduce the group epoch which represents the generation (or the version) of the group metadata. The group epoch is incremented whenever the group metadata is updated. There are a couple of cases to consider:

- A member joins or leaves the group.
- A member updates its subscriptions.
- A member updates its assignors.
- A member updates its assignors' reason or metadata.
- A member is fenced or removed from the group by the group coordinator.
- The partition metadata is updated. For instance when a new partition is added or a new topic matching the subscribed topics is created.

In all these cases, a new version of the group metadata is persisted by the group coordinator with an incremented group epoch. This also signals that a new assignment is required for the group.

Assignment Epoch - Compute the group assignment

Whenever the group epoch is larger than the target assignment epoch, the group coordinator will trigger the computation of a new target assignment based on the latest group metadata. When the new assignment is computed, the group coordinator persists it. The assignment epoch becomes the group epoch of the group metadata used to compute the assignment.

The group coordinator either directly computes the new target assignment for the group based on its default server-side assignor or requests a new assignment from one of the members in the group. The entire delegation logic for the latter is detailed later in the document.

Member Epoch - Reconciliation of the group

Once a new target assignment is installed, each member will independently reconcile their current assignment with their new target assignment. Ultimately, each member will converge to their target epoch and assignment. The reconciliation process requires three phases:

1. The group coordinator revokes the partitions which are no longer in the target assignment of the member. It does so by providing the intersection of the Current Partitions and the Target Partitions in the heartbeat response until the member acknowledges the revocation in the heartbeat response. The group coordinator will give the *rebalance timeout* to the member for the revocation process to complete or kick it out from the group otherwise.
2. When the group coordinator receives the acknowledgement of the revocation, it updates the member current assignment to its target assignment (and target epoch) and durably persist it.
3. The group coordinator assigns the new partitions to the member. It does so by providing the Target Partitions to the member while ensuring that partitions which are not revoked by other members yet are removed from this set. In other words, new partitions are incrementally assigned to the member when they are revoked by the other members.

The *rebalance timeout* is provided by the member when it joins the group. It is basically the max poll interval configured on the client side. The timer starts ticking when the heartbeat request is processed by the group coordinator.

Assignment Process

Whenever the group epoch is larger than the assignment epoch, the group coordinator must compute a new target assignment for the group. The group coordinator will either directly compute a new assignment with its server side assignor or delegate the assignment to a member of the group if a client-side assignor must be used.

The new target assignment for the group is basically a function of the current group metadata and the current target assignment. One important aspect to note here is that the assignment is declarative now instead of being incremental like it is in the current implementation. In other words, the assignor defines the desired state for the group and let the group coordinator converge to it.

Assignor Selection

The group coordinator has to determine which assignment strategy must be used for the group. The group's members may not have exactly the same assignors at any given point in time - e.g. they may migrate from an assignor to another one for instance. The group coordinator will chose the assignor as follow:

- A client side assignor is used if possible. This means that a client side assignor must be supported by all the members. If multiple are, it will respect the precedence defined by the members when they advertise their supported client side assignors.
- A server side assignor is used otherwise. If multiple server side assignors are specified in the group, the group coordinator uses the most common one. If a member does not provide an assignor, the group coordinator will default to the first one in `group.consumer.assignors`.

Server Side Mode

The server side assignor is pluggable and the client can choose the one that it wants to use by providing its name in the heartbeat request. If the selected assignor does not exist, the group coordinator will reject the heartbeat with an UNSUPPORTED_ASSIGNOR error. The list of supported assignors will be configured in the broker configuration.

We will support two assignors out of the box for Apache Kafka:

- range - org.apache.kafka.server.group.consumer.RangeAssignor - An assignor which co-partitions topics.
- uniform - org.apache.kafka.server.group.consumer.UniformAssignor - An assignor which uniformly assign partitions amongst the members. This is somewhat similar to the existing "sticky" assignor.

Note that in both cases, assignors are sticky. The goal is to minimise partition movements.

Client Side Mode

The client side assignment is executed by the consumer. The overall process has the following phases:

- The group coordinator selects a member to run the assignment logic. The selection is explained later in this chapter.
- The group coordinator notifies the member to compute the new assignment by setting the ShouldComputeAssignment field in its next heartbeat response.
- When the member receives this error, it is expected to call the ConsumerGroupPrepareAssignment API to get the current group metadata and the current target assignment.
- The member computes the new assignment with the relevant assignor.
- The member calls the ConsumerGroupInstallAssignment API to install the new assignment. The group coordinator validates it and persists it.

Note that the group coordinator always installs any new valid assignment, even if the group epoch has changed in the mean time, to ensure that the group can always make progress. We want to avoid the situation where a faulty member could prevent the whole group to move forward. The group coordinator only allows one inflight assignment at the time.

The chosen member is expected to complete the assignment process within the rebalance timeout. The time on the coordinator side starts ticking when the member is notified. If the process is not completed within the rebalance timeout, the group coordinator picks up another member to run the assignment. Note that the previous chosen member is not fenced here because the fencing is only done based on the session.

Metadata Version Handling (KIP-268)

Managing the compatibility of the metadata used by the client side assignors has been a challenge for our powerful users such as Kafka Streams. The metadata is usually versioned but there is not guarantee in the current protocol which ensures that the elected leader is able to handle all the versions used in the group. Kafka Streams introduces the so-called version probing (KIP-268) to mitigate this issue. This mechanism basically allows the leader to downgrade the version used by the other members in the group.

We propose to make the version a first class citizen concept in the new protocol. Every member will advertise the version used to encode their metadata, usually the highest that they support, and the minimum and the maximum version that they can handle. This allow the group coordinator to reason about the versions and to pick the member to run the assignment wisely.

The group coordinator will also ensure that any member joining with a non-overlapping version range is rejected with the UNSUPPORTED_ASSIGNOR error.

Member Selection

The group coordinator can generally pick any members to run the assignment. However, when the members support different version ranges, the group coordinator must select a member which is able to handle all the supported versions. For instance, if we have three members: A [1-5], B [3-4], C [2-4]. Member A must be selected because it supports all the other versions in the group.

Assignment Validation

Before installing any new assignment, the group coordinator will ensure that the following invariants are met:

- All partitions are assigned.
- A partition is assigned only once.
- All member ids are valid. They must correspond to members in the group.

Note that this validation is made with regarding to the metadata used to compute the assignment. The group may have already advanced to a newer group epoch - e.g. a member could have left during the assignment computation.

The installation will be rejected with an INVALID_ASSIGNMENT error if the invariants are not held.

Assignment Error

There could be cases where the the client side assignor can not compute a new assignment. For instance, in the context of Kafka Streams, the members may have a different topology. In this case, the client side assignor can return an error (*PartitionAssignor.Assignment#error*) to the group coordinator. In this case, the group coordinator automatically keeps the current target assignment for group.

Member ID

Every member is uniquely identified by a UUID. This is is called the Member ID. This UUID is generated on the server side and given to the member when it joins the group. It is used in all the communication with the group coordinator and must be kept during the entirely life span of the member (e.g. the consumer). In that sense, it is similar to an incarnation ID.

Heartbeat & Session

The member uses the ConsumerGroupHeartbeat API to establish a session with the group coordinator. The member is expected to heartbeat every *group.consumer.heartbeat.interval.ms* in order to keep its session opened. If it does not heartbeat at least once within the *group.consumer.session.timeout.ms*, the group coordinator will kick the member out from the group. *group.consumer.heartbeat.interval.ms* is defined on the server side and the member is told about it in the heartbeat response. The *group.consumer.session.timeout.ms* is also defined on the server side.

Joining & Leaving

The member joins the group by sending an heartbeat with no Member ID and a member epoch equals to 0. He can rejoins the group with a member epoch equals to 0. He can leaves the group by using a member epoch equals to -1. The member must commit its offsets before leaving.

Fencing

The group coordinator ensures that requests comes from a known Member ID. Any request is rejected with the UNKNOWN_MEMBER_ID error otherwise. It also ensures that the Member Epoch matches the expected member epoch. If not, the request is rejected with the FENCED_MEMBER_EPOCH error. In this case, the member is expected to immediately gives up all its partitions and rejoin the group with the same member ID and a member epoch equals to zero. Details for every API are given in the Public Interfaces section.

Static Membership (KIP-345)

Static membership, introduced in KIP-345, is still supported by this new rebalance protocol. When a member wants to leave temporary – e.g. while being bounced – it should send an heartbeat with a member epoch equals to -2. This signals to the group coordinator that the member left but will rejoin within the session timeout. When the member rejoins with the same instance ID, the group coordinator replaces the old member by the new member and gives back its current assignment.

If the leaving member does not rejoin within the session timeout, the group coordinator kicks it out from the group. If a new member joins with an existing instance ID before the previous member left, the new member is rejected with a UNRELEASED_INSTANCE_ID error as long as the previous member is still present.

Consumer Group States

EMPTY

When a consumer group is created or when the last member leaves the group, the consumer group is EMPTY.

ASSIGNING

When the group epoch is larger than the assignment epoch, the consumer group is ASSIGNING.

Consumer groups relying on the server-side assignor (e.g. regular consumers) are not expected to be in this state because the assignment is computed directly by the Group Coordinator.

RECONCILING

Until all the members have converged to the group epoch, the consumer group is RECONCILING.

STABLE

Once the reconciliation process is completed, the consumer group moves to the STABLE state.

DEAD

Like today, when the group remains EMPTY for a configured period, the group coordinator transitions it to DEAD to delete it.

Dynamic Group Configuration

The new rebalance protocol relies on server side configurations such as *group.consumer.heartbeat.interval.ms* and *group.consumer.session.timeout.ms.* Our goal is to give administrator the ability to use and tweak those settings for their entire consumers fleet. However, it may not always be possible to have values fitting all workloads. Therefore, we propose to extend the IncrementalAlterConfigs and the DescribeConfigs API to support a new resource type called GROUP. This allows users to override the default defined by the administrators. The dynamic group configurations are described in the Public Interfaces section.

The group configurations are stored in the controller like all the other dynamic configurations in the cluster. This allows configurations to be installed independently from whether the group exists or not. Configurations are also preserved if the group is deleted. In ZK mode, the dynamic group configurations will be store in the `/config/groups` znode. In KRaft mode, they are stored in the `ConfigRecord`.

Regex Based Subscription

The group coordinator is responsible of the regex based subscriptions. We will use Google RE2/J to compile and to execute the regular expressions on the server side. This means that all clients, regardless of their language, will use this common regular expression syntax. This should not be an issue for any common regular expressions but may require changes if specifics from the language are used.

MetadataVersion / IBP

The new group coordinator will rely on the *metadata.version* or IBP to enable the new protocol. The exact version will be determined when the feature is ready.

Fail Over

When the Group Coordinator fails over, the newly elected coordinator will load the state from the __consumer_offsets partition(s). When it is done with this, it has a few other duties for the newly loaded groups:

- It has to setup the session timeouts for all the members (like today).
- It has to check whether the topic-partition metadata has changed and potentially trigger a rebalance for the group if it has.
- It has to check whether new topics match the regex subscriptions and trigger a rebalance for the group if new topic do.
- It has to check whether a new assignment is required for the group (group epoch != assignment epoch). If it is the case, the group coordinator can directly compute it using the server side assignor or can trigger a client side assignment computation.

Persistence

We will introduce a new set of records to persist the new consumer group type in the existing *__consumer_offsets* topic. The records are detailed in the public interfaces section of this document.

Consumer

The semantics of the consumer will remain unchanged after this proposal is implemented. The goal is to swap the implementation of the group membership /assignment protocol by the new one.

Feature Flag

A new configuration setting will be used to determine whether the new protocol should be used or not. The feature flag allows the user to control when he starts using or migrating to the new protocol for its application. This is also required by our migration path as we will require to have the software on a specific version which is compatible with the new protocol.

In the case where a consumer would try to use the new protocol against a cluster which does not support it, either because the software is too old or because the feature is not enabled, the consumer would fail starting with a fatal exception.

In the beginning, the new protocol will be disabled by default. We envision enabling it by default in a future major release of Kafka.

Rebalance Process

The rebalance process in the consumer is basically the opposite of the process that was described earlier in this document. The consumer will know at any point in time its current epoch and the list of partitions that it owns. There are a few cases to consider:

Fenced

If the member is fenced by the group coordinator, it will immediately abandon all its partitions and call ConsumerRebalanceListener#onPartitionsLost. It will rejoin the group as a new member afterwards.

Revocation

When the member has got a new target assignment, the group coordinator will notify it that it has to revoke partitions if any partitions must be revoked. The member can determine the revoked partitions by computing the difference between its current local assignment and the assignment received from the group coordinator. The consumer stops fetching from those revoked partitions and, If auto commit is enabled, commits their offsets. Finally, it calls ConsumerRebalanceListener#onPartitionsRevoked and inform the group coordinator that the revocation process in its next heartbeat.

Assignment

When the member transitions to its next epoch, it will receives the its epoch, its assigned partitions and its pending partitions from the group coordinator. When the member transitions to its new epoch, PartitionAssignor#onAssignment is called if a customer assignor assignor is setup. PartitionAssignor#onAssignment receives the new metadata and the new target assignment for the member. The metadata is final for this epoch. The target assignment is also final for this epoch and contains partitions that are not revoked yet by other members. Therefore, the assignor does not know which of them are being fetched or not. It must rely on the ConsumerRebalanceListener for this. Finally, ConsumerRebalanceListener#onPartitionsAssigned is called with the assigned partitions and the consumer starts fetching those partitions.

Whenever new partitions are assigned to the member within the current epoch, the consumer calls ConsumerRebalanceListener#onPartitionsAssigned with those. This means that ConsumerRebalanceListener#onPartitionsAssigned can be called multiple times within an epoch. At most, it will be called N times where N is the number of assigned partitions in the current epoch.

Client-Side Assignor

By default, the consumer will entirely rely on the group coordinator but it will allow specifying a customer assignor on the client-side as already explained in this document. For this purpose, we propose to introduce a new and optional assignor interface in the Consumer called PartitionAssignor. The interface is specified in the public interfaces section of this document. The current assignor interface is strongly tied to the current group membership/assignment protocol so reusing it is not appropriate for two reasons:

- The new protocol does not really fit in the current interface and its semantic is different; and
- It seems preferable to let us evolve the current protocol independently if the need arises.

Topic ID and Topic Recreation

With KIP-516, the Consumer already uses topic Ids on the fetch patch and on the metadata path. However, it does not use them for the assignment and the committed offsets. This KIP closes that gap. The new consumer rebalance protocol works only with topic ids and the OffsetFetch and OffsetCommit APIs are updated to use topic ids as well. This strengthen the semantic of the Consumer. The consumer will be able to correctly handle topic recreation. When a topic is deleted, the Consumer will re-resolve the name to get the new topic id. Then, it updates it subscriptions with the new topic id. The rebalance protocol will resign the old topic id and assign the new one to members so members will consider the topic as a new/different topic. Adding the topic id to the OffsetFetch and OffsetCommit ensures that the Consumer won't use the old committed offsets with the new topic as well. The processing of the new topic with start fresh based on the offset reset policy.

Note that we don't plan to update the Consumer's public API to expose topic ids at this stage so end users won't be able to detect the recreations yet. We might consider doing this in a future KIP.

Deprecate Enforcing Rebalances

Consumer#enforceRebalance will be deprecated and will be a no-op if used when the new protocol is enable. A warning will be logged in this case. Enforcing a rebalance with the new protocol does not make any sense. Instead, power users will have the ability to trigger a reassignment by either providing a non-zero reason or by updating the assignor metadata.

Streams

Kafka Streams remains a power user of the consumer so it will continue extending the consumer by providing an implementation of the new assignor interface. Streams will also rely on a feature flag to enable the new rebalance protocol.

Member Metadata & Assignment Metadata

Member Metadata refers to the metadata provided by a given member from its assignor. *Assignment Metadata* refers to the metadata computed by the assignor for the member. The *Version*, *MinimumVersion*, *MaximumVersion*, *Reason* and *Error* fields are now first class citizen in the rebalance protocol so Stream does not have the specify them in the metadata anymore. The schemas for respectively the assignor metadata and the assignment metadata are detailed in the Public Interfaces section.

Note that Streams may take this opportunity to do further changes to its metadata. We may extend this KIP or do a follow-up KIP in the future for this.

Assignor Behavior

The assignor behavior remains similar to the existing assignor. The major difference is that the assignor must serialize the assignment metadata of each member with the correct version used by the member. Another difference is that the new assignor must be able to handle the old metadata format as well during the upgrade from the old to the new protocol. This upgrade path is detailed in the upgrade section of this document.

Member Behavior

Upon receiving the assignment, each member would respectively create, close, or recycle tasks as indicated and update the global assignment information, like today. We explained earlier that partitions are incrementally assigned to the member when they are revoked by the others. This means that the assignment metadata may already reference partitions which are not assigned to the member yet. The Streams assignor must consider the assigned partitions as the source of truth in this case.

Each member encodes the lag of its standby tasks in its metadata. We can not update the lag in every heartbeat request because that would constantly trigger reassignment in the group. Instead, when a) the task lag has been reduced within the *acceptable.recovery.lag* threshold or b) the task lag is consistently increasing for some time, the member should consider triggering a rebalance by sending its next heartbeat with the appropriate encoded reason and the updated task lags.

Supporting Online Consumer Group Upgrade

Upgrading to the new protocol or downgrading from it is possible by rolling the consumers, assuming that the new protocol is enabled on the server side, with the correct *group.protocol*. When the first consumer using the new rebalance protocol joins the group, the group is converted from a generic group to a consumer group. When the last consumer using the new rebalance protocol leaves the group, the group is converted back to a generic group. Note that the group epoch starts at the current group generation. During the migration, all the JoinGroup, SyncGroup and Heartbeat calls from the non-upgraded consumers are translated to the new protocol. How? The idea is to basically reconcile each member individually with the old protocol APIs.

Before explaining how that will work, let's recapitulate how the current protocol workflow. First, the members join or re-join the group with the JoinGroup API. The JoinGroup request contains the subscriptions, the owned partitions, etc. When all the members are have joined, the group coordinator picks a leader for the group and sends back the JoinGroup response to all the members. The leader is responsible for computing the assignment. Second, all the members collects their assignment - computed by the leader - by using the SyncGroup API. In parallel, the members heartbeat with the Heartbeat API in order to maintain their session. The Heartbeat API is also used by the group coordinator to inform the members about an ongoing rebalance. All those interactions are synchronized on the generation of the group. It is important to note that the consumer does not make any assumption about the generation id. It basically uses what it receives from the group coordinator. At least, this is how the Java client works. The current rebalance protocol supports two modes: Eager and Cooperative. In the eager mode, the consumer revokes all its partitions before rejoining the group during a rebalance. In the cooperative mode, the consumer does not revoke any partitions before rejoining the group. However, it revokes the partitions that it does not own anymore when it receives it new assignment and rejoins immediately if he had to revoke any partitions.

The new protocol relies on the ConsumerGroupHeartbeat API to do all the above. Concretely, the API updates the group state, provides the partitions owned by the consumer, gets back the assignment, and updates the session. We can remap those to the old protocol as follow: the JoinGroup API updates the group state and provides the partitions owned, the SyncGroup API gets back the assignment, and the Heartbeat API updates the session. The main difference here is that the JoinGroup and SyncGroup does not run continuously. The group coordinator has to trigger it when it is need by returning the REBALANCE_IN_PROGRESS error in the heartbeat response.

The idea is to manage each members individually while relying on the new engine to do the synchronization between them. Each member will use rebalance loops to update the group coordinator and collect their assignment. The group coordinator will ensure that a rebalance is triggered when one needs to update its assignments. This process is detailed in the next sub-chapters:

Member States

Let's start by defining a state machine for each member:

- PrepareRebalance - The member enters this state when a rebalance is required. The member must rejoin within the rebalance timeout or the coordinator kicks it out from the group. We will discuss later when a rebalance is required.
- CompletingRebalance - The member enters this state when the join group has been received. The member must sync within the rebalance timeout or the coordinator kicks it out from the group.
- Stable - The member enters this state when it has completed the rebalance loop.

We use the terminology already use in the current protocol intentionally here in order to facilitate the reasoning.

JoinGroup Handling

The JoinGroup API is handled as follow:

- The basic validations are applied. The request is rejected with the appropriate error if the validation fails.
    - The group must exist.
    - The member must exist or be created.

    - The embedded protocol must be on version >= 3.
    - The generation id must match the current member epoch. The generation id is provided in the embedded protocol.
- The group state is updated if needed.
    - The group epoch is incremented if a new assignment is required. See "Group Epoch - Trigger a rebalance" chapter for details about the rebalance triggers.
    - The Topics provided in the embedded consumer protocol are used to update the SubscribedTopicNames.
    - The Protocols are converted to Assignors where the UserData become the metadata and the version is set to -1.
- The current member assignment is updated if needed:
    - If the member has revoked all its partitions or the required partitions, the member can transition to its next epoch. The target assignment becomes the current assignment. The group coordinator replies with the new member epoch as the generation id.
    - If the member has to revoke partitions, the group coordinator replies with the current member epoch as the generation id.
- The member transitions to CompletingRebalance state.

SyncGroup Handling

The SyncGroup API is handled as follow:

- The basic validations are applied.
    - The group must exist.
    - The member must exist.
    - The generation id must match the current member epoch. The generation id is provided in the embedded protocol.
- The group coordinator replies with the current assignment. There are two cases to consider here:
    - If the member epoch is smaller than the target epoch, the coordinator replies with the intersection between the current assignment and target assignment. This is used to revoke the partitions not owned anymore. In this case, we know that the member will rejoins the group if it revokes partitions. That will automatically trigger another rebalance to collect the newly assigned partitions.
    - If the member epoch is equals to the target epoch, the coordinator replies with the current assignment - the partitions not revoked by other members yet.
- The group coordinator serializes the assignment with the embedded consumer protocol.
- The member transitions to Stable state.

Heartbeat Handling

The Heartbeat API is handled as follow:

- The basic validations are applied.
    - The group must exist.
    - The member must exist.
    - The generation id must match the current member epoch. The generation id is provided in the embedded protocol.
- The member session is updated.
- The group coordinator replies with REBALANCE_IN_PROGRESS if the member is in the PrepareRebalance; NONE is used otherwise.

Rebalance Triggers

The group coordinator will trigger a rebalance in the following cases:

- A new assignment is installed. In this case, all non-upgraded members must be rebalanced.
- A member is rebalanced when all its newly assigned partitions have been revoked by other members.

Public Interfaces

This section lists the changes impacting the public interfaces.

KRPC

New Errors

- FENCED_MEMBER_EPOCH - The member epoch is fenced by the coordinator. The member must abandon all its partitions and rejoins.
- STALE_MEMBER_EPOCH - The member epoch is stale. The member must retry after receiving its updated member epoch via the ConsumerGroupHeartbeat API.
- UNRELEASED_INSTANCE_ID - The instance ID is still used by another member. The member must leave first.
- UNSUPPORTED_ASSIGNOR - The assignor used by the member or its version range are not supported by the group.

ConsumerGroupHeartbeat API

The ConsumerGroupHeartbeat API is the new core API used by consumers to form a group. The API allows members to advertise their subscriptions, their state, their assignors, and their owned partitions. The group coordinator uses it to assign/revoke partitions to/from members. This API is also used as a liveness check.

Request Schema

The member must set all the (top level) fields when it joins for the first time or when an error occurs (e.g. request timed out). Otherwise, it is expected to only fill in the fields which have changed since the last heartbeat.

```
{
  "apiKey": TBD,
  "type": "request",
  "listeners": ["zkBroker", "broker"],
  "name": "ConsumerGroupHeartbeatRequest",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
      { "name": "GroupId", "type": "string", "versions": "0+", "entityType": "groupId",
        "about": "The group identifier." },
      { "name": "MemberId", "type": "string", "versions": "0+",
        "about": "The member id generated by the coordinator. The member id must be kept during the entire
lifetime of the member." },
      { "name": "MemberEpoch", "type": "int32", "versions": "0+",
        "about": "The current member epoch; 0 to join the group; -1 to leave the group." },
      { "name": "InstanceId", "type": "string", "versions": "0+", "nullableVersions": "0+", "default": "null",
        "about": "null if not provided or if it didn't change since the last heartbeat; the instance Id
otherwise." },
      { "name": "RackId", "type": "string", "versions": "0+",  "nullableVersions": "0+", "default": "null",
        "about": "null if not provided or if it didn't change since the last heartbeat; the rack ID of consumer
otherwise." },
      { "name": "RebalanceTimeoutMs", "type": "int32", "versions": "0+", "default": -1,
        "about": "-1 if it didn't chance since the last heartbeat; the maximum time in milliseconds that the
coordinator will wait on the member to revoke its partitions otherwise." },
      { "name": "SubscribedTopicNames", "type": "[]string", "versions": "0+", "nullableVersions": "0+",
"default": "null",
        "about": "null if it didn't change since the last heartbeat; the subscribed topic names otherwise." },
      { "name": "SubscribedTopicRegex", "type": "string", "versions": "0+", "nullableVersions": "0+",
"default": "null",
        "about": "null if it didn't change since the last heartbeat; the subscribed topic regex otherwise" },
      { "name": "ServerAssignor", "type": "string", "versions": "0+", "nullableVersions": "0+", "default":
"null",
        "about": "null if not used or if it didn't change since the last heartbeat; the server side assignor to
use otherwise." },
      { "name": "ClientAssignors", "type": "[]Assignor", "versions": "0+", "nullableVersions": "0+", "default":
"null",
        "about": "null if not used or if it didn't change since the last heartbeat; the list of client-side
assignors otherwise.", "fields": [
          { "name": "Name", "type": "string", "versions": "0+",
            "about": "The name of the assignor." },
          { "name": "MinimumVersion", "type": "int16", "versions": "0+",
            "about": "The minimum supported version for the metadata." },
          { "name": "MaximumVersion", "type": "int16", "versions": "0+",
            "about": "The maximum supported version for the metadata." },
          { "name": "Reason", "type": "int8", "versions": "0+",
            "about": "The reason of the metadata update." },
          { "name": "MetadataVersion", "type": "int16", "versions": "0+",
            "about": "The version of the metadata." },
          { "name": "MetadataBytes", "type": "bytes", "versions": "0+",
            "about": "The metadata." }
      ]},
      { "name": "TopicPartitions", "type": "[]TopicPartitions", "versions": "0+", "nullableVersions": "0+",
"default": "null",
        "about": "null if it didn't change since the last heartbeat; the partitions owned by the member.",
"fields": [
          { "name": "TopicId", "type": "uuid", "versions": "0+",
            "about": "The topic ID." },
          { "name": "Partitions", "type": "[]int32", "versions": "0+",
            "about": "The partitions." }
      ]}
  ]
}
```

Required ACL

- Read Group

Request Validation

INVALID_REQUEST is returned should the request not obey to the following invariants:

- GroupId must be non-empty.
- MemberId must be non-empty.
- MemberEpoch must be >= -1.
- InstanceId, if provided, must be non-empty.
- RebalanceTimeoutMs must be larger than zero in the first heartbeat request.
- SubscribedTopicNames or SubscribedTopicRegex must be, at minimum, in the first heartbeat request when member epoch is 0.
- SubscribedTopicRegex must be a valid regular expression.
- ServerAssignor and ClientAssignors cannot be used together.
- Assignor.Name must be non-empty.
- Assignor.MinimumVersion must be >= -1.
- Assignor.MaximumVersion must be >= 0 and >= Assignor.MinimumVersion.
- Assignor.Version must be in the >= Assignor.MinimumVersion and <= Assignor.MaximumVersion.

UNSUPPORTED_ASSIGNOR is returned should the request not obey to the following invariants:

- ServerAssignor must be supported by the server.
- ClientAssignors' version range must overlap with the other members in the group.

Request Handling

When the group coordinator handle a ConsumerGroupHeartbeat request:

1. Lookups the group or creates it.
2. Creates the member should the member epoch be zero or checks whether it exists. If it does not exist, UNKNOWN_MEMBER_ID is returned.
3. Checks whether the member epoch matches the member epoch in its current assignment. FENCED_MEMBER_EPOCH is returned otherwise. The member is also removed from the group.
    - There is an edge case here. When the group coordinator transitions a member to its target epoch, the heartbeat response with the new member epoch may be lost. In this case, the member will retry with the member epoch that he knows about and its request will be rejected with a FENCED_MEMBER_EPOCH. This is not optimal. Instead, the group coordinator could accept the request if the partitions owned by the members are a subset of the target partitions. If it is the case, it is safe to transition the member to its target epoch again.
4. Updates the members informations if any. The group epoch is incremented if there is any change. See "Group Epoch - Trigger a rebalance" chapter for details about the rebalance triggers.
5. Reconcile the member assignments as explained earlier in this document.

Response Schema

The group coordinator will only set the Assignment field until the member acknowledges that it has converged to the desired assignment. This is done to ensure that the members converge to the target assignment.

```
{
  "apiKey": TBD,
  "type": "response",
  "name": "ConsumerGroupHeartbeatResponse",
  "validVersions": "0",
  "flexibleVersions": "0+",
  // Supported errors:
  // - GROUP_AUTHORIZATION_FAILED
  // - NOT_COORDINATOR
  // - COORDINATOR_NOT_AVAILABLE
  // - COORDINATOR_LOAD_IN_PROGRESS
  // - INVALID_REQUEST
  // - UNKNOWN_MEMBER_ID
  // - FENCED_MEMBER_EPOCH
  // - UNSUPPORTED_ASSIGNOR
  // - UNRELEASED_INSTANCE_ID
  // - GROUP_MAX_SIZE_REACHED
  "fields": [
    { "name": "ThrottleTimeMs", "type": "int32", "versions": "0+",
      "about": "The duration in milliseconds for which the request was throttled due to a quota violation, or
zero if the request did not violate any quota." },
    { "name": "ErrorCode", "type": "int16", "versions": "0+",
      "about": "The top-level error code, or 0 if there was no error" },
    { "name": "ErrorMessage", "type": "string", "versions": "0+", "nullableVersions": "0+", "default": "null",
      "about": "The top-level error message, or null if there was no error." },
    { "name": "MemberId", "type": "string", "versions": "0+", "nullableVersions": "0+", "default": "null",
      "about": "The member id generated by the coordinator. Only provided when the member joins with
MemberEpoch == 0." },
    { "name": "MemberEpoch", "type": "int32", "versions": "0+",
      "about": "The member epoch." },
    { "name": "ShouldComputeAssignment", "type": "bool", "versions": "0+",
      "about": "True if the member should compute the assignment for the group." },
    { "name": "HeartbeatIntervalMs", "type": "int32", "versions": "0+",
```

```
      "about": "The heartbeat interval in milliseconds." },
    { "name": "Assignment", "type": "Assignment", "versions": "0+", "nullableVersions": "0+", "default": "null",
          "about": "null if not provided; the assignment otherwise.", "fields": [
        { "name": "Error", "type": "int8", "versions": "0+",
          "about": "The assigned error." },
        { "name": "AssignedTopicPartitions", "type": "[]TopicPartitions", "versions": "0+",
          "about": "The partitions assigned to the member that can be used immediately." },
        { "name": "PendingTopicPartitions", "type": "[]TopicPartitions", "versions": "0+",
          "about": "The partitions assigned to the member that cannot be used because they are not released by
their former owners yet." },
        { "name": "MetadataVersion", "type": "int16", "versions": "0+",
          "about": "The version of the metadata." },
        { "name": "MetadataBytes", "type": "bytes", "versions": "0+",
          "about": "The assigned metadata." }
        ]}
  ],
  "commonStructs": [
    { "name": "TopicPartitions", "versions": "0+", "fields": [
        { "name": "TopicId", "type": "uuid", "versions": "0+",
          "about": "The topic ID." },
        { "name": "Partitions", "type": "[]int32", "versions": "0+",
          "about": "The partitions." }
    ]}
  ]
}
```

Response Handling

If the response contains no error, the member will reconcile its current assignment towards its new assignment. It does the following:

1. It updates its member epoch.
2. It computes the difference between the old and the new assignment to determine the revoked partitions and the newly assignment partitions. There should be either revoked partitions or newly assignment partitions. The protocol never does both together.
    a. It revokes the partitions, commit all the offsets, and calls ConsumerRebalanceListener#onPartitionsRevoked.
    b. It assigns the new partitions, calls PartitionAssignor#onAssignment if one is defined and calls ConsumerRebalanceListener#onPartitionsAssigned.
3. After a revocation, It sends the next heartbeat immediately to acknowledge it.

If the response has ShouldComputeAssignment field set to true, the consumer starts the assignment process.

Upon receiving the UNKNOWN_MEMBER_ID or FENCED_MEMBER_EPOCH error, the consumer abandon all its partitions and rejoins with the same member id and the epoch 0.

Upon receiving the UNRELEASED_INSTANCE_ID error, the consumer should fail.

ConsumerGroupPrepareAssignment API

The ConsumerGroupPrepareAssignment API will be used by the consumer to get the information to feed its client-side assignor.

Request Schema

```
{
  "apiKey": TBD,
  "type": "request",
  "listeners": ["zkBroker", "broker"],
  "name": "ConsumerGroupPrepareAssignmentRequest",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
    { "name": "GroupId", "type": "string", "versions": "0+", "entityType": "groupId",
      "about": "The group identifier." },
    { "name": "MemberId", "type": "string", "versions": "0+",
      "about": "The member id assigned by the group coordinator." },
    { "name": "MemberEpoch", "type": "int32", "versions": "0+",
      "about": "The member epoch." }
  ]
}
```

Required ACL

- Read Group

Request Validation

INVALID_REQUEST is returned should the request not obey to the following invariants:

- GroupId must be non-empty.
- MemberId must be non-empty.
- MemberEpoch must be >= 0.

Request Handling

When the group coordinator handle a ConsumerGroupPrepareAssignmentRequest request:

1. Checks whether the group exists. If it does not, GROUP_ID_NOT_FOUND is returned.
2. Checks whether the member exists. If it does not, UNKNOWN_MEMBER_ID is returned.
3. Checks whether the member epoch matches the current member epoch. If it does not, STALE_MEMBER_EPOCH is returned.
4. Checks whether the member is the chosen one to compute the assignment. If it does not, UNKNOWN_MEMBER_ID is returned.
5. Returns the group state of the group.

Response Schema

```
{
  "apiKey": TBD,
  "type": "response",
  "name": "ConsumerGroupPrepareAssignmentResponse",
  "validVersions": "0",
  "flexibleVersions": "0+",
  // Supported errors:
  // - GROUP_AUTHORIZATION_FAILED
  // - NOT_COORDINATOR
  // - COORDINATOR_NOT_AVAILABLE
  // - COORDINATOR_LOAD_IN_PROGRESS
  // - INVALID_REQUEST
  // - INVALID_GROUP_ID
  // - GROUP_ID_NOT_FOUND
  // - UNKNOWN_MEMBER_ID
  // - STALE_MEMBER_EPOCH
  "fields": [
    { "name": "ThrottleTimeMs", "type": "int32", "versions": "0+",
      "about": "The duration in milliseconds for which the request was throttled due to a quota violation, or
zero if the request did not violate any quota." },
    { "name": "ErrorCode", "type": "int16", "versions": "0+",
      "about": "The top-level error code, or 0 if there was no error" },
    { "name": "ErrorMessage", "type": "string", "versions": "0+", "nullableVersions": "0+", "default": "null",
      "about": "The top-level error message, or null if there was no error." },
    { "name": "GroupEpoch", "type": "int32", "versions": "0+",
      "about": "The group epoch." },
    { "name": "AssignorName", "type": "string", "versions": "0+",
      "about": "The selected assignor." },
    { "name": "Members", "type": "[]Member", "versions": "0+",
      "about": "The members.", "fields": [
      { "name": "MemberId", "type": "string", "versions": "0+",
        "about": "The member ID." },
      { "name": "MemberEpoch", "type": "int32", "versions": "0+",
        "about": "The member epoch." },
      { "name": "InstanceId", "type": "string", "versions": "0+", "nullableVersions": "0+", "default": "null",
        "about": "The member instance ID." },
      { "name": "RackId", "type": "string", "versions": "0+", "nullableVersions": "0+", "default": "null",
        "about": "The member instance ID." },
      { "name": "SubscribedTopicIds", "type": "[]uuid", "versions": "0+",
        "about": "The subscribed topic IDs." },
      { "name": "Assignor", "type": "Assignor", "versions": "0+",
        "about": "The information of the selected assignor",
        "fields": [
        { "name": "Reason", "type": "int8", "versions": "0+",
          "about": "The reason of the metadata update." },
        { "name": "MetadataVersion", "type": "int16", "versions": "0+",
          "about": "The version of the metadata." },
        { "name": "MetadataBytes", "type": "bytes", "versions": "0+",
```

```
          "about": "The assignor metadata." }
      ]},
      { "name": "TopicPartitions", "type": "[]TopicPartition", "versions": "0+",
        "about": "The target topic-partitions of the member.",
        "fields": [
          { "name": "TopicId", "type": "uuid", "versions": "0+",
            "about": "The topic ID." },
          { "name": "Partitions", "type": "[]int32", "versions": "0+",
            "about": "The partitions." }
      ]}
    ]},
    { "name": "Topics", "type": "[]TopicMetadata", "versions": "0+",
      "about": "The topic-partition metadata.",
      "fields": [
        { "name": "TopicId", "type": "uuid", "versions": "0+",
          "about": "The topic ID." },
        { "name": "TopicName", "type": "string", "versions": "0+",
          "about": "The topic name." },
        { "name": "NumPartitions", "type": "int32", "versions": "0+",
          "about": "The number of partitions." }
    ]}
  ]
}
```

Response Handling

If the response contains no error, the member calls the client side assignor with the group state.

Upon receiving the UNKNOWN_MEMBER_ID error, the consumer abandon the process.

Upon receiving the STALE_MEMBER_EPOCH error, the consumer retries when receiving its next heartbeat response with its member epoch.

ConsumerGroupInstallAssignment API

The ConsumerGroupInstallAssignment API will be used by the consumer to install a new assignment for the group. The new assignment is the result of the client-side assignor.

Request Schema

```
{
  "apiKey": TBD,
  "type": "request",
  "listeners": ["zkBroker", "broker"],
  "name": "ConsumerGroupInstallAssignmentRequest",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
    { "name": "GroupId", "type": "string", "versions": "0+", "entityType": "groupId",
      "about": "The group identifier." },
    { "name": "MemberId", "type": "string", "versions": "0+",
      "about": "The member id assigned by the group coordinator." },
    { "name": "MemberEpoch", "type": "int32", "versions": "0+",
      "about": "The member epoch." },
    { "name": "GroupEpoch", "type": "int32", "versions": "0+",
      "about": "The group epoch." },
    { "name": "Error", "type": "int8", "versions": "0+",
      "about": "The assignment error; or zero if the assignment is successful." },
    { "name": "Members", "type": "[]Member", "versions": "0+",
      "about": "The members.", "fields": [
      { "name": "MemberId", "type": "string", "versions": "0+",
        "about": "The member ID." },
      { "name": "Partitions", "type": "[]TopicPartition", "versions": "0+",
        "about": "The assigned topic-partitions to the member.",
        "fields": [
          { "name": "TopicId", "type": "uuid", "versions": "0+",
            "about": "The topic ID." },
          { "name": "Partitions", "type": "[]int32", "versions": "0+",
            "about": "The partitions." }
        ]},
```

```
      { "name": "MetadataVersion", "type": "int32", "versions": "0+",
        "about": "The metadata version." }
      { "name": "MetadataBytes", "type": "bytes", "versions": "0+",
        "about": "The metadata bytes." }
    ]}
  ]
}
```

Required ACL

- Read Group

Request Validation

INVALID_REQUEST is returned should the request not obey to the following invariants:

- GroupId must be non-empty.
- MemberId must be non-empty.
- MemberEpoch must be >= 0.

Request Handling

When the group coordinator handle a ConsumerGroupInstallAssignmentRequest request:

1. Checks whether the group exists. If it does not, GROUP_ID_NOT_FOUND is returned.
2. Checks whether the member exists. If it does not, UNKNOWN_MEMBER_ID is returned.
3. Checks whether the member epoch matches the current member epoch. If it does not, STALE_MEMBER_EPOCH is returned.
4. Checks whether the member is the chosen one to compute the assignment. If it does not, UNKNOWN_MEMBER_ID is returned.
5. Validates the assignment based on the information used to compute it. If it is not valid, INVALID_ASSIGNMENT is returned.
6. Installs the new target assignment.

Response Schema

```
{
  "apiKey": TBD,
  "type": "response",
  "name": "ConsumerGroupInstallAssignmentResponse",
  "validVersions": "0",
  "flexibleVersions": "0+",
  // Supported errors:
  // - GROUP_AUTHORIZATION_FAILED
  // - NOT_COORDINATOR
  // - COORDINATOR_NOT_AVAILABLE
  // - COORDINATOR_LOAD_IN_PROGRESS
  // - INVALID_REQUEST
  // - INVALID_GROUP_ID
  // - GROUP_ID_NOT_FOUND
  // - UNKNOWN_MEMBER_ID
  // - STALE_MEMBER_EPOCH
  // - INVALID_ASSIGNMENT
  "fields": [
    { "name": "ThrottleTimeMs", "type": "int32", "versions": "0+",
      "about": "The duration in milliseconds for which the request was throttled due to a quota violation, or
zero if the request did not violate any quota." },
    { "name": "ErrorCode", "type": "int16", "versions": "0+",
      "about": "The top-level error code, or 0 if there was no error" },
    { "name": "ErrorMessage", "type": "string", "versions": "0+", "nullableVersions": "0+", "default": "null",
      "about": "The top-level error message, or null if there was no error." }
  ]
}
```

Response Handling

If the response contains no error, the member is done with the assignment process.

Upon receiving the STALE_MEMBER_EPOCH error, the consumer retries when receiving its next heartbeat response with its member epoch.

Upon receiving any other errors, the consumer abandon the process.

ConsumerGroupDescribe API

Request Schema

```
{
  "apiKey": TBD,
  "type": "request",
  "listeners": ["zkBroker", "broker"],
  "name": "ConsumerGroupDescribeRequest",
  "validVersions": "0",
  "flexibleVersions": "0+",
  "fields": [
    { "name": "GroupIds", "type": "[]string", "versions": "0+", "entityType": "groupId",
      "about": "The names of the groups to describe" },
    { "name": "IncludeAuthorizedOperations", "type": "bool", "versions": "0+",
      "about": "Whether to include authorized operations." }
  ]
}
```

Required ACL

- Read Group

Request Validation

INVALID_REQUEST is returned should the request not obey to the following invariants:

- GroupIds must be non-empty.

Request Handling

When the group coordinator handle a ConsumerGroupDescribeRequest request:

- Checks whether the group ids exists. If it does not, GROUP_ID_NOT_FOUND is returned.
- Looks up the groups and returns the response.

Response Schema

```
{
  "apiKey": 71,
  "type": "response",
  "name": "ConsumerGroupDescribeResponse",
  "validVersions": "0",
  "flexibleVersions": "0+",
  // Supported errors:
  // - GROUP_AUTHORIZATION_FAILED
  // - NOT_COORDINATOR
  // - COORDINATOR_NOT_AVAILABLE
  // - COORDINATOR_LOAD_IN_PROGRESS
  // - INVALID_REQUEST
  // - INVALID_GROUP_ID
  // - GROUP_ID_NOT_FOUND
  "fields": [
    { "name": "ThrottleTimeMs", "type": "int32", "versions": "0+",
      "about": "The duration in milliseconds for which the request was throttled due to a quota violation, or
zero if the request did not violate any quota." },
    { "name": "Groups", "type": "[]DescribedGroup", "versions": "0+",
      "about": "Each described group.",
      "fields": [
        { "name": "ErrorCode", "type": "int16", "versions": "0+",
          "about": "The describe error, or 0 if there was no error." },
        { "name": "ErrorMessage", "type": "string", "versions": "0+", "nullableVersions": "0+", "default":
"null",
          "about": "The top-level error message, or null if there was no error." }
        { "name": "GroupId", "type": "string", "versions": "0+", "entityType": "groupId",
          "about": "The group ID string." },
        { "name": "GroupState", "type": "string", "versions": "0+",
          "about": "The group state string, or the empty string." },
        { "name": "GroupEpoch", "type": "int32", "versions": "0+",
```

```
          "about": "The group epoch." },
        { "name": "AssignmentEpoch", "type": "int32", "versions": "0+",
          "about": "The assignment epoch." },
        { "name": "AssignorName", "type": "string", "versions": "0+",
          "about": "The selected assignor." },
        { "name": "Members", "type": "[]Member", "versions": "0+",
          "about": "The members.",
          "fields": [
          { "name": "MemberId", "type": "uuid", "versions": "0+",
            "about": "The member ID." },
          { "name": "InstanceId", "type": "string", "versions": "0+", "nullableVersions": "0+", "default":
"null",
            "about": "The member instance ID." },
          { "name": "RackId", "type": "string", "versions": "0+", "nullableVersions": "0+", "default": "null",
            "about": "The member rack ID." },
          { "name": "MemberEpoch", "type": "int32", "versions": "0+",
            "about": "The current member epoch." },
          { "name": "ClientId", "type": "string", "versions": "0+",
            "about": "The client ID." },
          { "name": "ClientHost", "type": "string", "versions": "0+",
            "about": "The client host." },
          { "name": "SubscribedTopicNames", "type": "[]string", "versions": "0+",
            "about": "The subscribed topic names." },
          { "name": "SubscribedTopicRegex", "type": "string", "versions": "0+", "nullableVersions": "0+",
"default": "null",
            "about": "the subscribed topic regex otherwise or null of not provided." },
          { "name": "Assignment", "type": "Assignment", "versions": "0+",
            "about": "The current assignment." },
          { "name": "TargetAssignment", "type": "Assignment", "versions": "0+",
            "about": "The target assignment." },
      { "name": "AuthorizedOperations", "type": "int32", "versions": "3+", "default": "-2147483648",
        "about": "32-bit bitfield to represent authorized operations for this group." }
    ]}
  ],
  "commonStructs": [
    { "name": "Assignment", "versions": "0+", "fields": [
      { "name": "Partitions", "type": "[]TopicPartitions", "versions": "0+",
        "about": "The assigned topic-partitions to the member.", "fields": [
        { "name": "TopicId", "type": "uuid", "versions": "0+",
          "about": "The topic ID." },
        { "name": "TopicName", "type": "string", "versions": "0+",
          "about": "The topic name." },
        { "name": "Partitions", "type": "[]int32", "versions": "0+",
          "about": "The partitions." }
      ]},
      { "name": "Error", "type": "int8", "versions": "0+",
        "about": "The assigned error." },
      { "name": "MetadataVersion", "type": "int32", "versions": "0+",
        "about": "The assignor metadata version." }
      { "name": "MetadataBytes", "type": "bytes", "versions": "0+",
        "about": "The assignor metadata bytes." }
    ]}
  ]
}
```

Response Handling

Nothing particular.

ListGroups API

The existing ListGroups API will be extended to support the notion of group types and to support the new group states.

Request Schema

The *TypesFilter* field is introduced. It allows listing groups of certain types.

```
{
  "apiKey": 16,
```

```
  "type": "request",
  "listeners": ["zkBroker", "broker"],
  "name": "ListGroupsRequest",
  // Version 1 and 2 are the same as version 0.
  //
  // Version 3 is the first flexible version.
  //
  // Version 4 adds the StatesFilter field (KIP-518).
  //
  // Version 5 adds the TypesFilter field (KIP-848).
  "validVersions": "0-5",
  "flexibleVersions": "3+",
  "fields": [
    { "name": "StatesFilter", "type": "[]string", "versions": "4+",
      "about": "The states of the groups we want to list. If empty all groups are returned with their state." },
    { "name": "TypesFilter", "type": "[]string", "versions": "5+",
      "about": "The types of the groups we want to list. If empty all groups are returned" }
  ]
}
```

Required ACL

- Describe Cluster

Request Validation

No particular changes.

Request Handling

The new types filter is handled.

Response Schema

The *GroupType* field is introduced. It represents the type of the group.

```
{
  "apiKey": 16,
  "type": "response",
  "name": "ListGroupsResponse",
  // Version 1 adds the throttle time.
  //
  // Starting in version 2, on quota violation, brokers send out
  // responses before throttling.
  //
  // Version 3 is the first flexible version.
  //
  // Version 4 adds the GroupState field (KIP-518).
  //
  // Version 5 adds the GroupType field (KIP-848).
  "validVersions": "0-5",
  "flexibleVersions": "3+",
  "fields": [
    { "name": "ThrottleTimeMs", "type": "int32", "versions": "1+",
      "ignorable": true,
      "about": "The duration in milliseconds for which the request was throttled due to a quota violation, or
zero if the request did not violate any quota." },
    { "name": "ErrorCode", "type": "int16", "versions": "0+",
      "about": "The error code, or 0 if there was no error." },
    { "name": "Groups", "type": "[]ListedGroup", "versions": "0+",
      "about": "Each group in the response.", "fields": [
      { "name": "GroupId", "type": "string", "versions": "0+",
        "entityType": "groupId",
        "about": "The group ID." },
      { "name": "ProtocolType", "type": "string", "versions": "0+",
        "about": "The group protocol type." },
      { "name": "GroupState", "type": "string", "versions": "4+", "ignorable": true,
        "about": "The group state name." },
      { "name": "GroupType", "type": "string", "versions": "5+", "ignorable": true,
        "about": "The group type name." }
```

```
    ]}
  ]
}
```

Response Handling

No changes.

OffsetCommit API

The version of the API is bumped to 9 to add support for topic ids. The request can either use topic ids or topic names. The consumer will only use topic ids when they are available whereas the admin client will continue to use topic names as per its API.

Request Schema

```
{
  "apiKey": 8,
  "type": "request",
  "listeners": ["zkBroker", "broker"],
  "name": "OffsetCommitRequest",
  // Version 1 adds timestamp and group membership information, as well as the commit timestamp.
  //
  // Version 2 adds retention time.  It removes the commit timestamp added in version 1.
  //
  // Version 3 and 4 are the same as version 2.
  //
  // Version 5 removes the retention time, which is now controlled only by a broker configuration.
  //
  // Version 6 adds the leader epoch for fencing.
  //
  // version 7 adds a new field called groupInstanceId to indicate member identity across restarts.
  //
  // Version 8 is the first flexible version.
  //
  // Version 9 adds TopicId field (KIP-848).
  "validVersions": "0-9",
  "flexibleVersions": "8+",
  "fields": [
    { "name": "GroupId", "type": "string", "versions": "0+", "entityType": "groupId",
      "about": "The unique group identifier." },
    // Renamed field.
    { "name": "GenerationIdOrMemberEpoch", "type": "int32", "versions": "1+", "default": "-1", "ignorable":
true,
      "about": "The generation of the group if the generic group protocol or the member epoch if the consumer
protocol." },
    { "name": "MemberId", "type": "string", "versions": "1+", "ignorable": true,
      "about": "The member ID assigned by the group coordinator." },
    { "name": "GroupInstanceId", "type": "string", "versions": "7+",
      "nullableVersions": "7+", "default": "null",
      "about": "The unique identifier of the consumer instance provided by end user." },
    { "name": "RetentionTimeMs", "type": "int64", "versions": "2-4", "default": "-1", "ignorable": true,
      "about": "The time period in ms to retain the offset." },
    { "name": "Topics", "type": "[]OffsetCommitRequestTopic", "versions": "0+",
      "about": "The topics to commit offsets for.", "fields": [
      // Updated field.
      { "name": "Name", "type": "string", "versions": "0+", "nullableVersions": "9+", "default": "null",
"entityType": "topicName",
        "about": "The topic name."},
      // New field.
      { "name": "TopicId", "type": "uuid", "versions": "9+", "about": "The unique topic ID" },
      { "name": "Partitions", "type": "[]OffsetCommitRequestPartition", "versions": "0+",
        "about": "Each partition to commit offsets for.", "fields": [
        { "name": "PartitionIndex", "type": "int32", "versions": "0+",
          "about": "The partition index." },
        { "name": "CommittedOffset", "type": "int64", "versions": "0+",
          "about": "The message offset to be committed." },
        { "name": "CommittedLeaderEpoch", "type": "int32", "versions": "6+", "default": "-1", "ignorable": true,
          "about": "The leader epoch of this partition." },
        // CommitTimestamp has been removed from v2 and later.
```

```
        { "name": "CommitTimestamp", "type": "int64", "versions": "1", "default": "-1",
          "about": "The timestamp of the commit." },
        { "name": "CommittedMetadata", "type": "string", "versions": "0+", "nullableVersions": "0+",
          "about": "Any associated metadata the client wants to keep." }
      ]}
    ]}
  ]
}
```

Required ACL

- Read Group

Request Validation

INVALID_REQUEST is returned should the request not obey to the following invariants:

- A topic has both a name and an ID set.

Request Handling

The MemberId and the GenerationIdOrMemberEpoch are verified. STALE_MEMBER_EPOCH or UNKNOWN_MEMBER_ID is returned accordingly.

Response Schema

```
{
  "apiKey": 8,
  "type": "response",
  "name": "OffsetCommitResponse",
  // Versions 1 and 2 are the same as version 0.
  //
  // Version 3 adds the throttle time to the response.
  //
  // Starting in version 4, on quota violation, brokers send out responses before throttling.
  //
  // Versions 5 and 6 are the same as version 4.
  //
  // Version 7 offsetCommitRequest supports a new field called groupInstanceId to indicate member identity
across restarts.
  //
  // Version 8 is the first flexible version.
  //
  // Version 9 adds TopicId field and can return STALE_MEMBER_EPOCH, UNKNOWN_MEMBER_ID
  // and UNKNOWN_TOPIC_ID errors (KIP-848).
  "validVersions": "0-9",
  "flexibleVersions": "8+",
  "fields": [
    { "name": "ThrottleTimeMs", "type": "int32", "versions": "3+", "ignorable": true,
      "about": "The duration in milliseconds for which the request was throttled due to a quota violation, or
zero if the request did not violate any quota." },
    { "name": "Topics", "type": "[]OffsetCommitResponseTopic", "versions": "0+",
      "about": "The responses for each topic.", "fields": [
      // Updated field.
      { "name": "Name", "type": "string", "versions": "0+", "nullableVersions": "9+", "default": "null",
"entityType": "topicName",
        "about": "The topic name."},
      // New field.
      { "name": "TopicId", "type": "uuid", "versions": "9+", "about": "The unique topic ID" },
      { "name": "Partitions", "type": "[]OffsetCommitResponsePartition", "versions": "0+",
        "about": "The responses for each partition in the topic.",  "fields": [
        { "name": "PartitionIndex", "type": "int32", "versions": "0+",
          "about": "The partition index." },
        { "name": "ErrorCode", "type": "int16", "versions": "0+",
          "about": "The error code, or 0 if there was no error." }
      ]}
    ]}
  ]
}
```

Response Handling

Upon receiving the STALE_MEMBER_EPOCH error, the consumer retries when receiving its next heartbeat response with its member epoch.

OffsetFetch API

The version of the API is bumped to 9 to add support for topic ids. The request can either use topic ids or topic names. The consumer will only use topic ids when they are available whereas the admin client will continue to use topic names as per its API.

Request Schema

```
{
  "apiKey": 9,
  "type": "request",
  "listeners": ["zkBroker", "broker"],
  "name": "OffsetFetchRequest",
  // In version 0, the request read offsets from ZK.
  //
  // Starting in version 1, the broker supports fetching offsets from the internal __consumer_offsets topic.
  //
  // Starting in version 2, the request can contain a null topics array to indicate that offsets
  // for all topics should be fetched. It also returns a top level error code
  // for group or coordinator level errors.
  //
  // Version 3, 4, and 5 are the same as version 2.
  //
  // Version 6 is the first flexible version.
  //
  // Version 7 is adding the require stable flag.
  //
  // Version 8 is adding support for fetching offsets for multiple groups at a time
  //
  // Version 9 adds GenerationIdOrMemberEpoch, MemberId and TopicId fields (KIP-848).
  "validVersions": "0-9",
  "flexibleVersions": "6+",
  "fields": [
    { "name": "GroupId", "type": "string", "versions": "0-7", "entityType": "groupId",
      "about": "The group to fetch offsets for." },
    // New fields.
    { "name": "GenerationIdOrMemberEpoch", "type": "int32", "versions": "9+", "default": "-1", "ignorable":
true,
      "about": "The generation of the group." },
    { "name": "MemberId", "type": "string", "versions": "9+", "ignorable": true,
      "about": "The member ID assigned by the group coordinator." },
    // End of new fields.
    { "name": "Topics", "type": "[]OffsetFetchRequestTopic", "versions": "0-7", "nullableVersions": "2-7",
      "about": "Each topic we would like to fetch offsets for, or null to fetch offsets for all topics.",
"fields": [
      { "name": "Name", "type": "string", "versions": "0-7", "entityType": "topicName",
        "about": "The topic name."},
      { "name": "PartitionIndexes", "type": "[]int32", "versions": "0-7",
        "about": "The partition indexes we would like to fetch offsets for." }
    ]},
    { "name": "Groups", "type": "[]OffsetFetchRequestGroup", "versions": "8+",
      "about": "Each group we would like to fetch offsets for", "fields": [
      { "name": "groupId", "type": "string", "versions": "8+", "entityType": "groupId",
        "about": "The group ID."},
      { "name": "Topics", "type": "[]OffsetFetchRequestTopics", "versions": "8+", "nullableVersions": "8+",
        "about": "Each topic we would like to fetch offsets for, or null to fetch offsets for all topics.",
"fields": [
        // Updated field.
        { "name": "Name", "type": "string", "versions": "8+", "nullableVersions": "9+", "default": "null",
"entityType": "topicName",
          "about": "The topic name."},
        // New field.
        { "name": "TopicId", "type": "uuid", "versions": "9+", "about": "The unique topic ID" },
        { "name": "PartitionIndexes", "type": "[]int32", "versions": "8+",
          "about": "The partition indexes we would like to fetch offsets for." }
      ]}
    ]},
```

```
    { "name": "RequireStable", "type": "bool", "versions": "7+", "default": "false",
      "about": "Whether broker should hold on returning unstable offsets but set a retriable error code for the
partitions."}
  ]
}
```

Required ACL

- Describe Group

Request Validation

INVALID_REQUEST is returned should the request not obey to the following invariants:

- A topic has both a name and an ID set.

Request Handling

The MemberId and the GenerationIdOrMemberEpoch are verified. STALE_MEMBER_EPOCH, UNKNOWN_MEMBER_ID or ILLEGAL_GENERATION is returned accordingly.

The admin client is not expected to provide the MemberId nor the GenerationIdOrMemberEpoch.

Response Schema

```
{
  "apiKey": 9,
  "type": "response",
  "name": "OffsetFetchResponse",
  // Version 1 is the same as version 0.
  //
  // Version 2 adds a top-level error code.
  //
  // Version 3 adds the throttle time.
  //
  // Starting in version 4, on quota violation, brokers send out responses before throttling.
  //
  // Version 5 adds the leader epoch to the committed offset.
  //
  // Version 6 is the first flexible version.
  //
  // Version 7 adds pending offset commit as new error response on partition level.
  //
  // Version 8 is adding support for fetching offsets for multiple groups
  //
  // Version 9 adds TopicId field and can return STALE_MEMBER_EPOCH, UNKNOWN_MEMBER_ID,
  // ILLEGAL_GENERATION, and UNKNOWN_TOPIC_ID errors.
  "validVersions": "0-8",
  "flexibleVersions": "6+",
  "fields": [
    { "name": "ThrottleTimeMs", "type": "int32", "versions": "3+", "ignorable": true,
      "about": "The duration in milliseconds for which the request was throttled due to a quota violation, or
zero if the request did not violate any quota." },
    { "name": "Topics", "type": "[]OffsetFetchResponseTopic", "versions": "0-7",
      "about": "The responses per topic.", "fields": [
      { "name": "Name", "type": "string", "versions": "0-7", "entityType": "topicName",
        "about": "The topic name." },
      { "name": "Partitions", "type": "[]OffsetFetchResponsePartition", "versions": "0-7",
        "about": "The responses per partition", "fields": [
        { "name": "PartitionIndex", "type": "int32", "versions": "0-7",
          "about": "The partition index." },
        { "name": "CommittedOffset", "type": "int64", "versions": "0-7",
          "about": "The committed message offset." },
        { "name": "CommittedLeaderEpoch", "type": "int32", "versions": "5-7", "default": "-1",
          "ignorable": true, "about": "The leader epoch." },
        { "name": "Metadata", "type": "string", "versions": "0-7", "nullableVersions": "0-7",
          "about": "The partition metadata." },
        { "name": "ErrorCode", "type": "int16", "versions": "0-7",
          "about": "The error code, or 0 if there was no error." }
      ]}
```

```
    ]},
    { "name": "ErrorCode", "type": "int16", "versions": "2-7", "default": "0", "ignorable": true,
      "about": "The top-level error code, or 0 if there was no error." },
    { "name": "Groups", "type": "[]OffsetFetchResponseGroup", "versions": "8+",
      "about": "The responses per group id.", "fields": [
      { "name": "groupId", "type": "string", "versions": "8+", "entityType": "groupId",
        "about": "The group ID." },
      { "name": "Topics", "type": "[]OffsetFetchResponseTopics", "versions": "8+",
        "about": "The responses per topic.", "fields": [
        { "name": "Name", "type": "string", "versions": "8+", "nullableVersions": "9+", "default": "null",
"entityType": "topicName",
          "about": "The topic name."},
        { "name": "TopicId", "type": "uuid", "versions": "9+", "about": "The unique topic ID" },
        { "name": "Partitions", "type": "[]OffsetFetchResponsePartitions", "versions": "8+",
          "about": "The responses per partition", "fields": [
          { "name": "PartitionIndex", "type": "int32", "versions": "8+",
            "about": "The partition index." },
          { "name": "CommittedOffset", "type": "int64", "versions": "8+",
            "about": "The committed message offset." },
          { "name": "CommittedLeaderEpoch", "type": "int32", "versions": "8+", "default": "-1",
            "ignorable": true, "about": "The leader epoch." },
          { "name": "Metadata", "type": "string", "versions": "8+", "nullableVersions": "8+",
            "about": "The partition metadata." },
          { "name": "ErrorCode", "type": "int16", "versions": "8+",
            "about": "The partition-level error code, or 0 if there was no error." }
        ]}
      ]},
      { "name": "ErrorCode", "type": "int16", "versions": "8+", "default": "0",
        "about": "The group-level error code, or 0 if there was no error." }
    ]}
  ]
}
```

Response Handling

Upon receiving the STALE_MEMBER_EPOCH error, the consumer retries when receiving its next heartbeat response with its member epoch.

DescribeConfigs API

Request Schema

The schema is the same but the ResourceType field can be set to GROUP (16).

Required ACL

- Describe Config on the group.

Request Validation

No changes.

Request Handling

The new GROUP resource type is handled.

Response Schema

No changes.

Response Handling

No changes.

AlterIncrementalConfigs API

The API is the same but supports a new resource type: GROUP (16). When GROUP is used, the resource name corresponds to the group id.

Request Schema

The schema is the same but the ResourceType field can be set to GROUP (16).

Required ACL

- Alter Config on the group.

Request Validation

No changes.

Request Handling

The new GROUP resource type is handled.

Response Schema

No changes.

Response Handling

No changes.

Records

This section describes the new record types required for the new protocol. The storage layout is based on the data model described earlier in this document.

As explained earlier, they will be persisted in the __consumer_offsets compacted topic. The compacted topic based storage requires a dedicated key type per record type in order for the compaction to work. The current protocol already uses versions from 0 to 2 (included) for the keys.

Group Metadata

Groups can be rather large so we propose to use several records to store a group in order to not be limited by the maximum batch size (1MB by default). Therefore we propose to store group metadata with two records types: the ConsumerGroupMetadata and the ConsumerGroupMemberMetadata.

A group with X members will be stored with X+2 records. One ConsumerGroupMemberMetadata per member, one ConsumerGroupPartitionMetadata, and one ConsumerGroupMetadata for the group at the end. Atomicity is not a concern here. All the records can be applied independently.

Moreover, the whole group does not necessarily have to be written for every epoch. Members who have not changed could be omitted as the compacted topic will retain their previous state anyway.

When a member is deleted, a tombstone for it is written to the partition.

ConsumerGroupMetadataKey

```
{
    "type": "data",
    "name": "ConsumerGroupMetadataKey",
    "validVersions": "3",
    "flexibleVersions": "none",
    "fields": [
              { "name": "GroupId", "type": "string", "versions": "3" }
    ]
}
```

ConsumerGroupMetadataValue

```
{
    "type": "data",
    "name": "ConsumerGroupMetadataValue",
    "validVersions": "0",
    "flexibleVersions": "0+",
    "fields": [
        { "name": "Epoch", "versions": "0+", "type": "int32" }
    ],
}
```

Note that the Epoch is always the latest epoch of the group.

ConsumerGroupPartitionMetadataKey

```
{
    "type": "data",
    "name": "ConsumerGroupPartitionMetadataKey",
    "validVersions": "4",
    "flexibleVersions": "none",
    "fields": [
              { "name": "GroupId", "type": "string", "versions": "4" }
    ]
}
```

ConsumerGroupPartitionMetadataValue

```
{
    "type": "data",
    "name": "ConsumerGroupPartitionMetadataValue",
    "validVersions": "0",
    "flexibleVersions": "0+",
    "fields": [
        { "name": "Epoch", "versions": "0+", "type": "int32" },
        { "name": "Topics", "versions": "0+",
          "type": "[]TopicMetadata", "fields": [
            { "name": "TopicId", "versions": "0+", "type": "uuid" },
            { "name": "NumPartitions", "versions": "0+", "type": "int32" }
          ]}
    ],
}
```

Note that the Epoch is always the latest epoch of the group.

ConsumerGroupMemberMetadataKey

```
{
    "type": "data",
    "name": "ConsumerGroupMemberMetadataKey",
    "validVersions": "5",
    "flexibleVersions": "none",
    "fields": [
        { "name": "GroupId", "type": "string", "versions": "5" },
        { "name": "MemberId", "type": "string", "versions": "5" }
    ]
}
```

ConsumerGroupMemberMetadataValue

```
{
    "type": "data",
    "name": "ConsumerGroupMemberMetadataValue",
    "validVersions": "0",
    "flexibleVersions": "0+",
    "fields": [
        { "name": "GroupEpoch", "versions": "0+", "type": "int32" },
        { "name": "InstanceId", "versions": "0+", "nullableVersions": "0+", "type": "string" },
        { "name": "RackId", "versions": "0+", "nullableVersions": "0+", "type": "string" },
        { "name": "ClientId", "versions": "0+", "type": "string" },
        { "name": "ClientHost", "versions": "0+", "type": "string" },
        { "name": "SubscribedTopicNames", "versions": "0+", "type": "[]string" },
        { "name": "SubscribedTopicRegex", "versions": "0+", "type": "string" },
        { "name": "Assignors", "versions": "0+",
          "type": "[]Assignor", "fields": [
            { "name": "Name", "versions": "0+", "type": "string" },
            { "name": "MinimumVersion", "versions": "0+", "type": "int16" },
            { "name": "MaximumVersion", "versions": "0+", "type": "int16" },
            { "name": "Reason", "versions": "0+", "type": "int8" },
```

```
                        { "name": "Version", "versions": "0+", "type": "int16" },
            { "name": "Metadata", "versions": "0+", "type": "bytes" }
          ]}
    ],
}
```

Note that the GroupEpoch is always the latest epoch of the group.

Target Assignment

The target assignment is stored in N + 1 records where N is the number of members in the group. The records for the members are written first and followed by the assignment metadata. When a new assignment is computed, the group coordinator will compare it with the current assignment and only write the difference between the two assignments to the __consumer_offsets partition. The assignment must be atomic so the group coordinator will ensure that all the records are written in a single batch. This limit the size of the batch to 1MB (the default value). Given the incremental nature of the protocol, 1MB should be sufficient in most case here.

ConsumerGroupTargetAssignmentMetadataKey

```
{
    "type": "data",
    "name": "ConsumerGroupTargetAssignmentMetadataKey",
    "validVersions": "6",
    "flexibleVersions": "none",
    "fields": [
              { "name": "GroupId", "type": "string", "versions": "6" }
    ]
}
```

ConsumerGroupTargetAssignmentMetadataValue

```
{
    "type": "data",
    "name": "ConsumerGroupTargetAssignmentMetadataValue",
    "validVersions": "0",
    "flexibleVersions": "0+",
    "fields": [
        { "name": "AssignmentEpoch", "versions": "0+", "type": "int32" },
    ]
}
```

The AssignmentEpoch corresponds to the group epoch used to compute the assignment. It is not necessarily the most recent group epoch because the assignment is computed asynchronously when a client-side assignor is used. When a client side assignor is used, the assignment is computed asynchronously. While it is computed for the group at epoch X, the group may have already advanced to epoch X+1 due to another event (e.g. new member joined). In this case, we have chosen to install the assignment computed for epoch X and to trigger a new assignment computation right away.

ConsumerGroupTargetAssignmentMemberKey

```
{
    "type": "data",
    "name": "ConsumerGroupTargetAssignmentMemberKey",
    "validVersions": "7",
    "flexibleVersions": "none",
    "fields": [
              { "name": "GroupId", "type": "string", "versions": "7" },
        { "name": "MemberId", "type": "string", "versions": "7" }
     ]
}
```

ConsumerGroupTargetAssignmentMemberValue

```
{
    "type": "data",
    "name": "ConsumerGroupTargetAssignmentMemberValue",
    "validVersions": "0",
    "flexibleVersions": "0+",
```

```
    "fields": [
        { "name": "Error", "versions": "0+", "type": "int8" },
        { "name": "TopicPartitions", "versions": "0+",
                 "type": "[]TopicPartition", "fields": [
                { "name": "TopicId", "versions": "0+", "type": "uuid" },
          { "name": "Partitions", "versions": "0+", "type": "[]int32" }
        ]},
        { "name": "MetadataVersion", "versions": "0+", "type": "int16" },
        { "name": "MetadataBytes", "versions": "0+", "type": "bytes" }
    ]
}
```

Current Member Assignment

The current member assignment represents, as the name suggests, the current assignment of a given member.

When a member is deleted from the group, a tombstone for it is written to the partition.

ConsumerGroupCurrentMemberAssignmentKey

```
{
    "type": "data",
    "name": "ConsumerGroupCurrentMemberAssignmentKey",
    "validVersions": "8",
    "flexibleVersions": "none",
    "fields": [
              { "name": "GroupId", "type": "string", "versions": "8" },
              { "name": "MemberId", "type": "string", "versions": "8" },
    ]
}
```

ConsumerGroupCurrentMemberAssignmentValue

```
{
    "type": "data",
    "name": "ConsumerGroupCurrentMemberAssignmentValue",
    "validVersions": "0",
    "flexibleVersions": "0+",
    "fields": [
        { "name": "MemberEpoch", "versions": "0+", "type": "int32" },
                { "name": "Error", "versions": "0+", "type": "int8" },
        { "name": "TopicPartitions", "versions": "0+",
          "type": "[]TopicPartition", "fields": [
            { "name": "TopicId", "versions": "0+", "type": "uuid" },
            { "name": "Partitions", "versions": "0+", "type": "[]int32" }
        ]},
        { "name": "MetadataVersion", "versions": "0+", "type": "int16" },
        { "name": "MetadataBytes", "versions": "0+", "type": "bytes" }
    ],
}
```

Offsets

OffsetCommitValue

```
{
  "type": "data",
  "name": "OffsetCommitValue",
  "validVersions": "0-4",
  "flexibleVersions": "4+",
  "fields": [
    { "name": "offset", "type": "int64", "versions": "0+" },
    { "name": "leaderEpoch", "type": "int32", "versions": "3+", "default": -1, "ignorable": true },
    { "name": "metadata", "type": "string", "versions": "0+" },
    { "name": "commitTimestamp", "type": "int64", "versions": "0+" },
```

```
    { "name": "expireTimestamp", "type": "int64", "versions": "1", "default": -1, "ignorable": true },
    // Adds TopicId field.
    { "name": "topicId", "type": "uuid", "versions": "4", "ignorable": true }
  ]
}
```

Broker API

The new PartitionAssignor interface will be introduced on the server side. Two implementations will be provided out of the box: RangeAssignor (range) and UniformAssignor (uniform).

```
package org.apache.kafka.common.errors;

public class PartitionAssignorException extends ApiException {

    public PartitionAssignorException(String message) {
        super(message);
    }

    public PartitionAssignorException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

```
package org.apache.kafka.server.group.consumer;

public interface PartitionAssignor {

    class AssignmentSpec {
        /**
         * The members keyed by member id.
         */
        Map<String, AssignmentMemberSpec> members;

        /**
         * The topics' metadata keyed by topic id
         */
        Map<Uuid, AssignmentTopicMetadata> topics;
    }

    class AssignmentMemberSpec {
        /**
         * The instance ID if provided.
         */
        Optional<String> instanceId;

        /**
         * The rack ID if provided.
         */
        Optional<String> rackId;

        /**
         * The topics that the member is subscribed to.
         */
        Collection<String> subscribedTopics;

        /**
         * The current target partitions of the member.
         */
        Collection<TopicPartition> targetPartitions;
    }

    class AssignmentTopicMetadata {
        /**
         * The topic name.
         */
        String topicName;
```

```
        /**
                 * The number of partitions.
                 */
                int numPartitions;
    }

    class GroupAssignment {
        /**
         * The member assignments keyed by member id.
         */
        Map<String, MemberAssignment> members;
    }

    class MemberAssignment {
        /**
         * The target partitions assigned to this member.
         */
        Collection<TopicPartition> targetPartitions;
    }

    /**
     * Unique name for this assignor.
     */
    String name();

    /**
     * Perform the group assignment given the current members and
     * topic metadata.
     *
     * @param assignmentSpec The assignment spec.
     * @return The new assignment for the group.
     */
    GroupAssignment assign(AssignmentSpec assignmentSpec) throws PartitionAssignorException;
}
```

Broker Metrics

Existing generic group metrics have been migrated, with the same metric names except for

NumGroups which reported the number of generic groups. This metric changed to

kafka.server:type=group-coordinator-metrics,name=group-count,protocol={consumer|generic}

- number of groups based on type where type is the rebalance protocol

kafka.server:type=group-coordinator-metrics,name=partition-count,state={loading|active|failed}

- number of __consumer_offsets partitions based on state

kafka.server:type=group-coordinator-metrics,name=event-queue-size

- event accumulator queue size

kafka.server:type=group-coordinator-metrics,name=consumer-group-count,state={empty|assigning|reconciling|stable|dead}

- number of consumer groups based on state

consumer group rebalances sensor

- kafka.server:type=group-coordinator-metrics,name=consumer-group-rebalance-rate
- kafka.server:type=group-coordinator-metrics,name=consumer-group-rebalance-count

partition load sensor: __consumer_offsets partition load time

- kafka.server:type=group-coordinator-metrics,name=partition-load-time-max
- kafka.server:type=group-coordinator-metrics,name=partition-load-time-avg

thread idle ratio sensor: thread busy - idle ratio

- kafka.server:type=group-coordinator-metrics,name=thread-idle-ratio-min
- kafka.server:type=group-coordinator-metrics,name=thread-idle-ratio-avg

Broker Configurations

New properties in the broker configuration.

Name	Type	Default	Doc
group.coordinator.threads	int	1	The number of threads used to run the state machines.
group.consumer.session.timeout.ms	int	45s	The timeout to detect client failures when using the consumer group protocol.
group.consumer.min.session.timeout.ms	int	45s	The minimum session timeout.
group.consumer.max.session.timeout.ms	int	60s	The maximum session timeout.
group.consumer.heartbeat.interval.ms	int	5s	The heartbeat interval given to the members.
group.consumer.min.heartbeat.interval.ms	int	5s	The minimum heartbeat interval.
group.consumer.max.heartbeat.interval.ms	int	15s	The maximum heartbeat interval.
group.consumer.max.size	int	MaxValue	The maximum number of consumers that a single consumer group can accommodate.
group.consumer.assignors	list	org.apache.kafka.server.group.consumer.UniformAssignor, org.apache.kafka.server.group.consumer.RangeAssignor	The server side assignors as a list of full class names. The first one in the list is considered as the default assignor to be used in the case where the consumer does not specify an assignor.

Group Configurations

New dynamic group properties.

Name	Type	Default	Doc
group.consumer.session.timeout.ms	int	45s	The timeout to detect client failures when using the consumer group protocol.
group.consumer.heartbeat.interval.ms	int	5s	The heartbeat interval given to the members.

Consumer API

New PartitionAssignor interface

The new PartitionAssignor interface will be introduced to replace the ConsumerPartitionAssignor interface. The interface is defined as follow:

```
package org.apache.kafka.clients.consumer;

public interface PartitionAssignor {

    class Metadata {
        /**
         * The metadata version.
         */
        int version;

        /**
         * The metadata bytes.
         */
        ByteBuffer bytes;
    }

    class AssignmentSpec {
        /**
         * The members keyed by member id.
```

```
     */
    Map<String, AssignmentMemberSpec> members;

    /**
     * The topics' metadata keyed by topic id
     */
    Map<Uuid, AssignmentTopicMetadata> topics;
}

class AssignmentMemberSpec {
    /**
     * The instance ID if provided.
     */
    Optional<String> instanceId;

    /**
     * The rack ID if provided.
     */
    Optional<String> rackId;

    /**
     * The topics that the member is subscribed to.
     */
    Collection<String> subscribedTopics;

              /**
             * The reason reported by the member.
             */
            byte reason;

            /**
             * The metadata reported by the member.
             */
    Metadata metadata;

    /**
     * The current target partitions of the member.
     */
    Collection<TopicPartition> targetPartitions;
}

class AssignmentTopicMetadata {
    /**
     * The topic name.
     */
    String topicName;

    /**
             * The number of partitions.
             */
            int numPartitions;
}

class GroupAssignment {
    /**
     * The assignment error.
     */
            byte error;

    /**
     * The member assignments keyed by member id.
     */
    Map<String, MemberAssignment> members;
}

class MemberAssignment {
    /**
     * The target partitions assigned to this member.
     */
    Collection<TopicPartition> targetPartitions;
```

```
                /**
                 * The metadata.
                 */
                Metadata metadata;
    }

        class AssignorMetadata {
                  /**
                 * The reason reported by the assignor.
                 */
                byte reason;

                 /**
                 * The metadata reported by the assignor.
                 */
                Metadata metadata;
    }

    /**
     * Unique name for this assignor.
     */
    String name();

    /**
     * The minimum version.
     */
    int minimumVersion();

    /**
     * The maximum version.
     */
    int maximumVersion();

    /**
     * Return assignor metadata that will be sent to the assignor.
     */
    AssignorMetadata metadata();

    /**
     * Perform the group assignment given the current members and
     * topic metadata.
     *
     * @param assignmentSpec The assignment spec.
     * @return The new assignment for the group.
     */
    GroupAssignment assign(AssignmentSpec assignmentSpec);

    /**
     * Callback which is invoked when the member received a new assignment
     * from the assignor/group coordinator. This is called once per epoch
     * and contains the target partitions for this members. This means that
     * partitions may not be assigned to the member yet. The rebalance
     * listener must be used to know this.
     *
     * @param byte The error reported by the assignor.
     * @param assignment The assignment computed by the assignor.
     * @param consumerGroupMetadata The group metadata.
     */
    void onAssignment(byte error, MemberAssignment assignment, ConsumerGroupMetadata consumerGroupMetadata);
}
```

New SubscriptionPattern class

We need to differentiate Google RE2/J regular expression from the java.util.regex.Pattern in our public APIs so we propose to introduce the SubscriptionPattern class for this purpose. This class is just a POJO as all the validation is on the server side.

```
package org.apache.kafka.clients.consumer;

/**
```

```
 * Represents a regular expression used to subscribe to topics. The pattern
 * must be a Google RE2/J compatible pattern.
 */
public class SubscriptionPattern {
    final private String pattern;

    public Pattern(final pattern) {
        this.pattern = pattern;
    }

    public String pattern() {
        return this.pattern;
    }
}
```

New Consumer methods

We introduce two new methods to subscribe with a SubscriptionPattern.

```
public interface Consumer<K, V> extends Closeable {
    ...

    /**
     * @see KafkaConsumer#subscribe(Pattern, ConsumerRebalanceListener)
     */
    void subscribe(SubscriptionPattern pattern, ConsumerRebalanceListener callback);

    /**
     * @see KafkaConsumer#subscribe(Pattern)
     */
    void subscribe(SubscriptionPattern pattern);
}
```

Deprecate Consumer methods

The following methods will be deprecated:

- Consumer#enforeRebalance
- Consumer#enforeRebalance(String)
- Consumer#subscribe(Pattern)
- Consumer#subscribe(Pattern, ConsumerRebalanceListener)

Deprecate ConsumerPartitionAssignor interface

The ConsumerPartitionAssignor interface will be deprecated in a future (major) release.

Deprecate Consumer configurations

The following configurations will be deprecated:

- partition.assignment.strategy
- session.timeout.ms
- heartbeat.interval.ms

Consumer Configurations

Name	Type	Default	Doc
group.protocol	enum	generic	A flag which indicates if the new protocol should be used or not. It could be: generic or consumer
group.remote.assignor	string	null	The server side assignor to use. It cannot be used in conjunction with group.local.assignor. `null` means that the choice of the assignor is left to the group coordinator.
group.local.assignors	list	empty	The list of client side (local) assignors as a list of full class names. It cannot be used in conjunction with group.remote.assignor.

Streams Member Metadata and Assignment Metadata

The changes here are mainly informative at this stage. They show how we could structure the Streams' metadata. We may decide to leverage this change to do more changes.

Member Metadata Schema

This is the schema of the metadata advertised by each member.

Name	Type	Doc
ProcessId	uuid / static	Identity of the instance that may have multiple consumers.
UserEndPoint	bytes / static	Used for cross-client communication.
ClientTags	map / static	Used for rack-aware assignment algorithm.
TopologyHash	uuid / dynamic	Only updatable when `reason` is not zero.
TaskLag	array / dynamic	Only updatable when `reason` is not zero.

Member Metadata Reasons

- None (0)
- Shutdown (1)
- WarmUpReady (2)
- WarmUpFailed (3)
- TopologyChanged (4)

Assignment Metadata Schema

Name	Type	Doc
ActiveTasks	list	Local assignment for this consumer.
StandbyTasks	map	Local standby tasks for this consumer.
WarmupTasks	map	Local warming up tasks for this consumer.
PartitionsByHost	map	Global assignment information used for IQ.

Assignment Metadata Errors

- None (0)
- Shutdown (1)
- AssignmentError (2)
- InconsistentTopology (3)

Streams API

New Topology methods

All the Topology#addSource methods using java.util.regex.Pattern will get a corresponding overload using SubscriptionPattern.

Deprecated methods

The following methods will be deprecated:

- Topology#addSource(String, Pattern)
- Topology#addSource(AutoOffsetReset, String, Pattern)
- Topology#addSource(TimestampExtractor, String, Pattern)
- Topology#addSource(AutoOffsetReset, TimestampExtractor, String, Pattern)
- Topology#addSource(String, Deserializer, Deserializer, Pattern)
- Topology#addSource(AutoOffsetReset, String, Deserializer, Deserializer, Pattern)
- Topology#addSource(AutoOffsetReset, String, TimestampExtractor, Deserializer, Deserializer, Pattern)

Streams Configurations

Name	Type	Default	Doc
group.protocol	enum	generic	A flag which indicates if the new protocol should be used or not. It could be: generic or consumer

Admin API

Admin#listConsumerGroups

The Admin#listConsumerGroups will be extended to support querying group types and retrieving/querying the new group states.

```
public class ListConsumerGroupsOptions extends AbstractOptions<ListConsumerGroupsOptions> {

    /**
     * If types is set, only groups with these types will be returned.
     */
    public ListConsumerGroupsOptions withTypes(Set<String> types) {
        this.types = types;
    }

    /**
     * Returns the list of Types that are requested or empty if no types
     * have been specified.
     */
    public Set<String> types() {
        return types;
    }
}

public class ConsumerGroupListing {

    /**
     * Consumer Group type, generic by default.
     */
    public String type() {
        return type;
    }
}

public enum ConsumerGroupState {
    UNKNOWN("Unknown"),
    PREPARING_REBALANCE("PreparingRebalance"),
    COMPLETING_REBALANCE("CompletingRebalance"),
    STABLE("Stable"),
    DEAD("Dead"),
    EMPTY("Empty"),
    ASSIGNING("Assigning"),
    RECONCILING("Reconciling");
}
```

Admin#describeConsumerGroups

The Admin#describeConsumerGroups will be extended to expose the new information related to the new protocol.

```
public class ConsumerGroupDescription {
    public String type() {
      return type;
    }
}

public class MemberDescription {
    /**
     * The current assignment of the member. Provided for both generic group and consumer group.
     */
    public MemberAssignment assignment() {}

    /**
     * The target assignment of the member. Provided only for consumer group.
     */
    public Optional<MemberAssignment> targetAssignment() {}
}
```

```
class Metadata {
    /**
     * The metadata version.
     */
        int version;

    /**
     * The metadata bytes.
     */
        ByteBuffer bytes;
}

public class MemberAssignment {
        /**
     * The partitions.
     */
    Set<TopicPartition> topicPartitions;

         /**
         * The error reported by the assignor. Provided only if the group is a ConsumerGroup type.
         */
        byte error;

    /**
     * The assigned metadata. Provided only if the group is ConsumerGroup type.
     */
    Optional<Metadata> metadata;
}
```

Admin#incrementalAlterConfigs and Admin#describeConfigs

The GROUP resource type is added.

```
public final class ConfigResource {
    /**
     * Type of resource.
     */
    public enum Type {
         GROUP((byte) 16), BROKER_LOGGER((byte) 8), BROKER((byte) 4), TOPIC((byte) 2), UNKNOWN((byte) 0);
    }
}
```

kafka-consumer-groups

--type

The kafka-consumer-group command line tool will be extended to support the –type filter which allows to list or to describe groups implementing a specific type.

```
kafka-consumer-groups.sh --bootstrap-server localhost:9092 --list --type <comma separated list of types>

kafka-consumer-groups.sh --bootstrap-server localhost:9092 --describe --type <comma separated list of types>
```

--validate-regex

The kafka-consumer-group command line tool will be extended to support the –validate-regex which allows to verify the regular expression use by a group to subscribe to topics.

```
kafka-consumer-groups.sh --bootstrap-server localhost:9092 --validate-regex <pattern>
```

Case Studies

All the examples shown in this chapter are based on the in-memory representation of the group coordinator.

Basic

Let's look at a few examples to illustrate the rebalance logic. Let's assume that the group is subscribed to the topic foo which has 3 partitions.

Let's start with an empty group:

- Group (epoch=0)
  - Empty
- Target Assignment (epoch=0)
  - Empty
- Member Assignment
  - Empty

Member A joins the group. The coordinator bumps the group epoch to 1, adds A to the group, and creates an empty member assignment.

- Group (epoch=1)
  - A
- Target Assignment (epoch=0)
  - Empty
- Member Assignment
  - A - epoch=0, partitions=[], pending-partitions=[]

The coordinator computes and installs the new target assignment. All the partitions are assigned to A.

- Group (epoch=1)
  - A
- Target Assignment (epoch=1)
  - A - partitions=[foo-0, foo-1, foo-2]
- Member Assignment
  - A - epoch=0, partitions=[], pending-partitions=[]

When A heartbeats, the group coordinator transitions it to its target epoch/assignment because it does not have any partitions to revoke. The group coordinator updates the member assignment and replies with the new epoch 1 and all the partitions.

- Group (epoch=1)
  - A
- Target Assignment (epoch=1)
  - A - partitions=[foo-0, foo-1, foo-2]
- Member Assignment
  - A - epoch=1, partitions=[foo-0, foo-1, foo-2], pending-partitions=[]

Member B joins the group. The coordinator adds the member to the group and bumps the group epoch to 2.

- Group (epoch=2)
  - A
  - B
- Target Assignment (epoch=1)
  - A - partitions=[foo-0, foo-1, foo-2]
- Member Assignment
  - A - epoch=1, partitions=[foo-0, foo-1, foo-2], pending-partitions=[]
  - B - epoch=0, partitions=[], pending-partitions=[]

The coordinator computes and installs the new target assignment.

- Group (epoch=2)
  - A
  - B
- Target Assignment (epoch=2)
  - A - partitions=[foo-0, foo-1]
  - B - partitions=[foo-2]
- Member Assignment
  - A - epoch=1, partitions=[foo-0, foo-1, foo-2], pending-partitions=[]
  - B - epoch=0, partitions=[], pending-partitions=[]

At this point B can transitions to epoch 2 but cannot get foo-2 until A revokes it.

- Group (epoch=2)
  - A
  - B
- Target Assignment (epoch=2)
  - A - partitions=[foo-0, foo-1]
  - B - partitions=[foo-2]
- Member Assignment
  - A - epoch=1, partitions=[foo-0, foo-1, foo-2], pending-partitions=[]
  - B - epoch=2, partitions=[], pending-partitions=[foo-2]

When A heartbeats, the group coordinator instructs it to revoke foo-2.

When A heartbeats again and acknowledges the revocation, the group coordinator transitions it to epoch 2 and releases foo-2.

- Group (epoch=2)
  - A
  - B
- Target Assignment (epoch=2)
  - A - partitions=[foo-0, foo-1]
  - B - partitions=[foo-2]
- Member Assignment
  - A - epoch=2, partitions=[foo-0, foo-1], pending-partitions=[]
  - B - epoch=2, partitions=[], pending-partitions=[foo-2]

When B heartbeats, he can now gets foo-2.

- Group (epoch=2)
  - A
  - B
- Target Assignment (epoch=2)
  - A - partitions=[foo-0, foo-1]
  - B - partitions=[foo-2]
- Member Assignment
  - A - epoch=2, partitions=[foo-0, foo-1]
  - B - epoch=2, partitions=[foo-2], pending-partitions=[]

Member C joins the group. The coordinator adds the member to the group and bumps the group epoch to 3.

- Group (epoch=3)
  - A
  - B
  - C
- Target Assignment (epoch=2)
  - A - partitions=[foo-0, foo-1]
  - B - partitions=[foo-2]
- Member Assignment
  - A - epoch=2, partitions=[foo-0, foo-1], pending-partitions=[]
  - B - epoch=2, partitions=[foo-2], pending-partitions=[]
  - C - epoch=0, partitions=[], pending-partitions=[]

The coordinator computes and installs the new target assignment.

- Group (epoch=3)
  - A
  - B
  - C
- Target Assignment (epoch=3)
  - A - partitions=[foo-0]
  - B - partitions=[foo-2]
  - C - partitions=[foo-1]
- Member Assignment
  - A - epoch=2, partitions=[foo-0, foo-1], pending-partitions=[]
  - B - epoch=2, partitions=[foo-2], pending-partitions=[]
  - C - epoch=0, partitions=[], pending-partitions=[]

When B heartbeats, the group coordinator transitions it to epoch 3 because B has no partitions to revoke. It persists the change and reply.

- Group (epoch=3)
  - A
  - B
  - C
- Target Assignment (epoch=3)
  - A - partitions=[foo-0]
  - B - partitions=[foo-2]
  - C - partitions=[foo-1]
- Member Assignment
  - A - epoch=2, partitions=[foo-0, foo-1], pending-partitions=[]
  - B - epoch=3, partitions=[foo-2], pending-partitions=[]
  - C - epoch=0, partitions=[], pending-partitions=[]

When C heartbeats, it transitions to epoch 3 but cannot get foo-1 yet.

- Group (epoch=3)
  - A
  - B
  - C
- Target Assignment (epoch=3)
  - A - partitions=[foo-0]
  - B - partitions=[foo-2]
  - C - partitions=[foo-1]
- Member Assignment

    - A - epoch=2, partitions=[foo-0, foo-1], pending-partitions=[]
    - B - epoch=3, partitions=[foo-2], pending-partitions=[]
    - C - epoch=3, partitions=[], pending-partitions=[foo-1]

When A heartbeats, the group coordinator instructs it to revoke foo-1.

When A heartbeats again and acknowledges the revocation, the group coordinator transitions it to epoch 3 and releases foo-1.

- Group (epoch=3)
    - A
    - B
    - C
- Target Assignment (epoch=3)
    - A - partitions=[foo-0]
    - B - partitions=[foo-2]
    - C - partitions=[foo-1]
- Member Assignment
    - A - epoch=3, partitions=[foo-0], pending-partitions=[]
    - B - epoch=3, partitions=[foo-2], pending-partitions=[]
    - C - epoch=3, partitions=[foo-1], pending-partitions=[]

All the members have eventually advanced to the group epoch (3).

Incremental Revocation & Assignment

Let's imagine a group with two members and six partitions.

- Group (epoch=21)
    - A
    - B
- Target Assignment (epoch=21)
    - A - partitions=[foo-0, foo-1, foo-2]
    - B - partitions=[foo-3, foo-4, foo-5]
- Member Assignment
    - A - epoch=21, partitions=[foo-0, foo-1, foo-2], pending-partitions=[]
    - B - epoch=21, partitions=[foo-3, foo-4, foo-5], pending-partitions=[]

C joins the group. The group coordinator adds it, bumps the group epoch, create the member assignment, and computes the target assignment.

- Group (epoch=22)
    - A
    - B
    - C
- Target Assignment (epoch=22)
    - A - partitions=[foo-0, foo-1]
    - B - partitions=[foo-3, foo-4]
    - C - partitions=[foo-2, foo-5]
- Member Assignment
    - A - epoch=21, partitions=[foo-0, foo-1, foo-2], pending-partitions=[]
    - B - epoch=21, partitions=[foo-3, foo-4, foo-5], pending-partitions=[]
    - C - epoch=0, partitions=[], pending-partitions=[]

C heartbeats, the group coordinator transitions it to epoch 22 but does not yet give it any partitions because they are not revoked yet.

- Group (epoch=22)
    - A
    - B
    - C
- Target Assignment (epoch=22)
    - A - partitions=[foo-0, foo-1]
    - B - partitions=[foo-3, foo-4]
    - C - partitions=[foo-2, foo-5]
- Member Assignment
    - A - epoch=21, partitions=[foo-0, foo-1, foo-2], pending-partitions=[]
    - B - epoch=21, partitions=[foo-3, foo-4, foo-5], pending-partitions=[]
    - C - epoch=22, partitions=[], pending-partitions=[foo-2, foo-5]

A heartbeats, the group coordinator instructs it to revoke foo-2.

B heartbeats, the group coordinator instructs it to revoke foo-5.

C heartbeats, no changes for it.

A heartbeats and acknowledges the revocation, the group coordinator transitions it to epoch 22, release foo-2, persists and reply.

- Group (epoch=22)
    - A
    - B
    - C

- Target Assignment (epoch=22)
  - A - partitions=[foo-0, foo-1]
  - B - partitions=[foo-3, foo-4]
  - C - partitions=[foo-2, foo-5]
- Member Assignment
  - A - epoch=22, partitions=[foo-0, foo-1], pending-partitions=[]
  - B - epoch=21, partitions=[foo-3, foo-4, foo-5], pending-partitions=[]
  - C - epoch=22, partitions=[foo-2], pending-partitions=[foo-5]

C heartbeats, the group coordinator gives it foo-2 which is now free but hold foo-5.

B heartbeats and acknowledges the revocation, the group coordinator transitions it to epoch 22, releases foo-5, persists and reply.

- Group (epoch=22)
  - A
  - B
  - C
- Target Assignment (epoch=22)
  - A - partitions=[foo-0, foo-1]
  - B - partitions=[foo-3, foo-4]
  - C - partitions=[foo-2, foo-5]
- Member Assignment
  - A - epoch=22, partitions=[foo-0, foo-1], pending-partitions=[]
  - B - epoch=22, partitions=[foo-3, foo-4], pending-partitions=[]
  - C - epoch=22, partitions=[foo-2, foo-5], pending-partitions=[]

C heartbeats, the group coordinator gives it foo-2 and foo-5.

Member Failure

Let's start with a group with three members and six partitions.

- Group (epoch=22)
  - A
  - B
  - C
- Target Assignment (epoch=22)
  - A - partitions=[foo-0, foo-1]
  - B - partitions=[foo-3, foo-4]
  - C - partitions=[foo-2, foo-5]
- Member Assignment
  - A - epoch=22, partitions=[foo-0, foo-1], pending-partitions=[]
  - B - epoch=22, partitions=[foo-3, foo-4], pending-partitions=[]
  - C - epoch=22, partitions=[foo-2, foo-5], pending-partitions=[]

A fails to heartbeat. The group coordinator removes it after the session timeout expires and bump the group epoch.

- Group (epoch=23)
  - B
  - C
- Target Assignment (epoch=22)
  - A - partitions=[foo-0, foo-1]
  - B - partitions=[foo-3, foo-4]
  - C - partitions=[foo-2, foo-5]
- Member Assignment
  - B - epoch=22, partitions=[foo-3, foo-4], pending-partitions=[]
  - C - epoch=22, partitions=[foo-2, foo-5], pending-partitions=[]

The group coordinator computes the new target assignment.

- Group (epoch=23)
  - B
  - C
- Target Assignment (epoch=23)
  - B - partitions=[foo-3, foo-4, foo-0]
  - C - partitions=[foo-2, foo-5, foo-1]
- Member Assignment
  - B - epoch=22, partitions=[foo-3, foo-4], pending-partitions=[]
  - C - epoch=22, partitions=[foo-2, foo-5], pending-partitions=[]

B and C heartbeat and transition to epoch 23.

- Group (epoch=23)
  - B
  - C
- Target Assignment (epoch=23)
  - B - partitions=[foo-3, foo-4, foo-0]
  - C - partitions=[foo-2, foo-5, foo-1]
- Member Assignment

    - B - epoch=23, partitions=[foo-3, foo-4, foo-0], pending-partitions=[]
    - C - epoch=23, partitions=[foo-2, foo-5, foo-1], pending-partitions=[]

Partition Added

Let's start with a group with two members and one partition.

- Group (epoch=22)
    - A
    - B
- Target Assignment (epoch=22)
    - A - partitions=[foo-0]
    - B - partitions=[]
- Member Assignment
    - A - epoch=22, partitions=[foo-0], pending-partitions=[]
    - B - epoch=22, partitions=[], pending-partitions=[]

A new partition foo-1 is created. The group coordinator detects it. It updates the group and bump the group epoch.

- Group (epoch=23)
    - A
    - B
- Target Assignment (epoch=22)
    - A - partitions=[foo-0]
    - B - partitions=[]
- Member Assignment
    - A - epoch=22, partitions=[foo-0], pending-partitions=[]
    - B - epoch=22, partitions=[], pending-partitions=[]

The group coordinator computes a new target assignment.

- Group (epoch=23)
    - A
    - B
- Target Assignment (epoch=23)
    - A - partitions=[foo-0]
    - B - partitions=[foo-1]
- Member Assignment
    - A - epoch=22, partitions=[foo-0], pending-partitions=[]
    - B - epoch=22, partitions=[], pending-partitions=[]

B and C heartbeat and transition to epoch 23.

- Group (epoch=23)
    - A
    - B
- Target Assignment (epoch=23)
    - A - partitions=[foo-0]
    - B - partitions=[foo-1]
- Member Assignment
    - A - epoch=23, partitions=[foo-0], pending-partitions=[]
    - B - epoch=23, partitions=[foo-1], pending-partitions=[]

Online Migration

We starts with a generic group.

- Generic Group (generation=22)
    - A
    - B
    - C
- Assignment
    - A - partitions=[foo-0, foo-1], pending-partitions=[]
    - B - partitions=[foo-3, foo-4], pending-partitions=[]
    - C - partitions=[foo-2, foo-5], pending-partitions=[]

A leaves and rejoins with the new protocol enabled. The group is converted. The current generation becomes the group epoch. The target assignment and the member assignments are created based on the current assignment.

- Group (epoch=22)
    - A (upgraded)
    - B
    - C
- Target Assignment (epoch=22)
    - A - partitions=[foo-0, foo-1]
    - B - partitions=[foo-3, foo-4]
    - C - partitions=[foo-2, foo-5]

- Member Assignment
    - A - epoch=22, partitions=[foo-0, foo-1], pending-partitions=[]
    - B - epoch=22, partitions=[foo-3, foo-4], pending-partitions=[]
    - C - epoch=22, partitions=[foo-2, foo-5], pending-partitions=[]

A uses the new protocol. B and C still use the old protocol.

B leaves the group. The group coordinator removes it and bumps the group epoch.

- Group (epoch=23)
    - A (upgraded)
    - C
- Target Assignment (epoch=22)
    - A - partitions=[foo-0, foo-1]
    - B - partitions=[foo-3, foo-4]
    - C - partitions=[foo-2, foo-5]
- Member Assignment
    - A - epoch=22, partitions=[foo-0, foo-1], pending-partitions=[]
    - C - epoch=22, partitions=[foo-2, foo-5], pending-partitions=[]

The group coordinator computes a new target assignment and installs it. It also triggers a rebalance for C.

- Group (epoch=23)
    - A (upgraded)
    - C (PreparingRebalance)
- Target Assignment (epoch=23)
    - A - partitions=[foo-0, foo-1, foo-3]
    - C - partitions=[foo-2, foo-5, foo-4]
- Member Assignment
    - A - epoch=22, partitions=[foo-0, foo-1], pending-partitions=[]
    - C - epoch=22, partitions=[foo-2, foo-5], pending-partitions=[]

C heartbeats and is notified that a rebalance is required. C revokes all its partitions (assuming Eager protocol is used here) and sends a JoinGroup request.

The group coordinator sees that C does not own any partitions any more, so it can transition to epoch 23 and transition to CompletingRebalance. The transition to epoch 23 is important here because the new epoch must be given to the member in the JoinGroup response. This is the new generation of the group for it.

- Group (epoch=23)
    - A (upgraded)
    - C (CompletingRebalance)
- Target Assignment (epoch=23)
    - A - partitions=[foo-0, foo-1, foo-3]
    - C - partitions=[foo-2, foo-5, foo-4]
- Member Assignment
    - A - epoch=22, partitions=[foo-0, foo-1], pending-partitions=[]
    - C - epoch=23, partitions=[foo-2, foo-5, foo-4], pending-partitions=[]

In the meantime, A heartbeats and transitions to epoch 23 as well.

- Group (epoch=23)
    - A (upgraded)
    - C (CompletingRebalance)
- Target Assignment (epoch=23)
    - A - partitions=[foo-0, foo-1, foo-3]
    - C - partitions=[foo-2, foo-5, foo-4]
- Member Assignment
    - A - epoch=23, partitions=[foo-0, foo-1, foo-3], pending-partitions=[]
    - C - epoch=23, partitions=[foo-2, foo-5, foo-4], pending-partitions=[]

C sends the SyncGroup request and collects its new assignment. All partitions are given because they are all free. C transitions to Stable.

- Group (epoch=23)
    - A (upgraded)
    - C (Stable)
- Target Assignment (epoch=23)
    - A - partitions=[foo-0, foo-1, foo-3]
    - C - partitions=[foo-2, foo-5, foo-4]
- Member Assignment
    - A - epoch=23, partitions=[foo-0, foo-1, foo-3], pending-partitions=[]
    - C - epoch=23, partitions=[foo-2, foo-5, foo-4], pending-partitions=[]

B rejoins the group with the new protocol. The group coordinator adds it and bumps the group epoch.

- Group (epoch=24)
    - A (upgraded)
    - B (upgraded)
    - C (Stable)
- Target Assignment (epoch=23)
    - A - partitions=[foo-0, foo-1, foo-3]

    - C - partitions=[foo-2, foo-5, foo-4]
- Member Assignment
    - A - epoch=23, partitions=[foo-0, foo-1, foo-3], pending-partitions=[]
    - B - epoch=0, partitions=[], pending-partitions=[]
    - C - epoch=23, partitions=[foo-2, foo-5, foo-4], pending-partitions=[]

The group coordinator computes a new target assignment. A rebalance is triggered for C to revoke foo-4.

- Group (epoch=24)
    - A (upgraded)
    - B (upgraded)
    - C (PreparingRebalance)
- Target Assignment (epoch=24)
    - A - partitions=[foo-0, foo-1]
    - B - partitions=[foo-3, foo-4]
    - C - partitions=[foo-2, foo-5]
- Member Assignment
    - A - epoch=23, partitions=[foo-0, foo-1, foo-3], pending-partitions=[]
    - B - epoch=0, partitions=[], pending-partitions=[foo-3, foo-4]
    - C - epoch=23, partitions=[foo-2, foo-5, foo-4], pending-partitions=[]

A heartbeats and he is told to revoke foo-3.

B heartbeats and transitions to epoch 24 but does not get any partitions yet because they are not free.

C heartbeats and he is told to rebalance. He revokes all its partitions and sends the JoinGroup request.

The group coordinator sees that C does not own any partitions any more, so it can transition to epoch 24 and transition to CompletingRebalance. foo-4 is released.

- Group (epoch=24)
    - A (upgraded)
    - B (upgraded)
    - C (CompletingRebalance)
- Target Assignment (epoch=24)
    - A - partitions=[foo-0, foo-1]
    - B - partitions=[foo-3, foo-4]
    - C - partitions=[foo-2, foo-5]
- Member Assignment
    - A - epoch=23, partitions=[foo-0, foo-1, foo-3], pending-partitions=[]
    - B - epoch=24, partitions=[foo-4], pending-partitions=[foo-3]
    - C - epoch=24, partitions=[foo-2, foo-5], pending-partitions=[]

C sends the SyncGroup request to collect its assignment. He transitions to Stable.

- Group (epoch=24)
    - A (upgraded)
    - B (upgraded)
    - C (Stable)
- Target Assignment (epoch=24)
    - A - partitions=[foo-0, foo-1]
    - B - partitions=[foo-3, foo-4]
    - C - partitions=[foo-2, foo-5]
- Member Assignment
    - A - epoch=23, partitions=[foo-0, foo-1, foo-3], pending-partitions=[]
    - B - epoch=24, partitions=[foo-4], pending-partitions=[foo-3]
    - C - epoch=24, partitions=[foo-2, foo-5], pending-partitions=[]

A heartbeats. He confirms the revocation of foo-3. He transitions to epoch 24. foo-3 is released.

- Group (epoch=24)
    - A (upgraded)
    - B (upgraded)
    - C (Stable)
- Target Assignment (epoch=24)
    - A - partitions=[foo-0, foo-1]
    - B - partitions=[foo-3, foo-4]
    - C - partitions=[foo-2, foo-5]
- Member Assignment
    - A - epoch=24, partitions=[foo-0, foo-1], pending-partitions=[]
    - B - epoch=24, partitions=[foo-3, foo-4], pending-partitions=[]
    - C - epoch=24, partitions=[foo-2, foo-5], pending-partitions=[]

B heartbeats and gets its assignment.

Compatibility, Deprecation, and Migration Plan

Kafka Broker Migration

Upgrading the cluster to the new rebalance protocol is pretty strait-forward. First, the cluster must be first upgraded to the new group coordinator. This can be done by rolling-upgrading the broker to a software version which supports it. The migration is seamless. Note that *group.coordinator.threads* may require some tuning depending on your workload. Second, the new protocol must be enabled by setting an IBP/MetadataVersion which supports it with the *kafka-features* command line tool.

Downgrading the cluster will be possible to a certain extend. It is possible to downgrade the IBP/MetadataVersion to disable the new rebalance protocol. In this case, all the consumer groups using the new protocol will be lost. It will be possible to downgrade to an earlier version of Kafka that does not support the new group coordinator. In this case, we will only support specific versions. The issue is that the current group coordinator errors out when the *__consumer_offsets* topics contains unknown records.

Kafka Consumer Migration

Upgrading consumers can be done online. First, all the consumers must be upgraded to a software version which supports the online migration. The version is not defined yet but the consumer will need to support the consumer embedded protocol version 3 (introduced in KIP-792). There are also other considerations if a custom client side assignor is used. Second, the consumer must be rolled to enable the new rebalance protocol. This is done by setting *group.protocol* to *consumer. group.remote.assignor* may need to be adjusted as well. When a consumer joins with the new protocol, the group is automatically converted from a generic group to a consumer group if the upgrade requirements are met (e.g. consumer embedded protocol version >=3). If they are not, the consumer is rejected with an INVALID_REQUEST error. During the upgrade process, consumers will continue to use the old rebalance APIs. The group coordinator will translate the JoinGroup, SyncGroup and Heartbeat API to the new rebalance protocol. This translation is an implementation detail but it is explained earlier in this document, see the Supporting Online Consumer Group Upgrade chapter. If a customer client side assignor or regex based subscriptions are used, please pay attention to the details provided in the following sub-chapters.

Downgrading the consumers is the exact opposite process. The consumer must be rolled to disable the new rebalance protocol. This is done by setting the the *group.protocol* to *generic*. When the last consumer using the new rebalance protocol leaves the group, the group is automatically converted down from a consumer group to a generic group.

Regex Based Subscriptions

The new rebalance protocol relies on the group coordinator to track the metadata changes so the regular expressions is not used locally anymore but remotely. The group coordinator uses the Google RE2/J engine so the regular expression used with either of the methods must be compatible. Usage of methods subscribing to topics using java.util.regex.Pattern should be replaced by their homolog using SubscriptionPattern. By default, the java.util.regex.Pattern used to subscribe is *toString*'ed and passed to the group coordinator so simple regular expressions should keep working without any changes. If it is not compatible, the group coordinator will reject the *ConsumerGroupHeartbeat* request with an INVALID_REQUEST error. For simple regular expressions, we don't expect any changes to be required. It is recommended to test out the regex with the *consumer-groups --verify-regex* command line tool or with another group before migrating consumers.

Client Side Assignor

ConsumerPartitionAssignor cannot be used with the new rebalance protocol. Instead, the client side assignor must implement *PartitionAssignor* and configure *group.local.assignors*. The Javadoc of *PartitionAssignor* in the Public Interfaces section of this document explains how it works. During the migration, only the *PartitionAssignor* based assignor is used to compute the assignment for the group. That means that the new assignor must be able to deserialize the subscription metadata used by the old assignor and the serialize assignment metadata for the old members. The group coordinator cannot translate the metadata that it receives. However, it will signal them by using -1 as the version for it. This means that the new assignor must support version from -1 to X during the online migration. If a consumer using the new protocol joins a group with non-empty metadata, the group coordinator will ensure that the joining member's assignor supports -1 in its version range. It it does not, the group coordinator will reject it with an INVALID_REQUEST error.

Kafka Streams Migration

Upgrading streams instances can be done online. First, all the instances must be upgraded to a software version which supports the online migration. This version will be driven by the underlying consumer requirements. Second, the instances must be rolled to enable the new rebalance protocol. This is done by setting *group.protocol* to *consumer.*

Downgrading the instances is the exact opposite process. The instances must be rolled to disable the new rebalance protocol. This is done by setting the the *group.protocol* to *generic*.

Test Plan

Our primary method for testing the implementation will be through Discrete Event Simulation (DES). DES allows us to test a large number of deterministically generated random scenarios which include various kinds of faults (such as network partitions). It allows us to define system invariants programmatically which are then checked after each step in the simulation. The protocol will be formally verified with a TLA+ model as well. Other than that, we will use the typical suite of unit/integration/system tests. System tests will be parameterised to run with both protocols.

Rejected Alternatives

An epoch per partition

We started this design by using an epoch per partition instead of relying on an epoch per member. Moving a partition from A to B would have require the following step: 1) revoke the partition from A; 2) bump the partition epoch; and 3) assign the partition to B. While this was very appealing at first, it was

unpractical in the end for two reasons: 1) migrating from the current protocol is much more difficult without a member id; and 2) the metadata associated to the assignment (e.g. Streams metadata) is not tight to a particular partitions. We ended up using a member epoch with an incremental reassignment algorithm which is pretty close to this.

An epoch per member not aligned to the group epoch

The current design ensures that each member epoch eventually converged to the group epoch. In a previous iteration of the design, we considered incrementing member epoch only when the member's assignment changed. The benefit of this is that it reduce the number of writes to the __consumer_offsets partitions. The downside is that it is harder to debug/understand for users and operators because they cannot rely on the member epoch to know if the member has converged to the desired assignment. In the end, we decided to favour the debuggability.

Not reusing the current coordinator

We considered not reusing the current group coordinator. Instead, the idea was to implement a brand new consumer group coordinator dedicated for the new rebalance protocol. The main benefits of this is that we could have moved away from the __consumer_offsets storage and use something more appropriate, perhaps closer to the KRaft metadata topic. This was rejected because migrating from a generic group to a consumer group would have been much more difficult.

No more client-side assignors, even for Kafka Streams

We considered removing the client-side assignor feature. From a consumer perspective, this is rarely used nowadays. Kafka Streams is its primary user so we thought about using a server side assignor in this case as well. We abandoned this for two reasons: 1) the Streams' assignor needs to know the entire Streams's topology so each member would have had to send it out to the server. The topology could be rather big (in MB) so this is not very practical; and 2) That would have introduced a strong dependency between the server version and the Streams version. Using new features in Streams would not be possible without upgrading the servers first.

Storing dynamic group configuration in the Group Coordinator

We considered storing the dynamic group configurations in the group coordinator in order to have the ability to tight their lifecycles to their group. We discarded this approach for two reasons: 1) This pattern does not fit very well in the IncrementalAlterConfig API as it would require to send updates about groups to the coordinator whereas all the other updates go to the controller; and 2) It seems preferable to decouple the life cycle of the dynamic configurations from the life cycle of the groups. Users may want to create configurations before the group is created and users may want to keep their configurations if the group is recreated.

Client side generated Member ID

We considered letting the client generate its member ID (or UUID) instead of relying on the coordinator to generate one when the member joins the group. We ultimately rejected this because 1) it introduces extra dependencies on the client. For instance in C++, UUID are not natively supported so an extra library must be used; and 2) the client could not generate it correctly.

Future Work

Eventually, we aim at deprecating the current membership/rebalance API. In order to get to this point, we would need to first move all the use cases away from it.

Connect Rebalance Protocol

Kafka Connect is the second protocol type which is currently supported by Apache Kafka. We propose to use a similar approach that the one used by the current proposal for Connect in the future. We would introduce a new connect group type and introduce a new set of APIs for Connect. The rebalance protocol is very similar to the consumer rebalance protocol but works with different resource types.

Membership/Leader Election API

The group membership protocol is also used outside of Apache Kafka. For instance, the Confluent Schema Registry uses it for leader election. It is not clear whether we really want to suppose such cases in the future. If we do, we could also define a new set of APIs for it. That would be much cleaner in the long run.

Metadata Transactions

The KIP proposes the rely on the atomicity of the batch to write the assignment to the __consumer_offsets partitions. This means that the assignment or, to be precise, the delta between two assignments can not be larger than 1MB where 1MB is the default batch size. In the future, we could imagine doing something similar to KIP-868 Metadata Transactions in the group coordinator. The solution outlined in KIP-868 does not work in our context because the __consumer_offsets is compacted. However, we could imagine a similar approach. We will tackle this in the future if needed.

Upgrade / Downgrade

The KIP proposes to rely on the IBP/MetadataVersion to decide whether a record or an API could be used or not. We have discussed the idea to use a dedicate feature flag instead of relying on metadata.version. That would allow decoupling the group coordinator from the quorum controller during upgrades. We also need to flush out how to handle downgrades. We will do this in a future KIP.