[DRAFT]Integrating Kafka Connect With New Consumer Rebalance Protocol

Background

KIP-848 introduces a new consumer rebalance protocol. It is a paradigm shift from how Consumer Rebalances are performed. In terms of what changes, these are the high level items:

1. Move away the rebalancing logic from the clients to brokers. This is the major point and the KIP explains in detail about the issue with thick clients.
2. Move away from the global synchronization barrier that the current protocol has to be truly incremental and co-operative.
3. One big proposed change is the re-write of Group Coordinator from scratch in Java.
4. Lastly, it introduces new APIs and tries to piggyback on the heartbeat APIs to transfer membership details. Currently, the heartbeat are just used to liveness but the KIP feels it would be good to enhance it.
5. 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.

Kafka Connect

Kafka Connect also has a concept of rebalances but the unit of rebalance is different. In Consumers and inturn Streams/ksqldb, the unit of rebalance is topic-partitions. However, in the case of Kafka Connect the rebalances happen within a bunch of connect workers running in a cluster and what gets “rebalanced” are connectors and tasks. Let’s take a look at few scenarios where a rebalance might be triggered in Kafka Connect:

- A new worker is added to the cluster.
- An existing worker bounces.
- A worker leaves the group permanently.

There’s also a concept of Assignors in Kafka Connect which is performed on the Worker side. Currently, 2 types of assignors are supported: Eager and Incremental Cooperative. The KIP also provides the provision of running Client Side assignors since Kafka Streams relies on Client Side assignors.

Changes needed in the new Rebalance Protocol

While there are similarities in Kafka Streams and Connect in terms of how they can be onboarded onto the new rebalance protocol. the rebalance protocol would need to be updated to be able to understand Kafka Connect semantics. This is because the new protocol is tailored around Topics/Partitions while Kafka Connect doesn’t deal with those. Here are some of the updates :

Connect group type

The new protocol introduces a concept of `types` within the group coordinator. This is basically to allow supporting different kinds of groups in the future and also to be able to differentiate between old and new consumer groups. Keeping the former reason in mind, we would introduce a new `type` called `connect` .

Data Model

The KIP introduces a logical data model to capture group membership and their assignments which the group coordinator would use for bookkeeping purposes. There are separate public interfaces which are defined to capture actual requests/responses between brokers <=> group-coordinator. This is again consumer group centric which we would try to leverage and expand for Kafka Connect.

Group Protocol

The new protocol introduces a new config called `group.protocol` . The older protocol is called `generic` while the new one is called `consumer` . To maintain clear separation, we would introduce a new possible value for `group.protocol` called `connect` .

Worker Group and Worker

Worker Group		
Name	**Type**	**Description**
Group ID	string	The group ID as configured on the worker.The ID uniquely identifies the worker cluster.
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.
Workers	[] Worker	The set of worker in the group.

Worker

Name	Type	Description
Member ID	**String**	**Unique Member ID**
url	string	The url of the worker.
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. (For Connect, currently it's between 0-2 i.e EAGER, COMPATIBLE and SESSIONED.).
Metadata	bytes	The metadata provided by the worker for this assignor.

Kafka Connect supports Client side Assignors (EAGER and Incremental Cooperative) and that's what is embedded here. Note that the KIP proposes topic subscriptions(literal and regex based) as part of the data model but we won't need it here. That's because a worker can start even without any connectors /tasks running on it while that's not the case for consumer groups. Also, we have reused the Assignor data model.

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 on the worker.The ID uniquely identifies the worker cluster.
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.
Workers	[]Worker	The assignment for each worker.
Worker		
Name	**Type**	**Description**
MemberID	**String**	**Unique memberID**
url	string	The url of the worker.
ConnectorIds	[]String	The set of connectors assigned to this worker.
Tasks	[] ConnectorTaskId	The tasks assigned to the worker

ConnectorTaskId		
Name	**Type**	**Description**
ConnectorId	String	Unique connector ID

TaskID	int8	The unique task id of a task in a connector

Current Assignment

The Current Assignment represents the current epoch and assignment of a worker. Note that workers 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 on the worker.The ID uniquely identifies the worker cluster.
Worker URL	string	The url of the worker.
Worker Epoch	int32	The current epoch of this worker. The epoch is the assignment epoch of the assignment currently used by this member. This epoch is the one used to fence the worker (e.g. offsets commit).
Error	int8	The error reported by the assignor.
Connectors	[]String	The current connectors 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 as follows:

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, group epoch is introduced which represents the generation (or the version) of the group metadata. The group epoch is incremented whenever the group metadata is updated. Rebalance is possible in the following cases:

1. A new worker joins.
2. A worker updates it’s assignors.
3. A worker updates its assignors' reason or metadata.
4. A worker is fenced or removed from the group by the group coordinator.
5. Connectors are added or removed.
6. Tasks are added/removed from Connectors(Couldn’t find it but think this would be applicable).

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.

In the case of Kafka Connect, we would continue using the Client Side assignors. This would be explained later on in the doc.

Member Epoch - Reconciliation of the group

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

1. The group coordinator revokes the connectors/tasks which are no longer in the target assignment of the member. It does so by providing the intersection of the Current connectors/tasks and the Target connectors/tasks in the heartbeat response until the worker acknowledges the revocation in the heartbeat response. We can repurpose the `rebalance.timeout.ms` config to put a cap on the rebalance process or else the worker would be kicked out of the group.
2. When the group coordinator receives the acknowledgement of the revocation, it updates the worker current assignment to its target assignment (and target epoch) and durably persist it.
3. The group coordinator assigns the new connectors/tasks to the worker. It does so by providing the Target connectors/tasks to the worker while ensuring that connectors/tasks which are not revoked by other workers yet are removed from this set. In other words, new connectors/tasks are incrementally assigned to the worker when they are revoked by the other workers.

Assignment Process

Whenever the group epoch is larger than the assignment epoch, the group coordinator must compute a new target assignment for the group. As already mentioned, for Connect we can piggyback on the Client Side Assignors already present.

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 Co-ordinator will use the assignors which are supported by all workers and if there are multiple such assignors, then the precedence order of assignments are honoured. This should be straightforward as as of today, Connect supports only 2 assignor modes => Eager and IncrementalCooperative.

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

- The group coordinator selects a worker to run the assignment logic. We will get it to it later.
- The group coordinator notifies the worker to compute the new assignment by returning the COMPUTE_ASSIGNMENT error in its next heartbeat response.
- When the worker receives this error, it is expected to call the **ConnectGroupPrepareAssignment** API to get the current group metadata and the current target assignment.
- The worker computes the new assignment with the relevant assignor.
- The worker calls the **ConnectGroupInstallAssignment** API to install the new assignment. The group coordinator validates it and persists it.

The worker should finish the assignment within `rebalance.timeout.ms` .

Worker Selection

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

- If we can't find such an overlapping worker, then we will throw a FATAL error.
- In the current Connect world, the Assignors are generally run on the Leader and the Leader assignment is sticky. We can evaluate if the stickiness is needed anymore or not.

Assignment Validation

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

- All connectors/tasks are assigned.
- A connector/task is assigned only once.
- All workers exists.

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 worker 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

If the Connect assignor can't compute the assignment, then it would return an error which would result in retaining the current assignments.

Member ID/Heartbeat & Session/Joining & Leaving/Fencing

These would remain as defined in the KIP.

Feature Flag

Similar to the KIP, a feature flag to enable/disable the new rebalance protocol.

Rebalance Process

This is the opposite of what was described above. The major difference from KIP is that, Connect has it's own WorkerRebalanceListener which is invoked based on assignment or revocation.

- If the worker is fenced by the group coordinator, it will immediately abandon all its tasks/connectors, stops these resources and invoke WorkerRebalanceListener#onRevoked. It will rejoin the group as a new member afterwards.
- Otherwise, the worker will compute the difference between its currently owned tasks/connectors and the assigned ones, as defined in the heartbeat response.
    - If there are any revoked partitions, it will revoke them, it stops (releases) these resources and call WorkerRebalanceListener#onRevoked.
    - If there are any newly assigned partitions, it starts the assigned connector and tasks and call WorkerRebalanceListener#onAssigned.

Client Side Assignor

The KIP talks about introducing a new interface for Client Side assignors called and deprecating the current ConsumerPartitionAssignor . Since connect has it's own assignor ConnectAssignor we would be extending the same.

Triggering Rebalances

The `IncrementalCooperativeAssignor` has a provision of triggering rebalances after a fixed `scheduled.rebalance.max.delay.ms` interval to allow any departed worker to come back so that it gets the same or similar assignments. Since the older rebalance protocol was triggered entirely through clients, this was possible while the new protocol offloads the rebalancing duties to the Group Coordinator. Having said that, the ClientSideAssignors can still trigger rebalances by setting the `reason` field which the Group Coordinator will need to understand. Here again, we will need to enhance the Group Coordinator logic based on `group.type` field `connect` .

Migration To the New Protocol

Upgrading to the new protocol or downgrading from it is possible by rolling the workers, assuming that the new protocol is enabled on the server side, with the correct `group.protocol` . When the first worker that supports the new protocol joins the group, the group is converted from `generic` to `connect` . Similarly when the last worker supporting the new protocol leaves, the group switches back to the old protocol. Note that the group epoch starts at the current group generation.

One important thing to note is that `JoinGroup, SyncGroup and Heartbeat` calls need to be translated into the new protocol. We will rely on ConnectGroupHeartbeat API for the translation of the 3 APIs. Concretely, the API updates the group state, provides the connectors/tasks owned by the worker, gets back the assignment, and 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 needed by returning the `REBALANCE_IN_PROGRESS` error in the heartbeat response.

The idea is to manage each workers individually while relying on the new engine to do the synchronization between them. Each worker 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 it’s assignments.

As in the older protocol, the possible worker states would still be

UNJOINED	The client is not part of a group
PREPARING_REBALANCE	The client has sent the join group request, but has not received response
COMPLETING_REBALANCE	The client has received join group response, but has not received assignment
STABLE	The client has joined and is sending heartbeats

<**Need to add more details>**

JoinGroup Handling

SyncGroup Handling

Heartbeat Handling

Rebalance Triggers

Public Interfaces

KRPC

New Errors

- FENCED_MEMBER_EPOCH - The member epoch does not correspond to the member epoch expected by the coordinator.
- COMPUTE_ASSIGNMENT - The member has been selected by the coordinator to compute the new target assignment of the group.
- UNSUPPORTED_ASSIGNOR - The assignor used by the member or its version range are not supported by the group.

ConnectGroupHeartbeat API

The ConnectGroupHeartbeat API is the new core API used by workers to form a group. The API allows workers to advertise their state, assignors and their owned connectors/tasks. The group coordinator uses it to assign/revoke connectors/tasks to/from workers. 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": "ConnectGroupHeartbeatRequest",
  "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 server. The member id must be kept during the entire lifetime of
the member. For connect, this corresponds
      to workerIds" },
    { "name": "MemberEpoch", "type": "int32", "versions": "0+", "default": "-1",
      "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 it not provided or if it didn't change since the last heartbeat; the instance Id
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": "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": "Version", "type": "int16", "versions": "0+",
          "about": "The version of the metadata." },
        { "name": "Metadata", "type": "bytes", "versions": "0+",
          "about": "The metadata." }
      ]},
      { "name": "ConnectorsAndTasks", "type": "[]ConnectorsAndTasks", "versions": "0+", "nullableVersions":
"0+", "default": "null",
      "about": "null if it didn't change since the last heartbeat; the connectors/tasks owned by the worker.
This will be set only when group.type is equal
      to connect",
      "fields": [
        { "name": "connectors", "type": "[]String", "versions": "0+",
          "about": "The Connectors assigned to this worker." },
        { "name": "tasks", "type": "[]ConnectorTaskID", "versions": "0+",
          "about": "The tasks assigned to this worker." }
      ]}
  ]
}
```

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.
- 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 handles a ConnectGroupHeartbeat 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 his request will be rejected with a FENCED_MEMBER_EPOCH. This is not optimal. Instead, the group coordinator could accept the request if the partitions or connectors/tasks owned by the members are a subset of the target assignments. This could be decided based on the `types` . 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.
5. Reconcile the member assignments as explained earlier in this document.

Response Schema

The group coordinator will only set the Assignment field when the member epoch is smaller than the target assignment epoch. This is done to ensure that the members converge to the target assignment.

```
{
  "apiKey": TBD,
  "type": "response",
  "name": "ConnectGroupHeartbeatResponse",
  "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
  // - COMPUTE_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." },
    { "name": "MemberEpoch", "type": "int32", "versions": "0+",
      "about": "The member epoch." },
    { "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": "ConnectorsAndTasks", "type": "[]ConnectorsAndTask", "versions": "0+",
          "about": "The assigned connectors/tasks to the member.",
          "fields": [
            { "name": "connectors", "type": "[]String", "versions": "0+","about": "The Connectors assigned to
this worker." },
            { "name": "tasks", "type": "[]ConnectorTaskID", "versions": "0+","about": "The tasks assigned to
this worker." }
        ]},
        { "name": "Version", "type": "int16", "versions": "0+",
          "about": "The version of the metadata." },
        { "name": "Metadata", "type": "bytes", "versions": "0+",
          "about": "The assigned metadata." }
    ]
  ]
}
```

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 connectors/tasks and the newly assigned ones. There should be either revoked connectors/tasks or newly assigned connectors/tasks The protocol never does both together.
    a. It revokes the connectors/tasks, release all resources, and calls WorkerRebalanceListener#onRevoked.
    b. It assigns the new connectors/tasks, calls ConnectAssignor#onAssignment and calls WorkerRebalanceListener#onAssigned.
3. After a revocation, It sends the next heartbeat immediately to acknowledge it.

Upon receiving the COMPUTE_ASSIGNMENT error, the worker starts the assignment process.

Upon receiving the UNKNOWN_MEMBER_ID or FENCED_MEMBER_EPOCH error, the worker abandons all its resources and rejoins with the same member id and the epoch 0.

ConnectGroupPrepareAssignment API

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

Request Schema

```
{
  "apiKey": TBD,
  "type": "request",
  "listeners": ["zkBroker", "broker"],
  "name": "ConnectGroupPrepareAssignmentRequest",
  "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 handles a ConnectGroupPrepareAssignmentRequest request:

1. Checks wether 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 wether the member epoch matches the current member epoch. If it does not, FENCED_MEMBER_EPOCH is returned.
4. Checks wether 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": "ConnectGroupPrepareAssignmentResponse",
  "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
  // - FENCED_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+",
        "about": "The member instance ID." },
      { "name": "Assignor", "type": "Assignor", "versions": "0+",
        "about": "The information of the selected assignor",
        "fields": [
        { "name": "Version", "type": "int16", "versions": "0+",
          "about": "The version of the metadata." },
        { "name": "Reason", "type": "int8", "versions": "0+",
          "about": "The reason of the metadata update." },
        { "name": "Metadata", "type": "bytes", "versions": "0+",
          "about": "The assignor metadata." }
      ]},
      { "name": "ConnectorsAndTasks", "type": "[]ConnectorsAndTask", "versions": "0+",
          "about": "The assigned connectors/tasks to the member.",
          "fields": [
            { "name": "connectors", "type": "[]String", "versions": "0+","about": "The Connectors assigned to
this worker." },
            { "name": "tasks", "type": "[]ConnectorTaskID", "versions": "0+","about": "The tasks assigned to
this worker." }
      ]}
    ]}
  ]
}
```

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

ConnectGroupInstallAssignment API

The ConnectGroupInstallAssignment API will be used by the member 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": "ConnectGroupInstallAssignment",
  "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": "ConnectorsAndTasks", "type": "[]String", "versions": "0+",
        "about": "The assigned topic-partitions to the member.",
        "fields": [
          { "name": "Connectors", "type": "string", "versions": "0+",
            "about": "The connectors assigned to this worker" },
          { "name": "tasks", "type": "[]ConnectorTaskID", "versions": "0+",
            "about": "The tasks assigned to this worker." }
        ]},
      { "name": "Version", "type": "int32", "versions": "0+",
        "about": "The metadata version." }
      { "name": "Metadata", "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.
- Both Partitions and ConnectorsAndTasks are set.

Request Handling

When the group coordinator handles a ConnectGroupInstallAssignment request:

1. Checks wether the group exists. If it does not, GROUP_ID_NOT_FOUND is returned.
2. Checks wether the member exists. If it does not, UNKNOWN_MEMBER_ID is returned.
3. Checks wether the member epoch matches the current member epoch. If it does not, FENCED_MEMBER_EPOCH is returned.
4. Checks wether 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": "ConnectGroupInstallAssignment",
  "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
  // - FENCED_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.

Upon receiving the FENCED_MEMBER_EPOCH error, the worker retries when receiving its next heartbeat response with its member epoch.

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

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.

They will be persisted in the __worker_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 ConnectWorkerGroupMetadata and the ConnectWorkerGroupMemberMetadata. Note that since these messages are independent of Consumer Groups, we are introducing new record types.

A group with X members will be stored with X+2 records. One ConnectWorkerGroupMemberMetadata per member, one ConnectWorkerGroupConnectorsTasksMetadata, and one ConnectWorkerGroupMetadata 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.

ConnectWorkerGroupMetadataKey

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

ConnectWorkerGroupMetadataValue

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

ConnectWorkerGroupConnectorsTasksMetadataKey

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

ConnectWorkerGroupConnectorsTasksMetadataValue

```
{
    "type": "data",
    "name": "ConnectWorkerGroupConnectorsTasksMetadataValue",
    "validVersions": "0",
    "flexibleVersions": "0+",
    "fields": [
        { "name": "Epoch", "versions": "0+", "type": "int32" },
        { "name": "ConnectorsTasks", "versions": "0+",
          "type": "[]ConnectorsTasks", "fields": [
            { "name": "Connectors", "versions": "0+", "type": "[]String" },
            { "name": "Tasks", "versions": "0+", "type": "[]ConnectorsTasks" }
          ]}
    ],
}
```

ConnectWorkerGroupMemberMetadataKey

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

ConnectWorkerGroupMemberMetadataValue

```
{
    "type": "data",
    "name": "ConnectWorkerGroupMemberMetadataValue",
    "validVersions": "0",
    "flexibleVersions": "0+",
    "fields": [
        { "name": "GroupEpoch", "versions": "0+", "type": "int32" },
        { "name": "InstanceId", "versions": "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" }
          ]}
    ],
}
```

Target Assignment

The target assignment is stored in a single record.

ConnectWorkerGroupTargetAssignmentKey

```
{
    "type": "data",
    "name": "ConnectWorkerGroupTargetAssignmentKey",
    "validVersions": "6",
    "flexibleVersions": "none",
    "fields": [
          { "name": "GroupId", "type": "string", "versions": "5" }
    ]
}
```

ConnectWorkerGroupTargetAssignmentValue

```
{
    "type": "data",
    "name": "GroupTargetAssignmentValue",
    "validVersions": "0",
    "flexibleVersions": "0+",
    "fields": [
        { "name": "AssignmentEpoch", "versions": "0+", "type": "int32" },
        { "name": "Members", "versions": "0+", "type": "[]Member", "fields": [
            { "name": "MemberId", "versions": "0+", "type": "string" },
            { "name": "Error", "versions": "0+", "type": "int8" },
            { "name": "ConnectorsTasks", "versions": "0+", "type": "[]ConnectorsTasks", "fields":
            [
              { "name": "Connectors", "versions": "0+", "type": "[]String" },
              { "name": "Tasks", "versions": "0+", "type": "[]ConnectorsTasks" }
            ]
        },
         { "name": "Version", "versions": "0+", "type": "int16" },
            { "name": "Metadata", "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.

ConnectWorkerGroupCurrentMemberAssignmentKey

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

ConnectWorkerGroupCurrentMemberAssignmentValue

```
{
    "type": "data",
    "name": "ConnectGroupCurrentMemberAssignmentValue",
    "validVersions": "0",
    "flexibleVersions": "0+",
    "fields": [
        { "name": "MemberEpoch", "versions": "0+", "type": "int32" },
        { "name": "Error", "versions": "0+", "type": "int8" },
        { "name": "ConnectorsTasks", "versions": "0+", "type": "[]ConnectorsTasks", "fields":
            [
              { "name": "Connectors", "versions": "0+", "type": "[]String" },
              { "name": "Tasks", "versions": "0+", "type": "[]ConnectorsTasks" }
            ]
        },
        { "name": "Version", "versions": "0+", "type": "int16" },
        { "name": "Metadata", "versions": "0+", "type": "bytes" }
    ],
}
```

Group Configurations

ConnectWorkerGroupConfigurationKey

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

ConnectWorkerGroupConfigurationValue

```
{
    "type": "data",
    "name": "ConnectWorkerGroupConfigurationValue",
    "validVersions": "0",
    "flexibleVersions": "0+",
    "fields": [
        { "name": "Configurations", "versions": "0+", "type": "[]Configuration",
          "fields": [
             { "name": "Name", "type": "string", "versions": "0+",
               "about": "The name of the configuration key." },
             { "name": "Value", "type": "string", "versions": "0+",
               "about": "The value of the configuration." }
        ]}
    ]
}
```

Broker Metrics

We can add them later on.

Client side API

ConnectAssignor

```
package org.apache.kafka.connect.runtime;

import org.apache.kafka.clients.consumer.Assignor;
import org.apache.kafka.connect.runtime.distributed.ExtendedAssignment;
import org.apache.kafka.connect.runtime.distributed.ExtendedWorkerState;
import org.apache.kafka.connect.runtime.distributed.WorkerCoordinator;

import java.util.List;

public interface ConnectAssignor {

    class Group {

         /**
         * The members.
         */
        List<GroupMember> members;
```

```
    /**
     * Connector's and tasks metadata.
     */
    WorkerCoordinator.ConnectorsAndTasks connectorsAndTasks;
}

    class GroupMember {
    /**
     * The member ID.
     */
    String memberId;

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

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

    /**
     * The version of the metadata encoded in {{@link GroupMember#metadata()}}.
     */
    int version;

    /**
     * The custom metadata provided by the member as defined
     * by {{@link PartitionAssignor#metadata()}}.
     */
    ByteBuffer metadata;

      /**
     * The worker state signifying the assigned connectors and
     * tasks.
     * Note
     */
    ExtendedWorkerState workerState;


}


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

    /**
     * The member assignment.
     */
    List<MemberAssignment> members;
}


class MemberAssignment {


             /**
     * The member ID.
     */
    String memberId;

    /**
     * The error reported by the assignor.
     */
    byte error;
```

```
        /**
         * The version of the metadata encoded in {{@link GroupMember#metadata()}}.
         */
        int version;

        /**
         * The custom metadata provided by the assignor.
         */
        ByteBuffer metadata;

        /**
         * The worker state signifying the assigned connectors and
         * tasks.
         */
        ExtendedAssignment assignment;


    }


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

        /**
         * The version of the metadata encoded in {{@link Metadata#metadata()}}.
         */
        int version;

        /**
         * The custom metadata provided by the assignor.
         */
        ByteBuffer metadata;
    }

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

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

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

    /**
     * Return serialized data that will be sent to the assignor.
     */
    Metadata metadata();

    /**
     * Perform the group assignment given the current members and
     * topic metadata.
     *
     * @param group The group state.
     * @return The new assignment for the group.
     */
    Assignment assign(Group group);

    /**
     * Callback which is invoked when the member received a new
     * assignment from the assignor/group coordinator.
     */
    void onAssignment(MemberAssignment assignment);
```

```
}
```

Worker and Assignment Metadata

We can reuse the current metadata encoded in the protocol.

Connect REST API Endpoints

All the APIs should work the way they work currently.

Case Studies

Let's take a look at few example scenarios to understand how the rebalancing would work in the new protocol. I am taking a few illustrations from KIP-415 t o keep it familiar. Similar to the KIP, *first letter of the recourse is a Connector instance (e.g. Connector A, Connector B, etc). Second letter is type: C for Connector, T for task. Number is regular task numbering. 0 for Connectors, greater or equal to 1 for Tasks. W represents a Worker, with W1 and W2, etc being different Workers joining the same group. Primes are used to represent a Worker that was member of the group and rejoins soon after a short period of being offline.*

The config topic contains the following connectors/tasks => AC0, AT1, AT2, BC0, BT1.

Let's use the IncrementalCooperativeAssignor as the client side assignor.

Empty Group

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

W1 joins the Group

The coordinator bumps the group epoch to 1, adds W1 to the group, and creates an empty member assignment.

- Group (epoch=1)
    - W1
- Target Assignment (epoch=0)
    - Empty
- Member Assignment
    - W1 - epoch=0, partitions=[]

W1 is selected as the member to run the client assignor. All connectors and tasks would be assigned to it and installed as target assignment

- Group (epoch=1)
    - W1
- Target Assignment (epoch=1)
    - W1 - connectorsAndTasks=[AC0, AT1, AT2, BC0, BT1]
- Member Assignment
    - W1 - epoch=0, connectorsAndTasks=[]

When W1 heartbeats, the group coordinator transitions it to its target epoch/assignment because it does not have any connectors/tasks to revoke. The group coordinator updates the member assignment and replies with the new epoch 1 and all the assignments.

- Group (epoch=1)
    - W1
- Target Assignment (epoch=1)
    - W1 - connectorsAndTasks=[AC0, AT1, AT2, BC0, BT1]
- Member Assignment
    - W1 - epoch=1, connectorsAndTasks=[AC0, AT1, AT2, BC0, BT1]

Since this is the first member in the group, there won't be any rebalances based on `scheduled.rebalance.max.delay.ms`.

W2 joins the Group

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

- Group (epoch=2)
    - W1
    - W2
- Target Assignment (epoch=1)
    - W1 - connectorsAndTasks=[AC0, AT1, AT2, BC0, BT1]
- Member Assignment
    - W1 - epoch=1, connectorsAndTasks=[AC0, AT1, AT2, BC0, BT1]
    - W2 - epoch=0, connectorsAndTasks=[]

W1 is chosen as the member to run the client side assignor. It computes and sends the assignments to coordinator which would install it.

- Group (epoch=2)
    - W1
    - W2
- Target Assignment (epoch=2)
    - W1 - connectorsAndTasks=[AC0, AT1, AT2]
    - W2- connectorsAndTasks=[BC0, BT1]
- Member Assignment
    - W1 - epoch=1, connectorsAndTasks=[AC0, AT1, AT2, BC0, BT1]
    - W2 - epoch=1, connectorsAndTasks=[]

W2 can be transitioned to epoch1 but it can't have BC0, BT1 until W1 revokes them.

When W1 heartbeats the next time, it would be asked to revoke them. When it acknowledges the revocation, it coordinator would transition it to epoch 2.

- Group (epoch=2)
    - W1
    - W2
- Target Assignment (epoch=2)
    - W1 - connectorsAndTasks=[AC0, AT1, AT2]
    - W2- connectorsAndTasks=[BC0, BT1]
- Member Assignment
    - W1 - epoch=2, connectorsAndTasks=[AC0, AT1, AT2]
    - W2 - epoch=1, connectorsAndTasks=[]

When W2 heartbeats, it would get it's assignments and be bumped to epoch 2.

- Group (epoch=2)
    - W1
    - W2
- Target Assignment (epoch=2)
    - W1 - connectorsAndTasks=[AC0, AT1, AT2]
    - W2- connectorsAndTasks=[BC0, BT1]
- Member Assignment
    - W1 - epoch=2, connectorsAndTasks=[AC0, AT1, AT2]
    - W2 - epoch=2, connectorsAndTasks=[BC0, BT1]

Worker Leaving the Group

Lets assume epoch is at 35 with 3 workers

- Group (epoch=35)
    - W1
    - W2
    - W3
- Target Assignment (epoch=35)
    - W1 - connectorsAndTasks=[AC0, AT1]
    - W2- connectorsAndTasks=[BC0, BT1]
    - W3- connectorsAndTasks=[AT2]
- Member Assignment
    - W1 - epoch=35, connectorsAndTasks=[AC0, AT1]
    - W2 - epoch=35, connectorsAndTasks=[BC0, BT1]

    - W3 - epoch=35, connectorsAndTasks=[AT2]

Let's say W2 fails to heartbeat and the group coordinator kicks it out after the session timeout expires and bump the group epoch.

- Group (epoch=36)
    - W1
    - W2
- Target Assignment (epoch=35)
    - W1 - connectorsAndTasks=[AC0, AT1]
    - W2- connectorsAndTasks=[BC0, BT1]
    - W3- connectorsAndTasks=[AT2]
- Member Assignment
    - W1 - epoch=35, connectorsAndTasks=[AC0, AT1]
    - W3 - epoch=35, connectorsAndTasks=[AT2]

W3 is chosen to run the client side assignor. However, this time it won't change any assignments to give the failing worker a chance to come back.

- Group (epoch=36)
    - W1
    - W3
- Target Assignment (epoch=36)
    - W1 - connectorsAndTasks=[AC0, AT1]
    - W3- connectorsAndTasks=[AT2]
- Member Assignment
    - W1 - epoch=35, connectorsAndTasks=[AC0, AT1]
    - W3 - epoch=35, connectorsAndTasks=[AT2]

When W1 and W3 heartbeat, they would transition to epoch 36 and get their new assignments (which would be the same).

- Group (epoch=36)
    - W1
    - W3
- Target Assignment (epoch=36)
    - W1 - connectorsAndTasks=[AC0, AT1]
    - W3- connectorsAndTasks=[AT2]
- Member Assignment
    - W1 - epoch=36, connectorsAndTasks=[AC0, AT1]
    - W3 - epoch=36, connectorsAndTasks=[AT2]

At this point, there are unassigned connectors. The chosen member would trigger another rebalance after delay `d`. This is in line with what `IncrementalCooperativeProtocol` supports. It would do so by updating the reason field which would instruct the Group Coordinator to trigger another rebalance.

- Group (epoch=37)
    - W1
    - W3
- Target Assignment (epoch=36)
    - W1 - connectorsAndTasks=[AC0, AT1]
    - W3- connectorsAndTasks=[AT2]
- Member Assignment
    - W1 - epoch=36, connectorsAndTasks=[AC0, AT1]
    - W2 - epoch=36, connectorsAndTasks=[AT2]

W3 is chosen to run the client side assignor. It would recompute the assignments and send them to the Group Coordinator.

- Group (epoch=37)
    - W1
    - W3
- Target Assignment (epoch=37)
    - W1 - connectorsAndTasks=[AC0, AT1, BC0]
    - W3- connectorsAndTasks=[BC1, AT2]
- Member Assignment
    - W1 - epoch=36, connectorsAndTasks=[AC0, AT1]
    - W3 - epoch=36, connectorsAndTasks=[AT2]

When W1 and W3 would heartbeat, they would receive their assignments, thereby marking the rebalance as done.

- Group (epoch=37)

    - W1
    - W3
- Target Assignment (epoch=37)
    - W1 - connectorsAndTasks=[AC0, AT1, BC0]
    - W3- connectorsAndTasks=[BC1, AT2]
- Member Assignment
    - W1 - epoch=37, connectorsAndTasks=[AC0, AT1, BC0]
    - W3 - epoch=37, connectorsAndTasks=[BC1, AT2]

Worker Bounces

This is the case for which incremental cooperative rebalance protocol was purpose built for. Basically, giving the worker to come back. Let's assume a similar setup as the previous example and let's say W2 becomes unresponsive

Initial State

- Group (epoch=35)
    - W1
    - W2
    - W3
- Target Assignment (epoch=35)
    - W1 - connectorsAndTasks=[AC0, AT1]
    - W2- connectorsAndTasks=[BC0, BT1]
    - W3- connectorsAndTasks=[AT2]
- Member Assignment
    - W1 - epoch=35, connectorsAndTasks=[AC0, AT1]
    - W2 - epoch=35, connectorsAndTasks=[BC0, BT1]
    - W3 - epoch=35, connectorsAndTasks=[AT2]

W2 becomes unresponsive and it kicked out by Group Coordinator

- Group (epoch=36)
    - W1
    - W2
- Target Assignment (epoch=35)
    - W1 - connectorsAndTasks=[AC0, AT1]
    - W2- connectorsAndTasks=[BC0, BT1]
    - W3- connectorsAndTasks=[AT2]
- Member Assignment
    - W1 - epoch=35, connectorsAndTasks=[AC0, AT1]
    - W3 - epoch=35, connectorsAndTasks=[AT2]

W3 is chosen to run the client side assignor. However, this time it won't change any assignments to give the failing worker a chance to come back.

- Group (epoch=36)
    - W1
    - W3
- Target Assignment (epoch=36)
    - W1 - connectorsAndTasks=[AC0, AT1]
    - W3- connectorsAndTasks=[AT2]
- Member Assignment
    - W1 - epoch=35, connectorsAndTasks=[AC0, AT1]
    - W3 - epoch=35, connectorsAndTasks=[AT2]

When W1 and W3 heartbeat, they would transition to epoch 36 and get their new assignments (which would be the same).

- Group (epoch=36)
    - W1
    - W3
- Target Assignment (epoch=36)
    - W1 - connectorsAndTasks=[AC0, AT1]
    - W3- connectorsAndTasks=[AT2]
- Member Assignment
    - W1 - epoch=36, connectorsAndTasks=[AC0, AT1]
    - W3 - epoch=36, connectorsAndTasks=[AT2]

Since W3 would wait for `d` delay before triggering another rebalance, but let's say before that W2 joins back. It may join with the last know epoch that it had seen or even with epoch 0. Let's say it joins with epoch 35. The Group coordinator would notice this, trigger a rebalance and update the group epoch to 37.

- Group (epoch=36)
    - W1
    - W2
    - W3
- Target Assignment (epoch=36)
    - W1 - connectorsAndTasks=[AC0, AT1]
    - W3- connectorsAndTasks=[AT2]
- Member Assignment
    - W1 - epoch=36, connectorsAndTasks=[AC0, AT1]
    - W3 - epoch=36, connectorsAndTasks=[AT2]

Since this would anyways trigger a new assignment, W3 if chosen would be asked to re-assign. W3 would notice that there is a rebalance delay in process so it would go ahead and re-assign the same assignments and send them back to the coordinator. After this, the group would finally transition to epoch 37 with original assignments. Final states:

- Group (epoch=37)
    - W1
    - W2
    - W3
- Target Assignment (epoch=37)
    - W1 - connectorsAndTasks=[AC0, AT1]
    - W2- connectorsAndTasks=[BC0, BT1]
    - W3- connectorsAndTasks=[AT2]
- Member Assignment
    - W1 - epoch=37, connectorsAndTasks=[AC0, AT1]
    - W2 - epoch=37, connectorsAndTasks=[BC0, BT1]
    - W3 - epoch=37, connectorsAndTasks=[AT2]