KIP-345: Introduce static membership protocol to reduce consumer rebalances

- Status
- Motivation
- Public Interfaces
    - Client Side Changes
        - Stream Side Change
    - Server Side Changes
    - Command Line API and Scripts
    - Client Behavior Changes
        - Kafka Streams Change
    - Server Behavior Changes
        - Join Group Logic Change
        - Leave Group Logic Change
            - Command Line API for Membership Management
    - Upgrade Process
    - Downgrade Process
        - Switching from Static Member to Dynamic Member
- Non Goal
- Rejected Alternatives
- Future Works

Status

Current state: Accepted

Discussion thread: here

JIRA:

KAFKA-7018 - Getting issue details... STATUS

KAFKA-7610 - Getting issue details... STATUS

KAFKA-7725 - Getting issue details... STATUS

KAFKA-7728 - Getting issue details... STATUS

KAFKA-7798 - Getting issue details... STATUS

KAFKA-6995 - Getting issue details... STATUS

KAFKA-8224 - Getting issue details... STATUS

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

Motivation

For stateful applications, one of the biggest performance bottleneck is the state shuffling. In Kafka consumer, there is a concept called "rebalance" which means that for given M partitions and N consumers in one consumer group, Kafka will try to balance the load between consumers and ideally have each consumer dealing with M/N partitions. Broker will also adjust the workload dynamically by monitoring consumers' health so that we could kick dead consumer out of the group, and handling new consumers' join group request. When the service state is heavy, a rebalance of one topic partition from instance A to B means huge amount of data transfer. If multiple rebalances are triggered, the whole service could take a very long time to recover due to data transfer.

*The idea of this KIP is to reduce number of rebalances by introducing a new concept called **static membership**. It would help with following example use cases.*

1. *Improve performance of heavy state applications. We have seen that rebalance is the major performance killer with large state application scaling, due to the time wasted in state shuffling.*
2. *Improve general rolling bounce performance. For example MirrorMaker processes take a long time to rolling bounce the entire cluster, because one process restart will trigger one rebalance. With the change stated, we only need constant number of rebalance (e.g. for leader restart) for the*

entire rolling bounce, which will significantly improves the availability of the MirrorMaker pipeline as long as they could restart within the specified timeout.

Background of Consumer Rebalance

Right now broker handles consumer state in a two-phase protocol. To solely explain consumer rebalance, we only discuss 3 involving states here: *RUNNING, PREPARE_REBALANCE and COMPLETING_REBALANCE*.

- When a consumer joins the group, if this is a new member or the group leader, the broker will move this group state from *RUNNING* to *PREPARE_REBALANCE.* The reason for triggering rebalance when leader rejoins is because there might be assignment protocol change (f*or example if the consumer group is using regex subscription and new matching topics show up*). If an old normal member rejoins the group, the state will not change.
- When moved to *PREPARE_REBALANCE* state, the broker will mark first joined consumer as leader, and wait for all the members to rejoin the group. Once we collected all current members' join group requests or reached rebalance timeout, we will reply the leader with current member information and move the state to *COMPLETING_REBALANCE*. All current members are informed to send SyncGroupRequest to get the final assignment.
- The leader consumer will decide the assignment and send it back to broker. As last step, broker will announce the new assignment by sending SyncGroupResponse to all the followers. Till now we finished one rebalance and the group generation is incremented by 1.

In the current architecture, during each rebalance consumer groups on broker side will assign new member a randomly generated id called `member.id` each time. This is to make sure we have unique identity for each group member. During client restart, consumer will send a JoinGroupRequest with a special UNKNOWN_MEMBER_ID (empty string), and broker will interpret it as a new member. To make this KIP work, we need to change both client side and server side logic to make sure we persist member identity by persisting a new `group.instance.id` (explained later) throughout restarts, which means we could reduce number of rebalances since we are able to apply the same assignment based on member identities. The idea is summarized as **static membership**, which in contrary to **dynamic membership** (the one our system currently uses), is prioritizing "state persistence" over "liveness".

We will be introducing two new terms:

- *Static Membership*: the membership protocol where the consumer group will not trigger rebalance unless
  - A new member joins
  - A leader rejoins (possibly due to topic assignment change)
  - An existing member offline time is over session timeout
  - Broker receives a leave group request containing alistof `group.instance.id`s (details later)
- Group instance id: the unique identifier defined by user to distinguish each client instance.

Public Interfaces

New Configurations

Consumer Configs

group.instance.id	The unique identifier of the consumer instance provided by end user. If set to non-null string, the consumer is treated as a static member, otherwise an null id indicates a dynamic member. **Default value**: null string.

Client Side Changes

The new `group.instance.id` config will be added to the Join/Sync/Heartbeat/OffsetCommit request/responses.

A list of tuples containing `group.instance.id` and `member.id` will be added to the LeaveGroupRequest, while removing the single `member.id` field.

```
JoinGroupRequest => GroupId SessionTimeout RebalanceTimeout MemberId GroupInstanceId ProtocolType GroupProtocols
  GroupId             => String
  SessionTimeout      => int32
  RebalanceTimeout          => int32
  MemberId            => String
  GroupInstanceId     => String // new
  ProtocolType        => String
  GroupProtocols      => [Protocol MemberMetadata]
  Protocol            => String
  MemberMetadata      => bytes

JoinGroupResponse => ThrottleTime ErrorCode GenerationId ProtocolName LeaderId MemberId Members
  ThrottleTime                  => int16
  ErrorCode                         => int16
```

```
  GenerationId                 => int32
  ProtocolName                 => String
  LeaderId                      => String
  MemberId                      => String
  Members                      => []JoinGroupResponseMember
                                        MemberId                => String
                                        GroupInstanceId  => String // new
                                        Metadata                      => bytes

SyncGroupRequest => GroupId GenerationId MemberId GroupInstanceId Assignments
  GroupId             => String
  GenerationId        => int32
  MemberId            => String
  GroupInstanceId     => String // new
  Assignments         => []SyncGroupRequestAssignment
                                  MemberId                => String
                                   Assignment            => bytes

SyncGroupResponse => ThrottleTime ErrorCode Assignment
  ThrottleTime                  => int16
  ErrorCode                      => int16
  Assignment                 => bytes

HeartbeatRequest => GroupId GenerationId MemberId GroupInstanceId
  GroupId             => String
  GenerationId        => int32
  MemberId            => String
  GroupInstanceId     => String // new

HeartbeatResponse => ThrottleTime ErrorCode Assignment
  ThrottleTime                  => int16
  ErrorCode                      => int16

OffsetCommitRequest => GroupId GenerationId MemberId GroupInstanceId Topics
  GroupId             => String
  GenerationId        => int32
  MemberId            => String
  GroupInstanceId     => String // new
  Topics                 => []OffsetCommitRequestTopic
                                        Name         => String
                                        Partitions   => []OffsetCommitRequestPartition
                                                                PartitionIndex
=> int32
                                                                CommittedOffset
=> int64
                                                                CommittedLeaderEpoch
=> int32
                                                                CommitTimestamp
=> int64

CommittedMetadata           => String

OffsetCommitResponse => ThrottleTime Topics
  ThrottleTime                  => int16
  Topics                => []OffsetCommitResponseTopic
                                        Name         => String
                                        Partitions   => []OffsetCommitResponsePartition
                                                                PartitionIndex
=> int32
                                                                ErrorCode
=> int16

LeaveGroupRequest => GroupId MemberIdentityList
  GroupId             => String
  MemberId            => String // removed
  MemberIdentityList  => []MemberIdentity // new
                                        MemberId         => String
                                        GroupInstanceId  => String
```

In the meantime, for better visibility for static members, we are also going to bump DescribeGroup request/response protocol to include `group.instance.id`:

```
DescribeGroupRequest => ThrottleTime Groups
  ThrottleTime           => int16
  Groups                 => []DescribeGroups
                                                        ErrorCode        => int16
                                                        GroupId          => String
                                                        GroupState       => String
                                                        ProtocolType     => String
                                                        ProtocolData     => int16
                                                        Members          => []DescribedGroupMember
                                                                                                MemberId
=> String

GroupInstanceId  => String // new

ClientId         => String

ClientHost       => String

MemberMetadata   => bytes

MemberAssignment => bytes
```

Of course, we would bump the Join/Sync/Heartbeat/OffsetCommit/Leave/Describe group request/response versions by 1.

We shall use new JoinGroupResponseMember struct to replace the current subscription struct.

ConsumerCoordinator.java

```
Map<String, ByteBuffer> allSubscriptions -> List<JoinGroupResponseData.JoinGroupResponseMember>
allSubscriptions;
```

We shall also add a new public function to `Subscription` class in `PartitionAssignor` to get `group.instance.id`:

PartitionAssignor.java

```
class Subscription {
        ...
        public Optional<String> groupInstanceId();
}
```

Similar to the MemberDescription interface (for describe group):

MemberDescription.java

```
class Subscription {
        ...
        public Optional<String> groupInstanceId();
}
```

We are also introducing a new error type. Will explain the handling in the following section.

Errors.java

```
FENCED_INSTANCE_ID(78, "This implies some group.instance.id is already in the consumer group, however the
corresponding member.id was not matching the record on coordinator", FencedInstanceIdException::new)
```

Stream Side Change

On Kafka Streams side, we plan to expose the list of `group.instance.id` for easy management. This will be done in KIP-414 to expose main consumer client ids which are equivalent to `group.instance.id`s.

Server Side Changes

We shall increase the cap of session timeout to 30 min for relaxing static membership liveness tracking.

KafkaConfig.scala

```
val GroupMaxSessionTimeoutMs = 1800000 // 30 min for max cap
```

For fault-tolerance, we also include `group.instance.id` within the member metadata to backup in the __consumer_offsets topic.

GroupMetadataManager

```
private val MEMBER_METADATA_V3 = new Schema(
  new Field(MEMBER_ID_KEY, STRING),
  new Field(GROUP_INSTANCE_ID_KEY, STRING), // new
  new Field(CLIENT_ID_KEY, STRING),
  new Field(CLIENT_HOST_KEY, STRING),
  new Field(REBALANCE_TIMEOUT_KEY, INT32),
  new Field(SESSION_TIMEOUT_KEY, INT32),
  new Field(SUBSCRIPTION_KEY, BYTES),
  new Field(ASSIGNMENT_KEY, BYTES))
```

Command Line API and Scripts

We will define one command line API to help us better manage consumer groups:

AdminClient.java

```
public static MembershipChangeResult removeMemberFromConsumerGroup(String groupId,
RemoveMemberFromConsumerGroupOptions options);
```

And a separate option class:

RemoveMemberFromGroupOptions.java

```
public class RemoveMemberFromGroupOptions extends AbstractOptions<RemoveMemberFromGroupOptions> {
        ...
        private List<MemberIdentity> members; // members to be removed
}
```

which will use the latest LeaveGroupRequest API to inform broker the permanent leaving of a bunch of consumer instances.

Proposed Changes

In short, the proposed feature is enabled if

1. Latest JoinGroupReq/Res and LeaveGroupReq/Res are supported on both client and broker.
2. `group.instance.id` is configured with non-null string.

Client Behavior Changes

On client side, we add a new config called `group.instance.id` in ConsumerConfig. On consumer service init, if the `group.instance.id` config is set, we will put it in the initial join group request to identify itself as a static member. Note that it is user's responsibility to assign unique `group.instance.id` for each consumers. This could be in service discovery hostname, unique IP address, etc. We also have logic handling duplicate `group.instance.id` in case client configuration contains duplicates.

For the effectiveness of the KIP, consumer with `group.instance.id` set will **not send leave group request** when they go offline, which means we shall only rely on **session.timeout** to trigger group rebalance. It is because the proposed rebalance protocol will trigger rebalance with this intermittent in-and-out which is not ideal. In static membership we leverage the consumer group health management to client application such as k8s. Therefore, it is also advised to make the session timeout large enough so that broker side will not trigger rebalance too frequently due to member come and go. By having a handful admin tool, user could proactively remove members if session timeout is too long in runtime.

Since the member id is randomly generated by broker, the persistence behavior of static membership will be hindered since the leader doesn't know whether this member is new or old. For leader to make better assignment decision, we are attaching `group.instance.id` on response members within the join group response.

One example is like (Thanks Jason for the idea!):

```
Suppose we have three consumers in the group with static instance ids: A, B, and C.
Assume a stable group and the respective memberIds are 1, 2, and 3.
So inside group coordinator, we have the following state:
members: {A=1, B=2, C=3}
generation: 5

In fact, the consumer leader of the group is not aware of the instance ids of the members.
So it sees the membership as:
members: {1, 2, 3}.
generation: 5

Now suppose that A does a rolling restart. After restarting,
the coordinator will assign a new memberId to A and let it continue using the previous assignment.
So we now have the following state:
members: {A=4, B=2, C=3}
generation: 5

The leader on the other hand still sees the members in the group as {1, 2, 3}
because it does not know that member A restarted and was given a new memberId.
Suppose that eventually something causes the group to rebalance (e.g. maybe a new topic was created).
When the leader attempts its assignment, it will see the members {2, 3, 4}.

However, appending group.instance.id for join group response provides some benefit
even for the simple partition assignors. Consider, the default range assignor, for example.
Basically it works by sorting the members in the group and
then assigning partition ranges to achieve balance. Suppose we have a partition with 9 partitions.
If the membership were {1, 2, 3}, then the assignment would be the following:
memberId: 1, assignment: {0, 1, 2}
memberId: 2, assignment: {3, 4, 5}
memberId: 3, assignment: {6, 7, 8}

Now when the membership changes to {2, 3, 4}, then all the assignments change as well:
memberId: 2, assignment: {0, 1, 2}
memberId: 3, assignment: {3, 4, 5}
memberId: 4, assignment: {6, 7, 8}

So basically all of the assignments change even though it's the same static members.
However, if we could consider the instanceId as the first sort key,
then we can compute the assignment consistently even across restarts:
instanceId: A, memberId: 1, assignment: {0, 1, 2}
instanceId: B, memberId: 2, assignment: {3, 4, 5}
instanceId: C, memberId: 3, assignment: {6, 7, 8}

And after the restart:
instanceId: A, memberId: 4, assignment: {0, 1, 2}
instanceId: B, memberId: 2, assignment: {3, 4, 5}
instanceId: C, memberId: 3, assignment: {6, 7, 8}

The full benefit of static assignment can only be realized
if the assignor knows the instance ids of the members in the group.
It shouldn't be necessary to do anything fancy with additional metadata.
```

Kafka Streams Change

KStream uses stream thread as consumer unit. For a stream instance configured with `num.threads` = 16, there would be 16 main consumers running on a single instance. If user specifies the client id, the stream consumer client id will be like: **User client id + "-StreamThread-" + thread id + "-consumer"**. If user client id is not set, then we will use process id. Our plan is to reuse the consumer client id to define `group.instance.id`, so effectively the KStream instance will be able to use static membership if end user defines unique `client.id` for stream instances.

For easy operation, we define a new field in StreamsMetadata to expose all the `group.instance.id` given on each stream instance, so that user could

1. Use REST API to get list of `group.instance.id` on stream instances user wants to remove
2. Shutdown targeting stream instances

3. Use command line API to batch remove offline consumers

******Update 04/25******

We are going to let stream user directly configures `group.instance.id`, for the sake of avoiding surprising triggering of static membership. On per thread basis, we will pass in **(user configured group.instance.id) + "-thread-" + thread id** to make sure each main consumer uses unique instance id within one Kafka Stream instance.

Server Behavior Changes

Join Group Logic Change

On server side, broker will keep handling join group request <= v3 as before. The `member.id` generation and assignment is still coordinated by broker, and broker will maintain an in-memory mapping of {group.instance.id member.id} to track member uniqueness. When receiving a known member's (A.K.A `group.instance.id` known) rejoin request, broker will return the cached assignment back to the member, without doing any rebalance.

For join group requests under static membership (with `group.instance.id` set),

- If the `member.id` uses UNKNOWN_MEMBER_ID,
  - if `group.instance.id` was found on the static map, we shall generate a member.id to reply to the member rejoin request immediately when the group is doing stable. This is to guard against duplicate consumers joining with same `group.instance.id`. We also expect that after KIP-394, all the join group requests are requiring `member.id` to physically enter the consumer group, so the behavior of static member is consistent with that proposal.
  - Following the above definition, it would never be possible for static members to receive a MEMBER_ID_REQUIRED exception, nor being put in pending member map.
  - if not found, we shall generate a new member id and add the new key-value pair into static member map.
- we are requiring member.id (if not unknown) to match the value stored in cache, otherwise reply FENCED_INSTANCE_ID. The edge case is that if we could have members with the same `group.instance.id` (for example mis-configured instances with a valid `member.id` but added a used `group.instance.id` on runtime). When `group.instance.id` has duplicates, we could refuse join request from members with an outdated `member.id`, since we update the mapping upon each join group request. In an edge case where the client hits this exception in the response, it is suggesting that some other consumer takes its spot. The client should immediately fail itself to inform end user that there is a configuration bug which is generating duplicate consumers with same identity. For first version of this KIP, we just want to have straightforward handling to expose the error in early stage and reproduce bug cases easily. The exception could be thrown on any client functions depending on Join/Sync/Heartbeat /OffsetCommit request/response.

For join group requests under dynamic membership (without `group.instance.id` set), the handling logic will remain unchanged. If the broker version is not the latest (< v4), the join group request shall be downgraded to v3.

Leave Group Logic Change

On server side, broker will keep handling leave group request <= v3 as before. We extended the LeaveGroupRequest API with a new tuple list which pairs `group.instance.id` to `member.id`. The reason to include `member.id` list instead of solely adding a `group.instance.id` list is to move LeaveGroupRequest towards a more consistent batch API in long term. The processing rules are following:

1. For static member, `group.instance.id` must be provided. Client could optionally provide a `member.id` when `group.instance.id` is configured non-null. If `member.id` is provided, the member will only be removed if the `member.id` matches. Otherwise, only the `group.instance.id` is used. The `member.id` serves as a validation here, which currently will not be used (set to empty string) but potentially useful if we do fully automated removal process.
2. For leave group requests under dynamic membership, the member will apply a singleton list of one tuple containing a `member.id` that it is currently using, and a `group.instance.id` which is set to null string. If this is the case, we shall just remove the given dynamic member the same way as current leave group logic.
3. Error cases expected are:
   a. Some instance ids (non-null) are not found, which means the request is not valid (*UNKNOWN_MEMBER_ID*)
   b. A theoretical case would be that both `member.id` and `group.instance.id` are set to empty string. We shall expose error in the server log. If the entire batch request is configured with empty strings, UNKNOWN_MEMBER_ID error will be returned.

If the broker version is not the latest (< v4), the leave group request shall be downgraded to v3.

Command Line API for Membership Management

RemoveMemberFromGroup will remove given instances and trigger rebalance immediately, which is mainly used for fast scale down/host replacement cases (we detect consumer failure faster than the session timeout). This API will first send a FindCoordinatorRequest to locate the correct broker, and initiate a *LeaveGroupRequest* to target broker hosting that coordinator.

The coordinator will decide whether to take this metadata change request based on its status on runtime. Error will be returned if

1. The broker is on an old version (*UNSUPPORTED_VERSION)*
2. Consumer group does not exist (*INVALID_GROUP_ID)*
3. Operator is not authorized. (*GROUP_AUTHORIZATION_FAILED*)
4. LeaveGroupRequest specific error

We need to enforce special access to these APIs for the end user who may not be in administrative role of Kafka Cluster. The solution is to allow a similar access level to the join group request, so the consumer service owner could easily use this API.

Scale Up

We will not plan to solve the scale up issue holistically within this KIP, since there is a parallel discussion about **Incremental Cooperative Rebalancing**, in which we will encode the "when to rebalance" logic at the application level, instead of at the protocol level.

For initial scale up, there is a plan to deprecate **group.initial.rebalance.delay.ms** (delivered in KIP-134) since we no longer needs it once static membership is delivered and the incremental rebalancing work is done.

Rolling Bounce

Currently broker accepts a config value called **rebalance timeout** which is provided by consumer **max.poll.intervals**. The reason we set it to poll interval is because consumer could only send request within the call of poll() and we want to wait sufficient time for the join group request. When reaching rebalance timeout, the group will move towards *COMPLETING_REBALANCE* stage and remove unjoined members. This is actually conflicting with the design of static membership, because those temporarily unavailable members will potentially reattempt the join group and trigger extra rebalances. Internally we would optimize this logic by having rebalance timeout only in charge of stopping *PREPARE_REBALANCE* stage, without removing non-responsive members immediately. There would not be a full rebalance if the lagging consumer sends a JoinGroupRequest within the session timeout.

So in summary, **the member will only be removed due to session timeout**. We shall remove it from both in-memory static `group.instance.id` map and member list.

Scale Down

Currently the scale down is controlled by session timeout, which means if user removes the over-provisioned consumer members it waits until session timeout to trigger the rebalance. This is not ideal and motivates us to change *LeaveGroupRequest* to be able to include a list of tuples of `group.instance.id` and `member.id` such that we could batch remove offline members and trigger rebalance immediately without them.

Fault-tolerance of Static Membership

To make sure we could recover from broker failure/coordinator transition, an in-memory `group.instance.id` map is not enough. We would reuse the _consumer_offsets topic to store the static member map information. When another broker takes over the leadership, it will load the static mapping info together.

Compatibility, Deprecation, and Migration Plan

Upgrade Process

The recommended upgrade process is as follow:

1. Upgrade your broker to include this KIP.
2. Upgrade your client to include this KIP.
3. Set `group.instance.id` to be unique for each consumer(or stream instance) and **`session.timeout.ms` to a reasonable number if necessary**
4. Rolling bounce the consumer group.

That's it! We believe that the static membership logic is compatible with the current dynamic membership, which means it is allowed to have static members and dynamic members co-exist within the same consumer group. This assumption could be further verified when we do some modeling of the protocol and unit test.

Downgrade Process

The downgrade process is also straightforward. End user could just

1. Unset `group.instance.id`, and change the session timeout to a smaller value if necessary
    a. For KStream user, unset `client.id` should do the work
2. Do a rolling bounce to switch back to dynamic membership. The dynamic member will be assigned with a new `member.id` which separates from previous generation.

The static membership metadata stored on broker will eventually be wiped out when the corresponding `member.id` reaches session timeout.

Switching from Static Member to Dynamic Member

A corner case is that although we don't allow static member to send *LeaveGroupRequest*, the broker could still see such a scenario where the *LeaveGroup Request* `member.id` points to an existing static member. The straightforward solution would be removing the member metadata all together including the static member info if the `group.instance.id` was left null corresponding. This approach ensures that downgrade process has no negative impact on the normal consumer operation, and avoids complicating the server side logic. In the long term, there could be potential use case to require static member to send LeaveGroupRequest, so we want to avoid changing the handling logic later.

Non Goal

We do have some offline discussions on handling leader rejoin case, where due to the possible topic assignment change (adding or removing topics), we still need to start a rebalance. However since the broker could also do the subscription monitoring work, we don't actually need to trigger rebalance on leader side blindly based on its rejoin request. This is a separate topic from 345 and we are tracking the discussion in this

KAFKA-7728 - Getting issue details... STATUS .

Update 2/12: It turns out that we could cover the leader rejoin case for static membership. When the leader joins with non-empty member.id, it indicates that the leader is joining with a different purpose than rolling bounce. If leader joins with UNKNOWN_MEMBER_ID, this alone is enough to suggest that it's doing a restart. We should expect no rebalance in this case.

Rejected Alternatives

In this pull request, we did an experimental approach to materialize member id(the identity given by broker, equivalent to the `group.instance.id` in proposal) on the instance local disk. This approach could reduce the rebalances as expected, which is the experimental foundation of KIP-345. However, KIP-345 has a few advantages over it:

1. It gives users more control of their `group.instance.id`; this would help for debugging purposes.
2. It is more cloud-/k8s-and-alike-friendly: when we move an instance from one container to another, we can copy the `group.instance.id` to the config files.
3. It doe not require the consumer to be able to access another dir on the local disks (think your consumers are deployed on AWS with remote disks mounted).
4. By allowing consumers to optionally specifying a `group.instance.id`, this rebalance benefit can be easily migrated to connect and streams as well which relies on consumers, even in a cloud environment.

Future Works

Beyond static membership we could unblock many stability features. We will initiate separate discussion threads once 345 is done. Examples are:

1. Pre-registration (proposed by Jason). Client user could provide a list of hard-coded `group.instance.id` so that the server could respond to scaling operations more intelligently. For example when we scale up the fleet by defining 4 new client instance ids, the server shall wait until all 4 new members to join the group before kicking out the rebalance, same with scale down.
2. Add hot standby hosts by defining `target.group.size` (proposed by Mayuresh). We shall keep some idle consumers within the group and when one of the active member go offline, we shall trigger hot swap due to the fact that current group size is smaller than `target.group.size`. With this change we might even not need to extend the session timeout since we should easily use the spare consumer to start working.
3. Add a script called *kafka-remove-member-from-group.sh* for end user to easily manipulate the consumer group. (proposed by Boyang) ***./bin /kafka-remove-member-from-group.sh --zookeeper localhost:2181 --broker 1 --group-id group-1 --group-instance-ids id_1,id_2 (comma separated id list)*** will immediately trigger a consumer group rebalance by transiting group state to *PREPARE_REBALANCE, while removing all the static members in the given list*.
4. Leverage `group.instance.id` for better generic sticky assignment (proposed by Jason). As we have discussed on the client side changes, for assignments relying on the natural order of `member.id`s (range/round robin/hash), the group.instance.id is preferred indicator than member.id because they persist through rolling bounce. Leader will choose to use `group.instance.id` over `member.id` if static membership is enabled.