Kafka Client-side Assignment Proposal

The new consumer currently relies on a server-side coordinator to negotiate the set of consumer processes that form the group and to assign the partitions to each member of the consumer group per some assignment strategy which group members must agree on. This provides assurance that the group will always have a consistent assignment and it enables the coordinator to validate that offsets are only committed from consumers that own the respective partition. However, it relies on the server having access to the code implementing the assignment strategy, which is problematic for two reasons:

1. First is just a matter of convenience. New assignment strategies cannot be deployed to the server without updating configuration and restarting the cluster. It can be a significant operational undertaking just to provide the capability to do this.
2. Different assignment strategies have different validation requirements. For example, with a redundant partitioning scheme, a single partition can be assigned to multiple consumers. This limits the ability of the coordinator to validate assignments, which is one of the main reasons for having the coordinator do the assignment in the first place.

If new assignment use cases were rare, this may be a viable solution, but we are already have a number of cases where more control over assignment is needed. For example:

- Co-partitioning: When joining two topics (in the context of KIP-28), it is necessary to assign the same partitions from more than one topic to the same consumer.
- Sticky partitioning: For stateful consumers, it is often best to minimize the number of partitions that have to be moved during a rebalance.
- Redundant partitioning: For some use cases, it is useful to assign each partition to multiple consumers. For e.g search indexers consuming from a Kafka topic need multiple replicas for the same partition. This would mean the same Kafka partition should be assigned to *n* consumer processes in such a search indexer application.
- Metadata-based assignment: In some cases, it is convenient to leverage consumer-local metadata to make assignment decisions. For example, if you can derive the rack from the FQDN of the Kafka brokers (which is common), then it would be possible to have rack-aware consumer groups if there was a way to communicate each consumer's rack to the partition assignment.

To address the problems pointed out above and support custom assignment strategies easily, we propose to move the assignment to the client. Specifically, we propose to separate the group management capability provided by the coordinator from partition assignment. We leave the coordinator to handle the former, while the latter is pushed into the consumer. This promotes separation of concerns and loose coupling.

More concretely, instead of the JoinGroup protocol returning each consumer's assignment directly, we modify the protocol to return the list of members in the group and have each consumer decide its assignment independently. This solves the deployment problem since it is typically an order of magnitude easier to update clients than servers. It also decouples the server from the needs of the assignment strategy, which allow us to support the above use cases without any server changes and provide some "future-proofing" for new use cases. For consumers, the join group protocol becomes more of an abstract group membership capability which, in addition to enabling assignment, can be used as a primitive to build other group management functions (such as leadership).

There are some disadvantages though. First, since the coordinator does not know the owners of a partition, it can no longer verify that offset commits come from the "right" consumer, which potentially opens the door to inconsistent processing. However, as mentioned above, the ability of the server to validate assignments (and therefore commits) would have to be handicapped anyway to support redundant partitioning. Also, with client-side assignment, d ebugging assignment bugs requires a little more work. Finding assignment errors may involve aggregating logs from each consumer in the group. In practice, the partitioning strategies used by most users will be simple and tested enough that such errors should be unlikely, but it is still a potential concern.

So far, we made an argument to separate group management from resource assignment. A significant benefit of this proposal is that it enables the group membership protocol to be used for other purposes. Below we outline all the use cases that would now be possible due to group management becoming a generic facility in the Kafka protocol.

1. The processor client (KIP-): Depending on the nature of your processing, your processor client might require a different partitioning strategy. For e. g. if your processing requires joins, it needs the co-partitioning assignment strategy for those topics and possibly a simple round robin for other topics.
2. Copycat: Here, you have a pool of worker processes in a copycat cluster that act as one large group. If one worker fails, the connector partitions that lived in that process need to be redistributed over the rest of the worker processes. Again, some connectors require a certain assignment strategy while a simple round robin works for others. The problem is the same - group management for a set of processes and assignment of resources amongst them that is really dictated by the application (copycat)
3. Single-writer producer: This use case may be a little out there since the transactional producer work hasn't quite shaped up. But the general idea is that you have multiple producers acting as a group, where only one producer is active and writing at any given point of time. If that producer fails, some other producer in the group becomes the single writer.
4. Consumer: A set of consumer processes need to be part of a group and partitions for the subscribed topics need to be assigned to each consumer processes, as dictated by the consumer application.

Given that there are several non-consumer use cases for a general group management protocol, we propose changing JoinGroupRequest and JoinGroupResponse such that it is not tied to consumer specific concepts.

Below we outline the changes needed to the protocol to make it more general and also the changes to the consumer API to support this.

Protocol

To support client-side assignment, we propose to split the group management protocol into two phases: group membership and state synchronization. The first phase is used to set the active members of the group and to elect a group leader. The second phase is used to enable the group leader to synchronize member state in the group (in other words to assign each member's state). From the perspective of the consumer, the first phase is used to collect member subscriptions, while the second phase is used to propagate partition assignments. The elected leader in the join group phase is responsible for setting the assignments for the whole group.

Below we describe the phases of this protocol in more detail.

Phase 1: Joining the Group

The purpose of the initial phase is to set the active members of the group. This protocol has similar semantics as in the initial consumer rewrite design. After finding the coordinator for the group, each member sends a JoinGroup request containing member-specific metadata. The join group request will park at the coordinator until all expected members have sent their own join group requests ("expected" in this case means all members that were part of the previous generation). Once they have done so, the coordinator randomly selects a leader from the group and sends JoinGroup responses to all the pending requests.

The JoinGroup request contains an array with the group protocols that it supports along with member-specific metadata. This is basically used to ensure compatibility of group member metadata within the group. The coordinator chooses a protocol which is supported by all members of the group and returns it in the respective JoinGroup responses. If a member joins and doesn't support any of the protocols used by the rest of the group, then it will be rejected. This mechanism provides a way to update protocol metadata to a new format in a rolling upgrade scenario. The newer version will provide metadata for the new protocol and for the old protocol, and the coordinator will choose the old protocol until all members have been upgraded.

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

JoinGroupResponse => ErrorCode GroupGenerationId GroupLeaderId MemberId Members
  ErrorCode           => int16
  GroupGenerationId   => int32
  GroupProtocol       => String
  GroupLeaderId       => String
  MemberId            => String
  Members             => [MemberId MemberMetadata]
    MemberId          => String
    MemberMetadata    => bytes
```

The JoinGroup response includes an array for the members of the group along with their metadata. This is only populated for the leader to reduce the overall overhead of the protocol; for other members, it will be empty. The is used by the leader to prepare member state for phase 2. In the case of the consumer, this allows the leader to collect the subscriptions from all members and set the partition assignment. The member metadata returned in the join group response corresponds to the respective metadata provided in the join group request for the group protocol chosen by the coordinator.

The error cases in this round of the protocol are similar to those described in the consumer rewrite design: Kafka 0.9 Consumer Rewrite Design.

Phase 2: Synchronizing Group State

Once the group members have been stabilized by the completion of phase 1, the active leader must propagate state to the other members in the group. This is used in the new consumer protocol to set partition assignments. Similar to phase 1, all members send SyncGroup requests to the coordinator. Once group state has been provided by the leader, the coordinator forwards each member's state respectively in the SyncGroup response. The message format is provided below:

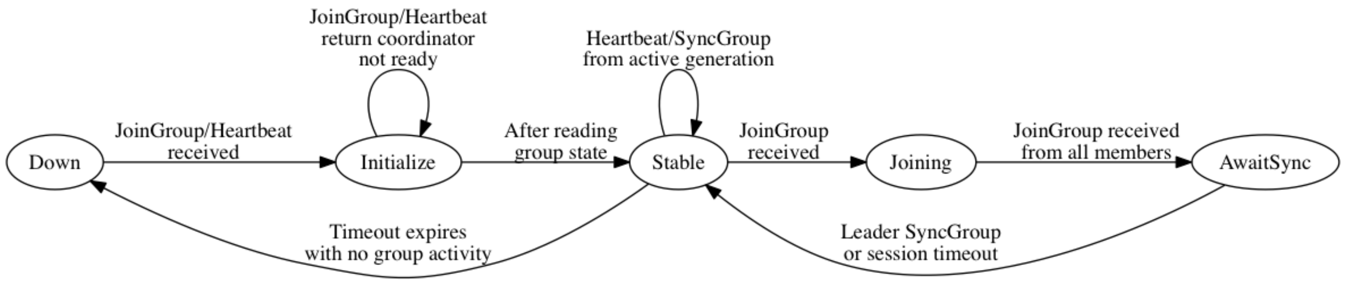
```
SyncGroupRequest => GroupId GroupGenerationId MemberId
  GroupId           => String
  GroupGenerationId => int32
  GroupState        => [MemberId MemberState]
    MemberId        => String
    MemberState     => bytes

SyncGroupResponse => ErrorCode MemberState
  ErrorCode         => int16
  MemberState       => bytes
```

The leader sets member states in the `GroupState` field. For followers, this array must be empty. Once the coordinator has received the group state from the leader, it can unpack each member's state and send it in the `MemberState` field of the SyncGroup response.

Coordinator State Machine

The coordinator maintains a state machine for each group with the following states:

- Down: There are no active members and group state has been cleaned up.
- Initialize: In this state, the coordinator reads group data from Zookeeper (or some other storage) in order to transition groups from failed coordinators. Any heartbeat or join group requests are returned with an error indicating that the coordinator is not ready yet.
- Stable: In this state, the coordinator either has an active generation or has no members and is awaiting the first JoinGroup. Heartbeats are accepted from members in this state and are used to keep group members active or to indicate that they need to join the group.
- Joining: The coordinator has received a JoinGroup request from at least one member and is awaiting JoinGroup requests from the rest of the group. Heartbeats or SyncGroup requests in this state return an error indicating that a rebalance is in progress.
- AwaitingSync: The join group phase has completed (i.e. all expected members of the group have sent JoinGroup requests) and the coordinator is awaiting group state from the leader. Unexpected coordinator requests return an error indicating that a rebalance is in progress.


JoinGroup/Heartbeat return coordinator not ready
Heartbeat/SyncGroup from active generation
JoinGroup/Heartbeat received
After reading group state
JoinGroup received
JoinGroup received from all members
Down
Initialize
Stable
Joining
AwaitSync
Timeout expires with no group activity
Leader SyncGroup or session timeout

Note that the generation is incremented on successful completion of the first phase (Joining). Before this phase completes, the old generation has an opportunity to do any necessary cleanup work (such as commit offsets in the case of the new consumer). Upon transition to the AwaitSync state, the coordinator begins a timer for each member according to their respective session timeouts. If the timeout expires for any member, then the coordinator must trigger a rebalance.

The group leader is responsible for synchronizing state for the group upon completion of the Joining state. If the leader's session timeout expires before the coordinator has received the leader's SyncGroup, then the generation becomes invalid and the members must rejoin. It is also possible for the leader to transmit an error in the SyncGroup request. In this case also, the generation becomes invalid and the error can be propagated to the other members of the group.

Note that the transition from AwaitSync to Stable occurs only when the leader's SyncGroup has been received. It is possible that the SyncGroup from followers may therefore arrive either in the AwaitSync state or in the Stable state. If the former, then the coordinator will park the request until the SyncGroup from the leader has been received (or its timeout has expired). If the latter, then the coordinator can respond to the SyncGroup request immediately using the leader's synchronized state. Clearly this requires the coordinator to store this state at least for the duration of the max session timeout in the group. It is also be possible that the member fails before collecting its state: in this case, the member's session timeout will expire and the group will rebalance.

Consumer Embedded Protocol

Above we outlined the generalized JoinGroup protocol that the consumer will use. Next we show how we will implement consumer semantics on top of this protocol. Other use cases for the join group protocol would be implemented similarly.

The two phases of the group protocol correspond to subscription and assignment for the new consumer. Each member of the group submits their subscription as member metadata. The leader of the group collects all subscription in its `JoinGroup` response and sends the assignment as member state in `SyncGroup`. There are several advantages to having a single assignor:

1. Since the leader makes the assignment for the full group, it is the single source of truth for the metadata used in its decision making. This avoids the need to synchronize metadata among all members that is required in a multi-assignor approach.
2. The leader of the group can enforce its own policy for controlling the rate of rebalancing. It doesn't have to rebalance after every metadata change, but can "batch" changes together to reduce the impact of metadata churn.
3. The leader is the only member that needs to receive the metadata from all members of the group. This reduces the overhead of the protocol.

The group protocol used by the consumer in the JoinGroup request corresponds to the assignment strategy that the leader will use to determine partition assignment. This allows the consumer to upgrade from one assignment strategy to another without downtime. The metadata corresponding to the assignment strategy can be strategy-specific, but generally it will include the group subscriptions for the member. The state returned to members in the SyncGroup will include the partitions assigned to that member.

For all assignment strategies, group members provide their subscriptions as an array of strings. This subscription can either be a list of topics or regular expressions (TODO: do we need distinguisher field to tell the difference? how about regex compatibility?). Partition assignments are provided in the SyncGroup response as an array of topics and partitions. The protocol supports custom data in both the subscription and assignment as a generic array of bytes to allow for custom assignor implementations. For example, a rack-aware assignor will generally need to propagate the rackId of each member to the leader in its subscription so that it can take it into account for assignment.

```
ProtocolType => "consumer"

GroupProtocol  => AssignmentStrategy
  AssignmentStrategy => String

MemberMetadata => Version Subscription AssignmentStrategies
  Version      => int16
  Subscription => Topics UserData
    Topics     => [String]
    UserData     => Bytes

MemberState => Version Assignment
  Version           => int16
  Assignment        => TopicPartitions UserData
    TopicPartitions => [Topic Partitions]
      Topic      => String
      Partitions => [int32]
    UserData     => Bytes
```

Protocol: Briefly, this is how the protocol works for the consumer.

1. Members JoinGroup with their respective subscriptions.
2. The leader collects member subscriptions from its JoinGroup response and performs the group assignment.
3. All members (including the leader) send SyncGroup to find their assignment.
4. Once created, there are two cases which can trigger reassignment:
    a. Topic metadata changes which have no impact on subscriptions cause resync. The leader computes the new assignment and sends SyncGroup.
    b. Membership or subscription changes cause rejoin.

Rolling Upgrades: To support rolling upgrades without downtime, there are two cases to consider:

1. Changes affecting subscription: the protocol directly supports differing subscriptions, so there is no need for special handling. Members will only be assigned partitions compatible with their subscription.
2. Assignment strategy changes: to support a change to the assignment strategy, new versions must enable support both for the old assignment strategy and the new one. The coordinator will choose the old assignment strategy until all members have been updated. Then it will choose the new strategy. This preference is implicit in the order of the strategies in the JoinGroup request.

Handling Coordinator Failures: This proposal largely shares the coordinator failure cases and recovery mechanism from the initial protocol documented in Kafka 0.9 Consumer Rewrite Design. The recovery process depends on whether group state is persisted (e.g. in Zookeeper). With no persistence, then group members will generally have to rejoin the group when the new coordinator becomes active. Below, we assume persistence and show how failures are treated at the various stages of the protocol.

1. All members send JoinGroup.
2. Generation metadata is persisted.
3. All members send SyncGroup.
4. Sync metadata is persisted.

Coordinator failures at these steps are handled in the following ways:

1. If the coordinator fails before all members have joined the group or before group metadata has been persisted, then all members will resend their JoinGroup requests once the new coordinator is stable.
2. The coordinator may fail after group metadata has been persisted, but before all members of the group have received a response to their JoinGroup requests. The problem here is that once the new coordinator is stable, the leader may try to immediately synchronize while other members are still trying to join. However, when the coordinator receives a JoinGroup request from any member, it must abort any active synchronization and force all members to rejoin.
3. If the coordinator fails before a pending synchronization has been persisted, then all members will re-initiate the SyncGroup once the new coordinator is ready.
4. If the coordinator fails after the metadata has been persisted, but before all members have received the SyncGroup response, then those members will initiate SyncGroup upon failover. Assuming synchronized state is persisted, then the coordinator can return that member's state immediately without forcing other members to resync. If it is not persisted, then a full group resync is required.

Other Interesting Cases:

- **Leader Failures**: The leader of each group is responsible for initiating group synchronization when topic metadata changes. A leader failure is detected by the coordinator through the expiration of its session timeout. The coordinator will respond by forcing all members to rejoin, which will allow a new leader to be elected.

- **Assignment Failure**: As mentioned above, there are several ways that the synchronization/assignment phase can fail. Generally, they are handled by having group members rejoin the group. The most interesting case is when the leader encounters an unrecoverable error when it computes the group's assignment. This could happen, for example, if group members don't agree on the assignment strategy to use. In this case, the assignment failure is forwarded to the broker which can then propagate it to awaiting members.
- **Subscription Change**: If a member changes its subscription, then it must force the group to be recreated by sending a JoinGroup request to the coordinator. This will cause the coordinator to reply to the other member's heartbeats with an error indicating that rejoin is needed, which will cause them to also send JoinGroup requests.
- **Topic Metadata Change:** The leader is responsible for detecting topic metadata changes which affect the group's subscription. When it finds a change, it can immediately compute the new assignment and initiate a SyncGroup with the coordinator.

TODO:

To support client-side assignment, we'd have to make the following changes:

1. Migrate existing assignment strategies from the broker to the client. Since the assignment interface is nearly the same, this should be straightforward.
2. Modify client/server for the new join group protocol. Since we're not really changing the protocol (just the information that is passed through it), this should also be straightforward.
3. Remove offset validation from the consumer coordinator. Just a couple lines to remove for this.
4. Add support for assignment versioning (if we decide we need it). Depending on what we do, may or may not be trivial.