Kafka 0.9 Consumer Rewrite Design

WARN: This is an obsolete design. The design that's implemented in Kafka 0.9.0 is described in this wiki.

Motivation

The motivation for moving to a new set of consumer client APIs with broker side co-ordination is laid out here.

Consumer API

The proposed consumer APIs are here. Several API usage examples are documented here.

Consumer HowTo

This wiki has design details on the new consumer. This level of detail may be too verbose for people who are just trying to write a non-java consumer client using the new protocol. This wiki provides a step by step guide for writing a non-java 0.9 client.

Group management protocol

Rebalancing is the process where a group of consumer instances (belonging to the same group) co-ordinate to own a mutually exclusive set of partitions of topics that the group is subscribed to. At the end of a successful rebalance operation for a consumer group, every partition for all subscribed topics will be owned by a single consumer instance within the group. The way rebalancing works is as follows. Every broker is elected as the coordinator for a subset of the consumer groups. The co-ordinator broker for a group is responsible for orchestrating a rebalance operation on consumer group membership changes or partition changes for the subscribed topics. It is also responsible for communicating the resulting partition ownership configuration to all consumers of the group undergoing a rebalance operation.

Consumer

1. On startup or on co-ordinator failover, the consumer sends a ConsumerMetadataRequest to any of the brokers in the bootstrap.brokers list. In the ConsumerMetadataResponse, it receives the location of the co-ordinator for it's group.
2. The consumer connects to the co-ordinator and sends a HeartbeatRequest. If an IllegalGeneration error code is returned in the HeartbeatRespons , it indicates that the co-ordinator has initiated a rebalance. The consumer then stops fetching data, commits offsets and sends a JoinGroupReque

st to it's co-ordinator broker. In the JoinGroupResponse, it receives the list of topic partitions that it should own and the new generation id for it's group. At this time, group management is done and the consumer starts fetching data and (optionally) committing offsets for the list of partitions it owns.
3. If no error is returned in the HeartbeatResponse, the consumer continues fetching data, for the list of partitions it last owned, without interruption.

Co-ordinator

1. In steady state, the co-ordinator tracks the health of each consumer in every group through it's failure detection protocol.
2. Upon election or startup, the co-ordinator reads the list of groups it manages and their membership information from zookeeper. If there is no previous group membership information, it does nothing until the first consumer in some group registers with it.
3. Until the co-ordinator finishes loading the group membership information for all groups that it is responsible for, it returns the CoordinatorStartupNotComplete error code in the responses of HearbeatRequests, OffsetCommitRequests and JoinGroupRequests. The consumer then retries the request after some backoff.
4. Upon election or startup, the co-ordinator also starts failure detection for all consumers in a group. Consumers that are marked dead by the co-ordinator's failure detection protocol are removed from the group and the co-ordinator triggers a rebalance operation for the consumer's group.
5. Rebalance is triggered by returning the IllegalGeneration error code in the HeartbeatResponse. Once all alive consumers re-register with the co-ordinator via JoinGroupRequests, it communicates the new partition ownership to each of the consumers in the JoinGroupResponse, thereby completing the rebalance operation.
6. The co-ordinator tracks the changes to topic partition changes for all topics that any consumer group has registered interest for. If it detects a new partition for any topic, it triggers a rebalance operation (as described in #5 above). It is currently not possible to reduce the number of partitions for a topic. The creation of new topics can also trigger a rebalance operation as consumers can register for topics before they are created.

Failure detection protocol

The consumer specifies a session timeout in the JoinGroupRequest that it sends to the co-ordinator in order to join a consumer group. When the consumer has successfully joined a group, the failure detection process starts on the consumer as well as the co-ordinator. The consumer initiates periodic heartbeats (HeartbeatRequest), every session.timeout.ms/heartbeat.frequency to the co-ordinator and waits for a response. If the co-ordinator does not receive a HeartbeatRequest from a consumer for session.timeout.ms, it marks the consumer dead. Similarly, if the consumer does not receive the HeartbeatResponse within session.timeout.ms, it assumes the co-ordinator is dead and starts the co-ordinator rediscovery process. The heartbeat.frequency is a consumer side config that determins the interval at which the consumer sends a heartbeat to the co-ordinator. The heartbeat.frequency puts a lower bound on the rebalance latency since the co-ordinator notifies a consumer regarding a rebalance operation through the heartbeat response. So, it is important to set it to a reasonably high value especially if the session.timeout.ms is relatively large. At the same time, care should be taken to not set the heartbeat.frequency too high since that would cause overhead due to request overload on the brokers. Here is the protocol in more detail -

1. After receiving a ConsumerMetadataResponse or a JoinGroupResponse, a consumer periodically sends a HeartbeatRequest to the coordinator every session.timeout.ms/heartbeat.frequency milliseconds.
2. Upon receiving the HeartbeatRequest, coordinator checks the generation number, the consumer id and the consumer group. If the consumer specifies an invalid or stale generation id, it send an IllegalGeneration error code in the HeartbeatResponse.
3. If the coordinator does not receive a HeartbeatRequest from a consumer at least once in session.timeout.ms, it marks the consumer dead and triggers a rebalance process for the group.
4. If the consumer does not receive a HeartbeatResponse from the coordinator after session.timeout.ms or finds the socket channel to the coordinator to be closed, it treats the coordinator as failed and triggers the co-ordinator re-discovery process.

Note that on coordinator failover, the consumers may discover the new coordinator before or after the new coordinator has finished the failover process including loading the consumer group metadata from ZK, etc. In the latter case, the new coordinator will just accept its heartbeat request as normal; in the former case, the new coordinator may reject its request, causing it to re-dicover the co-ordinator and re-connect again, which is fine. Also, if the consumer connects to the new coordinator too late, the co-ordinator may have marked the consumer dead and will be treat the consumer like a new consumer and trigger a rebalance.

State diagram

Consumer

Here is a description of the states -

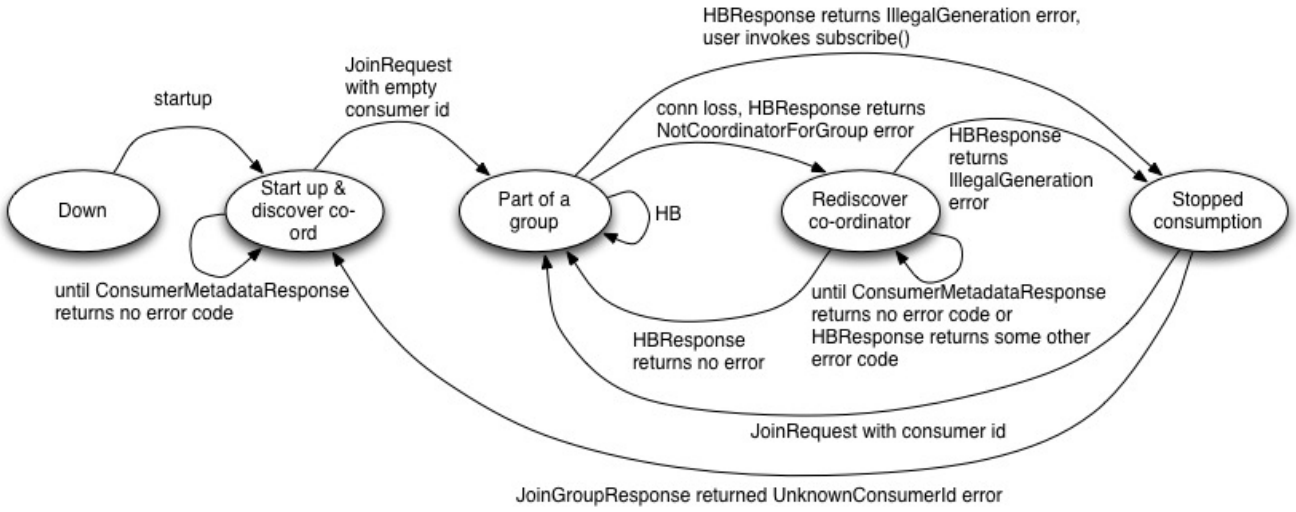
Down - The consumer process is down

Startup up & discover co-ordinator - In this state, the consumer discovers the co-ordinator for it's group. The consumer sends a JoinGroupRequest (with no consumer id) once it discovers the co-ordinator. The JoinGroupRequest can receive InconsistentPartitioningStrategy error code if some consumers in the same group specify conflicting partition assignment strategies. It can receive an UnknownPartitioningStrategy error code if the friendly name of the strategy in the JoinGroupRequest is unknown to the brokers. In this case, the consumer is unable to join a group.

Part of a group - In this state, the consumer is part of a group if it receives a JoinGroupResponse with no error code, a consumer id and the generation id for it's group. In this state, the consumer sends a HeartbeatRequest. Depending on the error code received, it either stays in this state or moves to Stopped Consumption or Rediscover co-ordinator.

Re-discover co-ord - In this state, the consumer does not stop consumption but tries to re-discover the co-ordinator by sending a ConsumerMetadataRequest and waiting for a response as long as it gets one with no error code.

Stopped consumption - In this state, the consumer stops consumption and commits offsets, until it joins the group again



Co-ordinator

Following is a state diagram that describes the state changes on the co-ordinator for a particular group.

Here is a description of the states -

Down - The co-ordinator is dead or demoted

Catch up - In this state, the co-ordinator is elected but not ready to serve requests

Ready - In this state, the newly elected co-ordinator has finished loading the group metadata for all groups that it is responsible for

Prepare for rebalance - In this state, the co-ordinator sends the IllegalGeneration error in the HeartbeatResponse for all consumers in the group and waits for the consumers to send it a JoinGroupRequest

Rebalancing - In this state, the co-ordinator has received a JoinGroupRequest from the consumers in the current generation and it increments the group generation id, assigns consumer ids where required and does the partition assignment

Steady - In the steady state, the co-ordinator accepts OffsetCommitRequests and heartbeats from all consumers in every group

Consumer id assignment

1. After startup, a consumer learns it's consumer id in the very first JoinGroupResponse it receives from the co-ordinator. From that point onwards, the consumer must include this consumer id in every HeartbeatRequest and OffsetCommitRequest it sends to the co-ordinator. If the co-ordinator receives a HeartbeatRequest or an OffsetCommitRequest with a consumer id that is different from the ones in the group, it sends an UnknownConsumer error code in the corresponding responses.
2. The co-ordinator assigns a consumer id to a consumer on a successful rebalance and sends it in the JoinGroupResponse. The consumer **can choose** to include this id in every subsequent JoinGroupRequest as well until it is shutdown or dies. **The advantage of including the consumer id in subsequent JoinGroupRequests is a lower latency on rebalance operations.** When a rebalance is triggered, the co-ordinator waits for all consumers in the previous generation to send it a JoinGroupRequest. The way it identifies a consumer is by it's consumer id. If the consumer chooses to send JoinGroupRequest with empty consumer id, the co-ordinator waits a full session.timeout.ms before it proceeds with the rest of the rebalance operation. It does this since there is no way to map the incoming JoinGroupRequest with a consumer in the absence of a consumer id. This puts a lower bound on the rebalance latency (session.timeout.ms). On the other hand, if the consumer sends it's consumer id on subsequent JoinGroupRequests, the co-ordinator can immediately identify the consumers and proceed with a rebalance once all the known consumers have sent a JoinGroupRequest.
3. The co-ordinator does consumer id assignment after it has received a JoinGroupRequest from all existing consumers in a group. At this point, it assigns a new id <group>-<uuid> to every consumer that did not have a consumer id in the JoinGroupRequest. The assumption is that such consumers are either newly started up or choose to not send the previously assigned consumer id.
4. If a consumer id is specified in the JoinGroupRequest but it does not match the ids in the current group membership, the co-ordinator sends an UnknownConsumer error code in the JoinGroupResponse and prevents the consumer from joining the group. This does not cause a rebalance operation for the rest of the consumers in the group, but also does not allow such a consumer to join an existing group.

Memory management

TBD

Request formats

For each consumer group, the coordinator stores the following information:

1) For each consumer group, the group metadata containing:

- List of topics the group subscribes to
- Group configs, including session timeout, etc.
- Consumer metadata for each consumer in the group. Consumer metadata includes hostname and consumer id.
- Current offsets for each consumed topic/partition.
- Partition ownership metadata, including the consumer-assigned-partitions map.

2) For each existing topic, a list of consumer groups that are currently subscribing to it.

It is assumed that all the following requests and responses have the common header. So the fields mentioned exclude the ones in the header.

ConsumerMetadataRequest

```
{
  GroupId                => String
}
```

ConsumerMetadataResponse

```
{
  ErrorCode              => int16
  Coordinator            => Broker
}
```

JoinGroupRequest

```
{
  GroupId                          => String
  SessionTimeout                   => int32
  Topics                           => [String]
  ConsumerId                       => String
  PartitionAssignmentStrategy => String
 }
```

JoinGroupResponse

```
{
  ErrorCode              => int16
  GroupGenerationId      => int32
  ConsumerId             => String
  PartitionsToOwn        => [TopicName [Partition]]
}
TopicName => String
Partition => int32
```

HeartbeatRequest

```
{
  GroupId                 => String
  GroupGenerationId       => int32
  ConsumerId              => String
}
```

HeartbeatResponse

```
{
  ErrorCode               => int16
}
```

OffsetCommitRequest (v1)

```
OffsetCommitRequest => ConsumerGroup GroupGenerationId ConsumerId [TopicName [Partition Offset TimeStamp
Metadata]]
  ConsumerGroup => string
  GroupGenerationId => int32
  ConsumerId => String
  TopicName => string
  Partition => int32
  Offset => int64
  TimeStamp => int64
  Metadata => string
```

Configs

Server side configs

This list is still in progress

max.session.timeout - The session timeout for any group should not be higher than this value to reduce overhead on the brokers. If it does, the broker sends a SessionTimeoutTooHigh error code in the JoinGroupResponse

partition.assignment.strategies - Comma separated list of properties that map the strategy's friendly name to the class that implements the strategy. This is used for any strategy implemented by the user and released to the Kafka cluster. By default, Kafka will include a set of strategies that can be used by the consumer. The consumer specifies a partitioning strategy in the JoinGroupRequest. It must use the friendly name of the strategy or it will receive an UnknownPartitionAssignmentStrategyException

Client side configs

The client side configs can be found here

Wildcard Subscription

With wildcard subscription (for example, whitelist and blacklist), the consumers are responsible to discover matching topics through topic metadata request. That is, its topic metadata request will contain an empty topic list, whose response then will return the partition info of all topics, it will then filter the topics that match its wildcard expression, and then update the subscription list using the subscribe() API. Again, if the subscription list has changed from the previous value, it will trigger rebalance for the consumer group by sending the JoinGroupRequest during the next poll().

Interesting scenarios to consider

Co-ordinator failover or connection loss to the co-ordinator

1. On co-ordinator failover, the controller elects a new leader for a subset of the consumer groups affected due to the co-ordinator failure. As part of becoming the leader for a subset of the offset topic partitions, the co-ordinator reads metadata for each group that it is responsible for, from zookeeper. The metadata includes the group's consumer ids, the generation id and the subscribed list of topics. Until the co-ordinator has read all

metadata from zookeeper, it returns the CoordinatorStartupNotComplete error code in HeartbeatResponse. It is illegal for a consumer to send a JoinGroupRequest during this time, so the error code returned to such a consumer is different (probably IllegalProtocolState).
2. If a consumer sends a ConsumerMetadataRequest to a broker before the broker has received the updated group metadata through the UpdateMetadataRequest from the controlller, the ConsumerMetadataResponse will return stale information about the co-ordinator. The consumer will receive NotCoordinatorForGroup error code on the heartbeat/commit offset responses. On receiving the NotCoordinatorForGroup error code, the consumer backs off and resends the ConsumerMetadataRequest.
3. The consumer does **not** stop fetching data during the co-ordinator failover and re-discovery process.

Partition changes for subscribed topics

1. The co-ordinator for a group detects changes to the number of partitions for the subscribed list of topics.
2. The co-ordinator then marks the group ready for rebalance and sends the IllegalGeneration error code in the HeartbeatResponse. The consumer then stops fetching data, commits offsets and sends a JoinGroupRequest to the co-ordinator.
3. The co-ordinator waits for all consumers to send it the JoinGroupRequest for the group. Once it receives all expected JoinGroupRequests, it increments the group's generation id in zookeeper, computes the new partition assignment and returns the updated assignment and the new generation id in the JoinGroupResponse. **Note that the generation id is incremented even if the group membership does not change.**
4. On receiving the JoinGroupResponse, the consumer stores the new generation id and consumer id locally and starts fetching data for the returned set of partitions. The subsequent requests sent by the consumer to the co-ordinator will use the new generation id and consumer id returned by the last JoinGroupResponse.

Offset commits during rebalance

1. If a consumer receives an IllegalGeneration error code, it stops fetching and commits existing offsets before sending a JoinGroupRequest to the co-ordinator.
2. The co-ordinator checks the generation id in the OffsetCommitRequest and rejects it if the generation id in the request is higher than the generation id on the co-ordinator. This indicates a bug in the consumer code.
3. The co-ordinator does **not** allow offset commit requests with generation ids older than the current group generation id in zookeeper either. This constraint is not a problem during a rebalance since until **all** consumers have sent a JoinGroupRequest, the co-ordinator does not increment the group's generation id. And from that point until the point the co-ordinator sends a JoinGroupResponse, it does not expect to receive any OffsetCommitRequests from any of the consumers in the group in the current generation. So the generation id on every OffsetCommitRequest sent by the consumer should always match the current generation id on the co-ordinator.
4. Another case worth discussing is when a consumer goes through a soft failure e.g. a long GC pause during a rebalance. If a consumer pauses for more than session.timeout.ms, the co-ordinator will not receive a JoinGroupRequest from such a consumer within session.timeout.ms. The co-ordinator marks the consumer dead and completes the rebalance operation by including the consumers that sent the JoinGroupRequest in the new generation.

Heartbeats during rebalance

1. Consumer periodically sends a HeartbeatRequest to the coordinator every session.timeout.ms/hearbeat.frequency milliseconds. If the consumer receives the IllegalGeneration error code in the HeartbeatResponse, it stops fetching, commits offsets and sends a JoinGroupRequest to the co-ordinator. Until the consumer receives a JoinGroupResponse, it does **not** send any more HearbeatRequests to the co-ordinator.
2. A higher heartbeat.frequency ensures lower latency on a rebalance operation since the co-ordinator notifies a consumer of the need to rebalance only on a HeartbeatResponse.
3. The co-ordinator *pauses* failure detection for a consumer that has sent a JoinGroupRequest until a JoinGroupResponse is sent out. It restarts the hearbeat timer once the JoinGroupResponse is sent out and marks a consumer dead if it does not receive a HeartbeatRequest from that time for another session.timeout.ms milliseconds. The reason the co-ordinator stops depending on heartbeats to detect failures during a rebalance is due to a design decision on the broker's socket server - Kafka only allows the broker to read and process one outstanding request per client. This is done to make it simpler to reason about ordering. This prevents the consumer and broker from processing a heartbeat request at the same time as a join group request for the same client. Marking failures based on JoinGroupRequest prevents the co-ordinator from marking a consumer dead during a rebalance operation. Note that this does not prevent the rebalance operation from finishing if a consumer goes through a soft failure during a rebalance operation. If the consumer pauses before it sends a JoinGroupRequest, the co-ordinator will mark it dead during the rebalance and complete the rebalance operation by including the rest of the consumers in the new generation. If a consumer pauses after it sends a JoinGroupRequest, the co-ordinator will send it the JoinGroupResponse assuming the rebalance completed successfully and will restart it's heartbeat timer. If the consumer resumes before session.timeout.ms, consumption starts normally. If the consumer pauses for session.timeout.ms after that, then it is marked dead by the co-ordinator and it will trigger a rebalance operation.
4. The co-ordinator returns the new generation id and consumer id only in the JoinGroupResponse. Once the consumer receives a JoinGroupResponse, it sends the next HeartbeatRequest with the new generation id and consumer id.

Co-ordinator failure during rebalance

A rebalance operation goes through several phases -

1. Co-ordinator receives notification of a rebalance - either a zookeeper watch fires for a topic/partition change or a new consumer registers or an existing consumer dies.
2. Co-ordinator initiates a rebalance operation by sending an IllegalGeneration error code in the HeartbeatResponse
3. Consumers send a JoinGroupRequest
4. Co-ordinator increments the group's generation id in zookeeper and writes the new partition ownership in zookeeper
5. Co-ordinator sends a JoinGroupResponse

Co-ordinator can fail at any of the above phases during a rebalance operation. This section discusses how the failover handles each of these scenarios.

1. If the co-ordinator fails at step #1 after receiving a notification but not getting a chance to act on it, the new co-ordinator has to be able to detect the need for a rebalance operation on completing the failover. As part of failover, the co-ordinator reads a group's metadata from zookeeper, including the list of topics the group has subscribed to and the previous partition ownership decision. If the # of topics or # of partitions for the

subscribed topics are different from the ones in the previous partition ownership decision, the new co-ordinator detects the need for a rebalance and initiates one for the group. Similarly if the consumers that connect to the new co-ordinator are different from the ones in the group's generation metadata in zookeeper, it initiates a rebalance for the group.

2. If the co-ordinator fails at step #2, it might send a HeartbeatResponse with the error code to some consumers but not all. Similar to failure #1 above, the co-ordinator will detect the need for rebalance after failover and initiate a rebalance again. If a rebalance was initiated due to a consumer failure and the consumer recovers before the co-ordinator failover completes, the co-ordinator will not initiate a rebalance. However, if **a ny** consumer (with an empty or unknown consumer id) sends it a JoinGroupRequest, it will initiate a rebalance for the entire group.
3. If a co-ordinator fails at step #3, it might receive JoinGroupRequests from only a subset of consumers in the group. After failover, the co-ordinator might receive a HeartbeatRequest from all alive consumers OR JoinGroupRequests from some. Similar to #1, it will trigger a rebalance for the group.
4. If a co-ordinator fails at step #4, it might fail after writing the new generation id and group membership in zookeeper. The generation id and membership information is written in one atomic zookeeper write operation. After failover, the consumer will send HeartbeatRequests to the new co-ordinator with an older generation id. The co-ordinator triggers a rebalance by returning an IllegalGeneration error code in the response that causes the consumer to send it a JoinGroupRequest. **Note that this is the reason why it is worth sending both the generation id as well as the consumer id in the HeartbeatRequest and OffsetCommitRequest**
5. If a co-ordinator fails at step #5, it might send the JoinGroupResponse to only a subset of the consumers in a group. A consumer that received a JoinGroupResponse will detect the failed co-ordinator while sending a heartbeat or committing offsets. At this point, it will discover the new co-ordinator and send it a heartbeat with the new generation id. The co-ordinator will send it a HeartbeatResponse with no error code at this point. A consumer that did **not** receive a JoinGroupResponse will discover the new co-ordinator and send it a JoinGroupRequest. This will cause the co-ordinator to trigger a rebalance for the group.

Slow consumers

Slow consumers can be removed from the group by the co-ordinator if it does not receive a heartbeat request for session.timeout.ms. Typically, consumers can be slow if their message processing is slower than session.timeout.ms, causing the interval between poll() invocations to be longer than session. timeout.ms. Since a heartbeat is only sent during the poll() invocation, this can cause the co-ordinator to mark a slow consumer dead. Here is how the co-ordinator handles a slow consumer -

1. If the co-ordinator does not receive a heartbeat for session.timeout.ms, it marks the consumer dead and breaks the socket connection to it.
2. At the same time, it triggers a rebalance by sending the IllegalGeneration error code in the HeartbeatResponse to the rest of the consumers in the group.
3. If the slow consumer sends a HeartbeatRequest **before** the co-ordinator receives a HeartbeatRequest from **any** of the rest of the consumers, it cancels the rebalance attempt and sends the HeartbeatResponses with no error code
4. If not, the co-ordinator proceeds with the rebalance and sends the slow consumer the IllegalGeneration error code as well.
5. Since the co-ordinator only waits for JoinGroupRequest from the "alive" consumers, it calls the rebalance complete once it receives join requests from the other consumers. If the slow consumer also happens to send a JoinGroupRequest, the co-ordinator includes it in the current generation if it has not already sent a JoinGroupResponse excluding this slow consumer.
6. If the co-ordinator has already sent a JoinGroupResponse, it let's the current rebalance round complete before triggering another rebalance attempt right after.
7. If the current round takes long, the slow consumer's JoinGroupResponse times out and it re-discovers the co-ordinator and resends the JoinGroupRequest to the co-ordinator.