Kafka Consumer Group Rebalances

Introduction

Consumer groups are an important characteristic of Kafka's distributed message processing for managing consumers and facilitating the ability to scale up applications. They group together consumers for any one topic, and the partitions within the topic are assigned across these consumers. Consumer group rebalance can be triggered by a number of factors as the participants of the group change, which leads to the reassignment of partitions across the consumers. During a rebalance message processing is paused, impacting throughput.

This is the first of a two part article that details the behaviour of consumer group rebalance with the Apache Java client. While many of the concepts are the same across different client libraries, there are also differences in terms of configuration options, rebalance behaviour, and supported rebalance strategies and features.

In this first part of the article the role of consumer groups, the consumer group rebalance, and the triggers that cause rebalances are covered. The configurations that impact both the duration of the rebalance and when a rebalance is triggered are detailed. In the second part the impact on the application's message processing during the rebalance is covered along with the rebalance strategies that can be applied. The options to reduce unnecessary rebalances and to mitigate the impacts of rebalances are explored.

Consumer Groups

When an application has a Kafka consumer implemented to consume messages from a topic, that consumer belongs to a consumer group. Within the consumer group, consumers are assigned topic partitions from which to consume. Group membership is managed on the broker side, and partition assignment is managed on the client side. The broker has no knowledge of what the resources are and how they are assigned amongst the consumers. This is a good example of why the Kafka client is considered a thick client. For more on this read about the role of the Kafka client.

The consumer is configured with a **group.id**, so that any other consumer instances with the same **group.id** will belong to the same consumer group. This facilitates the ability to scale up consumers, and this coupled with increasing the number of partitions in a topic provides a mechanism to increase message throughput.

The Group Coordinator manages the consumer group and the consumers. This is a Kafka component that lives on the broker side. It will make one consumer the lead, and this will be responsible for computing the topic partition assignments. These are returned to the Group Coordinator which then assigns the partitions to the consumers.

Given a single application instance, with a consumer with a **group.id** of 'foo' listening to a particular topic, and that topic has six partitions, then the consumer will poll for messages across all six partitions.

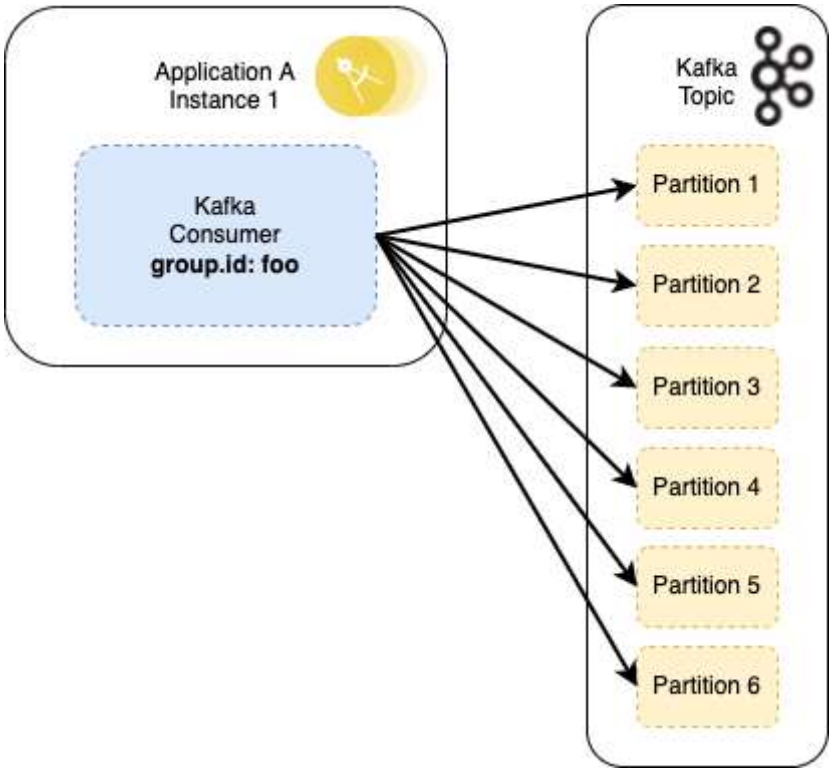

Figure 1: Single consumer group with one consumer

Now a second instance of the application is started. This therefore starts a second consumer instance with the same **group.id** of 'foo'. The second consumer instance sends a JoinGroup request to the Group Coordinator, and the partitions are reassigned across the consumer group to spread the load. With two members in the consumer group, three partitions are assigned to each consumer instance.

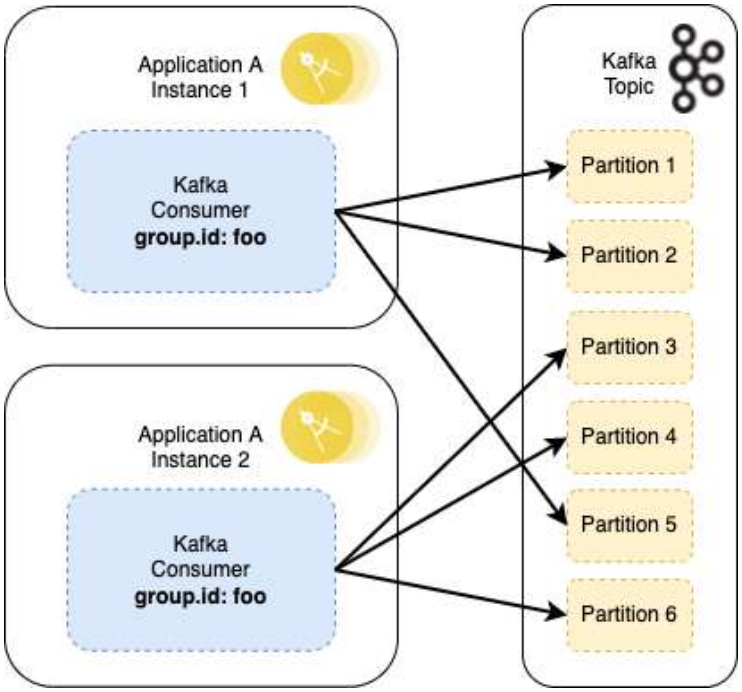

Figure 2: Single consumer group 'foo' with two consumers

Start a third application, and the partitions are again reassigned by the Group Coordinator, with each consumer now polling for messages from two partitions each.

If there are more consumer instances than partitions then those extra consumers will have no partitions assigned. A topic partition will only ever have one consumer listening to it from a given consumer group. So a consumer group composed of five consumers listening to a topic with three partitions will have two idle consumers.

If a consumer is started with a different **group.id** configured (as would be the case for a different service), and it is listening to the same topic, then this will be part of a separate consumer group. Its partition assignments are independent of those from any other consumer group.

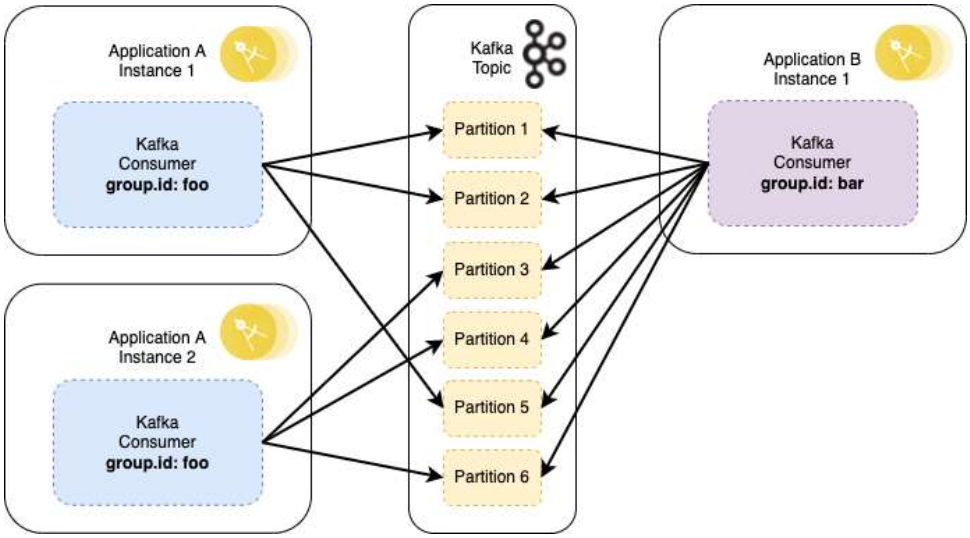

Figure 3: Two consumer groups 'foo' and 'bar'

Rebalance Triggers

There are several causes for a consumer group rebalance to take place. A new consumer joins a consumer group, an existing consumer leaves a consumer group, or the broker thinks a consumer may have failed. As well as these, any other need for resources to be reassigned will trigger a rebalance. An example is the creation of a topic where a consumer is configured with a pattern subscription that matches this topic name.

When a new consumer joins a consumer group it sends a JoinGroup request to the Group Coordinator on the broker. The topic partitions are then reassigned across all one or more consumers in the group. Likewise when a consumer leaves a group it notifies the Group Coordinator via a LeaveGroup request which again reassigns the topic partitions across the remaining consumers, if there are any.

When the Group Coordinator does not hear from a consumer within the expected timeframe, be it a heartbeat or the next poll() call, then it evicts the consumer from the group believing it may have failed. Once again the topic partitions are reassigned across any other consumers remaining in the group.

If a service has multiple consumers that subscribe to mutually exclusive topics but that share the same **group.id** then any rebalance triggered by any one consumer would still affect the other consumers in the group. In the following scenario **Consumer A** is subscribed to topic **abc**, whilst **Consumer B** is subscribed to topic **def**. They are in the same consumer group **foo**.

If **Consumer A** takes too long to process a batch and times out then it is removed from the consumer group triggering a rebalance. All partition assignments in the group are revoked and reassigned, including those for **Consumer B**.

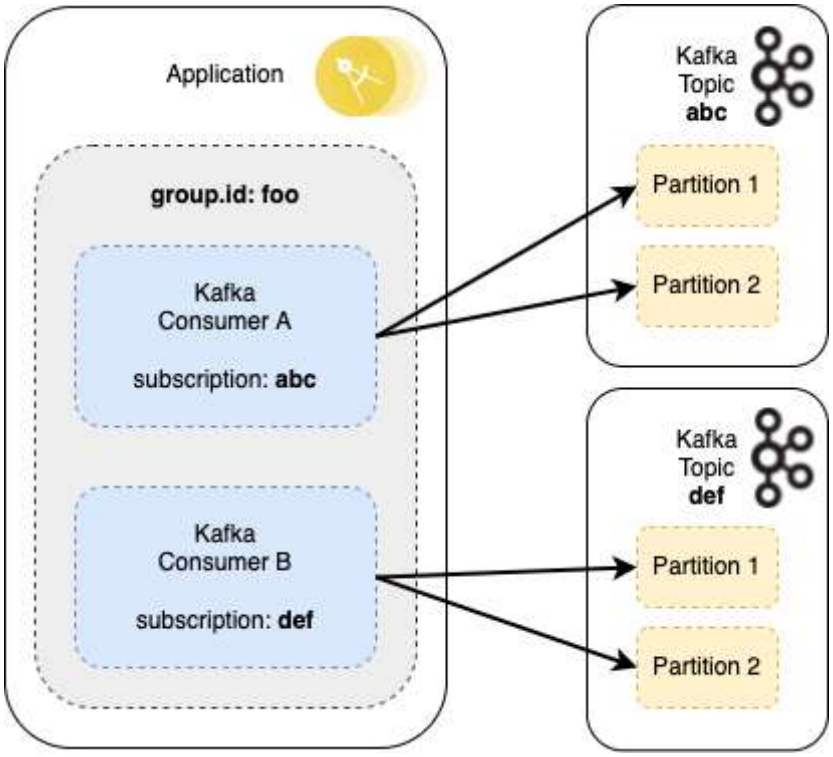

Figure 4: Consumer group spanning topics

When **Consumer A** eventually completes its poll and rejoins the consumer group, a further rebalance is triggered, and again all processing stops as partitions are revoked and reassigned. It can therefore be prudent to define separate consumer groups for consumers listening to different topics. e.g. *[service]-[topic]-consumer-group*.

Rebalance Configuration

Overview

For the Apache Java Kafka client the following are the key configurations on the consumer that impact how long rebalances can take to complete, and when a consumer may be considered to have failed by the broker causing a rebalance to be triggered.

Configuration	Description	Default
session.timeout.ms	The timeout to detect a consumer has failed. A heartbeat must be received by the Group Coordinator within this period.	45 seconds (from Kafka 3.0.0)
heartbeat.interval.ms	The interval between heartbeats sent by the consumer to the Group Coordinator. Used to ensure a consumer's session stays alive.	3 seconds
max.poll.interval.ms	The maximum time between poll invocations before a consumer is considered failed.	5 minutes

The following sections examine the impact of these configuration parameters.

Heartbeat and Session Timeout

The consumer sends periodic heartbeats to the Group Coordinator (which lives on the broker). This allows the Group Coordinator to monitor the health of the consumers in the group. A heartbeat must be received within the **session.timeout.ms**, and the heartbeats are sent based on the **heartbeat.interval.ms**. When the heartbeat is received by the Group Coordinator the **session.timeout.ms** resets, it responds to the consumer, and the next consumer heartbeat must be received within this reset timeout.

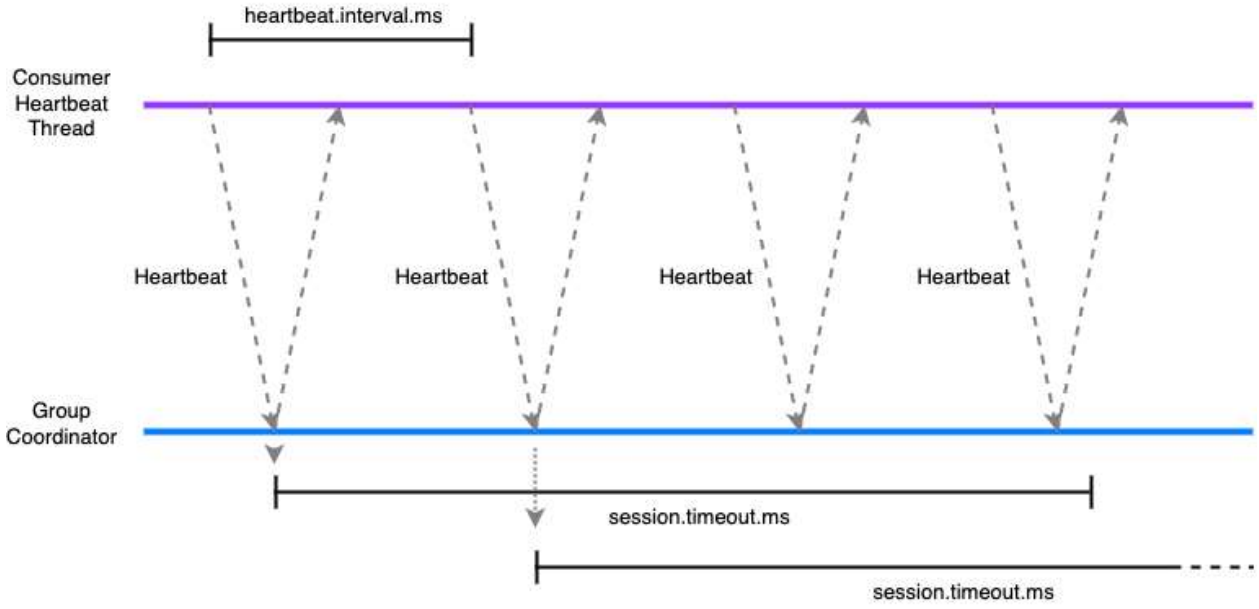

Figure 5: Consumer heart beating

It is recommended to configure the **heartbeat.interval.ms** to be no more than a third of **session.timeout.ms**. This ensures that if a heartbeat or two are lost due for example to a transient network issue, that the consumer is not considered to have failed. In this diagram two heartbeats are lost, but the third arrives before the session has timed out, so the Group Coordinator knows the consumer is still healthy.

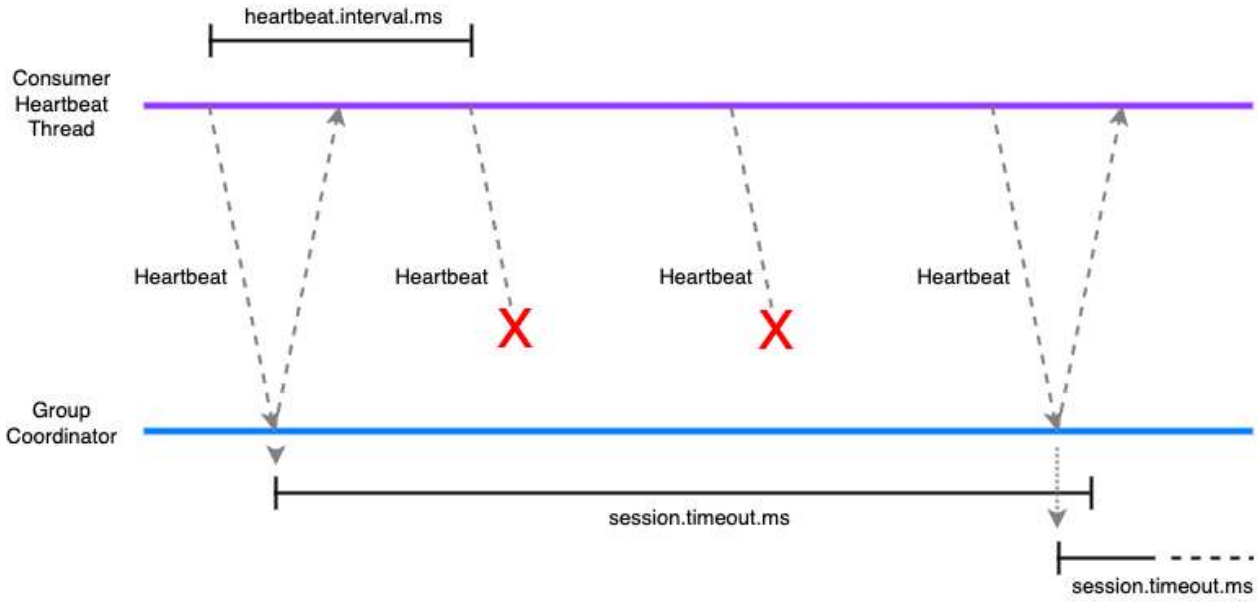

Figure 6: Failed heart beats

If the consumer does fail and stops heart beating then it is evicted from the consumer group once the session timeout expires, resulting in a consumer group rebalance.

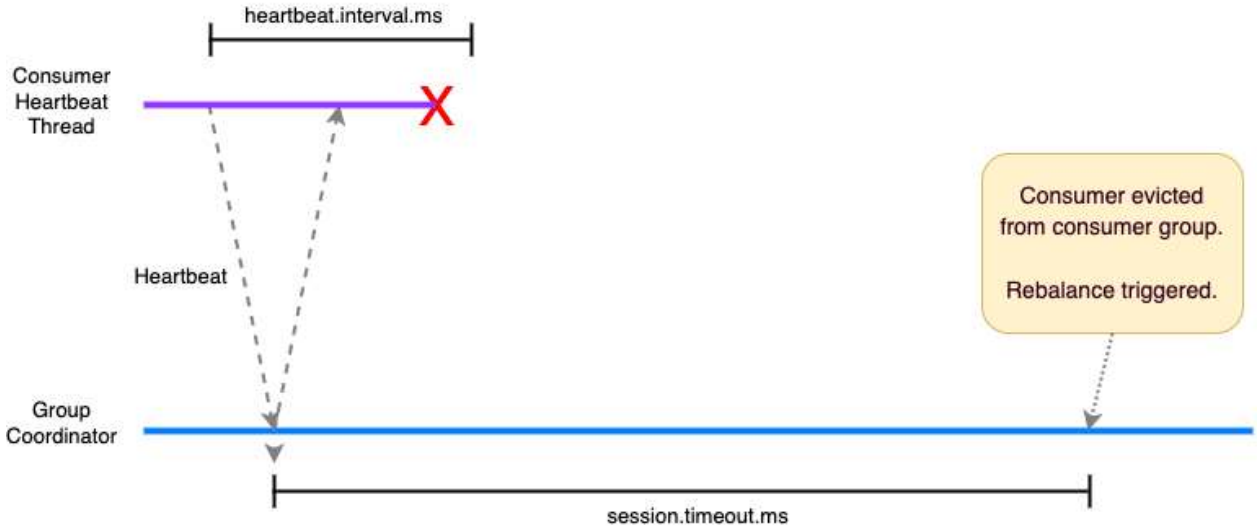

Figure 7: Consumer fails

Poll Interval

Heartbeats are performed on a separate thread to that of the main processing. The consumer polls its topic partitions on the main processing thread, and each call to poll() must happen within the configured **max.poll.interval.ms**. The following diagram adds the consumer

processing thread, showing the responsibility of this thread alongside that of the heartbeat thread.

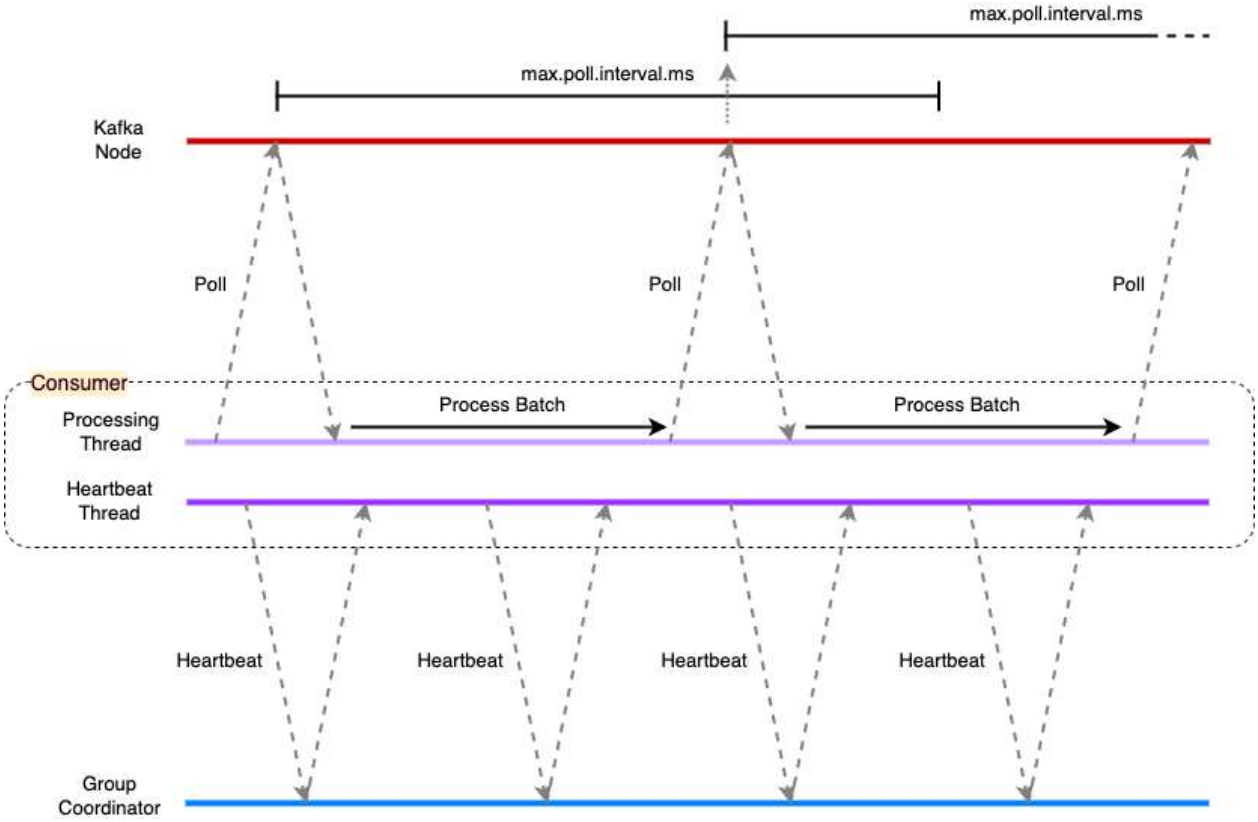

Figure 8: Consumer heart beating and polling

The first call to poll(), and any call to poll() that includes changes such as to partition assignments, results in the heartbeat thread being started. Each subsequent poll() call restarts the poll time, such that it has this full **max.poll.interval.ms** within which to complete.

The heartbeat thread checks the status of the consumer processing, and if the **max.poll.interval.ms** has been exceeded between polls then rather than a heartbeat it sends a LeaveGroup request. The Group Coordinator removes the consumer from the consumer group triggering a rebalance.

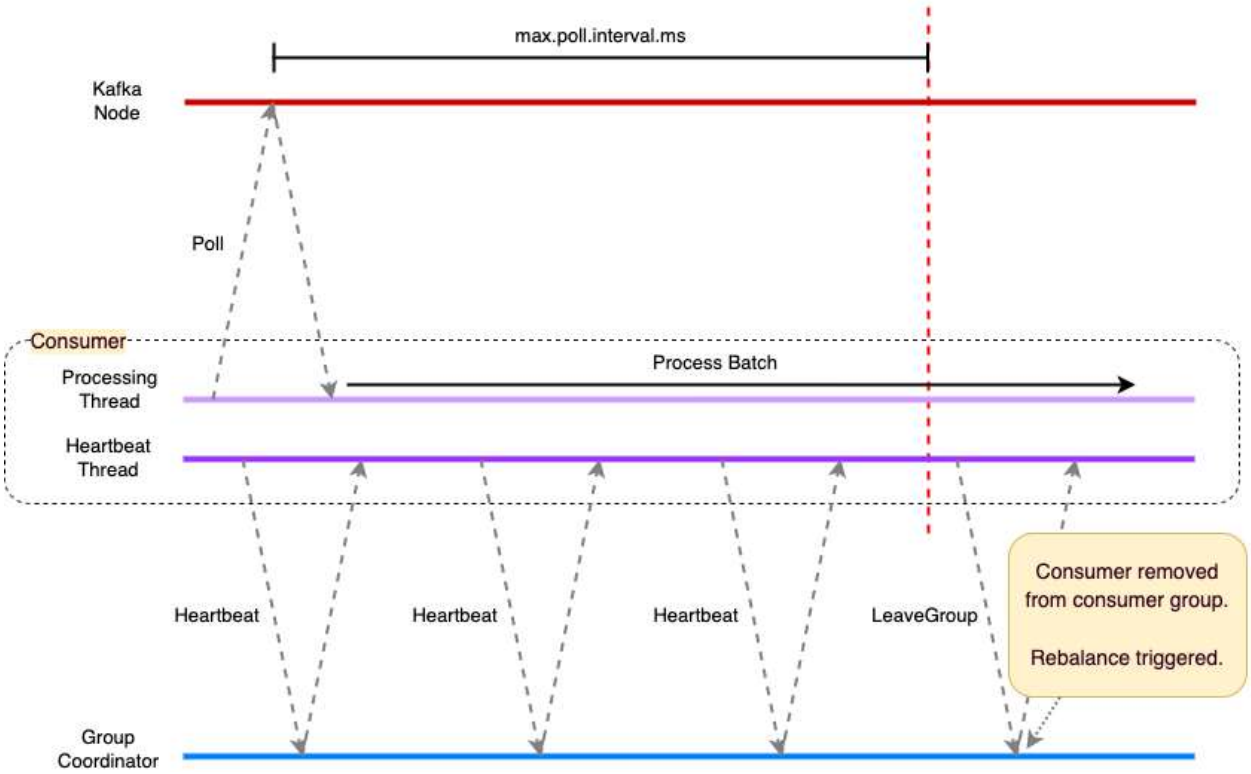

Figure 9: Consumer exceeds poll interval

When a rebalance is triggered the existing consumers will receive a response to their next heartbeat of 'Rebalance'. Each consumer has until the **max.poll.interval.ms** timeout to rejoin the group by calling poll(), as this triggers a JoinGroup request to the Group Coordinator. Note that for Kafka Connect a separate timeout is provided for this, **rebalance.timeout.ms**.

Configuring the **max.poll.interval.ms** therefore requires careful consideration. Set it too low and the risk is that the batch of messages consumed in a single poll are not processed in time leading to rebalancing and duplicate message delivery. Set the interval too high and it means that when a consumer does fail it takes longer before the broker is aware and the consumer's partitions are reassigned. During this processing the messages on the topic partitions assigned to the failed consumer are stuck.

Consumer Health

There are therefore two time outs to consider that have a bearing on when a consumer is considered healthy or to have failed and be evicted from a consumer group. If the main processing thread dies, leaving the heartbeat thread still running, the failure is detected by the **max.poll.interval.ms** being exceeded. If the whole application dies then this will be detected by no heartbeat being received within the **session.timeout.ms**.

The **max.poll.interval.ms** is essentially the main health check for the consumer processing. However by also utilising a heartbeat check on a separate thread it means that hard failures where the whole application has failed are detected more quickly.

From <https://www.linkedin.com/pulse/kafka-consumer-group-rebalance-1-2-rob-golder/>

This is the second in a two part article on Consumer Group Rebalance. In the first part consumer groups, rebalances, triggers for a rebalance, and the configuration options that impact rebalances were covered.
In this second part of the article the rebalance strategies available, the ability to reduce the number of unnecessary rebalances with Static Group Membership, and the risks to consider with rebalancing are all explored.

Rebalance Strategies

Eager Rebalance

With eager rebalancing (the default), when a consumer group rebalances, all processing by the consumers stops while the topic partitions are reassigned. This means the number of rebalances and their impact on an application is critical to understand as it can have a significant impact on throughput.
The following diagram illustrates the impact on an existing consumer group containing a single consumer when a new consumer joins the group, and the time outs in play. However it is applicable to all three rebalance trigger scenarios, whether the Group Coordinator has received a JoinGroup or a LeaveGroup request from a consumer, or it believes a consumer may have failed.

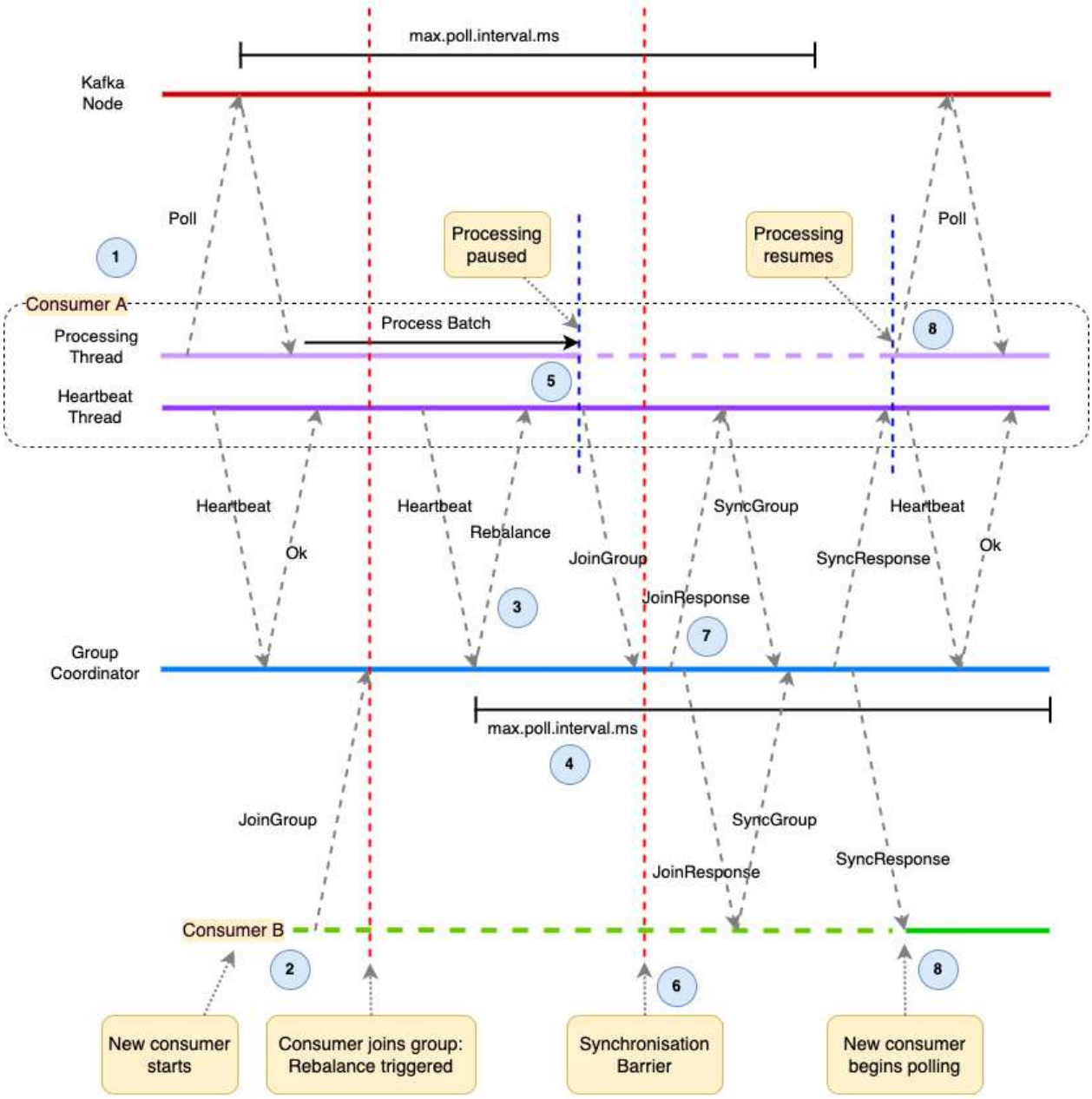

Figure 1: Eager Rebalance

1

Consumer A is polling the topic partition for messages, and heart beating to the Group Coordinator to tell it is healthy. The response to this heartbeat from the Group Coordinator is an acknowledgement of 'Ok', and the consumer continues its processing.

2

Consumer B starts and sends a JoinGroup request to the Group Coordinator triggering a rebalance. (Just the heartbeat thread for Consumer B is shown on the diagram for clarity).

3

The Group Coordinator responds to the next heartbeat it receives from its existing consumer, Consumer A, notifying it that a rebalance has begun.

4

Consumer A has until its **max.poll.interval.ms** to complete processing the messages from its current poll() and respond with its own JoinGroup request.

5

Consumer A's poll completes and now all message processing is paused. Its next call to poll() triggers the send of a JoinGroup request with the information on the topic subscriptions it is interested in to the Group Coordinator.

6

The Group Coordinator, being aware of all the consumers in the group, knows the point at which all existing topic partitions have been released and can be reassigned. The 'Synchronisation Barrier' has been reached as the consumer group has stabilised with two members.

7

The Group Coordinator sends JoinResponses to both consumers. One consumer is selected as leader and calculates the partition assignments. The consumers respond with SyncGroup requests. The group leader's SyncGroup request contains the computed partition assignments.

8

The Group Coordinator responds with a SyncResponse to each consumer notifying them of their assignments. Consumer A now resumes polling, and Consumer B begins polling.
The consumer group rebalance does not complete until all consumers have accepted their partition assignments. The diagram highlights the pause in processing during rebalance, and as the consumer group increases in size with more members the duration of this pause becomes more significant.

Incremental Rebalance

The larger a consumer group is, the longer it can take for a rebalance to take place. If the impact of eager consumer group rebalances stopping message processing while they are occurring is considered too great, then an Incremental Rebalance strategy could be adopted. (This is also known as Cooperative Rebalance). This time existing consumers that have been notified by the Group Coordinator that a rebalance is underway do not stop processing. Instead rebalancing occurs over two phases. As the consumers receive notification from the Group Coordinator that a rebalance has begun, the following now occurs:

1. The existing consumers send a JoinGroup request to the Group Coordinator, but continue processing messages from their assigned topic partitions.
2. The JoinGroup request contains their topic subscription information and the information on their current assignments.

3. Once the Group Coordinator receives JoinGroup Requests from all existing consumers (or they have timed out), it then sends JoinResponses to the consumers, and assigns a new group leader.
4. The new group leader responds with a SyncGroup request with the intended consumer to partition assignments.
5. The Group Coordinator notifies the consumers that must release partitions in a SyncResponse.
6. Only those consumers that need to revoke partitions actually stop processing in order for those partitions to be reassigned to another consumer.
7. New JoinGroup requests are sent from all the consumers to the Group Coordinator in a second round of the rebalance protocol with information on the partitions they still own and those they have revoked.
8. At this point the group has stabilised and the rebalance has reached the 'Synchronisation Barrier'. Partition assignment can now be completed.

Only those partitions that need to be reassigned are revoked. The other partitions are constantly owned by their consumers with no interruption to consumption of their messages.
The following diagram illustrates the incremental balance in action.

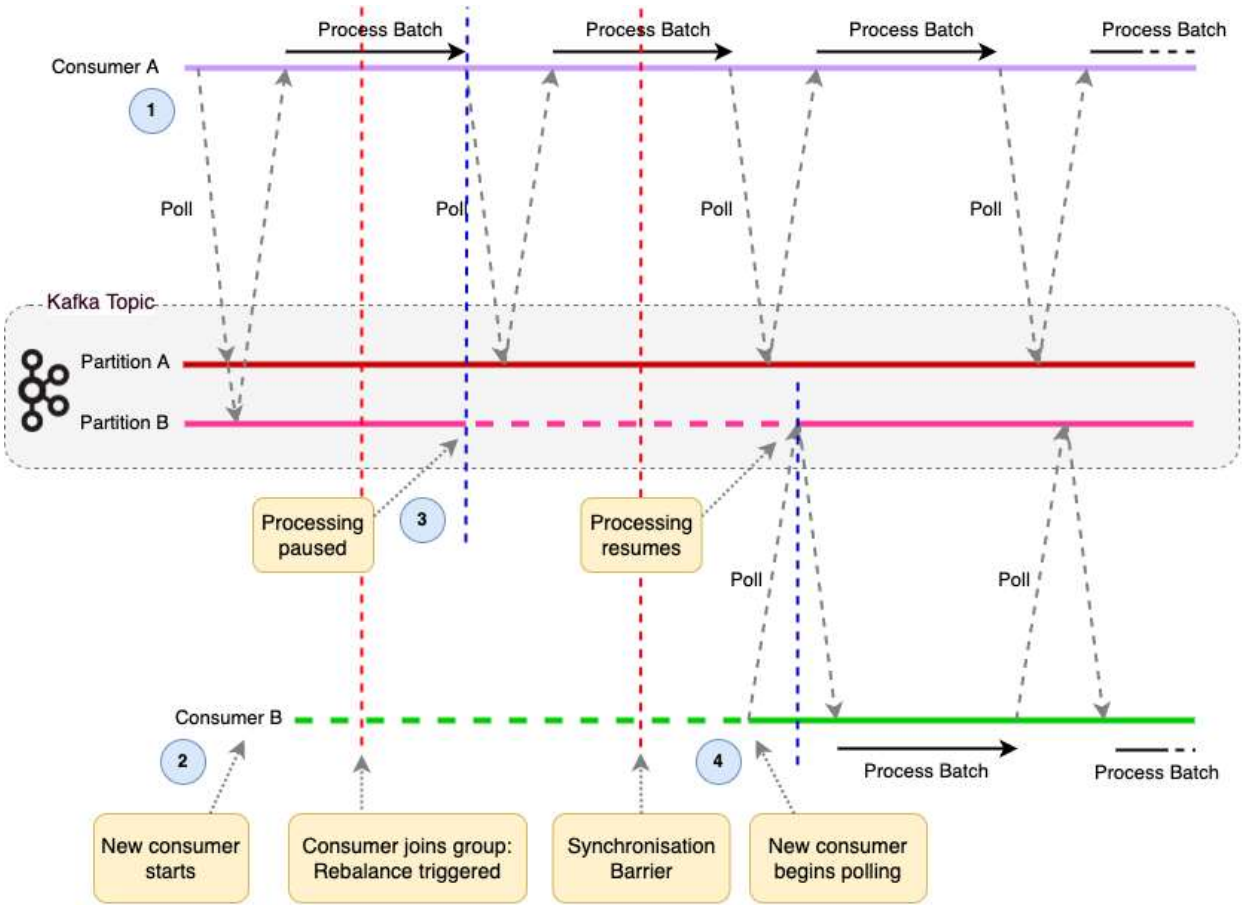

Figure 2: Incremental Rebalance

1

Consumer A is polling from two partitions when a second consumer.

2

Consumer B, starts and joins the group. This triggers the incremental rebalance.

3

Consumer A gives up its assignment of one of its partitions.

4

The partition is reassigned to Consumer B which begins consuming messages from it.
Meanwhile Consumer A does not stop processing from the other of the two partitions.
The diagram does not include the complexity around the heartbeating.
Incremental Rebalance takes two rounds of rebalancing to complete, so results in longer overall latency. However the impact of the rebalance is less severe to overall message processing.
Incremental Rebalance is configured by applying a **CooperativeStickyAssignor** to the consumer's **partition.assignment.strategy** setting.

Static Group Membership

The number of unnecessary rebalances, and hence the impact of rebalancing on throughput, can be reduced by using Static Group Membership. With the default rebalance protocol when a consumer starts it is assigned a new **member.id** (which is an internal Id in the Group Coordinator) irrespective of whether it is an existing consumer that has just restarted. Any consumer starting triggers a rebalance, and is assigned a new **member.id**. With this protocol the consumer cannot be re-identified as the same.
The Static Group Membership protocol introduces the ability to configure a unique **group.instance.id** on the consumer, marking it as a static member. The Group Coordinator maps this **group.instance.id** to the internal **member.id**. If a consumer dies and restarts it will send a JoinGroup request with this id to the Group Coordinator. In the scenario where the consumer shuts down it is not removed from the consumer group until its session times out based on the **session.timeout.ms**. When the consumer is restarted and rejoins the group, the Group Coordinator checks and finds the **group.instance.id** matches that of a static member it has registered in the consumer group. It therefore knows it is the same consumer instance and a rebalance is not triggered. The partitions that were assigned to that consumer are reassigned to it and processing of messages from those partitions now resumes. Meanwhile there was no interruption to the processing of messages on partitions assigned to other consumers.
The following diagram demonstrates static group membership. Two consumers belong to the same consumer group and have distinct **group.instance.id** values assigned. They are polling a partition each from the same topic. Consumer B stops and leaves the group, however a rebalance is not immediately triggered. The consumer rejoins the group before the **session.timeout.ms** times out and is reassigned its partition, ensuring no rebalance is required.

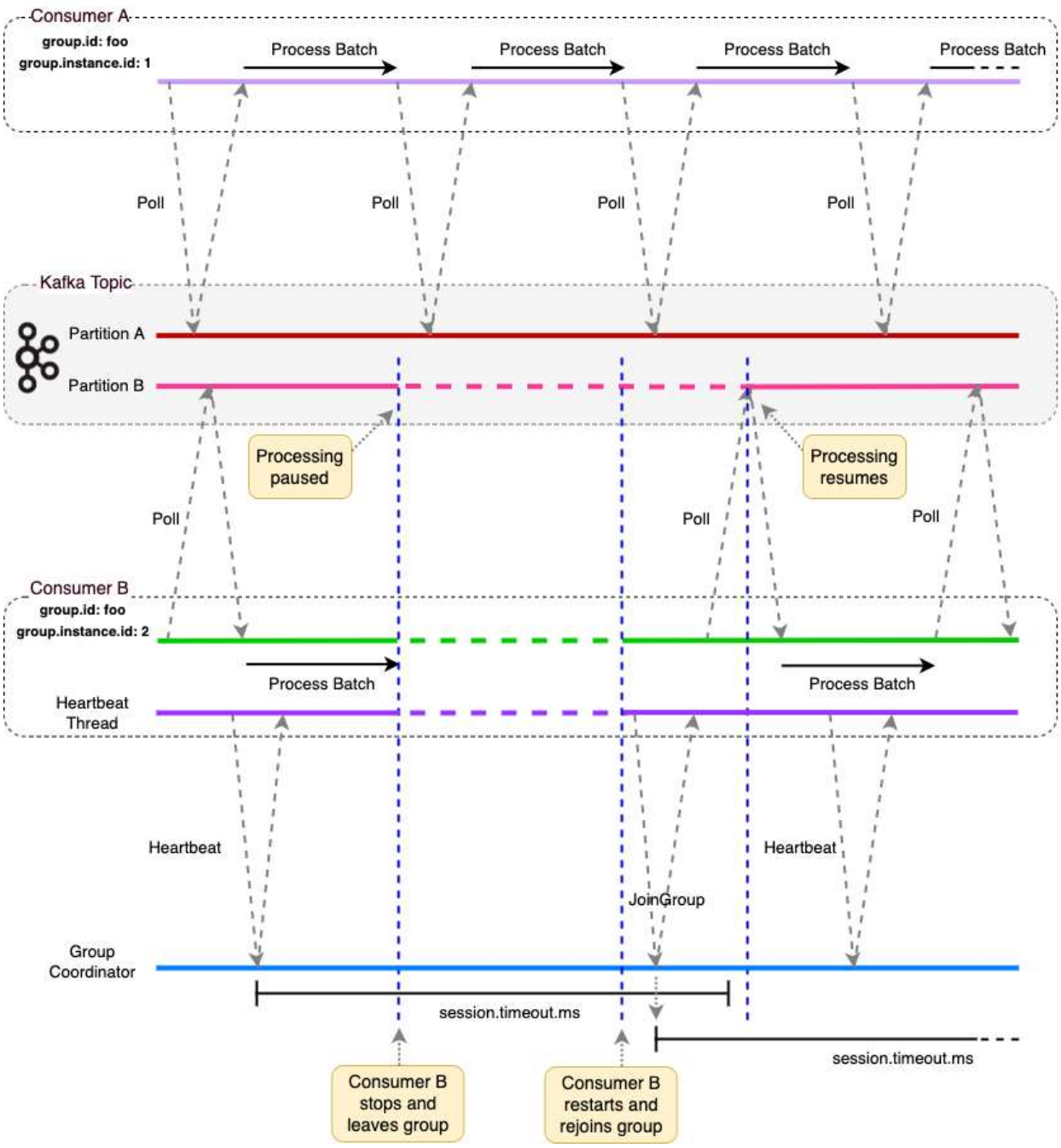

Figure 3: Static Group Membership

For clarity on the diagram the heartbeat thread is only shown for Consumer B.

This feature could be utilised for example by tieing the Id of the Kubernetes pod that an application is running in to the application consumer's **group.instance.id**. If the pod dies and restarts then the Group Coordinator will recognise the consumer as the **group.instance.id** will be the same, and the potentially costly rebalance is avoided.

Static Group Membership is of particular interest when maintaining state in the consumer that is otherwise lost, or must be reloaded, following a rebalance. For example, stateful retry allows a consumer to track retries of message batches across polls. The retry count would be lost on a

consumer rebalance if the partitions being polled are assigned to other consumers. Stateless vs stateful retry is covered in the article here.
Care needs to be taken when using Static Group Membership as when a consumer has died those partitions it was assigned will not be reassigned until the consumer has timed out. Therefore configuring a longer **session.timeout.ms** to allow a restarting consumer time to rejoin and avoid triggering a rebalance comes with the risk that a genuinely failed consumer that does not rejoin will leave partitions without a consumer assigned for longer. However configuring a **session.timeout.ms** that is too short may not allow enough time for a consumer to rejoin before it is removed from the consumer group. With the consumer no longer in the consumer group when it rejoins a rebalance is triggered.
For a consumer with static group membership it does not send a LeaveGroup request when it leaves a group (or indeed fails). Rather it stops heartbeating and remains in the group until the **session.timeout.ms** has been exceeded and is removed from the group by the Group Coordinator. This timeout should then be configured to be sufficiently long to allow time for the consumer to restart and be reassigned its partitions without the need for a rebalance.

Rebalance Risks

Duplicate Messages

A consumer that has exceeded its time out and is considered failed could still be processing the messages it has polled, and that processing could complete successfully. However its consumer offsets write will be rejected as the consumer group rebalance increments the generation Id, and any writes with the previous generation Id are rejected. Meanwhile a new consumer instance is assigned the topic partitions in a rebalance and this consumes and processes the same messages. It is always important to be aware that the application may receive duplicate messages and it must cater for these, if necessary, as required. More on deduplication patterns here.

Rebalance Storms

Rebalance does not complete until all existing consumers have either rejoined or exceeded the **max.poll.interval.ms**. If a consumer does indeed exceed the **max.poll.interval.ms** before it again polls as it is taking longer than expected to process its last batch of messages then when it does complete it will request to rejoin the group, triggering another rebalance. If the cause of the rebalance is for example due to slow responding downstream services that are affecting all consumers the upshot can be rebalance after rebalance being triggered as consumers are continually evicted and then rejoin, a ‘rebalance storm’. Static Group Membership and Incremental rebalancing can of course assist with this but whatever strategies are in place care must be taken with the rebalance configurations.

Conclusion

Consumer Group rebalance is a critical part of how Kafka manages consumer groups, which itself is an important feature that helps make Kafka a highly scalable distributed messaging. Understanding how rebalance works and how the various consumer configurations affect this is essential in ensuring that throughput is maximised and the system does not suffer from frequent periods where messages are not being processed.