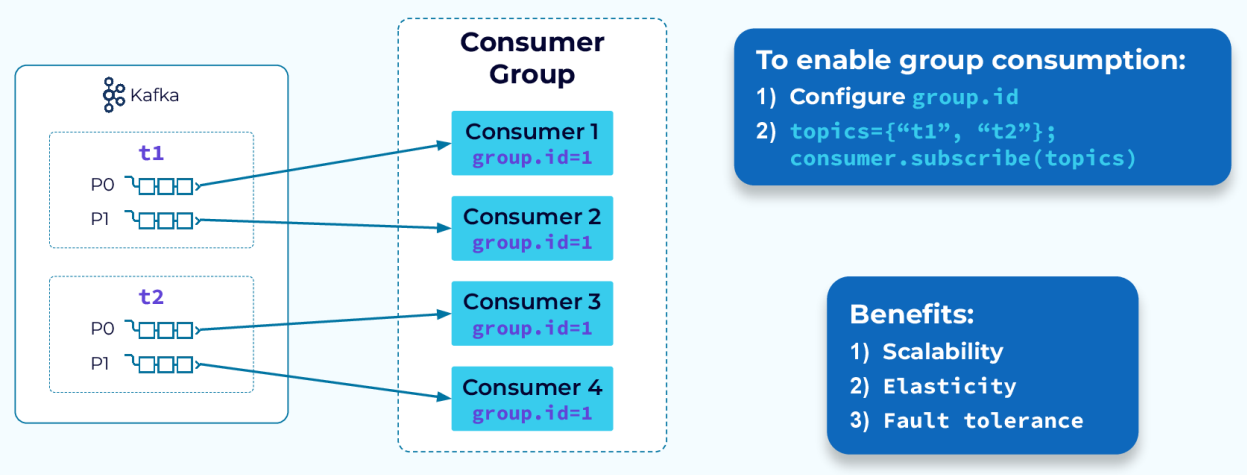
Consumer Group Protocol

- Kafka separates `storage` from `compute`.
- Storage is handled by the brokers and compute is mainly handled by consumers or frameworks built on top of consumers (Kafka Streams, ksqlDB).
- Consumer groups play a key role in the effectiveness and scalability of Kafka consumers.

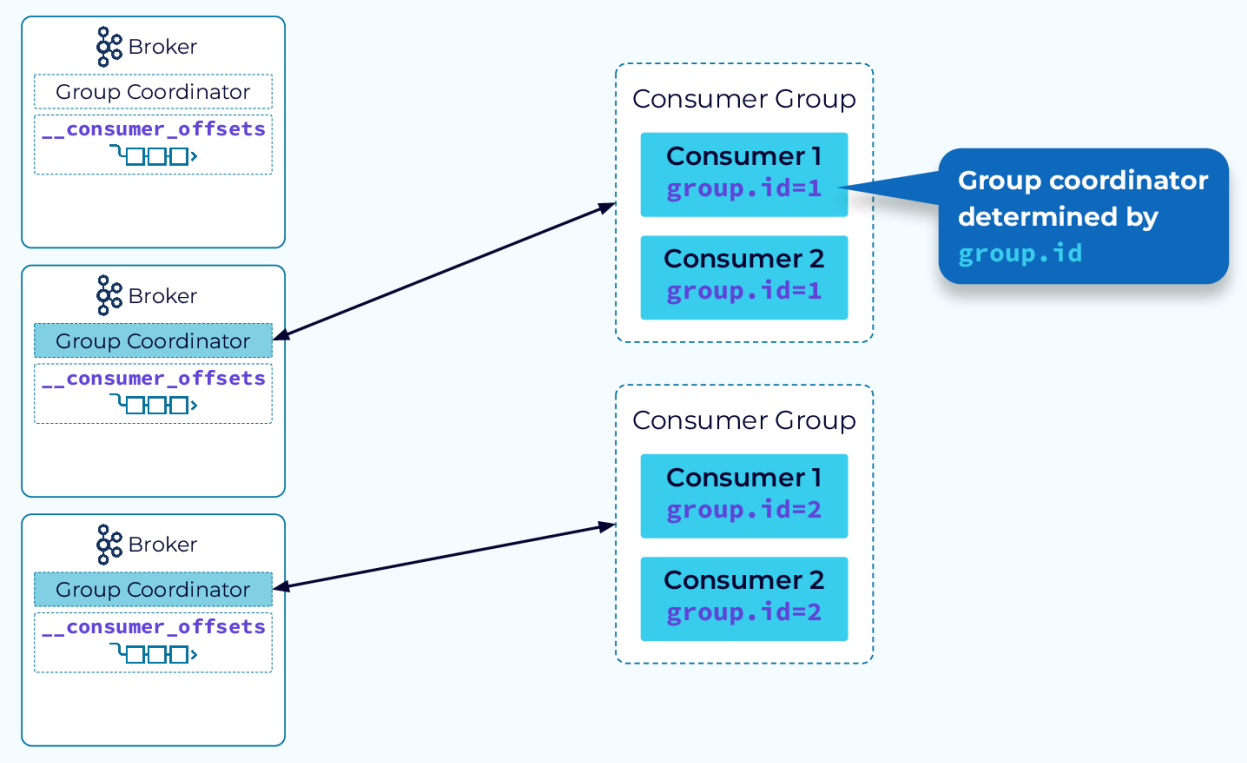
Kafka Consumer Group



- To define a consumer group we just need to set the group.id in the consumer config.
- Once that is set, every new instance of that consumer will be added to the group.
- Then, when the consumer group subscribes to one or more topics, their partitions will be evenly distributed between the instances in the group.

- This allows for parallel processing of the data in those topics.
- The unit of parallelism is the partition.
- For a given consumer group, consumers can process more than one partition but a partition can only be processed by one consumer.
- If our group is subscribed to two topics and each one has two partitions then we can effectively use up to four consumers in the group.
- We could add a fifth but it would sit idle since partitions cannot be shared.
- The assignment of partitions to consumer group instances is dynamic.
- As consumers are added to the group, or when consumers fail or are removed from the group for some other reason, the workload will be rebalanced automatically.

Group Coordinator

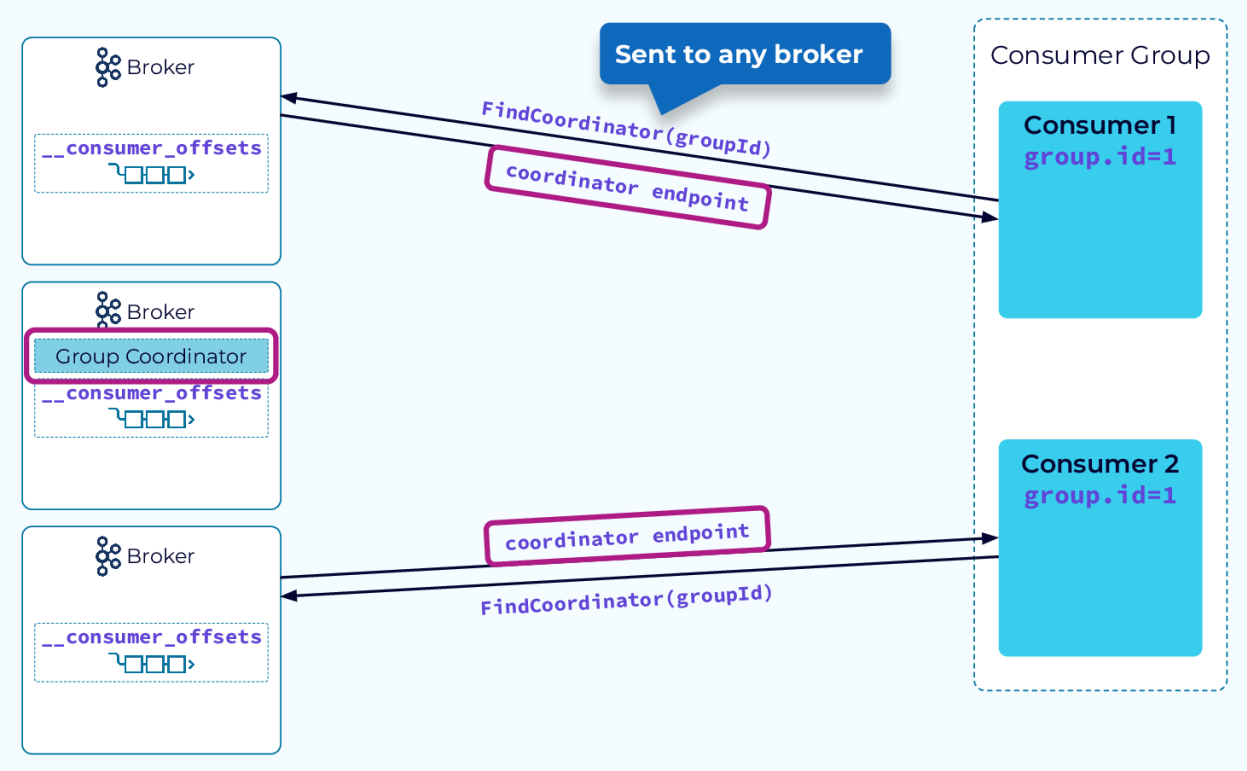


- The magic behind consumer groups is provided by the group coordinator.
- The group coordinator helps to distribute the data in the subscribed topics to the consumer group instances evenly and it keeps things balanced when group membership changes occur.
- The coordinator uses an internal Kafka topic to keep track of group metadata.
- In a typical Kafka cluster, there will be multiple group coordinators.
- This allows for multiple consumer groups to be managed efficiently.

Group Startup

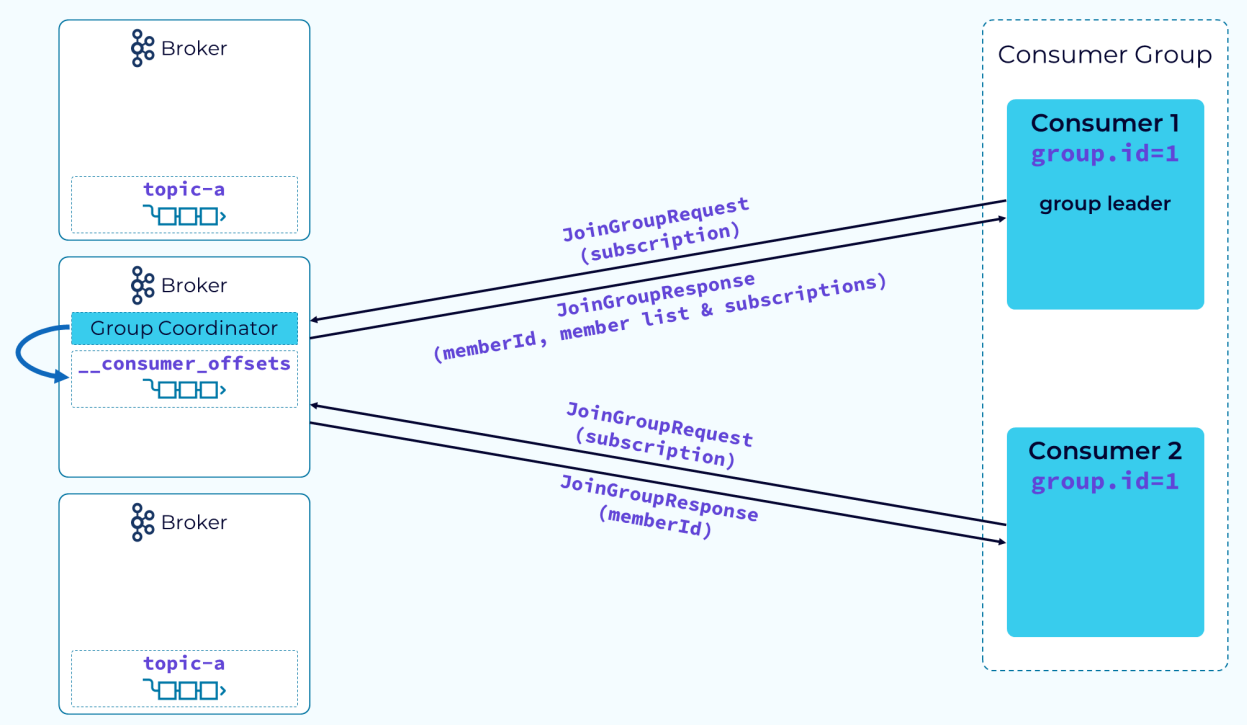
Let’s take a look at the steps involved in starting up a new consumer group.

Step 1 – Find Group Coordinator



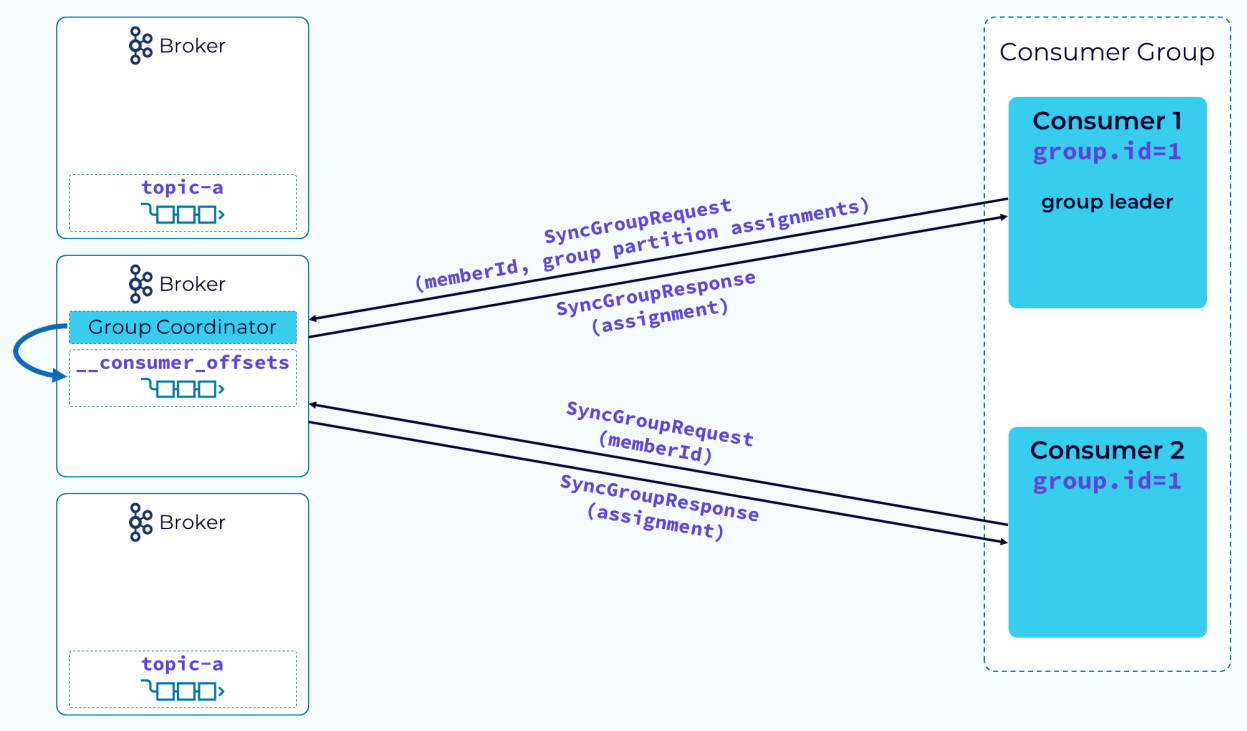
- When a consumer instance starts up it sends a `FindCoordinator` request that includes its group.id to any broker in the cluster.
- The broker will create a hash of the group.id and modulo that against the number of partitions in the internal `__consumer_offsets` topic.
- That determines the partition that all metadata events for this group will be written to.
- The broker that hosts the leader replica for that partition will take on the role of group coordinator for the new consumer group.
- The broker that received the FindCoordinator request will respond with the endpoint of the group coordinator.

Step 2 – Members Join



- Next, the consumers and the group coordinator begin a little logistical dance, starting with the consumers sending a JoinGroup request and passing their topic subscription information.
- The coordinator will choose one consumer, usually the first one to send the JoinGroup request, as the group leader.
- The coordinator will return a memberId to each consumer, but it will also return a list of all members and the subscription info to the group leader.
- The reason for this is so that the group leader can do the actual partition assignment using a configurable partition assignment strategy.

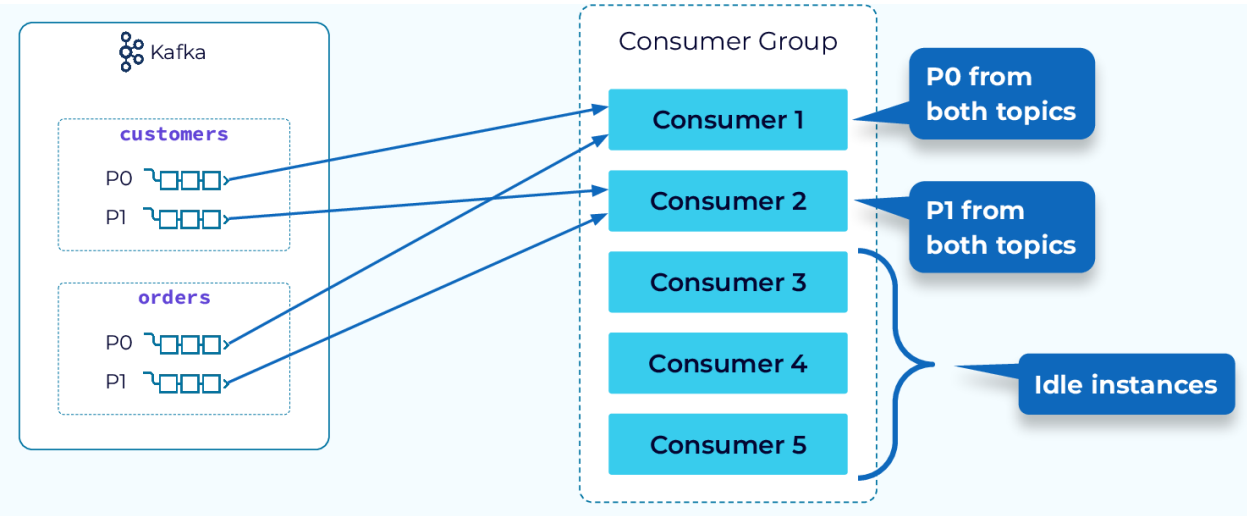
Step 3 – Partitions Assigned



- After the group leader receives the complete member list and subscription information, it will use its configured partitioner to assign the partitions in the subscription to the group members.

- With that done, the leader will send a SyncGroupRequest to the coordinator, passing in its memberId and the group assignments provided by its partitioner.
- The other consumers will make a similar request but will only pass their memberId.
- The coordinator will use the assignment information given to it by the group leader to return the actual assignments to each consumer.
- Now the consumers can begin their real work of consuming and processing data.

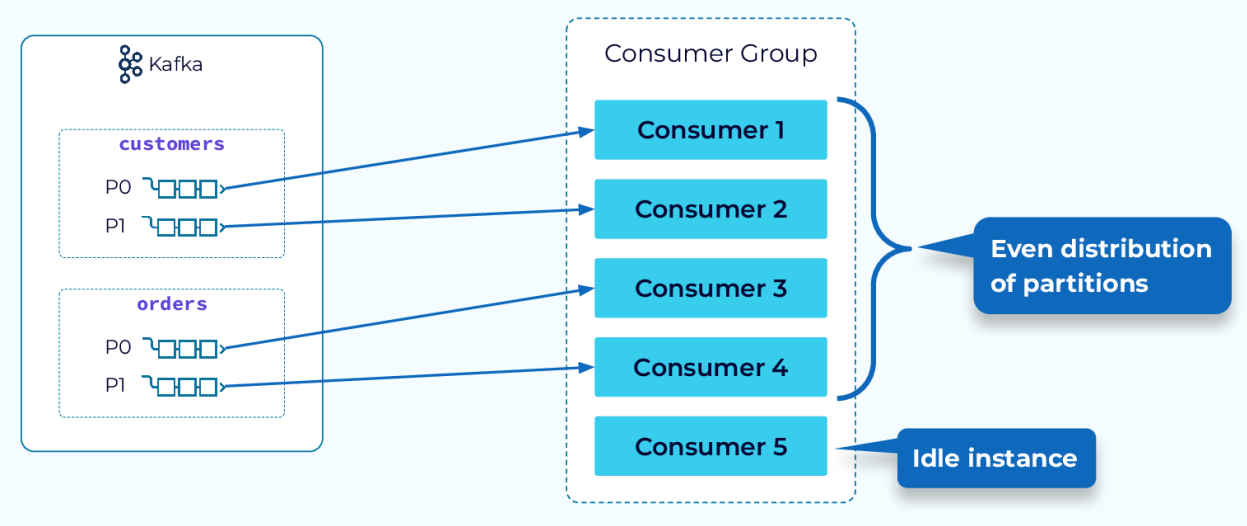
Range Partition Assignment Strategy



- Now, let's look at some of the available assignment strategies.
- First up is the range assignment strategy.
- This strategy goes through each topic in the subscription and assigns each of the partitions to a consumer, starting at the first consumer.
- What this means is that the first partition of each topic will be assigned to the first consumer, the second partition of each topic will be assigned to the second consumer, and so on.

- If no single topic in the subscription has as many partitions as there are consumers, then some consumers will be idle.
- At first glance this might not seem like a very good strategy, but it has a very special purpose.
- When joining events from more than one topic the events need to be read by the same consumer.
- If events in two different topics are using the same key, they will be in the same partition of their respective topics, and with the range partitioner, they will be assigned to the same consumer.

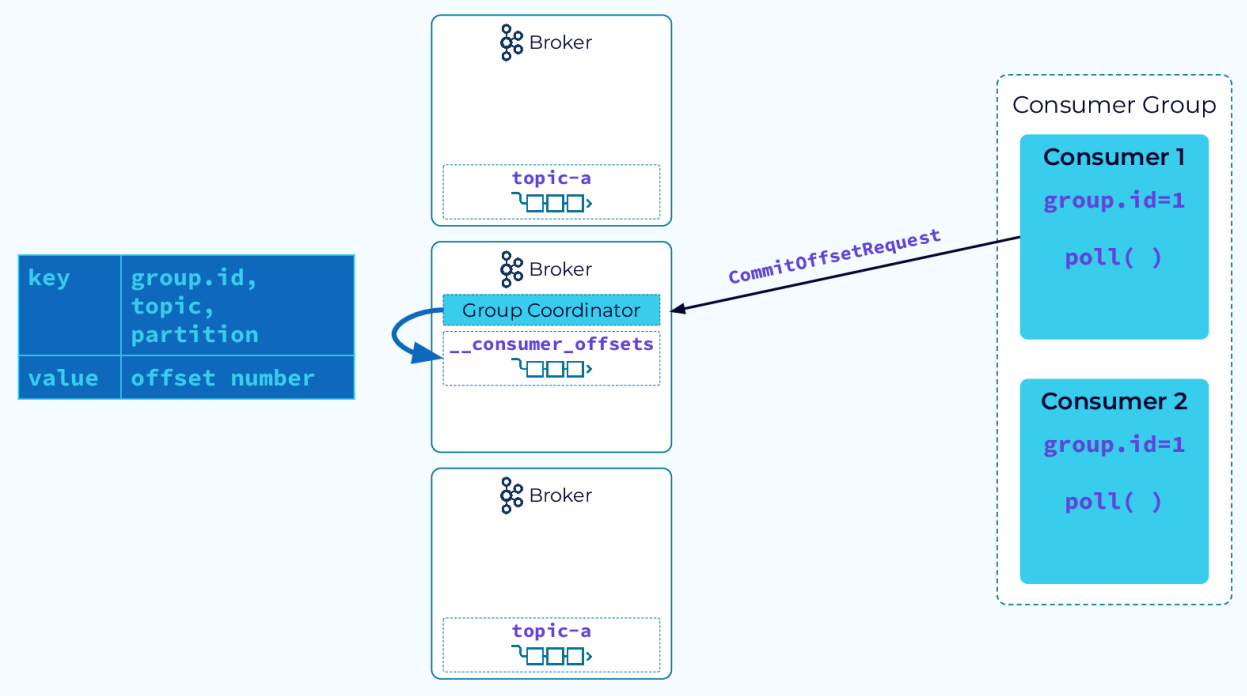
Round Robin and Sticky Partition Assignment Strategies



- Next, let's look at the Round Robin strategy.
- With this strategy, all of the partitions of the subscription, regardless of topic, will be spread evenly across the available consumers.
- This results in fewer idle consumer instances and a higher degree of parallelism.

- A variant of Round Robin, called the Sticky Partition strategy, operates on the same principle but it makes a best effort at sticking to the previous assignment during a rebalance.
- This provides a faster, more efficient rebalance.

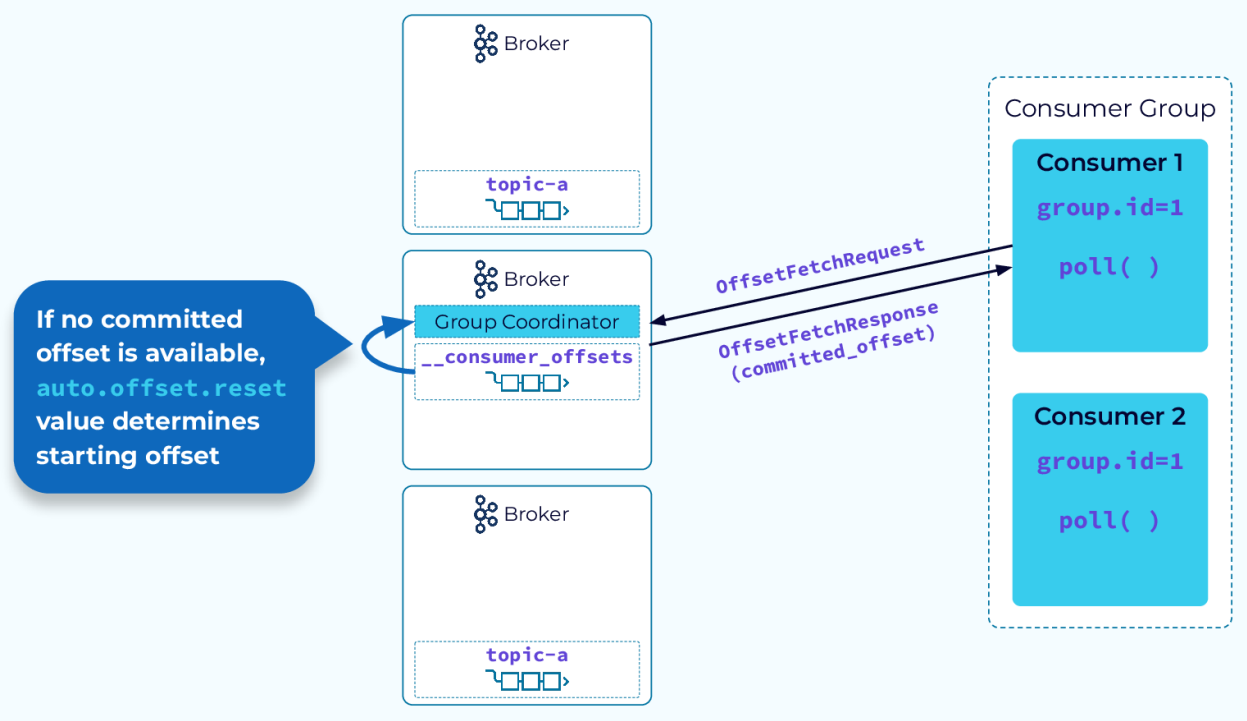
Tracking Partition Consumption



- In Kafka, keeping track of the progress of a consumer is relatively simple.
- A given partition is always assigned to a single consumer, and the events in that partition are always read by the consumer in offset order.
- So, the consumer only needs to keep track of the last offset it has consumed for each partition.

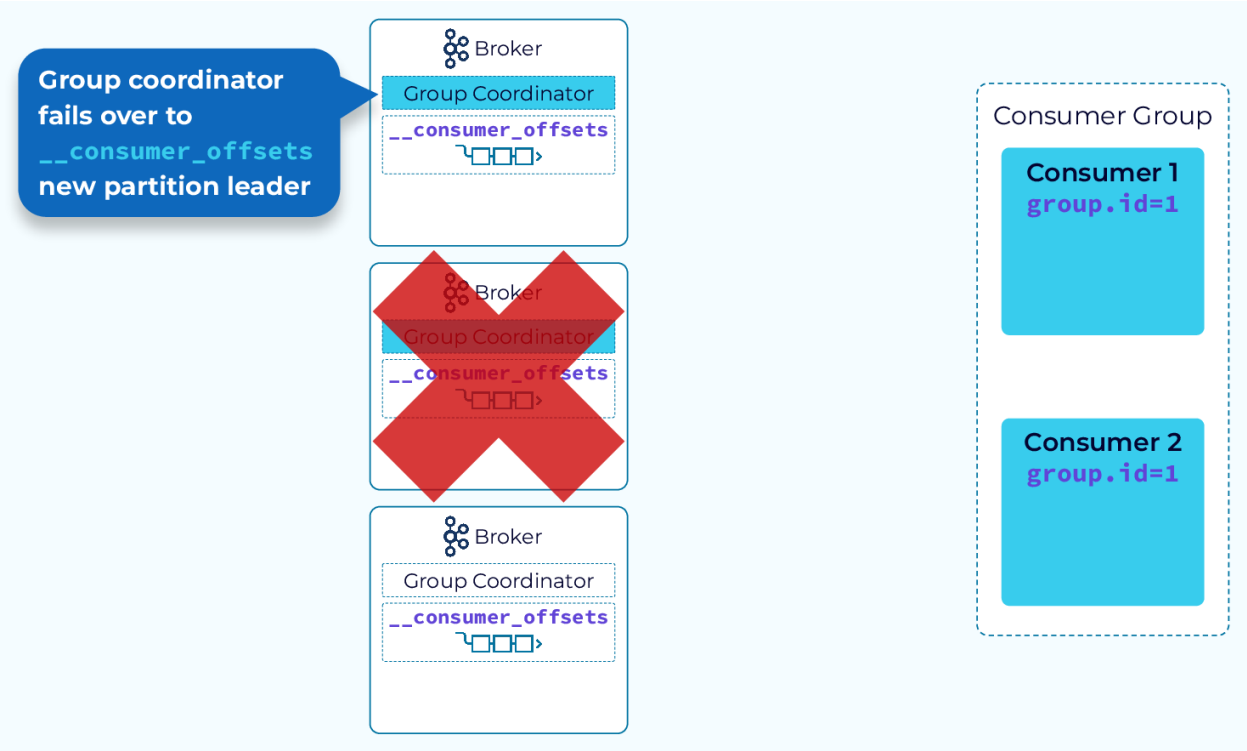
- To do this, the consumer will issue a CommitOffsetRequest to the group coordinator.
- The coordinator will then persist that information in its internal `__consumer_offsets` topic.

Determining Starting Offset to Consume



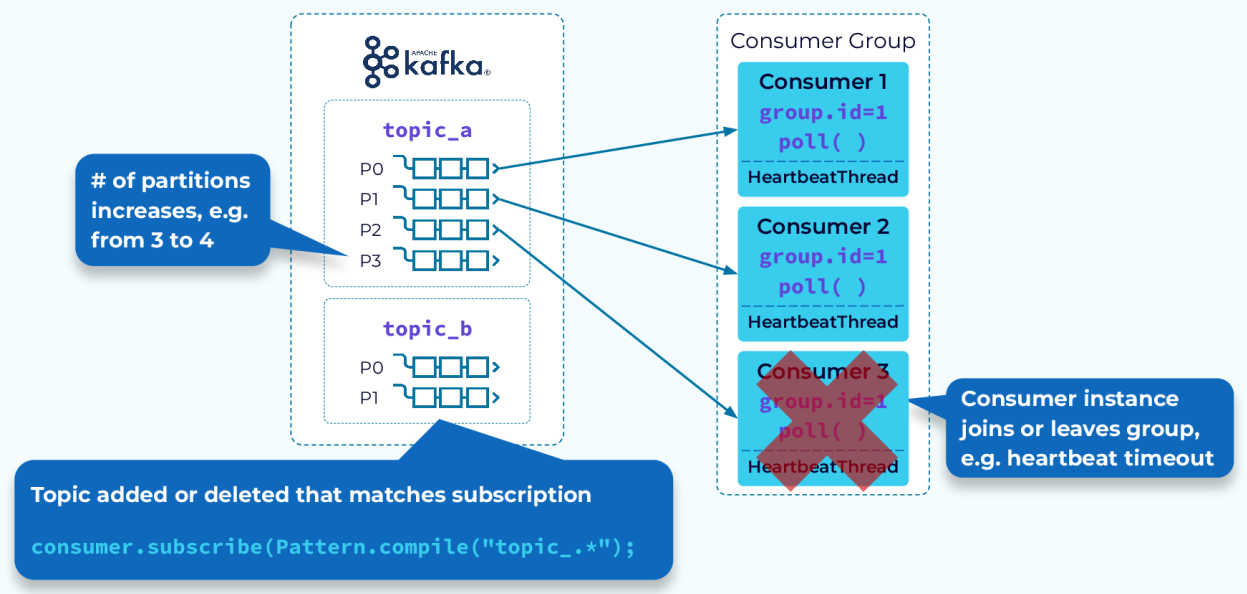
- When a consumer group instance is restarted, it will send an `OffsetFetchRequest` to the group coordinator to retrieve the last committed offset for its assigned partition.
- Once it has the offset, it will resume the consumption from that point.
- If this consumer instance is starting for the very first time and there is no saved offset position for this consumer group, then the `auto.offset.reset` configuration will determine whether it begins consuming from the earliest offset or the latest.

Group Coordinator Failover



- The internal __consumer_offsets topic is replicated like any other Kafka topic. Also, recall that the group coordinator is the broker that hosts the leader replica of the __consumer_offsets partition assigned to this group.
- So if the group coordinator fails, a broker that is hosting one of the follower replicas of that partition will become the new group coordinator.
- Consumers will be notified of the new coordinator when they try to make a call to the old one, and then everything will continue as normal.

Consumer Group Rebalance Triggers

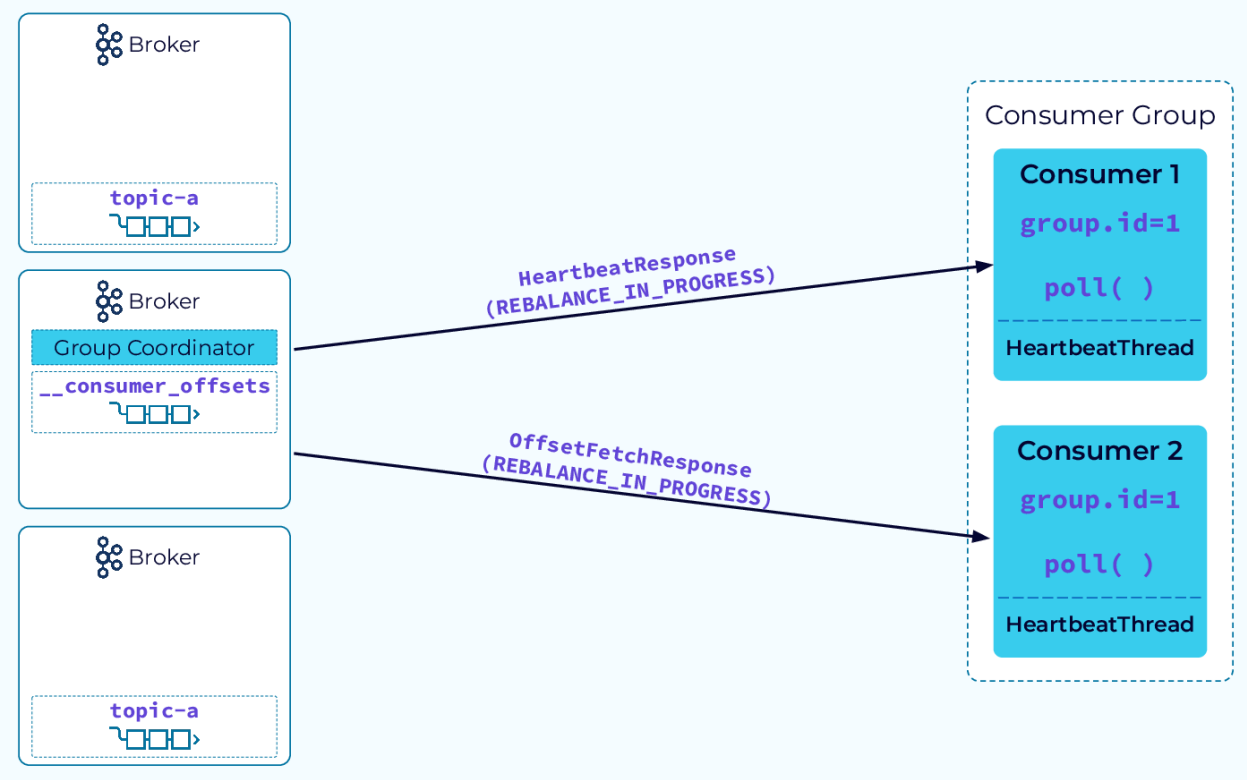


One of the key features of consumer groups is rebalancing. We'll be discussing rebalances in more detail, but first let's consider some of the events that can trigger a rebalance:

- An instance fails to send a heartbeat to the coordinator before the timeout and is removed from the group
- An instance has been added to the group
- Partitions have been added to a topic in the group's subscription
- A group has a wildcard subscription and a new matching topic is created
- And, of course, initial group startup

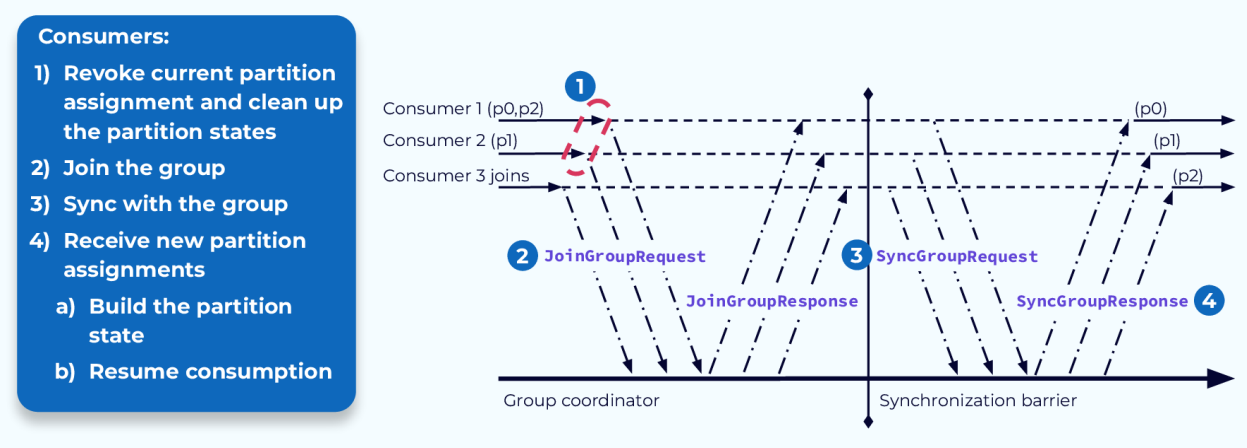
Next we'll look at what happens when a rebalance occurs.

Consumer Group Rebalance Notification



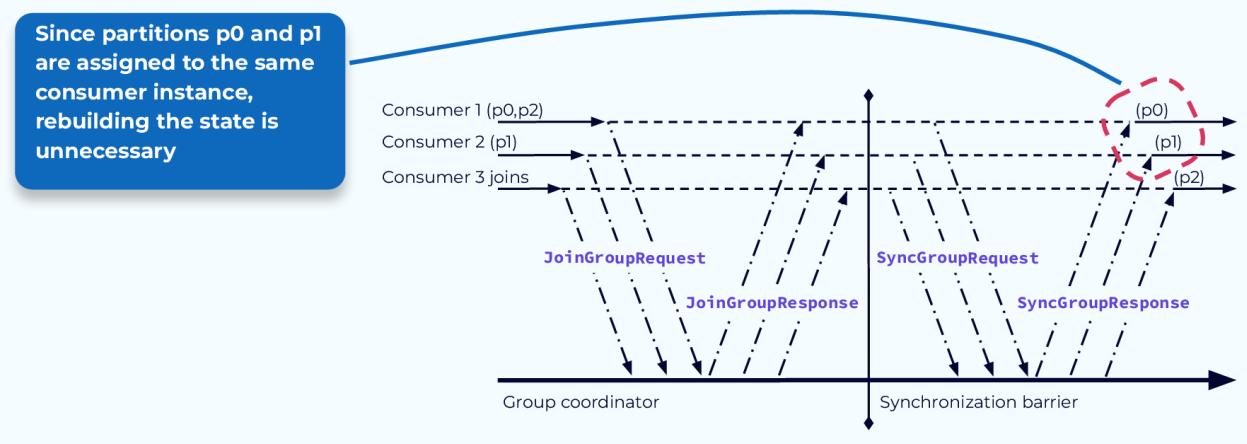
- The rebalance process begins with the coordinator notifying the consumer instances that a rebalance has begun.
- It does this by piggybacking on the HeartbeatResponse or the OffsetFetchResponse.
- Now the fun begins!

Stop-the-World Rebalance



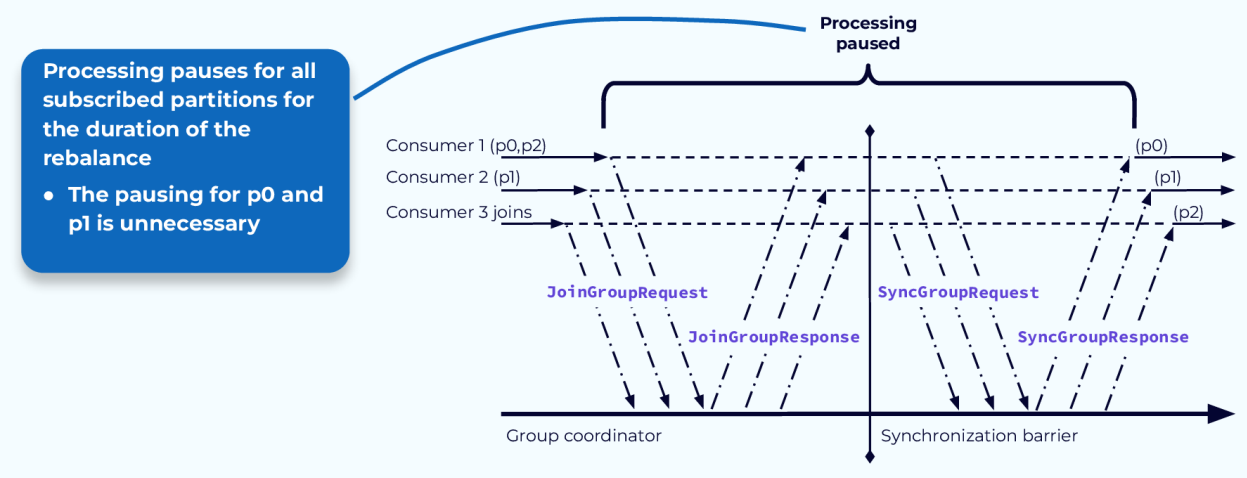
- The traditional rebalance process is rather involved.
- Once the consumers receive the rebalance notification from the coordinator, they will revoke their current partition assignments.
- If they have been maintaining any state associated with the data in their previously assigned partitions, they will also have to clean that up.
- Now they are basically like new consumers and will go through the same steps as a new consumer joining the group.
- They will send a JoinGroupRequest to the coordinator, followed by a SyncGroupRequest.
- The coordinator will respond accordingly, and the consumers will each have their new assignments.
- Any state that is required by the consumer would now have to be rebuilt from the data in the newly assigned partitions.
- This process, while effective, has some drawbacks. Let's look at a couple of those now.

Stop-the-World Problem #1 – Rebuilding State



- The first problem is the need to rebuild state.
- If a consumer application was maintaining state based on the events in the partition it had been assigned to, it may need to read all of the events in the partition to rebuild that state after the rebalance is complete.
- As you can see from our example, sometimes this work is being done even when it is not needed.
- If a consumer revokes its assignment to a particular partition and then is assigned that same partition during the rebalance, a significant amount of wasted processing may occur.

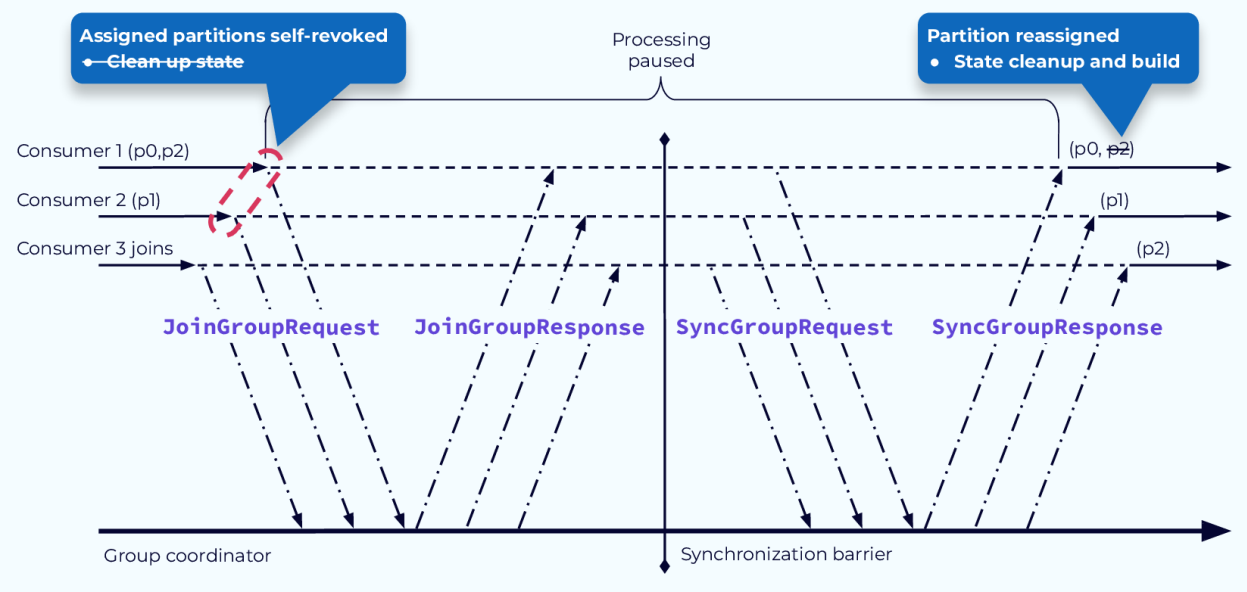
Stop-the-World Problem #2 – Paused Processing



- The second problem is that we're required to pause all processing while the rebalance is occurring, hence the name "Stop-the-world."
- Since the partition assignments for all consumers are revoked at the beginning of the process, nothing can happen until the process completes and the partitions have been reassigned.
- In many cases, as in our example here, some consumers will keep some of the same partitions and could have, in theory, continued working with them while the rebalance was underway.

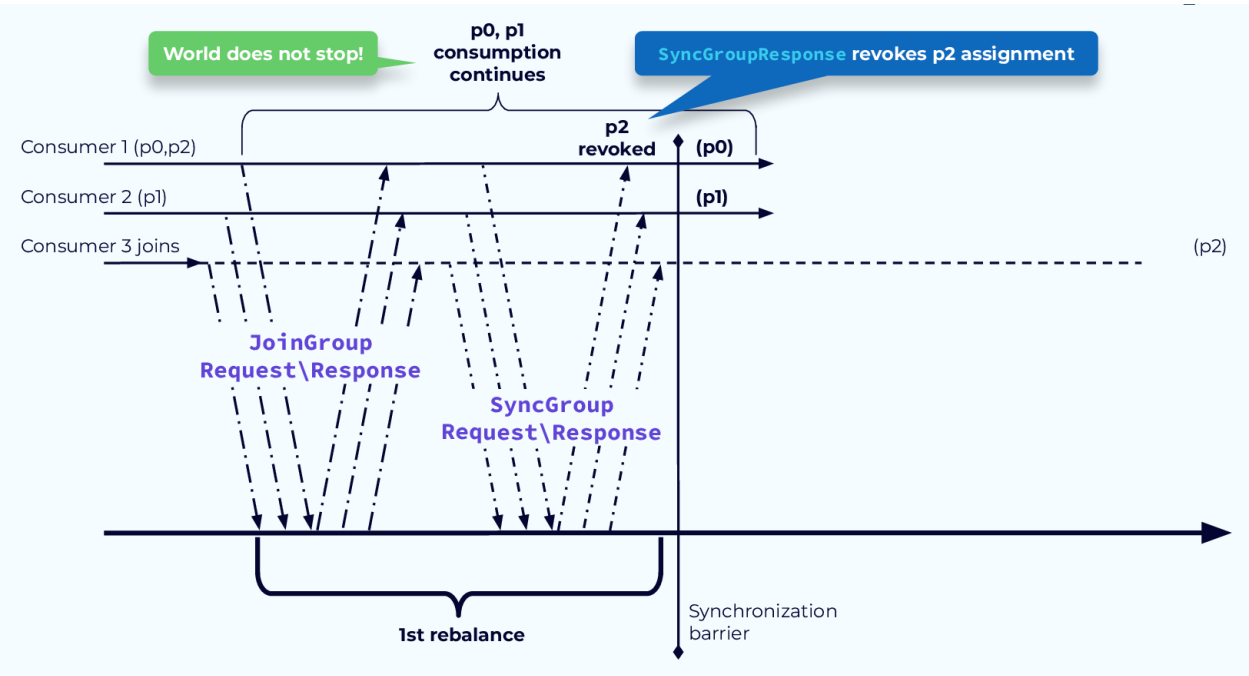
Let's see some of the improvements that have been made to deal with these problems.

Avoid Needless State Rebuild with StickyAssignor



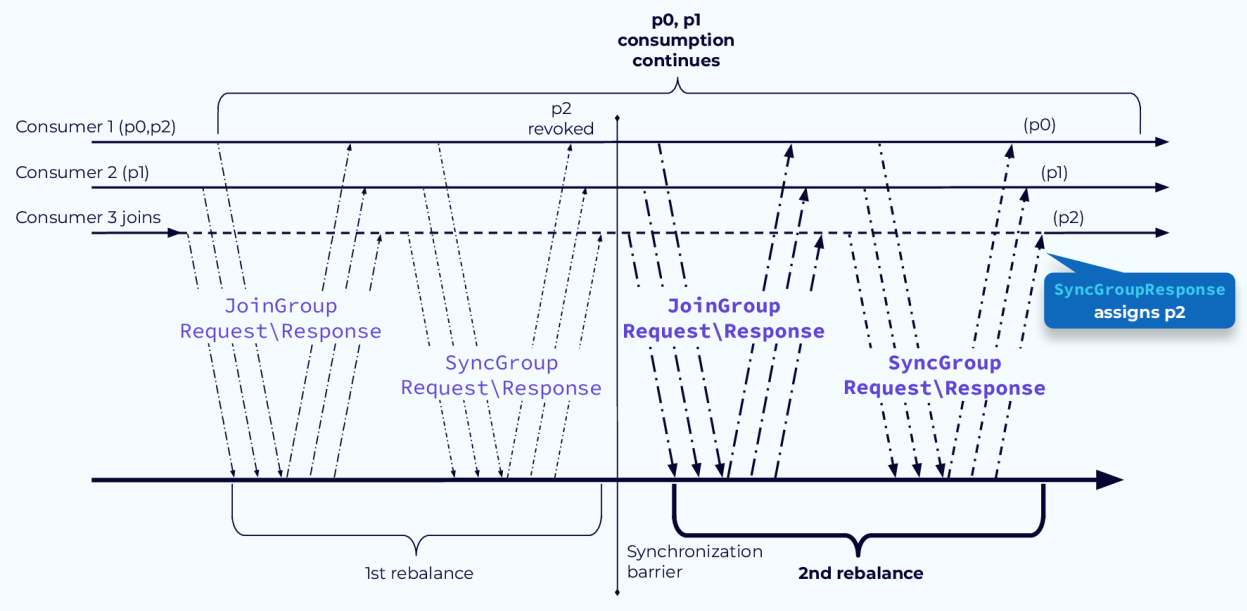
- First, using the new StickyAssignor we can avoid unnecessary state rebuilding.
- The main difference with the StickyAssignor, is that the state cleanup is moved to a later step, after the reassignments are complete.
- That way if a consumer is reassigned the same partition it can just continue with its work and not clear or rebuild state.
- In our example, state would only need to be rebuilt for partition p2, which is assigned to the new consumer.

Avoid Pause with CooperativeStickyAssignor Step 1



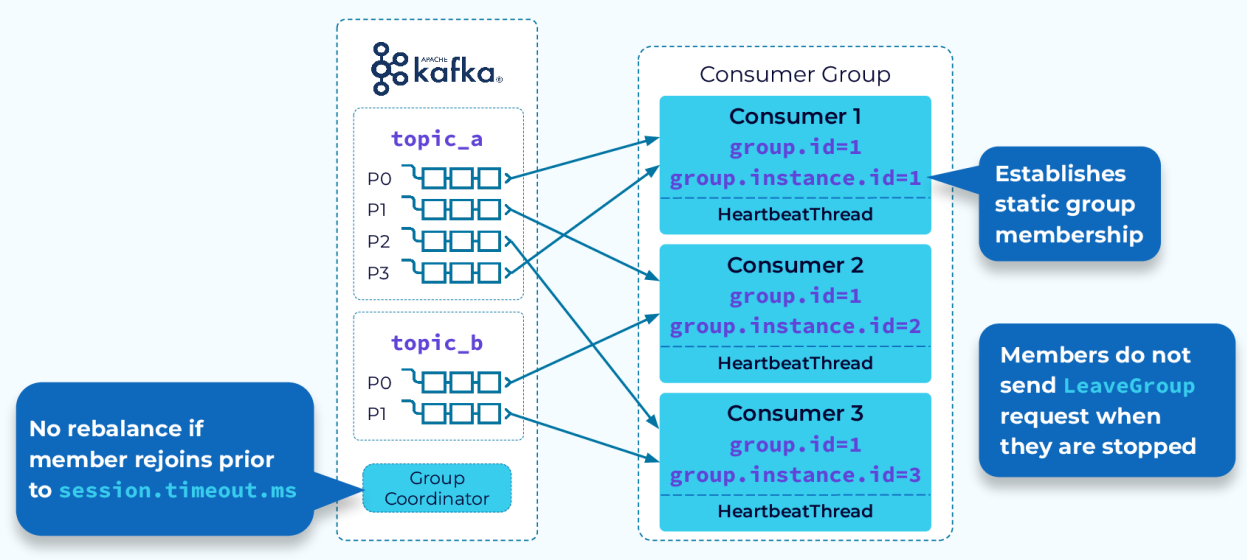
- To solve the problem of paused processing, we introduced the CooperativeStickyAssignor.
- This assignor works in a two-step process.
- In the first step the determination is made as to which partition assignments need to be revoked.
- Those assignments are revoked at the end of the first rebalance step.
- The partitions that are not revoked can continue to be processed.

Avoid Pause with CooperativeStickyAssignor Step 2



- In the second rebalance step, the revoked partitions will be assigned.
- In our example, partition 2 was the only one revoked and it is assigned to the new consumer 3.
- In a more involved system, all of the consumers might have new partition assignments, but the fact remains that any partitions that did not need to move can continue to be processed without the world grinding to a halt.

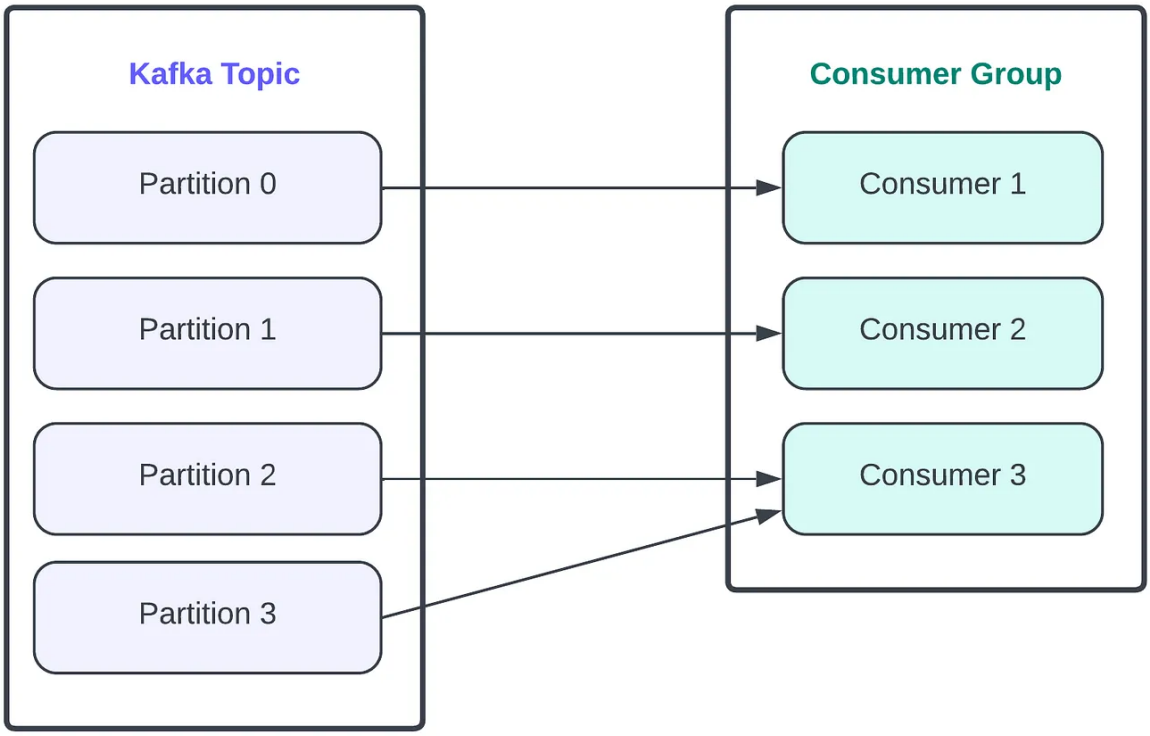
Avoid Rebalance with Static Group Membership



- As the saying goes, the fastest rebalance is the one that doesn't happen.
- That's the goal of static group membership. With static group membership each consumer instance is assigned a group.instance.id.
- Also, when a consumer instance leaves gracefully it will not send a LeaveGroup request to the coordinator, so no rebalance is started.
- When the same instance rejoins the group, the coordinator will recognize it and allow it to continue with its existing partition assignments. Again, no rebalance needed.
- Likewise, if a consumer instance fails but is restarted before its heartbeat interval has timed out, it will be able to continue with its existing assignments.

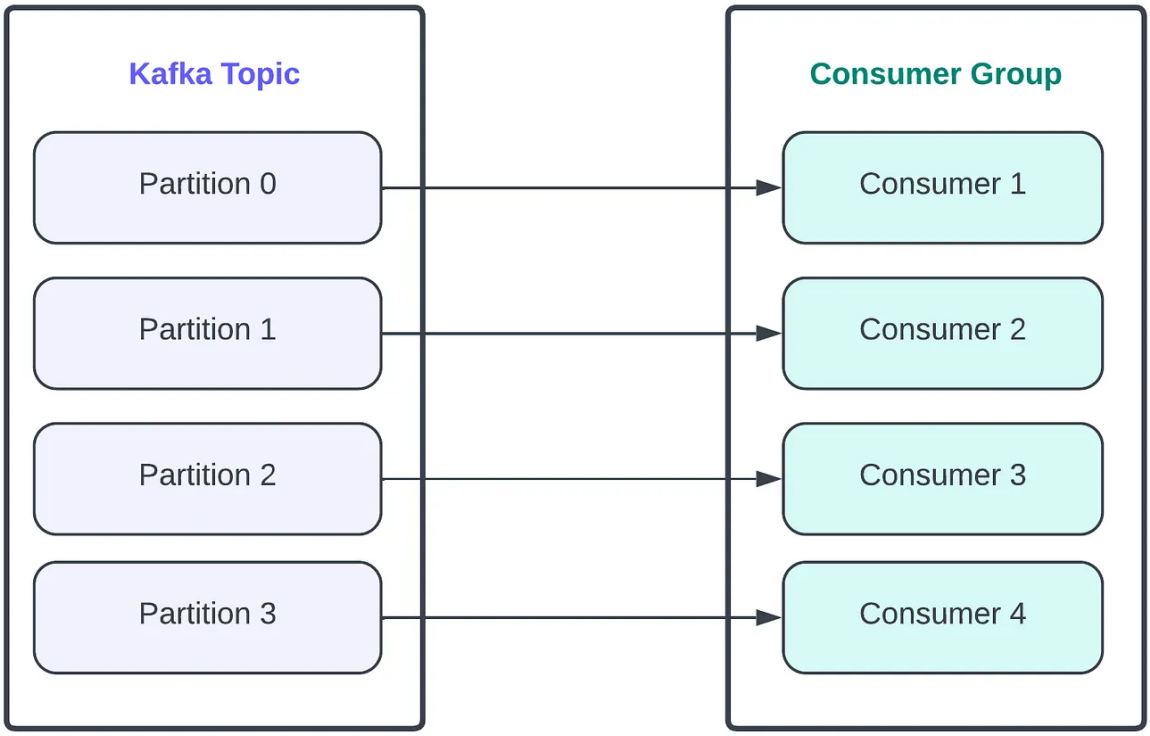
---

1. Having More Partitions Than Consumers in the Consumer Group

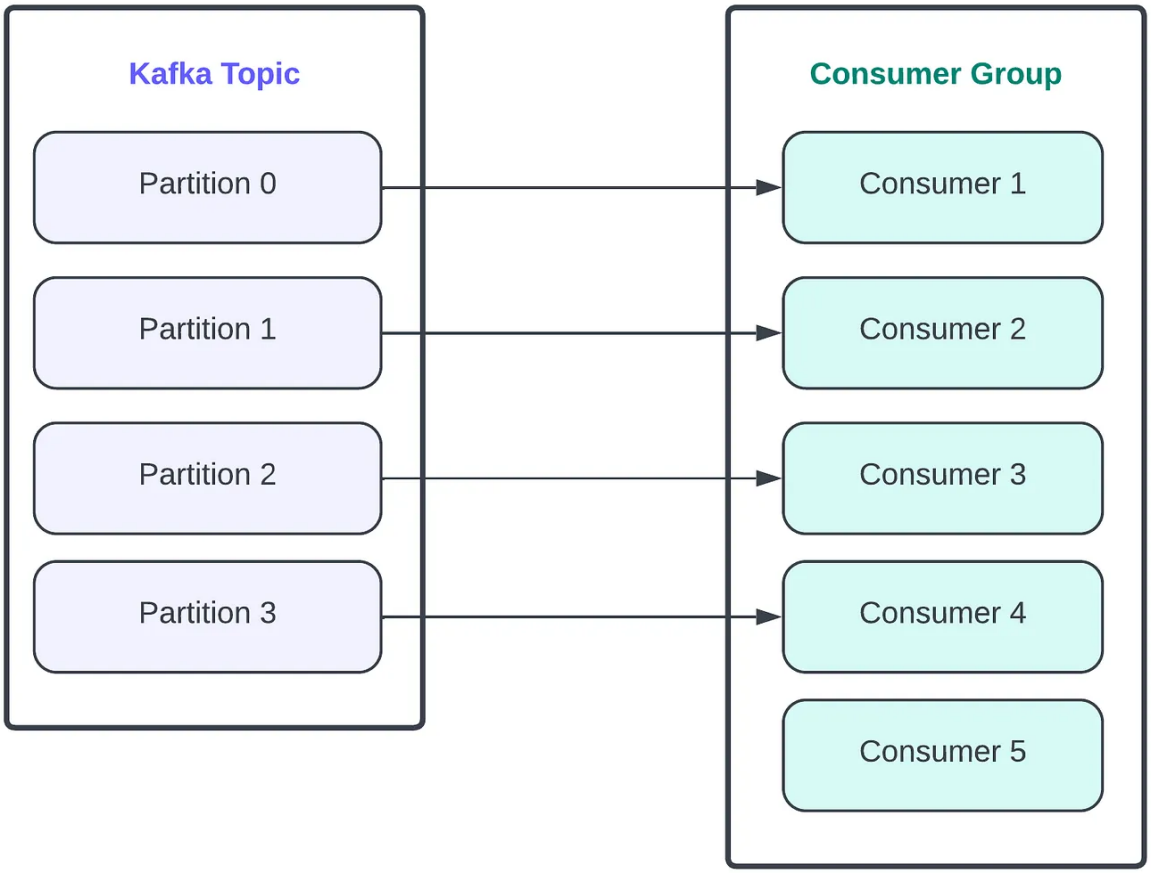


---

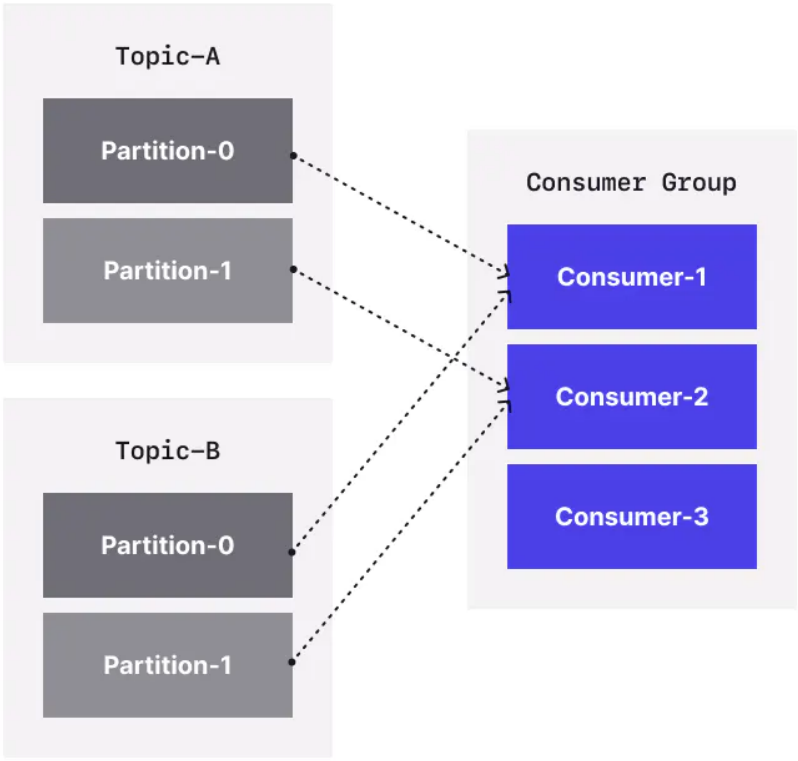
2. Having an Equal Number of Partitions and Consumers in the Consumer Group



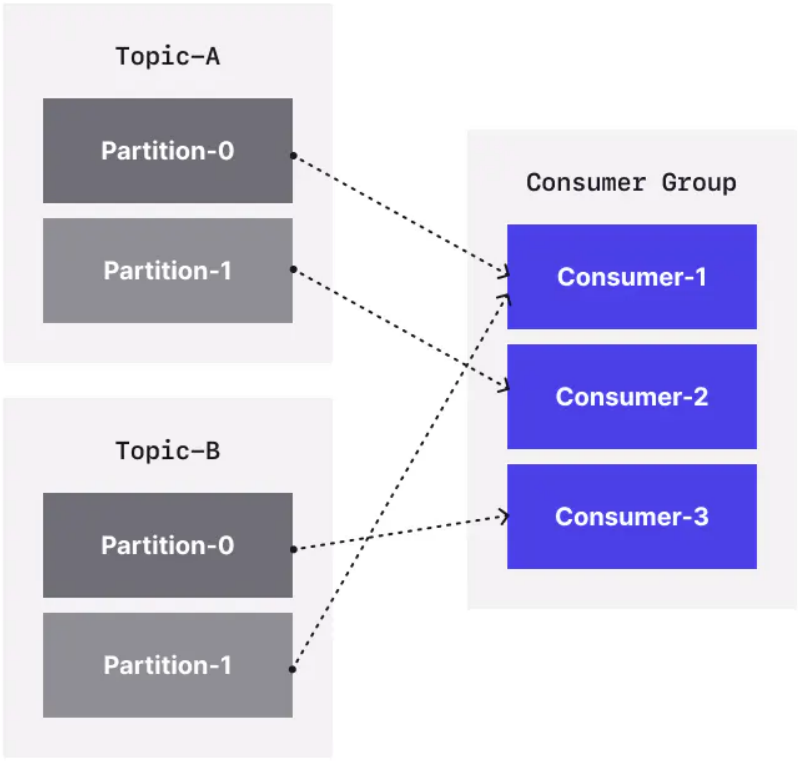
3. Having Fewer Partitions Than Consumers in the Consumer Group



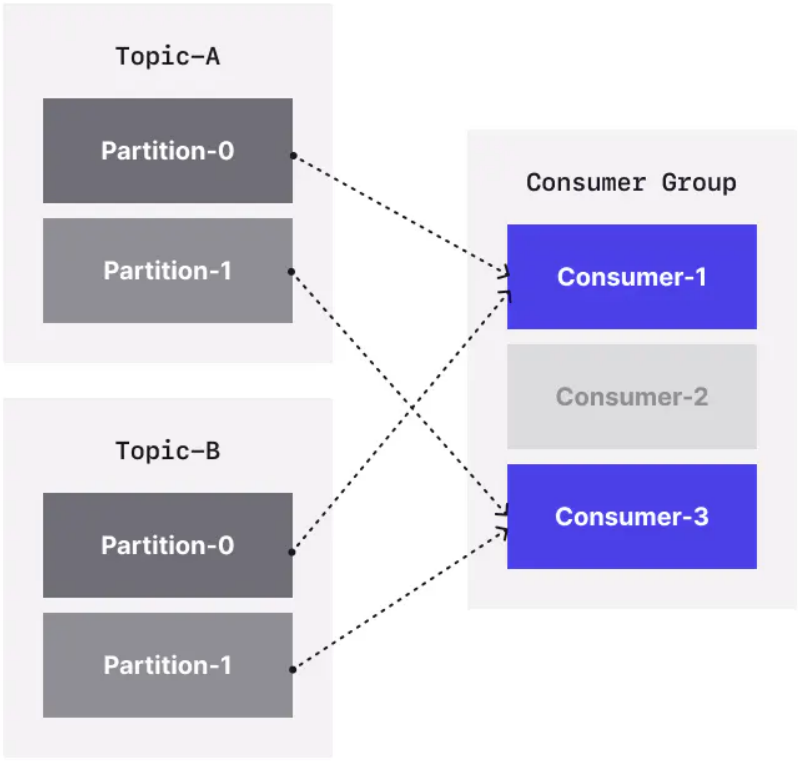
RangeAssignor



RoundRobinAssignor

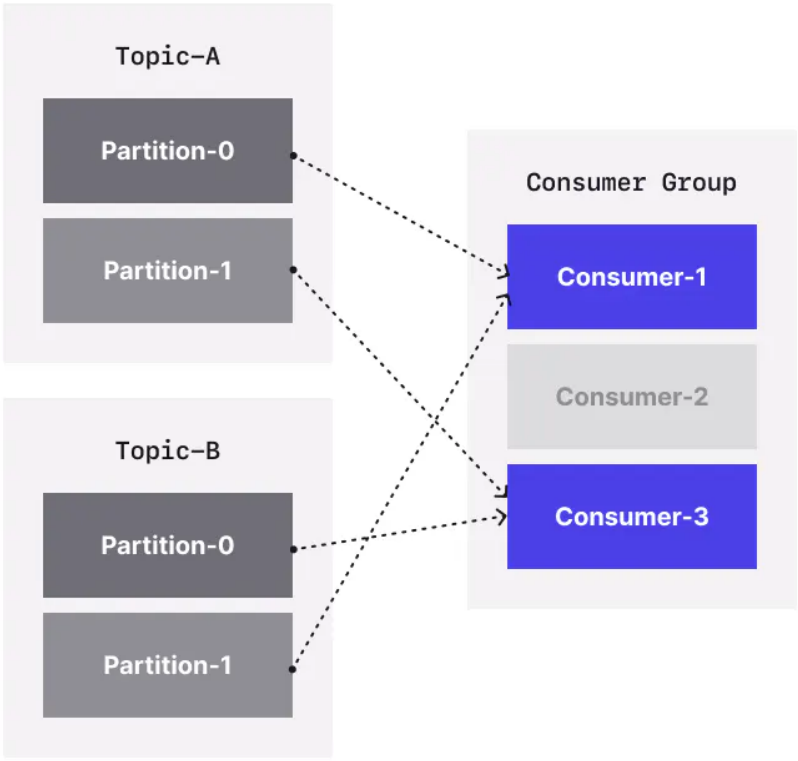

Topic–A
Partition-0
Partition-1
Topic–B
Partition-0
Partition-1
Consumer Group
Consumer-1
Consumer-2
Consumer-3

problem-1:

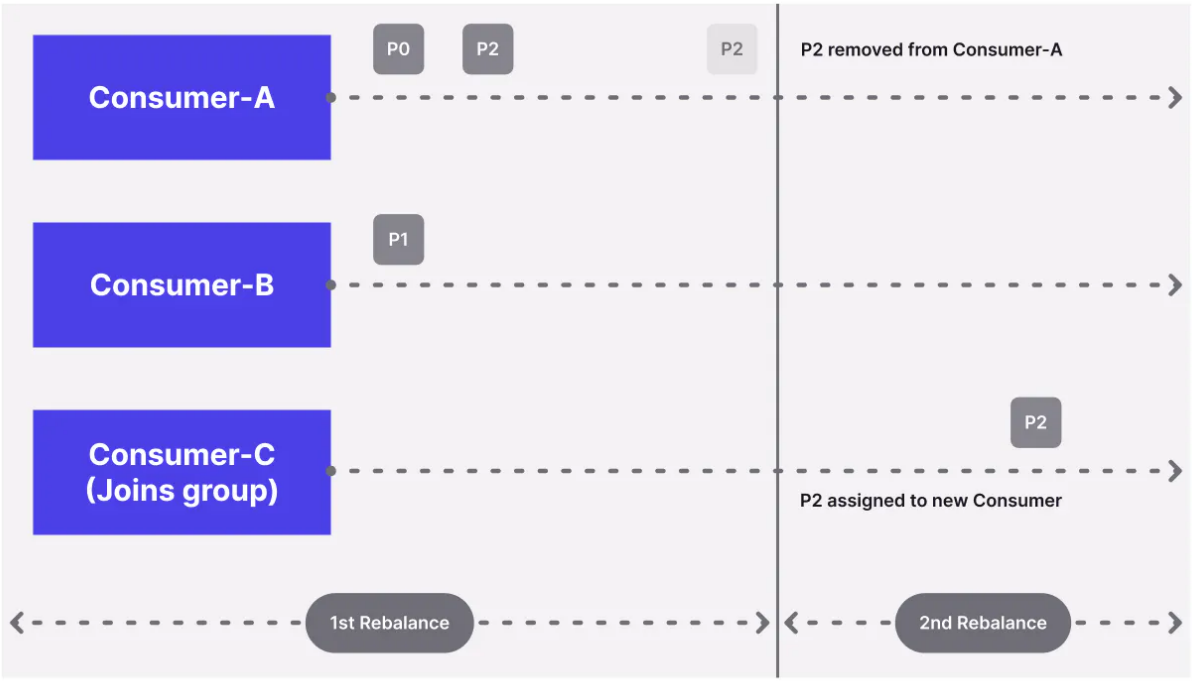


> unnecessary partition movement can impact Consumer performance.

StickyAssignor



CooperativeStickyAssignor


Consumer-A
P0
P2
P2
P2 removed from Consumer-A
Consumer-B
P1
Consumer-C
(Joins group)
P2
P2 assigned to new Consumer
1st Rebalance
2nd Rebalance