Chapter 15: Group Membership and Partition Assignment

The reader should be aware by now that partitions in a topic are allocated approximately evenly among the live members of a consumer group, and that partition allocations for one group are entirely independent of the allocations of any other group. On the whole, group membership is pivotal to Kafka's architecture; it acts as a load balancing mechanism and ensures that any given partition is assigned to at most one member of a group.

This chapter explores Kafka's group membership protocol and the underlying mechanisms by which Kafka ensures that members are always able to make progress in the consumption of records, and that records are processed in the prescribed order and on at most one consumer in a group.

Parts of this chapter contain sizable doses of theory on Kafka's inner workings, and might not immediately feel useful to a Kafka practitioner. Nonetheless, an appreciation of the underlying mechanics will go a long way in understanding the more practical material, such as session timeouts, liveness, partition assignment, and static membership.

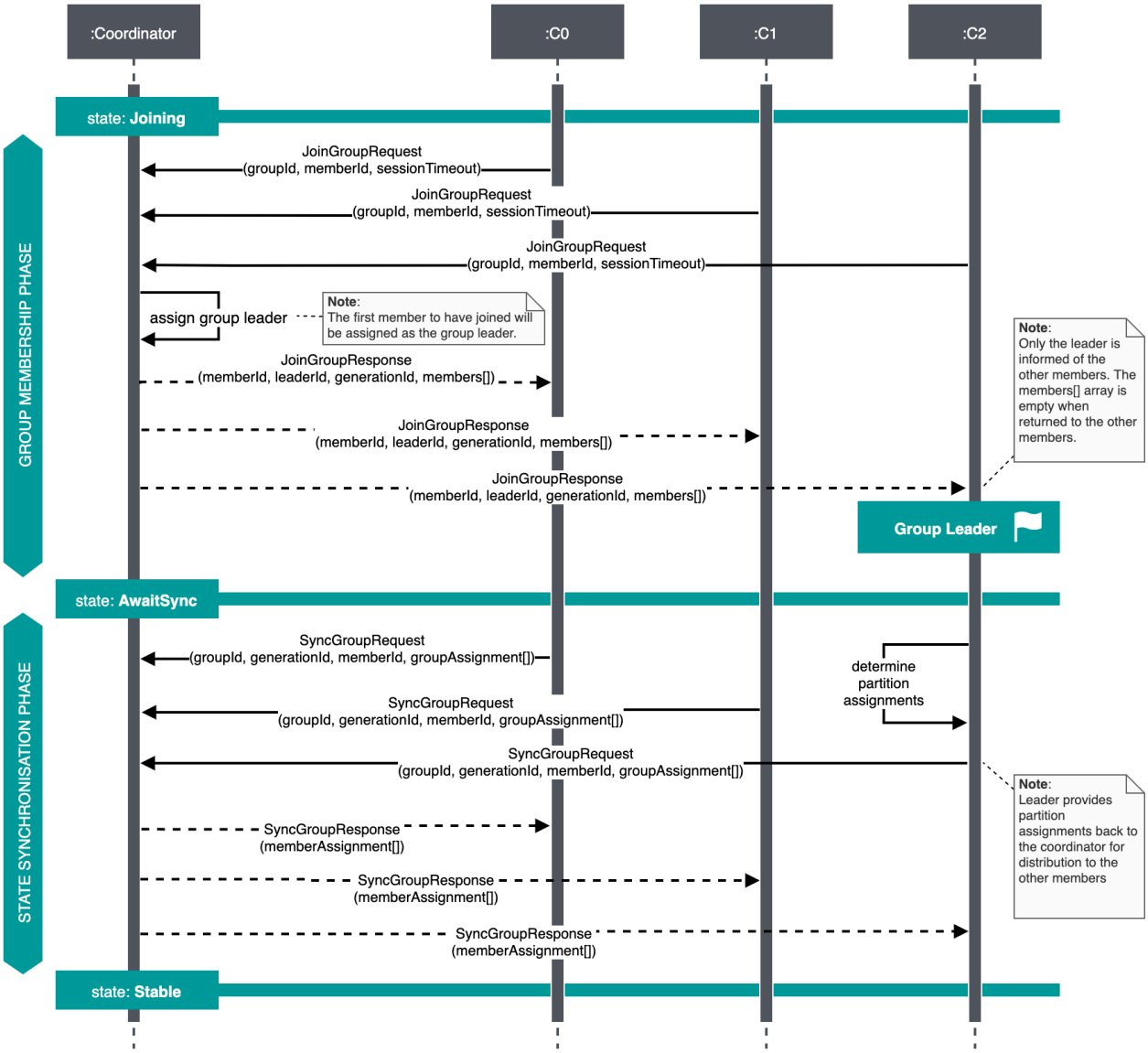
Group membership basics

A consumer group is a set of related consumers that contend for the assignment of a mutually exclusive set of partitions for the subscribed topics, such that a given topic-partition combination is assigned to *at most one* consumer. We say 'at most one' to cover the case when all consumers are offline or not responding. Rest assured — when all is well and there is at least one consumer in the group, a partition will be assigned to exactly one consumer.

Members of one consumer group cannot interfere with those of another group; in other words, consumer groups are completely isolated from one another.

Establishing group membership

Kafka's group management protocol is divided into two phases: *group membership* and *state synchronisation*. The group membership phase is used to identify the active members of the group and appoint a group leader. The state synchronisation phase is used by the group leader to derive the partition assignment for all group members, including itself, and to disseminate this state among the remaining population of consumers. The sequence diagram below illustrates the phases of the protocol. Explanations will follow.



Protocol phases

The group membership phase commences when a consumer sends a `JoinGroupRequest` to the group's coordinator. The request will contain the group ID (specified by the `group.id` configuration property), the internal member ID of the consumer (which is initially empty for new consumers), the desired session timeout (specified by `session.timeout.ms`), as well as the consumer's own metadata.

Upon receiving a `JoinGroupRequest`, the coordinator will update its state and select a leader from the group, if one has not been already assigned. The coordinator will park the request, intentionally delaying the response until all expected members of the group have sent their join requests. The delay acts as a synchronisation barrier, ensuring all members become simultaneously aware of one another. The responses will contain the set of individual member IDs and the ID of the group leader.

The coordinator of each group is implicitly selected from the partition leaders of the internal topic `__consumer_offsets`, which is used to store committed offsets. The group's ID is hashed to one of the partitions for the `__consumer_offsets` topic and the leader of that partition is taken to be the coordinator. By piggybacking on an existing leadership election process, Kafka conveniently avoids yet another arbitration process to elect group coordinators. Also, the management of consumer groups is divided approximately equally across all the brokers in the cluster, which allows the number of groups to scale by increasing the number of brokers.

State synchronisation

The responsibility of assigning topic-partitions to consumers falls upon the group leader, and is performed in the *state synchronisation* phase of the protocol. The assignment (sub-)protocol is conveniently tunnelled within the broader group management protocol and is concealed from the coordinator — in the sense that the coordinator largely treats the contents of state synchronisation messages as a 'black box'. As such, the assignment protocol is often referred to as an *embedded* protocol. Other embedded protocols may coexist, tunnelled in the same manner, potentially bearing no relation to partition assignment.

To make it abundantly clear, the terms 'group leader' and 'group coordinator' refer to different entities. The group leader is a consumer client that is responsible for performing partition assignment; the group coordinator is a broker that arbitrates group membership.

Once group membership has been established by the completion of the first phase, the coordinator will enter the `AwaitSync` state, waiting for the partition assignment to be performed on the leader. The leader will delegate to an implementation of a `ConsumerPartitionAssignor` to perform partition assignment, communicating the outcome of the assignment to the coordinator in a `SyncGroupRequest`.

In the meantime, all members will send their `SyncGroupRequest` messages to the coordinator. Having received the `SyncGroupRequest` from the leader, containing the partition assignments for each member, the coordinator replies with a `SyncGroupResponse` to all members of the group — informing them of their individual partition assignments and completing the protocol. Having done so, the coordinator enters the `Stable` state, and will maintain this state until group membership changes.

As of version 0.9.0.0, Kafka separates the management of group membership from partition assignment. While group arbitration is performed on the coordinator, the partition assignment is delegated to one of the members — the group leader. Before Kafka 0.9.0.0, the coordinator was responsible for both functions.

Delayed rebalance

Having the coordinator reply with a `JoinGroupResponse` immediately after all expected members have joined generally works well when a group is undergoing minor changes in population — for

example, as a result of scaling events or one-off consumer failures. However, if group members are joining at a rapid rate, the number of unnecessary rebalances will go through the roof. This 'swarm' effect can typically be felt on application startup or when previously connected clients reconnect *en masse* due to an intermittent network failure.

This issue was addressed in Kafka 0.11.0.0 as part of KIP-134[29]. A new broker property — `group.initial.rebalance.d` — was introduced for the group coordinator to allow additional time for consumers to join before sending `JoinGroupResponse` messages, thereby reducing the number of rebalances. The delay is only applied when the group is empty; membership changes in a non-empty group will not incur the initial rebalance delay. The default value of this property is `3000` (3 seconds).

A drawback of the initial rebalance delay is its treatment of a singleton group, where only one member exists. This is typical of test scenarios and local development. When using Kafka for small-scale testing or development, it is recommended that `group.initial.rebalance.delay.ms` is set to `0`.

Changing group membership

The coordinator maintains a generation ID — a monotonically-increasing 32-bit integer — corresponding to a point-in-time snapshot of the group membership state. When a new member wishes to join an existing group, they will send a `JoinGroupRequest` to the coordinator. Conversely, a member may gracefully leave the group at any point by sending a `LeaveGroupRequest`. Either action will result in the coordinator entering the `Joining` state, awaiting for *all* remaining members to rejoin the group.

While in the `Joining` state, the coordinator will reject heartbeats and other operations with a `REBALANCE_IN_PROGRESS` error. This will be detected by the remaining members upon their next heartbeat attempt, forcing them to rejoin with their existing member IDs. Once all members have successfully rejoined, the coordinator will increment the generation ID and transition to the `AwaitSync` state — initiating the synchronisation phase of the protocol and causing the rebalancing of partition assignments on the group leader. Upon successful rebalancing and state synchronisation, the coordinator will once again settle in the `Stable` state.

It may be possible for a member to miss a generational change in group membership — for example, due to a delayed heartbeat — and thereby be forcibly excluded from the group by the coordinator. The member might still legitimately believe it to be a part of the group and may attempt to fetch records or send commit requests. These members are referred to as 'zombies'. The use of an incrementing generation ID on the coordinator acts as a fencing mechanism — rejecting requests with an `ILLEGAL_GENERATION` error where the presented generation ID does not match the coordinator's generation ID. This forces zombie consumers to discard their assignments, clear their member ID, and rejoin the group from a clean slate.

[29] https://cwiki.apache.org/confluence/display/KAFKA/KIP-134%3A+Delay+initial+consumer+group+rebalance

State synchronisation barrier

When transitioning partition assignments from one consumer to another — following a member join or leave event — the consumers must collectively ensure that there is no time-overlap between the assignment of a partition to a new consumer and the release of the partition from its previous holder. While the time-separation between the revocation and the assignment is undesirable from a performance standpoint, allowing the two events to overlap in time is unacceptable. Therefore, we must conclude that some degree of separation is necessary and acceptable.

To achieve this separation, a consumer client must inform its user of any impending changes to the partition assignments, which is done through an `org.apache.kafka.clients.consumer.ConsumerRebalanceListene` callback interface, listed below sans the Javadoc comments.

```
public interface ConsumerRebalanceListener {
  void onPartitionsRevoked(Collection<TopicPartition> partitions);

  void onPartitionsAssigned(Collection<TopicPartition> partitions);

  default void onPartitionsLost(
      Collection<TopicPartition> partitions) {
    onPartitionsRevoked(partitions);
  }
}
```

A rebalance listener is registered with a `KafkaConsumer` client when calling one of its overloaded `subscribe()` methods.

After completing the group join, and just before initiating the state synchronisation phase, a consumer will assume that all partitions that it presently holds will be revoked by the leader, and will invoke the `onPartitionsRevoked()` callback accordingly, passing it the complete set of partitions that the consumer had owned prior to the rebalance. The execution of the callback may block for as long as necessary, waiting for the application to complete any in-flight activities. Only when the callback completes, will the client send a `SyncGroupRequest` message to the coordinator. (The synchronisation behaviour is different when incremental cooperative rebalancing is enabled, which will be discussed shortly.)

Later, upon completion of the state synchronisation phase, the consumer will learn of its updated assignments and will invoke `ConsumerRebalanceListener.onPartitionsAssigned()` with the partitions that the consumer now owns. (This may include partitions it previously owned if they haven't been reassigned.) The `onPartitionsRevoked()` method effectively acts as a barrier — satisfying the strict requirement that the `onPartitionsRevoked()` callback is invoked on *all* consumers before `onPartitionsAssigned()` is invoked on *any* consumer.

Without this barrier in place, it would have been possible for the new assignee of a partition to commence the processing of records before the outgoing consumer has had a chance to relinquish

control over that partition. The barrier might result in a short period of time where none of the partitions are being processed during the rebalancing phase. Kafka's design documentation colloquially refers to this as a *stop-the-world* effect, akin to its namesake in the area of garbage collection (GC) algorithms. Indeed, a stop-the-world rebalance in Kafka has the same drastic effect on the throughput of consumers as the GC has on the application as a whole. And unlike GC, which affects one process at a time, a stop-the-world rebalance affects all consumers simultaneously.

The final callback method — `onPartitionLost()` — signals to the application the loss of prior partitions due to a forced change of ownership. This will happen when a consumer fails to emit a heartbeat within a set deadline, thereby being considered 'dead' by the coordinator. The implications of partition loss are described in detail in the subsequent section on *liveness and safety*.

Incremental cooperative rebalancing

One of the main drawbacks of the original state synchronisation protocol, also known as the *eager rebalancing protocol*, is that it only allocates one phase to the entire rebalancing operation, which is one request-response message exchange in practical terms. This results in consumers taking an overly pessimistic view on the revoked partitions — forced to assume that all partitions might be revoked — whereas in reality only a subset of the partitions will typically be reassigned following a rebalance. This unnecessarily triggers the consumers to clean up all partitions, extending the duration of the rebalancing phase, and therefore, the extent of the stop-the-world pause.

Incremental cooperative rebalancing is a recent improvement of the state synchronisation protocol, introduced in Kafka 2.4.0 under KIP-429[30]. Under the cooperative model, the join and rebalance phases may be repeated, each introducing an incremental change to partition assignment. The cooperative rebalancing protocol introduces a second, follow-up join-rebalance round immediately following the first join-rebalance.

Cooperative rebalancing adds a new rebalance protocol version, which is referred to by the `COOPERATIVE` constant (`0x01` byte representation), in contrast to the original `EAGER` constant (`0x00`). The use of cooperative rebalancing is optional, enabled automatically in response to the chosen rebalancing strategy. (Rebalancing strategies will be discussed in one of the subsequent sections.) When the cooperative protocol is in force, the consumer does not invoke the `onPartitionsRevoked()` callback before sending the `SyncGroupRequest`; instead, the callback is deferred to such time when the `SyncGroupResponse` is received, and only for a non-empty set of revoked partitions. The client will then invoke the `onPartitionsAssigned()` callback, passing it the set of newly assigned partitions, even if that is an empty set. Having processed the callbacks, the client will initiate a second rejoin, followed by a subsequent rebalance.

The trick to making this work is to separate the revocations from the assignments by exactly one round, so that the first join-rebalance round comprises exclusively of revocations, or of assignments where no prior assignees exist (as in the case of the initial join). The second join-rebalance round comprises only of new assignments where a revocation was communicated in the previous round.

[30] https://cwiki.apache.org/confluence/display/KAFKA/KIP-429%3A+Kafka+Consumer+Incremental+Rebalance+Protocol

Thus, for any given partition, the `onPartitionsRevoked()` callback will always complete on the outgoing assignee before the `onPartitionsAssigned()` callback is instigated on the new assignee.

Between the two rounds, the group is said to be in an *unbalanced* state — not all partitions may be assigned to a consumer, even if there are sufficient members in the group. By contrast, the eager rebalance protocol will ensure that a partition is always assigned to a consumer at the conclusion of a synchronisation round, providing that at least one group member exists. The unbalanced state is resolved upon the completion of the second join-rebalance round.

The main benefit of cooperative rebalancing is in communicating the *exact* revocations to consumers, rather than coercing them towards a worst-case assumption. By reducing the amount of cleanup work consumers must perform during the `onPartitionsRevoked()` callback, the duration of the rebalancing stage is reduced, as is the effect of the stop-the-world pause.

A secondary benefit is the marked improvement of rebalancing performance and reduced disruptions to consumers in the event of a *rolling bounce* — where a rolling restart of a consumer population results in repeat leave, join, and state synchronisation events. Rebalancing still occurs, but the impact of this is greatly reduced as only the bounced clients' partitions get shuffled around.

> In theory, the protocol may be extended in the future to accommodate an arbitrary number of join-rebalance rounds and may even space them out if necessary. In the (hypothetical) extended protocol, the balanced state would be incrementally converged on after the completion of *N* rounds.
>
> Another prospective enhancement would be the addition of a 'scheduled rebalance timeout', granting temporarily departing consumers a grace period within which they can rejoin without sacrificing prior partition assignments. This is meant to combat the full effects of a rolling bounce by avoiding revocations.
>
> In addition, a scheduled rebalance timeout could deal with self-healing consumers, where the failure of a consumer is detected by a separate orchestrator and the consumer is subsequently restarted. (Kubernetes is often quoted as a stand-in for that role.) With a sufficiently long timeout, the orchestrator can detect the failure of a consumer process and restart its container without penalising the process by way of revoking its partitions. This allows the failed process to be restored faster and also minimises the impact on the healthy consumers. (Similar characteristics can be achieved using the existing static membership protocol, as we will shortly see.)

The main challenge of the cooperative model is the added join-rebalance round, leading to more network round-trips. For cooperative rebalancing to be effective, the underlying partition assignment strategy must be *sticky* — in other words, it must preserve the prior partition assignments as much as possible. Without a sticky rebalancing strategy, the cooperative model reduces to the eager rebalancing model, albeit with more overhead and complexity.

A word of caution: Enabling the cooperative protocol changes the semantics of the `ConsumerRebalanceListener` callback. Under the eager rebalance protocol, the `onPartitionsAssigned()` is handed the complete set of assigned partitions; whereas, under the cooperative protocol, the `onPartitionsAssigned()` is given just the set of new partitions that were acquired since the last rebalance.

Static membership

Static group membership takes a different approach to the conventional (join-triggered) partition assignment model by associating group members with a long-term, stable identity. Under the static model, members are allowed to leave and join a group without forfeiting their partition assignment or causing a rebalance, provided they are not away for longer than a set amount of time.

The main rationale behind static membership is twofold:

1. To reduce the impact of rebalancing and the associated interruptions when *inelastic* consumers bounce; and
2. To allow for an external health checking and healing mechanism for ensuring liveness of the consumer ecosystem.

The second point will be discussed later in this chapter, in the broader context of *liveness* and *safety* concerns. For the time being, we shall focus on the first point.

An 'inelastic' consumer is one that is expected to persist for an extended period of time, occasionally going down for a short while — for example, to perform a planned software deployment or as a result of an intermittent failure. In both cases, certain assumptions might be made as to the duration of the outage. The population of inelastic consumers tends to be more or less fixed in size. By contrast, an 'elastic' consumer is likely to be spawned in response to an elevated load demand and will be terminated when the demand tapers off.

Static assignment works by making minor amendments to the group management and state synchronisation protocol phases. It also adds a property — `group.instance.id` — to the consumer configuration, being a free-form stable identifier for the consumer instance.

Upon joining the group, a static member will submit its member ID in a `JoinGroupRequest`, along with the group instance ID in the member metadata. If the member ID is empty — indicating an initial join — the coordinator will issue a new member ID, as per the dynamic membership scenario. In addition to issuing a member ID, the coordinator will take note of the issued ID, associating it locally with the member's group instance ID. Subsequent joins with an existing member ID will proceed as per the dynamic group membership protocol.

Having joined the group, the consumer will impart its group instance ID to the group leader, via the coordinator. Like dynamic members, static members will have a different member ID on each

join, rendering it largely useless for correlating partition assignments across generations. The built-in assignor implementations have added provisions for this — using the group instance ID, where one is supplied, in place of the member ID.

Unlike its dynamic counterpart, a static consumer leaving the group will not send a `LeaveGroupRequest` to the coordinator. When the consumer eventually starts up, it will simply join with a new member ID, as per the explanation given above. The coordinator will still enforce heartbeats as per the consumer-specified `session.timeout.ms` property. If a bounced consumer fails to reappear within the heartbeat deadline, it will be expunged from the group, triggering a rebalance. In other words, there is no difference in the health check behaviour between the static and dynamic protocols from the coordinator's perspective. The session timeout is typically set to a value that is significantly higher than that for an equivalent dynamic consumer. The reasons for this will be discussed later.

With the recent introduction of incremental cooperative rebalancing and the support for static group membership, it is likely that the `group.initial.rebalance.delay.ms` property will be deprecated in the near future.

One of the challenges with the static group membership model is the added administrative overhead — namely, the need to provision unique group instance IDs to each partaking consumer. While this could be done manually if need be, ideally, it is performed automatically at deployment time. At any rate, a blunder in the manual process or a defect in the deployment pipeline might lead to a situation where two or more consumers end up with identical group instance IDs. This begs the question: will two identically-configured static consumers lead to a non-exclusive partition assignment?

Kafka deals with the prospect of misconfigured consumers by implementing an internal fencing mechanism. In addition to disclosing its group instance ID in the group membership phase, the consumer will also convey this information in `SyncGroupRequest`, `HeartbeatRequest`, and `OffsetCommitRequest` messages. The group coordinator will compare the disclosed group instance ID with the one on record, raising a `FENCED_INSTANCE_ID` error if the corresponding member IDs don't match, which propagates to the client in the form of a `FencedInstanceIdException`.

Fencing is best explained with an example. Suppose consumers *C0* and *C1* have been misconfigured to share an instance ID *I0*. Upon its initial join, *C0* presents an empty member ID and is assigned *M0* by the coordinator. The coordinator records the mapping *I0* ⟶ *M0*. *C1* then does the same; the coordinator assigns *M1* to *C1* and updates the internal mapping to *I0* ⟶ *M1*. When *C0* later attempts to sync with the *I0*-*M0* tuple, the coordinator will identify the offending mapping and respond with a `FENCED_INSTANCE_ID` error.

If *C0* completes both a join and a synchronisation request before *C1* does its join, then the inconsistency will be identified during the synchronisation barrier, just prior to the coordinator replying with a `SyncGroupResponse`. Either way, one of the consumers will be fenced.

Liveness and safety

It would be rather nice if consumers and brokers never failed, networks were reliable, and group membership changes were always cleanly demarcated by join and leave requests. Unfortunately, this cannot be; we have to contend with the bleak harshness of reality, which is particularly exacerbated when dealing with distributed systems.

In the realm of distributed systems, like in concurrent computing, there are two fundamental properties of interest: *liveness* and *safety*.

Liveness is a property of a system that requires it to make progress despite its internal components competing for shared resources. A system that exhibits liveness will guarantee that something 'good' will *eventually* happen.

Safety is an orthogonal property that guarantees that none of the critical invariants of a system will ever be violated. In other words, something 'bad' will *never* happen under a system that exhibits safety.

The 18th century English jurist William Blackstone expressed a formulation that has since become a staple of legal thinking in Anglo-Saxon jurisdictions. Blackstone's formulation was: *It is better that ten guilty persons escape than that one innocent suffer.*

Sir William Blackstone (1723 – 1780)

John Adams — the second president of the United States — remarked on Blackstone's formulation that, paraphrasing: *Guilt and crimes are so frequent, that all of them cannot be punished, whereas*

the sanctity of innocence must be protected. One way of looking at this remark, although it was not conveyed verbatim, is that the guilty will likely reoffend, and therefore will likely be caught eventually. Conversely, if the innocent were to lose faith in their security, they would be more likely to offend, for *virtue itself is no security*, to use Adams's own words.

John Adams (1735 — 1826)

Unifying Blackstone's formulation with Adams's, we can derive the following: *The guilty should be punished eventually, while the innocent must never be punished.* How does this relate to our previous discussion on distributed systems? As it happens, the aforementioned statement combines both *liveness* and *safety* properties. Enacting of punishment for the guilty, at some indeterminate

> point in time, is a manifestation of *liveness*. Ensuring that the innocent are never punished is *safety*.
>
> Interestingly, the liveness property may not be satisfied in a finite execution of a distributed system because the 'good' event might only theoretically occur at some time in the future. Eventual consistency is an example of a liveness property. Returning to the Blackstone-Adams formulation, there is no requirement that the guilty are caught within their lifetime, or ever, for that matter.
>
> What is the point of a property if its cardinal outcome may never be satisfied? Formally, the liveness property ensures that an eventual transition to a desirable state is theoretically possible from every conceivable state of the system; in other words, there is no state which categorically precludes the 'good' event from subsequently occurring. Theoretically, there is no crime that cannot be solved.
>
> By comparison, the safety property can be violated in a finite execution of a system. Just one occurrence of a 'bad' event is sufficient to void the safety property.

Kafka gives us its assurance, that providing at least one consumer process is available, then all records will eventually be processed in the order they were published, and no record will be delivered simultaneously to two or more competing consumers. Granted, in reality, this guarantee only holds if you use Kafka correctly. We have witnessed numerous examples in Chapter 10: Client Configuration where a trivially misconfigured, or even a naively-accepted default property will have a drastic impact on the correctness of a system. Let us look the other way for the moment, blissfully pretending that clients have been correctly configured and the applications are defect-free.

How does all this relate to liveness and safety? The preservation of record order and the assignment of any given partition to at most one consumer is a manifestation of the *safety* property. The eventual identification of a failed consumer and the prompt rebalancing of partition assignments takes care of *liveness*.

Kafka satisfies the liveness property in two ways:

1. **Checking availability** — by requiring consumers to periodically exchange heartbeats with the group coordinator, thereby indicating that the consumer process is running on the host and that a network path is available between the consumer process and the coordinator node.
2. **Checking progress** — by requiring that the consumer periodically invoke `poll()`, thereby indicating that it is able to handle its share of the partition assignment.

The consumer will send a `HeartbeatRequest` message at a fixed interval from a dedicated heartbeat thread. The coordinator maintains timers for each consumer, resetting the clock when a heartbeat is received. Having received the heartbeat, the coordinator will reply with a `HeartbeatResponse`. Conversely, if the coordinator does not receive a heartbeat within the time specified by the `session.timeout.ms` consumer property, it will assume that the consumer process has died. The coordinator will then transition to the `Joining` state, causing a rejoin of all remaining members. While the session timeout is configured on the consumer, it is communicated to the coordinator inside a `JoinGroupRequest` message, with the deadline subsequently enforced by the coordinator.

The `session.timeout.ms` property defaults to `10000` (10 seconds). It is recommended that this value is at least three times greater than the value of `heartbeat.interval.ms`, which defaults to `3000` (3 seconds). As a further constraint, the value of `session.timeout.ms` must be in the range specified by the `group.min.session.timeout.ms` and `group.max.session.timeout.ms` broker properties, being `6000` (6 seconds) and `1800000` (30 minutes), respectively, by default. By binding the range of the session timeout values on the broker, one can be sure that no client will connect with its `session.timeout.ms` setting outside the permitted range.

While the availability check is performed on the coordinator node, the progress check is performed locally on the consumer. The background heartbeat thread will maintain its periodic heartbeating for long as the last recorded poll time is within the deadline specified by the `max.poll.interval.ms` property, which defaults to `300000` (5 minutes). If the application fails to invoke `KafkaConsumer.poll()` within the specified deadline, the heartbeat thread will cease, and a `LeaveGroupRequest` will be sent to the coordinator. Stated otherwise, the heartbeat thread doubles up as a watchdog timer — as soon as the poll deadline is missed, the client internally 'self-destructs', causing a leave, followed by a rejoin.

The client should assign `session.timeout.ms` and `heartbeat.interval.ms` values based on the application's appetite for failure detection. Shorter heartbeat timeouts result in quicker failure detection at the cost of more frequent consumer heartbeating, which can overwhelm broker resources and lead to false-positive failure events — whereby a live consumer is mistakingly declared dead. Longer timeouts lead to less sensitive failure detection, but may leave some partitions unhandled in the event of consumer failure.

A certain degree of process predictability and some amount of slack in the session timeout are required for a consumer to consistently meet its heartbeat deadline — the *"prove to me that you are still alive"* check. Given a suitably reliable network and unsaturated consumer processes with regular garbage collector (GC) activity, the heartbeating process can be made to work reliably and predictably — even in the absence of a genuinely deterministic runtime environment. In the author's experience, the default values for the heartbeat interval and the session timeout are sufficient in most scenarios, occasionally requiring adjustments to compensate for heavily loaded consumer applications or network congestion.

The situation with the progress check — the *"prove to me that you are still consuming records"* audit — is somewhat more challenging, as it relies on a round of polling to complete within a bounded time. Depending on what the consumer does with each record, there is little one can do to ensure that the processing of records will unconditionally complete within a set timeframe.

When faced with nondeterministic record processing, one way of improving the situation is to cap the number of records that will be returned to the application by `KafkaConsumer.poll()`. This is controlled by the `max.poll.records` consumer property, which defaults to `500`. This doesn't change the way records are fetched from the broker, nor their quantity; instead, the `poll()` method artificially limits the number of records returned to the caller. The excluded records will remain stashed in the fetch buffer — to be returned in a subsequent call to `poll()`.

In reducing the number of records returned from `poll()`, the application effectively slackens its

processing obligations between successive polls, increasing the likelihood that a cycle will complete before the `max.poll.interval.ms` deadline elapses, particularly if the average and worst-case processing times of a record are well known.

One might think that liveness and safety only apply to niche areas — low-level systems programming, operating systems, and distributed consensus protocols. In actual fact, we are never completely insulated from these properties — we might just experience their effects in different and often subtle ways, sometimes barely realising it.

Dealing with failures

Whilst an average (mean or median) processing time can often be derived empirically, the distribution of processing times may, and often does, exhibit a 'long tail'. In other words, the worst-case processing time might not be bounded. To illustrate this, consider a fairly typical event streaming example where the processing of a record results in the updating of a database or perhaps invoking some downstream service. These sorts of operations are inherently fault-prone and may time out, requiring retries. In the worst-case scenario, a downstream dependency might be experiencing downtime, leaving the consumer in an indefinite retry loop. Clearly, even with one backlogged record, the consumer might not make the `max.poll.interval.ms` deadline. So what to do?

Unfortunately, Kafka does not have an answer of its own. The user is left to fend for themselves. As it happens, there are five strategies, at least, for dealing with this:

1. Set an absurdly large `max.poll.interval.ms` value, to effectively disable the progress check.
2. Maintain a reasonable progress deadline, allowing Kafka to detect progress failure and rebalance the group.
3. Detect an impending `max.poll.interval.ms` deadline and voluntarily relinquish the subscription at the consumer.
4. Implement a record-level deadline within the consumer, such that if a record fails to make progress within a set timeframe, it is recirculated back to the tail-end of the original topic.
5. Implement a record-level deadline; the record will be skipped if it is unable to make the deadline, and, ideally sent to a dead-letter topic for subsequent post-mortem.

The first option takes the stance of *"do it 'till it's done, whatever it takes"*. This may be appropriate if the correctness of the application depends on having every record processed and in the order that the records appear in the topic. We are effectively saying that there is no point progressing if the process cannot do its job, so it might as well stall until the downstream issue is rectified.

The main drawback of this approach is that it assumes that failure to make progress can only be attributed to an external cause, such as the failure of a downstream dependency. It fails to take into account potential failures on the consumer, such as a software defect that may have caused a deadlock, or some other *local* contingency that is impeding progress.

The second option yields a similar outcome to the first, in that it will not progress the consumption of records on the partition containing the troublesome record. The progress check is still in force, meaning that the poll loop will time out, and the coordinator will exclude the consumer from the group, triggering a partition rebalance. The new assignee of the troublesome partition will presumably experience the same issue as the outgoing consumer, and time out in the same manner. The previous assignee would have rejoined the group, and, in doing so, will have become eligible for another share of partition assignments. This cycle will continue until the downstream issue is resolved and the consumers are eventually able to make progress. The advantage of this approach is that it makes no assumptions as to the cause of the problem. If the progress was originally impeded by a deadlocked consumer, then the new consumer should proceed without a hitch. It also requires no additional complexity on the consumer's behalf.

This approach is idiomatic to Kafka, but it is not without its drawbacks. By rebalancing the partitions and letting the new assignee time out, the overall throughput of the topic may be impacted, causing periodic stutter. Another, more serious drawback, is that although Kafka will detect consumer failure, the consumer itself might not. Kafka will transfer ownership of the impacted partitions to the remaining consumers, while the outgoing consumer might assume that it still holds those partitions and will slowly work through its backlog. This problem is explored in more detail in the section titled *'Dealing with partition exclusivity'*.

The third strategy — voluntarily relinquishing a subscription — is an evolution of the second approach, adding timing logic and progress checks to the poll-process loop, so that the consumer process can preempt an impending deadline failure. This would involve proactively cleaning up its state and unsubscribing. The preemption mechanism implies either some form of non-blocking processing of records or the ability to interrupt the poll-process thread so that it does not block indefinitely. Once a subscription is forfeited, the consumer immediately resubscribes — resetting its state within the group.

The benefit of this approach is that it maintains the strongest safety guarantees with respect to record ordering and at-least-once delivery — much like the first two approaches. Akin to the second strategy, it satisfies the liveness property by proactively yielding its subscription, while still allowing the coordinator to detect consumer failures. This strategy has the added safety advantage of enforcing record processing exclusivity by adding a conservative local failure detection mechanism, ensuring that record processing does not overrun.

A minor drawback of this model is that it causes a rebalance as part of forfeiting the subscription, shortly followed by a second rebalance as the consumer rejoins the pack. Each rebalance will disrupt *all* consumers, albeit for a short time. In Kafka parlance, this is referred to as 'bouncing' — whereby consumers come in and out of the group, causing no net effect, but forcing an unnecessary reshuffling of partition assignments. To be fair, the behaviour of this strategy with respect to rebalancing is no worse than strategy #2. Note that the effect of rebalancing under this strategy cannot be ameliorated by employing a static membership model because unsubscribing from a topic will cause an implicit rebalance.

The fourth approach — implementing a record-level deadline — essentially involves starting a timer just before processing a record and ensuring that no operation blocks beyond the processing deadline.

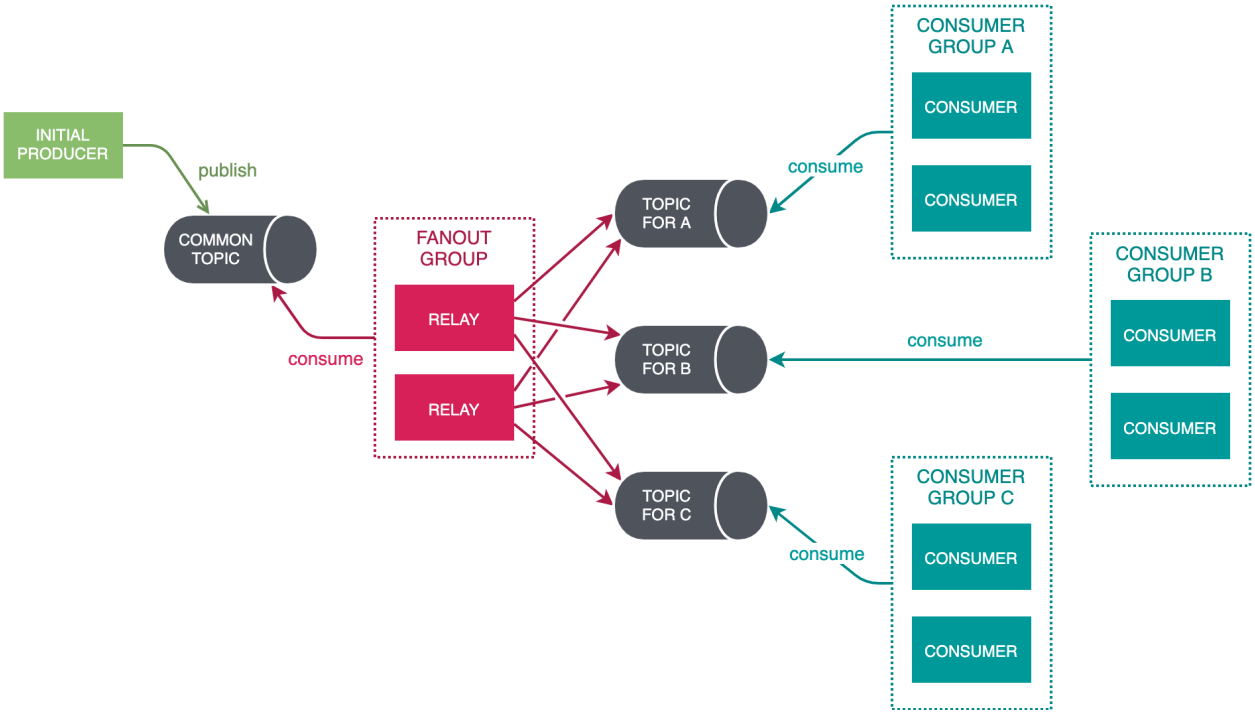
When this deadline is reached, the consumer takes corrective action. In this variant, the corrective action involves republishing the record to its original topic, such that it will be dealt with *eventually* by some consumer — the consumer that republished the record or some other consumer if the topic's partitions were reassigned between re-queuing the record and its subsequent consumption.

The record-level timeout must be significantly shorter than `max.poll.interval.ms` for it to be effective. In fact, it should be just under the quotient of dividing `max.poll.interval.ms` by `max.poll.records` — allowing for the worst-case scenario for all records in the batch. (*Worse than worst*, if such a phrase is grammatically admissible.)

This model is predicated on the consumers' capacity to arbitrarily reorder records; in other words, it only works if preserving the original record is not essential to the correct operation of the system. This is typically not the case in most event streaming scenarios, but it may be suitable when records are entirely self-contained and independent of one another, or when the records can be trivially reordered on the consumer. It should also work well in more traditional peer-to-peer message queuing scenarios, where messages might correspond to discrete tasks that can be processed out of order. Consider, for example, a queue-based video transcoding application, where each record is a pointer to a media file stored in an object store (such as AWS S3) along with instructions for the target format, resolution, bitrate, etc. Now, suppose the consumer-end of the transcoder requires a licensed codec for a specific video, and is experiencing a problem with a downstream license server. It would be reasonable for the transcoder to re-queue the record, leaving it in the queue until such time that the license server is back online.

There are several potential variations of the strategy above. The producer might include an expiry time in the record indicating how long the task should be in a queue before it is deemed useless to the business. The consumer would only attempt to re-queue the record for as long as it hasn't outlived its useful lifespan. In another variation, the consumer may want to limit the number of retry attempts. This can be accomplished by setting a retry counter when the record is first published, validating and decrementing it on each subsequent re-queuing attempt. When the record no longer qualifies for re-queuing, it will be discarded. The consumer might also wish to log this fact and publish a copy of the discarded record to a dead-letter topic.

Republishing records mutates the topic for *all* consumer groups. As such, reordering strategies are only effective for point-to-point messaging scenarios, involving just one consumer group. In order to apply a reordering strategy to (multi)point-to-multipoint scenarios, where multiple consumer groups share a source topic, it is necessary to implement a *fan-out* topology. This involves transforming a (multi)point-to-multipoint topic into a (multi)point-to-point topic, coupled to several point-to-point topics — one for each consumer group. The fan-out model eliminates sharing — each consumer group gets a private topic which it may mutate as it chooses.



Topic fanout

The final strategy is a limiting case of strategy #4, where the retry attempts counter is set to zero. In other words, a record that fails to complete within the allotted deadline is discarded immediately, with no second-chance re-queuing. This is the most relaxed model, which guarantees progress under a significantly relaxed notion of safety.

Dealing with partition exclusivity

If was previously mentioned that Kafka's assurance extends to the exclusive processing of records; namely, that *at most one consumer will be allocated to a partition, for any given consumer group.*

There is a subtle caveat here, albeit a crucial one — concealed in the wording of the statement above. The assurance is given only with respect to partition assignment; it doesn't cover the act of concurrently processing the record, despite what most Kafka practitioners might like to believe.

Specifically, the problem is this: consider two consumers *C0* and *C1* in a common group, contending over one partition *P*. *P* is initially assigned to *C0*, and the group coordinator has a stable view of the group membership, being the set containing *C0* and *C1*.

At some point in the course of processing records, *C0* is unable to satisfy its progress check while working through a batch of records. Furthermore, *C0* is blocked while processing one of the records in the batch, with more records remaining. The `KafkaConsumer` instance that is backing *C0* will detect that the application has failed to invoke `poll()` within the progress deadline, and the heartbeat thread will consequently send an explicit leave request. However, the consumer remains blocked on whatever operation it was doing, oblivious to the background goings-on of the heartbeat thread.

Having received the leave request, the coordinator will adjust the group membership to a singleton set comprising *C1*, which will be followed by a partition reassignment. It is possible that *C0* may have rejoined the group, but that hardly matters — *P* has been reassigned to *C1*. When *C0* eventually unblocks, it will proceed with its backlog, naively assuming that it is still the owner of *P*. In the worst-case scenario, *C0* could enter a critical section, where the effect of processing a record on *C0* may conflict with the concurrent or earlier processing of an identical record on *C1*.

This demonstrates that although the partition was assigned exclusively from the leader's perspective, the restrictions under this assignment were not reflected commensurately on the consumers. Under normal circumstances, when the consumer population changes, the new and existing consumers are notified of changes to their partition assignments via an optional `ConsumerRebalanceListener` callback. The callback, which will be discussed in detail later, essentially informs a consumer that its existing partitions were revoked and that new partitions were assigned. Crucially, the callback is globally blocking in the revocation phase — no consumer is allowed to proceed with their new partition assignment until *all* consumers have handled the revocation event. This gives a consumer a vital opportunity to complete the processing of any in-flight records, gracefully handing its workload over to the new consumer. However, *the callback is executed in the context of the polling thread*, during its next call to `KafkaConsumer.poll()`. In our earlier example, *C0* was blocked — clearly, there was no way it could have called `poll()`. Furthermore, as it was forcibly excluded from the group, the consumer would not have been eligible to partake in the blocking revocation callback; a member that is deemed dead is not able to affect the remaining population of the group. (Its abilities will be restored when it rejoins the group, by which time the group would have undergone at least one rebalance.)

Kafka 2.4.0 introduces another `ConsumerRebalanceListener` callback method — `onPartitionsLost()` — designed to indicate to the consumer that its partitions have been forcibly revoked. This method is a best-effort indicator only; unlike the `onPartitionRevoked()` method, it does not act as a synchronisation barrier — it cannot stop or undo the effects of the revocation, as the new assignee will likely have started processing records by the time the outgoing consumer has reacted to `onPartitionsLost()`. Furthermore, like its peer callback methods, `onPartitionsLost()` executes in the context of the polling thread, invoked from within a call to `KafkaConsumer.poll()`. Since the failure scenarios assume that the consumer may be blocked indefinitely, the usefulness of this callback is questionable. Still, at least now Kafka informs us of the calamity that ensued, whereas in previous versions the consumer had to discover this of its own accord. More often than not, the effects of two consumers operating on the same records were discovered by disheartened users of the system, sometimes long after the fact.

By default, and to maintain compatibility with pre-2.4.0 consumer applications, the `onPartitionsLost()` callback simply delegates to `onPartitionRevoked()`. However, the roles of these callback methods are vastly different. When upgrading a consumer application that implements the `ConsumerRebalanceListener` callback to utilise version 2.4.0 (or newer) of the client library, one should immediately override the default implementation.

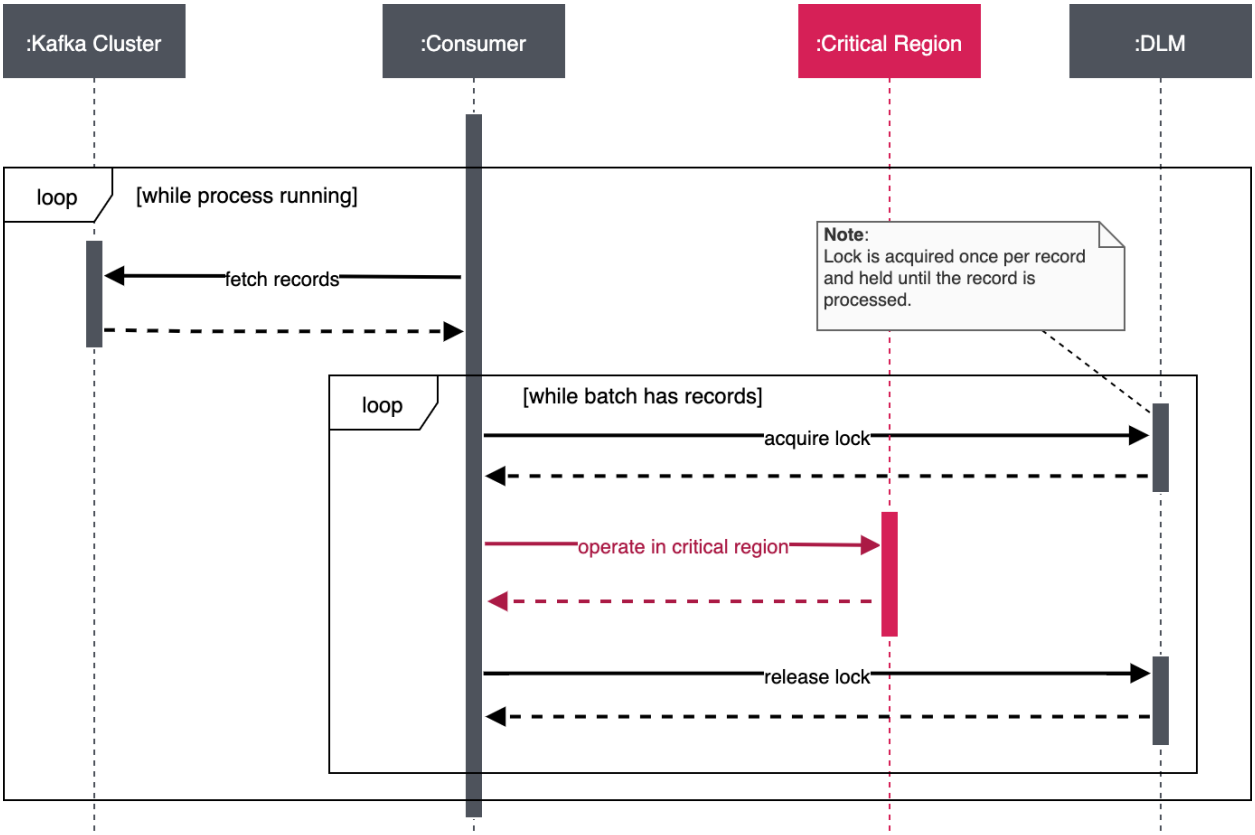
So how does one prevent the non-exclusive assignment condition? Again, Kafka is coy on answers.

Once more, we revert to outside strategies for dealing with partition exclusivity. There are three at our disposal:

1. Ensure that the poll loop always completes within the progress deadline set by `max.poll.interval.ms`, thereby sidestepping the problem.
2. Employ an external fencing mechanism for dealing with critical sections.
3. Employ fencing at the process level, terminating the outgoing process, or isolating it from the outside world. This may be used in addition to #2.

The first strategy has already been covered in the previous section. It tries to avoid a deadline overrun by either yielding the subscription, shedding the load, or requeueing records. It is not a universal strategy, in that it has drawbacks and might not apply in all cases.

The second strategy relies on an external arbitration mechanism — typically a distributed lock manager (DLM) — used to ensure exclusivity over a critical section. Before processing a record, the consumer process would attempt to acquire a named lock with the DLM, proceeding only upon successful acquisition. The name of the lock can be derived from the record's key or the partition number. This aligns critical sections to partitions and ensures that only one consumer may operate within a critical section at any one time; however, it does not preclude the record from being processed multiple times, as the two consumers may have handled the record at different, non-overlapping times. This is illustrated in the sequence diagram below.



Critical region — per-record

Having entered the critical section, the consumer will check that the effects of the record haven't already been applied by some other process. If it detects that the record has already been processed elsewhere, the consumer can safely skip the record and move on to the next.

In addition to promoting safety, the check for prior processing adds *idempotence* to record consumption — ensuring that the repeat processing of a record leads to no observable effects beyond those that were emitted during the first processing.

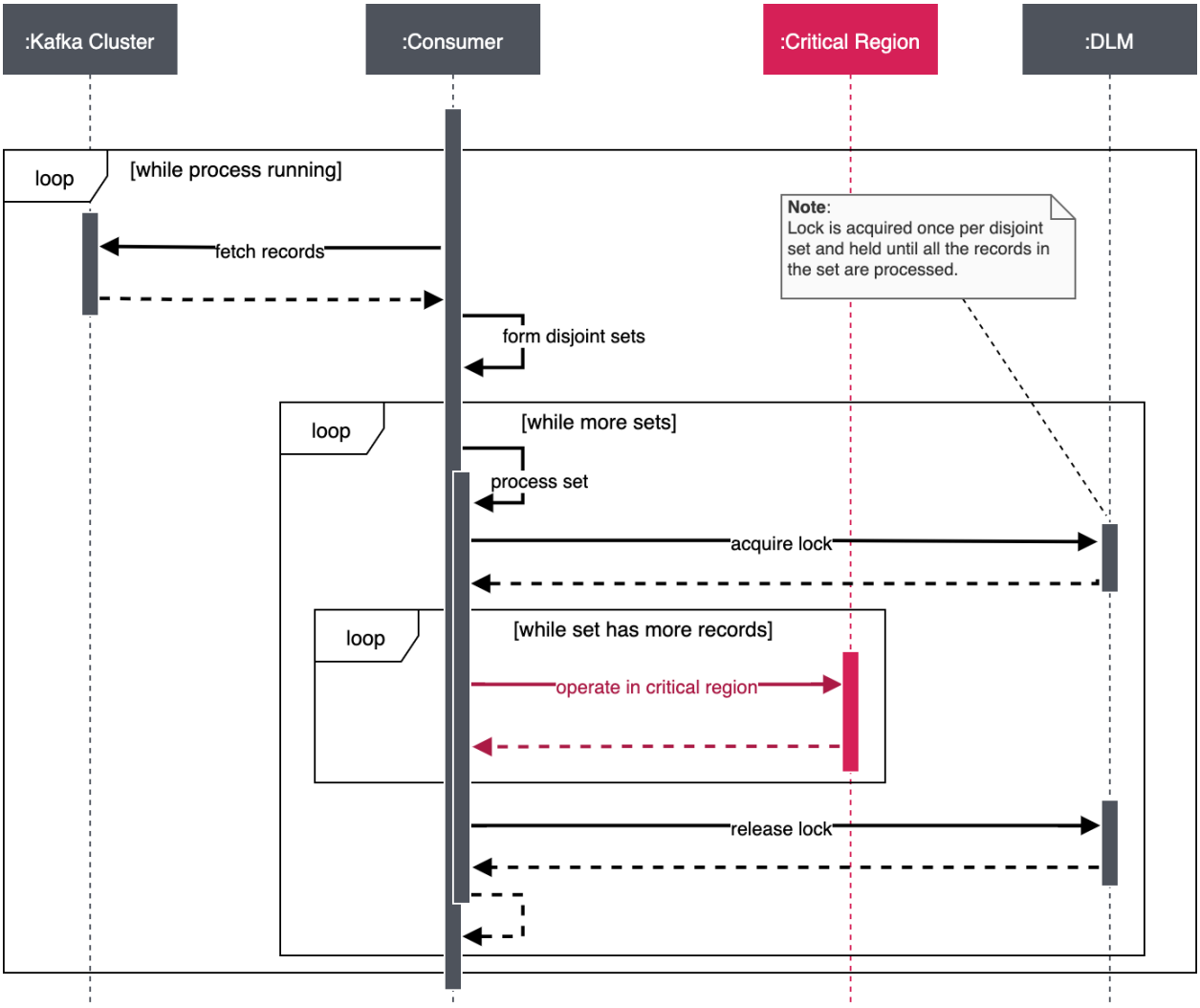
While traversing the critical section, a consumer's observation that a record has already been processed might lead to the following deduction. If one record has already been processed elsewhere, then it is likely due to a breach of the exclusivity constraint. Consequently, it is likely that multiple records have been processed elsewhere.

This deduction might lead to a tempting optimisation. Rather than trying the next record in the batch, the consumer might simply abort the batch altogether and unsubscribe from the topic, then resubscribe again. This will force a group membership change (it will cause two changes, to be precise) that will see the consumer receive a new partition assignment and start again.

However logical it might seem, this approach does not work. If the consumer is faced with a batch

> where the first, or indeed all records appear to be processed, then bailing out and resubscribing to the topic *might* advance the consumer's offsets to skip over the processed records, but it equally might not. This depends on whether the original consumer had committed its offsets in the first place. Consider a simple scenario involving a single partition with a number of records, which were previously processed by some consumer. The consumer crashed after processing the records but before committing its offsets. A new consumer takes over the topic, and receives the offset of the first record. It detects a duplicate and attempts to resubscribe, but the new assignment will give it the same starting offset — leading to a perpetual backoff cycle on the consumer. Having detected duplicate handling of a record, the correct procedure is to move on to the next record, doing so until the backlog has been worked through.

As a strategy, external fencing may be used in a broad range of scenarios as it does not mutate the topic or sacrifice any notion of safety. Other than the added complexity of requiring an external DLM, the drawback of fencing is its negative impact on throughput and latency. Since record processing must be surrounded by network I/O and persistent operations on the DLM, fencing may pose a challenge for both latency-sensitive and throughput-sensitive applications. This can be alleviated by reorganising records returned from `poll()` into disjoint ordered sets of key-related records, then processing these sets with dedicated locks. In the simplest case, the disjoint ordered sets may be derived from the partition number of the records — each record is allocated to a set based on its partition number. The sets are then processed either sequentially or concurrently on the consumer. Before processing each set, the consumer acquires a lock covering all elements in the set, releasing the lock upon the completion of the set. By coalescing visits to the critical section, the consumer reduces both the network I/O and the load on the DLM. This optimisation is illustrated in the sequence diagram below.



Critical region — disjoint sets

When designing a distributed consumer ecosystem, care must be taken to ensure that the DLM does not become a single point of failure. While focusing on safety, one must take care to not neglect liveness; if the DLM becomes unavailable, the entire consumer ecosystem will stall.

One does not have to use a dedicated DLM for arbitrating access to critical sections. Some persistent datastores, such as relational databases, Etcd, Redis, and Consul can act as locking primitives. (Any store that is both persistent and exposes a compare-and-swap operation may be used to construct a mutex.) In the case of a relational database (and some NoSQL systems), the database can be used to affect transactional semantics over a series of operations — using pessimistic locking or multi-version concurrency control. When the critical section is bound to a single database, the transactional capabilities of the database should be preferred over a DLM, as this minimises the amount of network I/O, simplifies the design of the application, and leads to a stronger consistency model.

> Notwithstanding the various controls one might employ to ensure partition exclusivity, it can be shown that in the absence of a deterministic system, the safety property cannot be unconditionally satisfied in all cases.
>
> Sources of nondeterminism in the execution of applications, such as interrupts, paging, heap compaction, garbage collection, and so forth, can lead to undetectable gaps in the evaluation of successive statements. Consider the following code, typical of 'safe' access to a critical section.

```
distributedLock.acquire();
try {
  criticalSection.enter();
  // ...
  criticalSection.leave();
} finally {
  distributedLock.release();
}
```

> The acquisition of an entry permit to a critical section may be invalidated before the section is subsequently entered due to an event that cannot be foreseen or corrected, for example, a garbage collector pause. In the example above, the `criticalSection.enter()` call may be preceded by an abnormally long pause, such that the DLM may deem the lock holder 'dead' and transfer the lock to the next contender. When the code eventually resumes executing and calls `criticalSection.enter()`, it will conflict with the new lock holder's actions and violate the safety property.

The third strategy — process-level fencing — augments strategy #2 by identifying and restarting (or otherwise isolating) processes that have had their partitions revoked as a result of a heartbeat failure. This can be accomplished using an external health check mechanism that continually queries the process on a well-known endpoint — to determine whether the process can satisfy its routine polling obligations. If the endpoint does not respond within a set timeframe, or if the response is negative, the process will be forcibly restarted. The health check should be tuned to respond negatively well before the `max.poll.interval.ms` deadline elapses, so that the process can be restarted before its death is detected on the coordinator and well before its partitions are reassigned.

Utilising static members

As mentioned earlier in this chapter, Kafka offers the notion of static group membership — allowing for an external health check and healing mechanism for ensuring the liveness of the consumer ecosystem. The requirement for static membership stems from the more contemporary use cases involving container orchestration engines such as Kubernetes. Under the 'Kubernetes' model, the container orchestrator is responsible for the ongoing health check of its subordinate containers, restarting them if a failure is detected.

Kubernetes, due to its overwhelming popularity, is often quoted as a ubiquitous 'placeholder' term for any control system that takes on additional health assurance responsibilities. When 'Kubernetes' is written in quotes, as in the preceding instance, one should assume that the conversation relates to any control system.

In the 'Kubernetes' deployment model, a container may be terminated spuriously as a result of a failed health check, only to be restarted shortly afterwards. It would be highly advantageous for the partitions assigned to the failed consumer to remain as such, with no intermediate reassignment, until the consumer rejoins the group. This model avoids a stop-the-world pause at the expense of individual partition availability — one or more partitions (depending on the failed consumer's assignment) will remain paused for some time. This may result in greater overall throughput and more pronounced positive skew in the distribution of response times — lower median latencies at the expense of a longer latency tail.

Under the static model, members are allowed to leave and join a group without forfeiting their partition assignment or causing a rebalance, providing that they are not away for longer than the `session.timeout.ms` deadline; failing to reappear within the heartbeat deadline causes a purge, triggering a subsequent rebalance. In this respect, there is no material difference in the liveness behaviour between the static and dynamic protocols from the coordinator's perspective.

Naturally, if one is using an external orchestrator for managing consumer health, it makes sense to desensitise the coordinator's own health check mechanism to prevent a premature expulsion of the bounced consumer, but still allow the coordinator act as a 'safety net' if the orchestrator does not detect and remediate the process failure in a timely manner. This can be achieved by setting the `session.timeout.ms` deadline to a value that exceeds the orchestrator's projected response time by some margin. The `max.poll.interval.ms` deadline would also be elevated accordingly.

The effect of `max.poll.interval.ms` is different under the static group membership scheme. The `KafkaConsumer` client will cease to heartbeat if the deadline is not met by the poll-process loop, but no further action will be taken — namely, the client will not send a leave request and no reassignment will occur. The stalled consumer process will be given up to `session.timeout.ms` to recover. If no heartbeat is received within the `session.timeout.ms` deadline, only then will the coordinator evict the failed consumer and reassign its partitions. In the course of introducing the static membership feature, the hard upper bound on `session.timeout.ms` was increased to `1800000` (30 minutes).

With the additional consumer health management options offered by static group membership, one might wonder whether this changes the liveness and safety landscape. The somewhat grim answer is: not by a lot.

The liveness property is fundamentally satisfied by identifying consumer failures. Whether the failure is identified by the coordinator or by an external process is mostly an implementation detail. A static model may reduce the impact of intermittent failure on the group, at the expense of individual partition stalls.

The safety property is achieved by conveying an exclusive partition assignment, on the assumption that the assignment is commensurately honoured on the consumers. We previously looked at process fencing as a mechanism for ensuring partition exclusivity among consumers. Static membership, coupled with an external orchestrator, offers a straightforward implementation path for the fencing strategy. The process health check will factor into account the last poll time — reporting a heavily backlogged process as failed. The orchestrator will restart the consumer process, causing it to rejoin with its durable group instance ID, restoring any prior assignments. The consumer's progress in the stream will be reset per the last committed offsets, allowing it to reprocess the last batch. The combination of locking critical sections and fencing processes effectively neutralises any residual likelihood of a lingering 'zombie' process.

Mixing membership models

The static membership model on its own is generally incompatible with the notion of consumer elasticity. With a programmatic assignment of the `group.instance.id` property, it is possible to accommodate the scale-up scenario. A newly spawned static consumer joining a group would acquire a share of partitions, working in much the same way as a dynamic consumer. The problem occurs on the scale-down path: a terminated static consumer will lead to partition unavailability for the period of `session.timeout.ms`, which has presumably been increased as part of switching to static membership. In other words, scale-up will be responsive while scale-down will be sluggish.

It is possible to mix elastic and inelastic consumers in the same consumer group. Consider a use case where an application is experiencing a mostly constant load throughout the day. Occasionally, the load peaks above the routine threshold, requiring additional burst capacity to process the stream. Assuming that throughput takes priority over latency and occasional individual partition pauses are acceptable, the consumer population might comprise a mixture of statically-configured and dynamic consumers. When utilising public cloud infrastructure, it is more cost-effective to provision long-term capacity for fixed loads, while utilising ephemeral instances (e.g. Spot Instances in AWS) for spillover capacity. Kafka's static membership feature dovetails nicely into this model.

When mixing consumer types, dynamic consumers should be configured differently for the `session.timeout.ms` property. Dynamic consumers will have a much shorter `session.timeout.ms` deadline, letting Kafka manage consumer liveness. Static consumers will have a more relaxed `session.timeout.ms` deadline, but will also typically utilise another orchestrator with a more aggressive health check interval.

Partition assignment strategy

All considerations around group membership ultimately serve one purpose — to ensure that partitions are assigned among the members of a consumer group. As previously discussed, partition assignment takes place on a single consumer — the leader of the group.

The principal motivation for performing assignment on a consumer process is that it enables the application developers to substitute one of the built-in assignment strategies with a custom one,

without having to include the implementation of the custom strategy in the classpath of each of the broker nodes. This is particularly crucial when a cluster is being utilised in a multitenancy arrangement, where different teams or even paying customers might share the same set of brokers. It would be intractable to redeploy a broker upon a change to the rebalancing strategy, due to the disruption this would cause to all clients. Under the consumer-led model, one can easily roll out an update to the assignment strategy by redeploying the consumers, with no change or interruption to the Kafka cluster.

One of the challenges of the consumer-led model is the possible disagreement as to the preferred assignor implementation among the population of consumers. The leader will attempt to determine the most-agreed-upon strategy from the list of preferred strategies submitted by each member. If no common strategy can be agreed upon, the leader will return an error to the coordinator, which will be propagated to the remaining group members.

A consumer specifies its preferred assignors by supplying a list of their fully-qualified class names via the `partition.assignment.strategy` property. The assignor implementation must conform to the `org.apache.kafka.clients.consumer.ConsumerPartitionAssignor` interface. The assignors appear in the order of priority; the first assignor is given the highest priority, followed by the next assignor, and so on.

Although assignors are specified as class names, they are communicated using their symbolic names in the *state synchronisation* phase of the protocol. This allows for interoperability between different client implementations — for example, Java clients intermixed with Python clients. As long as the assignor names match, the consumers can agree on a common one.

Built-in assignors

There are four built-in assignors at one's disposal. At an overview level, these are:

1. `RangeAssignor`: the default assignor implied in the client configuration. The range assignor works by laying out the partitions for a single subscribed topic in numeric order and the consumers in lexicographic order. The number of partitions is then dividend by the number of consumers to determine the range of partitions that each consumer will receive. Since the two numbers might not divide evenly, the first few consumers may receive an extra partition. This process is repeated for each subscribed topic.
2. `RoundRobinAssignor`: allocates partitions in a round-robin fashion. The round-robin assignor works by laying out all partitions across all subscribed topics in one list, then assigning one partition to the first consumer, the second partition to the next consumer, and so on — until all partitions have been assigned. When the number of consumers is exhausted, the algorithm rotates back to the first consumer.
3. `StickyAssignor`: maintains an evenly-balanced distribution of partitions while preserving the prior assignment as much as possible, thereby minimising the difference in the allocations between successive assignments.
4. `CooperativeStickyAssignor`: a variant of the sticky assignor that utilises the newer cooperative rebalancing protocol to reduce the stop-the-world pause.

Range assignor

The range assignor, along with the round-robin assignor, is among the two of the oldest assignors in Kafka's arsenal. The range assignor is the default, yielding contiguous partition numbers for consumers. (This is easier on the eye, not that it matters much.) To understand how it works, consider the following scenario, where a topic with eight partitions is grown from one consumer to two, then to three, and finally, to four consumers. Initially, consumer assignments would resemble the following:

Consumer	Partitions
C0	P0, P1, P2, P3, P4, P5, P6, P7

Following the addition of a second consumer, the partition assignments would be altered to:

Consumer	Partitions
C0	P0, P1, P2, P3
C1	P4, P5, P6, P7

Comparing the two states, we arrive at the following delta:

Consumer	Assigned	Revoked
C0		P4, P5, P6, P7
C1	P4, P5, P6, P7	

The change encompasses four assignments and four revocations across both consumers — four swaps, so to speak. This is on par with the best assignment schemes, representing the optimal outcome, in terms of minimising reassignments and maintaining an evenly-balanced consumer group. By 'evenly-balanced', it is meant that the difference between the partition count of the largest allocation and that of the smallest allocation cannot exceed one.

Now consider the addition of a third consumer.

Consumer	Partitions
C0	P0, P1, P2
C1	P3, P4, P5
C2	P6, P7

Consumer	Assigned	Revoked
C0		P3
C1	P3	P6, P7
C2	P6, P7	

There are three swaps in this scenario. How would this compare to an optimal assignment? To maintain even balance with minimal reshuffling, we would simply revoke *P3* from *C0* and *P7* from *C1*, transferring them to *C2*:

Consumer	Partitions
C0	P0, P1, P2
C1	P4, P5, P6
C2	P3, P7

Consumer	Assigned	Revoked
C0		P3
C1		P7
C2	P3, P7	

So in this case, the range assignor loses to the optimum by three reassignments to two. While not ideal, this is marginally better than another popular built-in strategy — the round-robin assignor.

Let's increase the consumer count to four.

Consumer	Partitions
C0	P0, P1
C1	P2, P3
C2	P4, P5
C3	P6, P7

Consumer	Assigned	Revoked
C0		P2
C1	P2	P4, P5
C2	P4, P5	P6, P7
C3	P6, P7	

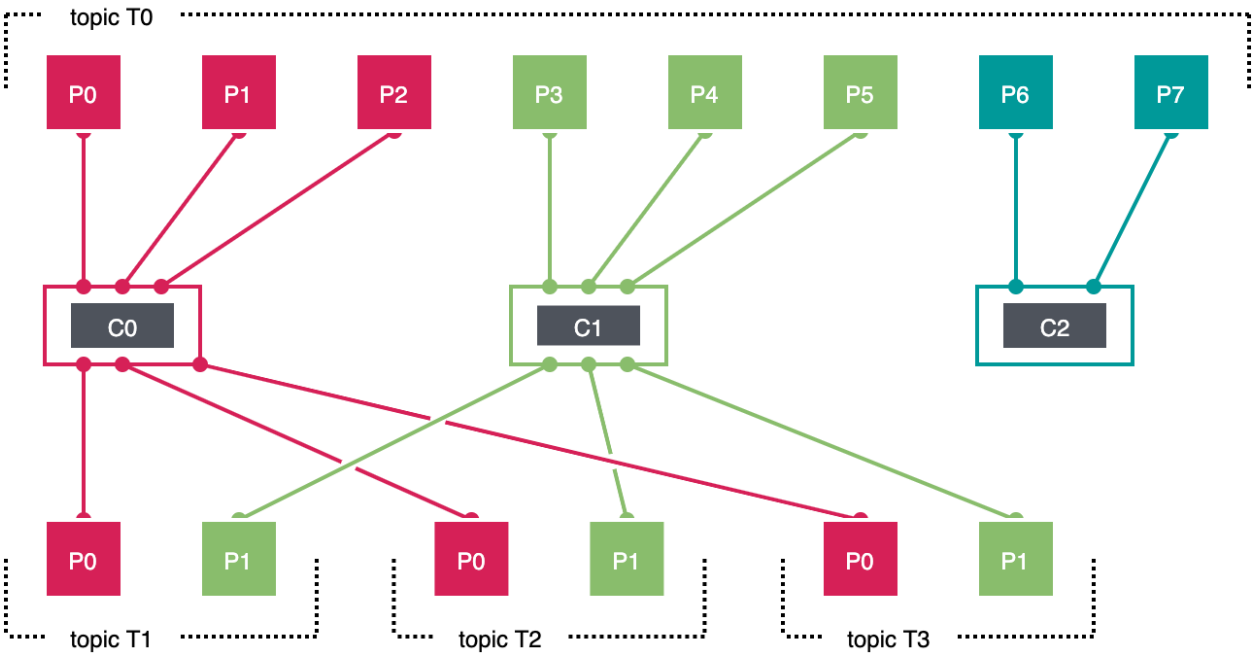
Comparing this with the optimum:

Consumer	Partitions
C0	P0, P1
C1	P3, P4
C2	P6, P7
C3	P2, P5

Consumer	Assigned	Revoked
C0		P2
C1		P6
C2		
C3	P2, P6	

The efficiency of a range assignor drops significantly compared to the optimum — five swaps to two. Beyond three consumers, the preservation of assignments is fairly poor.

Despite being the default option, the range assignor suffers from one severe deficiency: it leads to an uneven assignment of partitions to consumers having multiple topic subscriptions, where the partition count is, on average, smaller than the number of consumers. This phenomenon is illustrated below.



Range partition assignor

The range assignor considers the partitions for each topic separately, allocating them evenly among the consumers. In the above example, the first two consumers get an extra partition each for topic *T0*, which is reasonable, since 8 does not divide evenly into 3. So far, the range assignor seems fair.

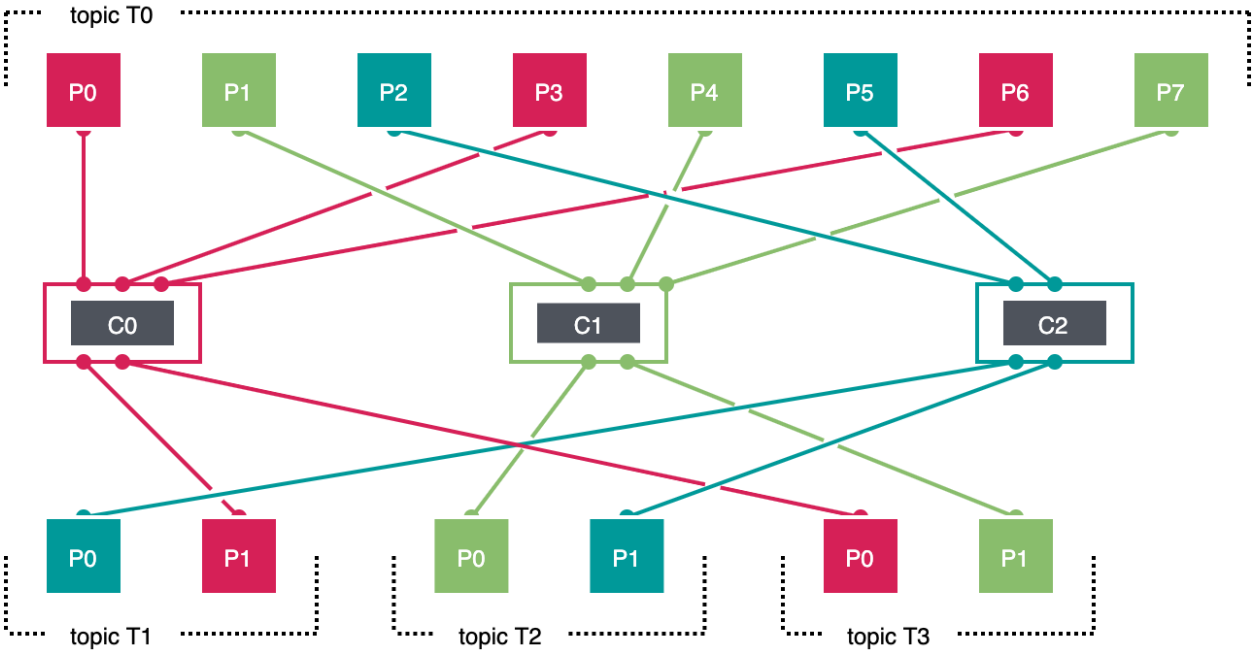
Moving on to *T1*, the two available partitions are assigned to consumers *C0* and *C1*, respectively. With insufficient partitions in *T1*, *C2* is left without an assignment. Again, this does not appear unreasonable when taken in isolation. However, when considering that the assignment for *T0* also favoured the first two consumers, it seems that *C2* is at a slight disadvantage, being towards the end of the pack. The inherent bias becomes more pronounced as we consider *T2* and *T3*; there are no new assignments for *C2*.

Kafka's performance doctrine of scaling consumers to increase parallelism does not always apply to its default assignor — under certain circumstances, as evident above, it will not scale no matter how many consumers are added into the mix — the new consumers will remain idle, despite having a sufficient aggregate number of partitions across all subscribed topics. This can be helped by ensuring that each topic is sufficiently 'wide' on its own. However, the range assignor is still inherently biased when the number of partitions does not divide cleanly into the number of consumers — giving preference to the consumers appearing higher in the lexicographical order of their member ID.

Round-robin assignor

Unlike the range assignor, the round-robin strategy aggregates all available partitions from every subscribed topic into a single vector, each element being a topic-partition tuple, then maps those topic-partitions onto the available consumers. In doing so, the round-robin assignor creates an even loading of consumers, providing that the subscription is uniform — in other words, each consumer is

subscribed to the same set of topics as every other consumer. This is usually the case in practice; it is rare to see a heterogeneous set of consumers with different subscription interests in the same group. The diagram below illustrates the effectiveness of a round-robin strategy in achieving a balanced loading.



Round-robin partition assignor

The preservation properties of a round-robin assignor are typically only slightly worse than that of the range assignor. Consider the previous test of subjecting a topic with eight partitions to a growing consumer population. Starting with the one-to-two (consumers) transition:

Consumer	Partitions
C0	P0, P2, P4, P6
C1	P1, P3, P5, P7

Consumer	Assigned	Revoked
C0		P1, P3, P5, P7
C1	P1, P3, P5, P7	

The performance is on par with the optimum. In fact, it can be trivially shown that every strategy will perform identically to the optimum for the one-to-two transition, in either direction.

Increasing the consumer population to three:

Consumer	Partitions
C0	P0, P3, P6
C1	P1, P4, P7
C2	P2, P5

Consumer	Assigned	Revoked
C0	P3	P2, P4
C1	P4	P3, P5
C2	P2, P5	

There are four swaps now, whereas two would have been optimum.

Now for the three-to-four transition:

Consumer	Partitions
C0	P0, P4
C1	P1, P5
C2	P2, P6
C3	P3, P7

Consumer	Assigned	Revoked
C0	P4	P3, P6
C1	P5	P4, P7
C2	P6	P5
C3	P3, P7	

The performance degrades to the same level as the range assignor, in both cases taking five swaps compared to the two swaps of the optimum strategy.

All in all, the round-robin assignor will show marginally worse preservation qualities than the range assignor in some cases. But in all cases, the round-robin assignor provides an ideal distribution of load in a homogeneous subscription. Frankly, one struggles to comprehend why the range assignor was chosen as the default strategy over round-robin, given that both are of the same vintage and the latter is more suitable for a broader range of scenarios.

Furthermore, the cost of rebalancing partitions is mostly attributable to the consumer ecosystem; consumers will contribute to the stop-the-world pause by blocking in their `onPartitionsRevoked()` callback. (Even without the optional callback, there will be some amount of cleanup within the `KafkaConsumer`, such as committing offsets if offset auto-commit is enabled.) Since both assignors utilise the eager rebalancing protocol, both sets of consumers will equally assume the worst-case scenario and revoke all partitions before synchronising state.

Sticky assignor

The observable degradation in preservation abilities of range and round-robin assignors as the number of consumers increases is attributable to their statelessness. The range and round-robin assignors arrive at the new allocations by only considering the requested subscriptions and the available topic-partitions. The sticky assignment strategy improves upon this by adding a third factor — the previous assignments — to arrive at an optimal number of swaps while maintaining an ideally balanced group. In other words, the sticky assignor performs as well as the theoretically-optimum strategy and retains the balancing qualities of round-robin.

The main issue with using a sticky assignor is that it is still based on the aging eager rebalance protocol. Without special consideration of the assignment strategy in the rebalance listener callback, a consumer application will still be forced down a worst-case assumption concerning partition revocations.

The documentation of the `StickyAssignor` suggests the following approach to mitigate the effects of the eager protocol. Ordinarily, a `ConsumerRebalanceListener` will commit offsets and perform any additional cleanup in the `onPartitionsRevoked()` callback, then initialise the new state as so:

```
new ConsumerRebalanceListener() {
  void onPartitionsRevoked(Collection<TopicPartition> partitions) {
    for (var partition : partitions) {
      commitOffsets(partition);
      cleanupState(partition);
    }
  }

  void onPartitionsAssigned(Collection<TopicPartition> partitions) {
    for (var partition : partitions) {
      initState(partition);
      initOffset(partition);
    }
  }
}
```

Under a sticky assignment model, one may want to defer the `cleanupState()` operations until after the partitions have been reassigned, providing that it is safe to do so. The amended listener code:

```
new ConsumerRebalanceListener() {
  Collection<TopicPartition> lastAssignment = List.of();

  void onPartitionsRevoked(Collection<TopicPartition> partitions) {
    for (var partition : partitions)
      commitOffsets(partition);
  }

  void onPartitionsAssigned(Collection<TopicPartition> partitions) {
    for (var partition : difference(lastAssignment, assignment))
      cleanupState(partition);

    for (var partition : difference(assignment, lastAssignment))
      initState(partition);
```

```
    for (var partition : assignment)
      initOffset(partition);

    this.lastAssignment = assignment;
  }
}
```

As a matter of fact, this approach could be used with any assignor; however, the sticky assignor ostensibly makes this more worthwhile. Whether this holds in practice, and to what extent, largely depends on the cost of the `cleanupState()` and `initState()` methods, especially if the former needs to block.

If `cleanupState()` is a blocking operation, then at least it will not hold back the `onPartitionsRevoked()` method when deferred, meaning that it will not contribute to the stop-the-world pause. By the same token, any consumer can alter its `ConsumerRebalanceListener` callbacks to take advantage of a deferred cleanup, irrespective of the chosen assignor. In other words, partition stickiness on its own does not yield an advantage in `onPartitionsRevoked()` — the gains result from refactoring and are independent of the chosen strategy. The sticky assignor helps in reducing the number of `cleanupState()` and `initState()` calls that are outside of the stop-the-world phase. The latter is significant, as it delays the start of record processing on the consumer, but does not affect the other consumers. The `cleanupState()` method, on the other hand, should ideally be executed asynchronously if possible — it is hard to imagine why state cleanup should hold up the `onPartitionsAssigned()` callback.

Cooperative sticky assignor

The cooperative sticky assignor is a relatively recent addition to the family, appearing in Kafka 2.4.0. It produces the same result as its eager `StickyAssignor` predecessor, utilising the newer incremental cooperative rebalance protocol.

The main difference between the two variants is that in the cooperative case, the consumer is told exactly which partitions will be revoked, reducing the blocking time of the `onPartitionsRevoked()` callback and alleviating the stop-the-world pause.

Upgrading assignors and rebalance protocols

As the assignor configuration is defined client-side, one may be tempted to change the assignor by simply updating the `partition.assignment.strategy` property to a new assignor and then bouncing the consumers. In reality, doing so in the presence of multiple consumers will halt the consumer group. When the first client bounces and rejoins the group, it will mandate an assignor that isn't supported by the remaining members, leading to an `INCONSISTENT_GROUP_PROTOCOL` error emitted by the group leader, manifesting as an `InconsistentGroupProtocolException` on the client.

The correct mechanism for changing an assignor is to perform two bounces. Before the first round, change the consumer configuration to reflect both the preferred and the outgoing assignor.

For example, if changing from the default range assignor to the cooperative sticky assignor, the `partition.assignment.strategy` property should be set to `org.apache.kafka.clients.consumer.RangeAssignor,c`. Having completed the first round, update the configuration to exclude the outgoing assignor. The assignors may initially appear in either order; the priority difference will be resolved by the start of the second round. After two rounds, the client configuration should only contain the cooperative assignor. This process ensures that at every point there is at least one common protocol that every consumer is willing to accept.

When migrating from the default consumer, it is essential that the `org.apache.kafka.clients.consumer.RangeAssig` is included explicitly in the strategy list, as the client will not add it by default.

This chapter has taken the reader on a deep-dive tour of Kafka's elaborate group management protocol — the underlying mechanism for dealing with consumers 'coming and going' gracefully. And not too gracefully, as it has turned out — consumers may spuriously disappear due to intermittent failures or reappear unexpectedly as lingering 'zombie' processes.

The ultimate intent of group membership is to allocate partitions to consumers and ensure exclusivity. As we have learned, the mechanics of partition assignment are fulfilled by an embedded protocol — an encapsulated communication between an elected group leader and the subordinate members, arbitrated by a dedicated coordinator node. The actual assignment is courtesy of a `ConsumerPartitionAssignor` implementation that executes entirely on the lead consumer; several built-in assignment strategies are available, catering to the needs of most consumers.

More recently, Kafka has improved support for container orchestration technologies like Kubernetes, improving rebalance performance and opening doors to external mechanisms for managing consumer health — means for promptly identifying and swiftly dealing with failures.

If there is anything that this chapter has imparted, it is that Kafka is no trivial creation. It is a complex beast, designed to fulfill a broad range of stream processing needs. For all its flexibility and configuration options, employing Kafka correctly is not straightforward by any stretch of the imagination. One requires a firm grasp on the theory of distributed and concurrent computing — at minimum, a thorough appreciation of the *liveness* and *safety* properties. *Anything that can go wrong, will go wrong* — as it was once famously said. Hopefully, the material in this chapter has prepared the reader for all sorts of situations.