PuigFlows

Guias tecnicas de ingenieria de datos y sistemas distribuidos

puigflows.com

Rebalanceo de Consumer Groups en Kafka: Cooperative Sticky, Static Membership y el Protocolo KIP-848

Kafka Consumer Group Rebalancing: Cooperative Sticky, Static Membership and the KIP-848 Protocol

Guía práctica del mecanismo que decide si tu consumidor de Kafka es estable o genera lag en cada despliegue: qué ocurre paso a paso durante un rebalanceo eager y por qué detiene al grupo entero, los tres relojes que lo disparan (heartbeat.interval.ms, session.timeout.ms y max.poll.interval.ms) y cómo distinguir un proceso muerto de uno vivo pero atascado, rebalanceo incremental cooperativo con la migración obligatoria en dos despliegues, pertenencia estática con group.instance.id y su contrapartida real, y el protocolo nuevo de KIP-848 que traslada el cálculo de la asignación al broker y mueve session.timeout.ms y heartbeat.interval.ms al servidor. Incluye un consumidor completo en Python con callbacks on_assign, on_revoke y on_lost bien diferenciados, el patrón pause/resume para trabajo lento sin destrozar la detección de fallos, ocho errores recurrentes, las métricas y comandos con los que diagnosticar un grupo atrapado en PreparingRebalance, y checklist de producción. Con código listo para producción en Python y configuración de cliente y broker.

Autor: Josue Garcia

Tiempo de lectura: ~53 min por idioma

Edicion: 2026-09-01 · Documento bilingue (Espanol / English)

Espanol

Rebalanceo de Consumer Groups en Kafka: Cooperative Sticky, Static Membership y el Protocolo KIP-848

Un rebalanceo mal entendido es la causa número uno de lag inexplicable

Casi todos los equipos que operan Kafka acaban teniendo la misma conversación. El consumidor lleva semanas funcionando, nadie ha tocado el código, y de pronto el lag sube en escalera durante los despliegues. O peor: se queda alto de forma permanente sin que ningún log muestre una excepción. Se revisan los brokers, la red, el tamaño de los mensajes. Casi siempre el culpable es el mismo: el grupo de consumidores está rebalanceando mucho más de lo que debería, y cada rebalanceo con el protocolo clásico detiene el consumo entero.

El rebalanceo es el mecanismo por el que Kafka reparte las particiones de un topic entre los miembros vivos de un grupo. Es lo que te permite escalar de tres a diez pods sin coordinar nada a mano. También es lo que convierte un despliegue rutinario en una pausa de treinta segundos si no lo has configurado. Esta guía cubre cómo funciona por dentro, qué relojes lo disparan, qué cambia con el rebalanceo cooperativo y la pertenencia estática, y qué hace realmente el protocolo nuevo de KIP-848 que alcanzó disponibilidad general en Kafka 4.0.

Qué ocurre exactamente durante un rebalanceo

Cada grupo tiene asignado un broker que actúa como group coordinator. El coordinador mantiene la lista de miembros, la generación actual del grupo (un contador que se incrementa en cada rebalanceo) y los offsets confirmados. En el protocolo clásico uno de los consumidores es además el líder del grupo, y es quien calcula el reparto de particiones en su propio proceso.

La secuencia clásica, conocida como rebalanceo eager, tiene cuatro pasos:

1. Algo cambia: un miembro entra, sale, deja de latir, o cambia el número de particiones del topic.
2. El coordinador marca el grupo como en rebalanceo y empieza a responder a los heartbeats con REBALANCE_IN_PROGRESS.
3. Todos los miembros revocan todas sus particiones y vuelven a unirse. Aquí es donde se para el mundo: nadie consume nada hasta que el último miembro haya respondido.
4. El líder calcula la asignación nueva, el coordinador la distribuye, y cada miembro reanuda el consumo en las particiones que le hayan tocado.

El detalle que duele es el paso tres. La duración del rebalanceo la marca el miembro más lento, y durante ese tiempo el grupo entero está parado — incluidos los nueve consumidores cuyas particiones no iban a moverse. Con un despliegue rolling de diez réplicas eso no es un rebalanceo: son diez rebalanceos consecutivos, cada uno con su pausa.

Los tres relojes que disparan un rebalanceo

Kafka expulsa a un consumidor por dos motivos completamente distintos, y confundirlos lleva a ajustar el parámetro equivocado. Desde la versión 0.10.1 el cliente tiene dos hilos: uno de fondo que envía heartbeats y el hilo de tu aplicación, que es el que llama a poll().

- heartbeat.interval.ms (por defecto 3000). Cada cuánto el hilo de fondo le dice al coordinador que sigue vivo. Es barato; no lo subas.
- session.timeout.ms (por defecto 45000 desde Kafka 3.0; antes eran 10000). Cuánto tiempo puede pasar el coordinador sin recibir un heartbeat antes de dar al miembro por muerto. Detecta procesos caídos, pods eliminados y particiones de red. La regla práctica es mantenerlo en al menos tres veces el intervalo de heartbeat.
- max.poll.interval.ms (por defecto 300000, cinco minutos). Cuánto tiempo puede tardar tu código entre dos llamadas a poll(). Detecta consumidores vivos pero atascados: el proceso late perfectamente mientras tu bucle lleva ocho minutos esperando a una API externa.

La distinción importa más de lo que parece. Si tus rebalanceos coinciden con picos de latencia de un servicio del que dependes, el problema es max.poll.interval.ms y subir el session timeout no arregla absolutamente nada. Si coinciden con presión de memoria, pausas largas de GC o nodos spot que desaparecen, el problema es la sesión. Antes de tocar un número, averigua cuál de los dos relojes se agotó.

Rebalanceo cooperativo: revocar sólo lo que se mueve

El rebalanceo incremental cooperativo (KIP-429) elimina la barrera global. En lugar de revocar todo y volver a repartir, el líder calcula la asignación objetivo y sólo se revocan las particiones que realmente cambian de dueño. Un consumidor que conserva sus ocho particiones no deja de consumir ni un milisegundo.

El precio es que hacen falta dos rondas: en la primera se revocan las particiones que se mueven, en la segunda se asignan a su nuevo dueño. Suena peor y es muchísimo mejor, porque el trabajo útil nunca se detiene.

```
from confluent_kafka import Consumer

conf = {
    "bootstrap.servers": "kafka-1:9092,kafka-2:9092",
    "group.id": "orders-processor",

    # Rebalanceo incremental: solo se revoca lo que cambia de dueno.
    "partition.assignment.strategy": "cooperative-sticky",

    # Deteccion de procesos muertos.
    "session.timeout.ms": 45000,
    "heartbeat.interval.ms": 3000,

    # Deteccion de procesos vivos pero atascados.
    "max.poll.interval.ms": 300000,

    # Nunca confirmes offsets que no has procesado.
    "enable.auto.commit": False,
    "auto.offset.reset": "earliest",
}
```

Si vienes del cliente Java hay una trampa importante en la migración: no puedes mezclar asignadores eager

y cooperativos en el mismo grupo. Los miembros no llegarían a un acuerdo y el grupo quedaría atascado. El cambio se hace en dos despliegues:

```
# Despliegue 1: toda la flota anuncia AMBAS estrategias.
# El grupo sigue usando Range porque es la primera que todos comparten.
partition.assignment.strategy=org.apache.kafka.clients.consumer.RangeAssignor,org.apache.kafka.clien
ts.consumer.CooperativeStickyAssignor

# Despliegue 2 (solo cuando el 100% de la flota tenga lo anterior):
# se elimina la estrategia eager y el grupo migra a cooperativo.
partition.assignment.strategy=org.apache.kafka.clients.consumer.CooperativeStickyAssignor
```

Saltarte el primer despliegue es uno de los errores que provoca incidentes largos, porque el síntoma — un grupo que rebalancea en bucle sin estabilizarse nunca — no apunta de forma obvia a la causa.

Pertenencia estática: reinicios que no cuentan como salidas

La pertenencia estática (KIP-345) resuelve un problema distinto. Cuando un consumidor se reinicia, el cliente le asigna un identificador de miembro nuevo, así que el coordinador ve una salida seguida de una entrada: dos rebalanceos por cada reinicio. Multiplica eso por el número de réplicas y ya sabes por qué tus despliegues generan una meseta de lag.

Al fijar group.instance.id, el miembro conserva su identidad entre reinicios. El cliente estático no envía LeaveGroup al cerrarse, y el coordinador le guarda sus particiones hasta que expire session.timeout.ms. Si el proceso vuelve antes de ese plazo, recupera exactamente las mismas particiones y no se produce ningún rebalanceo.

```
import os

conf["group.instance.id"] = os.environ["POD_NAME"]  # p. ej. orders-processor-2
```

El identificador tiene que ser estable y único. En Kubernetes eso significa un StatefulSet, cuyos pods se llaman orders-processor-0, orders-processor-1, etc. Usar el nombre de pod de un Deployment no sirve de nada: cambia en cada despliegue y vuelves al punto de partida. Y si dos procesos arrancan con el mismo valor, el segundo expulsa al primero con un FencedInstanceIdException.

La contrapartida es real y hay que aceptarla conscientemente: si el pod muere de verdad, sus particiones quedan sin consumir durante los 45 segundos completos del session timeout. Estás cambiando latencia de recuperación ante fallos por estabilidad durante los despliegues, que es un buen trato cuando despliegas varias veces al día y los pods se caen una vez al mes. Si tu SLA no tolera ese hueco, baja el session timeout o no uses pertenencia estática.

KIP-848: el rebalanceo deja de ser cosa del cliente

El protocolo nuevo de grupos de consumidores, disponible con carácter general desde Apache Kafka 4.0, es el cambio más profundo en esta área en una década. La idea central: se elimina la figura del líder del grupo y la sincronización global. El cálculo de la asignación se traslada al group coordinator, en el broker.

En lugar de una barrera JoinGroup + SyncGroup en la que todos esperan a todos, cada miembro mantiene un diálogo independiente con el coordinador a través de un único RPC, ConsumerGroupHeartbeat. El

coordinador publica la asignación objetivo con un número de época; cada consumidor reconcilia su estado hacia esa asignación a su ritmo, revocando primero y confirmando después. No hay ningún instante en que el grupo entero esté parado.

```
# Cliente Kafka 4.x
group.protocol=consumer
group.id=orders-processor
group.instance.id=orders-processor-0
max.poll.interval.ms=300000
enable.auto.commit=false

# Con el protocolo nuevo, estos ajustes del CLIENTE se ignoran:
#   session.timeout.ms
#   heartbeat.interval.ms
#   partition.assignment.strategy
# Ahora los decide el broker (ver abajo).
```

```
# Broker
group.coordinator.rebalance.protocols=classic,consumer
group.consumer.session.timeout.ms=45000
group.consumer.heartbeat.interval.ms=5000
```

Que estos parámetros se muevan al servidor sorprende a mucha gente durante la migración. Es deliberado: así el operador del clúster garantiza tiempos de detección coherentes en lugar de depender de que cada equipo configure bien su cliente. Si necesitas ajustar la detección de fallos, ahora es una conversación con quien opera Kafka, no un cambio en tu values.yaml.

Dos cosas más que conviene saber. La primera: los grupos pueden ser mixtos durante la migración — miembros con group.protocol=classic y consumer conviven en el mismo grupo, así que puedes migrar con un rolling normal. La segunda: el protocolo clásico sigue siendo el valor por defecto en la rama 4.x; el nuevo se activa explícitamente y está previsto que pase a ser el predeterminado en una versión mayor posterior. Si ya estás en brokers 4.x, actívalo primero en un grupo no crítico y compara la métrica de latencia de rebalanceo antes y después.

Un consumidor correcto de principio a fin

La configuración es la mitad del trabajo; la otra mitad son los callbacks de rebalanceo. Este es el patrón completo con confluent-kafka-python, que es el cliente que más se usa fuera del ecosistema JVM.

```
import logging
import os
import signal

from confluent_kafka import Consumer

log = logging.getLogger("orders")
running = True


def _stop(*_):
    global running
    running = False


signal.signal(signal.SIGTERM, _stop)
signal.signal(signal.SIGINT, _stop)
```

```
def on_assign(consumer, partitions):
    # Con el protocolo cooperativo la asignacion es INCREMENTAL.
    # Llamar a assign() aqui borraria las particiones que ya teniamos.
    log.info("asignadas: %s", [(p.topic, p.partition) for p in partitions])
    consumer.incremental_assign(partitions)


def on_revoke(consumer, partitions):
    # Revocacion limpia: todavia somos los duenos de estas particiones.
    # Este es el unico momento seguro para confirmar offsets.
    log.info("revocadas: %s", [(p.topic, p.partition) for p in partitions])
    try:
        consumer.commit(asynchronous=False)
    except Exception:
        log.exception("fallo el commit durante la revocacion")
    consumer.incremental_unassign(partitions)


def on_lost(consumer, partitions):
    # Perdimos las particiones sin aviso: expiro la sesion o nos expulsaron.
    # Otro consumidor puede tenerlas YA. Confirmar aqui corrompe los offsets.
    log.warning("particiones perdidas: %s", [(p.topic, p.partition) for p in partitions])
    consumer.incremental_unassign(partitions)


consumer = Consumer(conf)
consumer.subscribe(
    ["orders"],
    on_assign=on_assign,
    on_revoke=on_revoke,
    on_lost=on_lost,
)

try:
    while running:
        # librdkafka NO tiene max.poll.records: el tamano del lote se controla
        # aqui, con num_messages, y con fetch.max.bytes.
        msgs = consumer.consume(num_messages=100, timeout=1.0)
        if not msgs:
            continue

        for msg in msgs:
            if msg.error():
                log.error("error de fetch: %s", msg.error())
                continue
            handle(msg)  # debe ser idempotente

        consumer.commit(asynchronous=False)
finally:
    # Revoca, confirma y cierra de forma ordenada.
    # Ojo: un miembro estatico NO sale del grupo aqui. Eso es lo que queremos.
    consumer.close()
```

Tres decisiones concretas en ese código merecen explicación. incremental_assign e incremental_unassign en lugar de assign: con el protocolo cooperativo los callbacks reciben sólo el delta, no el conjunto completo, y usar assign() descartaría silenciosamente las particiones que conservabas. El commit dentro de on_revoke y no dentro de on_lost: en el primer caso aún eres el dueño legítimo, en el segundo puede que otro consumidor lleve segundos procesando esas particiones y tu commit retrasaría su offset. Y num_messages en lugar de max.poll.records: ese parámetro del cliente Java sencillamente no existe en librdkafka, y buscarlo en la documentación equivocada hace perder tardes enteras.

Cuando el trabajo es lento: pausa, no alargues el timeout

El reflejo habitual ante un max.poll.interval.ms agotado es subirlo a treinta minutos. Funciona, y a la vez destruye la detección de fallos: un pod colgado retendrá sus particiones media hora antes de que nadie se entere. La solución correcta es desacoplar el latido del procesamiento pausando las particiones y seguir llamando a poll().

```
msgs = consumer.consume(num_messages=20, timeout=1.0)

if msgs:
    # Pausar evita que se acumulen mensajes nuevos mientras trabajamos,
    # pero seguimos llamando a poll(), que es lo que reinicia el reloj
    # de max.poll.interval.ms y mantiene viva la sesion.
    assignment = consumer.assignment()
    consumer.pause(assignment)
    try:
        for msg in msgs:
            process_slow(msg)   # puede tardar minutos
            consumer.poll(0)    # no devuelve datos: las particiones estan pausadas
    finally:
        consumer.resume(assignment)

    consumer.commit(asynchronous=False)
```

Si el trabajo es realmente largo — transcodificar un vídeo, generar un informe de horas — el patrón adecuado ya no es este. Consume el mensaje, escribe una tarea en una cola de trabajos duradera, confirma el offset y deja que un worker independiente haga el trabajo pesado. Kafka es excelente transportando eventos y bastante mediocre como planificador de trabajos de larga duración.

Ocho errores que se repiten en producción

1. Confundir los dos relojes. Subir session.timeout.ms cuando el problema es el tiempo de procesamiento. No cambia nada y empeora la detección de fallos.

2. Dejar enable.auto.commit=true con procesamiento asíncrono. El auto-commit confirma lo que se ha entregado, no lo que se ha procesado. Un crash entre ambos momentos es pérdida de datos silenciosa, del tipo que nadie detecta hasta que un cliente reclama.

3. Confirmar offsets en on_lost. Ya no eres el dueño de esas particiones. Retrasas el offset de otro consumidor y provocas reprocesamiento masivo.

4. Migrar a cooperativo en un solo despliegue. El grupo entra en un bucle de rebalanceo del que no sale hasta que unificas la configuración. Siempre dos pasos.

5. group.instance.id inestable. Un UUID aleatorio o el nombre de pod de un Deployment tienen exactamente el mismo efecto que no configurar nada.

6. Más consumidores que particiones. Los sobrantes se quedan sin asignación, ociosos, consumiendo presupuesto. Escalar el grupo más allá del número de particiones no aumenta el rendimiento; hay que aumentar las particiones primero (y eso rompe el orden por clave si no lo planificas).

7. Suponer que el rebalanceo garantiza entrega exactamente una vez. No lo hace. Entre el último commit y la revocación siempre hay una ventana de reproceso. Tu handler tiene que ser idempotente, punto.

8. session.timeout.ms fuera del rango del broker. Si cae fuera de group.min.session.timeout.ms o

group.max.session.timeout.ms, el consumidor ni siquiera consigue unirse al grupo y el error es poco descriptivo.

Qué medir

Un grupo sano rebalancea cuando tú despliegas y casi nunca más. Estas son las señales que lo confirman, expuestas por el cliente vía JMX o por el exporter que uses:

- rebalance-rate-per-hour — la métrica principal. Si es mayor que tu frecuencia de despliegue, hay algo que investigar.
- rebalance-latency-avg y rebalance-latency-max — cuánto cuesta cada uno. Es la métrica que debería desplomarse al migrar a cooperativo o a KIP-848.
- last-rebalance-seconds-ago — excelente para alertar: si baja de forma repetida, estás en un bucle.
- time-between-poll-avg y time-between-poll-max — compáralo con tu max.poll.interval.ms. Si el máximo se acerca al límite, la expulsión es cuestión de tiempo.
- records-lag-max — el síntoma que ve el negocio, siempre por partición.

Para diagnosticar en caliente, la herramienta de línea de comandos sigue siendo la más directa:

```
# Estado del grupo y protocolo en uso (Stable, PreparingRebalance...)
kafka-consumer-groups.sh --bootstrap-server kafka:9092 \
  --describe --group orders-processor --state

# Miembros, host, client-id y particiones de cada uno.
# Si el mismo host aparece dos veces, tienes un group.instance.id duplicado.
kafka-consumer-groups.sh --bootstrap-server kafka:9092 \
  --describe --group orders-processor --members --verbose

# Lag por particion: donde se ve si el reparto esta desequilibrado.
kafka-consumer-groups.sh --bootstrap-server kafka:9092 \
  --describe --group orders-processor
```

Un grupo atrapado en PreparingRebalance durante minutos casi siempre significa una de dos cosas: un miembro que no responde y por el que se está esperando hasta agotar el timeout, o una mezcla incompatible de estrategias de asignación.

Checklist de producción

- enable.auto.commit=false y commit explícito después de procesar.
- cooperative-sticky como estrategia de asignación, migrada en dos despliegues.
- group.instance.id estable, con un StatefulSet si estás en Kubernetes.
- Callbacks on_revoke y on_lost distintos, con commit sólo en el primero.
- terminationGracePeriodSeconds suficiente para que consumer.close() termine.
- Handler idempotente, con clave de deduplicación derivada del evento y no del offset.
- Alerta sobre rebalance-rate-per-hour y sobre time-between-poll-max.
- Si operas brokers 4.x, un plan para probar group.protocol=consumer en un grupo no crítico.

El objetivo final es aburrido y se resume en una frase: que un despliegue no aparezca en tus gráficas de lag.

Cuando lo consigues, escalar el consumo deja de dar miedo, y esa es exactamente la propiedad por la que elegiste Kafka.

El protocolo por dentro: JoinGroup, SyncGroup y el generation id

Hasta ahora hemos hablado del rebalanceo como si fuera una caja negra que ocurre y ya. Merece la pena abrirla, porque casi todos los síntomas raros que verás en producción se explican mirando las cuatro peticiones que se intercambian el consumidor y el coordinador del grupo. No necesitas memorizar el formato binario, pero sí necesitas saber quién decide qué y en qué orden.

El coordinador del grupo es un broker concreto, no un servicio aparte. Se elige aplicando un hash sobre el group.id para obtener una partición del topic interno __consumer_offsets, y el líder de esa partición es el coordinador. De ahí salen dos consecuencias prácticas: si ese broker se reinicia, tu grupo se queda sin coordinador durante unos segundos y tendrá que redescubrirlo (verás FindCoordinator en los logs), y si tienes muchos grupos con nombres que colisionan en la misma partición, todos comparten destino. Es una de las razones por las que offsets.topic.num.partitions se deja alto (50 por defecto) y no se cambia a la ligera después de crear el clúster.

El baile clásico tiene cuatro pasos:

1. FindCoordinator — el consumidor pregunta a cualquier broker quién coordina su grupo. Se cachea, y se repite si el coordinador cae.
2. JoinGroup — el consumidor anuncia a qué topics se suscribe y qué estrategias de asignación soporta. Esta petición se bloquea en el broker hasta que todos los miembros conocidos han enviado la suya o hasta que expira rebalance.timeout.ms. Aquí es donde vive la mayor parte del "stop the world".
3. SyncGroup — el coordinador elige a un miembro como líder del grupo (normalmente el primero en llegar), le entrega la lista completa de miembros y sus suscripciones, y espera a que le devuelva el reparto. Los demás miembros mandan un SyncGroup vacío y se quedan esperando su trozo.
4. Heartbeat — a partir de aquí, latidos periódicos que mantienen viva la pertenencia y que son el canal por el que el coordinador avisa de que hay que rebalancear otra vez.

Lo que sorprende a mucha gente la primera vez: el reparto de particiones lo calcula un cliente, no el broker. El coordinador solo hace de árbitro y de buzón. Eso explica por qué una versión antigua de tu aplicación desplegada por error puede repartir particiones de forma distinta al resto de la flota, y por qué un cliente con un bug en su assignor puede dejar particiones huérfanas sin que el broker se entere de nada.

El pegamento que mantiene todo esto coherente es el generation id, un entero que se incrementa en cada rebalanceo completado. Cada commit de offsets lleva su generation id, y el coordinador rechaza cualquier commit que llegue con uno viejo devolviendo ILLEGAL_GENERATION. Es un mecanismo de fencing: si un consumidor se quedó colgado veinte segundos y vuelve creyendo que sigue siendo dueño de la partición 7, su commit se rechaza porque el mundo ya va por la generación siguiente. Sin esto, un consumidor zombi podría retroceder el offset de una partición que ya no le pertenece y provocar un reprocesado masivo.

Los errores que verás asociados a este ciclo, y lo que significan de verdad:

- REBALANCE_IN_PROGRESS — no es un error, es una señal. El coordinador te está diciendo "deja lo que estás haciendo y vuelve a hacer JoinGroup". Si aparece constantemente, tu grupo no consigue estabilizarse.
- ILLEGAL_GENERATION — llegaste tarde. Tu commit corresponde a un mundo que ya no existe. Casi siempre significa que el procesamiento tardó más que max.poll.interval.ms.

- UNKNOWN_MEMBER_ID — el coordinador ya te había expulsado y olvidado. Tienes que volver a entrar desde cero, sin identidad previa.
- COORDINATOR_NOT_AVAILABLE / NOT_COORDINATOR — el broker coordinador cambió. El cliente reintenta solo; si lo ves en ráfagas, mira la salud de los brokers, no la de tu consumidor.

Una regla mental que ahorra horas de depuración: cuando el problema es ILLEGAL_GENERATION mira tu bucle de procesamiento; cuando es NOT_COORDINATOR mira el clúster. Confundir los dos lleva a equipos enteros a tunear timeouts cuando lo que tenían era un broker reiniciándose cada noche.

Qué assignor elegir y por qué casi nunca es el que viene por defecto

La estrategia de asignación decide qué partición va a qué consumidor. Kafka trae varias en la caja y la diferencia entre ellas se nota mucho más de lo que la documentación sugiere.

RangeAssignor es el histórico y el que sale por defecto en el protocolo clásico. Ordena las particiones de cada topic y las reparte en bloques contiguos. Su gran virtud es que, si consumes varios topics con el mismo número de particiones y las mismas claves, la partición 3 de orders y la 3 de payments acaban en el mismo consumidor, lo que permite hacer joins locales sin red. Su gran defecto es que reparte fatal cuando el número de particiones no es múltiplo del número de consumidores: con 7 particiones y 3 consumidores obtienes 3, 2, 2, y con varios topics ese sesgo se acumula siempre sobre los mismos consumidores.

RoundRobinAssignor reparte particiones una a una entre todos los miembros. Equilibra mucho mejor, pero pierde la co-localización entre topics y, sobre todo, reasigna casi todo en cada rebalanceo: un miembro que entra o sale desplaza el reparto entero.

StickyAssignor añade la idea que importa: equilibra igual de bien que round-robin pero, entre todos los repartos igual de equilibrados, elige el que menos particiones mueve respecto al anterior. Sigue siendo eager (revoca todo antes de reasignar), así que la pausa persiste, pero al menos el estado local que reconstruyes después es menor.

CooperativeStickyAssignor es sticky más el protocolo incremental que ya vimos: no revoca lo que no se mueve. Si vas a elegir a ciegas, elige este. Es el único que combina reparto equilibrado, estabilidad entre rebalanceos y ausencia de pausa global.

Hay una quinta opción que la gente olvida: la asignación consciente del rack. Cuando el consumidor declara client.rack y el broker tiene réplicas etiquetadas por rack, el fetch puede servirse desde una réplica seguidora en la misma zona de disponibilidad en lugar de cruzar zonas hasta el líder. En nubes que cobran el tráfico entre AZ, esto no es una micro-optimización: es una línea de la factura. En las versiones 4.x volvió a estar disponible la asignación consciente del rack para grupos de consumidores, de modo que el reparto intenta activamente emparejar consumidores con particiones cuyas réplicas viven en su misma zona.

```
# Consumidor con lectura desde réplica local y reparto cooperativo
bootstrap.servers=kafka-1:9092,kafka-2:9092,kafka-3:9092
group.id=orders-processor
client.rack=eu-west-1b
partition.assignment.strategy=org.apache.kafka.clients.consumer.CooperativeStickyAssignor
enable.auto.commit=false
max.poll.records=200
max.poll.interval.ms=300000
```

Una advertencia que cuesta cara: client.rack solo sirve si los brokers están configurados con broker.rack y si tienes replica.selector.class apuntando al selector consciente del rack. Si configuras únicamente el lado

cliente, no pasa nada malo, simplemente no obtienes el ahorro y te quedas convencido de que sí.

Offsets, commits y el momento exacto en el que se duplican los mensajes

Un rebalanceo por sí solo no duplica ni pierde nada. Lo que duplica o pierde es la relación entre el momento en que confirmas el offset y el momento en que el trabajo queda realmente hecho. El rebalanceo se limita a exponer el problema que ya estaba ahí.

Con enable.auto.commit=true, el cliente confirma en segundo plano cada auto.commit.interval.ms (5 segundos por defecto) el offset de lo último que te ha entregado, no de lo último que has procesado. Si tu poll() devuelve 500 registros, procesas 120 y el proceso muere, el auto-commit puede haber confirmado ya los 500. Los 380 restantes no se vuelven a leer nunca. Esto no es un bug del auto-commit: es exactamente lo que promete, y por eso no sirve para nada donde perder un registro importe.

El commit manual invierte el riesgo. Si confirmas después de procesar, un fallo entre "procesado" y "confirmado" hace que el siguiente dueño de la partición vuelva a leer ese lote. Eso es at least once, y es el punto de partida sano para casi todo. La pregunta correcta no es "cómo evito los duplicados" sino "cómo hago que un duplicado no importe": claves primarias con UPSERT, escrituras idempotentes, deduplicación por identificador de evento en una ventana. Un consumidor cuyo efecto secundario es idempotente convierte el rebalanceo en un no-evento.

Aquí está la diferencia entre los dos commits que ofrece el cliente, y cuándo usar cada uno:

- commitAsync() en el camino caliente, dentro del bucle. No bloquea, encadena bien y si falla el siguiente commit lo arregla porque los offsets son monótonos.
- commitSync() en dos sitios y solo en dos: en el callback de revocación y en el cierre ordenado. Ahí sí quieres bloquear, porque después de esa línea pierdes el derecho a escribir.

Si usas transacciones (productor con transactional.id y consumidor con isolation.level=read_committed), el commit de offsets se hace dentro de la transacción con sendOffsetsToTransaction, no con commitSync. El offset y el resultado de procesarlo se hacen visibles a la vez o no se hacen visibles ninguno. Esto da exactly once dentro de Kafka, con dos matices que conviene decir en voz alta: no se extiende a sistemas externos salvo que ese sistema participe en la transacción o sea idempotente, y un rebalanceo aborta la transacción abierta, lo que significa que el trabajo del lote en curso se descarta y se repite. Es correcto, pero no es gratis.

```
from confluent_kafka import Consumer, KafkaException

consumer = Consumer({
    "bootstrap.servers": "kafka-1:9092",
    "group.id": "orders-processor",
    "enable.auto.commit": False,
    "partition.assignment.strategy": "cooperative-sticky",
    "auto.offset.reset": "latest",
    "max.poll.interval.ms": 300000,
})

def procesar(msg):
    # Idempotente: la misma clave dos veces deja el mismo estado
    db.execute(
        "INSERT INTO orders (id, payload) VALUES (%s, %s) "
        "ON CONFLICT (id) DO UPDATE SET payload = EXCLUDED.payload",
        (msg.key().decode(), msg.value()),
    )

consumer.subscribe(["orders"])
```

```
try:
    while True:
        msg = consumer.poll(1.0)
        if msg is None:
            continue
        if msg.error():
            raise KafkaException(msg.error())
        procesar(msg)
        # asíncrono en el camino caliente: barato y suficiente
        consumer.commit(message=msg, asynchronous=True)
finally:
    # síncrono al cerrar: la última palabra tiene que llegar
    consumer.commit(asynchronous=False)
    consumer.close()
```

Fíjate en el finally. close() hace algo que mucha gente ignora: envía un LeaveGroup explícito. Sin él, el grupo tiene que esperar a que expire session.timeout.ms para darse cuenta de que te fuiste. Un SIGKILL sin gracia convierte un despliegue de 3 segundos en uno de 45. Asegúrate de que tu terminationGracePeriodSeconds en Kubernetes es mayor que el tiempo que tarda tu bucle en salir, y de que capturas SIGTERM en lugar de ignorarlo.

El ConsumerRebalanceListener que sí funciona

El listener de rebalanceo es el único punto donde tu código se entera de que el suelo se mueve. Tiene tres callbacks y la mayoría de la gente implementa uno y medio.

onPartitionsRevoked se llama cuando todavía eres dueño de las particiones y te van a quitar. Es tu ventana para vaciar buffers, cerrar ficheros, hacer flush al sink y confirmar offsets. Todo lo que hagas aquí cuenta contra rebalance.timeout.ms, así que tiene que ser acotado: si tu flush puede tardar un minuto, tu timeout de rebalanceo tiene que contemplarlo o el grupo te expulsará mientras limpias.

onPartitionsAssigned se llama con las particiones nuevas ya tuyas. Es donde reconstruyes estado local, precargas cachés o haces seek() si gestionas los offsets fuera de Kafka. En el protocolo cooperativo recibe solo lo que se ha añadido, no el conjunto completo; escribir código que asuma lo contrario es el error número uno al migrar.

onPartitionsLost es el que casi nadie implementa y el que más importa. Se invoca cuando perdiste las particiones sin aviso: te expulsaron del grupo, otro ya las tiene. Aquí no debes confirmar offsets, porque el commit se rechazará o, peor, sobrescribirá el progreso de quien ya está trabajando. Lo correcto es descartar el trabajo en vuelo y limpiar. Si no lo implementas, el cliente cae en la implementación por defecto, que llama a onPartitionsRevoked, y acabas intentando confirmar offsets sobre particiones ajenas.

```
from confluent_kafka import Consumer

def on_assign(consumer, partitions):
    # Cooperativo: 'partitions' son SOLO las nuevas
    for tp in partitions:
        estado.cargar(tp.topic, tp.partition)
    print(f"asignadas +{len(partitions)}")

def on_revoke(consumer, partitions):
    # Todavía somos dueños: vaciar y confirmar
    sink.flush()
    try:
        consumer.commit(asynchronous=False)
    except Exception as e:
```

```
        print(f"commit en revocación falló: {e}")
    for tp in partitions:
        estado.descartar(tp.topic, tp.partition)

def on_lost(consumer, partitions):
    # Ya NO somos dueños: limpiar sin confirmar nada
    sink.abortar()
    for tp in partitions:
        estado.descartar(tp.topic, tp.partition)
    print(f"perdidas {len(partitions)} particiones sin aviso")

consumer = Consumer({
    "bootstrap.servers": "kafka-1:9092",
    "group.id": "orders-processor",
    "enable.auto.commit": False,
    "partition.assignment.strategy": "cooperative-sticky",
})
consumer.subscribe(
    ["orders"],
    on_assign=on_assign,
    on_revoke=on_revoke,
    on_lost=on_lost,
)
```

Un detalle sutil sobre el orden: en el protocolo cooperativo, la revocación y la asignación pueden ocurrir en rondas distintas. Primera ronda, revocas lo que se mueve. Segunda ronda, recibes lo que te toca. Entre las dos, tu consumidor sigue procesando lo que conservó. Si tu listener asume que a una revocación le sigue inmediatamente una asignación en el mismo ciclo, tendrás estado inconsistente durante unos segundos y no sabrás por qué.

KIP-848 en detalle: el bucle de reconciliación

El protocolo nuevo, disponible de forma general desde Apache Kafka 4.0, no es una optimización del anterior: es un modelo distinto. La diferencia de fondo es que desaparece la sincronización global. No hay una barrera en la que todos esperan a todos.

En el modelo nuevo, el coordinador del broker mantiene dos vistas por miembro: la asignación objetivo (lo que debería tener) y la asignación actual (lo que ha confirmado que tiene). Cuando algo cambia —entra un miembro, sale otro, cambia el número de particiones— el coordinador recalcula el objetivo para todo el grupo y lo publica. A partir de ahí, cada miembro converge a su propio ritmo mediante una única RPC, ConsumerGroupHeartbeat, que hace de latido y de canal de reconciliación a la vez. El miembro dice "tengo esto", el coordinador responde "deberías tener esto otro", el miembro revoca lo que sobra, confirma, y recibe lo que le falta.

Las consecuencias prácticas son grandes:

- El cálculo de la asignación vive en el broker. Ya no depende de qué versión de cliente resultó ser líder. Se elige con group.remote.assignor y es homogéneo para todo el grupo.
- Un miembro lento ya no bloquea al resto. En el protocolo clásico, el JoinGroup esperaba al más lento. Aquí, cada uno converge por su cuenta y los que no están afectados ni se enteran.
- El fencing pasa a ser por épocas. Cada miembro tiene su propia epoch, y el grupo tiene la suya. Un miembro con epoch vieja se rechaza sin necesidad de una generación global compartida.
- Los timeouts se mueven al servidor. session.timeout.ms y heartbeat.interval.ms dejan de ser configuración de cliente y pasan a ser política del clúster (group.consumer.session.timeout.ms y

group.consumer.heartbeat.interval.ms). Esto elimina de golpe la clase entera de incidentes en la que un despliegue nuevo trae timeouts distintos al resto de la flota.

Tres propiedades del consumidor desaparecen cuando activas el protocolo nuevo: partition.assignment.strategy, session.timeout.ms y heartbeat.interval.ms. Si las dejas en tu fichero de configuración, el cliente fallará al arrancar o las ignorará según la versión. Quitarlas es parte obligatoria de la migración, no una limpieza opcional.

```
# Cliente 4.x con el protocolo nuevo
bootstrap.servers=kafka-1:9092,kafka-2:9092,kafka-3:9092
group.id=orders-processor
group.protocol=consumer
group.remote.assignor=uniform
enable.auto.commit=false
max.poll.interval.ms=300000
# NO incluir: partition.assignment.strategy
# NO incluir: session.timeout.ms
# NO incluir: heartbeat.interval.ms
```

```
# Broker: ambos protocolos activos durante la migración
group.coordinator.rebalance.protocols=classic,consumer
group.consumer.session.timeout.ms=45000
group.consumer.heartbeat.interval.ms=5000
group.consumer.max.session.timeout.ms=60000
```

Lo que no cambia es igual de importante: max.poll.interval.ms sigue siendo tuyo y sigue siendo el reloj que mata consumidores lentos. El protocolo nuevo separa mejor "estoy vivo" de "estoy avanzando", pero no te perdona un bucle de procesamiento que tarda más de lo que dijiste que tardaría.

Migrar sin sustos: el procedimiento completo

La migración al protocolo nuevo está diseñada para ser online, pero tiene un orden y unos requisitos que no se pueden saltar. Este es el guion que funciona.

Antes de tocar nada. Confirma que todos los brokers están en 4.0 o superior y que group.coordinator.rebalance.protocols incluye consumer. Sube las librerías cliente a una versión compatible en todos los servicios que comparten el group.id, aunque los vayas a migrar más tarde. Un cliente antiguo dentro de un grupo que ya negoció el protocolo nuevo no entra: se queda fuera y no consume, y ese es el fallo más común de la migración.

Limpieza de configuración. Elimina las tres propiedades que dejan de existir. Hazlo en un despliegue propio, sin cambiar group.protocol todavía, para que si algo peta sepas exactamente qué fue. Este despliegue debería ser un no-op funcional.

El interruptor. Añade group.protocol=consumer y haz un rolling restart, instancia a instancia. Cuando el primer miembro con el protocolo nuevo entra en un grupo clásico, el coordinador convierte el grupo entero. Durante la transición conviven miembros de ambos tipos y el coordinador traduce entre ellos. No es un estado en el que quieras vivir semanas, pero sí es seguro durante el tiempo de un despliegue.

Verificación. Después de cada tanda, mira el tipo de protocolo del grupo y que el lag baja:

```
# Tipo de protocolo, estado y miembros
kafka-consumer-groups.sh --bootstrap-server kafka-1:9092 \
  --describe --group orders-processor --state

# Reparto real por miembro: quién tiene qué y cuánto lag arrastra
kafka-consumer-groups.sh --bootstrap-server kafka-1:9092 \
  --describe --group orders-processor --members --verbose
```

La vuelta atrás. El camino de regreso es el mismo al revés: quitas group.protocol=consumer, devuelves las propiedades clásicas y haces otro rolling restart. El grupo vuelve a ser clásico cuando el último miembro con protocolo nuevo se va. Que exista esta salida es lo que hace razonable migrar en horario laboral en lugar de un domingo a las tres de la mañana. Ten preparado el fichero de configuración de vuelta antes de empezar, no lo escribas bajo presión.

Un consejo de secuenciación: migra primero un grupo pequeño y aburrido, uno cuyo lag a nadie le quita el sueño. Deja pasar una semana con él. La mayor parte de lo que sale mal en estas migraciones no sale mal en el despliegue, sale mal tres días después con un pico de tráfico.

Autoescalado: la causa de rebalanceos que nadie mira

Hay una fuente de rebalanceos que no aparece en ninguna guía de tuning de Kafka porque no vive en Kafka: el autoescalador. Un HPA agresivo sobre una métrica ruidosa puede añadir y quitar réplicas cada minuto, y cada una de esas operaciones es un rebalanceo. El grupo pasa más tiempo repartiendo particiones que procesando.

Tres reglas que evitan el problema:

- Escala por lag, no por CPU. La CPU de un consumidor de Kafka sube y baja con la forma del tráfico y con el GC. El lag del grupo es la métrica que de verdad dice si necesitas más manos. KEDA con el scaler de Kafka lee el lag directamente del grupo.
- Ventanas de estabilización asimétricas. Sube rápido y baja despacio. Añadir un consumidor cuesta un rebalanceo y resuelve un problema; quitarlo cuesta otro rebalanceo y no resuelve nada urgente. Un scaleDown.stabilizationWindowSeconds de 300 o más es lo normal.
- El número de particiones es el techo. Con 12 particiones, la réplica número 13 no procesa nada: se queda con cero particiones asignadas, consumiendo memoria y generando latidos. Peor aún, su entrada y salida provoca rebalanceos reales que sí afectan a los otros doce. Fija maxReplicaCount al número de particiones y ahórrate la clase entera de incidentes.

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: orders-processor
spec:
  scaleTargetRef:
    name: orders-processor
  minReplicaCount: 2
  maxReplicaCount: 12          # = número de particiones del topic
  cooldownPeriod: 300
  triggers:
    - type: kafka
      metadata:
        bootstrapServers: kafka-1:9092
        consumerGroup: orders-processor
        topic: orders
```

```
      lagThreshold: "2000"
      offsetResetPolicy: latest
  advanced:
    horizontalPodAutoscalerConfig:
      behavior:
        scaleUp:
          stabilizationWindowSeconds: 30
        scaleDown:
          stabilizationWindowSeconds: 600
          policies:
            - type: Pods
              value: 1
              periodSeconds: 120
```

Si además combinas esto con pertenencia estática, gana otra propiedad interesante: los reinicios por actualización de imagen dejan de contar como cambios de topología, y solo los cambios reales de tamaño mueven particiones. La diferencia en el gráfico de rebalance-rate-per-hour suele ser de un orden de magnitud.

Kafka Streams: tareas, standbys y el protocolo KIP-1071

Kafka Streams usa consumidores por debajo, pero su unidad de reparto no es la partición: es la tarea. Una tarea agrupa una partición de cada topic de entrada de una subtopología, junto con los almacenes de estado asociados. Eso hace que un rebalanceo en Streams sea considerablemente más caro: no basta con empezar a leer desde otro offset, hay que reconstruir el estado local releyendo el changelog desde el principio o desde el último checkpoint.

De ahí sale la configuración que más impacto tiene en la disponibilidad de una aplicación Streams:

- num.standby.replicas — mantiene copias calientes del estado en otras instancias. Cuesta disco y ancho de banda, pero convierte un failover de minutos en uno de segundos. Para cualquier aplicación con estado que importe, uno es el mínimo razonable.
- acceptable.recovery.lag — cuánto puede ir por detrás un standby para que se considere apto y se le entregue la tarea directamente. Por defecto 10.000 registros.
- max.warmup.replicas — cuántas tareas puede estar calentando el sistema a la vez cuando decide moverlas. Limita el impacto del rebalanceo probabilístico que Streams hace en segundo plano.
- probing.rebalance.interval.ms — cada cuánto Streams comprueba si ya puede mover una tarea a su ubicación preferida. Diez minutos por defecto.

El comportamiento que sorprende: Streams hace rebalanceos a propósito, de forma periódica, para ir migrando tareas hacia instancias que ya tienen el estado caliente. Se llama rebalanceo probatorio y verlo en los logs no es señal de un problema. Lo que sí es señal de un problema es que esos rebalanceos nunca converjan, lo que suele significar que los standbys no consiguen ponerse al día.

```
Properties props = new Properties();
props.put(StreamsConfig.APPLICATION_ID_CONFIG, "orders-enrichment");
props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "kafka-1:9092");

// Identidad estable: los reinicios no mueven tareas
props.put(StreamsConfig.consumerPrefix("group.instance.id"),
          System.getenv("POD_NAME"));

// Estado caliente en otra instancia
```

```
props.put(StreamsConfig.NUM_STANDBY_REPLICAS_CONFIG, 1);
props.put(StreamsConfig.ACCEPTABLE_RECOVERY_LAG_CONFIG, 10_000L);
props.put(StreamsConfig.MAX_WARMUP_REPLICAS_CONFIG, 2);

// Margen para reconstruir estado sin que nos expulsen
props.put(StreamsConfig.consumerPrefix("max.poll.interval.ms"), 600_000);
```

La novedad reciente es KIP-1071, el protocolo de rebalanceo específico para Streams, que lleva al mundo de las tareas la misma idea de KIP-848: el broker calcula la asignación. Elimina la coordinación en el cliente, guarda la topología de forma centralizada y añade RPCs y herramientas administrativas propias, con lo que por fin se puede preguntar al clúster por qué se movió una tarea. Su conjunto de funcionalidades principales se considera listo para producción a partir de Kafka 4.2. Si tu aplicación Streams sufre pausas largas o churn frecuente, es la mejora estructural que estabas esperando; si funciona bien, no hay prisa.

Cuando el reparto exclusivo es el problema: share groups

Todo lo anterior parte de una premisa: una partición pertenece a un consumidor a la vez. Esa premisa es la que te da orden por clave, y también la que te obliga a rebalancear.

Hay cargas de trabajo donde el orden no importa nada. Repartir trabajos entre workers, procesar tareas independientes, consumir una cola de peticiones. En esos casos, el reparto exclusivo es puro coste: el paralelismo está limitado por el número de particiones, un consumidor lento retrasa a toda su partición, y cada cambio en la flota dispara un rebalanceo.

KIP-932, las colas para Kafka, introduce los share groups: varios consumidores leyendo la misma partición a la vez, con acuse de recibo por registro individual y contador de entregas. No hay asignación exclusiva de particiones, así que no hay rebalanceo en el sentido clásico. Puedes tener 50 consumidores sobre un topic de 6 particiones y todos trabajan. Un registro que falla se puede reintentar o mandar a una cola de errores sin bloquear a los demás. La funcionalidad se considera lista para producción en Apache Kafka 4.2.

La regla de decisión es simple y merece la pena escribirla en el documento de diseño: si necesitas orden por clave o estado local por partición, usa consumer groups y aprende a rebalancear bien. Si solo necesitas repartir trabajo, un share group elimina el problema en lugar de optimizarlo. Mucho esfuerzo de tuning se gasta en grupos que nunca necesitaron garantías de orden.

Leer un rebalanceo en los logs

Cuando algo va mal, la respuesta está casi siempre en los logs del cliente, pero hay que saber qué líneas importan. Esta es la secuencia de un rebalanceo sano en el protocolo clásico, anotada:

```
[ConsumerCoordinator] Revoke previously assigned partitions orders-3, orders-7
   -> tu onPartitionsRevoked está a punto de ejecutarse. El reloj corre.

[ConsumerCoordinator] (Re-)joining group
   -> JoinGroup enviado. A partir de aquí esperamos al miembro más lento.

[ConsumerCoordinator] Successfully joined group with generation Generation{generationId=42, ...}
   -> la barrera se abrió. Fíjate en el número: si crece varias veces por minuto, tienes churn.

[ConsumerCoordinator] Finished assignment for group at generation 42
   -> este mensaje SOLO aparece en el líder. Sirve para saber quién repartió.
```

```
[ConsumerCoordinator] Notifying assignor about the new Assignment(partitions=[orders-3, orders-11])
[ConsumerCoordinator] Adding newly assigned partitions: orders-11
   -> reparto entregado. En cooperativo verás "Adding" sin un "Revoke" previo de todo.
```

Y estas son las líneas que indican un problema real, con su traducción:

```
[ConsumerCoordinator] Member consumer-1-abc sending LeaveGroup request ...
  because the poll loop has exceeded max.poll.interval.ms
   -> Tu procesamiento tarda demasiado. No subas el timeout sin pensar:
      baja max.poll.records o pausa las particiones.

[ConsumerCoordinator] Attempt to heartbeat failed since group is rebalancing
   -> Normal si aparece una vez. Si se repite en bucle, el grupo no estabiliza.

[ConsumerCoordinator] Offset commit failed on partition orders-3 at offset 91820:
  The coordinator is not aware of this member.
   -> Te expulsaron y no te enteraste. Aquí es donde onPartitionsLost debería
      haber descartado el trabajo en vuelo.

[AbstractCoordinator] Group coordinator kafka-2:9092 is unavailable or invalid,
  will attempt rediscovery
   -> El problema es del clúster, no tuyo. Mira los brokers.
```

Un truco operativo que vale su peso en oro: cuando sospeches churn, no mires los logs de un pod, cuenta generaciones. Un simple recuento del generationId distinto por minuto en todos los pods te dice en diez segundos si tienes un problema de estabilidad o un problema de rendimiento. Son cosas distintas y se arreglan de forma distinta.

Dimensionar los timeouts con aritmética, no con intuición

La mayoría de los valores de max.poll.interval.ms que hay en producción se eligieron subiendo el número hasta que dejaron de aparecer errores. Se puede hacer mejor, y la cuenta es de servilleta.

Necesitas tres datos que ya tienes en tus métricas: el percentil 99 del tiempo de procesamiento por registro, cuántos registros devuelve un poll(), y cuánto tarda tu onPartitionsRevoked en el peor caso.

La regla es:

```
max.poll.interval.ms  >=  (p99_por_registro * max.poll.records) * factor_seguridad
rebalance.timeout.ms  >=  tiempo_de_flush_p99 * factor_seguridad
```

Con un factor de seguridad de 2 o 3, porque el p99 de un registro no es el p99 de un lote entero: los lotes acumulan cola. Un ejemplo concreto. Supón que enriqueces cada pedido con una llamada HTTP cuyo p99 es de 180 ms, y tienes max.poll.records=500:

```
0.180 s * 500 registros = 90 s por lote en el peor caso
90 s * 3 (factor de seguridad) = 270 s
max.poll.interval.ms = 300000   # 300 s, con margen
```

Ahora mira la otra cara: ese consumidor tarda hasta 90 segundos en volver a llamar a poll(). Durante 90 segundos no puede reaccionar a una revocación. Si tienes un despliegue en marcha, el rebalanceo se alarga hasta ese punto. La solución no es subir más el timeout, es bajar el lote:

```
max.poll.records = 100
0.180 s * 100 = 18 s por lote
18 s * 3 = 54 s  ->  max.poll.interval.ms = 60000
```

Mismo trabajo por segundo, cuatro veces más reactivo ante un rebalanceo. Este es el intercambio central y casi nadie lo hace explícito: lotes grandes optimizan el rendimiento en régimen estacionario; lotes pequeños optimizan la latencia de recuperación. Como los rebalanceos ocurren precisamente en los momentos malos, el lote pequeño suele ganar en la métrica que de verdad ve el usuario.

Si la llamada HTTP es el cuello de botella, hay una tercera vía mejor que las dos: paralelizar dentro del lote. Procesa los 500 registros con un pool de 20 hilos y el tiempo de lote baja a 4,5 segundos sin tocar el tamaño. Eso sí, mantén el commit en el hilo del poll() y confirma solo el prefijo completado, o habrás cambiado un problema de timeouts por un problema de huecos en los offsets.

Probar rebalanceos antes de que producción los pruebe por ti

El código de rebalanceo es, por definición, el que solo se ejecuta cuando algo cambia. Por eso casi nunca está cubierto por tests y por eso falla la primera vez que se ejecuta de verdad. Con Testcontainers es sencillo forzarlo en CI.

La prueba mínima que deberías tener: arranca un broker, crea un topic con varias particiones, arranca dos consumidores, mata uno, y verifica dos cosas — que todas las particiones acaban asignadas y que no se perdió ningún mensaje.

```
import pytest, threading, time
from testcontainers.kafka import KafkaContainer
from confluent_kafka import Consumer, Producer
from confluent_kafka.admin import AdminClient, NewTopic

@pytest.fixture(scope="module")
def broker():
    with KafkaContainer() as kafka:
        yield kafka.get_bootstrap_server()

def test_no_se_pierden_mensajes_en_rebalanceo(broker):
    AdminClient({"bootstrap.servers": broker}).create_topics(
        [NewTopic("t", num_partitions=6, replication_factor=1)]
    )
    p = Producer({"bootstrap.servers": broker})
    for i in range(3000):
        p.produce("t", key=str(i), value=str(i))
    p.flush()

    vistos, lock, parar = set(), threading.Lock(), threading.Event()

    def worker(nombre):
        c = Consumer({
            "bootstrap.servers": broker,
            "group.id": "test-grupo",
            "auto.offset.reset": "earliest",
            "enable.auto.commit": False,
            "partition.assignment.strategy": "cooperative-sticky",
        })
        c.subscribe(["t"])
        while not parar.is_set():
            m = c.poll(1.0)
            if m and not m.error():
                with lock:
```

```
                    vistos.add(int(m.value()))
                c.commit(message=m, asynchronous=False)
        c.close()

    a = threading.Thread(target=worker, args=("a",)); a.start()
    b = threading.Thread(target=worker, args=("b",)); b.start()
    time.sleep(10)

    parar.set(); a.join(); b.join()
    assert len(vistos) == 3000, f"faltan {3000 - len(vistos)} mensajes"
```

Es un test lento —diez segundos— y por eso mucha gente no lo escribe. Compáralo con el coste de descubrir en producción que tu on_revoke lanza una excepción no capturada. Ponlo en una suite de integración que corra en cada merge a la rama principal, no en cada commit, y tendrás la cobertura donde importa sin castigar el bucle de desarrollo.

Una variante útil del mismo test: en lugar de matar un consumidor, añade un tercero a mitad de camino y comprueba que ninguna partición queda sin dueño durante más de N segundos. Eso valida el camino cooperativo, que es el que de verdad usarás en cada despliegue.

Runbook: el grupo lleva veinte minutos rebalanceando

Un grupo que no consigue estabilizarse es el peor escenario porque el lag crece mientras nadie procesa. Este es el orden en el que hay que mirar las cosas, de más probable a menos.

Paso 1 — Confirma el síntoma. Mira el estado del grupo. Si aparece PreparingRebalance o CompletingRebalance de forma persistente, es churn real; si aparece Stable pero el lag sube, el problema es de rendimiento y este runbook no es el tuyo.

```
watch -n2 'kafka-consumer-groups.sh --bootstrap-server kafka-1:9092 \
  --describe --group orders-processor --state'
```

Paso 2 — Cuenta miembros. Si el número de miembros oscila, tienes pods entrando y saliendo. Mira reinicios del deployment, OOMKills y readiness probes fallando. Nueve de cada diez veces el rebalanceo perpetuo es un pod en CrashLoopBackOff, no un problema de Kafka.

```
kubectl get pods -l app=orders-processor \
  -o custom-columns=NAME:.metadata.name,RESTARTS:.status.containerStatuses[0].restartCount
```

Paso 3 — Busca el expulsado. Filtra los logs por max.poll.interval. Si un pod concreto aparece siempre, tienes un consumidor lento: un mensaje envenenado, una partición con claves calientes, o una dependencia externa degradada solo para ese subconjunto de datos.

Paso 4 — Busca al desconocido. Comprueba que no hay una versión antigua conviviendo. En una migración a medias, o durante un despliegue canary, es fácil acabar con dos versiones que anuncian conjuntos de assignors incompatibles. El síntoma clásico es que el grupo rebalancea y rebalancea sin llegar nunca a Stable.

Paso 5 — Mitiga, luego arregla. Si estás sangrando lag, la mitigación más rápida casi siempre es la misma: fija el número de réplicas (desactiva el autoescalado), y si hay un pod que se reinicia en bucle, sácalo del grupo escalando a cero y volviendo a subir con la configuración corregida. Un grupo estable con menos

consumidores procesa infinitamente más que un grupo inestable con muchos.

Después, la parte que importa de verdad: escribe qué cambió. Los rebalanceos perpetuos casi nunca aparecen solos. Aparecen después de un despliegue, de un cambio de límites de memoria, de añadir particiones a un topic o de que un servicio del que dependes se puso lento. La pregunta "¿qué se desplegó en las dos horas anteriores?" resuelve más incidentes de Kafka que cualquier ajuste de configuración.

Transacciones y exactly-once: el rebalanceo que rompe tu productor

Si tu servicio hace leer-procesar-escribir y quieres que las tres cosas ocurran juntas o no ocurran, necesitas un productor transaccional. Y ahí el rebalanceo deja de ser un problema de latencia para convertirse en un problema de correccion, porque el mecanismo que protege a Kafka de los zombis (el fencing) y el mecanismo que reparte particiones (el rebalanceo) hablan de la misma cosa desde dos sitios distintos.

El fencing transaccional se apoya en el transactional.id: cuando un productor se registra con ese id, el broker le da un epoch y expulsa a cualquier productor anterior con el mismo id y un epoch menor. El problema es que la propiedad de una particion se mueve entre miembros en cada rebalanceo. Antes de Kafka 2.6 la unica forma de que el fencing fuera correcto era codificar la particion de entrada dentro del transactional.id, algo asi como pedidos-in-7. Eso funcionaba, pero obligaba a mantener un productor por particion asignada: cientos de conexiones, cientos de buffers, y una reconfiguracion completa cada vez que cambiaba la asignacion.

KIP-447 lo resolvio moviendo la comprobacion al momento del commit. Ahora el productor envia los metadatos del grupo junto con los offsets, y el broker compara el generation id que trae el productor con el generation actual del grupo. Si tu instancia quedo fuera del grupo mientras procesaba, tu generation es vieja, la transaccion se rechaza entera y quien tenga ahora la particion la reprocesa desde el ultimo offset comprometido. Un productor por instancia, no por particion.

```
// Java: read-process-write atomico, un solo productor por instancia
producer.initTransactions();

while (running) {
    ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(500));
    if (records.isEmpty()) continue;

    producer.beginTransaction();
    try {
        Map<TopicPartition, OffsetAndMetadata> offsets = new HashMap<>();
        for (ConsumerRecord<String, String> r : records) {
            producer.send(new ProducerRecord<>("salida", r.key(), transform(r.value())));
            offsets.put(new TopicPartition(r.topic(), r.partition()),
                        new OffsetAndMetadata(r.offset() + 1));
        }

        // La clave de KIP-447: groupMetadata() lleva memberId y generationId.
        // El broker valla la transaccion si esta instancia ya no pertenece a esa generacion.
        producer.sendOffsetsToTransaction(offsets, consumer.groupMetadata());
        producer.commitTransaction();

    } catch (ProducerFencedException | CommitFailedException e) {
        // Nos echaron del grupo. La transaccion ya esta abortada del lado del broker.
        // Cerrar y recrear el productor; el trabajo lo rehara quien tenga ahora la particion.
        producer.close();
        throw e;
    } catch (KafkaException e) {
        producer.abortTransaction();
    }
```

```
}
```

Tres detalles que se pagan caros si se ignoran. El primero: los consumidores que leen el topic de salida tienen que ir con isolation.level=read_committed, o veran los mensajes de transacciones abortadas y todo el esfuerzo habra sido decorativo. El segundo: enable.auto.commit debe estar en false, porque el commit lo hace la transaccion, no el consumidor.

El tercero es el que descubre la gente a las tres de la manana. Una transaccion abierta bloquea el last stable offset del topic de salida: los lectores read_committed no avanzan mas alla del primer mensaje de una transaccion en curso. Si tu instancia cede particiones con una transaccion abierta y se limita a irse, esa transaccion se queda viva hasta que expire transaction.timeout.ms (60 segundos por defecto), y durante ese minuto tus consumidores de salida parecen colgados aunque el productor nuevo este escribiendo sin problemas. Por eso el listener de revocacion tiene que abortar:

```
consumer.subscribe(List.of("entrada"), new ConsumerRebalanceListener() {
    @Override
    public void onPartitionsRevoked(Collection<TopicPartition> revoked) {
        // No hay commit aqui: los offsets van dentro de la transaccion.
        // Lo que si hay que hacer es no dejar transacciones abiertas colgando el LSO.
        if (transactionInFlight) {
            producer.abortTransaction();
            transactionInFlight = false;
        }
    }

    @Override
    public void onPartitionsLost(Collection<TopicPartition> lost) {
        // Ya no somos duenos de nada; abortar y no intentar commit.
        if (transactionInFlight) {
            producer.abortTransaction();
            transactionInFlight = false;
        }
    }

    @Override
    public void onPartitionsAssigned(Collection<TopicPartition> assigned) { }
});
```

Sobre los timeouts: transaction.timeout.ms del productor no puede superar transaction.max.timeout.ms del broker (15 minutos por defecto), y conviene que sea comodamente mayor que el tiempo real de un ciclo poll-procesar-commit. Si lo dejas en 60 segundos y tu lote tarda 90, cada transaccion muere sola y el servicio entra en un bucle de aborto que parece un problema de red. Y al reves: un transaction.timeout.ms muy alto retrasa la recuperacion cuando una instancia muere de verdad, porque el LSO no se libera hasta que expira.

Un matiz que suele sorprender: con transacciones, un rebalanceo no duplica mensajes, pero si tira trabajo. Todo lo procesado desde el ultimo commit transaccional se descarta y se rehace. Si tus transacciones abarcan lotes grandes para amortizar el coste del commit, un grupo que rebalancea a menudo puede estar rehaciendo el 30% de su trabajo sin que ninguna metrica de lag lo delate. Mide la tasa de abortos, no solo el lag.

Si usas Kafka Streams, processing.guarantee=exactly_once_v2 implementa exactamente este patron por debajo, incluido el manejo de la revocacion. La ventaja real de v2 sobre el exactly_once original es precisamente KIP-447: un productor por hilo de stream en lugar de uno por tarea, que es lo que hizo viable EOS en topologias grandes.

Fuera de la JVM: rebalanceo cooperativo en librdkafka

Casi todo lo que se escribe sobre rebalanceos asume el cliente Java. Pero si tu servicio esta en Python, Go, C# o Rust, lo mas probable es que estes usando librdkafka por debajo, y ahi el contrato es distinto en un punto que rompe cosas en silencio.

La regla es esta: en cuanto registras un callback de rebalanceo, librdkafka deja de asignar y revocar particiones por su cuenta. Pasa a ser responsabilidad tuya, y el metodo que tienes que llamar depende del assignor configurado. Con assignors eager (range, roundrobin) usas assign() y unassign() con la lista completa. Con cooperative-sticky usas incremental_assign() y incremental_unassign(), que suman o restan sobre la asignacion actual. Mezclarlas es el error clasico: llamar a assign() dentro de un callback cooperativo reemplaza toda la asignacion en lugar de anadir, y el consumidor empieza a perder particiones que nadie le habia revocado. El sintoma no es una excepcion clara, es lag en particiones que deberian estar cubiertas.

```
from confluent_kafka import Consumer, TopicPartition, KafkaException

def on_assign(consumer, partitions):
    # Con cooperative-sticky, "partitions" son SOLO las nuevas.
    # incremental_assign las suma a lo que ya teniamos.
    for tp in partitions:
        estado.cargar(tp.topic, tp.partition)   # calentar cache, abrir ficheros, etc.
    consumer.incremental_assign(partitions)

def on_revoke(consumer, partitions):
    # "partitions" son SOLO las que se van. Las demas siguen fluyendo.
    # Committear solo estas: un commit global aqui pisa offsets ajenos.
    offsets = [TopicPartition(tp.topic, tp.partition, pendientes[(tp.topic, tp.partition)])
               for tp in partitions if (tp.topic, tp.partition) in pendientes]
    if offsets:
        try:
            consumer.commit(offsets=offsets, asynchronous=False)
        except KafkaException as e:
            log.warning("commit en revoke fallo, se reprocesara: %s", e)
    for tp in partitions:
        estado.descargar(tp.topic, tp.partition)
    consumer.incremental_unassign(partitions)

def on_lost(consumer, partitions):
    # Perdimos la sesion (timeout, particion de red). Ya no somos duenos:
    # NO committear, el broker rechazaria o pisaria offsets de otro miembro.
    for tp in partitions:
        estado.descartar(tp.topic, tp.partition)
    consumer.incremental_unassign(partitions)

consumer = Consumer({
    "bootstrap.servers": "kafka:9092",
    "group.id": "pedidos",
    "partition.assignment.strategy": "cooperative-sticky",
    "enable.auto.commit": False,
    "session.timeout.ms": 45000,
    "max.poll.interval.ms": 300000,
})
consumer.subscribe(["pedidos"], on_assign=on_assign, on_revoke=on_revoke, on_lost=on_lost)
```

Ese tercer callback, on_lost, es el equivalente de onPartitionsLost en Java y en la practica es el que separa un consumidor correcto de uno que corrompe offsets. on_revoke se llama cuando cedemos particiones ordenadamente y todavia somos miembros validos del grupo, asi que committear tiene sentido. on_lost se llama cuando ya nos han echado: el generation id que tenemos esta caducado y cualquier commit o se rechaza o, peor, retrocede el offset de un companero que ya avanzo. Si tu libreria no expone on_lost por

separado y solo tienes on_revoke, comprueba la version antes de asumir que estas cubierto.

Otro detalle que muerde: en librdkafka, max.poll.interval.ms se evalua contra el tiempo entre llamadas a poll() o consume(), igual que en Java, pero muchos wrappers de alto nivel esconden el poll dentro de un iterador. Un for record in consumer que hace trabajo largo por mensaje esta consumiendo ese presupuesto sin que se vea en el codigo. Si procesas en lotes, procesa en lotes de verdad con consume(num_messages=N, timeout=...) y controla tu el ritmo.

Para migrar de eager a cooperativo fuera de la JVM aplica la misma regla de los dos despliegues que en Java: primero todos los miembros con la lista cooperative-sticky,range para que sigan eligiendo range mientras convive gente vieja, y en el segundo despliegue se deja solo cooperative-sticky. Lo que cambia es que tienes que reescribir el callback antes del primer despliegue, no despues, porque el codigo con assign() dejara de ser correcto en el momento en que el grupo elija el assignor cooperativo.

Kafka Connect también rebalancea, y casi nadie lo mira

Un cluster de Connect es un grupo de consumidores mas. No reparte particiones: reparte connectors y tasks, pero usa el mismo coordinador de grupo, los mismos timeouts y las mismas patologias. Y como la mayoria de la gente instala Connect y se olvida, los rebalanceos de Connect suelen ser el punto ciego de la plataforma.

Curiosamente, Connect llego antes al rebalanceo cooperativo que el propio consumidor: KIP-415 aterrizo en Kafka 2.3, mientras que KIP-429 para el consumidor llego en 2.4. La motivacion era la misma, el efecto stop-the-world. Con el protocolo eager, anadir un worker o publicar la configuracion de un connector nuevo paraba todas las tasks del cluster, se recalculaba el reparto y se arrancaba todo otra vez. En un cluster con cincuenta connectors eso son varios segundos de nada moviendose cada vez que alguien toca un endpoint de administracion.

La configuracion relevante vive en el fichero del worker:

```
# Protocolo de rebalanceo del cluster de Connect.
#   eager      -> stop-the-world clasico (solo por compatibilidad)
#   compatible -> cooperativo incremental (KIP-415, Kafka 2.3)
#   sessioned  -> cooperativo + tokens de sesion firmados para el trafico interno
#                 entre workers. Es el valor por defecto desde Kafka 2.4.
connect.protocol=sessioned

# Cuando un worker desaparece, sus tasks quedan "lost". En lugar de reasignarlas
# de inmediato, Connect espera este tiempo por si el worker vuelve: un reinicio
# rodante no deberia mover trabajo de sitio.
# Por defecto son 5 minutos.
scheduled.rebalance.max.delay.ms=300000

# Tiempo maximo que el coordinador espera a que todos los workers se unan.
# Debe cubrir el arranque mas lento de un worker con todos sus plugins.
rebalance.timeout.ms=60000
session.timeout.ms=45000
heartbeat.interval.ms=3000
```

scheduled.rebalance.max.delay.ms es la pieza interesante y la que mas se toquetea mal. Su proposito es absorber reinicios: si tiras un worker para actualizarlo y vuelve en dos minutos, sus tasks siguen paradas pero nadie ha movido nada, y al volver las recupera tal cual. Sin esa espera, cada reinicio de worker provoca dos redistribuciones (una al irse, otra al volver) con el coste de reabrir conexiones a bases de datos, releer estado de los connectors de origen y, en connectors de sumidero, rehacer buffers.

La tentacion, cuando un worker muere de verdad y ves cinco minutos de tasks paradas, es bajarlo a cero. No lo hagas sin leer KAFKA-15693: desactivar el retraso puede dejar connectors y tasks sin asignar de forma indefinida en ciertas secuencias de rebalanceo. Si necesitas reaccionar mas rapido ante caidas reales, baja el valor a algo como 60000 en vez de anularlo, y asegurate de que tus despliegues rodantes reinician un worker en menos de ese tiempo.

Las tormentas de rebalanceo en Connect tienen ademas una causa propia que no existe en un consumidor normal: un connector que falla al arrancar y se reintenta en bucle. Cada intento fallido puede provocar una nueva ronda de asignacion, y el cluster se pasa el dia repartiendo trabajo que nunca llega a ejecutarse. La senal esta en el log del worker, no en las metricas del consumidor:

```
# Estado real de las tasks: busca FAILED en bucle o UNASSIGNED persistente
curl -s localhost:8083/connectors?expand=status | jq -r '
  to_entries[] | .key as $c |
  .value.status.tasks[]? | "\($c)\ttask=\(.id)\t\(.state)"' | sort | uniq -c

# Cuantas veces ha rebalanceado el cluster y cuanto ha durado la ultima ronda
curl -s localhost:8083/ | jq .

# En el log del worker, las lineas que importan:
#   "Rebalance started"
#   "Successfully joined group ... with generation N"   <- N subiendo rapido = tormenta
#   "Wasn't able to resume work after last rebalance"
```

Regla practica: si el generation id del grupo de Connect sube mas de una o dos veces por hora sin que nadie haya desplegado nada, tienes un connector enfermo o un timeout mal dimensionado, no un cluster ocupado. Y merece la pena vigilarlo con la misma alerta que usas para los grupos de consumidores de aplicacion, porque un Connect que rebalancea sin parar no genera lag visible en sus propios topics hasta que ya llevas horas perdiendo throughput.

Despliegues rodantes en Kubernetes que no despiertan al coordinador

La pertenencia estatica solo sirve si el identificador es realmente estable, y en Kubernetes eso significa elegir bien el objeto que ejecuta tus consumidores. Un Deployment genera nombres de pod aleatorios: cada reinicio cambia el nombre, y si derivas group.instance.id del hostname acabas creando un miembro nuevo en cada despliegue, que es exactamente lo contrario de lo que buscabas. Un StatefulSet da ordinales estables (consumidor-0, consumidor-1) que sobreviven a reinicios y actualizaciones.

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: pedidos-consumer
spec:
  replicas: 6
  serviceName: pedidos-consumer
  # OrderedReady reinicia de uno en uno: un solo miembro fuera a la vez.
  podManagementPolicy: OrderedReady
  updateStrategy:
    type: RollingUpdate
  template:
    spec:
      # Debe cubrir: drenar el lote en curso + commit + cierre limpio.
      terminationGracePeriodSeconds: 45
      containers:
        - name: app
          env:
```

```
            # El ordinal del pod es un group.instance.id gratis y estable.
            - name: POD_NAME
              valueFrom:
                fieldRef:
                  fieldPath: metadata.name
            - name: KAFKA_GROUP_INSTANCE_ID
              value: "$(POD_NAME)"
          lifecycle:
            preStop:
              exec:
                # Senal a la app para que deje de pedir mensajes nuevos y
                # cierre el lote en curso antes de que llegue el SIGTERM.
                command: ["/bin/sh", "-c", "touch /tmp/drain && sleep 20"]
          readinessProbe:
            # No marcar ready hasta despues del primer poll con particiones
            # asignadas. Si no, el rollout avanza al siguiente pod mientras
            # este todavia no consume nada.
            httpGet: { path: /ready, port: 8080 }
            periodSeconds: 2
            failureThreshold: 30
```

La aritmetica del rollout es la parte que se salta todo el mundo. Con pertenencia estatica, el coordinador no rebalancea si el miembro vuelve antes de que expire session.timeout.ms. Ese reloj empieza en el ultimo heartbeat del pod viejo y tiene que cubrir todo esto: el tiempo de gracia de terminacion, la programacion del pod nuevo, el pull de la imagen si no esta en cache, el arranque del proceso y el primer poll. En una JVM con Spring Boot y treinta beans, eso son facilmente 40 o 50 segundos. Un session.timeout.ms de 45000 con un terminationGracePeriodSeconds de 45 no deja margen para nada: el rebalanceo llega igual y encima te preguntas por que la pertenencia estatica no funciona.

Dos formas honestas de cuadrarlo. La primera es subir session.timeout.ms hasta cubrir el peor arranque con holgura, sabiendo que a cambio una caida real tarda mas en detectarse (y que el broker impone un maximo con group.max.session.timeout.ms, 30 minutos por defecto, pero cualquier valor por encima de dos o tres minutos convierte una caida en un incidente largo). La segunda, mejor si puedes, es reducir el arranque: imagenes precargadas en el nodo, lazy init de lo que no hace falta para consumir, y el primer poll lo antes posible en el ciclo de vida de la aplicacion, no despues de calentar todas las caches.

```
# Medir el hueco real del rollout, que es lo unico que importa:
kubectl get pods -l app=pedidos-consumer -w --output-watch-events \
  -o custom-columns=T:.metadata.creationTimestamp,NAME:.metadata.name,PHASE:.status.phase

# Y en paralelo, si el coordinador rebalanceo o no:
watch -n2 'kafka-consumer-groups.sh --bootstrap-server kafka:9092 \
  --describe --group pedidos --state | tail -3'
# STABLE durante todo el rollout = la pertenencia estatica esta haciendo su trabajo.
# PREPARING_REBALANCE en cada pod = el hueco supera session.timeout.ms.
```

Un aviso sobre podManagementPolicy: Parallel arranca y para todos los pods a la vez, lo cual acelera el despliegue y destruye la ventaja de la pertenencia estatica, porque el grupo entero desaparece durante unos segundos y el coordinador no tiene mas remedio que rehacer el reparto. Si necesitas velocidad, OrderedReady con un arranque rapido bate a Parallel con un arranque lento casi siempre, porque el rollout ordenado no genera trabajo extra aguas abajo.

Por ultimo, no confundas el escalado con el despliegue. Escalar de 6 a 8 replicas si tiene que rebalancear: hay particiones que cambian de dueno de verdad. Lo que la pertenencia estatica evita es el rebalanceo gratuito del reinicio, donde el reparto final es identico al inicial. Si tu HPA sube y baja replicas cada pocos minutos por una metrica ruidosa, ninguna configuracion de grupo te va a salvar: arregla la metrica o pon una ventana de estabilizacion generosa antes de tocar los timeouts de Kafka.

El techo del que nadie avisa: metadatos de grupo y grupos grandes

Hay un limite fisico en el protocolo clasico que no aparece en ninguna guia de introduccion y que aparece de golpe cuando un grupo crece: el estado del grupo tiene que caber en un mensaje de Kafka.

El mecanismo es este. En el JoinGroup, cada miembro envia su suscripcion y un bloque opaco de userData. En el SyncGroup, el lider envia la asignacion completa del grupo y el coordinador la reparte. Y el coordinador persiste todo eso como un registro en __consumer_offsets, para poder reconstruir el grupo si el broker que lo coordina se cae. Ese registro esta sujeto a max.message.bytes del topic, que por defecto es aproximadamente 1 MB.

Con el assignor range o roundrobin, userData va practicamente vacio y el problema tarda en llegar. Con sticky y cooperative-sticky, userData contiene la asignacion anterior del miembro, porque es la informacion que el assignor necesita para minimizar movimientos. Es decir, el tamano del estado crece aproximadamente con miembros por particiones asignadas. Un grupo de 400 miembros consumiendo un topic de 3.000 particiones, con nombres de topic largos como eventos-facturacion-normalizados-v3, se planta en el limite sin hacer nada raro.

```
# Sintoma: el grupo no consigue estabilizarse y el log del coordinador dice
#   org.apache.kafka.common.errors.RecordTooLargeException
#   ... appending metadata message for group pedidos generation 812
# El grupo se queda dando vueltas en PREPARING_REBALANCE o CompletingRebalance.

# Diagnostico rapido: tamano del grupo y particiones totales
kafka-consumer-groups.sh --bootstrap-server kafka:9092 --describe --group pedidos --members \
  | awk 'NR>1 {m++; p+=$3} END {print "miembros="m, "particiones="p, "producto="m*p}'

# Y el tamano real de los registros de metadatos del grupo:
kafka-log-dirs.sh --bootstrap-server kafka:9092 \
  --topic-list __consumer_offsets --describe | jq '.brokers[].logDirs[].partitions[]
  | select(.size > 0) | {partition, size}' | head
```

Las salidas clasicas eran todas incomodas. Subir max.message.bytes de __consumer_offsets funciona, pero es un topic interno critico y estas aumentando el coste de cada escritura de estado para todos los grupos del cluster. Partir el grupo en varios grupos mas pequenos funciona, pero te obliga a repartir topics o a montar filtrado en cliente. Volver a range reduce el userData, pero pierdes el sticky y con el la razon por la que querias cooperative-sticky. Ninguna es buena.

Aqui es donde KIP-848 deja de ser una mejora de latencia y pasa a ser un cambio estructural. Al mover el calculo de la asignacion al broker, desaparece el viaje de ida y vuelta del userData: los miembros ya no publican su asignacion anterior para que el lider la lea, porque el coordinador ya la tiene. El estado se mantiene y se persiste por miembro y de forma incremental, no como un unico registro monolitico que contiene el grupo entero. El limite de 1 MB deja de ser una funcion de miembros por particiones, y con el desaparece toda una clase de incidentes que solo aparecian a escala.

El corolario practico para dimensionar hoy: si tu grupo pasa de un centenar de miembros o el producto de miembros por particiones asignadas se acerca a las decenas de miles, migrar al protocolo nuevo no es una optimizacion opcional, es la forma de quitarte un techo. Y si todavia no puedes migrar, mide ese producto y ponle una alerta antes de que te lo encuentres en produccion.

```
# Migracion al protocolo nuevo, resumen operativo (Kafka 4.x)
# 1) Habilitar en el broker (GA desde Kafka 4.0):
#    group.coordinator.rebalance.protocols=classic,consumer
# 2) Migrar consumidores de uno en uno, sin parar el grupo:
#    group.protocol=consumer
#    (partition.assignment.strategy, session.timeout.ms y heartbeat.interval.ms
#     dejan de aplicarse en el cliente: ahora los fija el servidor con
#     group.consumer.session.timeout.ms y group.consumer.heartbeat.interval.ms)
# 3) Comprobar el tipo de grupo resultante:
kafka-consumer-groups.sh --bootstrap-server kafka:9092 --list --group-type consumer
kafka-consumer-groups.sh --bootstrap-server kafka:9092 --describe --group pedidos --state
```

Conviene subrayar el punto 2 porque cambia como se opera un grupo. Con el protocolo nuevo, los timeouts de sesion y heartbeat dejan de ser decision del equipo que escribe el consumidor y pasan a ser politica de plataforma, fijada en el broker. Eso elimina la clase de incidente en la que un servicio despliega un session.timeout.ms absurdo y desestabiliza un grupo compartido, pero tambien significa que ya no puedes ajustarlo desde el codigo cuando lo necesites: si tu aplicacion arranca lenta, hablalo con quien opera el cluster antes de migrar, no despues. max.poll.interval.ms, en cambio, sigue siendo del cliente, porque mide algo que solo el cliente conoce: cuanto tarda tu codigo en volver a pedir mensajes.

La evolucion no se detiene ahi. KIP-1251 introduce epochs de asignacion para los grupos de consumidores, que afinan la reconciliacion cuando llegan varias asignaciones solapadas en poco tiempo, un caso que hoy se resuelve de forma conservadora. Es la clase de detalle que no cambia tu codigo, pero que reduce el numero de ciclos que tarda un grupo grande en converger despues de un cambio.

Qué llevarse de todo esto

Si tuviera que reducir todo lo anterior a cinco frases que puedas repetir en una revisión de diseño:

- Un rebalanceo no es un fallo; es el mecanismo por el que Kafka mantiene la promesa de que cada partición tiene exactamente un dueño. El objetivo no es evitarlo, es hacerlo barato.
- Los tres relojes hacen cosas distintas. session.timeout.ms detecta procesos muertos, heartbeat.interval.ms es su frecuencia de muestreo, y max.poll.interval.ms detecta procesos vivos pero atascados. Tunear el equivocado no arregla nada.
- Cooperativo más pertenencia estática elimina la mayor parte del dolor de los despliegues, y ambos son cambios de configuración, no de arquitectura.
- El protocolo nuevo de KIP-848 mueve la decisión al broker y quita la barrera global. Migrar es un procedimiento con vuelta atrás, no un salto de fe.
- La mayoría de los rebalanceos patológicos no vienen de Kafka. Vienen de un autoescalador nervioso, un pod que muere, o un bucle de procesamiento que tarda más de lo que prometió.

El día que un despliegue de tu consumidor deje de aparecer en el gráfico de lag, sabrás que lo tienes resuelto. Ese es el listón, y es alcanzable con las piezas que ya vienen en la caja.

English

Kafka Consumer Group Rebalancing: Cooperative Sticky, Static Membership and the KIP-848 Protocol

A misunderstood rebalance is the number one cause of unexplained lag

Almost every team running Kafka ends up having the same conversation. The consumer has been fine for weeks, nobody touched the code, and suddenly lag climbs in steps during every deploy. Or worse: it stays high permanently and not a single log line shows an exception. People check the brokers, the network, the message sizes. The culprit is almost always the same: the consumer group is rebalancing far more often than it should, and every rebalance under the classic protocol stops the entire group from consuming.

Rebalancing is the mechanism Kafka uses to distribute a topic's partitions among the live members of a group. It is what lets you scale from three pods to ten without coordinating anything by hand. It is also what turns a routine deploy into a thirty-second pause if you never configured it. This guide covers how it works under the hood, which clocks trigger it, what changes with cooperative rebalancing and static membership, and what the new KIP-848 protocol — generally available since Kafka 4.0 — actually does differently.

What actually happens during a rebalance

Every group is assigned a broker that acts as its group coordinator. The coordinator tracks the member list, the current group generation (a counter that increments on every rebalance) and the committed offsets. Under the classic protocol one of the consumers is additionally the group leader, and it computes the partition assignment inside its own process.

The classic sequence, known as an eager rebalance, has four steps:

1. Something changes: a member joins, leaves, stops heartbeating, or the topic's partition count changes.
2. The coordinator marks the group as rebalancing and starts answering heartbeats with REBALANCE_IN_PROGRESS.
3. Every member revokes all of its partitions and rejoins. This is the stop-the-world moment: nobody consumes anything until the last member has responded.
4. The leader computes the new assignment, the coordinator distributes it, and each member resumes consuming whatever it was given.

Step three is the painful one. The rebalance takes as long as the slowest member, and for that whole window the entire group is idle — including the nine consumers whose partitions were never going to move. A rolling deploy across ten replicas is therefore not one rebalance: it is ten consecutive rebalances, each with its own pause.

The three clocks that trigger a rebalance

Kafka evicts a consumer for two completely different reasons, and confusing them leads to tuning the wrong knob. Since version 0.10.1 the client runs two threads: a background thread that sends heartbeats, and your

application thread, which is the one calling poll().

- heartbeat.interval.ms (default 3000). How often the background thread tells the coordinator it is alive. It is cheap; leave it alone.
- session.timeout.ms (default 45000 since Kafka 3.0; it used to be 10000). How long the coordinator waits without a heartbeat before declaring the member dead. This catches crashed processes, deleted pods and network partitions. The rule of thumb is to keep it at three times the heartbeat interval or more.
- max.poll.interval.ms (default 300000, five minutes). How long your code may take between two poll() calls. This catches consumers that are alive but stuck: the process heartbeats perfectly while your loop has spent eight minutes waiting on a third-party API.

The distinction matters more than it looks. If your rebalances line up with latency spikes in a downstream service, the problem is max.poll.interval.ms and raising the session timeout will change precisely nothing. If they line up with memory pressure, long GC pauses or spot nodes disappearing, the problem is the session. Before you touch a number, find out which of the two clocks ran out.

Cooperative rebalancing: only revoke what actually moves

Incremental cooperative rebalancing (KIP-429) removes the global barrier. Instead of revoking everything and redistributing from scratch, the leader computes the target assignment and only the partitions that genuinely change hands are revoked. A consumer that keeps its eight partitions never stops consuming for a single millisecond.

The cost is that it takes two rounds: the first revokes the partitions that are moving, the second assigns them to their new owner. It sounds worse and it is dramatically better, because useful work never stops.

```
from confluent_kafka import Consumer

conf = {
    "bootstrap.servers": "kafka-1:9092,kafka-2:9092",
    "group.id": "orders-processor",

    # Incremental rebalancing: only what changes hands is revoked.
    "partition.assignment.strategy": "cooperative-sticky",

    # Detects dead processes.
    "session.timeout.ms": 45000,
    "heartbeat.interval.ms": 3000,

    # Detects live-but-stuck processes.
    "max.poll.interval.ms": 300000,

    # Never commit offsets you have not processed.
    "enable.auto.commit": False,
    "auto.offset.reset": "earliest",
}
```

If you are coming from the Java client there is an important migration trap: you cannot mix eager and cooperative assignors in the same group. The members would never agree and the group would hang. The switch takes two deploys:

```
# Deploy 1: the whole fleet advertises BOTH strategies.
# The group keeps using Range because it is the first one everyone shares.
partition.assignment.strategy=org.apache.kafka.clients.consumer.RangeAssignor,org.apache.kafka.clien
ts.consumer.CooperativeStickyAssignor

# Deploy 2 (only once 100% of the fleet has the above):
# drop the eager strategy and the group migrates to cooperative.
partition.assignment.strategy=org.apache.kafka.clients.consumer.CooperativeStickyAssignor
```

Skipping the first deploy causes long incidents, because the symptom — a group that rebalances in a loop and never stabilises — does not obviously point at the cause.

Static membership: restarts that do not count as departures

Static membership (KIP-345) solves a different problem. When a consumer restarts, the client generates a fresh member ID, so the coordinator sees a departure followed by an arrival: two rebalances per restart. Multiply that by your replica count and you know exactly why your deploys produce a lag plateau.

By setting group.instance.id, a member keeps its identity across restarts. A static client does not send LeaveGroup on shutdown, and the coordinator holds its partitions until session.timeout.ms expires. If the process comes back before that deadline, it gets exactly the same partitions and no rebalance happens at all.

```
import os

conf["group.instance.id"] = os.environ["POD_NAME"]  # e.g. orders-processor-2
```

The identifier has to be stable and unique. In Kubernetes that means a StatefulSet, whose pods are named orders-processor-0, orders-processor-1, and so on. Using the pod name from a Deployment buys you nothing: it changes on every rollout and you are back where you started. And if two processes start with the same value, the second one fences the first with a FencedInstanceIdException.

The trade-off is real and you should accept it deliberately: if the pod truly dies, its partitions sit unconsumed for the full 45 seconds of the session timeout. You are trading failure-recovery latency for deploy-time stability, which is a good deal when you ship several times a day and pods crash once a month. If your SLA cannot tolerate that gap, lower the session timeout or skip static membership.

KIP-848: rebalancing stops being the client's job

The new consumer group protocol, generally available since Apache Kafka 4.0, is the deepest change in this area in a decade. The core idea: the group leader and the global synchronisation barrier are gone. Assignment computation moves to the group coordinator, on the broker.

Instead of a JoinGroup + SyncGroup barrier where everyone waits for everyone, each member holds an independent conversation with the coordinator through a single RPC, ConsumerGroupHeartbeat. The coordinator publishes a target assignment stamped with an epoch; each consumer reconciles towards that target at its own pace, revoking first and acknowledging afterwards. There is no moment at which the whole group is stopped.

```
# Kafka 4.x client
group.protocol=consumer
group.id=orders-processor
group.instance.id=orders-processor-0
max.poll.interval.ms=300000
enable.auto.commit=false

# Under the new protocol these CLIENT settings are ignored:
#   session.timeout.ms
#   heartbeat.interval.ms
#   partition.assignment.strategy
# The broker owns them now (see below).
```

```
# Broker
group.coordinator.rebalance.protocols=classic,consumer
group.consumer.session.timeout.ms=45000
group.consumer.heartbeat.interval.ms=5000
```

Those settings moving server-side surprises a lot of people mid-migration. It is deliberate: the cluster operator now guarantees consistent failure-detection times instead of hoping every team configured its client correctly. If you need to tune failure detection, that is now a conversation with whoever runs Kafka, not a change in your values.yaml.

Two more things worth knowing. First, groups can be mixed during migration — members running group.protocol=classic and consumer coexist in the same group, so a normal rolling deploy works. Second, the classic protocol is still the default across the 4.x line; the new one is opt-in and is expected to become the default in a later major release. If you are already on 4.x brokers, enable it on a non-critical group first and compare rebalance latency before and after.

A correct consumer, end to end

Configuration is half the job; the rebalance callbacks are the other half. This is the full pattern with confluent-kafka-python, the client most commonly used outside the JVM ecosystem.

```
import logging
import os
import signal

from confluent_kafka import Consumer

log = logging.getLogger("orders")
running = True


def _stop(*_):
    global running
    running = False


signal.signal(signal.SIGTERM, _stop)
signal.signal(signal.SIGINT, _stop)


def on_assign(consumer, partitions):
    # Under the cooperative protocol, assignment is INCREMENTAL.
    # Calling assign() here would wipe the partitions we already held.
    log.info("assigned: %s", [(p.topic, p.partition) for p in partitions])
```

```
    consumer.incremental_assign(partitions)


def on_revoke(consumer, partitions):
    # Clean revocation: we still own these partitions.
    # This is the only safe moment to commit offsets.
    log.info("revoked: %s", [(p.topic, p.partition) for p in partitions])
    try:
        consumer.commit(asynchronous=False)
    except Exception:
        log.exception("commit failed during revocation")
    consumer.incremental_unassign(partitions)


def on_lost(consumer, partitions):
    # We lost the partitions without warning: session expired or we were fenced.
    # Another consumer may ALREADY own them. Committing here corrupts offsets.
    log.warning("partitions lost: %s", [(p.topic, p.partition) for p in partitions])
    consumer.incremental_unassign(partitions)


consumer = Consumer(conf)
consumer.subscribe(
    ["orders"],
    on_assign=on_assign,
    on_revoke=on_revoke,
    on_lost=on_lost,
)

try:
    while running:
        # librdkafka has NO max.poll.records: batch size is controlled here,
        # with num_messages, and with fetch.max.bytes.
        msgs = consumer.consume(num_messages=100, timeout=1.0)
        if not msgs:
            continue

        for msg in msgs:
            if msg.error():
                log.error("fetch error: %s", msg.error())
                continue
            handle(msg)  # must be idempotent

        consumer.commit(asynchronous=False)
finally:
    # Revokes, commits and shuts down cleanly.
    # Note: a static member does NOT leave the group here. That is the point.
    consumer.close()
```

Three specific decisions in that code deserve an explanation. incremental_assign and incremental_unassign instead of assign: under the cooperative protocol the callbacks receive only the delta, not the full set, and calling assign() would silently drop the partitions you were keeping. Committing inside on_revoke but never inside on_lost: in the first case you are still the legitimate owner, in the second another consumer may have been processing those partitions for seconds and your commit would rewind its offset. And num_messages instead of max.poll.records: that Java-client setting simply does not exist in librdkafka, and looking it up in the wrong documentation costs people entire afternoons.

When the work is slow: pause, do not stretch the timeout

The reflex when max.poll.interval.ms expires is to raise it to thirty minutes. It works, and it simultaneously destroys failure detection: a hung pod will hold its partitions for half an hour before anyone notices. The

correct fix is to decouple heartbeating from processing by pausing the partitions and continuing to call poll().

```
msgs = consumer.consume(num_messages=20, timeout=1.0)

if msgs:
    # Pausing stops new messages piling up while we work, but we keep
    # calling poll(), which is what resets the max.poll.interval.ms clock
    # and keeps the session alive.
    assignment = consumer.assignment()
    consumer.pause(assignment)
    try:
        for msg in msgs:
            process_slow(msg)   # may take minutes
            consumer.poll(0)    # returns nothing: partitions are paused
    finally:
        consumer.resume(assignment)

    consumer.commit(asynchronous=False)
```

If the work is genuinely long-running — transcoding a video, generating a multi-hour report — this is no longer the right pattern. Consume the message, write a task into a durable job queue, commit the offset, and let an independent worker do the heavy lifting. Kafka is excellent at moving events and fairly mediocre as a scheduler for long-running jobs.

Eight mistakes that keep showing up in production

1. Confusing the two clocks. Raising session.timeout.ms when the real problem is processing time. It changes nothing and degrades failure detection.

2. Leaving enable.auto.commit=true with asynchronous processing. Auto-commit commits what was delivered, not what was processed. A crash between those two moments is silent data loss — the kind nobody notices until a customer complains.

3. Committing offsets in on_lost. You no longer own those partitions. You rewind another consumer's offset and trigger mass reprocessing.

4. Migrating to cooperative in a single deploy. The group enters a rebalance loop it will not leave until the configuration is uniform again. Always two steps.

5. An unstable group.instance.id. A random UUID or the pod name from a Deployment has exactly the same effect as configuring nothing at all.

6. More consumers than partitions. The surplus ones sit idle with no assignment, burning budget. Scaling the group beyond the partition count does not increase throughput; you have to add partitions first (and that breaks per-key ordering if you do not plan for it).

7. Assuming rebalancing gives you exactly-once delivery. It does not. There is always a reprocessing window between the last commit and the revocation. Your handler must be idempotent. Full stop.

8. session.timeout.ms outside the broker's range. If it falls outside group.min.session.timeout.ms or group.max.session.timeout.ms, the consumer cannot even join the group, and the error is not descriptive.

What to measure

A healthy group rebalances when you deploy and almost never otherwise. These are the signals that confirm it, exposed by the client over JMX or through whichever exporter you use:

- rebalance-rate-per-hour — the headline metric. If it exceeds your deploy frequency, something needs investigating.
- rebalance-latency-avg and rebalance-latency-max — what each one costs. This is the metric that should collapse when you move to cooperative or KIP-848.
- last-rebalance-seconds-ago — excellent for alerting: if it keeps resetting to zero, you are in a loop.
- time-between-poll-avg and time-between-poll-max — compare against your max.poll.interval.ms. If the max is creeping toward the limit, eviction is only a matter of time.
- records-lag-max — the symptom the business sees. Always track it per partition.

For live diagnosis, the command-line tool is still the most direct route:

```
# Group state and protocol in use (Stable, PreparingRebalance...)
kafka-consumer-groups.sh --bootstrap-server kafka:9092 \
  --describe --group orders-processor --state

# Members, host, client-id and each one's partitions.
# If the same host shows up twice, you have a duplicate group.instance.id.
kafka-consumer-groups.sh --bootstrap-server kafka:9092 \
  --describe --group orders-processor --members --verbose

# Per-partition lag: where an unbalanced assignment becomes visible.
kafka-consumer-groups.sh --bootstrap-server kafka:9092 \
  --describe --group orders-processor
```

A group stuck in PreparingRebalance for minutes almost always means one of two things: a member that is not responding and is being waited on until its timeout expires, or an incompatible mix of assignment strategies.

Production checklist

- enable.auto.commit=false, with an explicit commit after processing.
- cooperative-sticky as the assignment strategy, migrated across two deploys.
- A stable group.instance.id, backed by a StatefulSet if you are on Kubernetes.
- Separate on_revoke and on_lost callbacks, committing only in the former.
- A terminationGracePeriodSeconds long enough for consumer.close() to finish.
- An idempotent handler, keyed on something derived from the event rather than the offset.
- Alerts on rebalance-rate-per-hour and on time-between-poll-max.
- If you run 4.x brokers, a plan to trial group.protocol=consumer on a non-critical group.

The end goal is boring, and it fits in one sentence: a deploy should not show up in your lag graphs. Once you get there, scaling consumption stops being scary — and that is exactly the property you picked Kafka for.

Inside the protocol: JoinGroup, SyncGroup and the generation id

So far we have treated a rebalance as a black box that happens and then it is over. It is worth opening the box, because almost every strange symptom you will meet in production becomes obvious once you know which four requests the consumer and the group coordinator exchange. You do not need to memorise the

wire format, but you do need to know who decides what, and in which order.

The group coordinator is a specific broker, not a separate service. It is chosen by hashing the group.id down to a partition of the internal __consumer_offsets topic; the leader of that partition is your coordinator. Two practical consequences follow. If that broker restarts, your group is briefly coordinator-less and has to rediscover one (you will see FindCoordinator in the logs). And if you have many groups whose names hash to the same partition, they share a fate. This is one reason offsets.topic.num.partitions is left high (50 by default) and is not something to change casually after the cluster exists.

The classic dance has four steps:

1. FindCoordinator — the consumer asks any broker who coordinates its group. The answer is cached, and re-requested if the coordinator dies.

2. JoinGroup — the consumer announces which topics it subscribes to and which assignment strategies it supports. This request blocks on the broker until every known member has sent its own, or until rebalance.timeout.ms expires. This is where most of the "stop the world" lives.

3. SyncGroup — the coordinator picks one member as group leader (usually the first to arrive), hands it the full member list with their subscriptions, and waits for it to return the assignment. Everyone else sends an empty SyncGroup and blocks waiting for their share.

4. Heartbeat — from here on, periodic beats that keep membership alive and that double as the channel through which the coordinator says "we need to rebalance again".

The part that surprises most people the first time: the partition assignment is computed by a client, not by the broker. The coordinator only acts as referee and mailbox. That explains why an old version of your application, deployed by accident, can split partitions differently from the rest of the fleet, and why a client with a buggy assignor can leave partitions orphaned without the broker noticing anything at all.

The glue that keeps this coherent is the generation id, an integer that increments on every completed rebalance. Every offset commit carries its generation id, and the coordinator rejects any commit that arrives with a stale one, returning ILLEGAL_GENERATION. It is a fencing mechanism: if a consumer stalled for twenty seconds and comes back believing it still owns partition 7, its commit is refused because the world has already moved on. Without this, a zombie consumer could rewind the offset of a partition it no longer owns and trigger a mass reprocessing.

The errors you will see around this cycle, and what they actually mean:

- REBALANCE_IN_PROGRESS — not an error, a signal. The coordinator is telling you "drop what you are doing and send JoinGroup again". If it appears constantly, your group cannot stabilise.
- ILLEGAL_GENERATION — you were late. Your commit belongs to a world that no longer exists. Almost always it means processing took longer than max.poll.interval.ms.
- UNKNOWN_MEMBER_ID — the coordinator had already evicted and forgotten you. You have to rejoin from scratch, with no prior identity.
- COORDINATOR_NOT_AVAILABLE / NOT_COORDINATOR — the coordinating broker changed. The client retries on its own; if you see this in bursts, look at broker health, not at your consumer.

One mental rule that saves hours of debugging: when the problem is ILLEGAL_GENERATION, look at your processing loop; when it is NOT_COORDINATOR, look at the cluster. Confusing the two leads whole teams to tune timeouts when what they had was a broker restarting every night.

Which assignor to pick, and why it is almost never the default

The assignment strategy decides which partition goes to which consumer. Kafka ships several, and the difference between them matters far more than the documentation suggests.

RangeAssignor is the historical one and the default under the classic protocol. It sorts each topic's partitions and hands out contiguous blocks. Its great virtue is co-partitioning: if you consume several topics with the same partition count and the same keys, partition 3 of orders and partition 3 of payments land on the same consumer, which lets you join locally with no network hop. Its great flaw is that it distributes badly when partition count is not a multiple of consumer count: 7 partitions across 3 consumers gives you 3, 2, 2 — and with several topics that skew always piles onto the same consumers.

RoundRobinAssignor deals partitions out one at a time across all members. It balances much better, but it loses cross-topic co-location and, more importantly, it reshuffles nearly everything on each rebalance: one member joining or leaving displaces the entire layout.

StickyAssignor adds the idea that matters: it balances as well as round-robin but, among all equally balanced layouts, it picks the one that moves the fewest partitions relative to the previous one. It is still eager (it revokes everything before reassigning), so the pause remains, but at least the local state you rebuild afterwards is smaller.

CooperativeStickyAssignor is sticky plus the incremental protocol we already covered: it does not revoke what is not moving. If you have to choose blind, choose this one. It is the only one that combines balanced distribution, stability across rebalances, and the absence of a global pause.

There is a fifth option people forget: rack-aware assignment. When the consumer declares client.rack and brokers have rack-tagged replicas, fetches can be served from a follower replica in the same availability zone instead of crossing zones to reach the leader. In clouds that bill cross-AZ traffic, this is not a micro-optimisation — it is a line on the invoice. Rack-aware assignment for consumer groups returned in the 4.x line, so the assignment actively tries to pair consumers with partitions whose replicas live in their own zone.

```
# Consumer with local-replica reads and cooperative assignment
bootstrap.servers=kafka-1:9092,kafka-2:9092,kafka-3:9092
group.id=orders-processor
client.rack=eu-west-1b
partition.assignment.strategy=org.apache.kafka.clients.consumer.CooperativeStickyAssignor
enable.auto.commit=false
max.poll.records=200
max.poll.interval.ms=300000
```

An expensive gotcha: client.rack only does anything if the brokers are configured with broker.rack and you have replica.selector.class pointing at the rack-aware selector. If you configure only the client side, nothing bad happens — you simply do not get the saving, and you stay convinced that you did.

Offsets, commits, and the exact moment messages get duplicated

A rebalance on its own duplicates nothing and loses nothing. What duplicates or loses is the relationship between when you commit the offset and when the work is actually done. The rebalance merely exposes a problem that was already there.

With enable.auto.commit=true, the client commits in the background every auto.commit.interval.ms (5 seconds by default) the offset of the last record it handed you, not the last one you processed. If your poll() returns 500 records, you process 120, and the process dies, auto-commit may already have committed all 500. The remaining 380 are never read again. That is not an auto-commit bug: it is precisely what it promises, and it is why auto-commit is useless anywhere losing a record matters.

Manual commit inverts the risk. If you commit after processing, a failure between "processed" and "committed" makes the next owner of the partition re-read that batch. That is at least once, and it is the sane starting point for nearly everything. The right question is not "how do I avoid duplicates" but "how do I make a duplicate harmless": primary keys with UPSERT, idempotent writes, deduplication by event id inside a window. A consumer whose side effect is idempotent turns a rebalance into a non-event.

Here is the difference between the two commits the client offers, and when to use each:

- commitAsync() on the hot path, inside the loop. It does not block, it pipelines well, and if one fails the next one fixes it because offsets are monotonic.
- commitSync() in two places and only two: in the revocation callback and in the orderly shutdown. There you do want to block, because after that line you lose the right to write.

If you use transactions (producer with a transactional.id and consumer with isolation.level=read_committed), the offset commit happens inside the transaction via sendOffsetsToTransaction, not via commitSync. The offset and the result of processing it become visible together, or neither does. That gives you exactly-once within Kafka, with two caveats worth saying out loud: it does not extend to external systems unless that system joins the transaction or is idempotent, and a rebalance aborts the open transaction, meaning the in-flight batch is discarded and repeated. Correct, but not free.

```
from confluent_kafka import Consumer, KafkaException

consumer = Consumer({
    "bootstrap.servers": "kafka-1:9092",
    "group.id": "orders-processor",
    "enable.auto.commit": False,
    "partition.assignment.strategy": "cooperative-sticky",
    "auto.offset.reset": "latest",
    "max.poll.interval.ms": 300000,
})

def process(msg):
    # Idempotent: the same key twice leaves the same state
    db.execute(
        "INSERT INTO orders (id, payload) VALUES (%s, %s) "
        "ON CONFLICT (id) DO UPDATE SET payload = EXCLUDED.payload",
        (msg.key().decode(), msg.value()),
    )

consumer.subscribe(["orders"])
try:
    while True:
        msg = consumer.poll(1.0)
        if msg is None:
            continue
        if msg.error():
            raise KafkaException(msg.error())
        process(msg)
        # async on the hot path: cheap and good enough
        consumer.commit(message=msg, asynchronous=True)
finally:
    # sync on shutdown: the last word has to land
    consumer.commit(asynchronous=False)
    consumer.close()
```

Look at the finally. close() does something many people overlook: it sends an explicit LeaveGroup. Without it, the group has to wait for session.timeout.ms to expire before noticing you left. An ungraceful SIGKILL turns a 3-second deploy into a 45-second one. Make sure your Kubernetes terminationGracePeriodSeconds exceeds the time your loop needs to exit, and that you actually handle SIGTERM instead of ignoring it.

The ConsumerRebalanceListener that actually works

The rebalance listener is the only place your code learns that the ground is shifting. It has three callbacks and most people implement one and a half of them.

onPartitionsRevoked fires while you still own the partitions and they are about to be taken. It is your window to flush buffers, close files, flush the sink and commit offsets. Everything you do here counts against rebalance.timeout.ms, so it must be bounded: if your flush can take a minute, your rebalance timeout has to allow for it or the group will evict you mid-cleanup.

onPartitionsAssigned fires with the new partitions already yours. That is where you rebuild local state, warm caches, or seek() if you manage offsets outside Kafka. Under the cooperative protocol it receives only what was added, not the full set; writing code that assumes otherwise is the number one mistake when migrating.

onPartitionsLost is the one almost nobody implements and the one that matters most. It fires when you lost partitions without warning: you were evicted, someone else already has them. Here you must not commit offsets, because the commit will be rejected or, worse, overwrite the progress of whoever is already working. The right thing is to discard in-flight work and clean up. If you do not implement it, the client falls back to the default, which calls onPartitionsRevoked, and you end up trying to commit offsets for partitions that belong to someone else.

```
from confluent_kafka import Consumer

def on_assign(consumer, partitions):
    # Cooperative: 'partitions' are ONLY the new ones
    for tp in partitions:
        state.load(tp.topic, tp.partition)
    print(f"assigned +{len(partitions)}")

def on_revoke(consumer, partitions):
    # We still own them: flush and commit
    sink.flush()
    try:
        consumer.commit(asynchronous=False)
    except Exception as e:
        print(f"commit during revoke failed: {e}")
    for tp in partitions:
        state.drop(tp.topic, tp.partition)

def on_lost(consumer, partitions):
    # We no longer own them: clean up, commit nothing
    sink.abort()
    for tp in partitions:
        state.drop(tp.topic, tp.partition)
    print(f"lost {len(partitions)} partitions without warning")

consumer = Consumer({
    "bootstrap.servers": "kafka-1:9092",
    "group.id": "orders-processor",
    "enable.auto.commit": False,
    "partition.assignment.strategy": "cooperative-sticky",
})
consumer.subscribe(
```

```
    ["orders"],
    on_assign=on_assign,
    on_revoke=on_revoke,
    on_lost=on_lost,
)
```

A subtle point about ordering: under the cooperative protocol, revocation and assignment can happen in different rounds. First round, you revoke what is moving. Second round, you receive what is coming to you. In between, your consumer keeps processing whatever it kept. If your listener assumes a revocation is immediately followed by an assignment in the same cycle, you will have inconsistent state for a few seconds and no idea why.

KIP-848 in detail: the reconciliation loop

The new protocol, generally available since Apache Kafka 4.0, is not an optimisation of the old one: it is a different model. The core difference is that global synchronisation disappears. There is no barrier where everyone waits for everyone.

In the new model, the broker coordinator keeps two views per member: the target assignment (what it should have) and the current assignment (what it has confirmed it has). When something changes — a member joins, another leaves, the partition count changes — the coordinator recomputes the target for the whole group and publishes it. From there each member converges at its own pace through a single RPC, ConsumerGroupHeartbeat, which serves as both heartbeat and reconciliation channel. The member says "I have this", the coordinator answers "you should have that", the member revokes what is surplus, confirms, and receives what is missing.

The practical consequences are large:

- Assignment computation lives on the broker. It no longer depends on which client version happened to be leader. You choose it with group.remote.assignor and it is uniform for the whole group.
- A slow member no longer blocks the rest. Under the classic protocol, JoinGroup waited for the slowest. Here each one converges independently and unaffected members never even notice.
- Fencing becomes epoch-based. Each member has its own epoch, and the group has one too. A member with a stale epoch is rejected without needing a shared global generation.
- Timeouts move to the server. session.timeout.ms and heartbeat.interval.ms stop being client configuration and become cluster policy (group.consumer.session.timeout.ms and group.consumer.heartbeat.interval.ms). That eliminates an entire class of incidents where a new deployment brings different timeouts from the rest of the fleet.

Three consumer properties disappear when you switch on the new protocol: partition.assignment.strategy, session.timeout.ms and heartbeat.interval.ms. If you leave them in your configuration file the client will either fail to start or ignore them, depending on the version. Removing them is a mandatory part of the migration, not optional tidying.

```
# 4.x client on the new protocol
bootstrap.servers=kafka-1:9092,kafka-2:9092,kafka-3:9092
group.id=orders-processor
group.protocol=consumer
group.remote.assignor=uniform
enable.auto.commit=false
max.poll.interval.ms=300000
```

```
# DO NOT include: partition.assignment.strategy
# DO NOT include: session.timeout.ms
# DO NOT include: heartbeat.interval.ms
```

```
# Broker: both protocols enabled during the migration
group.coordinator.rebalance.protocols=classic,consumer
group.consumer.session.timeout.ms=45000
group.consumer.heartbeat.interval.ms=5000
group.consumer.max.session.timeout.ms=60000
```

What does not change is just as important: max.poll.interval.ms is still yours and still the clock that kills slow consumers. The new protocol separates "I am alive" from "I am making progress" much more cleanly, but it does not forgive a processing loop that takes longer than you said it would.

Migrating without drama: the full procedure

Migrating to the new protocol is designed to be online, but it has an order and a set of prerequisites you cannot skip. This is the script that works.

Before touching anything. Confirm every broker is on 4.0 or later and that group.coordinator.rebalance.protocols includes consumer. Upgrade client libraries to a compatible version in every service sharing the group.id, even the ones you plan to migrate later. An old client trying to join a group that has already negotiated the new protocol simply does not get in: it sits outside and consumes nothing, and that is the single most common migration failure.

Configuration cleanup. Remove the three properties that cease to exist. Do it as its own deployment, without changing group.protocol yet, so that if something breaks you know exactly what it was. This deployment should be a functional no-op.

The switch. Add group.protocol=consumer and do a rolling restart, one instance at a time. When the first member with the new protocol joins a classic group, the coordinator converts the whole group. During the transition, members of both kinds coexist and the coordinator translates between them. It is not a state you want to live in for weeks, but it is safe for the duration of a deployment.

Verification. After each batch, check the group's protocol type and that lag is coming down:

```
# Protocol type, state and members
kafka-consumer-groups.sh --bootstrap-server kafka-1:9092 \
  --describe --group orders-processor --state

# Actual layout per member: who has what, and how much lag they carry
kafka-consumer-groups.sh --bootstrap-server kafka-1:9092 \
  --describe --group orders-processor --members --verbose
```

The way back. The rollback is the same path in reverse: drop group.protocol=consumer, restore the classic properties, do another rolling restart. The group reverts to classic once the last new-protocol member leaves. The existence of this exit is what makes it reasonable to migrate during working hours instead of at three in the morning on a Sunday. Have the rollback config file ready before you start; do not write it under pressure.

One sequencing tip: migrate a small, boring group first — one whose lag nobody loses sleep over. Live with it for a week. Most of what goes wrong in these migrations does not go wrong during the deployment; it goes wrong three days later under a traffic spike.

Autoscaling: the source of rebalances nobody checks

There is a source of rebalances that appears in no Kafka tuning guide, because it does not live in Kafka: the autoscaler. An aggressive HPA on a noisy metric can add and remove replicas every minute, and each of those operations is a rebalance. The group spends more time redistributing partitions than processing.

Three rules that avoid the problem:

- Scale on lag, not CPU. A Kafka consumer's CPU rises and falls with traffic shape and with GC. Group lag is the metric that actually says whether you need more hands. KEDA's Kafka scaler reads lag straight from the group.
- Asymmetric stabilisation windows. Scale up fast, scale down slowly. Adding a consumer costs a rebalance and solves a problem; removing one costs another rebalance and solves nothing urgent. A scaleDown.stabilizationWindowSeconds of 300 or more is normal.
- Partition count is the ceiling. With 12 partitions, replica number 13 processes nothing: it sits with zero partitions assigned, burning memory and emitting heartbeats. Worse, its arrival and departure trigger real rebalances that do affect the other twelve. Pin maxReplicaCount to the partition count and skip that entire class of incidents.

```
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: orders-processor
spec:
  scaleTargetRef:
    name: orders-processor
  minReplicaCount: 2
  maxReplicaCount: 12          # = the topic's partition count
  cooldownPeriod: 300
  triggers:
    - type: kafka
      metadata:
        bootstrapServers: kafka-1:9092
        consumerGroup: orders-processor
        topic: orders
        lagThreshold: "2000"
        offsetResetPolicy: latest
  advanced:
    horizontalPodAutoscalerConfig:
      behavior:
        scaleUp:
          stabilizationWindowSeconds: 30
        scaleDown:
          stabilizationWindowSeconds: 600
          policies:
            - type: Pods
              value: 1
              periodSeconds: 120
```

Combine this with static membership and you gain another property: image-update restarts stop counting as topology changes, and only genuine size changes move partitions. The difference on the rebalance-rate-per-hour chart is usually an order of magnitude.

Kafka Streams: tasks, standbys and the KIP-1071 protocol

Kafka Streams uses consumers underneath, but its unit of distribution is not the partition: it is the task. A task groups one partition from each input topic of a sub-topology, along with the associated state stores. That makes a Streams rebalance considerably more expensive: it is not enough to start reading from a different offset, you have to rebuild local state by replaying the changelog from the beginning or from the last checkpoint.

Hence the configuration that most affects the availability of a Streams application:

- num.standby.replicas — keeps warm copies of state on other instances. It costs disk and bandwidth, but it turns a minutes-long failover into a seconds-long one. For any stateful application that matters, one is the reasonable minimum.
- acceptable.recovery.lag — how far behind a standby may be and still count as caught up, so a task can move to it directly. 10,000 records by default.
- max.warmup.replicas — how many tasks the system may be warming at once when it decides to move them. It bounds the impact of the background probing rebalances Streams performs.
- probing.rebalance.interval.ms — how often Streams checks whether it can finally move a task to its preferred home. Ten minutes by default.

The behaviour that surprises people: Streams rebalances on purpose, periodically, to migrate tasks toward instances that already have warm state. It is called a probing rebalance and seeing it in the logs is not a sign of trouble. What is a sign of trouble is those rebalances never converging, which usually means the standbys cannot catch up.

```
Properties props = new Properties();
props.put(StreamsConfig.APPLICATION_ID_CONFIG, "orders-enrichment");
props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "kafka-1:9092");

// Stable identity: restarts do not move tasks
props.put(StreamsConfig.consumerPrefix("group.instance.id"),
          System.getenv("POD_NAME"));

// Warm state on another instance
props.put(StreamsConfig.NUM_STANDBY_REPLICAS_CONFIG, 1);
props.put(StreamsConfig.ACCEPTABLE_RECOVERY_LAG_CONFIG, 10_000L);
props.put(StreamsConfig.MAX_WARMUP_REPLICAS_CONFIG, 2);

// Headroom to rebuild state without being evicted
props.put(StreamsConfig.consumerPrefix("max.poll.interval.ms"), 600_000);
```

The recent development is KIP-1071, the Streams-specific rebalance protocol, which brings KIP-848's idea into the world of tasks: the broker computes the assignment. It removes client-side coordination, stores topology centrally, and adds dedicated RPCs and admin tooling, so you can finally ask the cluster why a task moved. Its core feature set is considered production-ready as of Kafka 4.2. If your Streams application suffers long pauses or frequent churn, this is the structural improvement you have been waiting for; if it runs fine, there is no rush.

When exclusive assignment is the problem: share groups

Everything above rests on one premise: a partition belongs to one consumer at a time. That premise is what

gives you per-key ordering, and it is also what forces you to rebalance.

There are workloads where ordering does not matter at all. Distributing jobs across workers, processing independent tasks, draining a queue of requests. In those cases exclusive assignment is pure cost: parallelism is capped by partition count, one slow consumer holds back its whole partition, and every fleet change triggers a rebalance.

KIP-932, queues for Kafka, introduces share groups: several consumers reading the same partition concurrently, with per-record acknowledgement and a delivery counter. There is no exclusive partition assignment, so there is no rebalance in the classic sense. You can have 50 consumers on a 6-partition topic and all of them work. A record that fails can be retried or sent to an error queue without blocking the others. The feature is considered production-ready in Apache Kafka 4.2.

The decision rule is simple and worth writing into the design doc: if you need per-key ordering or per-partition local state, use consumer groups and learn to rebalance well. If you only need to distribute work, a share group removes the problem instead of optimising it. A great deal of tuning effort gets spent on groups that never needed ordering guarantees in the first place.

Reading a rebalance in the logs

When something goes wrong the answer is almost always in the client logs, but you have to know which lines matter. Here is a healthy classic-protocol rebalance, annotated:

```
[ConsumerCoordinator] Revoke previously assigned partitions orders-3, orders-7
   -> your onPartitionsRevoked is about to run. The clock is ticking.

[ConsumerCoordinator] (Re-)joining group
   -> JoinGroup sent. From here we wait for the slowest member.

[ConsumerCoordinator] Successfully joined group with generation Generation{generationId=42, ...}
   -> the barrier opened. Watch that number: if it climbs several times a minute, you have churn.

[ConsumerCoordinator] Finished assignment for group at generation 42
   -> this line appears ONLY on the leader. Useful for knowing who computed the split.

[ConsumerCoordinator] Notifying assignor about the new Assignment(partitions=[orders-3, orders-11])
[ConsumerCoordinator] Adding newly assigned partitions: orders-11
   -> assignment delivered. Under cooperative you will see "Adding" with no full "Revoke" before it.
```

And these are the lines that indicate a real problem, with their translation:

```
[ConsumerCoordinator] Member consumer-1-abc sending LeaveGroup request ...
  because the poll loop has exceeded max.poll.interval.ms
   -> Your processing is too slow. Do not raise the timeout reflexively:
      lower max.poll.records or pause the partitions.

[ConsumerCoordinator] Attempt to heartbeat failed since group is rebalancing
   -> Normal once. In a loop, the group is not stabilising.

[ConsumerCoordinator] Offset commit failed on partition orders-3 at offset 91820:
  The coordinator is not aware of this member.
   -> You were evicted and did not notice. This is where onPartitionsLost
      should have discarded the in-flight work.

[AbstractCoordinator] Group coordinator kafka-2:9092 is unavailable or invalid,
  will attempt rediscovery
   -> The problem is the cluster, not you. Go look at the brokers.
```

One operational trick worth its weight in gold: when you suspect churn, do not read one pod's logs — count generations. A simple count of distinct generationId values per minute across all pods tells you in ten seconds whether you have a stability problem or a throughput problem. They are different things and they are fixed differently.

Sizing timeouts with arithmetic instead of intuition

Most max.poll.interval.ms values in production were chosen by raising the number until the errors stopped. You can do better, and the maths fits on a napkin.

You need three numbers you already have in your metrics: the 99th percentile of per-record processing time, how many records a poll() returns, and how long your onPartitionsRevoked takes in the worst case.

The rule is:

```
max.poll.interval.ms  >=  (p99_per_record * max.poll.records) * safety_factor
rebalance.timeout.ms  >=  p99_flush_time * safety_factor
```

Use a safety factor of 2 or 3, because the p99 of one record is not the p99 of a whole batch: batches accumulate tail latency. A concrete example. Suppose you enrich each order with an HTTP call whose p99 is 180 ms, and you have max.poll.records=500:

```
0.180 s * 500 records = 90 s per batch in the worst case
90 s * 3 (safety factor) = 270 s
max.poll.interval.ms = 300000   # 300 s, with headroom
```

Now look at the other side: that consumer takes up to 90 seconds to call poll() again. For 90 seconds it cannot react to a revocation. If a deployment is in flight, the rebalance stretches out to that point. The fix is not a bigger timeout, it is a smaller batch:

```
max.poll.records = 100
0.180 s * 100 = 18 s per batch
18 s * 3 = 54 s  ->  max.poll.interval.ms = 60000
```

Same work per second, four times more responsive to a rebalance. This is the central trade-off and almost nobody makes it explicit: large batches optimise steady-state throughput; small batches optimise recovery latency. Since rebalances happen precisely during the bad moments, the small batch usually wins on the metric your users actually see.

If the HTTP call is the bottleneck, there is a third path better than either: parallelise within the batch. Process the 500 records with a pool of 20 threads and batch time falls to 4.5 seconds without changing the size. Just keep the commit on the poll() thread and commit only the completed prefix, or you will have traded a timeout problem for a gaps-in-offsets problem.

Testing rebalances before production tests them for you

Rebalance code is by definition the code that only runs when something changes. That is why it is almost never covered by tests, and why it fails the first time it genuinely runs. With Testcontainers it is straightforward

to force it in CI.

The minimum test you should have: start a broker, create a multi-partition topic, start two consumers, kill one, and assert two things — that every partition ends up assigned, and that no message was lost.

```
import pytest, threading, time
from testcontainers.kafka import KafkaContainer
from confluent_kafka import Consumer, Producer
from confluent_kafka.admin import AdminClient, NewTopic

@pytest.fixture(scope="module")
def broker():
    with KafkaContainer() as kafka:
        yield kafka.get_bootstrap_server()

def test_no_messages_lost_during_rebalance(broker):
    AdminClient({"bootstrap.servers": broker}).create_topics(
        [NewTopic("t", num_partitions=6, replication_factor=1)]
    )
    p = Producer({"bootstrap.servers": broker})
    for i in range(3000):
        p.produce("t", key=str(i), value=str(i))
    p.flush()

    seen, lock, stop = set(), threading.Lock(), threading.Event()

    def worker(name):
        c = Consumer({
            "bootstrap.servers": broker,
            "group.id": "test-group",
            "auto.offset.reset": "earliest",
            "enable.auto.commit": False,
            "partition.assignment.strategy": "cooperative-sticky",
        })
        c.subscribe(["t"])
        while not stop.is_set():
            m = c.poll(1.0)
            if m and not m.error():
                with lock:
                    seen.add(int(m.value()))
                c.commit(message=m, asynchronous=False)
        c.close()

    a = threading.Thread(target=worker, args=("a",)); a.start()
    b = threading.Thread(target=worker, args=("b",)); b.start()
    time.sleep(10)

    stop.set(); a.join(); b.join()
    assert len(seen) == 3000, f"missing {3000 - len(seen)} messages"
```

It is a slow test — ten seconds — which is why many people never write it. Compare that with the cost of discovering in production that your on_revoke throws an uncaught exception. Put it in an integration suite that runs on every merge to the main branch rather than on every commit, and you get the coverage where it counts without punishing the development loop.

A useful variant of the same test: instead of killing a consumer, add a third one halfway through and assert that no partition is ownerless for more than N seconds. That exercises the cooperative path, which is the one you will actually use on every deployment.

Runbook: the group has been rebalancing for twenty minutes

A group that cannot stabilise is the worst scenario, because lag grows while nobody processes. This is the order to check things in, from most to least likely.

Step 1 — Confirm the symptom. Look at the group state. Persistent PreparingRebalance or CompletingRebalance means real churn; Stable with rising lag means a throughput problem and this runbook is not yours.

```
watch -n2 'kafka-consumer-groups.sh --bootstrap-server kafka-1:9092 \
  --describe --group orders-processor --state'
```

Step 2 — Count members. If the member count oscillates, you have pods joining and leaving. Check deployment restarts, OOMKills and failing readiness probes. Nine times out of ten a perpetual rebalance is a pod in CrashLoopBackOff, not a Kafka problem.

```
kubectl get pods -l app=orders-processor \
  -o custom-columns=NAME:.metadata.name,RESTARTS:.status.containerStatuses[0].restartCount
```

Step 3 — Find the evicted one. Filter logs for max.poll.interval. If one specific pod always shows up, you have a slow consumer: a poison message, a partition with hot keys, or an external dependency degraded only for that subset of data.

Step 4 — Find the stranger. Check that no old version is coexisting. In a half-finished migration, or during a canary deployment, it is easy to end up with two versions announcing incompatible assignor sets. The classic symptom is a group that rebalances and rebalances and never reaches Stable.

Step 5 — Mitigate, then fix. If you are bleeding lag, the fastest mitigation is almost always the same: pin the replica count (disable autoscaling), and if a pod is restart-looping, take it out of the group by scaling to zero and coming back up with corrected configuration. A stable group with fewer consumers processes infinitely more than an unstable group with many.

Afterwards, the part that really matters: write down what changed. Perpetual rebalances almost never appear on their own. They appear after a deployment, after a memory-limit change, after adding partitions to a topic, or after a dependency of yours got slow. The question "what was deployed in the previous two hours?" resolves more Kafka incidents than any configuration tweak.

Transactions and exactly-once: the rebalance that breaks your producer

If your service does read-process-write and you want all three to happen together or not at all, you need a transactional producer. At that point a rebalance stops being a latency problem and becomes a correctness problem, because the mechanism that protects Kafka from zombies (fencing) and the mechanism that hands out partitions (rebalancing) are talking about the same thing from two different places.

Transactional fencing hangs off the transactional.id: when a producer registers with that id, the broker gives it an epoch and evicts any earlier producer using the same id with a lower epoch. The trouble is that partition ownership moves between members on every rebalance. Before Kafka 2.6, the only way to keep fencing correct was to encode the input partition inside the transactional.id, something like orders-in-7. That worked, but it forced you to keep one producer per assigned partition: hundreds of connections, hundreds of buffers, and a full reconfiguration every time the assignment changed.

KIP-447 fixed it by moving the check to commit time. The producer now sends the consumer group metadata

along with the offsets, and the broker compares the generation id the producer carries against the group's current generation. If your instance dropped out of the group while it was processing, your generation is stale, the whole transaction is rejected, and whoever owns the partition now reprocesses from the last committed offset. One producer per instance, not per partition.

```
// Java: atomic read-process-write with a single producer per instance
producer.initTransactions();

while (running) {
    ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(500));
    if (records.isEmpty()) continue;

    producer.beginTransaction();
    try {
        Map<TopicPartition, OffsetAndMetadata> offsets = new HashMap<>();
        for (ConsumerRecord<String, String> r : records) {
            producer.send(new ProducerRecord<>("output", r.key(), transform(r.value())));
            offsets.put(new TopicPartition(r.topic(), r.partition()),
                        new OffsetAndMetadata(r.offset() + 1));
        }

        // The heart of KIP-447: groupMetadata() carries memberId and generationId.
        // The broker fences the transaction if this instance is no longer in that generation.
        producer.sendOffsetsToTransaction(offsets, consumer.groupMetadata());
        producer.commitTransaction();

    } catch (ProducerFencedException | CommitFailedException e) {
        // We were kicked out of the group. The broker has already aborted the transaction.
        // Close and rebuild the producer; the new owner of the partition will redo the work.
        producer.close();
        throw e;
    } catch (KafkaException e) {
        producer.abortTransaction();
    }
}
```

Three details that cost you dearly if ignored. First: consumers reading the output topic must run with isolation.level=read_committed, or they will see records from aborted transactions and the whole effort was decorative. Second: enable.auto.commit must be false, because the transaction owns the commit, not the consumer.

The third is the one people discover at three in the morning. An open transaction pins the last stable offset of the output topic: read_committed consumers do not advance past the first record of an in-flight transaction. If your instance gives up partitions with a transaction still open and simply walks away, that transaction stays alive until transaction.timeout.ms expires (60 seconds by default), and for that entire minute your downstream consumers look stuck even though the new producer is writing happily. That is why the revocation listener has to abort:

```
consumer.subscribe(List.of("input"), new ConsumerRebalanceListener() {
    @Override
    public void onPartitionsRevoked(Collection<TopicPartition> revoked) {
        // No commit here: offsets travel inside the transaction.
        // What we must do is not leave an open transaction pinning the LSO.
        if (transactionInFlight) {
            producer.abortTransaction();
            transactionInFlight = false;
        }
    }

    @Override
    public void onPartitionsLost(Collection<TopicPartition> lost) {
```

```
        // We own nothing anymore; abort and never attempt a commit.
        if (transactionInFlight) {
            producer.abortTransaction();
            transactionInFlight = false;
        }
    }

    @Override
    public void onPartitionsAssigned(Collection<TopicPartition> assigned) { }
});
```

On timeouts: the producer's transaction.timeout.ms cannot exceed the broker's transaction.max.timeout.ms (15 minutes by default), and it should sit comfortably above the real duration of a poll-process-commit cycle. Leave it at 60 seconds while your batch takes 90 and every transaction dies on its own, sending the service into an abort loop that looks like a network problem. The opposite hurts too: a very high transaction.timeout.ms delays recovery when an instance really dies, because the LSO is not released until it expires.

One nuance that surprises people: with transactions a rebalance does not duplicate messages, but it does throw work away. Everything processed since the last transactional commit is discarded and redone. If your transactions cover large batches to amortise commit cost, a group that rebalances often can be redoing 30% of its work without any lag metric giving it away. Track the abort rate, not just lag.

If you use Kafka Streams, processing.guarantee=exactly_once_v2 implements exactly this pattern underneath, revocation handling included. The real advantage of v2 over the original exactly_once setting is precisely KIP-447: one producer per stream thread instead of one per task, which is what made EOS viable on large topologies.

Outside the JVM: cooperative rebalancing in librdkafka

Almost everything written about rebalances assumes the Java client. But if your service is in Python, Go, C# or Rust, you are most likely running librdkafka underneath, and there the contract differs in one place that breaks things silently.

The rule is this: the moment you register a rebalance callback, librdkafka stops assigning and revoking partitions on its own. It becomes your responsibility, and the method you must call depends on the configured assignor. With eager assignors (range, roundrobin) you use assign() and unassign() with the full list. With cooperative-sticky you use incremental_assign() and incremental_unassign(), which add to or subtract from the current assignment. Mixing them is the classic mistake: calling assign() inside a cooperative callback replaces the whole assignment instead of adding to it, and the consumer starts losing partitions nobody revoked. The symptom is not a clear exception, it is lag on partitions that should be covered.

```
from confluent_kafka import Consumer, TopicPartition, KafkaException

def on_assign(consumer, partitions):
    # With cooperative-sticky, "partitions" are ONLY the new ones.
    # incremental_assign adds them to what we already had.
    for tp in partitions:
        state.load(tp.topic, tp.partition)   # warm caches, open files, etc.
    consumer.incremental_assign(partitions)

def on_revoke(consumer, partitions):
    # "partitions" are ONLY the ones leaving. The rest keep flowing.
    # Commit just these: a blanket commit here stomps on offsets we still own.
```

```
    offsets = [TopicPartition(tp.topic, tp.partition, pending[(tp.topic, tp.partition)])
               for tp in partitions if (tp.topic, tp.partition) in pending]
    if offsets:
        try:
            consumer.commit(offsets=offsets, asynchronous=False)
        except KafkaException as e:
            log.warning("commit during revoke failed, will reprocess: %s", e)
    for tp in partitions:
        state.unload(tp.topic, tp.partition)
    consumer.incremental_unassign(partitions)

def on_lost(consumer, partitions):
    # We lost the session (timeout, network partition). We own nothing:
    # do NOT commit, the broker would reject it or we would stomp a peer's offsets.
    for tp in partitions:
        state.discard(tp.topic, tp.partition)
    consumer.incremental_unassign(partitions)

consumer = Consumer({
    "bootstrap.servers": "kafka:9092",
    "group.id": "orders",
    "partition.assignment.strategy": "cooperative-sticky",
    "enable.auto.commit": False,
    "session.timeout.ms": 45000,
    "max.poll.interval.ms": 300000,
})
consumer.subscribe(["orders"], on_assign=on_assign, on_revoke=on_revoke, on_lost=on_lost)
```

That third callback, on_lost, is the equivalent of Java's onPartitionsLost, and in practice it is what separates a correct consumer from one that corrupts offsets. on_revoke fires when we hand over partitions in an orderly way and are still a valid member of the group, so committing makes sense. on_lost fires when we have already been evicted: the generation id we hold is stale, and any commit is either rejected or, worse, rewinds the offset of a peer that already moved ahead. If your library does not expose on_lost separately and you only have on_revoke, check the version before assuming you are covered.

Another detail that bites: in librdkafka, max.poll.interval.ms is measured between calls to poll() or consume(), exactly as in Java, but many high-level wrappers hide the poll inside an iterator. A for record in consumer loop doing long work per message is burning that budget with nothing in the code to show for it. If you process in batches, batch for real with consume(num_messages=N, timeout=...) and control the pace yourself.

To migrate from eager to cooperative outside the JVM, the two-deployment rule from the Java section applies unchanged: first move every member to the list cooperative-sticky,range so they keep electing range while old members are still around, then in the second deployment leave only cooperative-sticky. What differs is that you must rewrite the callback before the first deployment, not after, because code using assign() stops being correct the moment the group elects the cooperative assignor.

Kafka Connect rebalances too, and almost nobody watches it

A Connect cluster is just another consumer group. It does not hand out partitions, it hands out connectors and tasks, but it uses the same group coordinator, the same timeouts and the same pathologies. And because most people install Connect and forget about it, Connect rebalances tend to be the platform's blind spot.

Curiously, Connect got cooperative rebalancing before the consumer did: KIP-415 landed in Kafka 2.3, while KIP-429 for the consumer arrived in 2.4. The motivation was the same, the stop-the-world effect. Under the

eager protocol, adding a worker or posting a new connector config stopped every task in the cluster, recomputed the assignment and started everything again. In a cluster with fifty connectors that is several seconds of nothing moving each time somebody touches an admin endpoint.

The relevant configuration lives in the worker properties file:

```
# Rebalance protocol for the Connect cluster.
#   eager      -> classic stop-the-world (compatibility only)
#   compatible -> incremental cooperative (KIP-415, Kafka 2.3)
#   sessioned  -> cooperative + signed session tokens for internal worker-to-worker
#                 traffic. This is the default since Kafka 2.4.
connect.protocol=sessioned

# When a worker disappears, its tasks are marked "lost". Instead of reassigning
# them immediately, Connect waits this long in case the worker comes back: a
# rolling restart should not move work around.
# Default is 5 minutes.
scheduled.rebalance.max.delay.ms=300000

# Maximum time the coordinator waits for every worker to rejoin.
# It must cover the slowest worker startup with all its plugins loaded.
rebalance.timeout.ms=60000
session.timeout.ms=45000
heartbeat.interval.ms=3000
```

scheduled.rebalance.max.delay.ms is the interesting knob and the one most often turned the wrong way. Its purpose is to absorb restarts: if you take a worker down to upgrade it and it returns in two minutes, its tasks stay idle but nothing has moved, and it picks them back up unchanged. Without that wait, every worker restart triggers two redistributions (one on the way out, one on the way back) with the cost of reopening database connections, rereading source connector state and, on sink connectors, rebuilding buffers.

The temptation, when a worker really dies and you are staring at five minutes of idle tasks, is to drop it to zero. Do not do that without reading KAFKA-15693: disabling the delay can leave connectors and tasks unassigned indefinitely under certain rebalance sequences. If you need to react faster to real failures, lower it to something like 60000 rather than turning it off, and make sure your rolling deployments restart a worker in less than that.

Connect rebalance storms also have a cause of their own that a plain consumer does not have: a connector that fails to start and retries in a loop. Each failed attempt can trigger a fresh assignment round, and the cluster spends the day handing out work that never runs. The signal is in the worker log, not in consumer metrics:

```
# Real task state: look for FAILED in a loop or persistent UNASSIGNED
curl -s localhost:8083/connectors?expand=status | jq -r '
  to_entries[] | .key as $c |
  .value.status.tasks[]? | "\($c)\ttask=\(.id)\t\(.state)"' | sort | uniq -c

# Cluster identity and version
curl -s localhost:8083/ | jq .

# In the worker log, the lines that matter:
#   "Rebalance started"
#   "Successfully joined group ... with generation N"   <- N climbing fast = storm
#   "Wasn't able to resume work after last rebalance"
```

Rule of thumb: if the Connect group's generation id climbs more than once or twice an hour with nobody deploying anything, you have a sick connector or a badly sized timeout, not a busy cluster. And it is worth watching with the same alert you use for application consumer groups, because a Connect cluster that

rebalances endlessly produces no visible lag on its own topics until you have already been bleeding throughput for hours.

Rolling deployments on Kubernetes that never wake the coordinator

Static membership only helps if the identifier is genuinely stable, and on Kubernetes that means picking the right object to run your consumers. A Deployment produces random pod names: every restart changes the name, and if you derive group.instance.id from the hostname you end up creating a brand new member on every deploy, which is the exact opposite of what you wanted. A StatefulSet gives you stable ordinals (consumer-0, consumer-1) that survive restarts and upgrades.

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: orders-consumer
spec:
  replicas: 6
  serviceName: orders-consumer
  # OrderedReady restarts one at a time: only one member out of the group at once.
  podManagementPolicy: OrderedReady
  updateStrategy:
    type: RollingUpdate
  template:
    spec:
      # Must cover: drain the in-flight batch + commit + clean shutdown.
      terminationGracePeriodSeconds: 45
      containers:
        - name: app
          env:
            # The pod ordinal is a free, stable group.instance.id.
            - name: POD_NAME
              valueFrom:
                fieldRef:
                  fieldPath: metadata.name
            - name: KAFKA_GROUP_INSTANCE_ID
              value: "$(POD_NAME)"
          lifecycle:
            preStop:
              exec:
                # Tell the app to stop fetching new records and close the
                # in-flight batch before SIGTERM lands.
                command: ["/bin/sh", "-c", "touch /tmp/drain && sleep 20"]
          readinessProbe:
            # Do not report ready until after the first poll with partitions
            # assigned. Otherwise the rollout moves to the next pod while this
            # one is still consuming nothing.
            httpGet: { path: /ready, port: 8080 }
            periodSeconds: 2
            failureThreshold: 30
```

The rollout arithmetic is the part everyone skips. With static membership, the coordinator does not rebalance if the member returns before session.timeout.ms expires. That clock starts at the old pod's last heartbeat and has to cover all of this: the termination grace period, scheduling the new pod, pulling the image if it is not cached, starting the process and the first poll. On a JVM running Spring Boot with thirty beans, that is easily 40 or 50 seconds. A session.timeout.ms of 45000 next to a terminationGracePeriodSeconds of 45 leaves no room at all: the rebalance happens anyway and you are left wondering why static membership is not working.

There are two honest ways to square it. The first is raising session.timeout.ms until it covers the worst-case startup with slack, accepting that a real crash now takes longer to detect (the broker caps it with

group.max.session.timeout.ms, 30 minutes by default, but anything above two or three minutes turns a crash into a long incident). The second, better if you can, is shrinking startup: images prepulled onto nodes, lazy initialisation of anything not needed to consume, and the first poll as early as possible in the application lifecycle rather than after warming every cache.

```
# Measure the real rollout gap, which is the only number that matters:
kubectl get pods -l app=orders-consumer -w --output-watch-events \
  -o custom-columns=T:.metadata.creationTimestamp,NAME:.metadata.name,PHASE:.status.phase

# And in parallel, whether the coordinator rebalanced or not:
watch -n2 'kafka-consumer-groups.sh --bootstrap-server kafka:9092 \
  --describe --group orders --state | tail -3'
# STABLE for the whole rollout = static membership is doing its job.
# PREPARING_REBALANCE on each pod = the gap exceeds session.timeout.ms.
```

A warning about podManagementPolicy: Parallel starts and stops every pod at once, which speeds up the deployment and destroys the benefit of static membership, because the entire group vanishes for a few seconds and the coordinator has no choice but to redo the assignment. If you need speed, OrderedReady with a fast startup beats Parallel with a slow one nearly every time, because an ordered rollout generates no extra work downstream.

Finally, do not confuse scaling with deploying. Scaling from 6 to 8 replicas does have to rebalance: partitions genuinely change owner. What static membership eliminates is the free rebalance of a restart, where the final assignment is identical to the initial one. If your HPA is adding and removing replicas every few minutes because of a noisy metric, no group configuration will save you: fix the metric or set a generous stabilisation window before touching any Kafka timeout.

The ceiling nobody warns you about: group metadata and large groups

There is a hard limit in the classic protocol that appears in no introductory guide and shows up all at once when a group grows: the group's state has to fit inside a single Kafka record.

The mechanism is this. In JoinGroup, every member sends its subscription plus an opaque userData blob. In SyncGroup, the leader sends the complete assignment for the group and the coordinator distributes it. And the coordinator persists all of that as a record in __consumer_offsets so it can rebuild the group if the coordinating broker dies. That record is bound by the topic's max.message.bytes, roughly 1 MB by default.

With the range or roundrobin assignor, userData is nearly empty and the problem stays away for a long time. With sticky and cooperative-sticky, userData carries the member's previous assignment, because that is exactly what the assignor needs to minimise movement. In other words, the state size grows roughly with members times assigned partitions. A group of 400 members consuming a 3,000-partition topic, with long topic names like billing-events-normalised-v3, walks into the limit without doing anything unusual.

```
# Symptom: the group never stabilises and the coordinator log says
#   org.apache.kafka.common.errors.RecordTooLargeException
#   ... appending metadata message for group orders generation 812
# The group spins in PREPARING_REBALANCE or CompletingRebalance.

# Quick diagnosis: group size and total partitions
kafka-consumer-groups.sh --bootstrap-server kafka:9092 --describe --group orders --members \
  | awk 'NR>1 {m++; p+=$3} END {print "members="m, "partitions="p, "product="m*p}'

# And the real size of the group metadata records:
kafka-log-dirs.sh --bootstrap-server kafka:9092 \
```

```
  --topic-list __consumer_offsets --describe | jq '.brokers[].logDirs[].partitions[]
  | select(.size > 0) | {partition, size}' | head
```

The classic escapes were all uncomfortable. Raising max.message.bytes on __consumer_offsets works, but it is a critical internal topic and you are increasing the cost of every state write for every group in the cluster. Splitting the group into several smaller ones works, but forces you to partition topics between them or add client-side filtering. Going back to range shrinks userData, but you lose stickiness, which was the reason you wanted cooperative-sticky in the first place. None of them is good.

This is where KIP-848 stops being a latency improvement and becomes a structural change. By moving assignment computation to the broker, the userData round trip disappears: members no longer publish their previous assignment for the leader to read, because the coordinator already has it. State is kept and persisted per member and incrementally, not as one monolithic record holding the entire group. The 1 MB limit stops being a function of members times partitions, and with it a whole class of incidents that only appeared at scale goes away.

The practical corollary for sizing today: if your group passes a hundred or so members, or if the product of members times assigned partitions approaches tens of thousands, migrating to the new protocol is not an optional optimisation, it is how you remove a ceiling. And if you cannot migrate yet, measure that product and put an alert on it before you meet it in production.

```
# Migrating to the new protocol, operational summary (Kafka 4.x)
# 1) Enable it on the broker (GA since Kafka 4.0):
#    group.coordinator.rebalance.protocols=classic,consumer
# 2) Migrate consumers one at a time, without stopping the group:
#    group.protocol=consumer
#    (partition.assignment.strategy, session.timeout.ms and heartbeat.interval.ms
#     no longer apply on the client: the server now sets them via
#     group.consumer.session.timeout.ms and group.consumer.heartbeat.interval.ms)
# 3) Check the resulting group type:
kafka-consumer-groups.sh --bootstrap-server kafka:9092 --list --group-type consumer
kafka-consumer-groups.sh --bootstrap-server kafka:9092 --describe --group orders --state
```

Point 2 deserves emphasis because it changes how a group is operated. Under the new protocol, session and heartbeat timeouts stop being a decision of whoever writes the consumer and become platform policy, set on the broker. That removes the class of incident where one service ships an absurd session.timeout.ms and destabilises a shared group, but it also means you can no longer tune it from your code when you need to: if your application starts slowly, talk to whoever operates the cluster before migrating, not after. max.poll.interval.ms, by contrast, remains a client setting, because it measures something only the client knows: how long your code takes to come back for more records.

The evolution does not stop there. KIP-1251 introduces assignment epochs for consumer groups, sharpening reconciliation when several overlapping assignments arrive in quick succession, a case handled conservatively today. It is the kind of detail that does not change your code but reduces how many cycles a large group needs to converge after a change.

What to take away from all this

If I had to compress everything above into five sentences you can repeat in a design review:

- A rebalance is not a failure; it is the mechanism by which Kafka keeps its promise that every partition has exactly one owner. The goal is not to avoid it, it is to make it cheap.

- The three clocks do different jobs. session.timeout.ms detects dead processes, heartbeat.interval.ms is its sampling rate, and max.poll.interval.ms detects live-but-stuck processes. Tuning the wrong one fixes nothing.
- Cooperative plus static membership removes most of the deployment pain, and both are configuration changes, not architectural ones.
- KIP-848's new protocol moves the decision to the broker and removes the global barrier. Migrating is a procedure with a rollback, not a leap of faith.
- Most pathological rebalances do not come from Kafka. They come from a nervous autoscaler, a pod that dies, or a processing loop that takes longer than it promised.

The day a deployment of your consumer stops showing up on the lag chart, you will know you have it solved. That is the bar, and it is reachable with the parts that already come in the box.